R. DE PRISCO

# DISPENSA DEL CORSO DI ALGORITMI AVANZATI


UNIVERSITÀ DI SALERNO - A.A. 2023-2024

*Aggiornamento: Autunno 2024*

# *Indice*

# *Introduzione*

Questa dispensa è un ausilio per lo studio del corso di Algoritmi Avanzati. Non sostituisce le fonti originali dalle quali è tratta buona parte del materiale presentato. Poiché tali fonti sono varie e in lingua inglese, e poiché il materiale da esse tratto è stato elaborato e adattato alle esigenze del corso, si è ritenuto opportuno fornire la presente dispensa come linea guida. Lo studente potrà così avere un documento di riferimento che presenta gli argomenti trattati nelle lezioni. Per uno studio approfondito, si consiglia di consultare anche le fonti citate nel testo.

Il corso di Algoritmi Avanzati vuole fornire un percorso di studio che prosegue gli argomenti trattati nel corso di Algoritmi. Il corso di Algoritmi ha fornito allo studente gli strumenti di base per progettare algoritmi efficienti. Il corso di Algoritmi Avanzati parte proprio dallo studio dell'efficienza degli algoritmi per proseguire in varie direzioni che fondamentalmente, ma non solo, ci permettono di affrontare problemi che non sappiamo risolvere in maniera efficiente. Pertanto il corso parte dallo studio dei problemi "difficili" per poi trattare in maniera introduttiva i seguenti argomenti: trattare l'intrattabile (studio di tecniche generali per affrontare problemi che non sappiamo risolvere in maniera efficiente), algoritmi approssimati, algoritmi randomizzati, algoritmi online e algoritmi distribuiti. Sebbene trattare problemi difficili, o semplicemente migliorare l'efficienza dell'algoritmo, è la motivazione di partenza, le tecniche algoritmiche presentate in questo corso sono di interesse anche indipendetente da questo particolare aspetto. Per esempio, gli algoritmi online permettono di gestire situazioni in cui l'input viene fornito in pezzi generati uno dopo l'altro in modo dinamico, mentre gli algoritmi distribuiti ci permettono di affrontare problemi che si presentano in un sistema distribuito, come, ad esempio, una rete di calcolatori.

Ognuno di questi argomenti è discusso in un capitolo. La trattazione di ciascun argomento, che potrebbe essere senza problemi oggetto di un intero corso, o anche più di un corso, è da intendersi introduttiva. L'obiettivo è quello di portare a conoscenza dello studente tutti questi argomenti e fornire per ciascuno di essi una introduzione più o meno approfondita a seconda dei casi.

**AVVERTENZA:** Questa dispensa contiene sicuramente errori e imprecisioni.

# 1
# *Problemi difficili*

## *1.1 Introduzione*

Molti problemi possono essere risolti con algoritmi efficienti, cioè con algoritmi che necessitano di tempo *polinomiale* per la risoluzione del problema. Vari approcci metodologici, come il *dividi e domina (dìvide et ìmpera)*, la tecnica *greedy* e la *programmazione dinamica*, aiutano nella progettazione di algoritmi efficienti.

Sfortunatamente (o fortunatamente per alcuni aspetti) non tutti i problemi possono essere risolti efficientemente. Esistono dei problemi per i quali non siamo in grado di fornire degli algoritmi che garantiscano il raggiungimento di una risposta in tempo polinomiale. Si noti che per essere efficiente, data una istanza di input, l'algoritmo deve garantire una risposta in tempo polinomiale. E questo deve essere vero per *tutte* le possibili istanze del problema.

Tuttavia, per molti problemi per i quali non siamo in grado di fornire algoritmi efficienti, non siamo nemmeno in grado di provare che tali algoritmi non esistono. Questo è uno dei più grandi problemi irrisolti nel campo dell'informatica ed è noto come la *questione* $\mathcal{P}$-$\mathcal{NP}$. In questo contesto, con $\mathcal{P}$ si indica l'insieme dei problemi per i quali si conoscono degli algoritmi efficienti, cioè algoritmi che trovano una soluzione in tempo polinomiale. Ad esempio, trovare il cammino minimo da una sorgente a una destinazione in un grafo è un problema in $\mathcal{P}$.

Nei successivi paragrafi definiremo formalmente la classe di problemi $\mathcal{NP}$. Per adesso, senza nessuna pretesa di formalità, anticipiamo che i problemi $\mathcal{NP}$ sono quei problemi per i quali è facile verificare che ciò che ci viene presentato come una soluzione del problema è effettivamente una soluzione[1]. Si consideri ad esempio il problema della fattorizzazione di un intero: sia dato un numero intero $n$ ottenuto come il prodotto di due interi $a$ e $b$, che però non si conoscono; fattorizzare $n$ significa trovare i due interi $a$ e $b$ usati per ottenere $n = \times b$. Questo problema, all'apparenza semplice, può essere molto difficile da risolvere. Se i due interi $a$ e $b$ sono numeri primi molto grandi, ad esempio numeri con circa 2000 cifre, allora è estremamente difficile trovarli. Tuttavia, anche se $a$ e $b$ sono così grandi, se qualcuno ce li dice, è facile verificare che $n = a \cdot b$, basta fare la moltiplicazione, che, anche per numeri molto grandi, si riesce a fare in maniera efficiente. Il problema della fattorizzazione è un problema in $\mathcal{NP}$.

È abbastanza intuitivo il fatto che se un problema appartiene a $\mathcal{P}$, allora appartiene

[1] L'acronimo $\mathcal{P}$ sta, come probabilmente si intuisce, per "polinomiale" (deterministico). Meno intuitivamente, l'acronimo $\mathcal{NP}$ sta per "non-deterministico polinomiale" e fa riferimento a una definizione alternativa, e equivalente, della classe $\mathcal{NP}$, come verrà spiegato nel paragrafo 1.4.

anche a $\mathcal{NP}$: trovare una soluzione è più "difficile" che verificarla! (Nel seguito daremo una prova di questa intuizione.) Quindi si ha che $\mathcal{P} \subseteq \mathcal{NP}$. Non sappiamo però se $\mathcal{P} = \mathcal{NP}$ oppure se $\mathcal{P} \subset \mathcal{NP}$. Quest'ultima ipotesi è quella più accreditata ma nessuno è stato capace di dimostrarla, quindi rimane la possibilità che $\mathcal{P} = \mathcal{NP}$.

La questione $\mathcal{P}$-$\mathcal{NP}$ è di fondamentale importanza e la sua risoluzione avrà grosse ripercussioni. Se da un lato possiamo "augurarci" che $\mathcal{P}$ sia uguale a $\mathcal{NP}$ per poter utilizzare algoritmi efficienti per tanti problemi, dall'altro siamo "contenti" che problemi difficili esistano. Infatti, alcuni di essi sono alla base dei moderni sistemi crittografici che ci permettono di usare tranquillamente la carta di credito per fare acquisti su Internet (e non solo). In particolare il problema della fattorizzazione, usato nel sistema di cifratura noto come RSA (dai nomi dei suoi ideatori, Rivest, Shamir e Adleman), è alla base del commercio elettronico, anche se i più moderni sistemi utilizzano approcci basati su curve ellittiche che risultano più efficienti rispetto a RSA.

Anche chi riuscirà a risolvere la questione $\mathcal{P}$-$\mathcal{NP}$ avrà ripercussioni: diventerà famoso e vincerà un milione di dollari messo in palio da Landon Clay. Che ha anche pensato ad altri problemi meritevoli di tale premio: una descrizione non tecnica dei problemi da un milione di dollari può essere trovata nel saggio di Marcus du Sautoy *L'equazione da un milione di dollari* [12].

## 1.2 *Problemi decisionali*

Prima di procedere nella discussione è necessario definire la nozione di problema decisionale e spiegare come questa si leghi al problema stesso. In questo paragrafo chiariremo cosa intendiamo per problema, per istanza di un problema e che cosa intendiamo per problema decisionale.

Un problema è una specifica domanda per la quale cerchiamo una risposta. È caratterizzato da una serie di parametri che nella definizione del problema vengono lasciati non specificati, sono, cioè, delle variabili. Oltre alla descrizione dei parametri, che di fatto costituiscono il problema, è necessario specificare che cosa è una soluzione al problema.

Un'istanza di un problema è data da un particolare assegnamento di valori alle variabili che descrivono il problema.

Un algoritmo che risolve un problema è un procedimento che partendo da un'istanza del problema (input) produce una soluzione (output) per quella particolare istanza.

Facciamo degli esempi. Consideriamo il problema del cammino minimo in un grafo $G = (V, E)$ con $n$ nodi[2] $V = \{v_1, v_2, \ldots, v_n\}$ ed $m$ archi $E = \{e_1, e_2, \ldots, e_m\}$. A ogni arco $e = (i, j)$ è associato un costo $c_{i,j}$. Un tale grafo potrebbe rappresentare un insieme di $n$ città (i nodi) fra di loro collegate da strade, collegamenti ferroviari, o altro, e ogni collegamento ha un costo. La domanda che ci poniamo è la seguente: date due specifiche città $s$ e $d$, quale è il cammino minimo da $s$ a $d$? In questo problema ci sono vari parametri: il numero $n$ di nodi, il numero $m$ di archi, gli archi, i costi degli archi, il nodo $s$ ed il nodo $d$. Una soluzione è un cammino da $s$ a $d$. Quando i parametri non sono specificati il problema è astratto. La Figura 1.1 riporta una rappresentazione del problema astratto: non conosciamo il numero di nodi, non conosciamo il numero di

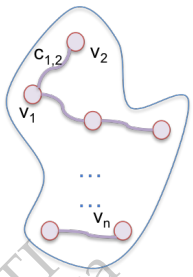



Figura 1.1: Un grafo astratto

[2] Nella descrizione di grafi utilizzeremo "nodi" e "vertici" come sinonimi. Per questo motivo spesso i nodi di un grafo sono indicati con la lettera $v$.

archi, non conosciamo quali nodi sono collegati da archi, non conosciamo i costi e non conosciamo i nodi $s$ e $d$.

Un'istanza del problema specifica i parametri. Ad esempio potremmo avere $n = 5$, $V = \{v_1, v_2, v_3, v_4, v_5\}$, $m = 7$ e $E = \{(1,2), (1,3), (1,5), (2,4), (3,4), (3,5), (4,5)\}$, con costi $c_{1,2} = 8, c_{1,3} = 3, c_{1,5} = 12, c_{2,4} = 15, c_{3,4} = 9, c_{3,5} = 5, c_{4,5} = 2$, e $s = v_1$, $d = v_4$. La Figura 1.2 mostra tale istanza del problema, la cui soluzione è il cammino $s, v_3, v_5, d$, con costo pari a 10. È facile verificare che in questo grafo non c'è un cammino da $s$ a $d$ con costo minore di 10.



Ricordiamo che esistono vari algoritmi efficienti per il problema del cammino minimo in un grafo. L'algoritmo di Dijkstra, che funziona per grafi con costi non negativi, ha complessità $O((|V| + |E|) \log |V|)$ se la coda a priorità usata dall'algoritmo viene implementata con un heap, mentre ha complessità $O(|V|^2)$ se il minimo deve essere cercato in un array non ordinato. L'algoritmo di Bellman-Ford funziona anche con costi negativi e ha complessità $O(|V| \cdot |E|)$. Entrambi gli algoritmi calcolano i cammini minimi da una sorgente data in input a tutti gli altri nodi del grafo. L'algoritmo di Floyd-Warshall, che ha complessità $O(|V|^3)$, calcola i cammini minimi fra tutte le coppie di nodi. Se il grafo non è orientato e i costi sono unitari, una semplice visita BFS che parte dalla sorgente calcola i cammini minimi in tempo $O(|V| + |E|)$.

Consideriamo un altro esempio: il problema del commesso viaggiatore[3], TSP (Travel Salesman Problem). Anche in questo caso abbiamo un grafo con degli archi a cui sono associati dei costi. Questa volta però non ci interessa il cammino minimo fra due città, bensì ci interessa trovare un *giro* che visiti una sola volta tutte le città ritornando nella città di partenza[4] e che abbia costo minimo. Il grafo astratto è uguale a quello del caso precedente: abbiamo un numero non specificato di nodi e un numero non specificato di archi. Una soluzione è un ciclo che contiene (una e una sola volta) tutti i nodi. Anche un'istanza è molto simile a quella del problema dei cammini minimi: specificheremo i nodi, gli archi ed i costi, ma non ci sono sorgente e destinazione. Ad esempio, lo stesso grafo riportato nella Figura 1.2 è un'istanza del problema del commesso viaggiatore, se ignoriamo $s$ e $d$. In questo caso la soluzione è data da $(v_1, v_3, v_5, v_4, v_2, v_1)$. Si noti che qualunque nodo può essere usato come partenza visto che la soluzione è un giro. In altre parole la soluzione $(v_1, v_3, v_5, v_4, v_2, v_1)$ è equivalente a $(v_3, v_5, v_4, v_2, v_1, v_3)$ e a tutti gli altri giri con la stessa sequenza di nodi.

[3] Il nome deriva dal fatto che un ipotetico commesso viaggiatore debba partire dalla propria città di residenza, visitare tutte le città una sola volta, e ritornare a casa.

[4] Quindi un giro è un ciclo che contiene tutti i nodi. Vedremo più avanti che per grafi non pesati un tale giro viene chiamato ciclo hamiltoniano.

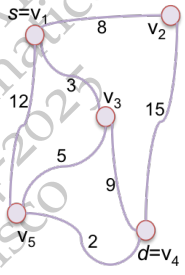



Figura 1.2: Un grafo concreto

Per il problema del commesso viaggiatore non si conoscono algoritmi efficienti; è uno dei problemi *difficili* che vedremo nel prosieguo del capitolo.

I problemi che abbiamo visto come esempi sono problemi di ottimizzazione: si cerca una soluzione che ottimizzi una metrica; negli esempi la metrica è un costo e l'obiettivo è quello di minimizzarlo. Nella maggior parte dei casi pratici abbiamo a che fare con problemi di ottimizzazione. Nei casi in cui non esiste una metrica che misuri la bontà delle soluzioni, siamo interessati a trovare una soluzione qualsiasi; in questi casi si parla di problemi di *ricerca* (di una soluzione).

Per affrontare lo studio dei problemi difficili è però conveniente considerare *problemi*

*decisionali*. Nella versione decisionale di un problema ci chiediamo se (almeno) una soluzione al problema esiste. Pertanto la risposta alla domanda posta dalla versione decisionale di un problema è un `si` oppure un `no`. Di primo acchito, può sembrare che questa restrizione sia molto forte. In realtà non lo è.

In primo luogo non è una limitazione perchè, visto che l'obiettivo è quello di individuare i problemi più difficili, se è problema è difficile nella sua versione decisionale a maggior ragione lo sarà nella versione di ricerca/ottimizzazione.

Inoltre se sappiamo risolvere la versione decisionale, possiamo facilmente trovare il valore della soluzione ottima al problema di ricerca. É sufficiente introdurre un parametro $k$ e cambiare la domanda in: esiste una soluzione il cui costo è minore uguale (se si cerca un minimo, o maggiore uguale se si cerca un massimo) di $k$? In questo modo il problema decisionale ci permette di sapere quale è il valore della soluzione ottima. Ad esempio, considerando il problema del commesso viaggiatore, potremmo chiederci se nel grafo $G$ esiste un giro il cui costo è minore o uguale a 100. Considerando l'istanza della Figura 1.2 la risposta sarà `si`. Quindi potremmo chiederci se esiste un giro con costo 50, e così via. Con una ricerca binaria riusciremo a scoprire, usando un numero logaritmico di esecuzioni dell'algoritmo per il problema decisionale, il valore 10 della soluzione ottima.

Poiché chiaramente la versione decisionale è più facile della versione di ricerca, lo studio fatto sulle versioni decisionali non crea problemi al riguardo: se è difficile la versione decisionale lo sarà anche quella di ricerca.

Inoltre, sebbene non sia stato provato che la versione decisionale è computazionalmente equivalente alla versione di ricerca, per quasi la totalità dei problemi per i quali si conosce un algoritmo efficiente per la versione decisionale del problema, si conosce anche un algoritmo efficiente per trovare una soluzione o la soluzione ottima nel caso di problemi di ottimizzazione. Approfondiremo questo aspetto nella Sezione 1.7.

## 1.3 *Grandezza del problema*

La difficoltà della risoluzione di un problema è ovviamente funzione della "taglia" dell'istanza da risolvere. Se la taglia è piccola anche un problema "difficile" può essere risolto facilmente. Abbiamo definito l'efficienza rispetto al tempo di esecuzione dell'algoritmo che per essere considerato efficiente deve operare in tempo polinomiale. Polinomiale rispetto a cosa? Rispetto alla taglia o grandezza del problema. Ma come misuriamo la grandezza di un'istanza di un problema? Una possibilità è quella di stabilire una (ragionevole) codifica del problema in termini di stringhe binarie e quindi misurare la lunghezza della stringa che serve per rappresentare le istanze del problema. Ad esempio per specificare un'istanza del problema del commesso viaggiatore dobbiamo codificare il numero di nodi del grafo, gli archi, i costi degli archi. Codificando ognuno di questi elementi con delle sequenze di bit e concatenandoli secondo uno schema che ci permetta di sapere cosa rappresentano i singoli pezzi per poterli poi interpretare correttamente, si ottiene una singola stringa binaria che rappresenta l'istanza del problema.

Ad esempio, consideriamo l'istanza del problema della ricerca del cammino minimo

nel grafo della Figura 1.2. Stabiliamo che nella rappresentazione dell'istanza specificheremo prima il numero di nodi, poi l'indice della sorgente seguito da quello della destinazione e di seguito delle triple che specificano archi e relativi costi, cioè una stringa del tipo

$$n, s, d, (v, v, c), (v, v, c), \ldots (v, v, c)\#$$

dove $n, s, d$ e i vari $v$ e $c$ sono numeri e # è un segnalatore di fine stringa. Si noti che una volta specificato il numero di nodi $n$, ogni singolo nodo può essere implicitamente identificato da un numero fra 1 e $n$. I simboli di cui abbiamo bisogno sono: le cifre per rappresentare i numeri, la virgola per separare gli elementi, le parentesi tonde per raggruppare gli archi con incluso il costo — in realtà potremmo anche fare a meno di queste parentesi, ma per facilità di lettura della codifica le utilizziamo — e un carattere di fine stringa per indicare che la rappresentazione è finita. Nel caso specifico quindi avremo la sequenza

$$5, 1, 4, (1, 2, 8), (1, 3, 3), (1, 5, 12), (2, 4, 15), (3, 4, 9), (3, 5, 5), (4, 5, 2)\#$$

In realtà tale sequenza la vogliamo rappresentare in binario, quindi dobbiamo anche stabilire una rappresentazione binaria per i simboli che utilizziamo[5]. Poichè ci servono 14 simboli in totale (10 cifre, le due parentesi tonde, la virgola e il carattere di fine stringa), possiamo utilizzare sequenze di 4 bit: i valori da 0000 a 1001 rappresentano le cifre, 1010 rappresenta "(", 1011 rappresenta ")", 1100 rappresenta "," e 1111 rappresenta il carattere di fine stringa "#". Le strighe 1101 e 1110 non vengono utilizzate, quindi la rappresentazione binaria dell'istanza della Figura 1.2 è:

0101 1100 0001 1100 0100 1100 1010 0001 1100 0010 1100 1000 1011 1100 1010
0001 1100 0101 1100 0001 0010 1011 1100 1010 0010 1100 0100 1100 0001 0101
1011 1100 1010 0011 1100 0100 1100 1001 1011 1100 1010 0011 1100 0101 1100
0101 1011 1100 1010 0100 1100 0101 1100 0010 1011 1111

La lunghezza di tale stringa è 224. Si noti come il costo degli archi $(1, 5)$ e $(2, 4)$, rispettivamente, 12 e 15, siano stati rappresentati con 2 cifre (quindi 8 bit). Questo perchè nella nostra codifica possiamo rappresentare solo le singole cifre; quindi per rappresentare valori più grandi, dobbiamo codificare i numeri rappresentando la sequenza delle singole cifre. Questo aspetto è importante, come spiegheremo fra un po'. Si noti che avremmo potuto anche usare la rappresentazione binaria del valore per specificare il costo degli archi, ma la sostanza del discorso non sarebbe cambiata: avremmo avuto bisogno di più bit per rappresentare valori più grandi.

Assumendo di usare delle codifiche ragionevoli[6], la lunghezza di tale stringa è proporzionale (quindi legata polinomialmente) ai parametri più significativi del problema. Ad esempio, nel caso del problema TSP, il parametro più significativo è il numero di nodi $n$: tale numero determina il massimo numero di archi e costi possibili ($n^2$) ed è legato polinomialmente alla lunghezza della stringa binaria che rappresenta il problema. Pertanto per semplificare la trattazione considereremo come misura della grandezza del problema il parametro più significativo.

Affinchè la lunghezza della stringa che rappresenta il problema sia polinomialmente legata al parametro più significativo è però necessario che i valori rappresentabili con un

[5] Codifica dei simboli:

| | |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| ( | 1010 |
| ) | 1011 |
| , | 1100 |
| | 1101 |
| | 1110 |
| # | 1111 |

[6] Ad esempio una codifica unaria è da escludere in quanto genererebbe delle stringhe irragionevolmente lunghe. Per approfondimenti si veda il Capitolo 34 di [8].

numero di simboli variabili, come nel caso dei costi degli archi nell'esempio precedente, siano anche essi legati polinomialmente al parametro più significativo. Quindi faremo implicitamente tale assunzione. Approfondiremo ulteriormente questo aspetto più tardi, quanndo parleremo di pseudopolinomialità.

Ovviamente la determinazione del parametro più significativo dipende dal problema. Per problemi che riguardano grafi è il numero di nodi. Nel seguito vedremo dei problemi che coinvolgono delle formule booleane; in tal caso il parametro più significativo, rispetto alla grandezza, è il numero di variabili.

## 1.4 Classi P e NP

Per la definizione delle classi $\mathcal{P}$ e $\mathcal{NP}$ si considerano problemi formulati in versione decisionale. La motivazione è legata al fatto che i problemi decisionali possono essere messi in corrispondenza con i linguaggi formali: tutte le stringhe che rappresentano istanze che ammettono una soluzione fanno parte del corrispondente linguaggio mentre tutte le altre stringhe no. Quindi un problema decisionale è un linguaggio formale. Trattare i problemi attraverso linguaggi formali ha dei vanataggi. Non approfondiremo questo aspetto ma, come esempio, nel paragrafo 1.7 c'è un ragionamento che mostra l'utilità dei linguaggi formali nel contesto della questione $\mathcal{P}$-$\mathcal{NP}$. Un ulteriore esempio è la possibilità di definire facilmente il "complemento" di un problema, semplicemente invertendo la risposta.

**Definizione 1.4.1** *La classe di problemi indicata con $\mathcal{P}$ è l'insieme dei problemi per i quali conosciamo un algoritmo efficiente, cioè polinomiale.*

Dunque, considerato che la difficoltà della versione decisionale è paragonabile a quella del problema stesso, per un problema in $\mathcal{P}$ *trovare* una soluzione costa tempo polinomiale. Per un problema come TSP, invece, non si conosce un algoritmo polinomiale e non si sa nemmeno che un tale algoritmo non esiste. Cioè non si sa se il problema TSP appartiene a $\mathcal{P}$. In altre parole non sappiamo se trovare una soluzione per TSP sia difficile o meno. Tuttavia se ci viene data una particolare soluzione, è facile *verificare* che il suo costo è minore della soglia stabilita dal problema. La classe $\mathcal{NP}$ è l'insieme dei problemi per i quali la verifica di una potenziale soluzione può essere fatta in tempo polinomiale.

Per rendere più formale la discussione occore specificare meglio cosa significa verificare una soluzione. Come abbiamo già detto l'input al problema viene codificato con una stringa binaria $s$. Denoteremo la lunghezza della stringa con $|s|$. Un problema decisionale $X$ può essere identificato con l'insieme delle stringhe binarie che corrispondono agli input per i quali la risposta al problema è "`si`". Un algoritmo $A$ per $X$ prende in input una stringa $s$ e risponde "`si`" oppure "`no`". Diremo anche che $A$ risolve $X$ se $A(s) =$`si` se e solo se $s \in X$. Il tempo di esecuzione di $A$ è polinomiale se esiste un polinomio $p()$ tale che per qualsiasi stringa di input $s$, $A(s)$ termina in al massimo $O(p(|s|))$ passi.

Un algoritmo permette di risolvere un problema. Se volessimo invece solo essere capaci di verificare una potenziale soluzione? Per formalizzare tale nozione,

avremo bisogno di un *certificato*. Un verificatore quindi prende in input due parametri, un'istanza del problema $s$ e un certificato $c$. L'output del verificatore può essere `ok`, se il certificato costituisce una prova del fatto che $s$ ammette soluzioni, oppure `nok` se invece il certificato non "convince".

**Definizione 1.4.2** *Un algoritmo $V$ è un "verificatore di esistenza" per un problema $X$ se per ogni stringa $s \in X$, cioè per ogni istanza del problema che ammette soluzioni, esiste una stringa $c$ (certificato), di lunghezza polinomiale in $|s|$, tale che $V(s, c) =$ `ok`, in tempo polinomiale in $|s|$.*

Si noti che il verificatore non ha l'obiettivo di stabilire se la stringa $s$ appartenga a $X$ o meno. Invece, ha come obiettivo quello di "farsi convincere" dal certificato $c$ che la stringa di input $s$ sia effettivamente in $X$. Quindi l'approccio del verificatore potrebbe essere spiegato, informalmente, come segue: non credo che l'istanza $s$ abbia una soluzione a meno che il certificato $c$ non costituisce una prova di questo fatto, oppure non rieca a trovare io stesso una soluzione. Si noti anche che un verificatore è efficiente per definizione (un verificatore non efficiente non è utile). Un certificato, tipicamente, è una soluzione e il verificatore deve solo "verificare" che la soluzione descritta dal certificato sia effettivamente valida. Quindi è abbastanza semplice fornire un verificatore (di esistenza).

**Definizione 1.4.3** *La classe $\mathcal{NP}$ è la classe dei problemi per i quali esiste un verificatore di esistenza di una soluzione.*

La "N" in $\mathcal{NP}$ deriva dal fatto che la verifica di una soluzione può anche essere vista come un approccio non deterministico alla risoluzione del problema. Cioè, una definizione alternativa della classe $\mathcal{NP}$ è quella che definisce i problemi $\mathcal{NP}$ come quei problemi che possono essere risolti con algoritmi polinomiali ma *non-deterministici*. Quindi $\mathcal{NP}$ sta per tempo *polinomiale non-deterministico*. Le due definizioni sono equivalenti.

**Lemma 1.4.4** *$\mathcal{P} \subseteq \mathcal{NP}$.*

DIMOSTRAZIONE. Informalmente il lemma è vero in quanto verificare una soluzione è più facile che trovarla. Formalmente dobbiamo provare che per un qualsiasi problema $X \in \mathcal{P}$ esiste un verificatore in tempo polinomiale. Poichè però $X \in \mathcal{P}$ è facile trovare un tale verificatore: è l'algoritmo efficiente che risolve $X$. Tale algoritmo può semplicemente ignorare il certificato che viene presentato al verificatore e risolvere il problema per poter decidere se esiste una soluzione. Si ricordi che dobbiamo solo stabilire se una soluzione esiste oppure no, e il certificato serve solo come prova dell'esistenza di una soluzione. Quindi trovarne un'altra, ignorando quella proposta, è perfettamente lecito. □

Quando un problema non è in $\mathcal{P}$ per mostrare che è in $\mathcal{NP}$ dovremo fornire un verificatore. Il verificatore si limiterà a controllare il certificato, quindi in genere è molto semplice. Ad esempio per il problema TSP basterà verificare che la potenziale soluzione,

cioè un giro del grafo, sia effettivamente un giro e abbia un costo minore o uguale alla soglia stabilita dal problema.

Ci sono tanti problemi in $\mathcal{NP}$ per i quali non si conoscono algoritmi efficienti e nemmeno si sa dire che tali algoritmi non esistono. Se tutti i problemi in $\mathcal{NP}$ ammettessero un algoritmo efficiente si avrebbe $\mathcal{P} = \mathcal{NP}$; se invece ci fosse qualche problema che non può essere risolto in modo efficiente, allora si avrebbe $\mathcal{P} \subset \mathcal{NP}$. Stabilire se $\mathcal{P} = \mathcal{NP}$ oppure se $\mathcal{P} \subset \mathcal{NP}$ è un problema aperto.

Nel seguito studieremo alcuni problemi per i quali non si sa se esiste o meno una soluzione efficiente. Trovare una soluzione efficiente per uno di questi problemi, o dimostrare che non esiste una tale soluzione, risolverebbe la questione $\mathcal{P}$-$\mathcal{NP}$. In particolare i problemi di cui tratteremo sono i seguenti:

- CircuitSat: Dato un circuito booleano, stabilire se esiste un assegnamento dell'input che produce il valore vero.
- Sat: Data una formula booleana, espressa come la congiunzione di clausole stabilire se esiste un assegnamento delle variabili che rende vera la formula.
- 3Sat: Un caso speciale di Sat in cui tutte le clausole hanno esattamente 3 termini.
- IndependentSet: Dato un grafo e un intero $k$, stabilire se esiste un insieme di nodi indipendenti (cioè senza archi che li collegano) di taglia almeno $k$.
- VertexCover: Dato un grafo e un intero $k$, stabilire se esiste un insieme di nodi che "copre" il grafo, cioè tale che tutti gli archi sono incidenti in almeno un nodo dell'insieme.
- SetCover: Una generalizzazione di VertexCover in cui vogliamo ricoprire un insieme arbitrario di oggetti utilizzando un insieme di insiemi.
- SetPacking: Una generalizzazione di IndependentSet in cui vogliamo trovare sottoinsiemi che non si intersecano fra loro.
- GraphColoring: Dato un grafo e un intero $k$, stabilire se i nodi del grafo possono essere colorati con al più $k$ colori facendo in modo che due nodi collegati da un arco non abbiano lo stesso colore.
- DirHamCycle: Dato un grafo, stabilire se esiste un ciclo hamiltoniano.
- TSP: Dato un grafo e un valore $C$, stabilire se è possibile visitare tutto il grafo con un "giro" di costo al massimo $C$.

Per ognuno dei problemi sopra elencati è facile fornire un verificatore, quindi tutti questi problemi appartengono a $\mathcal{NP}$.

## 1.5 Riduzioni

Lo studio della questione $\mathcal{P}$-$\mathcal{NP}$ riguarda la classificazione dei problemi in problemi "facili", cioè che possono essere risolti in tempo polinomiale, e problemi "difficili", per

i quali non si conosce un algoritmo polinomiale. Un punto cruciale è quello di poter dire che un problema A è "più facile" di un problema B, o equivalentemente, che un problema B è "più difficile" di un problema A. Per formalizzare affermazioni di questo tipo, sono molto utili le *riduzioni*. Informalmente, un problema A può essere ridotto a un problema B se possiamo usare B per risolvere A. In questo senso B è "più difficile" di A. Affinchè la riduzione sia utile è necessario che la difficoltà del processo di utilizzo di B per risolvere A non sia troppo grande: se la difficoltà della riduzione supera la difficoltà della risoluzione di B, potrebbe non convenire utilizzare la riduzione. Nel contesto della questione $\mathcal{P}$-$\mathcal{NP}$, è necessario che la riduzione sia efficiente, cioè che il lavoro necessario per utilizzare B per risolvere A sia polinomiale. In questo modo la risoluzione, in tempo polinomiale[7], di A attraverso B, usando quindi la riduzione, dipende solo dall'efficienza della soluzione per B.

[7] Si noti che ciò che interessa è la risolvibilità del problema in tempo polinomiale.

Per definire formalmente la nozione di riducibilità, supporremo di poter disporre di un oracolo per la risoluzione dei problemi. L'oracolo, schematizzato nella Figura 1.3, è capace di risolvere qualsiasi problema in tempo costante.

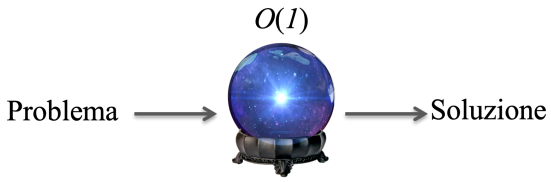



Figura 1.3: Oracolo: risolve qualsiasi problema in tempo $O(1)$

**Definizione 1.5.1** *Un problema A può essere ridotto a B, se è possibile risolvere una qualsiasi istanza di A usando tempo polinomiale e sfruttando un oracolo che risolve B. Scriveremo* $A \leq_P B$.

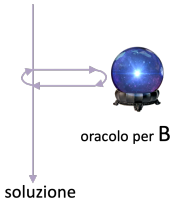



Figura 1.4: Risoluzione di A tramite una riduzione a B

Osserviamo che l'oracolo può essere chiamato più di una volta; spesso basterà una sola invocazione dell'oracolo, ma in generale è possibile sfruttarlo un numero polinomiale di volte. Quando si invoca l'oracolo è necessario fornire in qualche modo l'input del problema da risolvere e, successivamente, è necessario leggere l'output dell'oracolo. Nel costo della riduzione dobbiamo calcolare il tempo necessario sia a scrivere l'input per l'oracolo sia a leggere l'output dell'oracolo.

Dalla Definizione 1.5.1 si deduce immediatamente il seguente risultato:

**Lemma 1.5.2** *Assumiamo che* $A \leq_P B$. *Se B può essere risolto in tempo polinomiale allora anche A può essere risolto in tempo polinomiale.*

Dimostrazione. Poichè $A \leq_P B$, possiamo dare un algoritmo che risolve $A$ sfruttando la riduzione: questo algoritmo fa esattamente quello che viene fatto nella riduzione e al posto dell'oracolo, che non esiste, usa un algoritmo efficiente che risolve B, che invece esiste, in quanto $B$ può essere risolto in tempo polinomiale. Il costo totale di questo algoritmo sarà comunque polinomiale in quanto la riduzione richiede al massimo un numero polinomiale di passi, e anche se ognuno di questi passi fosse una invocazione dell'oracolo, quindi un utilizzo dell'algoritmo che risolve B, costerebbe comunque un

tempo polinomiale: moltiplicando un polinomio per un altro polinomio si ottiene comunque un polinomio. □

Il Lemma 1.5.2 è molto utile per trovare soluzioni efficienti a nuovi problemi riducendoli a problemi per i quali già si conosce una soluzione efficiente. Ad esempio il problema del matching bipartito può essere ridotto a un problema di massimo flusso (vedi il Capitolo 7 di KT2014 [20]).

La Definizione 1.5.1 implica anche un risultato simmetrico a quello del Lemma 1.5.2:

**Lemma 1.5.3** *Assumiamo che $A \leq_P B$. Se $A$ non può essere risolto in tempo polinomiale allora anche $B$ non può essere risolto in tempo polinomiale.*

In realtà i Lemmi 1.5.2 e 1.5.3 sono equivalenti, e rappresentano solo due modi diversi di vedere la stessa cosa. Il Lemma 1.5.3 fornisce un modo più diretto per provare che alcuni problemi sono "difficili". Infatti se $A$ è un problema difficile e si prova che $A \leq_P B$ allora si è provato che anche $B$ è un problema difficile.

Poichè di fatto non sappiamo se i problemi difficili che studiamo possano o meno essere risolti in modo efficiente, la riduzione è uno strumento per classificarli in base alla difficoltà relativa fra di essi.

Non è difficile vedere che la relazione $\leq_P$ è *transitiva*.

**Lemma 1.5.4** *Se $A \leq_P B$ e $B \leq_P C$ allora $A \leq_P C$.*

DIMOSTRAZIONE. Per ridurre $A$ a $C$ è sufficiente "passare" per $B$. Cioè data l'istanza di $A$ da risolvere, poichè $A \leq_P B$ si ha che avendo un oracolo per risolvere $B$ si può risolvere $A$. Anche non avendo un oracolo per $B$ (ma solo uno per $C$), si ha che poichè $B \leq_P C$ è possibile risolvere $B$ usando l'oracolo per $C$. Quindi si può comunque risolvere $A$ usando l'oracolo per $C$, il tutto sempre usando tempo polinomiale nei vari passaggi che non coinvolgono l'oracolo. □

### 1.5.1 IndependentSet e VertexCover

Prima di tutto definiamo formalmente i problemi IndependentSet e VertexCover e vediamo degli esempi. Successivamente vedremo che essi sono equivalenti dal punto di vista computazionale, cioè l'uno può essere ridotto all'altro.

Dato un grafo $G = (V, E)$, un insieme di nodi $S \subseteq V$ è *indipendente* se in $E$ non ci sono archi fra i nodi di $S$.

**Problema 1.5.5** IndependentSet*: Dato un grafo $G$ e un intero $k$, $G$ contiene un insieme indipendente di taglia almeno $k$?*

È facile trovare insiemi indipendenti di piccole dimensioni. Infatti ogni singolo nodo è un insieme indipendente di taglia 1. Per sapere se esiste un insieme indipendente di taglia 2 basta controllare tutte le coppie di nodi e vedere se almeno una non è collegata da un arco o equivalentemente se il grafo è completo, cioè tutte le coppie di nodi sono collegate da un arco (è sufficiente controllare il numero di archi: se è minore di

$n(n-1)/2$ allora esiste un insieme indipendente di taglia 2, se è uguale a $n(n-1)/2$ allora il grafo è completo e quindi non esiste un insieme indipendente di taglia 2).

Come esempio consideriamo il grafo riportato in Figura 1.5.

I nodi $\{2,4\}$ formano un insieme indipendente di taglia 2. I nodi $\{1,3,5\}$ formano un insieme indipendente di taglia 3. Non è difficile verificare che in questo grafo non esistono insiemi indipendenti di taglia più grande: basta controllare che tutti gli insiemi di taglia 4 hanno almeno un arco che collega una coppia di nodi.

Al crescere della taglia però diventa sempre più difficile trovare insiemi indipendenti o verificare che non esistono. Infatti il numero di sottoinsimi di taglia $k$ è $\binom{n}{k}$, quindi per controllarli tutti abbiamo bisogno di tempo esponenziale.

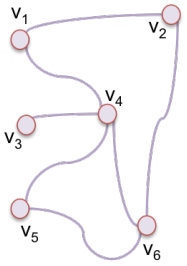


Figura 1.5: Un grafo con un insieme indipendente di taglia 3

Abbiamo posto il problema come un problema decisionale: dato un grafo *decidere* (stabilire) se in quel grafo esiste un insieme indipendente di taglia almeno $k$. Lo stesso problema potremmo vederlo come un problema di ottimizzazione: dato un grafo, trovare l'insieme indipendente più grande.

Passiamo adesso al problema VertexCover. Dato un grafo $G = (V, E)$ un insieme $S \subseteq V$ è un insieme di vertici che ricopre (gli archi di) $G$ se ogni arco $e \in E$ ha almeno uno dei suoi due vertici in $S$. Si noti che ad essere "coperti" sono gli archi del grafo e che ogni vertice del grafo "copre" tutti gli archi incidenti su di esso. È facile trovare degli insiemi di vertici ricoprenti: ad esempio $S = V$ è un insieme ricoprente. È più difficile trovare insiemi ricoprenti più piccoli. Considerando di nuovo il grafo della Figura 1.5 si ha che l'insieme di nodi $\{1,2,3,5\}$ è un insieme ricoprente di taglia 4 e che l'insieme di nodi $\{2,4,5\}$ è un insieme ricoprente di taglia 3. Il problema di ottimizzazione richiederebbe l'individuazione dalla taglia minima di un insieme ricoprente. Si può verificare che, in questo grafo, non esistono insiemi ricoprenti di taglia 2. Quindi l'insieme $\{2,4,6\}$ è un insieme ricoprente di taglia minima. Nella versione decisionale del problema ci chiediamo:

**Problema 1.5.6** VertexCover*: Dato un grafo $G$ e un intero $k$, $G$ contiene un insieme ricoprente di taglia al massimo $k$?*

Non si conoscono algoritmi efficienti per risolvere IndependentSet e VertexCover. Possiamo però provare che questi due problemi sono equivalenti nel senso che uno può essere ridotto all'altro, cioè si ha sia che IndependentSet$\leq_P$VertexCover ma anche che VertexCover$\leq_P$IndependentSet.

**Lemma 1.5.7** *Sia $G = (V, E)$ un grafo. Un inseme $S \subset V$ è un insieme indipendente se e solo se $V \setminus S$ è un insieme ricoprente.*

Dimostrazione. Sia $S$ un insieme indipendente. Consideriamo un arco qualsiasi $e = (u, v)$. Poichè $S$ è indipendente $u$ e $v$ non possono entrambi appartenere ad $S$. Quindi almeno uno dei due deve appartenere a $V \setminus S$. Poichè questo è vero per tutti gli archi si ha che $V \setminus S$ è un insieme ricoprente.

Assumiamo adesso che $V \setminus S$ sia un insieme ricoprente. Consideriamo due nodi qualsiasi $u, v$ dell'insieme $S$. Non può esistere l'arco $e = (u, v)$; infatti se esistesse tale arco esso non sarebbe "coperto" da nessun nodo di $V \setminus S$ contraddicendo il fatto che

$V \setminus S$ è un insieme ricoprente. Poichè ciò è vero per qualunque coppia di nodi di $S$, concludiamo che $S$ è un insieme indipendente. □

Usando il lemma precedente possiamo provare che i problemi IndependentSet e VertexCover sono di fatto equivalenti.

**Lemma 1.5.8** IndependentSet$\leq_P$VertexCover.

Dimostrazione. Per provare la riduzione dobbiamo far vedere come risolvere IndependentSet sfruttando un oracolo che risolve VertexCover, ed usando al massimo tempo polinomiale.

Consideriamo un'istanza del problema IndependentSet. Tale istanza è costituita da un grafo $G$ e un intero $k$: dobbiamo decidere se $G$ ha un insieme indipendente di taglia almeno $k$.

Abbiamo a disposizione un oracolo per VertexCover. Pertanto, per usarlo, dobbiamo costruire un'istanza di VertexCover. L'istanza deve essere costruita in modo tale che la soluzione per VertexCover fornita dall'oracolo ci permetta di trovare una soluzione per l'istanza di IndependentSet. Un'istanza per VertexCover è costituita anch'essa da un grafo $G'$ e da un intero $k'$: l'oracolo dirà se $G'$ ha un insieme ricoprente di taglia almeno $k'$.

Per creare l'istanza da dare in input all'oracolo usiamo $G' = G$ e $k' = n - k$, dove $n$ è il numero di nodi. Questo passaggio richiede tempo polinomiale (di fatto costante in quanto non dobbiamo fare quasi nulla, se non calcolare $k'$) e per "scrivere" l'input per l'oracolo è sufficiente tempo lineare in quanto occorre semplicemente copiare $G$ e scrivere $k'$.

Preparato l'input, l'oracolo ci dirà se $G' = G$ contiene un insieme ricoprente con al massimo $k' = n - k$ nodi. Sia $r \in \{\texttt{si}, \texttt{no}\}$ la risposta dell'oracolo. Per il Lemma 1.5.7 si ha che $r$ è anche la risposta alla domanda "Esiste un insieme indipendente di almeno $n - k' = k$ nodi?". Quindi $r$ è anche la risposta all'istanza del problema IndependentSet. □

**Lemma 1.5.9** VertexCover$\leq_P$IndependentSet.

Dimostrazione. Questa dimostrazione è simile a quella del lemma precedente. La si svolga come esercizio. □

Riassumendo, anche se per nessuno dei due problemi sappiamo se esiste o meno una soluzione efficiente, sappiamo che se riusciamo a risolvere in modo efficiente uno dei due possiamo risolvere efficientemente anche l'atro. E anche che se dimostriamo che uno dei due problemi non può essere risolto efficientemente, avremo dimostrato che anche l'altro non può essere risolto efficientemente.

### 1.5.2 SetCover

Consideriamo adesso un altro problema: SetCover. Il problema del ricoprimento degli archi può essere visto come un caso speciale di un problema più generale, SetCover, in cui vogliamo ricoprire un insieme arbitrario di oggetti utilizzando un insieme di insiemi. Più formalmente:

**Problema 1.5.10** SETCOVER: *dato un insieme $U$ di $n$ elementi, una collezione $S_1, \ldots, S_m$ di sottoinsiemi di $U$ e un intero $k$, esiste una collezione di al massimo $k$ fra gli $m$ sottoinsiemi $S_1, \ldots, S_m$ tali che l'unione di tali sottoinsiemi è uguale a $U$?*

La Figura 1.6 mostra un esempio con $n = 9$, $U = \{a, b, c, d, e, f, g, h, i\}$, $m = 7$ e $S_1 = \{a, b, c\}$, $S_2 = \{d, e\}$, $S_3 = \{g, h, i\}$, $S_4 = \{d, f, g, i\}$, $S_5 = \{d, f\}$, $S_6 = \{a, d, f\}$, e $S_7 = \{e, c, h\}$. Un insieme ricoprente è $\{S_1, S_4, S_7\}$.

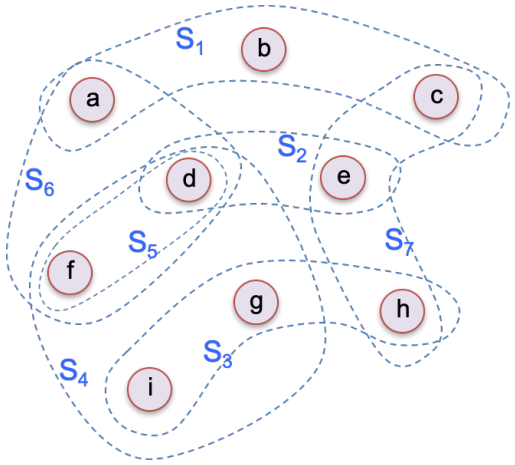



Figura 1.6: Grafo di esempio per SETCOVER

Un esempio concreto in cui potremmo modellare la realtà con un'istanza del problema SETCOVER è il seguente: in un importante sito turistico un'agenzia organizza escursioni giornaliere alle quali partecipano molti visitatori provenienti da tutto il mondo. L'agenzia ha a disposizione delle guide ognuna delle quali sa parlare un certo numero di lingue. Per ogni giro turistico l'agenzia vuole selezionare il minimo numero di guide da impiegare in modo tale che per ogni partecipante ci sia almeno una guida che parli la sua lingua. In questo caso l'insieme $U$ è l'insieme di tutte le lingue parlate dal gruppo di partecipanti all'escursione mentre ogni guida corrisponde a un sottoinsieme $S_i \subseteq U$, dato dalle lingue parlate dalla guida.

VERTEXCOVER è un caso speciale di SETCOVER: nel caso di VERTEXCOVER, l'insieme $U$ è dato dall'insieme degli archi, mentri gli insiemi $S_i$, uno per nodo, sono dati dagli archi incidenti nel nodo stesso (un nodo ricopre tutti gli archi ad esso incidenti). Pertanto non sorprende il fatto che:

**Lemma 1.5.11** VERTEXCOVER$\leq_P$SETCOVER.

DIMOSTRAZIONE. Per provare la riduzione dobbiamo far vedere come risolvere VERTEXCOVER sfruttando un oracolo che risolve SETCOVER, ed usando al massimo tempo polinomiale.

Consideriamo un'istanza del problema VERTEXCOVER. Tale istanza è costituita da un grafo $G$ e un intero $k$: dobbiamo decidere se $G$ ha un insieme ricoprente di al massimo $k$ nodi.

Abbiamo a disposizione un oracolo per SETCOVER. Pertanto, per usarlo, costruiamo

un'istanza per SetCover; una tale istanza è costituita da un insieme $U$, da $m$ sottoinsiemi di $U$, $S_1, \ldots, S_m$ e da un intero $k'$.

La specifica istanza che costruiamo è la seguente. L'insieme $U$ è costituito da tutti gli archi di $G$, cioè $U = E$. Inoltre per ogni vertice $i \in V$, consideriamo l'insieme $S_i \subseteq U$, costituito da tutti gli archi di $E$ incidenti su $i$. Infine, $k' = k$. La costruzione di questa istanza può essere chiaramente fatta in tempo polinomiale.

L'istanza di SetCover che abbiamo appena costruita ammette un insieme ricoprente di al massimo $k' = k$ degli insiemi $S_1, \ldots, S_n$ se e solo se il grafo $G$ ha un insieme ricoprente di al massimo $k = k'$ nodi. Infatti, sia $S_{i_1}, \ldots, S_{i_\ell}$ un insieme ricoprente per $U$ con $\ell \leq k'$, allora si ha che ogni elemento di $U$ è coperto da uno degli insiemi $S_{i_j}$. Poichè $U = E$ e per come abbiamo definito gli insiemi $S_{i_j}$ si ha che $\{i_1, \ldots, i_\ell\}$ è un insieme ricoprente di $G$ di taglia $\ell \leq k' = k$. Analogamente, sia $\{i_1, \ldots, i_\ell\}$ un insieme ricoprente per $G$, con $\ell \leq k$, allora $S_{i_1}, \ldots, S_{i_\ell}$ è un insieme ricoprente per $U$, con $\ell \leq k' = k$.

Pertanto per risolvere il problema VertexCover su $G$ possiamo semplicemente costruire l'istanza di SetCover come descritto prima e usare l'oracolo per risolvere questa istanza. La risposta data al problema SetCover è anche la risposta al problema VertexCover. □

Notiamo che sia nella prova appena fatta sia nelle prove fatte per le precedenti riduzioni, sebbene la definizione di riduzione permetta l'utilizzo dell'oracolo fino a un numero polinomiale di volte, l'oracolo viene invocato esattamente una volta. In tutti i casi, partendo da un'istanza del problema da risolvere, abbiamo creato un'istanza del problema per il quale abbiamo l'oracolo e abbiamo usato l'oracolo una sola volta per risolvere la nuova istanza, la cui soluzione è anche una soluzione del problema originario. Questo modo di procedere è abbastanza comune e sarà così per tutte le riduzioni che vedremo. Si rammenti però che in una riduzione è lecito usare l'oracolo fino a un numero polinomiale di volte.

### *1.5.3* SetPacking

Come VertexCover può essere generalizzato in SetCover, così IndependentSet può essere generalizzato nel seguente problema:

**Problema 1.5.12** SetPacking*: Dato un insieme $U$ di elementi, una collezione $S_1, \ldots, S_m$ di sottoinsiemi di $U$, e un intero $k$, esiste una collezione di almeno $k$ di questi sottoinsiemi tali che nessuno di loro si interseca con un altro?*

Non dovrebbe sorprendere (omettiamo la dimostrazione) che:

**Lemma 1.5.13** IndependentSet$\leq_P$SetPacking.

### *1.5.4* Sat *e* 3Sat

Sat e 3Sat sono 2 problemi simili formulati in termini di funzioni booleane. Essi modellano un ampio insieme di problemi in cui sono in gioco variabili decisionali a cui deve essere assegnato un valore nel rispetto di determinati vincoli.

SAT è definita come una formula booleana su un insieme di $n$ variabili booleane $x_1, \ldots, x_n$, espressa come l'AND di OR. Più precisamente, la formula è espressa come

$$\phi = C_1 \cdot C_2 \cdot \ldots \cdot C_k$$

dove ogni *clausola* $C_i = t_{i_1} + t_{i_2} + \ldots + t_{i_\ell}$, ed ogni *letterale* $t_{i_s}$ è una delle variabili $x_j$ oppure la sua negazione $\bar{x}_j$.

Il problema della soddisfacibilità di formule booleane è definito come segue.

**Problema 1.5.14** SAT: *Data una formula booleana $\phi$, espressa come l'*AND *di* OR *su un insieme di variabili $\{x_1, \ldots, x_n\}$, esiste un'assegnazione di valori delle variabili che rende vera la formula?*

Consideriamo come esempio la seguente formula

$$\phi_1 = (x_1 + x_4) \cdot \overline{x_3} \cdot (x_1 + \overline{x_2} + x_3 + \overline{x_4}) \cdot (x_2 + x_3) \cdot (\overline{x_1} + x_2 + x_3).$$

La formula $\phi$ è composta da 5 clausole $\phi_1 = C_1 \cdot C_2 \cdot C_3 \cdot C_4 \cdot C_5$, con:

$$\begin{aligned} C_1 &= x_1 + x_4 \\ C_2 &= \overline{x_3} \\ C_3 &= x_1 + \overline{x_2} + x_3 + \overline{x_4} \\ C_4 &= x_2 + x_3 \\ C_5 &= \overline{x_1} + x_2 + x_3 \end{aligned}$$

L'assegnamento $x_1 = 1$, $x_2 = 1$, $x_3 = 0$ e $x_4 = 1$, ha come conseguenza $\phi = 1$. La formula $\phi_2 = \overline{x_1} \cdot (x_1 + x_2) \cdot \overline{x_2}$, invece, non ammette assegnamenti che la rendono vera.

Esiste un caso speciale del problema SAT che di fatto è equivalente alla formulazione generale. Tale caso speciale si ha quando tutte le clausole sono composte da esattamente 3 letterali. Ad esempio la formula

$$\phi_3 = (x_1 + x_2 + x_4) \cdot (x_2 + \overline{x_3} + x_4) \cdot (\overline{x_1} + \overline{x_2} + x_3) \cdot (\overline{x_1} + \overline{x_3} + \overline{x_4})$$

è composta da 4 clausole, ognuna avente esattamente 3 letterali:

$$\begin{aligned} C_1 &= x_1 + x_2 + x_4 \\ C_2 &= x_2 + \overline{x_3} + x_4 \\ C_3 &= \overline{x_1} + \overline{x_2} + x_3 \\ C_4 &= \overline{x_1} + \overline{x_3} + \overline{x_4} \end{aligned}$$

**Problema 1.5.15** 3SAT: *Data una formula booleana $C_1 \cdot C_2 \cdot \ldots \cdot C_k$ su un insieme di variabili $\{x_1, \ldots, x_n\}$, con ognuna delle clausole $C_i$ avente esattamente 3 letterali, esiste un'assegnazione di valori delle variabili che rende vera la formula?*

Nel caso della formula $\phi_3$ la risposta a tale domanda è affermativa: un assegnamento che rende vera la formula è $x_1 = 1$, $x_2 = 1$, $x_3 = 1$, $x_4 = 0$.

La definizione del problema 3Sat è la stessa del problema Sat fatta eccezione per il vincolo dei 3 letterali per clausola. Quindi $3\text{Sat} \leq_P \text{Sat}$, in quanto 3Sat è un caso particolare di Sat. Meno intuitivo è il fatto che che una qualsiasi istanza di Sat può essere trasformata in una equivalente istanza di 3Sat: l'Esercizio 6 chiede di provare questo fatto. Dunque si ha che $\text{Sat} \leq_P 3\text{Sat}$, pertanto i due problemi sono equivalenti.

Proviamo adesso che 3Sat può essere ridotto a IndependentSet.

**Lemma 1.5.16** $3\text{Sat} \leq_P$ IndependentSet.

Dimostrazione. Abbiamo a disposizione un oracolo per risolvere IndependentSet e vogliamo risolvere 3Sat. Sia $\phi = C_1 \cdot C_2 \cdot \ldots \cdot C_k$, l'istanza di 3Sat in cui ogni clausola $C_i$ è costituita da 3 letterali delle variabili $x_1, \ldots, x_n$.

Per trovare un assegnamento alle variabili che renda vera la formula dobbiamo individuare in ognuna delle clausole un particolare letterale al quale deve essere dato il valore vero (ne serve almeno uno per clausola). La cosa potrebbe sembrare facile in quanto per rendere vera una clausola basta dare il valore vero anche a un solo dei tre letterali. Il problema è che letterali in clausole diverse potrebbero essere in conflitto fra di loro. Ad esempio una clausola potrebbe contenere $x_i$ e un'altra clausola potrebbe contenere $\bar{x}_i$. In questo caso non possiamo scegliere entrambi questi letterali per rendere vere le due clausole in quanto non c'è modo di renderli entrambi veri. Tuttavia se evitando tutti i letterali che vanno in conflitto riusciamo comunque ad individuarne uno da poter rendere vero per ogni clausola, allora abbiamo trovato un assegnamento che rende vera la formula.

Quanto detto poc'anzi è utile per trasformare l'istanza di 3Sat in una equivalente istanza di IndependentSet, anche se la costruzione del grafo non è immediata. Lo costruiamo nel seguente modo. Il grafo contiene $3k$ nodi, 3 per ognuna delle $k$ clausole, ed ogni nodo in un gruppo di 3 corrisponde a uno dei letterali della clausola.

Più formalmente, per $j = 1, \ldots, k$, costruiamo 3 vertici, $v_{j,1}, v_{j,2}$ e $v_{j,3}$. I tre vertici corrispondenti alla stessa clausola saranno uniti da 3 archi per formare un triangolo, come mostrato nella Figura 1.7. L'intera formula conterrà quindi $k$ di questi triangoli, uno per ogni clausola. Usando come esempio la formula $\phi_3$ vista in precedenza si avrebbe il grafo mostrato nella Figura 1.8.

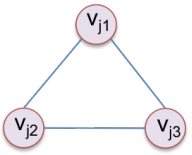



Figura 1.7: Il triangolo di nodi che rappresenta la clausola $C_j$

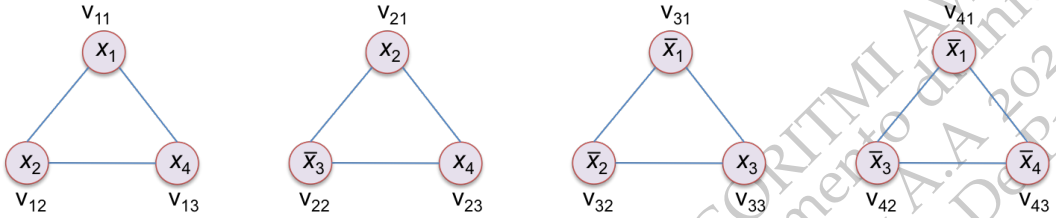



Figura 1.8: Grafo dei triangoli per la formula $\phi_3$

Gli archi nei triangoli servono a selezionare un solo nodo per ogni triangolo nell'insieme indipendente: questo corrisponderà a rendere vero quel letterale (e questo ci basta per rendere vera la clausola alla quale il letterale appartiene) per ogni clausola. Tuttavia non possiamo rendere veri i letterali in modo arbitrario in quanto così facendo potremmo dover rendere vero sia un letterale che la sua negazione e ciò è, come abbiamo già detto, ovviamente impossibile. Pertanto dovremo aggiungere al grafo che stiamo costruendo degli archi per codificare i conflitti fra i letterali. Dovremo aggiungere un arco fra due nodi in triangoli diversi che rappresentano due letterali uno la negazione dell'altro. Come esempio, il grafo della Figura 1.8 diventerà il grafo della Figura 1.9, che chiameremo $G_{3\text{SAT}}$. La costruzione richiede tempo polinomiale in quanto il numero di nodi e di archi da specificare è proporzionale al numero di clausole della formula di partenza.

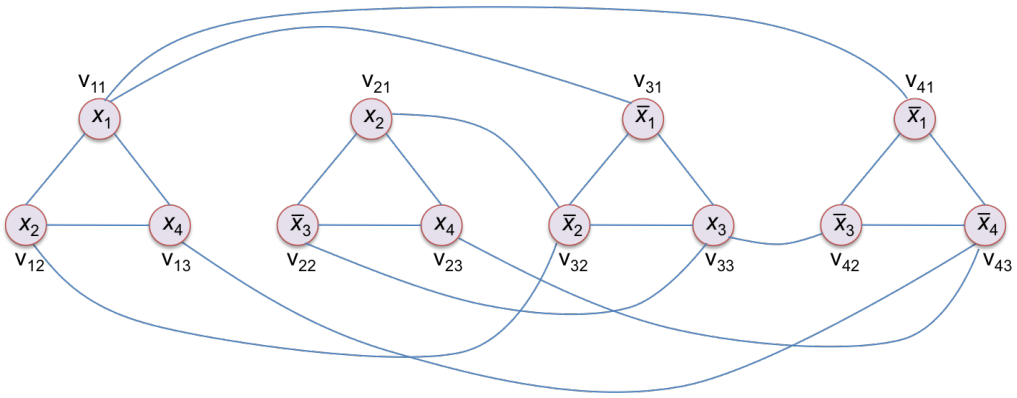



Figura 1.9: Grafo $G_{3\text{SAT}}$ per la formula $\phi_3$

Il grafo così costruito ha la seguente proprietà: esiste un insieme indipendente di taglia $k$ se e solo se la formula di partenza è soddisfacibile. Infatti se la formula è soddisfacibile allora esiste un assegnamento che rende vero almeno un letterale per ogni clausola. I nodi del grafo corrispondenti a questi letterali formano un insieme indipendente: infatti non può esistere un arco fra due di questi nodi in quanto ognuno di essi fa parte un gruppo di 3 diverso e ogni coppia non può essere in conflitto in quanto tutti hanno valore vero. Viceversa se esiste un insieme indipendente di $k$ vertici (l'insieme indipendente non può avere più di $k$ nodi in quanto ce ne sarebbero due appartenenti allo stesso triangolo e che quindi sarebbero collegati da un arco), è possibile soddisfare la formula assegnando vero ai letterali corrispondenti a quei $k$ vertici: ognuno di essi è in una clausola diversa in quanto i 3 nodi di una clausola sono legati da un arco e nessuna coppia è in conflitto in quanto le coppie in conflitto sono anch'esse legate da un arco.

Riconsiderando l'esempio della Figura 1.9 si ha che l'insieme di nodi $\{v_{11}, v_{21}, v_{33}, v_{43}\}$ è un insieme indipendente di taglia 4. Tale insieme corrisponde all'assegnamento $x_1 = 1$, $x_2 = 1$, $x_3 = 1$, $x_4 = 0$ che come abbiamo visto in precedenza rende vera la formula $\phi_3$. Un qualsiasi altro insieme indipendente di $G_{3\text{SAT}}$ corrisponde a un assegnamento che rende vera $\phi_3$ e un qualsiasi assegnamento che rende vera $\phi_3$ corrisponde a un insieme

indipendente nel grafo $G_{3\text{SAT}}$.

A questo punto usiamo l'oracolo per risolvere il problema INDEPENDENTSET e vedere se esiste un insieme indipendente di taglia $k$: se esiste la formula è soddisfacibile, se non esiste la formula non è soddisfacibile. □

### 1.5.5 CIRCUITSAT

Prima di tutto dobbiamo specificare cosa intendiamo per circuito. Consideriamo gli operatori booleani standard: $\wedge$ (AND), $\vee$ (OR), e $\neg$ (NOT). Un circuito è rappresentato da un grafo direzionato che descrive un circuito fisico fatto di porte AND, OR e NOT, come mostrato ad esempio nella Figura 1.10.

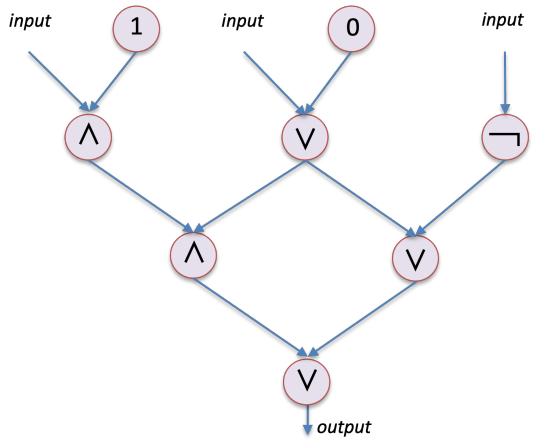



Figura 1.10: Un circuito

I nodi con un arco entrante etichettato con *input* rappresentano l'input al circuito; i nodi etichettati con una costante (0 oppure 1), rappresentano dei valori costanti usati dal circuito; infine i nodi etichettati con un operatore booleano, $\wedge$, $\vee$ e $\neg$, rappresentano le corrispondenti operazioni booleane. Gli archi entranti in un nodo che rappresenta una operazione portano l'input per l'operazione, mentre l'arco uscente veicola l'output dell'operazione. L'ouput dell'operazione senza archi uscenti è l'output del circuito.

Un circuito calcola una funzione booleana dei suoi input: la funzione è quella descritta dal circuito stesso: ognuno degli operatori booleani prende l'input e calcola l'output fino ad arrivare al nodo senza archi uscenti la cui computazione sui valori passati sugli archi entranti (o sull'arco entrante) fornisce il valore di output del circuito.

Ad esempio, il circuito della Figura 1.10 sull'input $1, 0, 0$ assegnato ai 3 nodi di input, produce come output delle 3 porte booleane più in alto, rispettivamente, $1, 0, 1$, quindi come output delle successive 2 porte booleane 0 e 1, ed infine come output della porta $\vee$ più in basso, e quindi come output del circuito, il valore 1.

Avendo definito un circuito, possiamo ora definire il problema della soddisfacibilità di un circuito. Dato un circuito, il problema della soddisfacibilità del circuito consiste nello stabilire se esiste un assegnamento di input che causa un valore di output pari a 1. Se ciò è possibile diremo che il circuito è soddisfacibile. Nell'esempio precedente

il circuito è soddisfacibile, infatti abbiamo visto che l'assegnamento $1, 0, 0$ all'input produce come output il valore 1.

**Lemma 1.5.17** CircuitSat$\leq_P$3Sat.

Dimostrazione. Consideriamo una istanza arbitraria di CircuitSat. Vogliamo costruire una equivalente istanza di 3Sat, problema per il quale abbiamo a disposizione un oracolo. Per costruire tale formula utilizzeremo una variabile per ogni input (variabile o costante) e per ogni porta del circuito. Per far sì che la formula sia equivalente al circuito, tutte le variabili che sono di fatto vincolate perchè rappresentano il valore di una costante o di una porta del circuito saranno rappresentate da clausole che in qualche modo forzano il valore corretto.

Per differenziare le variabili "libere" da quelle vincolate (che rappresentano valori costanti o il valore di output di una porta del circuito) utilizzeremo per le prime la lettera $x$, quindi $x_1, x_2, \ldots$ saranno le variabili di input del circuito, mentre per le altre la lettera $y$, quindi $y_1, y_2, \ldots$ saranno le variabili che rappresentano le costanti o i valori di uscita delle porte del circuito.

La formula $\phi$ che vogliamo costruire deve vincolare i valori delle variabili $y$ in modo tale che esse rappresentino la computazione del circuito.

Consideriamo ad esempio una porta $\neg$ con input $a$ (che può essere una $x$ o una $y$) e output $y$ (vedi Figura 1.11). Vogliamo inserire una clausola per codificare il fatto che $y$ deve essere la negazione di $a$. Tale clausola deve valere 1 se e solo se $y = \overline{a}$. In altre parole cerchiamo una clausola $C$ tale che la seguente tavola della verità sia soddisfatta:

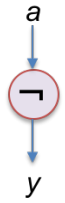



Figura 1.11: Porta NOT

| $a$ | $y$ | $C$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

In questo modo il valore di $y$ sarà il negato di $a$ se e solo se $C = 1$, quindi $C$ codifica la porta $\neg$. La formula $C = (a \vee y) \wedge (\overline{a} \vee \overline{y})$ soddisfa la proprietà richiesta. Dunque in realtà stiamo inserendo due clausole, ognuna di due letterali, per codificare una porta $\neg$.

Analogamente possiamo fare per le porte $\wedge$ e $\vee$. Consideriamo una porta AND con input $a$ e $b$ (che possono essere sia delle $x$ che delle $y$) e output $y$ (vedi Figura 1.12). In questo caso la clausola $C$ deve codificare il fatto che $y = a \wedge b$. Considerando tutti i possibili valori di $a, b$, e $y$, inseriremo un 1 nella tavola di verità di $C$ per quelle combinazioni che soddisfano il vincolo:


Figura 1.12: Porta AND

| $a$ | $b$ | $y$ | $C$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

La formula $C = (\overline{y} \vee a) \wedge (\overline{y} \vee b) \wedge (y \vee \overline{a} \vee \overline{b})$ soddisfa la tavola della verità richiesta. Quindi inseriamo 3 clausole, due con 2 letterali e una con 3 letterali.

Infine consideriamo una porta OR con input $a$ e $b$ e output $y$ (vedi Figura 1.13). In questo caso la clausola $C$ deve codificare il fatto che $y = a \vee b$. La tavola della verità che specifica $C$ è la seguente:

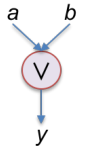



Figura 1.13: Porta OR

| $a$ | $b$ | $y$ | $C$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

La formula $C = (y \vee \overline{a}) \wedge (y \vee \overline{b}) \wedge (\overline{y} \vee a \vee b)$ soddisfa la tavola della verità richiesta. Come per una porta AND anche per una porta OR inseriamo 3 clausole, due con 2 letterali e una con 3 letterali.

Dobbiamo inserire anche delle clausole che codificano gli input di valori costanti. Ma questo è facile da fare in quanto per il valore 1 inseriamo la corrispondente variabile $y$ in forma vera, mentre per il valore 0 inseriamo la corrispondente variabile $y$ in forma negata, cioè $\overline{y}$.

L'ultima clausola da inserire è quella che codifica l'output ed ovviamente sarà la variabile $y$ di uscita della porta finale del circuito (in forma vera).

Codificando gli input e tutte le porte del circuito avremo creato una formula $\phi = C_1 \cdot C_2 \cdot \ldots$ equivalente al circuito, nel senso che il valore calcolato dal circuito per un determinato assegnamento delle variabili di input $x_1, x_2, \ldots$ è lo stesso valore che ha $\phi$ sullo stesso assegnamento delle variabili $x_1, x_2, \ldots$. La formula $\phi$ contiene un certo numero di clausole (il numero di clausole dipende da quante porte ci sono nel circuito e dalla tipologia delle porte), ognuna delle quali ha $1, 2$ o $3$ letterali.

L'ultimo passo da fare è quello di mostrare che la formula costruita, in cui le clausole hanno 1, 2 o 3 letterali, può essere trasformata in un'altra equivalente in cui tutte le clausole hanno esattamente 3 letterali.

Per fare ciò possiamo inserire delle variabili fittizie $z_1$ e $z_2$ forzandole a valere 0. Con tali variabili a disposizione è facile aumentare il numero di letterali di una clausola

semplicemente aggiungendo $z_1$ o $\overline{z_1}$ e $z_2$. Per forzare il valore di $z_1$ e $z_2$ ad essere 0 possiamo inserire le clausole $\overline{z_1}$ e $\overline{z_1}$.

Questo però crea il problema che queste due clausole hanno un solo letterale e non 3. Possiamo risolvere questo problema usando due ulteriori variabili fittizie, $z_3$ e $z_4$ e inserendo le seguenti 4 clausole al posto di $\overline{z_1}$:

$$(\overline{z_1} \vee z_3 \vee z_4), (\overline{z_1} \vee z_3 \vee \overline{z_4}), (\overline{z_1} \vee \overline{z_3} \vee z_4), (\overline{z_1} \vee \overline{z_3} \vee \overline{z_4}),$$

e le seguenti 4 clausole al posto di $\overline{z_2}$:

$$(\overline{z_2} \vee z_3 \vee z_4), (\overline{z_2} \vee z_3 \vee \overline{z_4}), (\overline{z_2} \vee \overline{z_3} \vee z_4), (\overline{z_2} \vee \overline{z_3} \vee \overline{z_4}).$$

La presenza di queste 8 clausole impone che per soddisfare la formula dobbiamo necessariamente avere $z_1 = z_2 = 0$, indipendentemente dai valori di $z_3$ e $z_4$.

Questo ci permette di trasformare le clausole con 1 o 2 letterali in equivalenti clausole con esattamente 3 letterali: se una clausola ha un solo letterale $t$, la trasformiamo in $t \vee z_1 \vee z_2$; se ne ha due allora la trasformiamo in $t_1 \vee t_2 \vee z_1$. □

Facciamo un esempio. Consideriamo il circuito della Figura 1.10 ed associamo ad ogni porta una variabile così come mostrato nella Figura 1.14.

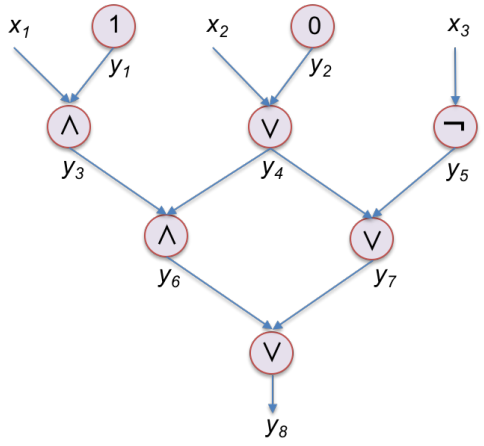



Figura 1.14: Il circuito della Figura 1.10 con le variabili ($x_i$ e $y_i$) associate alle porte.

Abbiamo usato la lettera $x$ per le variabili che rappresentano l'input ($x_1, x_2$ e $x_3$) e la lettera $y$ per le variabili vincolate al valore di uscita di una porta (da $y_1$ a $y_8$). La variabile $y_1$ rappresenta il valore costante 1 pertanto inseriremo la clausola $C_1 = y_1$. Per la variabile $y_2$ che rappresenta il valore costante 0, invece inseriremo la clausola $C_2 = \overline{y_2}$. La variabile $y_3$ rappresenta il valore di uscita di una porta AND i cui input sono $x_1$ e $y_1$. Pertanto, per la variabile $y_3$, inseriremo le clausole $C_3 = \overline{y_3} + x_1$, $C_4 = \overline{y_3} + y_1$, $C_5 = y_3 + \overline{x_1} + \overline{y_1}$. La variabile $y_4$, invece, rappresenta il valore di uscita di una porta OR i cui input sono $x_2$ e $y_2$. Pertanto, per la variabile $y_4$, inseriremo le clausole $C_6 = y_4 + \overline{x_2}$, $C_7 = y_4 + \overline{y_2}$, $C_8 = \overline{y_4} + x_2 + y_2$. Per la variabile $y_5$ che rappresenta l'output di una porta NOT il cui input è $x_3$, introdurremo le clausole $C_9 = y_5 + \overline{x_3}$, $C_{10} = \overline{x_3} + y_5$. Procedendo in modo simile, per la variabile $y_6$ introdurremo le clausole

$C_{11} = \overline{y_6} + x_3$, $C_{12} = \overline{y_6} + y_4$, $C_{13} = y_6 + \overline{x_3} + \overline{y_4}$, per la variabile $y_7$ le clausole $C_{14} = y_7 + \overline{y_4}$, $C_{15} = y_7 + \overline{y_5}$, $C_{16} = \overline{y_7} + y_4 + y_5$, e per la variabile $y_8$ le clausole $C_{17} = y_8 + \overline{y_6}$, $C_{18} = y_8 + \overline{y_7}$, $C_{19} = \overline{y_8} + y_6 + y_7$. Infine, introduciamo una clausola per il valore di output del circito: $C_{20} = y_8$.

Queste 20 clausole permettono di rappresentare il circuito; tuttavia non tutte le clausole hanno 3 letterali. Pertanto introdurremo le otto clausole aggiuntive con le variabili $z_1, z_2, z_3$ e $z_4$, e modificheremo le clausole con uno o due letterali aggiungendo $z_1$ e $z_2$, ottenendo la formula 3SAT finale che contiene 28 clausole (mostrate a margine).

$$
\begin{aligned}
C_1 &= y_1 + z_1 + z_2 \\
C_2 &= \overline{y_2} + z_1 + z_2 \\
C_3 &= \overline{y_3} + x_1 + z_1 \\
C_4 &= \overline{y_3} + y_1 + z_1 \\
C_5 &= y_3 + \overline{x_1} + \overline{y_1} \\
C_6 &= y_4 + \overline{x_2} + z_1 \\
C_7 &= y_4 + \overline{y_2} + z_1 \\
C_8 &= \overline{y_4} + x_2 + y_2 \\
C_9 &= y_5 + \overline{x_3} + z_1 \\
C_{10} &= y_5 + \overline{x_3} + z_1 \\
C_{11} &= \overline{y_6} + x_3 + z_1 \\
C_{12} &= \overline{y_6} + y_4 + z_1 \\
C_{13} &= y_6 + \overline{x_3} + \overline{y_4} \\
C_{14} &= y_7 + \overline{y_4} + z_1 \\
C_{15} &= y_7 + \overline{y_5} + z_1 \\
C_{16} &= \overline{y_7} + y_4 + y_5 \\
C_{17} &= y_8 + \overline{y_6} + z_1 \\
C_{18} &= y_8 + \overline{y_7} + z_1 \\
C_{19} &= \overline{y_8} + y_6 + y_7 \\
C_{20} &= y_8 + z_1 + z_2 \\
C_{21} &= \overline{z_1} + z_3 + z_4 \\
C_{22} &= \overline{z_1} + z_3 + \overline{z_4} \\
C_{23} &= \overline{z_1} + \overline{z_3} + z_4 \\
C_{24} &= \overline{z_1} + \overline{z_3} + \overline{z_4} \\
C_{25} &= \overline{z_2} + z_3 + z_4 \\
C_{26} &= \overline{z_2} + z_3 + \overline{z_4} \\
C_{27} &= \overline{z_2} + \overline{z_3} + z_4 \\
C_{28} &= \overline{z_2} + \overline{z_3} + \overline{z_4}
\end{aligned}
$$

### *1.5.6* DIRHAMCYCLE

Questo problema è definito per un grafo direzionato e non pesato. Dato un grafo $G = (V, E)$ un ciclo $C$ in $G$ è un ciclo *hamiltoniano* se visita ogni vertice esattamente una volta.

**Problema 1.5.18** DIRHAMCYCLE*: Dato un grafo direzionato G, G contiene un ciclo hamiltoniano?*

**Lemma 1.5.19** $3\text{SAT} \leq_P$ DIRHAMCYCLE.

DIMOSTRAZIONE. Consideriamo una qualsiasi istanza di 3SAT. Siano $x_1, \ldots, x_n$ le variabili e $C_1, \ldots, C_k$ le clausole. Il nostro obiettivo è quello di sfruttare un oracolo che risolve DIRHAMCYCLE per risolvere l'istanza di 3SAT. Per fare questo dovremo codificare in qualche modo l'istanza di 3SAT usando un grafo direzionato non pesato.

Poichè sono coinvolte $n$ variabili, ci sono $2^n$ possibili diverse combinazioni dei valori da assegnare come input alle variabili. L'idea è quella di costruire un grafo che contiene $2^n$ diversi cicli hamiltoniani, ognuno dei quali corrisponde a un possibile assegnamento dei valori di input delle variabili nell'istanza di 3SAT. Dopo aver costruito questo grafo inseriremo dei nodi che modellano i vincoli imposti dalle clausole della formula.

Costruiamo $n$ cammini $P_1, \ldots, P_n$, dove $P_i$ è formato dai nodi $v_{i,1}, v_{i,2}, \ldots, v_{i,b}$ per un valore di $b = 3k + 3$. Inseriamo sia gli archi $(v_{i,j}, v_{i,j+1})$ che gli archi nell'altra direzione $(v_{i,j+1}, v_{i,j})$. Quindi ogni cammino $P_i$ può essere percorso sia da "sinistra a destra" da $v_{i,1}$ a $v_{i,b}$ che da "destra a sinistra" da $v_{i,b}$ a $v_{i,1}$.

I cammini sono inoltre uniti dai seguenti archi. Per ogni $i = 1, 2, \ldots, n-1$, inseriamo un arco da $v_{i,1}$ a $v_{i+1,1}$ e un altro arco sempre da $v_{i,1}$ ma questa volta a $v_{i+1,b}$. Cioè ci sono due archi che dal primo nodo del cammino $P_i$ portano rispettivamente al primo e all'ultimo nodo del cammino $P_{i+1}$. In modo simmetrico inseriamo due archi dall'ultimo nodo del cammino $P_i$, al primo e all'ultimo nodo del cammino $P_{i+1}$, cioè gli archi da $(v_{i,b}, v_{i+1,1})$ e $(v_{i,b}, v_{i+1,b})$. Inoltre inseriamo altri due nodi, $s$ e $t$ e gli archi $(s, v_{1,1}), (s, v_{1,r}), (v_{n,1}, t), (v_{n,b}, t)$ ed infine l'arco $(t, s)$. Il grafo risultante è mostrato nella Figura 1.15.

Prima di procedere con la prova, cerchiamo di capire quali sono i cicli hamiltoniani nel grafo che abbiamo costruito. Poichè c'è un solo arco che esce da $t$, sappiamo che un qualsiasi ciclo hamiltoniano dovrà usare tale arco, quindi in qualunque ciclo hamiltoniano dobbiamo passare necessariamente da $t$ ad $s$. Quando siamo in $s$ possiamo procedere o passando nel primo o nell'ultimo nodo di $P_1$. Se passiamo nel primo

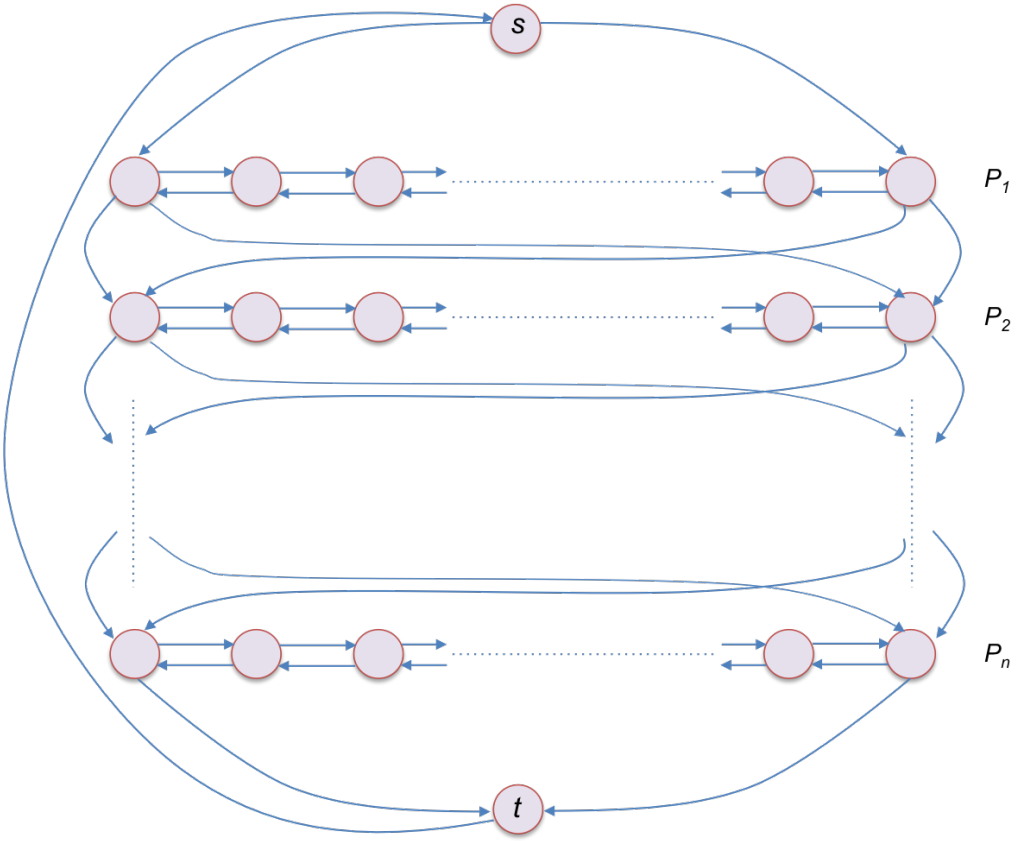



Figura 1.15: Riduzione di 3SAT a DIRHAMCYCLE

attraverseremo $P_1$ da sinistra a destra, se passiamo nell'ultimo attraverseremo $P_1$ da destra a sinistra. Si noti che sebbene dal primo o dall'ultimo nodo di $P_1$ si abbia anche la possibilità di passare ai nodi di $P_2$ senza attraversare $P_1$, se lo facessimo non avremmo modo di visitare gli altri nodi di $P_1$ in quanto per ritornarci dovremmo necessariamente ripassare da $t$ a $s$. Stesso discorso per gli altri cammini, quindi per visitare tutti i nodi dobbiamo visitare i cammini $P_i$, in ordine, cioè prima $P_1$ poi $P_2$ e così via fino a $P_n$. Inoltre ogni cammino potrà essere visitato o da sinistra a destra o da destra a sinistra passando per tutti i nodi in quanto non c'è modo di "saltare" visto che ogni nodo diverso dal primo e dall'ultimo in un cammino è collegato solo al nodo precedente e al successivo.

Pertanto le possibilità di creare cicli diversi fra di loro sono esattamente $n$, una per ogni $P_i$, ed in ogni caso abbiamo 2 scelte, attraversare $P_i$ da destra a sinistra oppure farlo da sinistra a destra. Pertanto ci sono esattamente $2^n$ diversi cicli hamiltoniani nel grafo che abbiamo costruito. Questo ci permette di associare ognuno di questi cicli a una delle $2^n$ diverse possibili combinazioni per l'input di 3SAT: se $P_i$ viene attraversato da sinistra a destra allora $x_i = 1$, se invece attraversiamo $P_i$ da destra a sinistra allora $x_i = 0$.

A questo punto inseriamo dei nodi per modellare la clausole del problema 3SAT. Ogni clausola rappresenta un "vincolo" che di fatto esclude alcune possibili scelte per l'input in quanto queste scelte renderebbero la formula falsa. L'effetto sarà quello di "eliminare" i corrispondenti cicli hamiltoniani dal nostro grafo. Se alla fine un ciclo sopravvive, quel ciclo corrisponde a un assegnamento che rende vera la formula.

Consideriamo ad esempio la clausola

$$C_j = \overline{x_1} + x_3 + x_4.$$

Con la nostra associazione ai cicli hamiltoniani, questa clausola richiede di attraversare $P_1$ da destra a sinistra ($x_1 = 0$ rende vera la clausola), oppure di attraversare $P_3$ da sinistra a destra ($x_2 = 1$ rende vera la clausola), oppure di attraversare $P_3$ da sinistra a destra ($x_3 = 1$ rende vera la clausola). Per "forzare" la direzione voluta inseriamo un nodo $c_j$ extra nei cammini $P_1, P_3$ e $P_4$. Più precisamente, inseriamo $k$ nodi extra, $c_1, c_2, \ldots, c_k$ uno per ogni clausola $C_j$, $j = 1, 2, \ldots, k$, e, per evitare conflitti, useremo i nodi $3j$ e $3j+1$ per i letterali della clausola $j$. Quindi se il letterale è $x_i$, per inserire $c_j$ nel cammino $P_i$ useremo degli archi fra i nodi $(v_{i,3j}, c_j)$ e $(c_j, v_{i,3j+1})$ per forzare l'attraversamento di $P_i$ da sinistra a destra, mentre se il letterale è $\overline{x_i}$ inseriamo gli archi $(v_{i,3j+1}, c_j)$ e $(c_j, v_{i,3j})$ per forzare l'attraversamento di $P_i$ da destra a sinistra. Si noti che questo schema lascia sempre un nodo "di transito" per ogni coppia di nodi $3j$ e $3j+1$, i nodi $3j+2$, sia a sinistra che a destra. Questo nodo è fondamentale per la correttezza della costruzione, come vedremo fra poco.

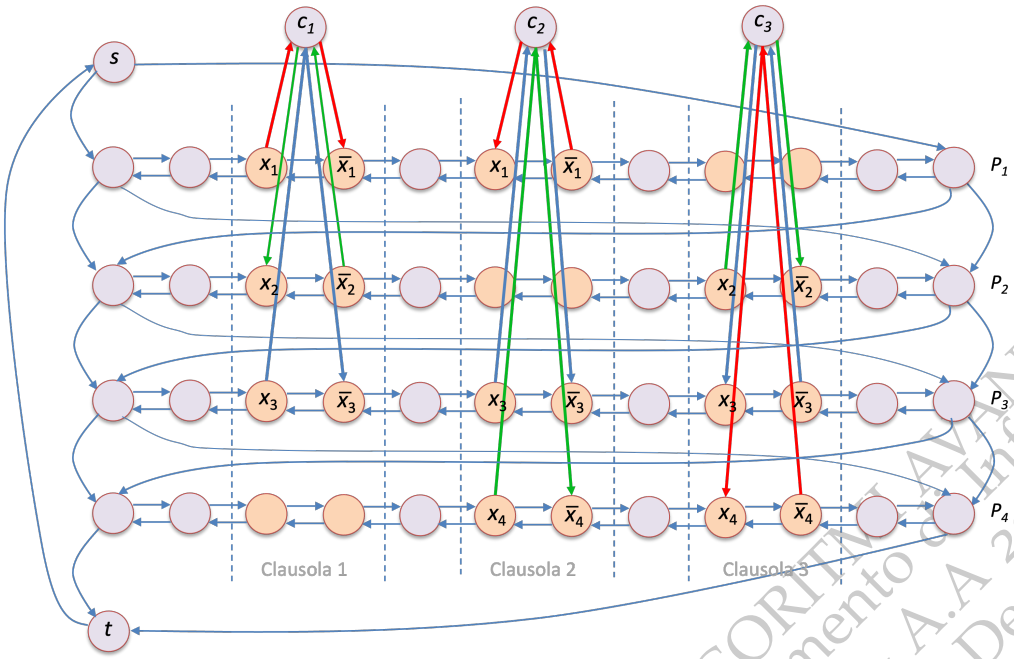



Figura 1.16: Il grafo finale per la formula $\phi_4$

Come esempio, la Figura 1.16 mostra il grafo finale per la formula

$$\phi_4 = (x_1 + \overline{x_2} + x_3) \cdot (\overline{x_1} + x_3 + x_4) \cdot (x_2 + \overline{x_3} + \overline{x_4}).$$

Abbiamo completato la costruzione del grafo e non ci resta che provare che effettivamente l'istanza iniziale di 3SAT è soddisfacibile se e solo se il grafo che abbiamo costruito ha un ciclo hamiltoniano.

Quindi, assumiamo che esista un assegnamento che rende vera l'istanza di 3SAT. Consideriamo il seguente percorso nel nostro grafo: se $x_i$ ha ricevuto il valore 1 attraversiamo $P_i$ da sinistra a destra, altrimenti lo attraversiamo da destra a sinistra. Per ogni clausola $C_j$ riusciremo ad attraversare il nodo $c_j$ facendo le necessarie "deviazioni" dal cammino $P_i$ (tramite i nodi $v_{i,3j}$ e $v_{i,3j+1}$) e seguendo la direzione scelta. Infatti le deviazioni sono state costruite proprio per attraversare i nodi $c_j$ nella direzione indicata dal valore delle variabili. La Figura 1.17 mostra un possibile percorso per l'assegnamento $x_1 = 1, x_2 = 1, x_3 = 1$ e $x_4 = 0$ che rende la formula $\phi_4$ vera. Si noti che i nodi $c_j$ possono essere visitati dalla coppia di nodi che corrispondono a un qualsiasi letterale che rende vera la clausola. Nell'esempio della Figura 1.17, il nodo $c_1$, che è stato visitato da $x_3$ "deviando" il cammino $P_3$, poteva essere visitato anche sfruttando il nodo $x_1$ nel cammino $P_1$. Ovviamente basta (e si deve) visitarlo una volta sola.

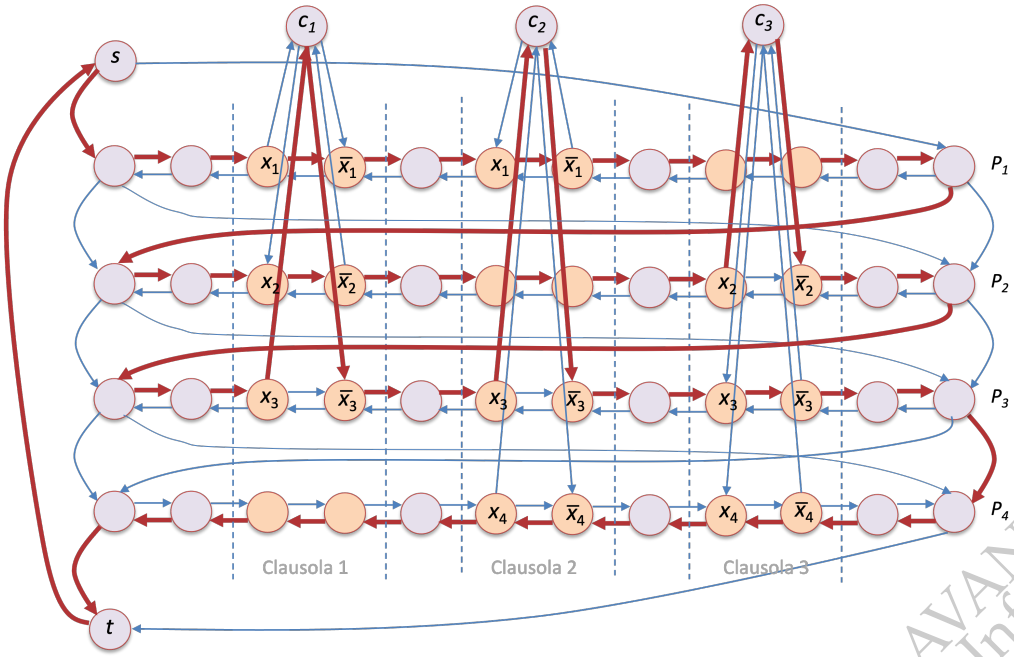



Figura 1.17: Un ciclo hamiltoniano per l'assegnamento $x_1 = 1, x_2 = 1, x_3 = 1$ e $x_4 = 0$.

Viceversa, assumiamo che esista un ciclo hamiltoniano $C$ nel grafo che abbiamo costruito. Osserviamo che se il ciclo $C$ passa per $c_j$ arrivando da $v_{i,3j}$ deve necessariamente lasciare $c_j$ andando a $v_{i,3j+1}$, altrimenti non potrebbe più visitare il nodo di transito $v_{i,3j+2}$. In modo simmetrico, se si visita $c_j$ provenendo da $v_{i,3j+1}$ dovrà lasciarlo

andando a $v_{i,3j}$, altrimenti il nodo di transito $v_{i,3j-1}$ non potrà più essere visitato. In altre parole l'unico modo per visitare i nodi $c_j$ è quello di seguire le deviazioni costruite sui cammini (e per questo il nodo di transito svolge un ruolo importante). Duqnue il nodo di transito svolge un ruolo fondamentale: quando si lascia un cammino per visitare un nodo $c_j$ bisogna necessariamente ritornare su quel cammino altrimenti non si avrebbe più la possibilità di vistare il successivo nodo di transito.

Pertanto $C$ attraversa tutti i cammini da destra a sinistra o viceversa, seguendo le devizioni per visitare anche i nodi $c_j$. Ignorando le deviazioni, quindi, possiamo assegnare dei valori alle variabili $x_i$ in funzione di come il ciclo attraversa i cammini $P_i$: se li attraversa da sinistra a destra assegniamo il valore 1 altrimenti il valore 0. Poichè il ciclo attraversa ovviamente anche i nodi $c_j$, che, ricordiamo, rappresentano il fatto che la clausola è vera, un tale assegnamento rende vere tutte le clausole e quindi anche l'istanza del problema 3SAT. □

Si può considerare il problema anche in un grafo non direzionato: di fatto è un caso particolare visto che un grafo non direzionato può essere rappresentato facilmente da un grafo direzionato semplicemente sostituendo ogni arco non direzionato fra due nodi $u$ e $v$ con gli archi $(u, v)$ e $(v, u)$. Quindi chiaramente UNDHAMCYCLE$\leq_P$DIRHAMCYCLE. Tuttavia UNDHAMCYCLE non è più facile di DIRHAMCYCLE. Si può dimostrare, ad esempio, che VERTEXCOVER$\leq_P$UNDHAMCYCLE e siccome, come vedremo in seguito, VERTEXCOVER$\leq_P$3SAT, per la proprietà transitiva si ha che DIRHAMCYCLE$\leq_P$UNDHAMCYCLE.

### *1.5.7* TSP

Un commesso viaggiatore deve visitare $n$ città, $c_1, \ldots, c_n$. Il commesso risiede nella città $c_1$. Il problema consiste nel trovare una sequenza delle città che permetta al commesso viaggiatore di visitare ogni città esattamente una volta, di ritornare a casa, il tutto viaggiando il meno possibile. Questo problema è simile a quello dei cicli hamiltoniani; in questo caso però ad ogni arco è associato un peso (distanza).

Il problema può essere modellato con un grafo $G = (V, E)$, in cui $V = \{c_1, \ldots, c_n\}$ e $E = \{(c_i, c_j)\}$. Ad ogni arco $(c_i, c_j)$ associeremo la distanza $d(c_i, c_j)$ che serve per raggiungere $c_j$ partendo da $c_i$. Tale distanza potrebbe anche non rappresentare la distanza fisica. Ovviamente fornisce la metrica per misurare ciò che il commesso viaggiatore vuole minimizzare durante il "giro". Pertanto non è nemmeno necessario che sia simmetrica. Possiamo pensare alla "distanza" come a un costo in cui il commesso viaggiatore incorre andando da una città all'altra. L'obiettivo è quello di trovare una sequenza $c_{i_1}, \ldots, c_{i_n}$ con $i_1 = 1$ (cioè si parte dalla città di residenza del commesso viaggiatore) tale che $\sum_j d(c_{i_j}, c_{i_{j+1}}) + d(i_n, i_1)$ sia minima.

Il problema ha molte applicazioni pratiche. La prima, suggerita anche dalla formulazione stessa, è quella della pianificazione di consegne da parte di un corriere che partendo dal punto di smistamento della merce deve effettuare varie tappe per poi ritornare in sede. Anche altri problemi simili nel settore dei trasporti (taxi, camion, flotte navali) possono essere formnulati con TSP per minimizzare il tempo di viaggio e/o i costi del carburante. Oltre a questi casi che sembrano abbastanza naturali, TSP trova applicazioni anche in problemi apparentemente molto diversi, come la produzione

industriale di utensili per la quale una macchina deve lavorare su parti in diverse posizioni e si deve quindi spostare da una all'altra. Più in generale nelle applicazioni che coinvolgono macchine robot è spesso necessario pianificare le varie attività (città) riducendo i tempi/costi di spostamento (metrica). Problemi simili sorgono nei più svariati campi, dalla biologia computazionale alle telcomunicazioni, dalla meteoreologia alla medicina.

La versione decisionale del problema è la seguente:

**Problema 1.5.20** TSP*: Dato un insieme di distanze fra n città, e un limite D sulla distanza, esiste un giro di lunghezza al massimo D?*

**Lemma 1.5.21** DirHamCycle$\leq_P$ TSP*.*

Dimostrazione. Abbiamo a disposizione un oracolo per TSP. Sia $G = (V, E)$ un grafo direzionato con $n$ nodi, per il quale vogliamo sapere se esiste un ciclo hamiltoniano. Definiamo la seguente istanza del problema TSP. Abbiamo una città $c_i$ per ogni nodo $v_i$ del grafo $G$. Inoltre definiamo le distanze in questo modo: $d(c_i, c_j)$ vale 1 se esiste l'arco $(v_i, v_j)$, 2 altrimenti.

Il grafo $G$ ha un ciclo hamiltoniano se e solo se esiste un giro di lunghezza al massimo $n$ per il commesso viaggiatore. Infatti assumiamo che esista un ciclo hamiltoniano in $G$, questo significa che esiste una sequenza di $n$ nodi $v_1, \ldots, v_n$ per i quali esistono gli archi $(v_1, v_2), (v_2, v_3, ) \ldots, (v_{n-1}, v_n)$ ed infine $(v_n, v_1)$. Per come è stato definito l'istanza di TSP, la corrispondente sequenza di città $c_1, c_2, \ldots, c_n$ è un giro di lunghezza $n$ che permette al commesso viaggiatore di partire da $c_1$ e ritornarci dopo aver visitato una e una sola volta tutte le altre città. Viceversa, assumiamo che esista un giro di lunghezza $n$ per il commesso viaggiatore e che tale giro sia composto dalle città $c_1, c_2, \ldots, c_n$. Per come è stato definita l'istanza di TSP, questo significa che nel grafo $G$ esistono gli archi $(v_1, v_2), (v_2, v_3, ) \ldots, (v_{n-1}, v_n)$ ed infine $(v_n, v_1)$. Infatti la distanza fra tutte le coppie di questo giro deve necessariamente essere 1 in quanto la lunghezza totale è $n$ e se ce ne fosse anche solo una coppia a distanza 2 la lunghezza sarebbe maggiore. Pertanto $v_1, \ldots, v_n$ è un ciclo hamiltoniano per $G$. □

Notiamo che nella riduzione abbiamo sfruttato il fatto che le "distanze" fra le città possono essere asimmetriche: questo ci ha permesso di creare un'istanza di TSP che, grazie all'asimmetria delle distanze, ha codificato la presenza o l'assenza di archi nel grafo del problema originario.

Si può considerare il problema del commesso viaggiatore nel caso in cui le distanze siano simmetriche. Questa restrizione sulle distanze non rende il problema più facile: anche in questa versione il problema rimane difficile, e lo si può vedere riducendo UndHamCycle ad esso (la riduzione in questo caso non è semplice).

### *1.5.8* GraphColoring

Il problema della colorazione di grafi prende spunto dal problema della colorazione di una mappa: in una mappa si assegnano dei colori ai vari stati in modo tale che stati adiacenti, cioè che condividono un confine, abbiano un colore diverso. Storicamente il

problema nasce con la seguente domanda: è possibile colorare una qualsiasi mappa con al massimo 4 colori?

Generalizzando il problema a un grafo non direzionato $G$, vogliamo assegnare un colore ad ogni nodo del grafo in modo tale che se esiste l'arco $(u, v)$ allora i colori di $u$ e di $v$ sono diversi. Una mappa è un caso particolare in cui c'è un nodo per ogni stato e gli archi rappresentano le adiacenze. Si noti che una mappa è un caso ristretto in quanto le adiacenze devono rispettare dei vincoli fisici, mentre in generale gli archi possono collegare una coppia di nodi qualsiasi.

Formalmente definiamo una $k$-colorazione del grafo come una funzione $f : V \to \{1, 2, \ldots, k\}$ tale che per ogni arco $(u, v)$ si ha $f(u) \neq f(v)$. In pratica stiamo usando gli interi $1, 2, \ldots, k$, come nomi per i $k$ colori che si possono usare per colorare il grafo. Se esiste una $k$-colorazione per $G$ allora diremo che $G$ è $k$-colorabile.

È abbastanza evidente che l'esistenza di una $k$-colorazione dipende dal grafo. Un grafo senza archi potrebbe essere colorato con un solo colore. All'altro estremo, un grafo completo ha bisogno di un colore diverso per ogni nodo. La questione che ci poniamo è la seguente.

**Problema 1.5.22** GRAPHCOLORING*: Dato un grafo $G$ e un intero $k$, il grafo $G$ è $k$-colorabile?*

Per specificare il parametro $k$, useremo anche la notazione $k$-GRAPHCOLORING.

Sebbene il problema nasca storicamente per la colorazione delle mappe, GRAPHCOLORING ha molteplici applicazioni come, ad esempio, nell'allocazione di risorse in presenza di particolari vincoli, nella progettazione di compilatori, nelle comunicazioni wireless. Si veda il paragrafo 8.7 di [20] per una descrizione di tali applicazioni.

Il caso $k = 2$ è un caso particolare la cui soluzione viene lasciata come esercizio (vedi Esercizio 15). Per $k = 2$ si può dare un algoritmo efficiente che decide se il grafo è 2-colorabile. Consideriamo dunque $k \geq 3$. Osserviamo che, abbastanza intuitivamente, il problema per $k > 3$ è più difficile del caso $k = 3$. In altre parole possiamo provare che:

**Lemma 1.5.23** 3-GRAPHCOLORING$\leq_P k$-GRAPHCOLORING, $k \geq 4$.

DIMOSTRAZIONE. Consideriamo una istanza di 3-GRAPHCOLORING e trasformiamola in una istanza di $k$-GRAPHCOLORING, $k \geq 4$, semplicemente inserendo $k - 3$ nodi fittizi, ognuno collegato a tutti gli altri nodi. Questi 3 nodi fittizi richiedono $k - 3$ colori diversi fra di loro e diversi dai colori di tutti gli altri nodi. In pratica per i nodi rimanenti dobbiamo risolvere il problema con i restanti 3 colori. Quindi il grafo ottenuto è $k$-colorabile se e solo se il grafo di partenza è 3-colorabile. Pertanto sapendo risolvere GRAPHCOLORING possiamo risolvere anche 3-GRAPHCOLORING. □

Consideriamo adesso il caso $k = 3$. Nel passaggo da $k = 2$ a $k = 3$, abbastanza sorprendentemente, il problema diventa molto più difficile. Addirittura si ha che:

**Lemma 1.5.24** 3SAT$\leq_P$3-GRAPHCOLORING.

DIMOSTRAZIONE. Partiamo da un'istanza di 3SAT, quindi una formula booleana fatta dalla congiunzione (AND) di $k$ clausole $C_1, \ldots, C_k$, ognuna con 3 letterali delle variabili $x_1, \ldots, x_n$. Vogliamo risolvere il problema sfruttando un oracolo per 3-GRAPHCOLORING.

Useremo un nodo per ogni variabile $x_i$ in forma vera e un nodo per ogni varibiale in forma negata $\overline{x_i}$ 2 nodi speciali, $T, F$ che rappresentano rispettivamente i valori *true* e *false*, ed infine un nodo speciale $B$, che chiameremo nodo di base.

L'idea è quella di codificare l'impossibilità di rendere entrambi veri due letterali rendendo impossibile l'assegnamento dello stesso colore ai nodi che rappresentano i due letterali. Il nodo di base servirà per legare insieme i vari vincoli.

Pertanto iniziamo con il creare un arco fra ogni coppia di nodi $x_i$ e $\overline{x_i}$, e uniamo entrambi con un arco al nodo di base. Facciamo lo stesso per i nodi $T$ e $F$. Quindi ogni tripla $(x_i, \overline{x_i}, B)$ forma un triangolo, come pure la tripla $(T, F, B)$. La Figura 1.18 mostra il grafo per 3 variabili.

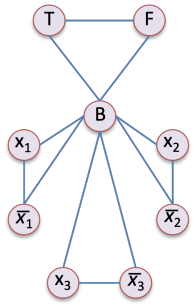



Figura 1.18: Costruzione del grafo

Notiamo che per la presenza di questi archi è necessario che in ognuna di queste triple i nodi ricevano colori diversi. Pertanto data una 3-colorazione del grafo che stiamo costruendo potremo far riferimento al "colore di $T$", al "colore di $F$" e al "colore di $B$". Inoltre poichè $B$ è connesso a tutti i nodi $x_i$ e $\overline{x_i}$, ogni nodo $x_i$ o $\overline{x_i}$ riceverà o il colore di $T$ o il colore di $F$.

Questo grafo di base fornisce una corrispondenza 1 a 1 fra gli assegnamenti di valori booleani alle variabili $x_1, \ldots, x_n$ ed i colori dei nodi che rappresentano le variabili. Ora dobbiamo estendere questo grafo di base per codificare i vincoli dovuti alle clausole in modo tale che solo gli assegnamenti che rendono vera la formula corrispondano a delle 3-colorazioni del grafo.

Quindi per ogni clausola $C_i$ introdurremo dei nodi fittizi e dei nuovi archi per escludere le colorazioni che corrispondono ad assegnamenti che rendono falsa la formula. Procediamo con un esempio e consideriamo la clausola $C_i = x_1 \vee \overline{x_2} \vee x_3$. Con la trasformazione usata per la costruzione del grafo $G$ tale clausola si traduce in: almeno uno dei nodi $x_1, \overline{x_2}$ e $x_3$ deve ricevere il colore di $T$. Per "forzare" questa proprietà dobbiamo trovare un sottografo che inserito in $G$ faccia in modo da eliminare le 3-colorazioni che non soddisfano questa proprietà. Trovare tale sottografo richiede un po' di sforzo: uno che funziona è mostrato nella Figura 1.19

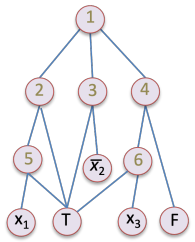



Figura 1.19: Costruzione del grafo, secondo widget

I 6 nuovi nodi si uniscono al grafo di base usando 5 nodi fra quelli già esistenti: i nodi $T, F$ ed i nodi della clausola, cioè $x_1, \overline{x_2}$ e $x_3$. Per provare che il sottografo impone la proprietà richiesta, assumiamo per assurdo che tutti e tre i nodi della clausola ricevano il colore di $F$. Poichè $x_1$ e $x_3$ hanno il colore di $F$ i nodi etichettati con 5 e 6, che sono collegati anche a $T$, dovranno necessariamente ricevere il colore di $B$. Consideriamo ora i 3 nodi etichettati 2, 3 e 4. Il nodo 2 è collegato a un nodo che riceve il colore di $B$ ed al nodo $T$, quindi dovrà essere colorato con il colore di $F$. Il nodo 3 è collegato a $T$ ed a $\overline{v_2}$ e quindi deve ricevere il colore di $B$. Infine il nodo 4 è collegato a un nodo che riceve il colore di $B$ ed al nodo $F$ quindi deve ricevere il colore di $T$. Riassumendo, i nodi 2, 3 e 4 devono ricevere, ripsettivamente, i colori di $F$, di $B$ e di $T$. Tutti questi nodi sono collegati al nodo 1 e per questo nodo non ci sono più colori disponibili. Quindi, nel grafo appena costruito, non è possibile che $x_1, \overline{x_2}$ e $x_3$ ricevano tutti il colore di $F$.

Si noti la tecnica utilizzata per la costruzione del sottografo: si parte dal nodo 1 e lo si collega ai nodi 2, 3, 4 con l'intento di "forzare" su questi 3 nodi tutti e tre i colori in modo da rendere impossibile la colorazione. Poi si collegano i 3 nodi o direttamente ai letterali oppure ad altri nodi fittizi creati per forzare i colori necessari a creare la

contraddizione. Inserendo opportunamente nodi ed archi è possibile forzare qualsiasi colorazione.

Non ci rimane da osservare che basta che uno solo fra $x_1, \overline{x_2}$ e $x_3$ riceva il colore di $T$ per rendere possibile la 3-colorazione del grafo (questa parte della dimostrazione è lasciata come esercizio: partire da tutte le possibile colorazioni in cui almeno uno dei letterali ha il colore di $T$ e trovare una colorazione del sottografo).

Per completare la costruzione del grafo finale, procediamo all'inserimento di un sottografo di 6 nodi simile a quello mostrato nella Figura 1.19 per ogni clausola; si noti che non c'è nessuna relazione fra il fatto che il letterale sia negato ed il fatto che questo è l'unico letterale che non richiede un nodo fittizio al terzo livello. Al posto di $x_1, \overline{x_2}$ e $x_3$ possiamo mettere qualunque combinazione, come ad esempio $x_1, x_2, x_3$; la struttura del sottografo da inserire è sempre lo stessa, cambiano solo i letterali in funzione della clausola.

Chiamiamo $G'$ il grafo finale. La figura 1.20 mostra tale grafo per la formula $(x_1 + \overline{x_2} + x_3) + (\overline{x_1} + \overline{x_2} + x_3) + (x_1 + \overline{x_2} + \overline{x_3})$.

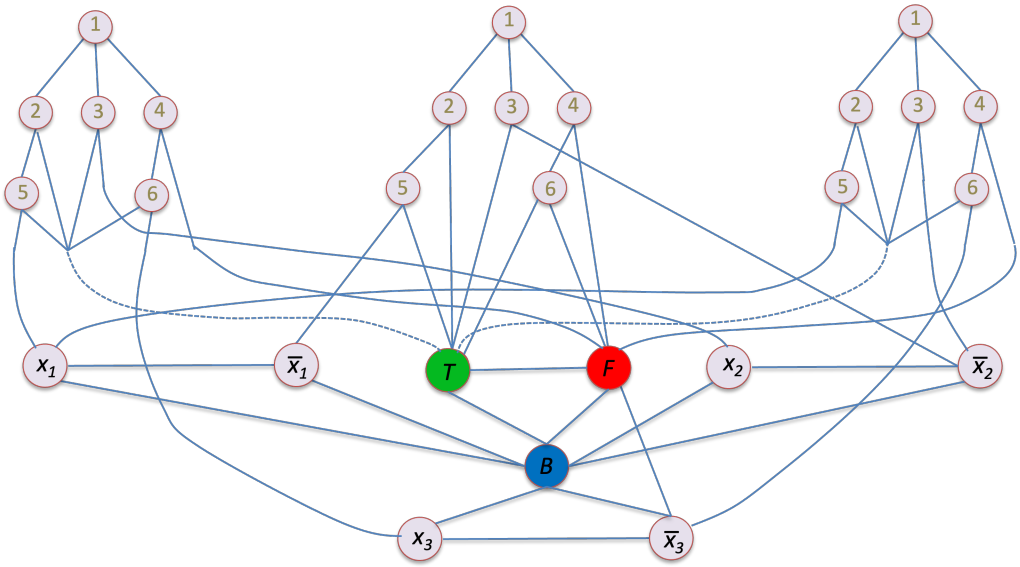



Figura 1.20: Grafo $G'$ per la formula $(x_1 + \overline{x_2} + x_3) + (\overline{x_1} + \overline{x_2} + x_3) + (x_1 + \overline{x_2} + \overline{x_3})$

L'istanza iniziale di 3SAT è soddisfacibile se e solo se $G'$ ammette una 3-colorazione. Per provarlo, assumiamo prima che esista una assegnazione di valori alle variabili $x_1, \ldots, x_n$ che renda vera la formula. Possiamo costruire una 3-colorazione di $G$ semplicemente colorando prima i 3 nodi speciali $T, F$ e $B$ e poi assegnando a $v_i$ il colore di $T$ se $x_i = 1$ ed il colore di $F$ se $x_1 = 0$. Per $\overline{v_i}$ il colore a quel punto sarà l'unico rimasto (quello di $F$ se $x_i = 0$ e quello di $T$ se $x_1 = 1$). Infine la colorazione si estende ai sottografi aggiunti come descritto nella costruzione del grafo: sappiamo che è possibile in quanto per ogni clausola almeno uno dei letterali ha ricevuto il valore vero.

Viceversa, assumiamo che $G'$ ammetta una 3-colorazione. In tale colorazione ogni nodo $v_i$ riceve o il colore di $T$ o il colore di $F$; nel primo caso poniamo $x_i = 1$ e nel secondo caso poniamo $x_i = 0$. Con questo assegnamento, in ognuna della clausole ci

sarà almeno un letterale vero. Infatti se così non fosse, ci sarebbe una clausola in cui tutti i letterali hanno il valore $F$. Questo tuttavia renderebbe impossibile la 3-colorazione del grafo nel sottografo costruito per quella clausola. Quindi possiamo concludere che ogni clausola è vera e quindi la formula è vera. □

Concludiamo la discussione sulla colorazione dei grafi con una nota: una mappa geografica è sempre colorabile con 4 colori! La prova di questo risultato è stata ottenuta solo nel 1976 da Appel e Haken e fa ricorso a una dimostrazione che utilizza una semplice induzione sul numero di stati e la prova del passo induttivo richiede l'analisi di circa 2000 casi. La prova è stata finalizzata solo grazie all'aiuto di un computer. Il problema di trovare una prova breve, ragionevolmente controllabile senza l'aiuto di un computer, è ancora aperto.

## *1.6 Problemi NP-completi*

Tutti i problemi che abbiamo visto in precedenza appartengono a $\mathcal{NP}$ (per ognuno di essi è facile fornire un verificatore di potenziali soluzioni). Per nessuno di essi è stato trovato un algoritmo efficiente. Trovare un algoritmo efficiente per un singolo problema permetterebbe di classificare quel problema come "facile", ma lascerebbe aperta la questione $\mathcal{P}$-$\mathcal{NP}$. Per poter risolvere la questione $\mathcal{P}$-$\mathcal{NP}$ ci si è posto la seguente domanda: quali sono i problemi di $\mathcal{NP}$ più difficili? La strumento della riduzione che abbiamo introdotto in precedenza è ideale per individuare i problemi più difficili di $\mathcal{NP}$, che sono quei problemi $X \in \mathcal{NP}$ tali che un qualsiasi altro problema $Y \in \mathcal{NP}$ può essere ridotto a $X$:

$$\mathcal{NP}\text{-completi} = \{X \in \mathcal{NP} | \forall Y \in \mathcal{NP}, Y \leq_P X\}.$$

Tali problemi vengono detti $\mathcal{NP}$-completi. La motivazione per l'individuazione di problemi $\mathcal{NP}$-completi è abbastanza ovvia:

**Lemma 1.6.1** *Sia $A$ un problema $\mathcal{NP}$-completo. Allora $A$ è risolvibile in modo efficiente se e solo se $\mathcal{P} = \mathcal{NP}$.*

Dimostrazione. Se $A$ può essere risolto in modo efficiente allora possiamo usarlo per risolvere in modo efficiente un qualunque altro problema in $\mathcal{NP}$. Infatti sia $X$ un qualunque altro problema in $\mathcal{NP}$, poichè $A$ è $\mathcal{NP}$-completo, si ha che $X \leq_P A$ e quindi anche $X$ può essere risolto in tempo polinomiale. Per cui $\mathcal{P} = \mathcal{NP}$.

Viceversa, se $\mathcal{P} = \mathcal{NP}$ allora tutti i problemi in $\mathcal{NP}$ possono essere risolti in tempo polinomiale e quindi anche $A$ può essere risolto in tempo polinomiale. □

Avendo definito la classe $\mathcal{NP}$-completi il prossimo passo è quello di individuare dei problemi appartenenti a tale classe. Sebbene la definizione sia chiara e semplice, non è altrettanto chiaro e immediato il fatto che effettivamente dei problemi $\mathcal{NP}$-completi esistano. Un problema $\mathcal{NP}$-completo deve avere una proprietà estremamente forte: deve poter essere usato per risolvere *qualunque* altro problema in $\mathcal{NP}$. Quindi per dimostrare che un problema $X$ è $\mathcal{NP}$-completo dobbiamo trovare un modo per poter "trasformare"

un qualsiasi altro problema in un'istanza di $X$. In altre parole dobbiamo creare una riduzione come quelle viste precedentemente ma con una piccola grande differenza: mentre negli esempi di riduzione visti finora dovevamo ridurre uno specifico problema ad $X$, adesso dobbiamo ridurre ad $X$ un *qualsiasi* altro problema in $\mathcal{NP}$, cioè dobbiamo ridurre ad $X$ tutti gli altri problemi in $\mathcal{NP}$.

**Teorema 1.6.2** CircuitSat *è $\mathcal{NP}$-completo.*

La prova di questo risultato va al di là degli obiettivi di questo corso. Tuttavia l'idea di base della dimostrazione è quella che un circuito può di fatto "implementare" qualsiasi algoritmo. Astraendo molto dai complessi dettagli formali, la cosa non dovrebbe creare incredulità in quanto un qualsiasi algoritmo viene implementato usando le operazioni logiche di base, AND, OR e NOT, che sono i componenti di base di un circuito.

Una pietra miliare dello studio dei problemi $\mathcal{NP}$-completi è il lavoro di Cook "The Complexity of Theorem-Proving Procedures", pubblicato nel 1971 che prova che Sat è $\mathcal{NP}$-completo. La scoperta dell'esistenza di problemi $\mathcal{NP}$-completi è attribuita sia a Cook, per il lavoro già citato, che a Levin che ha provato indipendentemente da Cook l'esistenza di problemi $\mathcal{NP}$-completi, incluso Sat, in un lavoro del 1973 "Universal Search Problems" (pubblicato in russo, "Universal'nye zadachi perebora"). Spesso si parla di teorema di Cook-Levin per indicare la $\mathcal{NP}$-completezza di Sat. Il problema CircuitSat è chiaramente simile a Sat. Una prova della $\mathcal{NP}$-completezza di CircuitSat può essere trovata nel libro CLRS2009 [8] (nel capitolo che tratta la $\mathcal{NP}$-completezza).

**Lemma 1.6.3** *Sia $A$ un problema $\mathcal{NP}$-completo. Sia $B$ un problema in $\mathcal{NP}$. Se $A \leq_P B$, allora anche $B$ è $\mathcal{NP}$-completo.*

Dimostrazione. Poichè $A$ è $\mathcal{NP}$-completo, un qualsiasi altro problema $X$ può essere ridotto ad $A$, cioè $X \leq_P A$. Poichè $A \leq_P B$, per la proprietà transitiva della riduzione si ha che $X \leq_P B$. Quindi un qualsiasi problema $X \in \mathcal{NP}$ può essere ridotto a $B$, pertanto $B$ è $\mathcal{NP}$-completo. □

Il precedente lemma e il fatto che CircuitSat sia $\mathcal{NP}$-completo, implicano la $\mathcal{NP}$-completezza di tutti i problemi che abbiamo studiato nella sezione 1.5. La seguente figura riassume le riduzioni viste in precedenza.

$$\text{CircuitSat} \leq \text{3Sat} \leq \begin{cases} \text{Sat} \\ \text{GraphColoring} \\ \text{IndependentSet} \leq \begin{cases} \text{VertexCover} \leq \text{SetCover} \\ \text{SetPacking} \end{cases} \\ \text{HamCycle} \leq \text{TSP} \end{cases}$$

Dal fatto che CircuitSat è $\mathcal{NP}$-completo tale catena di riduzioni prova che tutti i problemi visti sono $\mathcal{NP}$-completi (ricordiamo che per ognuno di essi è facile fornire un verificatore efficiente, quindi tutti appartengono a $\mathcal{NP}$).

## 1.7 *Problemi decisionali e di ricerca*

Nella sezione 1.2 abbiamo discusso brevemente della relazione fra la versione decisionale di un problema e quella di ottimizzazione. Ricordiamo che si parla di problema *decisionale* quando la risposta che cerchiamo è un sì o un no alla domanda "esiste una soluzione?" e di problema di *ricerca* quando cerchiamo una soluzione al problema; nel caso dei problemi di ottimizzazione cerchiamo una soluzione ottima. In precedenza abbiamo detto che il limitarsi a considerare problemi decisionali non è una grossa restrizione: stiamo studiando la difficoltà dei problemi quindi se un problema è difficile nella versione decisionale a maggior ragione lo sarà nella versione di ricerca.

In questa sezione approfondiamo un po' di più la relazione fra problemi decisionali e problemi di ricerca. Abbiamo già osservato che se sappiamo risolvere in maniera efficiente la versione decisionale di un problema, sappiamo anche dire, sempre in modo efficiente, quale è il valore di una soluzione ottima al corrispondente problema di ottimizzazione: basta fare una ricerca binaria sul valore della soluzione ottima e con un numero logaritmico di ripetizioni dell'algorimo scopriamo il valore della soluzione ottima.

Non è stato provato che la versione decisionale è sempre equivalente alla corrispondente versione di ricerca/ottimizzazione, ma per la maggior parte dei problemi per i quali si conosce un algoritmo efficiente si è capaci sia di risolvere in modo efficiente la versione decisionale che il corrisponde problema di ricerca/ottimizzazione.

Un caso in cui questo non è vero è il test di primalità. Nella versione decisionale occorre stabilire se un numero $n$ è primo oppure no, cioè se è il prodotto di fattori. La versione di ricerca di questo problema è quella della ricerca dei fattori di $n$; in questo caso non c'è nulla da ottimizzare! Si è a lungo creduto che il test di primalità non appartenesse alla classe $\mathcal{P}$. Nel 2002 però è stato dimostrato, da Agrawal, Kayal e Saxena [1], che è possibile effettuare il test di primalità con un algoritmo deterministico polinomiale. Per il corrispondente problema di ricerca, cioè quello di trovare i fattori di $n$, invece, come abbiamo detto anche nell'introduzione di questo capitolo, non si conosce un algoritmo efficiente. Ovviamente se si dovesse dimostrare che $\mathcal{P} = \mathcal{NP}$ anche il problema di trovare i fattori di un numero verrebbe risolto in tempo polinomiale.

Dunque per molti problemi sappiamo risolvere in maniera efficiente sia la versione decisionale che quella di ricerca. Ci sono dei casi in cui sappiamo risolvere in maniera efficiente la versione decisionale ma non sappiamo se la versione di ricerca può essere risolta in maniera efficiente. La cosa è legata alla questione $\mathcal{P}$-$\mathcal{NP}$.

Infine osserviamo che per i problemi $\mathcal{NP}$-completi la versione decisionale è equivalente alla versione di ricerca. Infatti, la versione decisionale è automaticamente risolta dalla versione di ricerca, mentre la ricerca di una soluzione si può spesso fare usando come subroutine la versione decisionale sfruttando tecniche come ricerca binaria o tentativi incrementali.

Come esempio consideriamo il problema SAT. Sia $A$ un algoritmo che risolve la versione decisionale in tempo $O(f(n))$, dove $n$ è la grandezza del problema. L'algoritmo $A$ ci dice se esiste un assegnamento delle variabili che soddisfi la formula di input $\phi = (x_1, x_2, \ldots, x_n)$. Possiamo fissare un valore per $x_1$ ($x_1 = 0$ oppure $x_1 = 1$) e ottenere

una nuova istanza di SAT e usando l'oracolo possiamo sapere se esiste un assegnamento che rende vera la formula originale nei due casi considerati.

La cosa si può iterare fissando valori successivi fino a fissarli tutti. Scriveremo $A()$ per indicare l'utilizzo di $A$ nella formula originaria e $A(0)$ o $A(1)$ o per indicare l'utilizzo di $A$ su $\phi$ con $x_1$ fissato al valore specificato. Ovviamente potremo usare $A$ fissando anche altri valori. Ad esempio $A(1,0,1)$ indica l'utilizzo di $A$ con $x_1 = 1$, $x_2 = 0$ e $x_3 = 1$ in $\phi$.

Possiamo sfruttare $A$ per costruire un algoritmo TROVASOLUZIONE che trova una soluzione in tempo $O(n \cdot f(n))$. L'algoritmo TROVASOLUZIONE "trova" una soluzione fissando i valori delle singole variabili e "chiedendo" ad $A$ se le scelte portano a un assegnamento che rende vera la formula. Ad esempio possiamo fissare $x_1 = 1$ e usare $A(1)$ per capire se esiste un assegnamento che renda vera $\phi$ con $x_1 = 1$. Se non esiste possiamo provare con $x_1 = 0$. Se per nessuno dei due casi esiste, allora la formula non è soddisfacibile. Possiamo ripetere il test per ogni variabile, riuscendo così a trovare una soluzione (se esiste) eseguendo per un numero polinomiale (lineare) di volte l'algoritmo decisionale.

**Algorithm 1:** Algoritmo TROVASOLUZIONE

**for** $i = 1, 2, \ldots, n$ **do**
  $s_i =$`null`
**if** $A(1)$ **then**
  $s_1 = 1$
**else**
  **if** $A(0)$ **then** $s_1 = 0$ **else return** `null`
**if** $A(s_1, 1)$ **then**
  $s_2 = 1$
**else**
  **if** $A(s_1, 0)$ **then** $s_2 = 0$ **else return** `null`
...
**if** $A(s_1, \ldots, s_{n-1}, 1)$ **then**
  $s_n = 1$
**else**
  **if** $A(s_1, \ldots, s_{n-1}, 0)$ **then** $s_n = 0$ **else return** `null`
**return**$(s_1, \ldots, s_n)$

L'esempio di SAT è particolarmente semplice in quanto fissando il valore di una variabile otteniamo una nuova istanza dello stesso problema SAT; il problema, cioè, è *auto-riducibile*. E ovviamente questa strategia è possibile per tutti i problemi auto-riducibili. Non tutti i problemi, però, hanno questa caratterestica.

Possiamo sfruttare SAT anche per ridurre la versione di ricerca alla versione decisionale di un qualsiasi altro problema $X \in \mathcal{NP}$ a patto che la riduzione $X \leq_P$SAT, che esiste in quanto SAT è $\mathcal{NP}$-completo, preservi il certificato, cioè a patto che ci sia un modo per risalire ad una soluzione di $X$ da una soluzione di SAT. In questo caso l'algoritmo TROVASOLUZIONE ci permetterà di trovare una soluzione anche per $X$, a partire dalla soluzione per SAT.

Questa cosa è possibile per tutti i problemi $\mathcal{NP}$-completi.

Non daremo una prova formale ma solo l'intuizione. L'argomentazione si basa sulla

rappresentazione tramite linguaggi. In particolare dovremo definire il linguaggio che corrisponde alle soluzioni del problema e trattare tale linguaggio come un problema.

Il linguaggio associato al problema $X$ è

$$L_X = \{x|\ x \text{ è un'istanza che ammette soluzioni}\}.$$

Assumiamo di avere un oracolo $\mathcal{O}_X$ per il problema $X$. Tale oracolo, preso in input una stringa $x$ che rappresenta un'istanza di $X$ risponde con `1` se $x$ ammette una soluzione e con `0` altrimenti. Questo non ci dice nulla su una soluzione di $x$. Consideriamo, fissata l'istanza $x$, il linguaggio

$$L_x = \{w|\ w \text{ è una soluzione dell'istanza } x\}.$$

L'oracolo $\mathcal{O}_X$ ci dice se $L_x$ è vuoto oppure no.

Poichè ogni linguaggio è un problema decisionale, un qualsiasi insieme di stringhe $L_{str}$ è un problema decisionale e come tale può essere ridotto a un problema $\mathcal{NP}$-completo. Ma $X$ è un problema $\mathcal{NP}$-completo, quindi $L_{str} \leq_P X$. Questo significa che dato $L_{str}$ possiamo sapere se esso è vuoto oppure no riducendo $L_{str}$ a $X$ per poi usare $\mathcal{O}_X$. La riduzione da $L_{str}$ a $X$ individua una istanza $x_{str}$ di $X$ tale che $x_{str}$ ammette soluzioni se e solo se $L_{str}$ non è vuoto. Indicheremo con $\mathcal{O}(L_{str})$ l'uso dell'oracolo che decide $L_{str}$.

A questo punto possiamo trovare un elemento $w \in L_x$ con un approccio molto simile a quello usato per SAT. L'idea è quella di trovare un elemento di $L$, "scoprendo" un bit alla volta, più o meno come per SAT abbiamo trovato il valore di una variabile alla volta.

Per trovare un $w \in L_x$ possiamo definire una serie di altri linguaggi

$$\begin{aligned}
L_0 &= \{0y|w = 0y \in L_x\},\\
L_1 &= \{1y|w = 1y \in L_x\},\\
L_{00} &= \{00y|w = 00y \in L_x\},\\
L_{01} &= \{01y|w = 01y \in L_x\},\\
L_{10} &= 10y|w = 10y \in L_x,\\
L_{11} &= 11y|w = 11y \in L_x,
\end{aligned}$$

etc. Ognuno di questi si riduce a $X$.

Per trovare il primo bit posso usare $\mathcal{O}_X(L_0)$ e $\mathcal{O}_X(L_1)$. Se entrambe le risposte sono `0`, allora non esiste nessuna soluzione, cioè $L_x$ è vuoto. Se almeno una delle due riposte è `1`, allora $L_x$ non è vuoto. Se $\mathcal{O}_X(L_0) = 1$ allora esiste $w \in L_x$ con $w = 0y \in L_x$, cioè un stringa $w$ che iniza per 0; in questo caso possiamo proseguire la ricerca fissando il primo bit a o. Se $\mathcal{O}_X(L_1) = 1$ allora esiste $w \in L_x$ con $w = 1y \in L_x$, cioè un stringa $w$ che iniza per 1; in questo caso possiamo proseguire la ricerca fissando il primo bit a `1`. Se sono `1` entrambe esistono stringhe $w \in L_x$ che iniziano per 0 e stringhe che iniziano per 1; in questo caso possiamo procedere la ricerca con uno qualsiasi dei due valori per il primo bit.

Se esiste una stringa di $L_x$ che inizia per 0, possiamo scoprire il secondo bit usando $\mathcal{O}_X(L_{00})$ e $\mathcal{O}_X(L_{01})$. Se esiste una stringa di $L_x$ che inizia per 1, possiamo scoprire il secondo bit usando $\mathcal{O}_X(L_{10})$ e $\mathcal{O}_X(L_{11})$. Chiaramente basterà una delle due possibilità Proseguendo in modo simile scopriamo un bit alla volta fino ad arrivare a una soluzione dell'istanza originale $x$ del problema $X$.

## 1.8 Pseudopolinomialità

Consideriamo il problema della somma di un sottoinsieme, SubsetSum, che è formulato nel seguente modo: dati i numeri naturali $w_1, \ldots, w_n$ e un altro numero naturale $W$, vogliamo selezionare il sottoinsieme $S$ tale che $\sum_{i \in S} w_i \leq W$ e, tenendo presente questo vincolo, tale che $\sum_{i \in S} w_i$ sia quanto più grande possibile. Questo problema, nella sua formulazione più generale è conosciuto come *problema dello zaino*. Nel problema dello zaino ogni $w_i$ è il *peso* di un oggetto $i$ che ha un valore $v_i$ mentre $W$ è il peso massimo che lo zaino può sopportare. L'obiettivo è mettere nello zaino un sottoinsieme di valore massimo con il vincolo di non superare il peso massimo. Nel problema SubsetSum si ha che $v_i = w_i$.

Ritornando al problema della somma di un sottoinsieme serve considerare una versione decisionale. Per poter fare ciò cambiamo leggermente il vincolo sul numero $W$ e richiediamo che la somma dei numeri del sottoinsieme debba essere esattamente $W$. Questo perché vogliamo formulare il problema con una domanda del tipo "esiste" un sottoinsieme tale che ...; se il vincolo fosse $\leq$ la risposta sarebbe sempre "sì" (il sottoinsieme vuoto soddisferebbe sempre il vincolo).

**Problema 1.8.1** SubsetSum*: Dati i numeri naturali $w_1, \ldots, w_n$ e un altro numero $W$, esiste un sottonsieme di $\{w_1, \ldots, w_n\}$ la cui somma è esattamente $W$?*

Per trovare il valore di $W$ più grande possibile possiamo usare una ricerca binaria. Il problema SubsetSum può essere risolto con un algoritmo di programmazione dinamica che trova una soluzione in tempo $O(nW)$. Ricordiamo velocemente tale algoritmo.

Indichiamo con $S(i, j)$ la soluzione al sottoproblema $\{w_1, \ldots, w_i\}$ e somma pari a $j$, per $i = 1, 2, \ldots, n$ e $j = 0, 1, 2, \ldots, W$; $S(i, j) = \texttt{1}$ quando la soluzione è `si` e $S(i, j) = \texttt{0}$ quando la soluzione è `no`. Il valore $S(n, W)$ è la soluzione all'istanza di input. Con la programmazione dinamica dobbiamo riempire la matrice bidimensionale $S(i, j)$.

Per $j = 0$ chiaramente il problema è banalmente risolvibile in quanto l'insieme vuoto è una soluzione. Quindi si ha che $S(i, 0) = \texttt{1}$ per tutti i valori di $i = 1, 2, \ldots, n$. Per la prima riga $S(1, j) = \texttt{1}$ solo per $j = 0$ e $j = w_1$, mentre per tutte gli altri elementi il valore è 0. Per riempire il resto della matrice possiamo sfruttare la seguente osservazione. Data una soluzione al problema per $i - 1$ possiamo trovare una soluzione per $i$. Infatti, avendo a disposizione anche $w_i$, oltre a $w_1, \ldots, w_{i-1}$, possiamo o usare o non usare $w_i$. Se non lo usiamo, allora l'unica possibilità è quella di sfruttare $S(i - 1, j)$, mentre se lo usiamo dobbiamo sfruttare $S(i - 1, j - w_i)$ insieme a $w_i$. pertanto la relazione di ricorrenza che permette di calcolare i valori della matrice $S$, per le riche successive alla prima, è

$$S(i, j) = \begin{cases} S(i - 1, j), \text{ se } w_i < j \\ S(i - 1, j) \texttt{ or } S(i - 1, j - w_i) \text{ se } w_i \geq j. \end{cases}$$

Consideriamo un esempio: $n = 5$, $w_1 = 2, w_2 = 3, w_3 = 4, w_4 = 7, w_5 = 10$ e $W = 15$. La matrice $S$ è la seguente:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 0 | 1+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 1- | 1+ | 0 | 1+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 0 | 1- | 1- | 1+ | 1- | 1+ | 1+ | 0 | 1+ | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 1 | 0 | 1- | 1- | 1- | 1- | 1- | 1* | 0 | 1* | 1+ | 1+ | 1+ | 1+ | 1+ | 0 |
| 10 | 1 | 0 | 1- | 1- | 1- | 1- | 1- | 1- | 0 | 1- | 1* | 1- | 1* | 1* | 1* | 1+ |

dove 1- indica che il corrispondente $w_i$ non è stato usato nella soluzione, mentre 1+ indica che $w_i$ è stato usato; 1* indica che sono possibili entrambe le soluzioni, con e senza $w_i$.

È facile verificare che l'algoritmo può essere implementato in $O(nW)$ in quanto bisogna solo riempire la matrice $S$. Perchè quindi lo stiamo prendendo in considerazione ora che parliamo di problemi difficili da risolvere? Il tempo di esecuzione dell'algoritmo di programmazione dinamica, $O(nW)$, non è polinomiale nel senso più stretto del termine. La difficoltà è dovuta alla presenza di $W$ nel tempo di esecuzione. Se $W$ è piccolo allora non è un problema, ma se è grande lo diventa. Ad esempio, assumendo di avere a che fare con $n = 100$ numeri rappresentabili con 100 bit ognuno, la lunghezza dell'input è di 10.100 bit (i 100 numeri più $W$), ma il valore di $W$ può essere dell'ordine di $2^{100}$, e quindi l'algoritmo impiegherebbe un tempo lunghissimo. Questo perchè il valore di $W$ è di fatto *esponenziale* nella grandezza dell'input (il numero di bit necessari a rappresentarlo). In queste situazioni si parla di tempo *pseudopolinomiale*.

Quindi la questione che vogliamo affrontare è: possiamo risolvere SubsetSum in tempo (puramente) polinomiale? In altre parole, esiste un algoritmo con tempo polinomiale in $n$ e $\log W$, cioè polinomiale in $n$? La risposta è no[8]! Non vedremo la prova, che ovviamente si basa su una riduzione di un problema $\mathcal{NP}$-completo a SubsetSum, ma si ha che:

[8] A meno che $\mathcal{P}$ non sia uguale a $\mathcal{NP}$.

**Lemma 1.8.2** SubsetSum *è $\mathcal{NP}$-completo.*

Vale la pena rimarcare ancora una volta che il problema SubsetSum è "difficile" in quanto $W$ può assumere valori esponenziali in $n$. Nei casi in cui $W$ è polinomiale in $n$, il problema diventa risolvibile in tempo polinomiale in quanto $O(nW)$ diventa polinomiale in $n$.

## 1.9 Co-NP e l'asimmetria di NP

Dato un problema $X$, definiamo il problema complementare $\overline{X}$ nel seguente modo: data una stringa di input $s$, $s \in \overline{X}$ se e solo se $s \notin X$. La classe Co-$\mathcal{NP}$ è definita come segue.

**Definizione 1.9.1** *Co-$\mathcal{NP}$ è l'insieme dei problemi $X$ tali che $\overline{X} \in \mathcal{NP}$.*

Riflettendo sulla definizione di verificatore che abbiamo usato per la classe $\mathcal{NP}$, volendola applicare a $\overline{X}$, il verificatore *di esistenza* di una soluzione diventerebbe il verificatore della *non* esistenza di una soluzione. Per capire meglio questa affermazione consideriamo uno specifico problema, ad esempio $X =$ DirHamCycle. Abbiamo che

$$X = \{1010011..., 0100110..., \ldots\}$$

dove ognuna delle stringhe è una codifica di un grafo di input che ammette un ciclo hamiltoniano, e, per definizione si ha che

$$\overline{X} = \{\text{tutte le altre stringhe}\}.$$

Una qualsiasi stringa di $\overline{X}$ è la codifica di un grafo che non ammette un ciclo hamiltoniano (o una stringa che non ha significato, che comunque è un qualcosa che non ammette un ciclo hamiltoniano). Sia $s$ una stringa binaria che rappresenta un grafo di input. Come si fa a stabilire se $s \in X$ oppure se $s \in \overline{X}$? Se $s \in X$ basta fornire un ciclo hamiltoniano per $s$, cioè una soluzione. In questo senso il certificato che serve a convincere il verificatore è una soluzione che il verificatore deve solo controllare. Se invece $s \in \overline{X}$ (e quindi non ci sono soluzioni) fornire un "non-ciclo" non convince sul fatto che non esistano cicli hamiltoniani per $s$. (Se la stringa $s$ non è la codifica di un grafo allora banalmente la stringa appartiene a $\overline{X}$ senza bisogno di nessuna certificazione; ma il verificatore deve operare su qualsiasi stringa, e le strighe di interesse sono quelle che rappresentano grafi.)

Dunque, con la classe Co-$\mathcal{NP}$ stiamo individuando quei problemi per i quali è facile dire che non esiste una soluzione. Osserviamo subito che per provare che esiste (almeno) una soluzione a un problema è sufficiente fornirne una, mentre provare che un problema non ammette soluzioni è molto più difficile in quanto una singola *non*-soluzione non potrà servire come "prova". Un modo per provare che non esiste nessuna soluzione è analizzarle tutte, ma questo ovviamente può richiedere tempo esponenziale e quindi non è efficiente, mentre il verificatore deve essere efficiente. Si noti come questo crei un'asimmetria fra queste classi.

Se un problema $X$ appartiene a $\mathcal{P}$, allora anche $\overline{X} \in \mathcal{P}$. Infatti avendo a disposizione un algoritmo polinomiale che risolve $X$, lo stesso algoritmo permette di risolvere $\overline{X}$. Ricordando che stiamo trattando problemi decisionali, la cui soluzione è un `sì` oppure un `no`, per risolvere $\overline{X}$ basterà invertire la risposta ottenuta risolvendo $X$. Quindi anche $\overline{X} \in \mathcal{P}$. Se definissimo Co-$\mathcal{P}$ in modo simile, in pratica avremmo che $\mathcal{P}$=Co-$\mathcal{P}$.

Lo stesso ragionamento non vale per $X \in \mathcal{NP}$. Infatti in questo caso abbiamo a disposizione un algoritmo che verifica una soluzione di $X$ in tempo polinomiale. Questo non ci dice niente su $\overline{X}$, almeno non in tempo polinomiale. Quindi la seguente domanda sorge spontanea:

**Problema 1.9.2** *$\mathcal{NP}$= Co-$\mathcal{NP}$?*

Lo studio di Co-$\mathcal{NP}$ è strettamente legato alla questione $\mathcal{P}$-$\mathcal{NP}$, infatti si ha che:

**Lemma 1.9.3** *Se $\mathcal{NP} \neq$ Co-$\mathcal{NP}$, allora $\mathcal{P} \neq \mathcal{NP}$.*

DIMOSTRAZIONE. Assumiamo che $\mathcal{NP} \neq$ Co-$\mathcal{NP}$. Procediamo per assurdo assumendo che $\mathcal{P} = \mathcal{NP}$. Allora si avrebbe che

$$X \in \mathcal{NP} \implies X \in \mathcal{P} \implies \overline{X} \in \mathcal{P} \implies \overline{X} \in \mathcal{NP} \implies X \in \text{Co-}\mathcal{NP}$$

ma anche che

$$X \in \text{Co-}\mathcal{NP} \implies \overline{X} \in \mathcal{NP} \implies \overline{X} \in \mathcal{P} \implies X \in \mathcal{P} \implies X \in \mathcal{NP}.$$

Quindi si avrebbe che $\mathcal{NP} \subseteq$Co-$\mathcal{NP}$ e Co-$\mathcal{NP} \subseteq \mathcal{NP}$ e quindi che $\mathcal{NP}=$ Co-$\mathcal{NP}$. Questo è un assurdo perchè contraddice l'ipotesi di partenza. □

Poichè la definizione di Co-$\mathcal{NP}$ è asimmetrica (è facile certificare l'esistenza di una soluzione, basta verificarla, ma non è altrettanto facile certificare che non esiste una soluzione) è interessante chiedersi se esistano problemi che effettivamente appartengono a $\mathcal{NP}\cap$Co-$\mathcal{NP}$.

Determinare se una rete ammette un flusso di valore almeno $v$ è uno di questi. Il problema appartiene a $\mathcal{NP}$ in quanto è facile fornire un certificato che provi che esiste una soluzione: il certificato è un flusso con valore almeno $v$. In questo caso è possibile anche fornire un certificato, verificabile in tempo polinomiale, che mostra che non esiste un tale flusso: il certificato è un taglio di capacità più piccola di $v$; per il teorema del massimo flusso e minimo taglio, si ha che non può esistere un flusso di valore più grande. Quindi detto $X$ il problema, si ha che $X \in \mathcal{NP}$, ma anche $X \in$Co-$\mathcal{NP}$ in quanto $\overline{X} \in \mathcal{NP}$.

In realtà, qualunque problema $X \in \mathcal{P}$, ha la proprietà che $X \in \mathcal{NP}$ e $X \in$Co-$\mathcal{NP}$. Quindi si ha che

$$\mathcal{P} \subseteq \mathcal{NP} \cap \text{ Co-}\mathcal{NP}.$$

Un'altra questione aperta è

**Problema 1.9.4** *$\mathcal{P} = \mathcal{NP} \cap$ Co-$\mathcal{NP}$?*

## *1.10* PSPACE

La classi $\mathcal{P}$ e $\mathcal{NP}$ classificano i problemi in funzione del tempo necessario a risolverli. Una classificazione simile può essere fatta considerando lo *spazio* (cioè la memoria) necessaria a risolvere un problema. La classe PSPACE è l'analogo di $\mathcal{P}$, ma considerando lo spazio al posto del tempo: PSPACE è l'insieme dei problemi che possono essere risolti usando spazio polinomiale. Si noti come nella definizione non venga menzionato il tempo, quindi per risolvere un problema in PSPACE si può usare quanto tempo si vuole!

Il seguente fatto è immediato.

**Lemma 1.10.1** *$\mathcal{P} \subseteq$ PSPACE.*

Dimostrazione. Sia $X$ un problema in $\mathcal{P}$. Allora esiste un algoritmo che in tempo polinomiale risolve $X$. Lo stesso algoritmo risolve $X$ usando spazio polinomiale: infatti poichè $X$ usa tempo polinomiale può utilizzare al massimo un numero polinomiale di celle di memoria, quindi usa spazio polinomiale. Pertanto $X \in$ PSPACE. □

PSPACE però è più ampio di $\mathcal{P}$. Il punto cruciale è che avendo a disposizione tempo illimitato è possibile riusare la memoria. Ad esempio, possiamo risolvere il problema 3Sat usando spazio polinomiale

**Lemma 1.10.2** *3Sat$\in$ PSPACE.*

DIMOSTRAZIONE. Possiamo usare un approccio a forza bruta, facendo attenzione a riusare la memoria utilizzata in modo tale da non usare mai più di un numero polinomiale di celle di memoria.

In particolare, se la formula è definita su $n$ variabili, useremo $n$ celle di memoria (in realtà bastano esattamente $n$ bit) per specificare una soluzione.

Ogni possibile soluzione può essere rappresentata come una sequenza binaria di $n$ bit, dove l'$i$-esimo bit rappresenta il valore della variabile $x_i$: 1 rappresenta vero e 0 rappresenta falso. Qundi usando un contatore con $n$-bit riusciamo ad enumerare tutti i $2^n$ possibili assegnamenti delle variabili.

Per ognuno degli assegnamenti controlleremo il valore della formula, e questo può essere fatto con un numero costante di celle di memoria addizionali (ad esempio, potremmo valutare la prima clausola e memorizzare il valore in una variabile, poi procedere valutando le altre clausole se e solo se tutte le precedenti hanno valore vero).

Quindi riusciremo a controllare il valore della formula su tutti i possibile assegnamenti usando tempo esponenziale ma spazio polinomiale. □

Il lemma precedente ha una importante conseguenza:

**Lemma 1.10.3** *$\mathcal{NP} \subseteq$ PSPACE.*

DIMOSTRAZIONE. Sia $X$ un qualsiasi problema in $\mathcal{NP}$. Poichè 3SAT è un problema $\mathcal{NP}$-completo, si ha che $X \leq_P$ 3SAT, e quindi esiste un algoritmo che risolve $X$ usando un numero polinomiale di passi e un oracolo che risolve 3SAT (che può essere chiamato un numero polinomiale di volte). Ma dal lemma precedente sappiamo che possiamo risolvere 3SAT usando spazio polinomiale. Quindi se sostituiamo all'oracolo un algoritmo che usa spazio polinomiale per risolvere 3SAT otteniamo un algoritmo che risolve $X$ in spazio polinomiale. Pertanto $X \in$ PSPACE. □

Si noti che come per $\mathcal{P}$ si ha che $X \in \mathcal{P} \equiv \overline{X} \in \mathcal{P}$, anche per PSPACE si ha che $X \in$ PSPACE $\equiv \overline{X} \in$ PSPACE.

**Lemma 1.10.4** *Co-$\mathcal{NP} \subseteq$ PSPACE.*

DIMOSTRAZIONE. Sia $X \in$ Co-$\mathcal{NP}$. Allora $\overline{X} \in \mathcal{NP}$. Dal lemma precedente si ha quindi che $\overline{X} \in$ PSPACE e quindi $X \in$ PSPACE. □

La Figura 1.21, riassume queste inclusioni fra le classi. Come per la questione $\mathcal{P}$-$\mathcal{NP}$, anche PSPACE è un mistero irrisolto: sebbene si creda che PSPACE sia molto più ampio di $\mathcal{P}$, e che quindi esistano problemi appartenenti a PSPACE ma non a $\mathcal{P}$ o anche problemi non appartenenti $\mathcal{NP}$ o Co-$\mathcal{NP}$, non si sa se $\mathcal{P} \neq$ PSPACE.

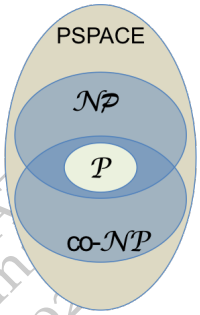


Figura 1.21: Classe PSPACE

### 1.10.1 QSAT

Come esempio di problema in PSPACE vediamo ora un problema legato a 3SAT, il problema QSAT in cui oltre a una formula da soddisfare abbiamo dei quantificatori per ogni variabile. Sia $\Phi(x_1, \ldots, x_n)$ una formula booleana nella forma

$$C_1 \cdot C_2 \ldots \cdot C_k$$

dove ogni clausola $C_i$ è la disgiunzione (l'OR) di 3 letterali. In altre parole, la formula è un'istanza di 3SAT. Assumiamo per semplicità che $n$ sia dispari, il problema QSAT chiede se

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \dots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \dots, x_n) = 1?$$

Cioè, vogliamo sapere se possiamo scegliere un valore per $x_1$ in modo tale che per tutti i valori di $x_2$, esiste una scelta per $x_3$, e così via che rende vera la formula $\Phi$.

Il problema 3SAT, usando i quantificatori verrebbe codificato come:

$$\exists x_1 \exists x_2 \dots \exists x_{n-1} \exists x_n \Phi(x_1, \dots, x_n) = 1?$$

Un esempio concreto di istanza di QSAT è il seguente. Supponiamo che

$$\Phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$$

e chiediamo se

$$\exists x_1 \forall x_2 \exists x_3 \Phi(x_1, x_2, x_3) = 1.$$

La risposta è sì. Infatti possiamo scegliere $x_1 = 1$. A questo punto se $x_2 = 1$, scegliamo $x_3 = 0$ rendendo vera $\Phi$, mentre se $x_2 = 0$ scegliamo $x_3 = 1$ rendendo vera $\Phi$.

### *1.10.2 Un algoritmo per QSAT*

Il problema QSAT può essere risolto in spazio polinomiale. Possiamo usare un approccio molto simile a quello usato per provare che 3SAT può essere risolto in spazio polinomiale. La difficoltà aggiuntiva deriva dalla presenza dei quantificatori. Per 3SAT è sufficiente trovare un assegnamento che rende vera la formula (tutti i quantificatori sono $\exists$). Per tenere conto dei quantificatori possiamo utilizzare un approccio ricorsivo. Se il primo quantificatore della formula è $\exists x_i$, consideriamo entrambi i possibili valori in sequenza. Quindi poniamo $x_i = 0$ e ricorsivamente vediamo se la rimanente parte della formula vale 1. Poi facciamo lo stesso per $x_i = 1$. Basta che in uno solo dei casi la formula valga 1 per ottenere una soluzione. Nel caso invece in cui dobbiamo gestire come primo quantificatore $\forall x_i$, allora procediamo come prima a valutare ricorsivamente entrambe le possibilità per $x_i$, ma questa volta la formula è vera solo se entrambi i sottoproblemi sono veri. Una descrizione informale dell'algoritmo è la seguente.

**Algorithm 2:** Algoritmo per $\text{QSAT}(c_1, \ldots, c_{i-1}, x_i, x_{i+1}, \ldots, x_n)$

```
a = QSAT(c_1, ..., c_{i-1}, 0, x_{i+1}, ..., x_n)        ▷ x_i = 0
Libera la memoria usata per la chiamata ricorsiva
b = QSAT(c_1, ..., c_{i-1}, 1, x_{i+1}, ..., x_n)        ▷ x_i = 1
Libera la memoria usata per la chiamata ricorsiva
if Il primo quantificatore è ∃x_i then
    if a = 1 oppure b = 1 then
        return 1;
    else
        return 0;
if Il primo quantificatore è ∀x_i then
    if a = b = 1 then
        return 1;
    else
        return 0;
```

Analizziamo l'utilizzo di spazio di memoria. Indichiamo con $S(n)$ lo spazio necessario a risolvere un'istanza con $n$ variabili. L'algoritmo ricorsivo effettua 2 chiamate su problemi di taglia $n-1$. Se stessimo valutando il tempo dovremmo avere una relazione di ricorrenza del tipo $S(n) = 2S(n-1) + f(n)$. Ma stiamo valutando lo spazio e la seconda chiamata ricorsiva (ri)utilizza lo stesso spazio usato dalla prima quindi la relazione di ricorrenza sarà

$$S(n) \leq S(n-1) + p(n)$$

dove $p(n)$ è un polinomio in $n$ che tiene conto delle variabili locali necessarie a far funzionare l'algoritmo (di fatto serve solo memorizzare i risultati delle 2 chiamate ricorsive). Quindi, risolvendo la relazione di ricorrenza, abbiamo

$$S(n) \leq p(n) + p(n-1) + p(n-2) + \ldots + p(1) \leq np(n).$$

Pertanto l'algoritmo proposto risolve QSAT usando spazio polinomiale.

### 1.10.3 PSPACE-*completi*

In modo analago a quanto fatto per $\mathcal{NP}$, possiamo definire i problemi PSPACE-completi. Non lo dimostreremo, ma:

**Lemma 1.10.5** QSAT *è* PSPACE-*completo.*

## 1.11 *Note bibliografiche*

Gli argomenti trattati in questo capitolo sono principalmenti tratti da CLRS2009 [8] e KT2014 [20] che possono essere consultati per approfondimenti. La NP-completezza del problema SAT è stata provata indipendentemente da Cook [7] e Levin [25]. Una prova della $\mathcal{NP}$-completezza di CIRCUITSATpuò essere trovata in CLRS2009 [8] (Lemma

34.6) mentre KT2014 [20] fornisce delle motivazioni intuitive a supporto del risultato. Sebbene datato, GJ1979 [14] rimane un'ottima guida alla NP-completezza e un buon catalogo di problemi NP-completi (ovviamente aggiornato al 1979). L'appartenenza di QSAT a PSPACE-completi è stata provata nel 1972 da Stockmeyer e Meyer. QSAT è stato il primo problema ad essere stato provato PSPACE-completo. Quindi è l'analogo di SAT per $\mathcal{NP}$. Per molti altri problemi si è provato l'appartenenza a PSPACE-completi riducendo QSAT ad essi.

## 1.12 *Esercizi*

1. Provare che VERTEXCOVER$\leq_P$INDEPENDENTSET (dimostrazione del Lemma 1.5.9).
2. Provare che INDEPENDENTSET$\leq_P$SETPACKING (dimostrazione del Lemma 1.5.13).
3. Dato un grafo $G = (V, E)$, un *clique* (cricca) è un sottoinsieme $C \subseteq V$ tale che $E$ contiene un arco per ogni coppia di nodi di $C$, cioè $C$ è un sottografo completo. Il problema CLIQUE consiste nell'individuare una clique di taglia massima.
   (a) Fornire una rappresentazione binaria che permetta di descrivere le istanze del problema CLIQUE.
   (b) Indicare il parametro più significativo in relazione alla grandezza del problema e argomentare sulla sua relazione rispetto alla lunghezza della stringa ottenuta con la rappresentazione precedente.
   (c) Fornire la versione decisionale del problema CLIQUE.
   (d) Adattare la rappresentazione binaria per la versione decisionale.
4. Fornire un verificatore per ognuno dei problemi studiati in questo capitolo (CIRCUITSAT, SAT, 3SAT, INDEPENDENTSET, VERTEXCOVER, SETCOVER, SETPACKING, GRAPHCOLORING, GRAPHCOLORING, DIRHAMCYCLE, TSP). Analizzare il tempo necessario alla verifica della potenziale soluzione argomentando la polinomialità rispetto alla taglia dell'input.
5. Si provi che INDEPENDENTSET$\leq_P$SETPACKING (Lemma 1.5.13).
6. Si provi che SAT$\leq_P$3SAT
7. Si consideri il seguente problema: dati due grafi $G_1$ e $G_2$ stabilire se $G_1$ e $G_2$ sono isomorfi (cioè uguale a meno di una ridenominazione dei nodi). Provare che tale problema appartiene alla classe $\mathcal{NP}$.
8. Per ridurre un problema $A$ a un problema $B$ per il quale si ha a disposizione un oracolo, quante volte si può invocare l'oracolo che risolve $B$? Si dia una motivazione della risposta.
9. Si consideri il grafo della Figura 1.22. Si trovi l'insieme indipendente più grande. Quale è il corrispondente insieme ricoprente?



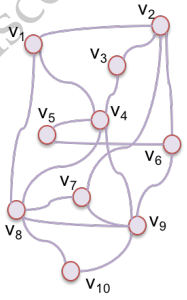



Figura 1.22: Grafo Esercizio 9

10. Abbiamo visto che VertexCover e IndependentSet sono equivalenti, nel senso che l'uno si riduce all'altro grazie al Lemma 1.5.7 che asserisce che un insieme $S \subset V$ è un insieme indipendente se e solo se $V \setminus S$ è un insieme ricoprente. Visto che il problema SetCover è una generalizzazione del problema VertexCover e che il problema SetPacking è una generalizzazione del problema IndependentSet, è abbastanza naturale pensare che SetCover e SetPacking potrebbero essere equivalenti. Una naturale generalizzazione del Lemma 1.5.7 adattata ai problemi SetCover e SetPacking è la seguente: un insieme $S_{i_1}, S_{i_2}, \ldots, S_{i_\ell}$ è ricoprente (cioè è una soluzione a SetCover) se solo se $S_{i_1}, S_{i_2}, \ldots, S_{i_m} \setminus S_{i_1}, S_{i_2}, \ldots, S_{i_\ell}$ è un insieme indipendente (cioè una soluzione a SetPacking). Fornire una prova di tale asserzione oppure mostrare che non è vera.

11. Si consideri il problema 4Sat, definito come 3Sat, ma con 4 letterali in ogni clausola. Si riduca 4Sat a IndependentSet.

12. Si consideri il circuito riportato nella figura 1.23. Si fornisca la formula 3Sat della riduzione da CircuitSat a 3Sat.

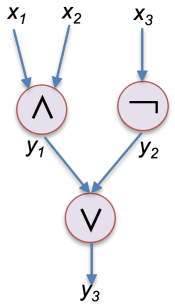



Figura 1.23: Circuito

13. ($\star$) Si provi che il problema Clique, definito nell'Esercizio 3 è $\mathcal{NP}$-completo.

14. ($\star$) Si provi che il problema SubsetSum è $\mathcal{NP}$-completo.

15. Un grafo $G = (V, E)$ è bipartito se $V$ può essere partizionato in $V = V_1 \cup V_2$ in modo tale che gli archi hanno tutti un vertice in $V_1$ e uno in $V_2$. Provare che un grafo $G$ è 2-colorabile se e solo se $G$ è bipartito. Fornire un algoritmo efficiente per stabilire se un grafo è bipartito.

16. Nella riduzione di 3Sat a GraphColoring i 3 nodi corrispondenti ai 3 letterale di una clausola vengono uniti in un sottografo con 6 nodi fittizi. Quale è la proprietà garantita da tale sttografo? Riesci a ricostruire il sottografo?

17. Nella riduzione 3Sat$\leq_P$ DirHamCycle, Lemma 1.5.19, sono stati usati dei nodi definiti "di transito". Quale è la loro funzione? Si fornisca una motivazione.

18. Riguardo l'equivalenza fra versioni decisionali e di ricerca, abbiamo visto come Sat sia *auto-riducibile*, cioè sia possibile risolvere la versione di ricerca di Sat usando come subroutine la versione deicisionale. Mostra che anche Clique è auto-riducibile.

19. Prova che per un problema $\mathcal{NP}$-completo la versione decisionale permette di risolvere anche la versione di ricerca.

# 2
# *Trattare l'intrattabile*

In questo capitolo discuteremo di come affrontare problemi per i quali non abbiamo algoritmi efficienti. Per tali problemi dobbiamo necessariamente usare un approccio non efficiente. Tuttavia, in molti casi, possiamo utilizzare degli accorgimenti che limitano l'inefficienza.

## *2.1 Introduzione*

Supponiamo che vi venga dato un problema per il quale dovete fornire un algoritmo. Dopo aver pensato un po' ed aver provato ad usare le tecniche imparate durante i corsi di algoritmi vi accorgete di non essere riusciti a cavare un ragno dal buco. E se il problema fosse di quelli difficili, cioè un problema $\mathcal{NP}$-hard? Sarebbe uno di quelli per i quali nessuno è riuscito a trovare un algoritmo efficiente perché magari tale algoritmo non esiste!

Quando null'altro funziona, possiamo usare la tecnica meno creativa possibile: provare tutte le possibilità. Fare cioè una *ricerca esaustiva* su tutto lo spazio delle soluzioni. Tale approccio, noto anche come *ricerca a forza bruta*, in linea teorica risolve qualsiasi problema. Ovviamente lo fa usando molto tempo. Non è difficile fornire degli esempi in cui una tale tecnica, usando anche i computer più veloci finora costruiti, impiega anni, secoli, millenni, migliaia di millenni e più per risolvere il problema.

Consideriamo ad esempio il problema della fattorizzazione. Dato un numero intero maggiore di 1, stabilire quali sono i suoi fattori. Questo problema ha un'enorme importanza pratica in quanto la sicurezza di molti sistemi crittografici si basa sulla difficoltà di fattorizzare un numero $N = a \cdot b$ che è ottenuto come prodotto di due numeri primi $a$ e $b$ molto grandi. Se il numero $N$ che ci viene fornito come input è piccolo, non è difficile scoprire i suoi fattori e quindi anche $a$ e $b$ nel caso in cui $N$ sia il prodotto di due primi. Ad esempio se $n = 221$ l'approccio a forza bruta, che consiste nel dividere $N$ per tutti gli interi a partire da 2 fino a che non si trova un divisore (che può essere $N$ stesso se $N$ è primo), non impiegherà molto tempo per scoprire che $221 = 13 \cdot 17$. Tuttavia più grande è $N$ e più tempo servirà a trovare i fattori di $N$. Si noti che la "taglia" del problema della fattorizzazione è di $n = \log N$ bit e quindi provare a dividere per tutti i valori da 2 a $N$ costa tempo esponenziale rispetto alla taglia dell'input e quindi basta aumentare $N$ per arrivare abbastanza rapidamente a tempi di esecuzione

di giorni, mesi, anni, secoli o millenni (e anche più). E anche piccoli accorgimenti, come provare a dividere solo fino a $N/2$ non aiutano molto. Considerando le capacità computazionali dei computer attuali, per rendere il problema intrattabile, si usano numeri dell'ordine di 2048 bit.

## 2.2 *Esercizio di programmazione*

Scrivere un programma che esamina un numero $N$ per stabilire se è primo oppure se è il prodotto di fattori dividendolo per tutti i numeri più piccoli di $N/2$. Nel caso $N$ sia il prodotto di fattori si stampino i fattori. Si faccia anche stampare al programma il tempo che impiega per risolvere il problema. Si utilizzi il programma per capire in che modo varia il tempo di esecuzione al crescere della taglia dell'input. Nel riquadro è riportato un programma Java.

```
import java.util.Scanner;
import java.math.BigInteger;
import java.util.*;
public class Fattorizzazione {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Inserisci un numero: ");
    BigInteger n = scanner.nextBigInteger();
    BigInteger ZERO = BigInteger.ZERO;
    BigInteger TWO = BigInteger.valueOf(2);
    if (n.compareTo(TWO)<0) {
      System.out.println(String.format("Il numero inserito %s e' minore di 2",n.toString()));
      System.exit(0);
    }
    long t1,t2;
    List divisori = new ArrayList();
    t1 = System.currentTimeMillis();
    System.out.println(String.format("tempo ms: %s",t1));
    BigInteger nhalf = n.divide(TWO);
    for (BigInteger i = TWO; i.compareTo(nhalf) < 0; i = i.add(BigInteger.ONE)) {
      BigInteger risultato[] = n.divideAndRemainder(i);
      if (risultato[1].compareTo(ZERO)==0) {
        divisori.add(i);
      }
    }
    if (divisori.size()==0) {
      System.out.println(String.format("Il numero inserito (%d) e' primo", n));
    }
    else {
      System.out.println(String.format("Il divisori di %d sono "+divisori, n));
    }
    t2 = System.currentTimeMillis();
    System.out.println(String.format("tempo ms: %s",t2));
    long delta=t2-t1;
    System.out.println("Millisecondi: "+delta);
  }
}
```

La RSA Security, fondata dagli inventori della crittografia a chiave pubblica, Rivest, Adleman e Shamir, ha proposto dal 1991 al 2007 una sfida pubblica per la fattorizzazione di alcuni numeri semiprimi (cioè prodotti di due primi). Questo è il numero RSA-2048: 25195908475657893494027183240048398571429282126204032027777137836043662020707595556264018525880784406918290641249515082189298559149176184502808489120072844992687392807287776735971418347270261896375014971824691165077613379859095700097330459748808428401797429100642458691817195118746121515172654632282216869987549182422433637259085141865462043576798423387184774447930039314577632734460428151262720373431465319777741603199066554187639792933441952154134189948544473456738382414323551374637452445581765720739613204877320213289422737825372917228294684727232791786085784383827967976681454101180701125416111747449856336733624406566430860213949463952247371907021798609437027705392171762931767523846733559336007192120190516660962842566091518151763905834618762500099490873609034492399732654517215251907705843641409746313796288297039540173795833190281776984645997105034327671136651161610704402135969536231442952484937187110145765403590278729074496661521199876964992851600370447613779516684922887 5. Ha 617 cifre decimali (2048 bit) e non è stato ancora fattorizzato. RSA-1024 ha 309 cifre decimali ed è stato fattorizzato nel 2003 grazie a uno sforzo congiunto di un sistema distribuito composto da circa 30.000 computers che hanno contribuito con potenza di calcolo per molti mesi.

## 2.3 *Ricerca esaustiva intelligente*

Se proprio dobbiamo usare una ricerca esaustiva possiamo sfruttare degli accorgimenti che permettono di migliorare il tempo di esecuzione (che rimarrà comunque non polinomiale). Durante la ricerca esaustiva possiamo eliminare grossi sottoinsiemi di potenziali soluzioni in un solo colpo diminuendo così il tempo totale. Ad esempio, nell'esercizio precedente abbiamo risolto il problema tentando di dividere $N$ per tutti i numeri più piccoli fino ad $N/2$, scartando a priori i numeri più grandi di $N/2$ in quanto essi non possono dividere $N$. Il vantaggio in questo caso è stato quello di dimezzare l'insieme dei potenziali divisori di $N$. Ovviamente, in questo caso, basta raddoppiare il valore di $N$ per vanificare questo piccolo vantaggio. Tuttavia, anche se piccoli, i vantaggi possono essere significativi in pratica.

Le tecniche conosciute con i nomi di *backtracking* e *branch and bound* permettono di ottenere dei miglioramenti. Il backtracking si basa sull'osservazione che a volte è possibile rifiutare una potenziale soluzione esaminando solo una piccola parte che però è sufficiente per capire che non siamo di fronte ad una soluzione del nostro problema. Analogamente il branch and bound applica lo stesso principio a problemi di ottimizzazione, scartando in un solo colpo tutte le soluzioni per le quali siamo in grado di dire, senza esaminarle nei dettagli, che il loro valore non può essere quello ottimale. Nel seguito faremo due esempi per illustrare queste tecniche.

## 2.4 *Backtracking*

### 2.4.1 SAT

Consideriamo il problema SAT introdotto nel capitolo precedente: data una formula booleana stabilire se esiste un assegnamento delle variabili che la rende vera. Se un tale assegnamento esiste, la formula si dice soddisfacibile. Per semplicità considereremo questo problema nella sua formulazione in forma normale congiuntiva, cioè usando formule booleane che sono costruite come AND di OR. Questa non è una restrizione in quanto una qualsiasi formula può essere espressa in forma normale congiuntiva.

Per essere concreti nella spiegazione, consideriamo un esempio:

$$\phi = (a + \neg b + c + \neg d) \cdot (a + b) \cdot (a + \neg b) \cdot (\neg a + c) \cdot (\neg a + \neg c) \tag{2.1}$$

Dato uno specifico assegnamento di valori alle variabili $a, b, c$ e $d$ è facile verificare se $\phi$ è vera o falsa. Basta applicare la formula. Ad esempio se $a = 1$, $b = 0$, $c = 1$, $d = 0$, si ha che

$$\begin{aligned} \phi &= (1 + \neg 0 + 1 + \neg 0) \cdot (1 + 0) \cdot (1 + \neg 0) \cdot (\neg 1 + 1) \cdot (\neg 1 + \neg 1) \\ &= (1 + 1 + 1 + 1) \cdot (1 + 0) \cdot (1 + 1) \cdot (0 + 1) \cdot (0 + 0) \\ &= (1) \cdot (1) \cdot (1) \cdot (1) \cdot (0) \\ &= 0 \end{aligned}$$

Per trovare un assegnamento che renda $\phi$ vera o per stabilire che non esiste un tale assegnamento possiamo usare una ricerca esaustiva: ci sono 16 possibili assegnamenti delle 4 variabili. Possiamo controllarli tutti per scoprire se uno di essi rende la formula vera oppure se non esiste nessun assegnamento che rende la formula vera. Se guardassimo tale processo con un albero delle decisioni, che in questo caso possiamo a ragione chiamare albero degli assegnamenti, in cui ogni volta decidiamo se assegnare valore 1 o 0 ad una delle variabili, le 16 possibilità corrisponderebbero all'albero mostrato nella Figura 2.1.

In questo esempio ci sono solo 16 foglie nell'albero degli assegnamenti. In generale ci saranno $2^n$ foglie, dove $n$ è il numero di variabili usate nella formula, quindi un numero esponenziale. Ogni foglia corrisponde ad un assegnamento di tutte le variabili, mentre ogni nodo interno corrisponde ad un assegnamento parziale delle variabili. La radice corrisponde all'assegnamento di nessuna variabile, cioè alla formula iniziale. Per verificare tutti gli assegnamenti con una ricerca esaustiva dobbiamo visitare l'intero albero degli assegnamenti con un costo che è proporzionale al numero totale di nodi dell'albero.

Il backtracking ci aiuta a non esplorare tutto l'albero se non nei casi in cui è strettamente necessario. Infatti è possibile fermarsi prima (a parte quando troviamo una soluzione) quando ci accorgiamo che è inutile proseguire su un determinato cammino nell'albero in quanto sicuramente la formula non sarà soddisfatta dagli assegnamenti specificati in quel sottoalbero. Da qui il nome di "backtracking": torniamo indietro per provare altre strade. Vediamo come si applica tale tecnica all'esempio di cui prima.

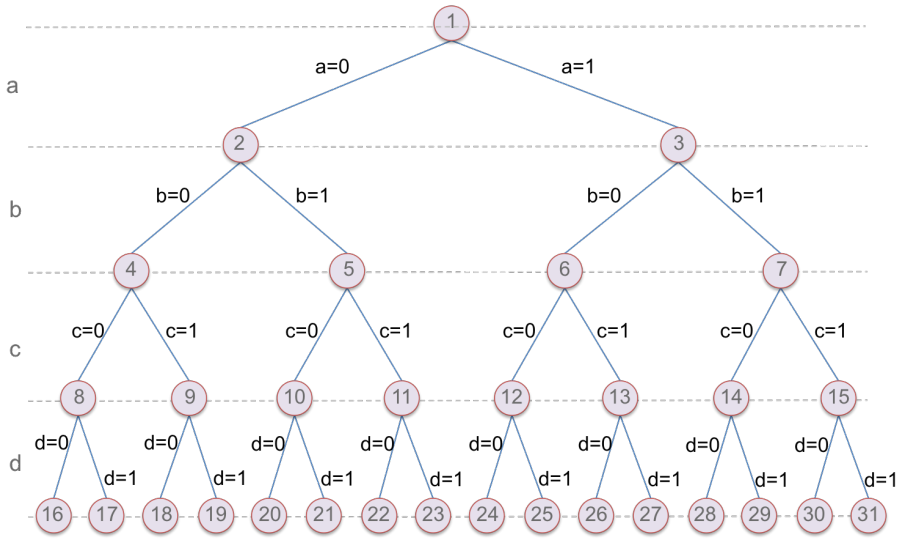



Figura 2.1: Albero degli assegnamenti

Nel primo passo consideriamo la scelta per la variabile $a$. Questo ci porterà ad un nuovo problema, sottoproblema del precedente, nel quale abbiamo assegnato un valore ad $a$. Supponiamo di esplorare prima la scelta $a = 0$. La risultante formula sarà:

$$\begin{aligned} \phi(0, b, c, d) &= (\neg b + c + \neg d) \cdot (b) \cdot (\neg b) \cdot (1) \cdot (1) \\ &= (\neg b + c + \neg d) \cdot (b) \cdot (\neg b) \end{aligned}$$

Se, a partire da questo nuovo problema, consideriamo ora la scelta per la variabile $b$ avremo un nuovo sottoproblema. Supponiamo di esplorare l'assegnamento $b = 0$, avremmo che

$$\phi(0, 0, c, d) = (c + \neg d) \cdot (0) \cdot (1) = 0$$

A questo punto è inutile procedere ad esplorare ulteriori assegnamenti a $c$ e $d$: la formula rimarrà falsa. Conviene "tornare indietro" e provare altre scelte risparmiandoci così la visita di quel particolare sottoalbero.

La Figura 2.2 mostra il ragionamento fatto in precedenza. Il nodo radice corrisponde alla formula iniziale $\phi_1 = \phi$. La scelta $a = 0$ porta alla formula $\phi_2 = \phi(0, b, c, d)$, e l'ulteriore scelta $b = 0$ porta alla formula $\phi(0, 0, c, d) = 0$. La figura mette in evidenza come le scelte $a = 0$ e $b = 0$ da sole determinano il valore di $\phi = 0$. La restante parte di quel sottoalbero può essere ignorata in quanto non conterrà nessun assegnamento che rende vera la formula.

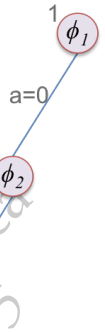



Figura 2.2: Valutazione parziale

Procedendo analogamente per le altre scelte, otteniamo l'albero mostrato nella Figura 2.3. In questo particolare esempio, abbiamo scoperto che la $\phi$ non è soddisfacibile senza esplorare tutto l'albero. Questo si traduce in un algoritmo più veloce rispetto alla ricerca esaustiva normale. Si ricordi però che in ogni caso la complessità rimane esponenziale.

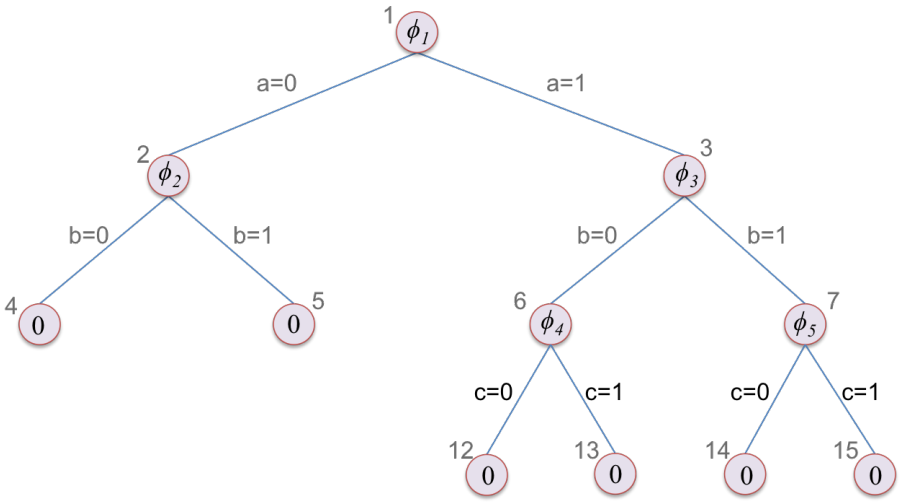



Figura 2.3: Albero di backtrack

### 2.4.2 Sudoku

Consideriamo ora un problema meno teorico e più conosciuto anche da chi non si occupa di informatica: il Sudoku. Partiamo da una versione più semplice, denominata LatinSquare. Nel problema LatinSquare abbiamo una scacchiera $n \times n$, in cui ogni cella può essere vuota oppure contenere un numero fra 1 e $n$. Il problema consiste nel riempire tutte le caselle inizialmente vuote con numeri compresi fra 1 e $n$ in modo tale che in ogni riga e in ogni colonna non ci siano numeri ripetuti.

Osserviamo che se la scacchiera è completamente vuota è facile trovare una soluzione: nella prima riga possiamo inserire la sequenza $\langle 1, 2, \ldots, n-1, n \rangle$, e nelle successive righe uno shift ciclico (nella riga $i$ uno shift di $i-1$ posizioni, $i = 1, 2, \ldots, n$), come mostrato nella Figura 2.4 per il caso $n = 4$.

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 4 | 1 | 2 | 3 |
| 3 | 4 | 1 | 2 |
| 2 | 3 | 4 | 1 |

Figura 2.4: Un esempio di LatinSquare per $n = 4$

Se invece la scacchiera contiene già degli elementi potrebbe non essere possibile trovare un completamento che soddisfa il vincolo richiesto dal problema. In pratica la presenza di elementi iniziali riduce lo spazio delle potenziali soluzioni, ma lo può ridurre talmente tanto da renderlo vuoto. Un esempio è mostrato nella Figura 2.5: per non violare il vincolo del problema sulla seconda e terza riga, l'ultimo elemento deve necessariamente essere 1 e quindi ci sarebbe una violazione sulla quarta colonna. Per questa istanza non esiste un completamento.

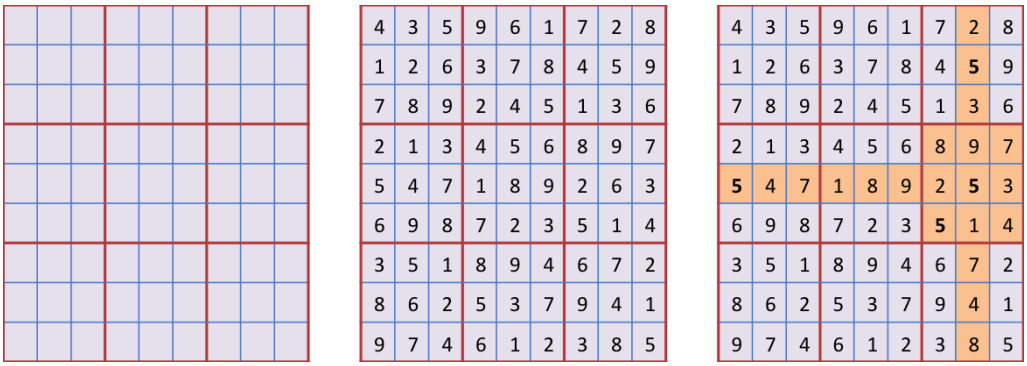
| | | | |
|---|---|---|---|
| 1 | 2 | | 4 |
| 4 | 3 | 2 | |
| 2 | 4 | 3 | |
| | | | |

Figura 2.5: Un esempio di LatinSquare per $n = 4$

Data una configurazione iniziale, in generale, per stabilire se la scacchiera può essere completata non sappiamo far meglio di una ricerca esaustiva; infatti il problema LatinSquare è NP-hard[1].

Il Sudoku è una versione più complessa di LatinSquare: la scacchiera del Sudoku contiene $n^2$ sottoscacchiere di dimensione $n \times n$, quindi è una scacchiera di $n^2 \times n^2$ caselle che devono essere riempite usando i numeri da 1 a $n^2$, in modo tale che in ogni riga, in ogni colonna ed in ogni sottoscacchiera $n \times n$, non ci siano mai numeri ripetuti. La Figura 2.6 mostra, per il caso $n = 3$, che è quello utilizzato nelle riviste di enigmistica, una scacchiera vuota, una configurazione legale ed una configurazione che vìola le 3 proprietà: nella riga, colonna e sottoschachiera evidenziate si ripete un 5.

[1] C. J. Colbourn. The complexity of completing partial latin square. *Discrete Applied Mathematics*, 8(1):25–30, 1984



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 3 | 5 | 9 | 6 | 1 | 7 | 2 | 8 |
| 1 | 2 | 6 | 3 | 7 | 8 | 4 | 5 | 9 |
| 7 | 8 | 9 | 2 | 4 | 5 | 1 | 3 | 6 |
| 2 | 1 | 3 | 4 | 5 | 6 | 8 | 9 | 7 |
| 5 | 4 | 7 | 1 | 8 | 9 | 2 | 6 | 3 |
| 6 | 9 | 8 | 7 | 2 | 3 | 5 | 1 | 4 |
| 3 | 5 | 1 | 8 | 9 | 4 | 6 | 7 | 2 |
| 8 | 6 | 2 | 5 | 3 | 7 | 9 | 4 | 1 |
| 9 | 7 | 4 | 6 | 1 | 2 | 3 | 8 | 5 |

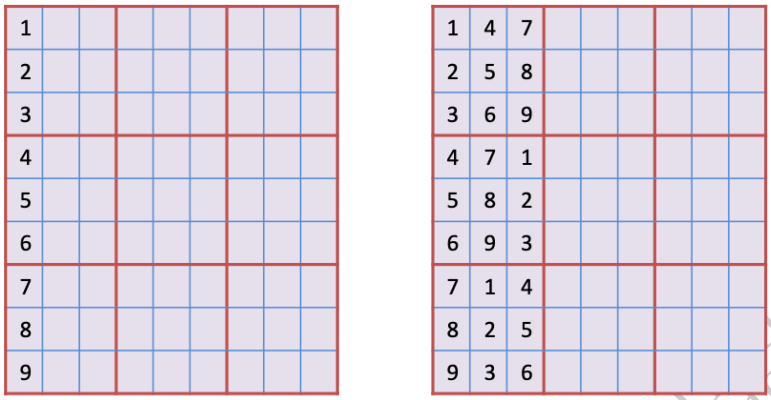
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 3 | 5 | 9 | 6 | 1 | 7 | 2 | 8 |
| 1 | 2 | 6 | 3 | 7 | 8 | 4 | **5** | 9 |
| 7 | 8 | 9 | 2 | 4 | 5 | 1 | 3 | 6 |
| 2 | 1 | 3 | 4 | 5 | 6 | 8 | 9 | 7 |
| **5** | 4 | 7 | 1 | 8 | 9 | 2 | **5** | 3 |
| 6 | 9 | 8 | 7 | 2 | 3 | **5** | 1 | 4 |
| 3 | 5 | 1 | 8 | 9 | 4 | 6 | 7 | 2 |
| 8 | 6 | 2 | 5 | 3 | 7 | 9 | 4 | 1 |
| 9 | 7 | 4 | 6 | 1 | 2 | 3 | 8 | 5 |

Figura 2.6: Scacchiere SUDOKU per $n = 3$

Anche per il SUDOKU è facile trovare una soluzione se la scacchiera è vuota; tuttavia la soluzione è leggermente meno immediata per via del vincolo sulle sottoscacchiere. Una soluzione, che chiameremo *soluzione base*, è la seguente: si riempie la prima sottoscacchiera con i numeri da 1 a $n^2$ procedendo prima per colonne e poi per righe; le altre sottoscacchiere verranno riempite con una permutazione fatta sia sulle righe che sulle colonne.

Più formalmente, chiameremo schacchiera "madre" la scacchiera del SUDOKU. La scacchiera madre contiene $n^2$ sottoscacchiere che numeriamo partendo dall'alto verso il basso e da sinistra verso destra (quindi la sottoscacchiera 1 è quella in alto a sinistra, la sottoscacchiera $n^2$ quella in basso a destra. Nella prima "riga" di sottoscacchiere troviamo le sottoscacchiere numerate da 1 a $n$. Nella seconda quelle da $n+1$ a $2n$, e così via, fino all'ultima riga dove troviamo le sottoscacchiere dalla $(n-1)n+1$ a $n^2$.

Per riempire le $n^2$ sottoscacchiere procediamo in questo modo. Iniziamo riempiendo la prima colonna della scacchiera madre con i numeri da 1 a $n$. Quindi procediamo a riempire le colonne dalla 2 alla $n$ della scacchiera madre con degli shift ciclici di $n$ posizioni. La Figura 2.7 mostra questi due passi per il caso $n = 3$.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |
| 6 | | | | | | | | |
| 7 | | | | | | | | |
| 8 | | | | | | | | |
| 9 | | | | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 7 | | | | | | |
| 2 | 5 | 8 | | | | | | |
| 3 | 6 | 9 | | | | | | |
| 4 | 7 | 1 | | | | | | |
| 5 | 8 | 2 | | | | | | |
| 6 | 9 | 3 | | | | | | |
| 7 | 1 | 4 | | | | | | |
| 8 | 2 | 5 | | | | | | |
| 9 | 3 | 6 | | | | | | |

Figura 2.7: Prime $n$ colonne della soluzione *base* del SUDOKU $n = 3$

Abbiamo così riempito la prima sottoscacchiera di ogni riga, cioè le sottoscacchiere numero $1, n+1, 2n+1, \ldots, (n-1n)n+1$. A questo punto procediamo a rimpiere le restanti sottoscacchiere operando un shift ciclico diagonale sulla scacchiera immedi-

atamente alla sinistra. Lo shift ciclico diagonale si ottiene spostando l'elemento $(i, j)$ nella posizione $(i+1, j+1)$ all'interno della sottoscacchiera. La Figura 2.8 mostra un esempio di shift ciclico diagonale.

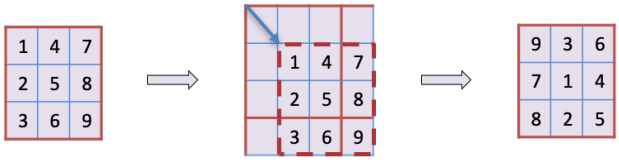



Figura 2.8: Shift ciclico diagonale.

Quindi, per ogni riga, la "prossima" scacchiera, numero $i$, viene riempita con lo shift ciclico diagonale della scacchiera precedente, numero $i-1$.

La Figura 2.9 mostra il risultato finale per il caso $n = 3$.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 7 | 9 | 3 | 6 | 5 | 8 | 2 |
| 2 | 5 | 8 | 7 | 1 | 4 | 6 | 9 | 3 |
| 3 | 6 | 9 | 8 | 2 | 5 | 4 | 7 | 1 |
| 4 | 7 | 1 | 3 | 6 | 9 | 2 | 5 | 8 |
| 5 | 8 | 2 | 1 | 4 | 7 | 3 | 6 | 9 |
| 6 | 9 | 3 | 2 | 5 | 8 | 1 | 4 | 7 |
| 7 | 1 | 4 | 6 | 9 | 3 | 8 | 2 | 5 |
| 8 | 2 | 5 | 4 | 7 | 1 | 9 | 3 | 6 |
| 9 | 3 | 6 | 5 | 8 | 2 | 7 | 1 | 4 |

Figura 2.9: Permutazione colonne sottoscacchiere

Come per il LatinSquare, anche per il Sudoku, la presenza di numeri iniziali può rendere il problema non risolvibile, e per scoprirlo non sappiamo fare meglio di una ricerca esaustiva: non dovrebbe sorprendere il fatto che anche il Sudoku è un problema $\mathcal{NP}$-hard.

**Lemma 2.4.1** LatinSquare$\leq_P$Sudoku.

Dimostrazione. Consideriamo un'istanza del problema LatinSquare. Indichiamo con $c_1, \ldots, c_n$ le $n$ colonne della scacchiera. Costruiamo la seguente istanza del Sudoku per una scacchiera di dimensione $n^2 \times n^2$. Partiamo dalla soluzione base costruita come spiegato in precedenza da una scacchiera vuota. Individuiamo le colonne delle sottoscacchiere da 1 a $n$ che contengono i numeri da 1 a $n$ e sostituiamo con $c_1, \ldots, c_n$ tali colonne.

La Figura 2.10 mostra con un esempio per il caso $n = 3$ la costruzione dell'istanza del problema Sudoku. A sinistra c'è un'istanza del problema LatinSquare, con le colonne $c_1, c_2$ e $c_3$. Nel centro c'è la soluzion base e sono state individuate le colonne con i numeri da 1 a $n$. A destra l'istanza del Sudoku in cui le colonne $c_1, c_2$ e $c_3$ sono state inserite nelle prime 3 sottoscacchiere, nelle colonne individuate nel passo percedente.

La trasformazione descritta sopra può essere fatta in tempo polinomiale.

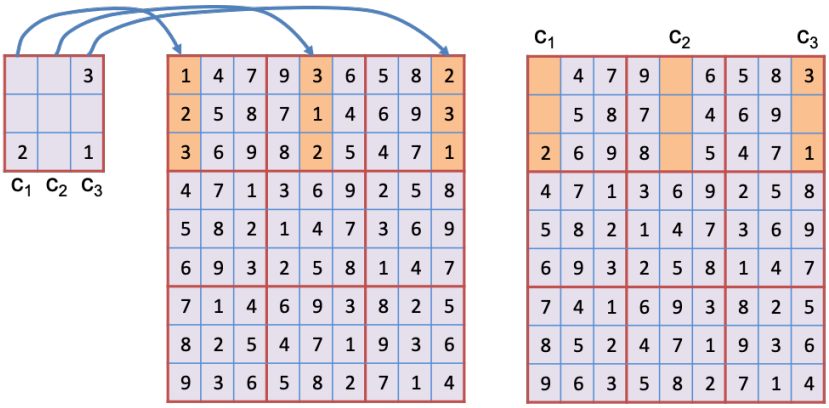



Figura 2.10: Istanza SUDOKU da istanza LATINSQUARE.

Per come è stata costruita l'istanza del SUDOKU si ha che nelle caselle vuote delle colonne $c_i$ è possibile inserire solo i numeri da 1 a $n$. Pertanto non rimane che risolvere il problema del SUDOKU, chiamando l'oracolo. Le colonne $c_i$ forniscono una soluzione anche all'istanza iniziale del problema LATINSQUARE: infatti la proprietà di non ripetizione sulle righe e sulle colonne della scachiera del SUDOKU, garantisce che non ci siano ripetizioni, sia sulle righe che sulle colonne, nei numeri contenuti nelle colonne $c_i$. Pertanto le colonne $c_i$ della soluzione del SUDOKU sono una soluzione che completa l'istanza iniziale del LATINSQUARE. □

Il problema LATINSQUARE è $\mathcal{NP}$-hard. Quindi anche il problema del SUDOKU è $\mathcal{NP}$-hard.

Poichè non esiste una soluzione efficiente per il SUDOKU possiamo utilizzare la ricerca esaustiva. Specificheremo una potenziale soluzione con una sequenza $\sigma$ di $n^4$ numeri:

$$\sigma = [\sigma_1, \sigma_2, \ldots, \sigma_{n^4}]$$

ordinati per righe (cioè i primi $n^2$ corrispondono alla prima riga della scacchiera, i successivi alla seconda e così via). Proseguiremo la spiegazione considerando il caso $n = 3$ per il quale $\sigma$ ha 81 elementi. Per trovare una soluzione consideriamo tutte le possibili sequenze di 81 elementi, in cui ogni elemento è un numero fra 1 e 9. Una sequenza è una soluzione del SUDOKU, se soddisfa i vincoli del problema. Si noti che quando partiamo da una soluzione parziale, cioè quando alcuni numeri della sequenza sono già fissati, dovremo considerare meno possibilità.

Per verificare una sequenza dovremo controllare tutte le righe, tutte le colonne, e tutte le sottoscacchiere $3 \times 3$. Tuttavia quando troviamo una violazione di un vincolo in una parte della sottosequenza possiamo scartare in un solo colpo tutte le sequenze che hanno quella particolare sottosequenza. Ad esempio tutte le sequenze che inziano con $1, 2, 3, 4, 5, 6, 7, 8, 9, 3$ non sono una soluzione in quanto ci sono due 3 nella prima sottoscacchiera $3 \times 3$.

Anche per il SUDOKU, come fatto per SAT, possiamo usare un albero per rappresentare tutte le possibili sequenze. In questo caso l'albero è leggermente più complicato; in realtà è solo più grande, il che lo rende più complicato da disegnare. Infatti è un albero che ha 81 livelli, ognuno dei quali corrisponde ad una casella della scacchiera, ed ogni

nodo dell'albero ha 9 figli, che corrispondono ai 9 possibili valori che possiamo inserire nelle caselle.

Ogni arco dell'albero rappresenta un assegnamento di uno specifico valore ad una specifica casella della scacchiera. Ogni nodo interno dell'albero corrisponde ad una sequenza parziale, data dagli assegnamenti dalla radice a quel nodo. Ogni foglia rappresenta una possibile sequenza. L'albero è enorme: ha $9^{81}$ foglie, e $9^{81}$ è un numero estremamente grande:

$$9^{81} = 196627050475552913618075908526912116283103450944214766927315415537966391196809.$$

Con un computer molto veloce, ad esempio capace di controllare $4 \cdot 10^9$ sequenze al secondo[2] avremmo bisogno di $9^{81}/(4 \cdot 10^9)$, cioè

$$49156762618888228404518977131728029070775862736053691731828853884491$$

secondi, il che significa più di 24277016742770 anni.

Usando il backtracking possiamo esplorare lo spazio delle soluzioni più efficientemente.

[2] Un computer con una CPU da 4GhZ può eseguire $4 \cdot 10^9$ istruzioni al secondo. Per controllare una sequenza non basta una singola istruzione.

## 2.5 *Esercizio di programmazione*

Scrivere un programma che utilizza il backtracking per risolvere il SUDOKU.

```
/* Il codice di un programma Java e' disponibile nel sito del corso */
```

## 2.6 *Branch and bound*

### 2.6.1 TSP

L'idea di base del backtracking può essere estesa a problemi di ottimizzazione. Il problema della soddisfacibilità di formule booleane è un problema decisionale, cioè un problema in cui si deve rispondere solo sì (se esiste una soluzione) oppure no (se non esiste una soluzione). Ogni soluzione è equivalente ad ogni altra soluzione e l'unica domanda che ci si pone è: esiste (almeno) una soluzione al problema? Nei problemi di ottimizzazione invece ogni soluzione ha un valore ed il problema è quello di trovare la soluzione ottima (un minimo o un massimo).

Come esempio considereremo il problema del commesso viaggiatore (Traveling Salesman Problem, TSP) già introdotto nel capitolo precedente: abbiamo un grafo $G = (V, E)$ con dei costi (distanze) associati ad ogni arco; ogni nodo rappresenta un luogo (città) ed ogni arco il costo per spostarsi da una città all'altra o la distanza fra le città; il commesso viaggiatore deve trovare un giro del grafo (cioè di tutte le città) in modo tale da visitare ogni città esattamente una volta e viaggiando/spendendo il meno possibile.

In questo problema l'albero delle decisioni rappresenta tutti i possibili "giri" del grafo. Poichè dobbiamo visitare ogni nodo esattamente una volta non ha importanza da quale nodo partiamo; detto in altre parole, possiamo scegliere un qualsiasi nodo come punto di partenza. L'albero delle decisioni rappresenta tutti i possibili cammini: ogni nodo dell'albero rappresenterà un cammino parziale dal nodo di partenza ad un altro nodo $v$ ed i suoi figli saranno tutti i possibili nodi raggiungibili da $v$ che non sono stati ancora visitati.

Consideriamo ad esempio il grafo mostrato nella Figura 2.11. Scegliendo di partire dal nodo $a$, le possibili scelte per il prossimo nodo sono $b, c, d$ ed $e$. Da $b$ possiamo raggiungere $a, c$, ed $e$, e così via. Scartando cammini che ci portano in nodi già visitati, possiamo costruire l'albero "dei percorsi" mostrato nella Figura 2.12.

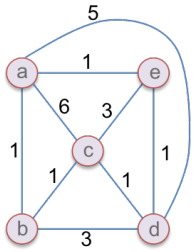



Figura 2.11: Grafo

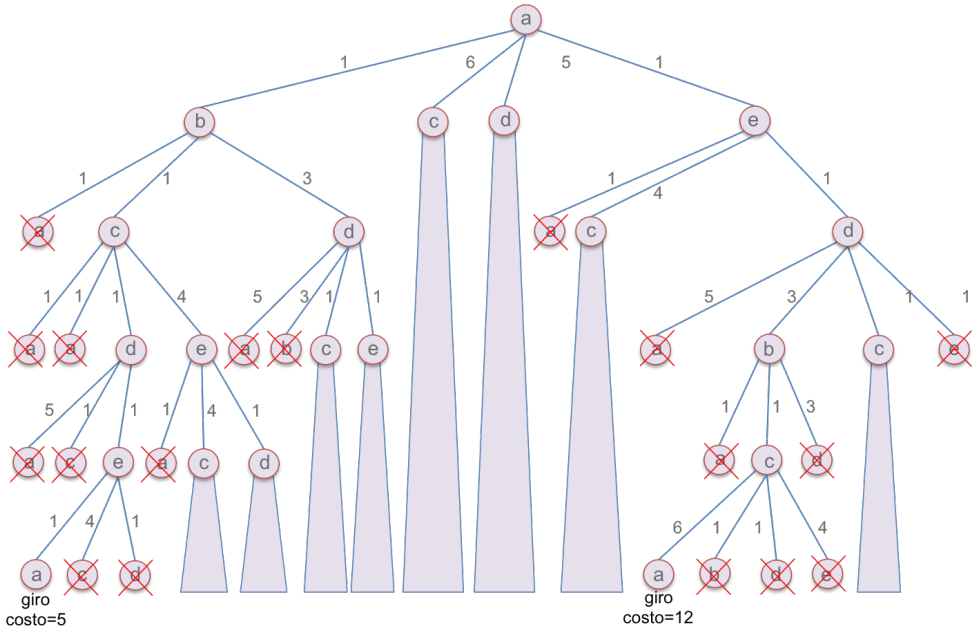



Figura 2.12: Albero dei percorsi

Ovviamente tale albero ha un numero di nodi esponenziale in $n$, dove $n$ è il numero di nodi nel grafo $G$. La Figura riporta solo una parte dell'albero evidenziando due possibili soluzioni (giri) una con costo 5 e l'atra con costo 12. I nodi contrassegnati con una X rossa sono percorsi che formano un ciclo e quindi non sono soluzioni del problema. Esplorare tutto l'albero costa tempo esponenziale in $n$.

Usando la stessa idea che ha portato un miglioramento nel caso del backtracking possiamo evitare di visitare parte di questo enorme albero se riusciamo in qualche modo a dire che in un determinato sottoalbero non c'è una soluzione ottima e quindi è inutile andare a cercarla lì. Per fare questo notiamo che stiamo cercando un valore minimo (il giro con distanza totale minima). Pertanto se riusciamo a dire che tutti i percorsi che possiamo individuare a partire da un determinato nodo dell'albero hanno un costo totale superiore al costo di un altro percorso già individuato possiamo concludere che in quel sottoalbero non ci sono soluzioni ottime e quindi possiamo evitare di visitarlo.

Un limite ovvio e semplice da ottenere è il costo del cammino parziale che corrisponde

al nodo stesso. Un nodo interno dell'albero rappresenta un determinato percorso parziale $\alpha$ dal nodo di partenza, che nel nostro esempio è $a$, a un nodo $v = v(\alpha)$ che è il nodo al quale si arriva tramite il percorso parziale. Il costo del percorso $\alpha$ è chiaramente un limite inferiore per tutti i giri che hanno $\alpha$ come parte iniziale.

Inoltre possiamo ottenere dei limiti inferiori più alti osservando che il costo di un qualsiasi prosieguo $\beta$ che permetta di completare $\alpha$ in un giro, dovrà essere la somma di almeno:

- il costo dell'arco con costo minimo da $a$ ad un nodo non presente in $\alpha$ (perchè dobbiamo finire il giro in $a$, quindi dobbiamo prima o poi ritornare in tale nodo).
- costo dell'arco con costo minimo da $v$ ad un nodo non presente in $\alpha$ (perchè qualunque percorso che estende $\alpha$ ha un arco dal nodo $v$ ad un nodo $w \notin \alpha$).
- costo di un albero minimo ricoprente dei nodi non presenti in $\alpha$ (perchè dobbiamo visitare tutti i nodi ancora non visitati).

Una volta stabilito un limite inferiore al costo di un qualsiasi giro che è un completamento del percorso parziale $\alpha$, se tale costo è più alto del costo di un giro già noto, possiamo evitare di visitare il sottoalbero radicato nel nodo $v(\alpha)$.

Riprendendo l'esempio precedente e sfruttando la tecnica di branch and bound, riusciamo a risolvere il problema visitando una parte molto più piccola dell'albero, come mostrato nella Figura 2.13. In questo esempio è stato usato il costo del cammino parziale già costruito, come limite inferiore al costo totale dei giri che si possono ottenere a partire dal cammino parziale. Dopo aver trovato il primo giri di costo 5, che in questo esempio è il costo ottimale, potremo evitare di esplorare i sottoalberi di quei nodi in cui il limite inferiore è maggiore o uguale a 5. La figura mette in evidenza il fatto che usando la tecnica di branch and bound eviteremo di esplorare 7 sottoalberi, di cui 2 che partono dal livello 1 dell'albero, 1 che parte da livello 2 ed altri 4 che partono dal livello 3.

## 2.7 *Scelta dell'albero di backtracking*

La tecnica di backtracking, come abbiamo visto, si basa sulla costruzione di un albero che rappresenta tutte le possibili soluzioni. La costruzione di tale albero può essere fatta in vari modi ed è importante scegliere quello più utile per poter poi implementare il backtrack stesso. Consideriamo ad esempio il problema VertexCover. Ricordiamo il problema: dato un grafo $G = (V, E)$ e un intero $k$, vogliamo sapere se esiste un insieme di nodi $S \subseteq V$ di al massimo $k$ nodi tale che per ogni arco $(u, v) \in E$, risulta che almeno un nodo fra $u$ e $v$ appartiene a $S$. Un insieme $S$ con tale caratteristica è un *ricoprimento (degli archi) di G*, cioè un *vertex cover*. Dunque nel problema VertexCover le possibili soluzioni sono tutti i possibili sottoinsiemi di $V$ con al più $k$ nodi. Per semplicità consideriamo una variante del problema in cui ci chiediamo se esistono ricoprimenti di *esattamente k* nodi. Questo per far sì che solo le foglie dell'albero rappresentino soluzioni e non pure i nodi interni; ma non è importante per quello di cui stiamo discutendo. Come possiamo costruire un albero che rappresenta tali sottoinsiemi?

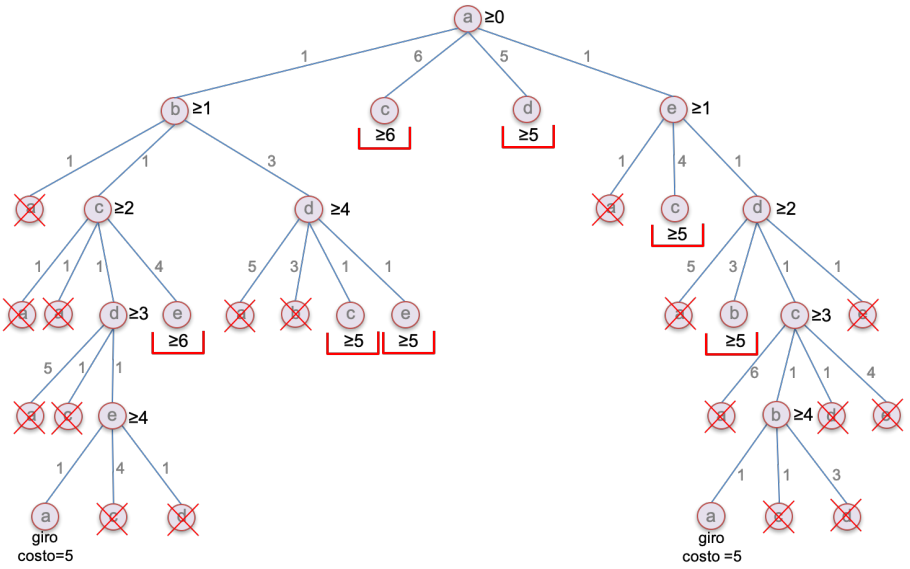



Figura 2.13: Albero dei percorsi branch-and-bound

Consideriamo questa idea: la radice dell'albero rappresenta l'insieme vuoto. Ogni livello dell'albero rappresenta l'inserimento di un nuovo nodo nell'insieme $S$. L'albero ha $k$ livelli. La radice ha quindi esattamente $n$ figli in quanto nell'insieme vuoto possiamo inserire un qualsiasi nodo. I nodi del primo livello invece avranno esattamente $n-1$ figli in quanto essi rappresentano un insieme di un nodo e quindi quel nodo non può essere inserito di nuovo. Analogamente i nodi del secondo livello avranno $n-2$ figli, ed in generale i nodi del livello $i$ avranno $n-i$ figli. Le foglie di questo albero rappresentano tutti i possibili sottoinsiemi di $k$ nodi.

Tuttavia, questa scelta non ci aiuta per il backtrack. Infatti per mettere in pratica la strategia di backtrack, quando ci troviamo in un nodo $a$ dell'albero di backtracking dobbiamo riuscire a dire se nel sottoalbero radicato in $a$ ci sono soluzioni al problema. Non è semplice farlo: di fatto significa risolvere un sottoproblema che consiste nello stabilire se con i nodi rimanenti è possibile ricoprire il sottografo formato dagli archi non ancora ricoperti.

Consideriamo ora quest'altra idea: la radice dell'albero rappresenta l'insieme $V$ mentre ogni livello rappresenta la cancellazione di un nodo dall'insieme precedente. Analogamente al caso precedente la radice avrà esattamente $n$ figli, in quanto possiamo eliminare uno qualsiasi dei nodi di $V$, mentre i nodi dei livelli successivi avranno meno figli, $n-i$ al livello $i$, in quanto ci sono meno nodi da poter eliminare. L'albero in questo caso avrà $n-k$ livelli perchè da un insieme di $n$ nodi vogliamo arrivare ad un insieme di $k$ nodi. Come nel caso precedente le foglie rappresentano tutti i sottoinsiemi di taglia $k$.

Consideriamo ad esempio il grafo nella Figura 2.14. La Figura 2.15 mostra l'albero.

La radice $\alpha$ corrisponde all'insieme di tutti i nodi che è banalmente un insieme ricoprente. Da questo insieme possiamo provare a togliere un nodo alla volta. Il primo livello corrisponderà a sottoinsiemi di 4 vertici del grafo. Ad esempio il nodo $\beta_1$ corrsiponde al sottoinsieme $\{b,c,d,e\}$ (abbiamo tolto il nodo $a$), il nodo $\beta_2$ corrsiponde

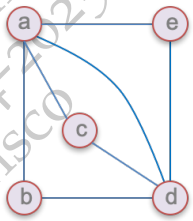



Figura 2.14

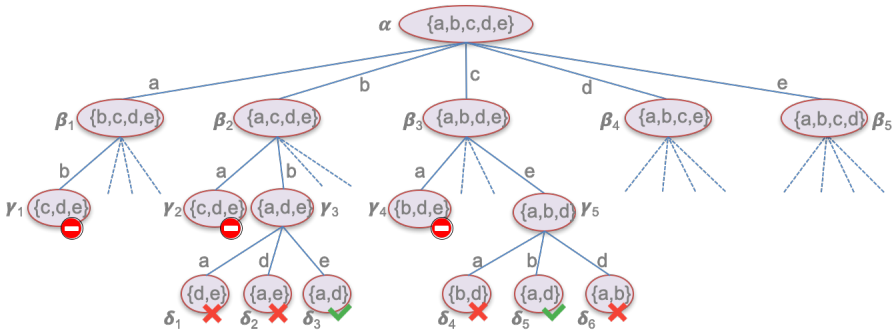



Figura 2.15: Albero di backtrack

al sottoinsieme $\{a, c, d, e\}$ (abbiamo tolto il nodo $b$), e così via.

Ora però se consideriamo un nodo $x$ interno all'albero abbiamo un modo semplice per dire che nel sottoalbero radicato in $x$ non ci sono ricoprimenti: se esiste un arco $e = (u, v)$ per il quale né $u$ né $v$ sono nell'insieme rappresentato da $x$, allora non è possibile ricoprire $e$. Quindi possiamo fermare al nodo $a$ l'esplorazione dell'albero senza scendere nel sottoalbero in esso radicato. Ad esempio il nodo $\gamma_1$ rappresenta l'insieme $\{c, d, e\}$. Questo insieme non è ricoprente in quanto l'arco $(a, b)$ non è ricoperto; pertanto nessun sottoinsieme di $\{c, d, e\}$ può ricoprire il grafo. Dunque l'esplorazione del sottoalbero radicato in $\gamma_1$ è inutile.

Notiamo però che in questo albero ci sono sottoinsiemi che sono rappresentati da più nodi. Ad esempio i nodi $\gamma_1$ e $\gamma_2$ rappresentano entrambi il sottoinsieme $\{c, d, e\}$. Questo significa che nell'albero ci sono dei percorsi ridondanti che ovviamente sono un problema: la visita dell'albero richiede più tempo del necessario.

Una rappresentazione che evita questo problema è la seguente: ogni livello dell'albero rappresenta un vertice del grafo e ogni nodo dell'albero ha due figli che rappresentano, rispettivamente, la non appartenenza e l'appartenza all'insieme del vertice. In pratica è la stessa che abbiamo usato per il problema Sat, con i vertici al posto delle variabili ed i valori booleani che indicano l'appartenza all'insieme. La Figura 2.16 mostra l'albero.

Le 32 foglie di questo albero corrispondono ai $2^5 = 32$ possibili sottoinsiemi che si possono avere con i 5 vertici del grafo. Ogni nodo interno rappresenta un insieme di sottoinsiemi con caratteristiche in comune. Ad esempio il nodo 4 rappresenta tutti i sottoinsiemi che non contengono né il vertice $a$ né il vertice $b$. Quindi, come osservato in precedenza, nessuno dei sottoinsiemi rappresentati dal nodo 4 può essere un insieme ricoprente. Pertanto è inutile visitare il sottoalbero radicato in 4.

Inoltre possiamo osservare che poichè l'insieme ricoprente può contenere al massimo $k$ nodi non possiamo usare più di $k$ volte l'arco che collega un padre al suo figlio destro. Dunque ogni volta che raggiungiamo un nodo usando $k$ archi "destri" ci fermiamo. Ad esempio per $k = 2$ avremmo l'albero di backtrack mostrato nella Figura 2.17.:

In conclusione, il modo in cui si costruisce l'albero di backtracking è importante in quanto determina l'efficacia della strategia. Ovviamente la cosa dipende dal problema che si considera e quindi le specifiche scelte vanno fatte caso per caso.

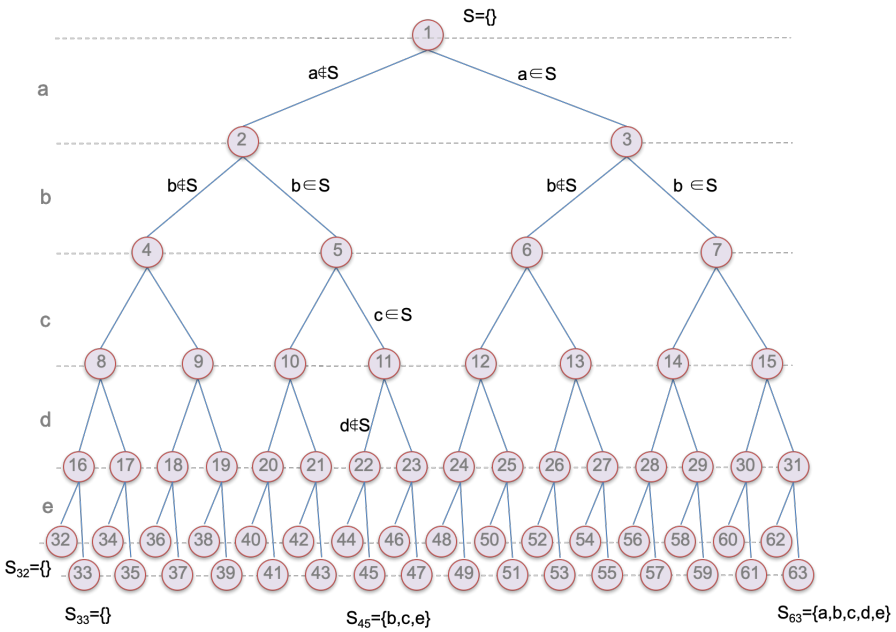



Figura 2.16: Albero completo

## 2.8 *Sfruttare le caratteristiche del problema*

In alcuni casi è possibile sfruttare le specifiche caratteristiche del problema o dell'input al problema per ottenere dei miglioramenti. Nel seguito vedremo alcuni esempi.

### 2.8.1 *Vertex Cover*

Riprendiamo nuovamente il problema VertexCover. Poichè il problema è difficile, la complessità della soluzione è una funzione esponenziale della taglia dell'input. Tuttavia in tale funzione gioca un ruolo fondamentale anche il parametro $k$. Usando un approccio a forza bruta, possiamo risolvere il problema semplicemente considerando tutti i possibili sottoinsiemi di $V$ di taglia $k$. I sottoinsiemi di taglia $k$ sono esattamente $\binom{n}{k}$ e controllare se un sottoinsieme è un ricoprimento degli archi richiede tempo $O(kn)$. Pertanto l'algoritmo a forza bruta richiede tempo $O(kn\binom{n}{k}) = O(kn^{k+1})$. Se $k$ è fissato la complessità della soluzione a forza bruta è polinomiale! In altre parole l'intrattabilità del problema si manifesta solo quando $k$ cresce con $n$. Quindi se dobbiamo risolvere il problema per $k = 2$, anche l'algoritmo a forza bruta è "efficiente": la sua complessità è $O(n^3)$. Discorso analogo per altri valori di $k$; ovviamente più $k$ diventa grande, più l'algoritmo diventa inefficiente.

Infatti, anche per valori relativamente piccoli di $k$ un tempo di esecuzione $O(kn^{k+1})$ può essere molto grande. Ad esempio per $n = 1000$ e $k = 10$, ignorando le costanti nascoste nella notazione asintotica e assumendo che il computer su cui gira il programma possa eseguire un milione di istruzioni di alto livello per ogni secondo, occorrerebbero $10^{28}$ secondi[3]. E questo nonostante il valore di $k$ sia relativamente piccolo, caso per il quale avevamo detto che il problema poteva essere trattabile. Tuttavia se al

[3] 1 secolo dura meno di $10^{10}$ secondi; $10^{28}$ è un tempo spaventosamente grande, ordini di magnitudine più grande dell'età dell'universo

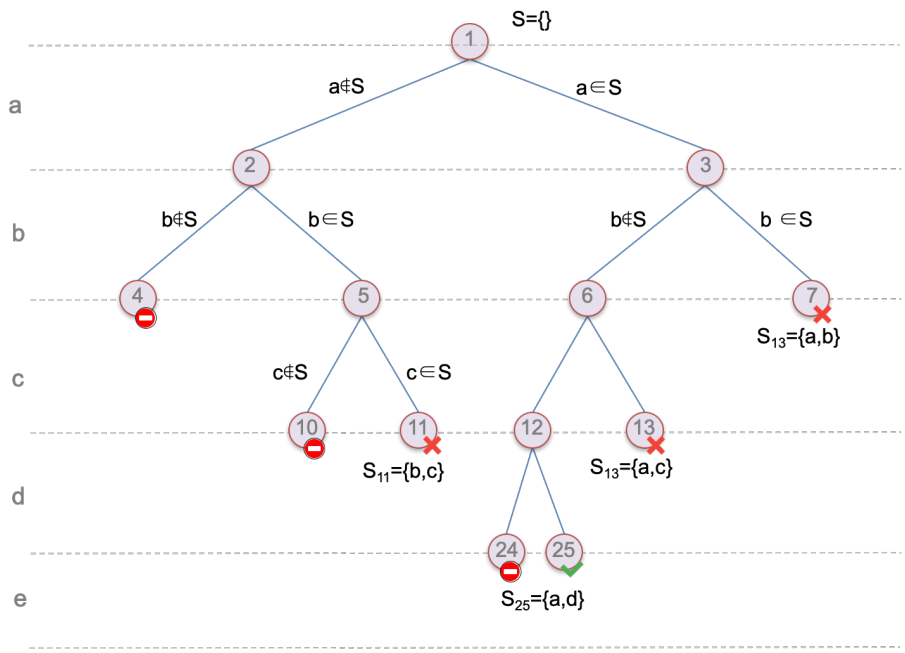



Figura 2.17: Albero di backtrack per $k = 2$

posto della forza bruta usiamo un approccio più scaltro, possiamo ottenere un algoritmo che, sebbene, ovviamente, esponenziale come quello a forza bruta, risulta più efficiente per valori piccoli di $k$. Vedremo un algoritmo il cui tempo di esecuzione è $O(2^k kn)$. Si noti come in questo caso l'esponenzialità del tempo di esecuzione sia dovuta a $k$ che compare come esponente di una costante; nell'approccio a forza bruta $k$ compare come esponente di $n$. Questa differenza è sostanziale. Sebbene per $k$ grandi il tempo di esecuzione può comunque essere incredibilmente grande, per $k$ piccoli, anche al crescere di $n$ il tempo di esecuzione può essere ragionevole. Ad esempio, per il caso considerato in precedenza, cioè $n = 1000$ e $k = 10$, usando lo stesso computer, il tempo di esecuzione diventa di circa 10 secondi. Al crescere di $k$ ovviamente il tempo diventa comunque molto grande; tuttavia l'aver spostato la dipendenza esponenziale da $k$ fuori dall'esponente di $n$ ha creato una situazione nettamente migliore dal punto di vista pratico.

Vediamo l'algoritmo. Iniziamo con l'osservare che se un grafo ha un insieme ricoprente piccolo non può avere molti archi. Più formalmente abbiamo il seguente lemma.

**Lemma 2.8.1** *Se $G = (V, E)$ ha un insieme ricoprente di taglia $k$ allora $|E| \leq k(n-1)$, dove $n = |G|$ è il numero di nodi del grafo.*

DIMOSTRAZIONE. Sia $S$ un insieme ricoprente di taglia $k$. Ogni nodo di $S$ può "ricoprire" al massimo $n-1$ archi (ogni nodo può essere collegato al massimo a tutti gli altri $n-1$ nodi). Quindi $S$ può ricoprire al massimo $k(n-1)$ archi. D'altra parte per definizione di insieme ricoprente si ha che tutti gli archi sono ricoperti da $S$, quindi il numero totale di archi non può essere superiore a $k(n-1)$. □

Il Lemma 2.8.1 permette di assumere che il grafo da esaminare ha al massimo $k(n-1)$ archi. Infatti possiamo controllare la cardinalità di $E$ e se è maggiore di $k(n-1)$ risolviamo il problema immediatamente rispondendo che $G$ non ha un insieme ricoprente di taglia $k$.

L'idea alla base dell'algoritmo che descriveremo fra poco è molto semplice ed elegante. Consideriamo un arco $e = (u, v)$ di $G$. Sia $S$ un qualsiasi insieme ricoprente $S$ di taglia $k$; almeno uno fra $u$ e $v$ deve appartenere a $S$. Supponiamo che $u \in S$. Allora, se cancelliamo $u$ e tutti gli archi incidenti su $u$, deve essere possibile ricoprire gli archi rimanenti con un insieme di al massimo $k-1$ nodi. Cioè, se consideriamo il grafo $G - \{u\}$, ottenuto da $G$ cancellando $u$ e tutti gli archi ad esso incidenti, si ha che deve esistere un insieme ricoprente di taglia al massimo $k-1$. Stesso ragionamento può essere fatto nel caso in cui sia $v$ ad appartenere a $S$. Possiamo formalizzare quanto appena detto con il seguente lemma.

**Lemma 2.8.2** *Sia $e = (u, v)$ un arco di $G$. Il grafo $G$ ha un insieme ricoprente di taglia $k$ se e solo se almeno uno dei grafi $G \setminus \{u\}$ e $G \setminus \{v\}$ ha un insieme ricoprente di taglia $k-1$.*

Dimostrazione. Assumiamo che $G$ abbia un insieme ricoprente $S$ di taglia $k$. Allora $S$ deve contenere almeno uno fra $u$ e $v$; supponiamo che sia $u$ ad appartenere a $S$. Si consideri l'insieme $S' = S - \{u\}$; tale insieme è un insieme ricoprente per $G - \{u\}$; infatti tutti gli archi non incidenti ad $u$ sono ricoperti da nodi di $S'$. Inoltre $|S'| = k-1$. Se $u$ non appartiene a $S$ allora $v$ appartiene ad $S$; in tal caso $S'' = S - \{v\}$ è un insieme ricoprente per $G - \{v\}$.

Viceversa, supponiamo che almeno uno dei grafi $G - \{u\}$ e $G - \{v\}$ abbia un insieme ricoprente $S$ di taglia $k-1$. Supponiamo sia $G - \{u\}$ (se fosse $G - \{v\}$ basterà sostituire $u$ con $v$). Per definizione, $S$ ricopre tutti gli archi di $G - \{u\}$; se consideriamo $S' = S \cup \{u\}$ si ha che $S'$ ricopre $G$ in quanto gli archi che $G$ ha in più rispetto a $G - \{u\}$ sono quelli incidenti su $u$. □

Il Lemma 2.8.2 suggerisce in modo naturale il seguente algoritmo ricorsivo.

**Algorithm 3:** RecursiveVC($G, k$)

**if** $G = (V, E)$ *non ha archi (caso base)* **then**
  restitusci l'insieme vuoto $S = \{\}$
**if** $|E| > k(n-1)$ **then**
  $G$ non ha un insieme ricoprente di taglia $\leq k$
**else**
  Sia $e = (u, v)$ un arco di $G$.
  Risolvi RecursiveVC($G - \{u\}, k-1$) e RecursiveVC($G - \{v\}, k-1$)
  **if** *nessuno dei due sottoproblemi ha soluzione* **then**
    $G$ non ha un insieme ricoprente di taglia $\leq k$
  **else**
    Uno dei due, diciamo $G - \{u\}$, ha un ricoprimento $T$
    $T \cup \{u\}$ è un ricoprimento di taglia $\leq k$ di $G$
    (Se fosse stato $G - \{v\}$, il ricoprimento sarebbe stato $T \cup \{v\}$)

Rimane da vedere la complessità dell'algoritmo. Ogni chiamata dell'algoritmo con parametro $k$ produce due chiamate ricorsive con parametro $k-1$. Pertanto ci saranno in totale $2^{k+1}$ chiamate ricorsive e in ogni chiamata ricorsiva il tempo speso è di $O(kn)$. Possiamo analizzare formalmente il tempo necessario usando una relazione di ricorrenza. Se denotiamo con $T(n,k)$ il tempo necessario a risolvere il problema su un grafo di $n$ nodi per una taglia dell'insieme ricoprente di $k$, avremo che esiste una costante $c$ per la quale

$$\begin{aligned} T(n,k) &\leq 2T(n-1,k-1) + ckn, \\ T(n,1) &\leq cn. \end{aligned}$$

Possiamo procedere per induzione su $k$ e provare che $T(n,k) \leq c \cdot 2^k kn$. Per $k = 1$ il limite è rispettato; assumiamo che sia vero per $k-1$ e proviamo che è vero per $k$:

$$\begin{aligned} T(n,k) &\leq 2T(n-1,k-1) + ckn, \\ &\leq 2c \cdot 2^{k-1}(k-1)n + ckn, \\ &= c \cdot 2^k kn - c \cdot 2^k n + ckn, \\ &\leq c \cdot 2^k kn. \end{aligned}$$

Riassumendo, l'algoritmo RecursiveVC fornisce un buon miglioramento dal punto di vista pratico. Tuttavia rimane esponenziale e nessun algoritmo esponenziale può rimanere efficiente al crescere dei parametri. Ad esempio per $k = 40$ anche l'algoritmo RecursiveVC, fatto girare sullo stesso computer usato in precedenza, richiederà un cospicuo numero di anni per terminare.

### 2.8.2 *Independent Set*

Consideriamo adesso un altro problema per illustrare un caso in cui si riesce a sfruttare un vincolo sulla struttura dell'input per "ammorbidire" l'intrattabilità del problema. Il problema che consideriamo è IndependentSet che, come abbiamo visto nel capitolo precedente, è un problema $\mathcal{NP}$-completo. Ricordiamo il problema: dato un grafo $G = (V, E)$ e un intero $k$, vogliamo sapere se esiste un insieme di nodi $S \subseteq V$ di almeno $k$ nodi tale che non ci sia nessun arco fra i nodi di $S$.

In questo caso vogliamo sfruttare una restrizione sull'input. Al posto di considerare il problema su grafi qualsiasi restringiamo l'attenzione a grafi che sono alberi. La semplicità della struttura di un albero rispetto al caso generale di un grafo qualsiasi permette di fornire un algoritmo efficiente. Sebbene utilizzeremo il problema IndependentSet come esempio, l'approccio può essere sfruttato in generale; ovviamente non è detto che funzioni per qualsiasi problema.

Vediamo adesso l'algoritmo. Iniziamo con la seguente semplice osservazione: ogni arco $e = (u, v)$ del grafo $G$ si ha che un qualsiasi insieme indipendente $S$ di $G$ può contenere al massimo uno fra $u$ e $v$. Vogliamo sfruttare questa osservazione per progettare un algoritmo greedy che possa scegliere, dato un arco $e$, di inserire nell'insieme indipendente uno dei due nodi su cui $e$ incide.

Per prendere la decisione greedy sfrutteremo il fatto che il grafo sul quale operiamo è un albero. Poichè il grafo è un albero esiste almeno una foglia. Sia $e = (u, v)$ un

arco tale che $v$ è una foglia. Se includiamo $v$ nell'insieme indipendente non potremo inserire $u$; ma $u$ è l'unico altro nodo che non potremo inserire nell'insieme in quanto $v$ è collegato solo ad $u$. Analogamente, se inseriamo $u$ nell'insieme indipendente non potremo inserire $v$; in questo caso però potrebbero esserci altri nodi (l'eventuale padre di $u$ e eventuali altri figli di $u$) che non potranno essere inseriti nell'insieme indipendente. Quindi inserire $v$ nell'insieme indipendente non previene la possibilità di costruire un insieme indipendente massimale. Questo è l'ingrediente chiave per avere un algoritmo greedy che trova l'ottimo. Più formalmente abbiamo il seguente lemma.

**Lemma 2.8.3** *Sia $T = (V, E)$ un albero e $v$ una foglia di $T$. Allora esiste un insieme indipendente massimale che contiene $v$.*

DIMOSTRAZIONE. Sia $S$ un insieme indipendente massimale e sia $e = (u, v)$ l'unico arco incidente su $v$. Poichè $S$ è massimale almeno uno fra $u$ e $v$ deve essere contenuto in $S$; se così non fosse potremmo costruire l'insieme indipendente $S \cup \{v\}$ di cardinalità $|S| + 1$ contraddicendo il fatto che $S$ è massimale. Se è proprio $v$ ad appartenere ad $S$ il lemma è soddisfatto. Se invece è $u$ ad appartenere ad $S$ possiamo costruire l'insieme indipendente $S' = S - \{u\} \cup \{v\}$, cioè togliere $u$ ed inserire $v$, ottenendo un nuovo insieme indipendente che è di taglia massimale, in quanto $|S'| = |S|$, e contiene $v$. □

Il lemma precedente suggerisce un semplice algoritmo greedy, che individua le foglie dell'albero e le inserisce nell'insieme indipendente e ripete l'operazione dopo aver cancellato i nodi (e gli archi ad essi incidenti) collegati al nodo inserito. Si noti che poichè la cancellazione di archi può disconettere l'albero, durante l'esecuzione dell'algorimo potremmo ritrovarci con una foresta. Quanto detto riguardo un nodo foglia continua ovviamente a valere; semplicemente avremo vari alberi che potremo trattare singolarmente. Anzi avremo più foglie da scegliere, almeno una per ogni albero. Pertanto l'algoritmo opererà su una foresta, ma non è necessario nessun accorgimento particolare.

**Algorithm 4:** GREEDYIS($F$)

```
S={} è l'insieme indipendente inzialmente vuoto
while F ha almeno un arco do
    Sia e = (u, v) un arco di F tale che v è una foglia.
    S = S ∪ {v}
    Cancella da F i nodi u, v e tutti gli archi ad essi incidenti
Aggiungi a S i nodi restanti return S
```

La correttezza dell'algoritmo deriva direttamente dal Lemma 2.8.3. Il lemma garantisce che per ogni foglia $v$ esiste un insieme indipendente $S$ massimale che la contiene. Chiaramente $S$ non potrà contenere $u$, per cui il fatto che l'algoritmo lo cancelli non è un problema. Inoltre non è un problema nemmeno cancellare gli archi incidenti su $u$: infatti se da un lato la cancellazione di archi può solo portare ad insiemi più grandi, dall'altro, non c'è il rischio che questi archi cancellati possano far sì che in $S$ vengano inseriti nodi fra i quali esiste un arco, in quanto $u$ stesso non è in $S$.

Concludiamo con l'osservare che la scelta greedy permessa dalla struttura ad albero in realtà può funzionare anche in un grafo generico a patto che venga soddisfatta

la proprietà cruciale garantita da una foglia: se $v$ è una foglia collegata al padre da $e = (u, v)$ allora $u$ è l'unico "vicino" di $v$. Quindi anche se il grafo non è un albero ma esiste un nodo $v$ che ha un unico vicino $u$, allora possiamo inserire $v$ nell'insieme indipendente che stiamo costruendo in quanto siamo sicuri che tale scelta greedy non preclude la possibilità di costruire un insieme massimale, cioè esiste un insieme massimale che include $v$. Quindi l'approccio greedy descritto per gli alberi può essere usato parzialmente anche su grafi generici quando si crea la situazione di un nodo "foglia".

Infine osserviamo che affinchè il tutto funzioni è necessario che l'implementazione dell'algoritmo permetta di trovare in modo efficiente ad ogni iterazione una foglia; ciò non è difficile da fare ma è chiaramente fondamentale.

### *2.8.3 Insiemi indipendenti pesati*

Consideriamo adesso una generalizzazione del problema IndependentSet: WeightedIS, nel quale abbiamo dei pesi per ogni nodo e l'insieme indipendente non deve più massimizzare il numero di nodi ma la somma dei pesi dei nodi. Più formalmente, abbiamo un albero $T = (V, E)$ ed anche una funzione che associa ad ogni nodo $v \in V$ il peso $w_v$. Vogliamo trovare un insieme indipendente $S$ nel grafo $T$, tale che il peso totale di $S$, $w(S) = \sum_{v \in V} w_v$, sia quanto più grande possibile.

Osserviamo innanzitutto che la presenza dei pesi rende la decisione sull'inserimento dei nodi più difficile. Infatti se riconsideriamo l'approccio usato per l'algoritmo greedy ci accorgiamo immediatamente che non funziona più: mentre con l'assenza di pesi inserire il nodo foglia non precludeva la possibilità di trovare comunque un insieme massimale, con i pesi è facile verificare che ciò non è più vero. Consideriamo un arco $e = (u, v)$ in cui $v$ è una foglia. Se includiamo $v$ non possiamo includere $u$. Questo non è un problema se $w_u \leq w_v$. Se invece $w_u > w_v$ escludendo $v$ ci potremmo ritrovare con un insieme indipendente con peso non massimale. La presenza dei pesi rende la decisione sul se includere o meno un nodo, più difficile da prendere usando solo informazioni locali senza guardare al resto del grafo.

Tuttavia anche con la presenza di pesi possiamo comunque dire qualcosa se sono coinvolte delle foglie: se un nodo $u$ ha molti figli $v_1, v_2, \ldots$ che sono foglie, allora dovremo prendere la stessa decisione per tutte le foglie. Cioè o le includiamo tutte o le escludiamo tutte. In altre parole per il sottoalbero radicato in $u$ ci sono solo due possibili scelte: o includiamo $u$, e quindi escludiamo tutti i suoi figli, oppure escludiamo $u$, ed in questo caso dobbiamo includere tutti i sui figli.

Questa osservazione permette di sviluppare un algoritmo di programmazione dinamica. Ricordiamo che la programmazione dinamica è utile quando il problema può essere scomposto in sottoproblemi e le soluzioni dei sottoproblemi permettono di costruire la soluzione al problema originale[4]. Per il prolema WeightedIS, i sottoproblemi possono essere definiti nel seguente modo. Possiamo fissare arbitrariamente una radice dell'albero, diciamo il nodo $r$. Fissare la radice significa che tutti gli archi saranno orientati per "allontanarsi" dalla radice, quindi per ogni nodo $u$, il padre di $u$, $p(u)$, è il nodo che precede $u$ nel cammino dalla radice a $u$. Gli altri nodi collegati da un arco ad

[4] Inoltre, l'efficienza dell'algoritmo è migliore quando molti dei sottoproblemi si ripresentano spesso per cui è sufficiente risolverli solo una volta risparmiando così tempo quando si ripresentano; tutavia questo apsetto non è fondamentale nella discussione che stiamo facendo.

a $u$ saranno figli di $u$ e li denoteremo nel loro insieme con $children(u)$. Il nodo $u$ e tutti i suoi discendenti formano un sottoalbero $T_u$ radicato nel nodo $u$.

I sottoalberi rappresentano i sottoproblemi. L'albero $T_r$ è il problema originale. Se $u \neq r$ è una foglia, allora $T_u$ ha un solo nodo (il nodo $u$). Se invece $u$ ha figli che sono foglie allora il sottoalbero $T_u$ ha la forma che ci permette di applicare l'osservazione fatta in precedenza.

Quindi, per risolvere il problema usando la programmazione dinamica, operiamo partendo dalle foglie e risalendo nell'albero. Per ogni nodo $u$, abbiamo bisogno di risolvere il sottoproblema rappresentato dall'albero $T_u$ dopo aver risolto i sottoproblemi radicati nei figli di $u$. Per costruire un insieme indipendente $S$ di peso massimo per $T_u$, dobbiamo cosiderare i due possibili casi: includiamo $u$ in $S$ oppure non lo includiamo. Nel caso in cui non includiamo $u$, per i figli di $u$ abbiamo la possibilità di includerli o meno, in funzione di se conviene oppure no. Questa osservazioni suggeriscono la definizione di due sottoproblemi per ogni sottoalbero $T_u$: il sottoproblema $\textsc{opt}_{inc}(u)$, che denota il peso massimo di un insieme indipendente per $T_u$ che *include* $u$ e il sottoproblema $\textsc{opt}_{esc}(u)$, che denota il peso massimo di un insieme indipendente per $T_u$ che *esclude* $u$.

Vediamo ora come calcolare $\textsc{opt}_{inc}$ e $\textsc{opt}_{esc}$. Per una foglia $u$ si ha che $\textsc{opt}_{inc}(u) = w_u$ e $\textsc{opt}_{esc}(u) = 0$. Per tutti gli altri nodi si ha la seguente relazione di ricorrenza

$$\begin{aligned} \textsc{opt}_{inc}(u) &= w_u + \sum_{v \in children(u)} \textsc{opt}_{esc}(v) \\ \textsc{opt}_{esc}(u) &= \sum_{v \in children(u)} \max\{\textsc{opt}_{inc}(v), \textsc{opt}_{esc}(v)\}. \end{aligned}$$

La relazione di ricorrenza definita poc'anzi si traduce immediatamente in un algoritmo di programmazione dinamica. L'unica accortenza che dobbiamo avere è quella di visitare l'albero in modo tale che quando calcoliamo $\textsc{opt}_{inc}$ e $\textsc{opt}_{esc}$ per un nodo $u$ dobbiamo aver già calcolato i corrispondenti valori per i figli di $u$. Per garantire questo fatto è sufficiente visitare l'albero con una visita post-order.

**Algorithm 5:** DynProWIS($T$)

Scegli un nodo $r \in T$ come radice.
**for** *tutti i nodi u di* $T_r$ *in post-order* **do**
  **if** *u è una foglia* **then**
    $\textsc{opt}_{inc}(u) = w_u$
    $\textsc{opt}_{esc}(u) = 0$
  **else**
    $\textsc{opt}_{inc}(u) = w_u + \sum_{v \in children(u)} \textsc{opt}_{esc}(v)$
    $\textsc{opt}_{esc}(u) = \sum_{v \in children(u)} \max\{\textsc{opt}_{inc}(v), \textsc{opt}_{esc}(v)\}$
**return** $\max\{\textsc{opt}_{inc}(r), \textsc{opt}_{esc}(r)\}$

Come per tutti gli algoritmi di programmazione dinamica il mero calcolo del valore della soluzione ottima restituito da DynProWIS($T$) non fornisce anche una soluzione che ha quel valore. Ma è facile modificare l'algoritmo memorizzando ad ogni passo

l'informazione sul se il nodo $u$ è stato inserito o meno nell'insieme e quindi, una volta ottenuto il valore finale, riscostruire una soluzione con quel valore.

## 2.9 *Algoritmi di ricerca locale*

Un'altra tecnica di carattere generale che può essere utilizzata per affrontare problemi di ottimizzazione difficili caratterizzati da uno spazio delle possibili soluzioni enorme, sono gli algoritmi di ricerca (dell'ottimo) locale. L'idea alla base di questa tecnica è la seguente: partendo da una soluzione qualsiasi si cerca di migliorarla facendo pochi cambiamenti alla soluzione stessa. A tal fine si definiscono delle regole in base alle quali due soluzioni sono considerate vicine (cioè passare dall'una all'altra richiede pochi cambiamenti). In base a tali regoli, per ogni soluzione $s$ si può definire il *vicinato* di $s$, come tutte quelle soluzioni $s'$ che sono vicine ad $s$. Un generico algoritmo di ricerca locale può essere descritto tramite il seguente pseudocodice:

**Algorithm 6:** GENERICLS

Sia $s$ una soluzione qualsiasi
**while** *Esiste* $s'$ *vicino di* $s$ *con costo*$(s')$ *migliore di costo*$(s)$ **do**
  $s \leftarrow s'$.
**return** $s$

Come si può intuire dallo pseudocodice, un algoritmo di ricerca locale è estremamente semplice: basta definire una regola di vicinanza fra le soluzioni (che ovviamente dipende strettamente dal problema) ed implementare un ciclo in cui ad ogni iterazione si cerca di migliorare la soluzione attuale cercandone una migliore fra quelle nel vicinato. Dunque un grosso vantaggio è la semplicità di implementazione dell'algoritmo. Il rovescio della medaglia è che per questo tipo di approccio è spesso difficile fornire delle garanzie sulla soluzione che l'algoritmo produrrà. Infatti l'approccio viene detto di "ricerca locale" in quanto fornisce una soluzione che è ottima solo localmente e quindi non riusciamo a dire se lo è anche globalmente o più genericamente quanto si discosta dall'ottimo globale. Se siamo fortunati, un algoritmo di ricerca locale può anche trovare una soluzione ottima globalmente oppure una soluzione il cui valore non si discosta molto dalla soluzione ottima globale. Comunque, ci sono molti casi in cui gli algoritmi di ricerca locale si sono dimostrati una valida alternativa.

Un ruolo fondamentale è svolto dalla relazione di vicinanza: tale relazione è definita arbitrariamente e può essere sia molto restrittiva, limitando la grandezza del vicinato e quindi rendendo l'algoritmo più efficiente, sia molto permissiva, aumentando la grandezza del vicinato e quindi rendendo l'algoritmo meno efficiente. Non dovrebbe sorprendere che più è restrittiva la relazione di vicinanza e meno ci si può aspettare riguardo la bontà della soluzione: cercare una soluzione migliore su insiemi più grandi implica una probabilità maggiore di trovare un ottimo locale migliore.

Procediamo con un esempio, e, a tal fine, riprendiamo il problema VERTEXCOVER: Dato un grafo $G = (V, E)$ un ricoprimeno è un sottoinsieme $S \subseteq V$ tale che ogni arco di $E$ ha almeno un vertice in $S$; vogliamo trovare un ricoprimento che abbia il minimo numero di nodi. Per questo problema lo spazio $\mathcal{S}$ delle (potenziali) soluzioni è dato da tutti i sottoinsiemi $S \subseteq V$ che formano un ricoprimento. Osserviamo che poichè a

priori non sappiamo se un particolare insieme è o meno un insieme ricoprente, di fatto dovremo esaminare tutti i possibili insiemi di vertici.

Il costo di una soluzione $S$ è la cardinalità della soluzione stessa: $costo(S) = |S|$. Dobbiamo adesso definire una relazione di "vicinanza" per poter esplorare $\mathcal{S}$ passando da una soluzione ad un'altra ad essa vicina. Una possibile relazione è la seguente: due soluzioni sono vicine se una può essere ottenuta dall'altra aggiungendo o rimuovendo un solo vertice; quindi data una soluzione $S$ tutte le soluzioni ad essa vicine sono quelle che possiamo ottenere o rimuovendo un vertice di $S$, o aggiungendo un vertice di $V - S$ ad $S$. Dunque, per ogni $S$ ci sono esattamente $n$ insiemi di vertici vicini ad $S$, e quindi al massimo $n$ insiemi che effettivamente sono dei ricoprimenti.

Pertanto data una soluzione $S$, in tempo proporzionale ad $n$ riusciamo a considerare tutte le soluzioni vicine ad $S$ (a tale tempo dovremo aggiungere quello necessario per controllare se un insieme è ricoprente o meno). Dunque l'algoritmo di ricerca locale per VertexCover procede nel seguente modo: data una soluzione $S$, con costo $|S|$, considera tutti le soluzioni vicine ad $S$ e se ne trova una $S'$ con costo minore di $S$ si "sposta" nella nuova soluzione $S'$. Si noti che $V$ è sicuramente un insieme ricoprente, quindi è un buon punto di partenza.

**Algorithm 7:** VertexCoverLS

$S \leftarrow V$
**while** *Esiste* $S'$ *vicino di* $S$ *con* $|S'| < |S|$ **do**
  $S \leftarrow S'$.
**return** $S$

L'algoritmo VertexCoverLS procede passando da una soluzione ad una con costo minore, finchè tale passaggio è possibile. Abbiamo già detto, però, che la soluzione restituita dall'algoritmo è una soluzione ottima *localmente*, ma potrebbe non essere ottima *globalmente*. Questo è dovuto al modo in cui lo spazio delle soluzioni è stato esplorato. Possiamo fare un analogo con la ricerca del minimo di una funzione continua di una variabile in cui il valore della soluzione $x$ è $y = f(x)$ e le soluzioni vicine ad $x$ sono quelle in un intorno $[x - \epsilon, x + \epsilon]$ di $x$. Se la funzione è crescente l'algoritmo si sposterà nell'intorno sinistro, se invece è decrescente si sposterà nell'intorno destro. In un punto di minimo (locale) l'algoritmo si ferma e restituisce il punto di minimo locale, che potrebbe non essere il minimo della funzione.

Le Figure 2.18 e 2.19 mostrano dei casi dove l'algoritmo di ricerca locale si ferma in un punto di minimo locale; è possibile, come nel caso della Figura 2.20 che l'algoritmo di ricerca locale trovi il minimo globale, quando il minimo locale trovato corrisponde al minimo globale.

Nel caso di una funzione di una variabile è facile visualizzare il "percorso" seguito dall'algoritmo, come è stato mostrato nelle Figure 2.18-2.20. Per problemi reali il percorso seguito è più difficile da visualizzare ma il concetto di base è lo stesso. Per il problema VertexCover, ad esempio, dovremmo visualizzare tutti gli insiemi ricoprenti e quindi tracciare il percorso fatto. Ovviamente quali sono esattamente gli insiemi ricoprenti dipende dal grafo di partenza. Consideriamo un caso semplice in cui il grafo non ha archi cioè $E = \{\}$. In questo caso un qualunque sottoinsieme di vertici è ricoprente, incluso l'insieme vuoto, e l'algoritmo VertexCoverLS, partendo da $S = V$,

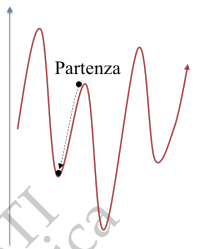



Figura 2.18: La ricerca termina in un minimo locale.

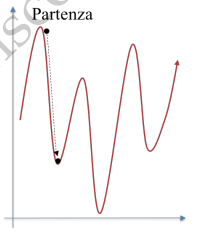



Figura 2.19: La ricerca termina in un minimo locale.

rimuoverà un vertice in ogni iterazione e, dopo $n$ iterazioni, restituirà come soluzione l'insieme vuoto che è la soluzione ottima (globale).

Se invece il grafo $G$ è un grafo a stella come mostrato nella Figura 2.21, allora il ricoprimento ottimo è l'insieme che contiene il solo nodo centrale $v_1$ della stella. In questo caso l'algoritmo potrebbe non restituire tale insieme ma, invece, restituire il ricoprimento $\{v_2, \ldots, v_n\}$. Questo succede quando $v_1$ viene scelto nella prima iterazione come nodo da cancellare: a quel punto l'algoritmo non può cancellare nessun altro nodo in quanto non avrebbe più un insieme ricoprente.

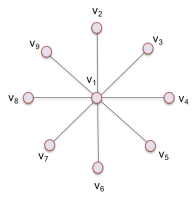



Figura 2.21: Un grafo a stella.

Un altro caso sufficientemente semplice da analizzare è quello di un grafo in cui i nodi sono uniti in un unico cammino, cioè gli archi sono $E = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$. Un insieme minimo ricoprente ottimale è l'insieme che contiene i nodi con indice pari $\{v_2, v_4, \ldots\}$. In questo caso però l'algortitmo VertexCoverLS potrebbe restiture vari minimi locali. Ad esempio partendo dall'insieme ricoprente $\{v_2, v_3, v_5, v_6, v_8, v_9, \ldots\}$, cioè dall'insieme di vertici meno i vertici con indice $1 + 3k$, $k = 0, 1, \ldots$, non possiamo togliere nessun nodo, quindi questo insieme ricoprente è un minimo locale; tuttavia ha molti più nodi rispetto all'insieme ricoprente ottimale.

Un miglioramento che si può apportare alla strategia generale per dimunuire la probabilità di bloccarsi in un minimo locale è data dalla strategia detta di *simulated annealing*[5]. Un algoritmo di simulated annealing è una ricerca locale nella quale si da la possibilità, con una certa probabilità, di passare anche ad una soluzione peggiore, nella speranza che da questa soluzione peggiore poi si possa andare verso soluzioni migliori. Nel descrivere lo pseudocodice assumeremo di avere a che fare con un problema di minimizzazione (per problemi di massimizzazione basterà invertire la logica di confronto delle soluzioni).

[5] Il nome, ricottura simulata, deriva dal fatto che l'idea si base sul processo metallurgico di fusione in cui i metalli vengono raffreddati in maniera graduale.

**Algorithm 8:** SimulatedAnnealing

Sia $s$ una soluzione qualsiasi
**repeat**
  Scegli in modo casuale una soluzione $s'$ vicino a $s$
  $\Delta \leftarrow costo(s') - costo(s)$
  **if** $\Delta < 0$ **then**
    $s \leftarrow s'$
  **else**
    $s \leftarrow s'$ con probabilità $e^{-\Delta/T}$
**until** *$s$ non viene cambiata*
**return** $s$

Il parametro $T$, che per analogia con il processo metallurgico al quale è ispirata la strategia viene detto *temperatura*, svolge un ruolo fondamentale. Se $T = 0$ allora la probabilità di sostituire $s$ con una soluzione peggiore $s'$ è 0 e quindi l'algoritmo si riduce alla strategia generale che permette di spostarsi solo in soluzioni migliori. Se $T$ invece è grande, allora la probabilità di passare a soluzioni peggiori aumenta (per $T \to \infty$ la probabilità tende a 1).

Osserviamo che la probabilità di passare a soluzioni peggiori serve per poter "sfuggire" a minimi locali. Tuttavia può provocare comportamenti poco efficienti. Si consideri

di nuovo il caso del problema VertexCover su un grafo senza archi. In questo caso abbiamo osservato come la strategia standard ad ogni passo tolga un nodo per arrivare facilmente alla soluzione ottima. Ammettendo la possibilità di saltare a soluzioni peggiori, magari con alta probabilità, si ha che l'algoritmo oscilli avvicinandosi e allontandosi dalla soluzione ottima senza mai raggiungerla e con la concreta possibilità di non terminare mai.

Quindi se da un lato vogliamo una probabilità alta per poter sfuggire i punti di minimo locale, dall'altro vogliamo una probabilità bassa per evitare comportamenti poco efficienti.

Dunque, quale valore della temperatura dovremmo usare?

Il trucco della strategia di simulated annealing è quello di utilizzare un valore variabile per la temperatura, iniziando la ricerca con una temperatura alta e gradualmente riducendola a zero. Quindi inzialmente la ricerca procede in modo abbastanza libero, cioè permettendo ababstanza spesso spostamenti verso soluzioni peggiori. Poi gradualmente la temperatura viene diminuita e questo permette sempre meno di spostarsi verso soluzioni peggiori, fino ad arrivare a zero, per garantire la convergenza dell'algoritmo. La strategia (*annealing schedule*, che potremmo tradurre con *schema di raffreddamento*) con cui la temperatura viene abbassata è parte dell'algoritmo e può essere progettata in funzione dello specifico problema.

## 2.10 *Note bibliografiche*

Ulteriori approfondimenti sulla ricerca esaustiva intelligente possono essere trovati nel Capitolo 9 del libro DPV2008 [9]. Gli algoritmi RecursiveVC, GreedyIS e DynProWIS, sono presentati nel Capitolo 10 di KT2014 [20], dove è possibile trovare ulteriori approfondimenti. Per approfondimenti sugli algoritmi di ricerca locale si veda il Capitolo 4 di MP2017 [28], e il Capitolo 9 di DPV2008 [9] e il Capitolo 12 di KT2014 [20].

## 2.11 *Esercizi*

1. Nell'esercizio di programmazione per la scoperta dei fattori di un intero $n$ abbiamo usato l'accorgimento di provare a dividere $n$ per tutti gli interi fino a $n/2$ sfruttando il fatto che un intero più grande di $n/2$ non può dividere $n$. Analizza il vantaggio derivante da tale accorgimento. Di quanti bit bisogna "allungare" $n$ per perdere tale vantaggio?

2. Disegnare l'albero di backtracking per la formula boolena $\phi = (\neg a + b + \neg c + d) \cdot (\neg a + \neg b) \cdot (\neg a + b) \cdot (a + \neg c) \cdot (a + c)$.

3. Si consideri l'istanza del problema LatinSquare riportata in figura 2.22 (quindi $n = 2$). Si applichi la riduzione al problema del Sudoku mostrando la corrispondente istanza di tale problema che deriva dalla riduzione.

| | |
|---|---|
| | 2 |
| | |

Figura 2.22: Istanza di LatinSquare per l'esercizio 3.

4. Si fornisca una restrizione sull'input, coinvolgendo almeno $n$ elementi della scacchiera, che renda il problema LatinSquare risolvibile in tempo polinomiale. Si dia

un algoritmo efficiente per la restirizione individuata.

5. Si consideri il grafo riportato in figura 2.23 che rappresenta un'istanza del problema del commesso viaggiatore. Si applichi (disegnando l'albero) la tecnica di branch-and-bound per risolvere il problema.

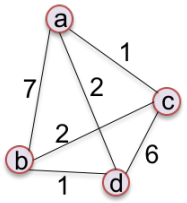



Figura 2.23: Istanza di TSP per l'esercizio 5.

6. Fornire un algoritmo di baktrack o di branch-and-bound per il problema IndependentSet: descrivere l'albero di branch-and-bound specificando in maniera dettagliata cosa rappresentano i nodi ed i cammini.

7. Consideriamo il problema 3Sat-3: Data una formula $\phi(x_1, x_2, \ldots, x_n) = C_1 \cdot \ldots \cdot C_m$ definita come la congiunzione di clausole in cui ogni clausola ha esattamente 3 letterali (com per 3Sat) e ognuna delle $n$ variabili appare, senza o con negazione, in esattamente 3 clausole. 3Sat-3 è come 3Sat con un vincolo aggiuntivo. Mentre una formula per 3Sat potrebbe non essere soddisfattibile, una formula per 3Sat-3 è sempre soddsifattibile; quindi il problema consiste nel trovare un assegnamento che la renda vera. Sfruttare i vincoli del problema per fornire un algoritmo polinomiale per 3Sat-3.

8. Consideriamo il seguente problema di schedulazione: ci sono due macchine identiche $M_1$ e $M_2$ che devono eseguire un insieme di $n$ lavori che richiedono tempi di esecuzione $t_1, \ldots, t_n$. I lavori devono essere partizionati in due sottoinsiemi $A$ e $B$, che verranno eseguiti dalle due macchine, in modo tale che il tempo di esecuzione più grande sia minimizzato; cioè detti $T_1 = \sum_{j \in A} t_j$ e $T_2 = \sum_{j \in B} t_j$ scegliere $A$ e $B$ in modo tale da minimizzare $\Delta = |T_1 - T_2|$. Questo problema di scheduling è $\mathcal{NP}$-completi. Valutiamo quindi un algoritmo di ricerca locale. Come relazione di vicinanza fra le soluzioni possiamo considerate la seguente: due soluzioni sono vicine se possiamo passare dall'una all'altra spostando un lavoro da una macchina all'altra. Una soluzione di partenza potrebbe essere quella che mette i primi $n/2$ lavori in $A$ ed i restanti in $B$.

   (a) Quanto è buona la soluzione che otterremo con questo algortimo di ricerca locale? Assumendo che non ci siano lavori che che dominano il tempo di esecuzione, cioè assumendo che $t_j \leq \frac{1}{2} \sum_{i=1}^{n} t_i$ per tutti i lavori $j = 1 = 2, \ldots, n$, provare che tutte le soluzioni ottime localmente si ha che $\frac{1}{2} T_1 \leq T_2 \leq 2T_1$, cioè che i tempi $T_1$ e $T_2$ sono abbastanza bilanciati.

   (b) Quante volte un lavoro può essere spostato da una macchina all'altra? Cioè l'algoritmo converge verso una soluzione? Per essere sicuri che ciò accada consideriamo la seguente variante: se ci sono più lavori che possono essere spostati, allora spostiamo sempre il lavoro con il tempo di esecuzione più grande. Provare che con questa variante ogni lavoro viene spostato al massimo una volta e quindi la sequenza di ricerca della soluzione non può contenere più di $n$ soluzioni.

   (c) Fornire un esempio in cui l'algoritmo trova un minimo locale che non è globale.

# 3
# *Algoritmi Approssimati*

Gli algoritmi di approssimazione sono algoritmi che garantiscono una soluzione sufficientemente vicina a quella ottimale. Sono utili per risolvere problemi non risolvibili con algoritmi deterministici o per migliorare l'efficienza della soluzione deterministica, il tutto a scapito della precisione della soluzione trovata. Ovviamente è richiesto che l'algoritmo di approssimazione funzioni in modo efficiente, cioè in tempo polinomiale. Quindi l'obiettivo di un algoritmo di approssimazione è quello di trovare una soluzione in tempo polinomiale offrendo una garanzia sulla lontananza massima dalla soluzione ottima.

In questo capitolo vedremo algoritmi di approssimazione per alcuni problemi difficili.

## *3.1 Approssimazione e efficienza*

Considereremo problemi di ottimizzazione: o ricerca di un massimo (profitto) o ricerca di un minimo (costo). Un algoritmo $A$ garantisce un'approssimazione $\rho$, $\rho \geq 1$, se il valore $V$ della soluzione prodotta da $A$ non si discosta dal valore $OPT$ della soluzione ottima, più di un fattore $\rho$, cioè se

$$V \leq \rho \cdot OPT \quad \text{per problemi di minimizzazione,}$$

oppure se

$$V \geq \frac{OPT}{\rho} \quad \text{per problemi di massimizzazione,}$$

come mostrato nella Figura 3.1

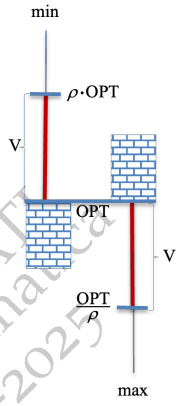


Figura 3.1: Approssimazione per problemi di ottimizzazione.

Quando questa condizione è garantita, l'algoritmo $A$ viene detto $\rho$-approssimato. Ad esempio un algoritmo 2-approssimato garantisce che la soluzione fornita non sia più grande del doppio della soluzione ottima, nel caso di problemi di minimizzazione, oppure più piccola della metà dell'ottimo nel caso di problemi di massimizzazione. Si noti come deve necessariamente essere $\rho \geq 1$. Gli algoritmi esatti, cioè non approssimati, e che quindi trovano la soluzione ottima, hanno un fattore di approssimazione pari a 1, sono cioè 1-approssimati. Quanto più grande è il fattore di approssimazione tanto peggiore potrà essere la soluzione fornita dall'algoritmo di approssimazione.

Si noti come abbiamo implicitamente assunto che $\rho$ sia una costante. In realtà il fattore di approssimazione può dipendere dalla taglia del problema o anche dal tempo

di esecuzione dell'algoritmo, come vedremo nel prosieguo del capitolo. Tuttavia, molti degli algoritmi di approssimazione che vedremo garantiscono una soluzione il cui valore si può allontanare dal valore ottimo per un fattore multiplicativo costante, e questo succede per la maggior parte dei casi.

Per il problema VertexCover vedremo algoritmi 2-approssimati. Si noti che questo significa che una soluzione può discostarsi da quella ottima fino al 100%. Ovviamente ridurre questo fattore è importante in pratica. Un algoritmo approssimato che garantisca una soluzione che si discosti molto poco da quella ottima è praticamente un algoritmo che fornisce quasi la soluzione ottima; ad esempio una soluzione che si discosta dello 0.5% da quella ottima potrebbe essere più che sufficiente[1]. In alcuni casi riusciamo a fornire degli algoritmi che possono produrre una soluzione approssimata buona quanto si vuole; ovviamente per fare ciò serve più tempo ma il tempo necessario, seppur maggiore, rimane polinomiale. Approfondiremo questo aspetto nella Sezione 3.5. Sfortunatamente questo non è sempre possibile e ci sono casi in cui gli algoritmi di approssimazione sono anche peggiori, non riuscendo a garantire un discostamento di un fattore costante e quindi la soluzione trovata peggiora al crescere della grandezza dell'input.

[1] Ovviamente la percentuale va valutata in relazione allo specifico problema.

Una motivazione può essere identificata nel seguente fatto: se $\mathcal{P} \neq \mathcal{NP}$, i problemi $\mathcal{NP}$-completi differiscono considerevolmente per quanto riguarda la loro approssimabilità. Ad esempio si può dimostrare che se $\mathcal{P} \neq \mathcal{NP}$, l'algoritmo approssimato che vedremo per il problema della selezione dei centri fornisce la migliore approssimazione possibile se ci vincoliamo ad usare algoritmi polinomiali. Per altri problemi, come ad esempio il problema del ricoprimento con vertici, l'algoritmo che vedremo è il migliore che si conosce ma dire se è il migliore possibile è un problema aperto. Non discuteremo di limiti dell'approssimabilità dei problemi: anche se per alcuni problemi (come quello della selezioni dei centri) dimostrare un limite all'approssimabilità non è molto difficile, spesso tali dimostrazioni sono estremamente tecniche.

## 3.2 VertexCover

Consideriamo il problema del ricoprimento con vertici. Dato un grafo $G = (V, E)$, un ricoprimento con vertici del grafo è un insieme $S \subseteq V$ tale che ogni arco in $E$ ha almeno uno dei due vertici in $S$.

Il problema di ottimizzazione che consideriamo consiste nel trovare un vertex cover che contenga il minimo numero di nodi possibile. Una variante più generale, WeightedVC, associa ad ogni nodo $i$ un peso $w_i \geq 0$. Il peso di un insieme di vertici $S$, è la somma dei pesi dei vertici appartenenti all'insieme, $w(S) = \sum_{i \in S} w_i$. Il problema WeightedVC è quello di trovare un ricoprimento il cui peso sia minimo.

È facile vedere che VertexCover $\leq_P$ WeightedVC: Basta usare $w_i = 1$, per tutti i nodi $i$, per fare in modo che il peso di un insieme di nodi sia esattamente la sua cardinalità. Quindi, dato che VertexCover è $\mathcal{NP}$-hard, anche WeightedVC è $\mathcal{NP}$-hard. In entrambi i casi possiamo dare un algorimo di approssimazione.

### 3.2.1 *Un algoritmo di approssimazione per* VertexCover

Per ottenere un algoritmo di approssimazione per VertexCover possiamo utilizzare un approccio semplice: selezionare due nodi qualsiasi uniti da un arco e inserirli nell'insieme ricoprente, cancellare tutti gli archi incidenti sui due nodi selezionati e ripetere il processo fino a che ci sono archi da coprire. Intuitivamente questo approccio funziona bene in quanto dato un arco $e = (u, v)$, un qualsiasi insieme ricoprente, quindi anche quello ottimale, dovrà includere almeno uno fra $u$ e $v$. Inserendoli entrambi saremo sicuri di avere questa proprietà e inoltre la nostra soluzione non potrà avere più del doppio del minimo numero di nodi necessari per un ricoprimento.

Lo pseudocodice è presentato nell'algoritmo ApproxVertexCover.

**Algorithm 9:** ApproxVertexCover($G = (V, E)$)

$S = \{\}$
*TempSet* $= E$
**while** *TempSet* $\neq \{\}$ **do**
  Sia $e = (u, v)$ un arco in *TempSet*
  $S = S \cup \{u, v\}$
  rimuovi da *TempSet* tutti gli archi incidenti su $u$ o su $v$
Restituisci $S$

Vediamo un esempio di esecuzione. Nella Figura 3.2 è schematizzata l'esecuzione dell'algoritmo su uno speicifico grafo, quello a sinistra, con 6 vertici e 8 archi. Nella

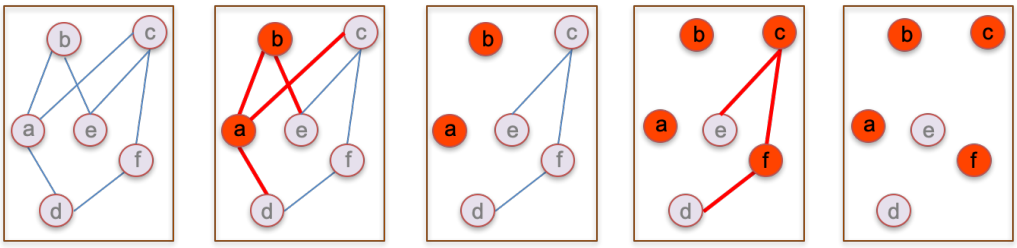



Figura 3.2: Esempio di esecuzione di ApproxVertexCover

prima iterazione l'algoritmo sceglie l'arco $(a, b)$. Quindi inserisce $a$ e $b$ in $S$ e cancella dal grafo tutti gli archi ad essi incidenti. Nella seconda iterazione l'algoritmo sceglie l'arco $(c, f)$. Quindi inserisce $c$ e $f$ in $S$ e cancella dal grafo tutti gli archi ad essi incidenti. A questo punto non ci sono più archi quindi l'algoritmo resituisce $S = \{a, b, c, f\}$.

Si noti che nella seconda iterazione l'algoritmo avrebbe potuto scegliere anche larco $(e, c)$ (o l'arco $(d, f)$, forzando così una terza iterazione che avrebbe portato ad inserire in $S$ tutti i nodi del grafo.

In questo particolare grafo la soluzione ottima è un ricoprimento di 3 nodi, $S^* = \{a, e, f\}$. Dunque in questo caso l'algoritmo approssimato ha fornito una soluzione relativamente vicina all'ottima, anche se avrebbe potuto fornirne una più lontana

che, sempre inq uesto specifico esempio, sarebbe stata anche la soluzione abnale che comprende tutti i nodi del grafo. Ma, al di là della bontà della soluzione, siamo sicuri che qualunque sia il grafo l'algoritmo effettivamente resituisce un ricoprimento? Sì, come provato nel seguente lemma.

**Lemma 3.2.1** *L'algoritmo* ApproxVertexCover *restituisce un ricoprimento.*

Dimostrazione. Sia $S$ l'insieme costruito dall'algoritmo. Il fatto che l'algoritmo restituisce $S$ significa anche che *TempSet* è (alla fine) vuoto. Supponiamo per assurdo che $S$ non sia un ricoprimento. Allora esiste un arco $e = (u, v)$ non ricoperto. Ovviamente tale arco è presente nell'insieme *TempSet* all'inizio dell'algoritmo e poichè né $u$ nè $v$ sono stati inseriti in $S$ esso non verrà mai cancellato da *TempSet*. Questo non è possibile perché abbiamo detto che (alla fine) *TempSet* è vuoto. □

L'algoritmo ApproxVertexCover opera in tempo polinomiale nel numero di nodi $n$ del grafo. Infatti il ciclo **while** può durare al massimo un numero di iterazioni pari al numero di archi visto che in ogni iterazione verrà cancellato dall'insieme *TempSet* come minimo l'arco $e$, se non anche altri archi. Il numero di archi è al massimo $n^2$. Il tempo necessario alle operazioni da svolgere in ogni singola iterazione dipende da come vengono rappresentati gli insiemi; senza preoccuparci di quale sia l'implementazione più efficiente possiamo sicuramente dire che è possibile implementarle in tempo $O(n^2)$ per l'insieme degli archi e $O(n)$ per quelli dei nodi. Quindi l'intero algoritmo, se l'implementazione è ragionevole, non può costare più di $O(n^4)$. Si noti che non ci stiamo preoccupando di quale sia la migliore implementazione, ma solo del fatto che il tempo necessario è polinomiale. Dal punto di vista pratico è ovviamente opportuno individuare la migliore implementazione possibile, cioè quella con tempo di esecuzione più basso. Per quanto detto poc'anzi si ha il seguente lemma.

**Lemma 3.2.2** *L'algoritmo* ApproxVertexCover *è efficiente.*

Abbiamo visto che l'algoritmo effettivamente restituisce un ricoprimento e che lo fa in tempo polinomiale. Ci rimane da provare una qualche garanzia sull'approssimazione della soluzione fornita da ApproxVertexCover.

L'intuizione, come abbiamo detto, è che non dovrebbe essere peggiore di 2 volte quella ottima. Infatti si ha il seguente risultato.

**Lemma 3.2.3** *L'algoritmo* ApproxVertexCover *è 2-approssimato.*

Dimostrazione. Consideriamo una qualsiasi istanza $I$ del problema e sia $S$ la soluzione fornita da ApproxVertexCover su input $I$ e indichiamo con $O$ la soluzione ottima per l'istanza $I$.

Durante la costruzione di $S$ l'algoritmo ApproxVertexCover ha selezionato un'insieme di archi, uno in ogni iterazione del ciclo **while**. Sia $E(S)$ l'insieme di tali archi.

Osserviamo che la soluzione $S$ contiene un numero di nodi pari esattamente a

$$|S| = 2|E(S)|,$$

in quanto per ogni arco selezionato vengono inseriti i due nodi che l'arco collega.

Inoltre, poichè ogni arco deve essere coperto da almeno un nodo in qualunque insieme ricoprente, quindi anche in $O$, e poichè gli archi inclusi in $E(S)$ non possono avere nodi in comune (in quanto ogni volta che un arco viene selezionato vengono cancellati tutti gli altri archi incidenti sui due nodi dell'arco), si ha che

$$|E(S)| \leq |O|,$$

cioè la soluzione ottima deve contenere almeno un nodo per ogni arco di $E(S)$.

Pertanto si ha che

$$|S| \leq 2|O|.$$

□

### 3.2.2 *Un algoritmo di approssimazione per* WeightedVC

Vediamo adesso la versione più generale in cui i vertici hanno dei pesi. Possiamo usare un approcio simile anche se ora dobbiamo tenere conto dei pesi. Pensiamo ai pesi come a dei costi: per coprire ogni arco dovremo *pagare* qualcosa, una frazione del costo totale. In questa prospettiva possiamo pensare ad ogni arco come ad un *agente* disponibile a pagare qualcosa per far coprire l'arco. L'algoritmo, oltre a trovare un ricoprimento, determinerà anche il "prezzo" che tale ricoprimento determina per ogni singolo arco. Diremo che i pagamenti $p_e$ per gli archi incidenti in un vertice $i$ sono *giusti* se la somma non supera il costo del nodo, cioè se $\sum_{e=(i,\cdot)} p_e \leq w_i$.

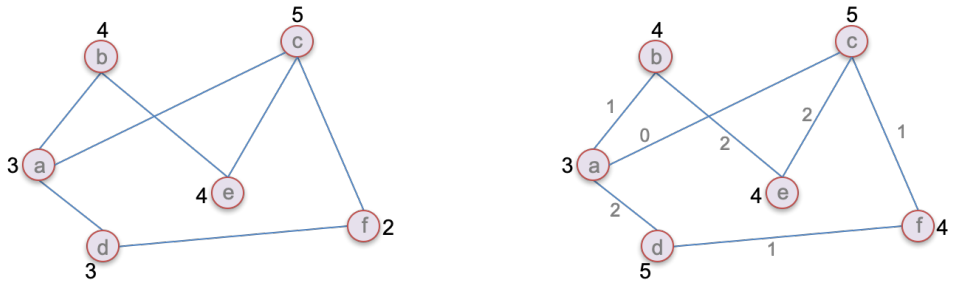



Figura 3.3: Grafo pesato e pagamenti giusti sugli archi

La Figura 3.3 mostra nella parte sinistra un grafo pesato e nella parte destra lo stesso grafo con dei pagamenti sugli archi. Ad esempio l'arco $(a, b)$ ha un pagamento $p_{(a,b)} = 1$ mentre l'arco $(b, e)$ ha un pagamento $p_{(b,e)} = 2$. Il pagamento relativo a un nodo è la somma dei pagamenti degli archi incidenti sul nodo. Ad esempio, per il nodo $a$ il pagamento totale è pari a 3, mentre per il nodo $e$ il pagamento totale è 4.

Si noti come il pagamento di un arco venga imputato ad entrambi i nodi incidenti. Quindi, ad esempio, il pagamento $p_{(a,b)} = 1$ contribuisce sia al pagamento totale relativo al nodo $a$ sia a quello relativo al nodo $b$.

I pagamenti mostrati nella figura sono giusti in quanto, per ogni singolo nodo, la somma dei pagamenti su tutti gli archi incidenti sul nodo stesso è minore del peso del nodo. Ad esempio per il nodo ha il pagamento totale è 3 ed è uguale al peso del nodo $a$. Per il nodo $f$ il pagamento totale è 2 ed è minore del peso (4) del nodo stesso.

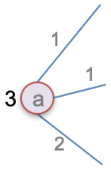



Figura 3.4: Pagamenti non giusti

La Figura 3.13 mostra un esempio di pagamenti non giusti: il pagamento totale per il nodo $a$ è 4 ed è maggiore del peso del nodo che è pari a 3.

Quando la somma dei pagamenti sugli archi incidenti è uguale al peso del nodo, $\sum_{e=(a,\cdot)} p_e \leq w_a$, diremo che il nodo è *saturo*. Ad esempio, il nodo $a$ è saturo mentre il nodo $b$ non lo è.

Vediamo l'algoritmo approssimato con il quale vogliamo sia trovare un ricoprimento, sia stabilire i pagamenti. L'algoritmo segue un approccio *greedy* rispetto al modo in cui sceglie i pagamenti, cercando cioè di pagare quanto meno possibile.

**Algorithm 10:** ApproxWeightedVC$(G, w)$

$p_e = 0$ per tutti gli archi $e \in E$
**while** $\exists e = (i, j) \in E$ *tale che né* $i$ *né* $j$ *sono saturi* **do**
    Sia $e$ un tale arco
    Incrementa $p_e$ senza violare la proprietà di pagamento giusto.
Sia $S$ l'insieme dei nodi saturi
Restituisci $S$

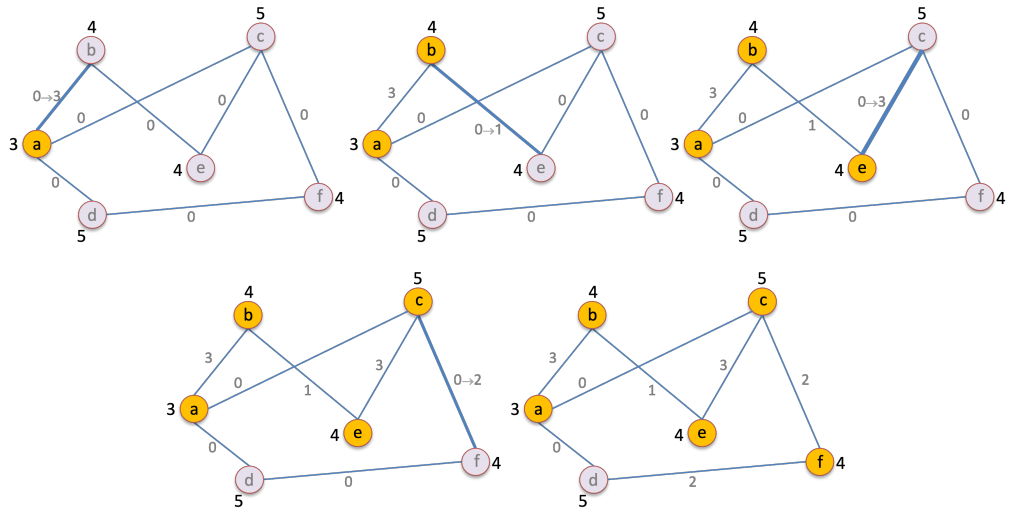


Figura 3.5: Esecuzione ApproxWeightedVC

Come esempio consideriamo l'esecuzione dell'algoritmo ApproxWeightedVC sul grafo mostrato nella Figura 3.5. I pagamenti iniziali sono 0 quindi nessun nodo è saturo. Supponiamo che l'algoritmo selezioni l'arco $(a, b)$. Possiamo incrementare il prezzo di tale arco fino a 3, in quanto a quel punto il nodo $a$ diventa saturo. A quel punto l'algoritmo potrebbe selezionare l'arco $(b, e)$ ed aumentare il suo prezzo fino a 1: poiché il peso di $b$ è 4 e su $b$ incidono sia l'arco $(a, b)$ che l'arco $(b, e)$, il pagamento su quest'ultimo può essere al massimo 1 per non violare la proprietà di pagamento giusto. A questo punto anche il nodo $b$ è saturo. Quindi nella successiva iterazione l'algoritmo potrebbe selezionare l'arco $(e, c)$ e portare il pagamento fino a 3 saturando

così il nodo $e$. Quindi si potrebbe incrementare il pagamento sull'arco $(c, f)$ fino a 2, saturando il nodo $c$ ed infine portare il pagamento sull'arco $(f, d)$ a 2 per saturare il nodo $f$. A questo punto non ci sono più archi che incidono su due nodi non saturi e quindi l'algoritmo termina restituendo l'insieme dei nodi saturi, cioè $\{a, b, c, e, f\}$.

**Lemma 3.2.4** *L'algoritmo* ApproxWeightedVC *resitutisce un ricoprimento.*

Dimostrazione. Sia $S$ l'insieme dei nodi saturi restituti dall'algoritmo. Sia $e = (u, v)$ un arco. Se $e$ non fosse coperto, significherebbe che nessuno dei due nodi è in $S$, cioè nessuno dei due nodi è saturo. Quindi si potrebbe aumentare il pagamento sull'arco $e$. Ciò non può essere perchè l'algoritmo si ferma solo quando non può più aumentare pagamenti sugli archi. Quindi almeno uno fra $u$ e $v$ deve esser eincluso in $e$. Questo è vero per tutti gli archi. Quindi $S$ è un ricoprimento. □

Rimane da provare il fattore di approssimazione. Diamo prima un risultato che non ha che fare direttamente con l'algoritmo ma deriva dalla defizioni di pagamento giusto.

**Lemma 3.2.5** *Per ogni ricoprimento $S^*$, e ogni pagamento giusto $p_e$, si ha che $\sum_{e \in E} p_e \leq w(S^*)$.*

Dimostrazione. Consideriamo un ricoprimento $S^*$. Dalla definizione di pagamenti giusti, si ha che $\sum_{e=(i,\cdot)} p_e \leq w_i$, per tutti i nodi $i \in S^*$. Sommando su tutti i nodi di $S^*$ otteniamo

$$\sum_{i \in S^*} \sum_{e=(i,\cdot)} p_e \leq \sum_{i \in S^*} w_i = w(S^*).$$

Osserviamo ora che, poichè $S^*$ è un ricoprimento, ogni arco $e \in E$ appare almeno in una sommatoria del termine a sinistra della precedente diseguaglianza. Potrebbe apparire più di una volta se il ricoprimento contiene entrambi i vertici dell'arco. Pertanto si ha che:

$$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e=(i,\cdot)} p_e,$$

dalla quale si conclude che

$$\sum_{e \in E} p_e \leq w(S^*).$$

□

Vediamo il precedente lemma considerando l'esempio della Figura 3.5. Si ha che $\sum_{e \in E} p_e = 3 + 1 + 3 + 2 + 2 = 11$. Se consideriamo il ricoprimento $S^* = \{a, b, c, e, f\}$ si ha che $\sum_{i \in S^*} \sum_{e=(i,\cdot)} p_e = (3 + 0 + 0) + (3 + 1) + (0 + 3 + 2) + (1 + 3) + (2 + 2) = 3 + 4 + 5 + 4 + 4 = 20 = w(S^*)$. In questo esempio il pagamento di ogni arco compare esattamente due volte e i nodi di $S^*$ sono tutti saturi, quindi otteniamo un'uguaglianza. Se consideraimo il ricoprimento $S^* = \{a, b, c, d\}$ si ha che $\sum_{i \in S^*} \sum_{e=(i,\cdot)} p_e = (3 + 0 + 0) + (3 + 1) + (0 + 3 + 2) + (0 + 2) = 3 + 4 + 5 + 2 = 14$ mentre $w(S^*) = 3 + 4 + 5 + 5 = 17$. In questo caso il pagamento sull'arco $(d, f)$ compare una sola volta in quanto il nodo $d \in S^*$ non è saturo.

Infine proviamo che ApproxWeightedVC è 2-approssimato.

**Lemma 3.2.6** *L'insieme $S$ ed i prezzi $p_e$ calcolati dall'algoritmo* ApproxWeightedVC *soddisfano $w(S) \leq 2\sum_{e \in E} p_e$.*

Dimostrazione. Tutti i nodi di $S$ sono saturi, quindi si ha che $\sum_{e=(i,\cdot)} p_e = w_i$ per tutti i nodi $i \in S$. Pertanto,

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,\cdot)} p_e.$$

Un arco $e = (i, j)$ può apparire al massimo due volte nelle sommatorie del termine a destra (questo succede quanto entrambi $i$ e $j$ fanno parte di $S$), pertanto si ha che

$$\sum_{i \in S} \sum_{e=(i,\cdot)} p_e \leq 2 \sum_{e \in E} p_e.$$

□

Possiamo esemplificare il precedente lemma considerando di nuovo il caso della Figura 3.5. Abbiamo che $S = \{a, b, c, e, f\}$. Quindi si ha che $\sum_{i \in S} \sum_{e=(i,\cdot)} p_e = (3 + 0 + 0) + (3 + 1) + (0 + 3 + 2) + (1 + 3) + (2 + 2) = 3 + 4 + 5 + 4 + 4 = 20$. Mentre $2\sum_{e \in E} p_e = 2(3 + 1 + 3 + 2 + 2) = 22$. La differenza di 2 è dovuta all'unico arco che nella prima sommatora non compare 2 volte, cioè l'arco $(d, f)$ che è l'unico a non essere coperto da 2 nodi.

**Lemma 3.2.7** *L'insieme $S$ calcolato dall'algoritmo* ApproxWeightedVC *è un ricoprimento con vertici ed il suo costo è al massimo 2 volte il costo di un ricoprimento ottimo.*

Dimostrazione. Dal Lemma 3.2.4 si ha che $S$ è un ricoprimento. Il limite sull'approssimazione ottenuta dall'algoritmo si deriva facilmente dai Lemmi 3.2.5 e 3.2.6. Sia $S^*$ un ricoprimento ottimale. Dal Lemma 3.2.6 si ha che $w(S) \leq 2\sum_{e \in E} p_e$ mentre dal Lemma 3.2.5 si ha che $\sum_{e \in E} p_e \leq w(S^*)$. Unendo le due diseguaglianze di ha

$$w(S) \leq 2 \sum_{e \in E} p_e \leq 2w(S^*).$$

□

### 3.2.3 *Programmazione lineare*

Un altro algoritmo approssimato per il problema WeightedVC può essere ottenuto sfruttando una tecnica molto potente usata in ricerca operativa: la *programmazione lineare*. Tale tecnica è l'oggetto di interi corsi per cui non avremo la pretesa di fornire nessuna descrizione approfondita, ma il nostro obiettivo è solo quello di introdurre le idee di base e conoscere quanto basta per applicare la programmazione lineare alla progettazione di algoritmi approssimati.

Nella prossima sezione, utilizzeremo la programmazione lineare per progettare un nuovo algoritmo di approssimazione per WeightedVC.

Per introdurre la programmazione lineare è utile richiamare alcune nozioni di algebra lineare per la risoluzione simultanea di un insieme di equazioni lineari. Usando la notazione matriciale, abbiamo un vettore $x$ di incognite, una matrice $A$ di coefficienti,

un vettore $b$ di termini noti. Vogliamo risolvere l'equazione $Ax \geq b$. Ad esempio, se $x = [x_1, x_2]$ è il vettore delle incognite, un sistema di equazioni $Ax \geq b$ è:

$$\begin{aligned} x_1 + 2x_2 &\geq 6 \\ 2x_1 + x_2 &\geq 6 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

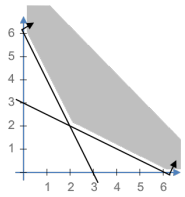


Figura 3.6: Regione ammissibile

dove le ultime 2 equazioni servono a limitare la ricerca a incognite non negative.

Tali equazioni determinano una regione di soluzioni. La Figura 3.6 riporta la regione delle soluzioni dell'esempio appena fatto. Il vettore $b = [6, 6]$ mentre la matrice $A$ è:

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}.$$

In un problema di programmazione lineare abbiamo una funzione obiettivo specificata da un vettore di coefficienti $c$, che determina la funzione rappresentando una combinazione lineare delle incognite: $c^T x$, dove $c^T$ è il vettore dei coeffienti trasposto, e $c^T x$ è il prodotto di due vettori, che quindi è la combinazione lineare delle incognite espressa dai coefficienti $c$. Ad esempio se $c = (3/2, 1)$ allora la funzione obiettivo è $3/2x_1 + x_2$.

Un problema di programmazione lineare in forma standard è il seguente.

*Data una matrice $A$ di $m \times m$, e dei vettori $b \in \mathbb{R}^m$, $c \in \mathbb{R}^m$, trovare un vettore $x \in \mathbb{R}^m$, per risolvere il seguente problema di ottimizzazione:*

$$\min c^T x \text{ vincolato a } Ax \geq b \text{ e } x \geq 0.$$

Per evitare problemi con la rappresentazione dei numeri reali, assumeremo che tutti i numeri coinvolti nella matrice $A$ e nei vettori $b$ e $c$ sono interi.

Nell'esempio fatto in precedenza la soluzione è $x_1 = 2, x_2 = 2$, che corrisponde al punto in cui si intersecano i due vincoli del problema (le due rette). Tale soluzione ci da un valore della funzione obiettivo pari a 5.

Un problema di programmazione lineare può essere anche formulato in versione decisionale nel seguente modo: Data una matrice $A$, ed i vettori $b$ e $c$ ed un valore $\gamma$, esiste un vettore $x$ tale che $x \geq 0$, $Ax \geq b$ e $c^T x \leq \gamma$? Tale problema appartiene a $\mathcal{NP}$. Intuitivamente ciò è credible: infatti dato un vettore $x$, è facile verificare che il vettore $x$ soddisfa tutti i vincoli imposti dal problema. Ma è veramente facile effettuare la verifica? Una difficoltà potrebbe nascondersi nei valori delle incognite $x_i$ che essendo dei numeri reali (possono esserlo anche se tutti i coefficienti sono interi) potrebbero richiedere moltissimi bit per la loro rappresentazione. Un numero irrazionale richiederebbe infiniti bit, non può nemmeno essere rappresentato. Tuttavia, si può dimostrare che se esiste una soluzione allora esiste una soluzione razionale che può essere specificata con un numero polinomiale di bit. Quindi di fatto il problema è in $\mathcal{NP}$ in quanto si può fornire tale soluzione come certificato. Si può dimostrare che oltre ad appartenere a $\mathcal{NP}$ il problema appartiene anche a Co-$\mathcal{NP}$. Per molto tempo, la programmazione lineare, è

stato il più famoso esempio di problema appartenente sia $\mathcal{NP}$ che a Co-$\mathcal{NP}$. Poi sono stati scoperti degli algoritmi che lo risolvono in tempo polinomiale.

In pratica però per la risoluzione di un problema di programmazione lineare si utilizza il cosiddetto *metodo del simplesso* che pur non essendo polinomiale funziona molto bene, e su istanze di input reali è più efficiente degli algoritmi polinomiali. Solo raramente il metodo del simplesso mostra il suo "carattere esponenziale". Questo fatto ha costituito anche motivo di riflessione sulla definizione di efficienza degli algoritmi: abbiamo definito efficiente un algoritmo polinomiale; ora stiamo presentando un problema per il quale conviene in pratica usare un algoritmo esponenziale.

Il lettore interessato potrà approfondire la conoscenza della programmazione lineare in altri corsi. Per i nostri obiettivi, ci poniamo la seguente domanda: come possiamo utilizzare la programmazione lineare per progettare algoritmi di approssimazione?

### 3.2.4 WeightedVC *con programmazione lineare*

Ritorniamo al problema della copertura con vertici. Abbiamo un grafo $G = (V, E)$, con dei pesi $w_i$ associati ai vertici $i \in S$ e vogliamo trovare un ricoprimento $S \subseteq V$ di peso totale minimo. Il ricoprimento $S$ deve coprire tutti gli archi, cioè per ogni arco $e = (i, j)$ almeno uno fra $i$ e $j$ deve appartenere a $S$.

Vogliamo sfruttare la programmazione lineare per risolvere il problema. Useremo una variabile $x_i$ per ogni nodo $i \in V$. Il valore di $x_i$ nella soluzione ci dirà se inserire o meno il nodo $i$ nel ricoprimento: se $x_i = 1$ allora $i$ fa parte del ricoprimento se $x_i = 0$ allora $i$ non fa parte del ricoprimento. Ovviamente dovremo fare in modo che $x_i$ non possa assumere altri valori.

Utilizzeremo le disuguaglianze per codificare il vincolo che i nodi selezionati debbano formare un ricoprimento mentre la funzione obiettivo potrà essere sfruttata per minimizzare il peso totale. Quindi per ogni arco $(i, j) \in E$, codificheremo il fatto che almeno uno fra $i$ e $j$ deve appartenere al ricoprimento con la disuguaglianza $x_i + x_j \geq 1$. La funzione obiettivo sarà semplicemente data dai pesi che fungeranno da coefficienti $c = [w_1, \ldots, w_n]$. Quindi il problema del ricoprimento con vertici può essere formulato con il seguente problema di programmazione lineare.

$$\min \sum_{i \in V} w_i x_i$$

vincolato da

$$x_i + x_j \geq 1 \text{ per ogni arco } (i, j) \in E$$

$$x_i \in \{0, 1\} \text{ per ogni } i \in V.$$

Purtroppo il vincolo $x_i \in \{0, 1\}$ rende il problema notevolmente più complicato. Infatti si tratta ora di risolvere un problema di programmazione lineare per il quale cerchiamo delle soluzioni intere. Chiameremo questo problema IntegerProgramming.

**Lemma 3.2.8** *Un vettore $x$ è una soluzione del problema* IntegerProgramming *se e solo se l'insieme di vertici corrispondenti a $x$ è un ricoprimento per $G$. Inoltre $w(S) = w^T x$.*

DIMOSTRAZIONE. Sia $S$ un ricoprimento e $x$ una soluzione del problema INTEGERPROGRAMMING. Per ogni arco $(i,j) \in E$ si ha che almeno uno fra $i$ e $j$ appartiente $S$, quindi almeno uno fra $x_i$ e $x_j$ vale 1 pertanto $x_i + x_j \geq 1$. Ovviamente tutti gli $x_i$ valgono o 0 o 1.

Viceversa sia $x$ un vettore che soddisfa i vincoli del problme INTEGERPROGRAMMING e sia $S = \{i | x_i = 1\}$. Poichè $x_i + x_j \geq 1$ per ogni arco $(i,j) \in E$, si ha che $S$ è un copertura di $G$.

L'uguagliza $w(S) = w^T x$ è immediata in quanto $w^T x = \sum_{i \in S} w_i$. □

Quello che abbiamo mostrato in realtà è che VERTEXCOVER$\leq_P$INTEGERPROGRAMMING. Il problema INTEGERPROGRAMMING è un problema $\mathcal{NP}$-completo. Per poter procedere allora rinunciamo al vincolo $x_i \in \{0,1\}$, ma richiediamo più semplicimente che $0 \leq x_i \leq 1$. Questo riporta il problema di programmazione lineare in una forma risolvibile efficientemente. Tuttavia ora non sappiamo più se la soluzione al problema LINEARPROGRAMMING fornisce una soluzione buona per il problema originario. Prima di tutto non abbiamo più una ovvia corrispondenza fra la soluzione di LINEARPROGRAMMING e quella di VERTEXCOVER.

Dobbiamo quindi trovare un modo per definire una copertura a partire da una soluzione del problema LINEARPROGRAMMING. Se $x_i = 1$ oppure $x_i = 0$ ci troviamo nella stessa situazione di INTEGERPROGRAMMING e possiamo dire che, rispettivamente, $i \in S$ o che $i \notin S$. Ma cosa facciamo se $0 < x_i < 1$? La scelta più ovvia è quella di arrotondare il valore, cioè scegliere il valore intero più vicino. Pertando definiamo la copertura ponendo $S = \{i | x_i \geq 0.5\}$. L'insieme così definito è effettivamente una copertura? E, se lo è, che possiamo dire sul suo costo? Iniziamo con l'osservare che è effettivamente è una copertura.

**Lemma 3.2.9** *L'insieme di vertici derivanti da una soluzione del problema* LINEARPROGRAMMING *è una copertura del grafo.*

DIMOSTRAZIONE. Consideriamo un arco $e = (i,j) \in E$. Sia $x$ una soluzione del problema LINEARPROGRAMMING. Dai vincoli si ha che $x_i + x_j \geq 1$. Quindi è necessario che o $x_i \geq 1/2$ oppure che $x_j \geq 1/2$. Quindi almeno uno dei quindi almeno uno dei due vertici $i$ e $j$ sarà incluso nell'insieme di vertici corrispondenti a $x$. □

Nel seguito utilizzeremo la seguente notazione. Indicheremo con $x^*_{IP}$ la soluzione ottima al problema INTEGERPROGRAMMING e con $S^*_{IP}$ la corrispondente copertura. Indicheremo con $x^*_{LP}$ la soluzione ottima al problema LINEARPROGRAMMING. Data la soluzione ottima $x^*_{LP}$ indicheremo con $S_{LP}$ la corrispondente copertura. Si noti che non usiamo l'asterisco in quanto non sappiamo se la copertura è ottimale. Infatti mentre per il problema INTEGERPROGRAMMING c'è una corrispondenza biunivoca fra soluzioni del programma lineare e ricoprimenti del grafo, per il problema LINEARPROGRAMMING la corrispondenza non è biunivoca. Ad esempio, se per un arco $e = (i,j)$ si avesse $x_i = 0.7$ e $x_j = 0.2$, tale soluzione non sarebbe ammissibile nel problema LINEARPROGRAMMING mentre l'arco $e$ sarebbe coperto. In altre parole è come se la ricerca della soluzione tramite LINEARPROGRAMMING escludesse alcune soluzioni del problema originale e questo potrebbe portarci a "perdere" la soluzione ottima. Non è un problema in quanto stiamo cercando un algoritmo di approssimazione.

Che possiamo dire del peso di $S_{LP}$?

**Lemma 3.2.10** $w(S_{LP}) \leq 2 \cdot w(S^*_{IP})$.

DIMOSTRAZIONE. Poichè $x^*_{IP}$ è una soluzione a un problema di programmazione lineare intera, si ha che le singole componenti sono 0 o oppure 1 e poichè $S^*_{IP}$ include tutte le componenti corrispondenti a 1 si ha che

$$w(S^*_{IP}) = \sum_{i \in S^*_{IP}} w_i = \sum_{i \in V} w_i x_i = w(x^*_{IP}).$$

D'altra parte poichè il problema LINEARPROGRAMMING ($0 \leq x_i \leq 1$) ha una regione ammissibile che include quella del problema INTEGERPROGRAMMING ($x_i \in \{0, 1\}$), si ha che

$$min_{x_{LP}} \sum_{i \in V} w_i x_i^{LP} \leq min_{x_{IP}} \sum_{i \in V} w_i x_i^{IP},$$

e quindi che

$$w(x^*_{LP}) \leq w(x^*_{IP}).$$

Pertanto

$$\begin{aligned} w(x^*_{IP}) &\geq w(x^*_{LP}) \\ &= \sum_{i \in V} w_i x_i^{LP*} \\ &\geq \sum_{i \in S_{LP}} w_i x_i^{LP*} \\ &\geq \frac{1}{2} \sum_{i \in S_{LP}} w_i \\ &\geq \frac{1}{2} w(S_{LP}). \end{aligned}$$

Quindi concludiamo che

$$w(S_{LP}) \leq 2w(x^*_{IP}) = 2w(S^*_{IP}).$$

□

### 3.2.5 *Migliorare il fattore di approssimazione*

Nelle precedenti sezioni abbiamo presentato vari approcci per algoritmi di approssimazione per il problema VERTEXCOVER e per la sua generalizzazione WEIGHTEDVC. Tutti gli algoritmi presentati hanno un fattore di approssimazione pari a 2. È un caso? È possibile migliorare tale fattore? È l'analisi fatta che può essere migliorata oppure con tali algoritmi non possiamo sperare di fare meglio? In quest'ultimo caso, esistono altri algoritmi con un fattore di approssimazione migliore?

Per stabilire se l'analisi di un algoritmo di approssimazione è la migliore possibile si possono cercare degli esempi di input per i quali l'algoritmo fornisce un'approssimazione che è proprio quella dell'analisi. Questo implicherebbe che l'analisi è stretta e che il fattore trovato è intrinseco nell'algoritmo. Consideriamo ad esempio la seguente tipologia

di grafo: grafi bipartiti completi $K_{n/2,n/2}$, con $n$ pari, in cui ognuno degli $n/2$ nodi a sinistra è collegato ad ognuno degli $n/2$ nodi a destra (vedi Figura 3.7).

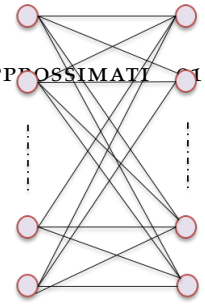


Figura 3.7: Grafo bipartito completo

Come si comporta ApproxVertexCover su questo tipo di grafi? Non è difficile vedere che selezionerà tutti i nodi, quindi la soluzione fornita è composta da $n$ nodi, mentre un ricoprimento ottimale può essere ottenuto selezionando o tutti gli $n/2$ nodi a sinistra o tutti gli $n/2$ nodi a destra. Quindi l'algoritmo, per questi grafi fornisce sempre una soluzione che è 2-approssimata. Dunque l'analisi fatta è stretta, cioè è la migliore in quanto ci sono casi in cui l'algoritmo fornisce soluzioni che hanno un fattore di approssimazione uguale a quello dell'analisi.

É l'algoritmo ApproxVertexCover a non essere sufficientemente buono oppure è il problema che non ammette algoritmi (efficienti) che garantiscano un'approssimazione migliore di 2? Questo è un problema aperto.

## 3.3 *Load Balancing*

Vediamo adesso un altro problema per il quale l'approccio greedy funziona: il problema di bilanciamento del carico (Load Balancing). Abbiamo $m$ macchine $M_1, \ldots, M_m$ e un insieme di $n$ compiti $C_1, \ldots, C_n$ da svolgere sulle macchine. Ognuno dei compiti necessita di un tempo di esecuzione pari a $t_j$. Vogliamo assegnare i compiti alle macchine (ogni compito viene assegnato ad una macchina, ogni macchina può svolgere più di un compito, uno dopo l'altro) in modo tale che il carico su ogni macchina sia quanto più bilanciato possibile; cioè la differenza fra il carico massimo e quello minimo deve essere quanto più piccola possibile. Ad esempio nella Figura 3.8 sono mostrati i carichi per 3 macchina; il carico massimo è di 10 unità di tempo e quello minimo di 7, quindi la differenza tra minimo e masismo in questo caso èdi 2 unità di tempo. L'obiettivo è minimizzare tale differenza.

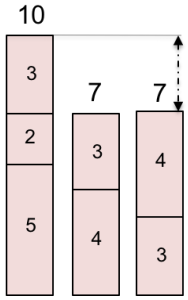



Figura 3.8: Carico per $5, 4, 3, 4, 3, 2, 3$

Più formalmente, dato un assegnamento dei compiti alle macchine, denotiamo con $A(i)$ l'insieme dei compiti assegnati alla macchina $i$. Poichè la macchina svolge i compiti ad essa assegnati uno dopo l'altro, il tempo di esecuzione necessario alla macchina $M_i$ per completare tutto il lavoro è dunque

$$T_i = \sum_{j \in A(i)} t_j.$$

Faremo riferimento a tale tempo di esecuzione come il "carico" della macchina $M_i$. Vogliamo minimizzare il carico massimo

$$T = \max_i T_i.$$

Non lo proviamo ma questo problema, apparentemente semplice, è $\mathcal{NP}$-hard. Pertanto presentiamo un algoritmo di approssimazione che si basa sulla tecnica greedy. La scelta greedy che operiamo è la seguente: nell'assegnare un nuovo compito alle macchine scegliamo quella che al momento ha il carico più piccolo.

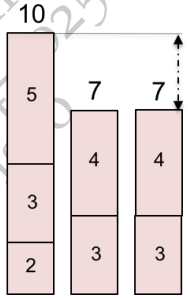



Figura 3.9: Carico per $2, 3, 3, 3, 4, 4, 5$

**Algorithm 11:** ApproxLoadInOrder

$T_i = 0$ e $A(i) = \{\}$ per tutte le macchine $M_i$
**for** $j \leftarrow 1$ **to** $n$ **do**
  Sia $M_i$ una macchina tale che $T_i = \min_k T_k$
  Assegna il compito $C_j$ alla macchina $M_i$
  $A(i) = A(i) \cup \{j\}$
  $T_i = T_i + t_j$

La Figura 3.8 mostra il risultato dell'esecuzione di questo algoritmo sull'istanza $C = \{5, 4, 3, 4, 3, 2, 3\}$ e $M = \{1, 2, 3\}$. I carichi sono stati processati nell'ordine in cui sono scritti nell'insieme. L'ordine con cui si processano i carichi è importante in quanto la soluzione fornita dall'algoritmo può cambiare se tale ordine cambia.

Processando i compiti in ordine crescente $2, 3, 3, 3, 4, 4, 5$ si ottiene il carico della Figura 3.9, mentre processando compiti in ordine decrescente $5, 4, 4, 3, 3, 3, 2$ si ottiene il carico della Figura 3.10. Tuttavia, nessuna di queste soluzioni è quella ottimale, che è mostrata nella Figura 3.11. Notiamo che le soluzioni non ottimali non si discostano in maniera significativa dall'ottimo. Nel seguito dimostreremo che ciò non è un caso.

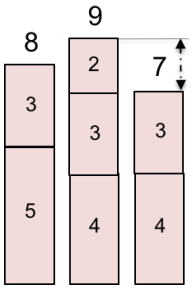


Figura 3.10: Carico per $5, 4, 4, 3, 3, 3, 2$

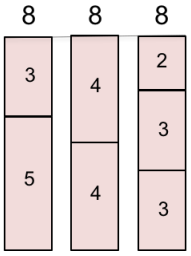


Figura 3.11: Carico ottimale

*Analisi.* Denotiamo con $T$ il carico massimo che risulta dall'algoritmo. Denotiamo, anche se non lo conosciamo, con $T^*$ il carico massimo ottimale. Il fatto che non conosciamo $T^*$ è ovviamente un problema in quanto per fornire delle garanzie sulla relazione fra $T$ e $T^*$ dovremmo conoscere $T^*$. Possiamo però individuare un limite inferiore per $T^*$ confrontare tale limite con $T$: otterremo un'analisi meno precisa nel senso che sarà pessimistica e quindi la garanzia sulla relazione fra $T$ e $T^*$ continuerà a valere.

Chiaramente l'analisi sarà tanto più precisa quanto migliore è il limite che troviamo per $T^*$. Un limite su $T^*$ è, ad esempio, $T^* \geq t_j$, per un indice $j$ qualsiasi: l'ottimo non può essere inferiore al tempo necessario per eseguire un compito qualsiasi! Tuttavia un tale limite non è molto buono in quanto potrebbe essere estremamente lontano dall'ottimo. Una tale situazione si ha, ad esempio, quando $t_j$ è estremamente piccolo in confronto ai tempi di esecuzione degli altri lavori. In ogni caso tale limitazione ci sarà utile e la possimao scrivere come

$$T^* \geq \max_{1 \leq j \leq n} t_j. \tag{3.1}$$

Un altro limite è

$$T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j \tag{3.2}$$

perchè se tutte le macchine avessero come carico qualcosa meno di una frazione $\frac{1}{m}$ del totale non si riuscerebbe a svolgere tutti i compiti in quanto ci sono solo $m$ macchine. Anche in questo caso però il limite potrebbe essere molto lontano dall'ottimo se c'è un solo compito il cui tempo di esecuzione è estremamente più grande di tutti gli altri. Anche questo limite ci sarà comunque utile.

Possiamo ora provare il seguente lemma.

**Lemma 3.3.1** *L'algoritmo* ApproxLoadInOrder *produce un carico massimo di* $T \leq 2T^*$.

Dimostrazione. Consideriamo una macchina $M_i$ il cui carico è proprio il carico massimo $T$ prodotto dall'algoritmo. Guardiamo l'ultimo compito $C_j$ assegnato a $M_i$. Quando tale compito è stato assegnato ad $M_i$, $M_i$ aveva il carico più piccolo. Poichè $C_j$ è l'ultimo compito assegnato a $M_i$, il carico di $M_i$ prima dell'assegnamento è di $T_i - t_j$. Quindi nel momento in cui $C_j$ è stato assegnato ad $M_i$, tutte le macchine avevano un carico pari almeno a $T_i - t_i$. Pertanto si ha che

$$\sum_{k=1}^{m} T_k \geq m(T_i - t_j),$$

o, equivalentemente, che

$$T_i - t_j \leq \frac{1}{m} \sum_{k=1}^{m} T_k.$$

D'altra parte $\sum_{k=1}^{m} T_k = \sum_{j=1}^{n} t_j$, quindi si ha che

$$T_i - t_j \leq \frac{1}{m} \sum_{j=1}^{n} t_j$$

e, usando l'equazione (3.2), si ha che

$$T_i - t_j \leq T^*.$$

Quindi abbiamo che

$$T_i \leq T^* + t_j$$

e sfruttando l'equazione (3.1), si ha che

$$T_i \leq T^* + T^* = 2T^*.$$

D'altra parte $T_i = T$, quindi $T \leq 2T^*$. □

L'analisi dell'approssimazione fornita dall'algortimo, cioè la prova che la soluzione fornita è al massimo due volte il valore ottimale, è molto buona, nel senso che esistono casi in cui effettivamente l'algoritmo si comporta fornendo delle soluzioni il cui carico massimo è quasi il doppio di quello ottimale; quindi l'analisi fornisce una buona valutazione del reale comportamento dell'algoritmo.

Consideriamo il caso in cui $n = m(m-1)+1$. I primi $m(m-1)$ compiti richiedono un tempo di esecuzione $t_j = 1$. L'ultimo compito invece ha un tempo di esecuzione molto più grande, $t_n = m$. L'algoritmo ApproxLoadInOrder distribuisce i primi $m(m-1)$ compiti equamente sulle macchine, in modo tale che ogni macchina riceva $m-1$ compiti, e l'ultimo compito su una qualsiasi delle macchine, generando un carico massimo di $2m-1$. La soluzione ottimale in questo caso sarebbe stata quella di mettere il compito grande da solo su una macchina e distribuire equamente gli altri $m(m-1)$ compiti, $m$ su ognuna delle $m-1$ macchine, per un carico massimo di $m$. Pertanto in questo paricolare esempio, l'approssimazione è di $\frac{2m-1}{m} = 2 - 1/m$, che al crescere di $m$ tende a 2.

Questo caso è veramente il peggiore possibile, nel senso, che con un po' di sforzo si può migliorare la prova del lemma precedente per mostrare che effetivamente il fattore di approssimazione non è 2, ma proprio $2 - 1/m$.

*Miglioramento.* Il caso peggiore mostrato nell'esempio è dovuto al fatto che il compito più grande è stato processato per ultimo quando ormai gli altri compiti erano già stati assegnati alle macchine. In altre parole i compiti grandi sono "scomodi" da piazzare quando ormai gli altri compiti sono stati già piazzati. Questa osservazione suggerisce che forse conviene assegnare i compiti in ordine di grandezza decrescente.

**Algorithm 12:** ApproxLoadDec

$T_i = 0$ e $A(i) = \{\}$ per tutte le macchine $M_i$
Ordina i compiti per $t_j$ decrescente
`/* Quindi ora abbiamo` $t_1 \geq t_2 \geq \ldots \geq t_n$ `*/`
**for** $j \leftarrow 1$ **to** $n$ **do**
  Sia $M_i$ una macchina tale che $T_i = \min_k T_k$
  Assegna il compito $C_j$ alla macchina $M_i$
  $A(i) = A(i) \cup \{j\}$
  $T_i = T_i + t_j$

Questa variante garantisce una soluzione che è al massimo 1.5 volte l'ottimo. Per provarlo, iniziamo con l'osservare che

$$\text{Se } n > m \text{ allora } T^* \geq 2t_{m+1}. \tag{3.3}$$

Infatti, consideriamo gli $m+1$ compiti con carico più grande $t_1 \geq t_2 \geq \ldots \geq t_m \geq t_{m+1}$ e vediamo come possono essere assegnati. Poichè ci sono $m$ macchine almeno due di questi compiti devono essere assegnati alla stessa macchina $M_j$. Poichè entrambi i compiti assegnati a $M_j$ hanno durata $\geq t_{m+1}$, si ha che $T_j \geq 2t_{m+1}$. Poichè il carico massimo è per definizione il massimo fra i carichi delle macchine, si ha l'equazione (3.3).

**Lemma 3.3.2** *L'algoritmo* ApproxLoadDec *garantisce che* $T \leq \frac{3}{2}T^*$.

Dimostrazione. Distinguiamo due casi: $n \leq m$ e $n \geq m+1$. Nel primo caso abbiamo un numero di macchine maggiore o uguale al numero di compiti, quindi l'algoritmo assegna al massimo un solo compito ad ogni macchina. Pertanto in questo caso abbiamo $T = T^*$.

Dobbiamo perciò preoccuparci dell'altro caso, in cui $n \geq m+1$. Consideriamo una macchina $M_i$ che ha carico massimo $T_i = T$. Se ad $M_i$ è stato assegnato un solo lavoro, allora si ha $T_i = T^*$, in quanto l'ottimo non può essere più piccolo del tempo di esecuzione di un qualsiasi lavoro. Qundi possiamo limitarci a considerare il caso in cui $M_i$ riceve almeno 2 compiti. Sia $C_j$ l'ultimo compito assegnato a $M_i$. Poichè $M_i$ riceve almeno 2 compiti, deve essere $j \geq m+1$ in quanto l'algoritmo assegnerà i primi $m$ compiti a $m$ macchine diverse. Poichè i compiti vengono assegnati in ordine di durata decrescente, si ha che $t_j \leq t_{m+1}$, e, visto che $n \geq m+1$, dall'equazione (3.3), si ha che $t_{m+1} \leq \frac{1}{2}T^*$, e pertanto otteniamo $t_j \leq \frac{1}{2}T^*$.

A questo punto possiamo procedere esattamente come nella prova del lemma precedente per arrivare a dire che

$$T_i - t_j \leq T^*.$$

Nella prova precedente da questo punto avevamo raggiunto la conclusione sfruttando 3.1. Qui possiamo concludere sfruttando $t_j \leq \frac{1}{2}T^*$ ottenendo

$$T_i \leq T^* + t_j \leq T^* + \frac{1}{2}T^* = \frac{3}{2}T^*.$$

□

Concludiamo con una nota sulla bontà di questa approssimazione. L'analisi che porta al fattore di approssimazione 3/2 non è stretta. Si può infatti dimostrare con un analisi più complessa che l'algoritmo ApproxLoadDec garantisce un fattore di approssimazione pari a 4/3 e tale bound è stretto [15].

## 3.4 *Problema della selezione del centro*

Consideriamo il seguente problema: abbiamo $n$ punti e vogliamo selezionare $k$ "centri" per questi $n$ punti. Per "centri" qui intendiamo il fatto che vogliamo che i $k$ punti scelti siano quanto più vicino possibile agli $n$ punti iniziali. Per definire più precisamente il problema, immaginiamo che gli $n$ punti iniziali siano delle città e che i $k$ punti da selezionare siano dei luoghi dove costruire dei centri commerciali. Vogliamo fare in modo che la distanza da ogni città a un centro commerciale sia la più piccola possibile.

Indicheremo con $S$ l'insieme delle città e con $C$ l'insieme dei centri. Formalmente avremo una funzione distanza che misura la distanza fra i vari luoghi (città e centri commerciali). Ovviamente la distanza rappresenta un costo e come in altri problemi potrebbe non essere la distanza fisica ma misurare altro. Assumeremo comunque che la metrica utilizzata soddisfi le proprietà di una distanza, cioè

- $dist(a, a) = 0$ per ogni $a \in S$.
- simmetria: $dist(a, b) = dist(b, a)$ per tutti $a, b \in S$.
- disuguaglianza triangolare: $dist(a, b) + dist(b, c) \geq dist(a, c)$.

Il nostro obiettivo è quello di far percorrere meno strada possibile per raggiungere i centri commerciali. Più precisamente, il centro commerciale più vicino per una città $a$ è a distanza $dist(a, C) = \min_{c \in C} dist(a, c)$. Quindi il problema è quello di scegliere i centri $C$ in modo tale da minimizzare il massimo di tali distanze fra tutte le città. Più formalmente diremo che un insieme di centri $C$ forma una $r$-copertura se ogni città si trova ad una distanza di al massimo $r$ da almeno uno dei centri commerciali, cioè se $dist(a, C) \leq r$ per ogni città $s \in S$. Chiameremo *raggio di copertura di* $C$, indicato con $r(C)$, il più piccolo valore di $r$ per il quale $C$ è una $r$-copertura. Quindi $r(C)$ è la massima distanza da una qualsiasi città al centro commerciale più vicino: assumendo che tutti si rechino presso il centro commerciale più vicino, nessuno dovrà percorrere una distanza maggiore di $r(C)$. L'obiettivo del problema è quello di selezionare $C$ in modo tale da minimizzare $r(C)$.

### 3.4.1 *Algoritmo* ApproxCentriConR

Una prima idea: selezioniamo il primo centro in modo tale che sia la posizione migliore se ci fosse solo questo centro. Per i successivi li posizioniamo ognuno in modo tale da diminuire ogni volta quanto più possibile il raggio di copertura.

Questo approccio è troppo semplice: è facile trovare dei casi in cui trova delle soluzioni molto inefficienti. Ad esempio, consideriamo il caso in cui ci sono solo due città e dobbiamo costruire due centri commerciali. Ovviamente la soluzione migliore è costruire un centro commerciale in ognuna della due città in modo tale da avere $r(C) = 0$. Tuttavia, la scelta greedy proposta seleziona il punto equidistante fra le due città $a$ e $b$ come luogo per il primo centro e dovunque si posizioni il secondo non c'è modo di ridurre il risultante raggio di copertura $r(C) = dist(a, b)/2$.

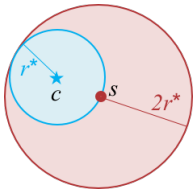



Figura 3.12: Scegliere una città per coprire tutto ciò che è coperto da un centro di una soluzione ottima significa usare un raggio di $2r^*$

Supponiamo adesso di conoscere a priori il valore $r^*$ del raggio di copertura ottimo (nell'esempio precedente tale valore era 0). Indicheremo con $C^*$ un insieme per il quale $r(C^*) = r^*$. Possiamo sfruttare il fatto di sapere che una tale soluzione esiste anche se non la conosciamo. Consideriamo una qualsiasi città $s \in S$. Deve esistere un centro $c \in C^*$ che copre $s$, cioè tale che $dist(s, c) \leq r^*$. L'idea è che potremmo usare la città $s$ come centro al posto di $c$, visto che non sappiamo dove $c$ sia, ma sappiamo che dista al massimo $r^*$ da $c$; quindi scegliendo $s$ possiamo "sbagliare" di al massimo $r^*$. Per essere sicuri che $s$ copra le stesse città coperte da $c$ dobbiamo quindi accettare che il raggio di copertura possa aumentare da $r^*$ a $2r^*$, come mostrato nella Figura 3.12.

**Algorithm 13:** ApproxCentriConR

```
S' = S                                          /* S' città da coprire */
C = {}
while S' ≠ {} do
    Seleziona un qualsiasi s ∈ S'
    C = C ∪ {s}
    Cancella tutte le città a distanza minore o uguale a 2r* da s
if |C| ≤ k then
    Restituisci C
else
    Dichiara che non esiste un insieme di k centri con raggio di copertura ≤ r*.
```

**Lemma 3.4.1** *Se l'algoritmo* ApproxCentriConR *restituisce un insieme $C$, tale insieme ha raggio di copertura $r(C) \leq 2r^*$.*

Dimostrazione. Una città viene cancellata dall'insieme $S'$ solo quando viene coperta da un centro che dista al massimo $2r^*$. Pertanto per ogni città $s$ si ha che $dist(s, C) \leq 2r^*$, e quindi $r(C) \leq 2r^*$. □

Rimane da provare che se l'insieme $C$ costruito dall'algoritmo ha più di $k$ centri, e quindi l'algoritmo non restituisce $C$ ma dichiara che non esiste un insieme di al più $k$ centri con raggio di copertura $r^*$, allora effettivamente non può esistere un insieme di centri con raggio di copertura $r^*$. In realtà, questa situazione non si può verificare, in

quanto sarebbe un assurdo: abbiamo supposto che $r^*$ è il raggio di copertura ottimale per gli insiemi di $k$ centri, quindi deve esistere un insieme con tale raggio di copertura. Proviamolo formalmente.

**Lemma 3.4.2** *Supponiamo che l'algoritmo* ApproxCentriConR *selezioni più di k centri. Allora, per ogni insieme $C'$ contenente al massimo k centri, si ha che $r(C') > r^*$.*

Dimostrazione. Sia $C$ l'insieme di centri selezionati dall'algoritmo con $|C| > k$. Per contraddizione, assumiamo che esista un insieme $C'$ con raggio di copertura $r(C') \leq r^*$ e contenente al massimo $k$ centri. Ognuno dei centri $c \in C$ è una delle città di $S$; per questo motivo chiameremo città-centri i centri di $C$. Per definizione esiste un centro $c' \in C'$ a distanza al massimo $r^*$ da una qualsiasi città-centro $c \in C$; cioè per ogni $c \in C$, esiste un $c' \in C'$ tale che $dist(c, c') \leq r^*$. Diremo che $c'$ è *vicino* a $c$.

Osserviamo che quando l'algoritmo ApproxCentriConR seleziona una città-centro cancella dalla lista tutte le città a distanza minore o uguale a $2r^*$. Quindi due qualsiasi città-centri $c, d$ di $C$ sono a distanza più grande di $2r^*$. Pertano se $c'$ è vicino ad uno dei due non può essere vicino all'altro.

Poichè $C'$ ha un raggio di copertura $r(C') \leq r^*$, si ha che ogni città-centro $c \in C$ è vicino ad un centro $c' \in C'$ e, per quanto detto poc'anzi, tutti questi centri di $C'$ sono diversi. Questo significa che $|C'| \geq |C|$ e poichè $C$ contiene più di $k$ centri, si ha che $|C'| > k$. Questa è una contraddizione, quindi l'assunzione che $C'$ sia un insieme di copertura con al massimo $k$ centri e raggio di copertura $r(C') \leq r^*$ è assurda. Tale insieme non può esistere. □

### 3.4.2 *Algoritmo* ApproxCentriSenzaR

L'algoritmo ApproxCentriConR si basa sulla conoscenza del raggio di copertura ottimo $r^*$. La conoscenza di tale valore è indispensabile per implementare l'algoritmo. Come facciamo se non lo conosciamo?

L'approccio che abbiamo descritto è utile anche se non conosciamo $r^*$. Infatti possiamo "indovinarlo" facendo una serie di tentativi. Osserviamo che $r^*$ è maggiore di 0 e minore della distanza massima fra due città $r_{max}$. Questo ci permette di fare una ricerca binaria, iniziando con il valore $r^* = r_{max}/2$. L'algoritmo può fornire due risposte: un insieme $C$ oppure l'affermazione che non esiste una soluzione con raggio di copertura $r^* = r_{max}/2$. Nel primo caso ripetiamo la ricerca nell'intervallo sinistro, mentre nel secondo caso la ripetiamo nell'intervallo destro. Pertanto, potremmo iterativamente mantenere due valori $r_0 < r_1$ in modo tale che sappiamo sempre che il valore di $r^*$ è maggiore di $r_0$ ma anche che abbiamo una soluzione di raggio al massimo $2r_1$. Ad ogni iterazione proviamo con $r^* = (r_0 + r_1)/2$. Dall'output dell'algoritmo capiremo o che una soluzione ottima con tale raggio non esiste e che quindi potremo aggiornare $r_0 = (r_0 + r_1)/2$ oppure che esiste una soluzione con raggio di copertura $2r^* = (r_0 + r_1) < 2r_1$, e quindi potremo aggiornare $r_1 = r^*$. In entrambi i casi abbiamo ristretto il nostro intervallo di ricerca. Potremo fermarci quando i valori di $r_0$ e $r_1$ sono sufficientemente vicini, nel qual caso la soluzione di raggio $2r_1$ è abbastanza vicina ad una soluzione 2-approssimata.

La tecnica che abbiamo appena descritto può essere applicata in generale. Per il caso specifico del problema della selezione dei centri, tuttavia, esiste una soluzione più elegante. Di fatto possiamo usare lo stesso approccio dell'algoritmo precedente anche senza conoscere il raggio di copertura ottimale.

L'algorimo precedente, grazie alla conoscenza di $r^*$, seleziona ripetutamente una delle città come prossimo centro, assicurandosi che sia almeno a distanza $2r^*$ da tutti gli altri centri selezionati. Questo lo fa eliminando dall'insieme $S'$ tutte le città a distanza al massimo $2r^*$ dalla città-centro selezionata. Possiamo ottenere lo stesso effetto anche senza la conoscenza di $r^*$, semplicemente selezionando la città $s$ che è la più distante da tutte le città-centro già selezionate. Se esistono delle città che distano almeno $2r^*$ da tutte le città-centro già selezionate la città $s$ deve essere una di queste (visto che è quella più distante!). Quindi l'algoritmo diventa il seguente.

**Algorithm 14:** ApproxCentriSenzaR: Selezione dei centri senza la conoscenza di $r^*$

```
if k ≥ |S| then
    Restitusci C = S
Seleziona una qualsiasi città b_1 e poni C = {b_1}
while |C| < k do
    Seleziona la città b che massimizza dist(b, C)
    C = C ∪ {b}
Restituisci C
```

**Lemma 3.4.3** *L'algoritmo* ApproxCentriSenzaR *restituisce un insieme $C$ di $k$ centri con $r(C) \leq 2r^*$, dove $r^*$ è il raggio di copertura ottimale.*

Dimostrazione. Sia $r^*$ il raggio di copertura ottimale per un insieme di $k$ centri. Sia $C$ l'insieme di $k$ centri restituito dall'algoritmo ApproxCentriSenzaR.

Se $k \geq |S|$ allora $r(C) = 0$ e quindi il lemma è vero. Pertanto consideriamo solo il caso $k < |S|$. Per assurdo, assumiamo che $r(C) > 2r^*$. Questo significa che esiste una città $s$ che dista più di $2r^*$ dal centro ad essa più vicino, cioè tale che $dist(s, C) > 2r^*$.

Sia $B$ il valore di $C$ durante l'esecuzione dell'algoritmo, cioè inizialmente $B = \{b_1\}$, poi alla prima iterazione diventa $B = \{b_1, b_2\}$, alla seconda $B = \{b_1, b_2, b_3\}$, e così via, e sia $b = b_i$ la città selezionata in una generica iterazione dell'algoritmo. Poichè $b$ è selezionata con il criterio di massimizzare la distanza verso tutte le città già presenti in $B$, si ha che quando $b$ viene selezionata deve necessariamente essere

$$dist(b, B) \geq dist(s, B).$$

Questo perché $b$ è la città più distante da tutte le città di $B$ e come caso limite si ha $b = s$ e quindi la disugaglianza è ovviamente valida.

Inoltre aggiungendo nuove città a $B$ la distanza $dist(s, B)$ non può che diminuire, quindi

$$dist(s, B) \geq dist(s, C).$$

Ricordando che $dist(s, C) > 2r^*$, si ha

$$dist(b, B) \geq dist(s, B) \geq dist(s, C) > 2r^*,$$

e quindi che

$$dist(b, B) > 2r^*.$$

Ciò è vero per ognuna della $k$ iterazioni dell'algoritmo ApproxCentriSenzaR. Questo significa che ApproxCentriSenzaR è una corretta "implementazione" delle prime $k$ iterazioni dell'algoritmo ApproxCentriConR, in quanto in ogni iterazione seleziona una città che dista più di $2r^*$ da tutte le cità già selezionate.

Poichè $dist(s, C) > 2r^*$, si ha che ApproxCentriConR non terminerà l'esecuzione dopo le prime $k$ iterazioni in quanto $s$ non verrà cancellata da $S$. Questo significa che ApproxCentriConR concluderà l'esecuzione non fornendo nessun insieme ma dichiarando che tutti gli insiemi di al più $k$ centri hanno un raggio di copertura $> r^*$. Abbiamo già provato che l'algoritmo ApproxCentriConR fornisce correttamente questa risposta, quindi possiamo conlcudere che il raggio di copertura ottimale deve essere $> r^*$. Questa è una contraddizione in quanto abbiamo supposto che $r^*$ è il raggio di copertura ottimale per insiemi con al più $k$ centri, quindi deve esistere un insieme con raggio di copertura $r^*$.

La contraddizione prova che l'assunzione $r(C) > 2r^*$ è assurda. Dunque $r(C) \leq 2r^*$ □

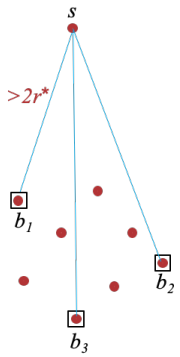



Figura 3.13: L'algoritmo non si ferma perché $s$ non è stata cancellata

## 3.5 *FPTAS: Il problema dello zaino*

Gli algoritmi approssimati che abbiamo visto hanno tutti un fattore di approssimazione costante. Quindi la sua "distanza" dall'ottimo dipende da questo fattore costante. Vedremo adesso un problema per il quale riusciamo a dare un algoritmo approssimato per il quale il fattore di approssimazione è variabile e la soluzione può essere buona quanto si vuole, cioè la si può far avvicinare a piacere all'ottimo. Ovviamente per calcolare una soluzione migliore l'algoritmo necessiterà di più tempo, ma il tempo necessario, sebbene maggiore, rimane, una volta fissato il valore del fattore di approssimazione, polinomiale.

Dunque un obiettivo significativo è quello di fornire algoritmi approssimati capaci di trovare soluzioni vicine a piacere all'ottimo. Per formalizzare questo aspetto utilizzeremo un parametro $\epsilon$, un numero reale che può essere piccolo a piacere, e vorremo un algoritmo che garantisca una soluzione che non si discosti da quella ottima più di fattore $(1 + \epsilon)$. Si noti come questo leghi $\epsilon$ a $\rho$ con la relazione $\epsilon = 1 - \rho$. Detto $V$ il valore della soluzione fornita da un algoritmo di approssimazione, richiederemo che

$$V \leq (1 + \epsilon)OPT \quad \text{per problemi di minimizzazione, e}$$

$$V \geq \frac{OPT}{1 + \epsilon} \quad \text{per problemi di massimizzazione.}$$

Chiaramente quanto più piccolo è $\epsilon$ (che equivale a quanto più $\rho$ si avvicina a 1) tanto migliore sarà l'approssimazione. Se ad esempio $\epsilon = 0.005$, l'algoritmo fornisce una soluzione peggiore di quella ottima di al massimo lo 0.5%. Inoltre l'algorimo fornirà la soluzione in un tempo che è polinomiale per l'$\epsilon$ scelto. È importante qui notare che sebbene $\epsilon$ possa essere scelto a piacere, esso è una costante: si sceglie a priori e non può cambiare.

Visto come funzione di $\epsilon$, il tempo di esecuzione potrebbe non essere polinomiale. Quanto più piccolo diventa $\epsilon$ tanto più cresce il tempo di esecuzione. In questa ottica $\epsilon$ è un parametro e la polinomialità dell'algoritmo di approssimazione va valutata anche in funzione di $\epsilon$; più precisamente per avere un algoritmo pienamente efficiente il tempo di esecuzione deve essere polinomiale, oltre che nella taglia $n$ dell'istanza, anche in $1/\epsilon$ (visto che per trovare soluzioni migliori $\epsilon$ deve decrescere).

Tuttavia questo non è un problema dal punto di vista dell'applicazione in casi reali nei quali è spesso sufficiente una buona approssimazione, ad esempio $\epsilon = 0.005$, e quindi non c'è bisogno di rendere $\epsilon$ infinitamente piccolo.

Gli algoritmi di approssimazione che operano in questo modo vengono detti *schemi di approssimazione pienamente polinomiali* (FPTAS, Fully Polinomial-Time Approximation Schemes). Chiaramente trovare uno schema di approssimazione pienamente polinomiale è molto desiderabile, ma non sempre possibile. Un problema che ammette un FPTAS è il problema dello zaino.

Nel problema dello zaino, abbiamo $n$ oggetti $1, \ldots, n$ che dobbiamo stipare in uno zaino. Ogni oggetto ha un peso $w_i$ ed un valore $v_i$. C'è un limite massimo $W$ al peso totale che lo zaino può sostenere. L'obiettivo è quello di inserire un sottoinsieme $S$ di oggetti nello zaino in modo tale da massimizzare il valore totale $\sum_{i \in S} v_i$ e rispettare il vincolo sul peso $\sum_{i \in S} w_i \leq W$.

*Algoritmo di programmazione dinamica.* Il problema dello zaino, assumendo che i pesi $w_i$ siano degli interi, può essere risolto con un algoritmo di programmazione dinamica che opera in in tempo $O(n^2 W)$, dove $W$ è la capacità dello zaino. Questo algoritmo è quello che, probabilmente, è stato studiato nei corsi di base di algoritmi. Tale algoritmo è *pseudopolinomiale*[2] in quanto pur essendo polinomiale in $n$ e lineare in $W$, si ha che $W$ può essere esponenziale in $n$. Quindi l'algoritmo è efficiente solo per valori piccoli di $W$.

[2] Si ricordi la discussione fatta nella Sezione 1.8.

Descriviamo ora un altro algoritmo, sempre di programmazione dinamica, che risolve il problema dello zaino in tempo $O(n^2 v_{max})$, dove $v_{max} = \max_i\{v_i\}$, assumendo che i valori $v_i$ siano degli interi (per approfondimenti si veda il capitolo su Algoritmi Approssimati di [20]). Ovviamente anche questa variante continua ad essere pseudopolinomiale, ma può essere sfruttata per progettare uno schema di approssimazione pienamente polinomiale.

I sottoproblemi li definiamo in base a due parametri: un indice $i$ e un valore "obiettivo" $V$: $\mathrm{OPT}(i, V)$ è la più piccola capacità per la quale è possibile inserire nello zaino un sottoinsieme degli oggetti $\{1, 2, \ldots, i\}$ ottenendo un valore di almeno $V$. Si noti che avendo definito il problema in questo modo dobbiamo considerare la possibilità che non sia risolvibile: se il valore totale degli oggetti disponibli, $\sum_{i=1}^{n} v_i$, è più piccolo di $V$ allora, indipendentemente dalla capacità dello zaino, non potremo mai raggiungere un valore totale di almeno $V$. E se per il problema iniziale possiamo ragionevolmente assumere che $\sum_{i=1}^{n} v_i \geq V$ (in altre parole possiamo ignorare i valori di $V$ per cui tale vincolo non è soddisfatto), non vale lo stesso per i sottoproblemi. Per codificare la possibilità che il problema non sia risolvibile useremo il valore di $\infty$ per la capacità dello zaino.

I sottoproblemi sono definiti per $i = 0, \ldots, n$ e per $V = 0; \ldots, \sum_{j=1}^{i} v_j$. Poichè $\sum_{j=1}^{i} v_j < nv_{max}$ la grandezza della tabella è al massimo $n^2 v_{max}$ e conseguetemente il tempo di esecuzione dell'algoritmo è $O(n^2 v_{max})$.

Consideriamo il sottoproblema $(i, V)$. Se $\sum_{i=1}^{n} v_i < V$ allora $OPT(i, V) = \infty$ in quanto non c'è modo di risolverlo. Se, invece, $\sum_{i=1}^{n} v_i \geq V$, allora la soluzione ottima $OPT(i, V)$ è legata a quella di sottoproblemi più piccoli nel seguente modo:

- Se $i \notin OPT(i, V)$ allora $OPT(i, V) = OPT(i-1, V)$
- Se $i \in OPT(i, V)$ allora $OPT(i, V) = w_i + OPT(i-1, \max\{0, V - v_i\})$.

La presenza del $\max\{0, V - v_i\}$ è dovuta al fatto che $V - v_i$ potrebbe essere minore di 0 e ciò non avrebbe senso e quindi lo sostituiamo con 0 (equivalentemente si potrebbe definire $OPT(i, v) = 0$ per tutti i $v < 0$).

Per definizione di $v_{max}$, si ha che$\sum_{j=1}^{i} v_j \leq nv_{max}$, quindi avremo al massimo $n^2 v_{max}$ sottoproblemi per cui l'algoritmo necessita tempo $O(n^2 v_{max})$. La soluzione al problema originale sarà il massimo valore $V$ per il quale $\mathrm{OPT}(n, V) \leq W$.

Consideriamo un esempio. Abbiamo uno zaino di capacità $W = 60$ e 3 oggetti di peso $w_1 = 40$, $w_2 = 20$ e $w_3 = 50$ e di valore $v_1 = 1$, $v_2 = 1$ e $v_3 = 3$. Decidendo di usare le righe per l'indice $i$ e le colonne per i valori $V$, la tabella ha dimensione $4 \times 6$. Possiamo inizializzarla riempiendo la prima colonna con degli 0 e il resto della prima riga con $\infty$:

| | $V = 0$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $i = 0$ | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 1 | 0 | | | | | |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |

Calcoliamo ora i valori delle soluzioni ottime dei sottoproblemi.

- $(1, 1)$: Quindi $\sum_{j=1}^{i} v_j = 1 \geq V = 1$, quindi il problema è risolvibile.

$$OPT(1, 1) = \min\{OPT(0, 1), 40 + OPT(0, 0)\} = \min\{\infty, 40\} = 40.$$

- $(1, 2)$: Quindi $\sum_{j=1}^{i} v_j = 1 < V = 2$, quindi il problema non è risolvibile. $OPT(1, 2) = \infty$. Chiaramente lo stesso varrà per il resto della riga 1.
- $(2, 1)$: Quindi $\sum_{j=1}^{i} v_j = 2 \geq V = 1$, quindi il problema è risolvibile.

$$OPT(2, 1) = \min\{OPT(1, 1), 20 + OPT(1, 0)\} = \min\{40, 20\} = 20.$$

- $(2, 2)$: Quindi $\sum_{j=1}^{i} v_j = 2 \geq V = 2$, quindi il problema è risolvibile.

$$OPT(2, 2) = \min\{OPT(1, 2), 20 + OPT(1, 1)\} = \min\{\infty, 60\} = 60.$$

- $(2, 3)$: Quindi $\sum_{j=1}^{i} v_j = 2 < V = 3$, quindi il problema non è risolvibile. $OPT(2, 3) = \infty$. Chiaramente lo stesso varrà per il resto della riga 2.

- $(3,1)$: Quindi $\sum_{j=1}^{i} v_j = 5 \geq V = 1$, quindi il problema è risolvibile.

$$OPT(3,1) = \min\{OPT(2,1), 50 + OPT(2, \max\{0,-2\})\} = \min\{20, 50\} = 20.$$

- $(3,2)$: Quindi $\sum_{j=1}^{i} v_j = 5 \geq V = 2$, quindi il problema è risolvibile.

$$OPT(3,2) = \min\{OPT(2,2), 50 + OPT(2, \max\{0,-1\})\} = \min\{60, 50\} = 50.$$

- $(3,3)$: Quindi $\sum_{j=1}^{i} v_j = 5 \geq V = 3$, quindi il problema è risolvibile.

$$OPT(3,3) = \min\{OPT(2,3), 50 + OPT(2,0)\} = \min\{\infty, 50\} = 50.$$

- $(3,4)$: Quindi $\sum_{j=1}^{i} v_j = 5 \geq V = 4$, quindi il problema è risolvibile.

$$OPT(3,4) = \min\{OPT(2,4), 50 + OPT(2,1)\} = \min\{\infty, 70\} = 70.$$

- $(3,5)$: Quindi $\sum_{j=1}^{i} v_j = 5 \geq V = 5$, quindi il problema è risolvibile.

$$OPT(3,5) = \min\{OPT(2,5), 50 + OPT(2,2)\} = \min\{\infty, 110\} = 110.$$

Per cui la tabella finale è:

| | $V=0$ | 1 | 2 | **3** | 4 | 5 |
|---|---|---|---|---|---|---|
| $i=0$ | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 1 | 0 | 40 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 2 | 0 | 20 | 60 | $\infty$ | $\infty$ | $\infty$ |
| 3 | 0 | 20 | 50 | **50** | 70 | 110 |

Per trovare la soluzione al problema iniziale dobbiamo considerare l'ultima riga e prendere la capacità più grande inferiore a $W = 60$. In questo esempio è 50 e corrisponde al valore $V = 3$. Questo valore è stato ottenuto selezionando il solo terzo oggetto (come succede in qualsiasi algoritmo di programmazione dinamica con un po' di *bookkeeping* si può risalire dal valore nella tabella alla soluzione ottima). Quindi la soluzione ottima è quella di prendere il solo terzo oggetto con peso $w_3 = 50 < W = 60$ e valore $v_3 = 3$.

*Algoritmo approssimato.* L'idea è quella di sfruttare l'approssimazione per rendere piccolo il valore di $v_{max}$. Infatti accettando di lavorare su dati approssimati, e quindi di avere una soluzione approssimata, possiamo "scalare" i valori di un fattore $b$ in modo tale da rendere ragionevolmente piccolo il valore di $v_{max}$. Più nel dettaglio, sia $Z = (v_i, w_i, W)$ il problema di partenza. Consideriamo il problema $\tilde{Z} = (\tilde{v_i}, w_i, W)$, ottenuto arrotondando ogni valore $v_i$ a $\tilde{v_i} = \lceil \frac{v_i}{b} \rceil b$. Osserviamo che il problema $\tilde{Z}$ è "equivalente" al problema $\overline{Z} = (\overline{v_i}, w_i, W)$, dove $\bar{v_i} = \lceil \frac{v_i}{b} \rceil$. I problemi $\tilde{Z}$ e $\overline{Z}$ sono equivalenti nel senso che possiamo risolvere $\overline{Z}$ e la soluzione ottima fornita per tale problema è ottima anche per $\tilde{Z}$, cambia solo il valore che è scalato di un fattore $b$. Di contro, la soluzione ottima per $\tilde{Z}$ potrebbe non essere ottima per il problema originario $Z$, in quanto i valori $\tilde{v_i}$ sono "approssimazioni" dei valori originali $v_i$. Tuttavia la soluzione

ottima per $\tilde{Z}$ è una soluzione approssimata per $Z$. Scegliendo opportunamente il fattore $b$ potremo ottenere dei valori abbastanza piccoli di $\overline{v_i}$ per fare in modo che $O(n^2\overline{v}_{max})$ sia sufficientemente piccolo (polinomiale in $n$) e allo stesso tempo assicurare che l'approssimazione della soluzione per $Z$ sia abbastanza buona.

Iniziamo con l'osservare che:

**Lemma 3.5.1** *Per ogni $i$ si ha $v_i \leq \tilde{v}_i \leq v_i + b$.*

Dimostrazione. Immediato dalla definizione di $\tilde{v}_i = \lceil \frac{v_i}{b} \rceil b$. □

Inoltre si ha che

**Lemma 3.5.2** *Il problema dello zaino sui valori $\tilde{v}_i$ ed il problema dello zaino sui valori $\overline{v_i}$ hanno lo stesso insieme di soluzioni ottime, ed il valore di una soluzione ottima nei due casi differisce per un fattore $b$.*

Dimostrazione. Il lemma è immediato in quanto i pesi non cambiano mentre i valori sono tutti scalati di un fattore $b$. □

Si noti che il Lemma 3.5.2 ci permette di risolvere il problema $\tilde{Z}$ risolvendo $\overline{Z}$. A questo punto possiamo descrivere l'algoritmo di approssimazione per $Z$.

**Algorithm 15:** ZainoApprox($Z = (w, v, \epsilon)$)

$b = \frac{\epsilon}{2n} v_{max}$
Risolvi il problema dello zaino $\bar{Z} = (\bar{v}_i, w_i, W)$; sia $S$ la soluzione
Restituisci l'insieme $S$

**Lemma 3.5.3** *L'insieme $S$ restituito dall'algortimo* ZainoApprox *è tale che $\sum_{i \in S} w_i \leq W$.*

Dimostrazione. Questo fatto è immediato in quanto abbiamo arrotondato i valori ma i pesi sono gli stessi quindi la soluzione esatta al problema arrotondato $\tilde{Z}$ è comunque una soluzione ammissibile per il problema originale $Z$. □

Vediamo ora il tempo di esecuzione.

**Lemma 3.5.4** *L'algoritmo* ZainoApprox *calcola una soluzione in tempo polinomiale per $\epsilon > 0$ fissato.*

Dimostrazione. Calcolare il valore di $b$ richiede tempo costante. La risoluzione del problema sui valori $\overline{v}$ richiede tempo $O(n^2\overline{v}_{max})$ dove $\overline{v}_{max} = \max_i \bar{v}_i$.

Ma

$$\bar{v}_{max} = \left\lceil \frac{v_{max}}{b} \right\rceil = \left\lceil \frac{2nv_{max}}{\epsilon v_{max}} \right\rceil = \left\lceil \frac{2n}{\epsilon} \right\rceil = O(n\epsilon^{-1}).$$

Quindi il tempo di esecuzione totale è $O(n^2\bar{v}_{max}) = O(n^3\epsilon^{-1})$. Dato che $\epsilon$ è fissato, cioè è costante, abbiamo un tempo di esecuzione polinomiale in $n$, più precisamente $O(n^3)$ con una costante nascosta nella notazione $O$ che è tanto più grande quanto più piccolo è $\epsilon$. □

Ci rimane da valutare la bontà della soluzione approssima. Che garanzie abbiamo sull'approssimazione ottenuta?

**Lemma 3.5.5** *Sia $A$ la soluzione calcolata dall'algoritmo* ZainoApprox *e sia $S^*$ una soluzione ottima per $Z$. Si ha che $(1+\epsilon)\sum_{i\in A} v_i \geq \sum_{i\in S^*} v_i$.*

Dimostrazione. Si noti che $A$ è ottima per $\tilde{Z}$ e $\overline{Z}$ (ma potrebbe non esserlo per $Z$). Poichè $A$ è ottima per $\tilde{Z}$, si ha che

$$\sum_{i\in S^*} \tilde{v}_i \leq \sum_{i\in A} \tilde{v}_i. \tag{3.4}$$

Usando il Lemma 3.5.1 si ha che:

$$\begin{aligned}
\sum_{i\in S^*} v_i &\leq \sum_{i\in S^*} \tilde{v}_i \text{ (Lemma 3.5.1)} && (3.5)\\
&\leq \sum_{i\in A} \tilde{v}_i \text{ (Eq. 3.4)} && (3.6)\\
&\leq \sum_{i\in A} (v_i + b) \text{ (Lemma 3.5.1)} && (3.7)\\
&\leq nb + \sum_{i\in A} v_i \ (|A| \leq n). && (3.8)
\end{aligned}$$

Osserviamo che dalla precedente catena di disuguaglianze possiamo estrapolare la disuguaglianza

$$\sum_{i\in A} \tilde{v}_i \leq nb + \sum_{i\in A} v_i$$

dalla quale si ha

$$\sum_{i\in A} v_i \geq \sum_{i\in A} \tilde{v}_i - nb \tag{3.9}$$

Osserviamo inoltre che

$$\sum_{i\in A} \tilde{v}_i \geq \tilde{v}_{max}.$$

Questo perchè la soluzione $A$ è ottima per i valori $\tilde{v}_i$ e quindi il suo valore deve essere maggiore del valore che otteniamo scegliendo solo l'oggetto con valore massimo (stiamo assumendo che $w_i \leq W$, per tutti gli oggetti: gli oggetti che hanno un peso maggiore di $W$ possono essere eliminati a priori). Ma $\tilde{v}_{max} = \lceil \frac{v_{max}}{b} \rceil\, b = \lceil \frac{2n}{\epsilon} \rceil b$, quindi

$$\sum_{i\in A} \tilde{v}_i \geq \lceil \frac{2n}{\epsilon} \rceil b.$$

Mettendo insieme la (3.9) e la precedente disuguaglianza si ha che

$$\begin{aligned}
\sum_{i\in A} v_i &\geq \sum_{i\in A} \tilde{v}_i - nb\\
&\geq \left\lceil \frac{2n}{\epsilon} \right\rceil b - nb\\
&\geq \frac{2n}{\epsilon} b - nb\\
&= nb\left(\frac{2}{\epsilon} - 1\right)
\end{aligned}$$

Una semplice analisi della funzione $2\epsilon^{-1} - 1$ mostra che essa è $\geq \epsilon^{-1}$ per $\epsilon \leq 1$ (assumere che $\epsilon$ sia piccolo non è un problema visto che vorremo scegliere $\epsilon$ piccolo per otterene buone approssimazioni). Pertanto per $\epsilon \leq 1$ si ha che

$$\sum_{i \in A} v_i \geq \frac{nb}{\epsilon}$$

cioè che

$$nb \leq \epsilon \sum_{i \in A} v_i.$$

Ricordando la relazione 3.8, si ha che

$$\sum_{i \in S^*} v_i \leq \sum_{i \in A} v_i + nb \leq \sum_{i \in A} v_i + \epsilon \sum_{i \in A} v_i \leq (1 + \epsilon) \sum_{i \in A} v_i$$

cioè che

$$\sum_{i \in A} v_i \geq \frac{\sum_{i \in S^*} v_i}{1 + \epsilon}.$$

□

Dunque per il problema dello zaino esiste uno schema di approssimazione pienamente polinomiale (FPTAS). Un FPTAS è il massimo che possiamo aspettarci da un problema NP-hard (assumendo che $\mathcal{P} \neq \mathcal{NP}$): ci permette di ottenere una soluzione con un'approssimazione buona a piacere usando sempre tempo polinomiale. Sono pochi i problemi che ammettono un FPTAS.

## 3.6 *Note bibliografiche*

Approfondimenti sugli argomenti presentati in questo capitolo possono essere trovati nel Capitolo 11 di KT2014 [20], nel Capitolo 35 di CLRS2009 [8] e in V2001 [33].

## 3.7 *Esercizi*

1. Fornire un esempio di grafo per il quale l'algoritmo ApproxVertexCover fornisce sempre una soluzione subottimale.

2. Fornire un esempio di grafo per il quale l'algoritmo ApproxVertexCover fornisce sempre una soluzione ottima.

3. Fornire un algoritmo efficiente (ottimale) per il problema VertexCover nel caso in cui l'input sia un albero.

4. Consideriamo il seguente approccio gredy per il problema VertexCover: inserire nell'insieme ricoprente il nodo con il più alto numero di archi incidenti e quindi cancellare tali archi prima di operare un nuovo inserimento. Fornire un esempio che mostra che tale algoritmo non è 2-approssimato.

5. Il Lemma 1.5.7 suggerisce un algoritmo approssimato per il problema IndependentSet basato sull'algoritmo ApproxVertexCover: usare ApproxVertexCover per trovare un insieme ricoprente approssimato $S$ e restituire come soluzione approssimata del probelma IndependentSet i nodi che non sono in $S$. Analizzare tale algoritmo e mostrare che potrebbe non avere lo stesso fattore di approssimazione di ApproxVertexCover. Dare una giustificazione e individuare la condizione sotto la quale l'algortimo proposto garantisce una soluzione 2-approssimata.

6. L'analisi dell'algoritmo ApproxWeightedVC è stretta oppure può essere migliorata? Giustificare la risposta.

7. Nell'algoritmo approssimato per il problema WeightedVC basato sulla programmazione lineare, l'insieme ricoprente è definito come $S = \{i | x_i \geq 0.5\}$. Il signor Sergio Aumenta sostiene che usando $S = \{i | x_i \geq 0.75\}$ si ottiene un algoritmo approssimato migliore. La signora Elvira Diminuisce, invece, sostiene che un'approssimazione migliore si ottiene usando $S = \{i | x_i \geq 0.25\}$. Cosa rispondi a Sergio e Elvira?

8. L'algoritmo ApproxLoadInOrder fornisce una soluzione il cui valore è al massimo 2 volte l'ottimo. Ci sono casi in cui può restituire la soluzione ottima? Se sì fornire un esempio, se no argomentare il perchè.

9. L'algoritmo ApproxLoadInOrder fornisce una soluzione il cui valore è al massimo 2 volte l'ottimo. Il valore dipende dall'ordine in cui i lavori sono elencati e quindi allocati. Supponiamo di avere un modo per calcolare la soluzione ottima. Data la soluzione ottima, è possibile riordinare i lavori in modo tale che ApproxLoadInOrder produca la soluzione ottima?

10. Supponi che un corriere di spedizione espressa utilizzi dei furgoni per spedire $n$ pacchi da una sede $A$ ad una sede $B$. Ogni furgone ha una capacità $C$ e i pacchi da spedire, il cui peso verrà indicato con $w_1, w_2, \ldots, w_n$, vengono caricati sul furgone in ordine di arrivo e quando un forgone non può contenere il prossimo pacco (la somma dei pesi supera $C$), viene fatto partire e si inizia a caricare un nuovo furgone. Tale algoritmo è un algoritmo di approssimazione.

    (a) Fornire un esempio in cui non viene calcolata la soluzione ottima.

    (b) Provare che l'algoritmo è 2-approssimato.

11. Un tuo amico chimico ti pone il seguente problema. Sta studiando delle molecole $M_1, M_2, \ldots, M_n$ e deve condurre degli esperimenti su ognuna di esse. Questi esperimenti sono costosi e quindi vorrebbe selezionare un sottoinsieme rappresentativo fatto dal minor numero possibile di molecole sfruttando il fatto che le molecole sono note e che è possibile definire una distanza di similarità fra di esse $d(M_i, M_j)$. L'idea è che se due molecole sono abbastanza simili, cioè $d(M_i, M_j) \leq \delta$, per una fissata soglia $\delta$, allora basta condurre gli esperimenti su una di esse per avere informazioni su entrambe. Aiuta il tuo amico a fornire un algoritmo di approssimazione per selezionare un insieme di molecole sulle quali condurre gli esperimenti. (Possiamo assumere che per ogni molecola $M_i$ ce ne sia almeno un'altra $M_j$ con $d(M_i, M_j) \leq \delta$;

se così non fosse per una qualche molecole $M_i$, tale molecola deve necessariamente essere oggetto degli esperimenti, quindi il problema non si porrebbe e la si potrebbe escludere dall'insieme di input.)

12. Si consideri l'algoritmo ApproxCentriConR. Tale algoritmo seleziona una qualsiasi città fra quelle non ancora cancellate come prossimo centro. Si consideri la seguente variante: si sceglie una qualsiasi città e si considera la circonferenza di raggio $r^*$ in essa centrata; il prossimo centro viene scelto su tale circonferenza. L'algoritmo continua ad essere valido con la stessa approssimazione? Se sì, spiegare il perchè. Se no, dire se può essere fatto funzionare con una approssimazione diversa.

# 4
# *Algoritmi Randomizzati*

L'idea di usare la casualità negli algoritmi può sembrare strana di primo acchito: già abbiamo spesso la percezione che i computer si comportino in maniera poco comprensibile (naturalmente ciò è dovuto a errori di vario tipo), vogliamo aggiungere ulteriori problemi con scelte casuali? In realtà le cose stanno diversamente e le scelte casuali possono essere di aiuto nella risoluzione di un problema. In questo capitolo vedremo come delle scelte casuali possano aiutarci a risolvere un problema che non è risolvibile in altro modo o a risolvere più efficientemente un problema che con un algoritmo deterministico richiederebbe una soluzione più costosa. Come quasi sempre accade, a un vantaggio corrisponde uno svantaggio. I benefici portati dalle scelte casuali, dovranno essere "pagati" con una piccola probabilità di insuccesso, che però può essere controllata e quindi assume valori accettabili in pratica. Inoltre l'insuccesso può anche essere non molto dannoso, come ad esempio impiegare il tempo che comunque sarebbe stato necessario nel caso pessimo.

## *4.1 Introduzione*

Un algoritmo è definito come un processo deterministico (una sequenza di passi ben definiti) che a partire da un input produce un output, come schematizzato nella Figura 4.1.


Figura 4.1: Algoritmo deterministico

Un'algoritmo deterministico produce sempre lo stesso output $y = f(x)$ su un dato input $x$. Per un algoritmo deterministico occorre provare che esso produce sempre la soluzione (corretta) e tipicamente si richiede che lo faccia in un tempo ragionevole (polinomiale nella grandezza dell'input).

Un algoritmo randomizzato è un algoritmo che fa delle scelte casuali: ha a disposizione un generatore di numeri casuali che può utilizzare durante l'esecuzione. La Figura 4.2 rappresenta in modo schematico un algoritmo randomizzato.

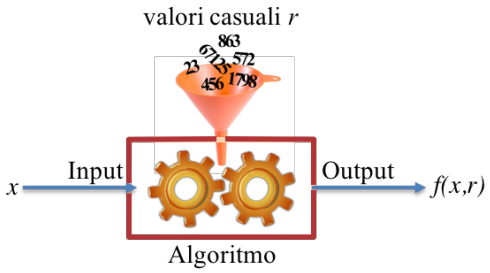



Figura 4.2: Algoritmo randomizzato

L'uso di valori casuali fa sì che l'output $y = f(x, r)$ non dipenda più esclusivamente dall'input $x$, come succede per il caso deterministico, ma anche dai valori casuali $r$ utilizzati durante l'esecuzione.

Le scelte casuali possono migliorare il comportamento dell'algoritmo. Se si pensa all'analisi del caso medio e del caso pessimo si capisce il perchè. Un algoritmo deterministico può avere un comportamento molto inefficiente su dei particolari input. Tuttavia questi input che causano il comportamento molto inefficiente possono essere individuati proprio perchè l'algoritmo fa delle scelte deterministiche. Magari lo stesso algoritmo si comporta bene mediamente, e quindi solo in qualche raro caso manifesta un comportamento inefficiente. Fare delle scelte casuali ci permette di "mescolare" i dati in input. Questo fa sì che non si possa individuare a priori il caso pessimo e pertanto il comportamento atteso dell'algoritmo è quello del caso medio. Ci sono anche altre situazioni in cui la casualità è di aiuto. Ad esempio nei sistemi distribuiti, di cui parleremo successivamente, è utile per uscire da situazioni di impasse dovute alla simmetria del sistema.

### 4.1.1 *Un esempio semplice*

Consideriamo un esempio semplice: il gioco delle tre carte. In questo gioco chi distribuisce le carte, che per i nostri scopi assumeremo essere onesto, ha tre carte che vengono mischiate e posizionate di dorso sul tavolo. Delle tre carte due non hanno valore mentre una sì. Lo scopo del gioco è quello di indovinare quale è la carta che ha valore. Per giocare potremo scegliere prima una carta, poi un'altra carta e infine quella restante; ogni volta che decidiamo di scegliere una carta paghiamo 1€. Quando scopriamo la carta di valore otteniamo una vincita di 2€. Quindi se indoviniamo la carta con un solo tentativo abbiamo un guadagno di 1€, se la indoviniamo con due tentativi andiamo in pari, se la indoviniamo con tre tentativi perdiamo 1€. Supponiamo di dover giocare a questo gioco con un algoritmo che decide in quale ordine scoprire le carte. Si noti come in questo esempio l'input $x$ è la posizione delle carte che può variare a ogni partita in quanto viene decisa da chi mette le carte sul tavolo. L'algoritmo (deterministico) invece deve essere stabilito una volta per tutte e non potrà cambiare. Quindi, un

algoritmo deterministico non può far altro che scegliere una determinata sequenza e chiedere di scoprire le carte in quell'ordine. Ad esempio un algoritmo $A_1$ potrebbe chiedere di scoprire le carte in questo ordine: $2, 3, 1$. Un altro algoritmo $A_2$ potrebbe usare l'ordine $1, 3, 2$ (in questo esempio semplice esistono solo 6 diversi algoritmi deterministici). Qualunque sia l'algoritmo, che assumiamo sia noto a tutti quindi anche a chi distribuisce le carte, è possible fare in modo che l'algoritmo perda sempre 1€: basta posizionare la carta vincente nella posizione che l'algoritmo scoprirà per ultima. Poichè lo scopo di chi distribuisce le carte è quello di guadagnare soldi, piazzerà la carta vincente sempre nell'ultima posizione scelta dall'algoritmo deterministico. Pertanto il "guadagno" dell'algoritmo è una perdita netta di 1€ a partita. Quindi se giochiamo $n$ partite perderemo $n$€.

Diamo adesso all'algoritmo la possibilità di fare delle scelte casuali. Grazie a questa capacità l'algoritmo ora non deve più stabilire a priori una determinata sequenza per scoprire le carte ma può decidere l'ordine con cui scoprire le carte durante l'esecuzione, cioè *dopo* che le tre carte sono state sistemate sul tavolo. Adesso chi distribuisce le carte, che come prima conosce l'algoritmo ma non le scelte casuali perchè queste verranno fatte al momento dell'esecuzione, non può più sistemare la carta vincente nell'ultima posizione che l'algoritmo chiederà di scoprire! Quale è adesso il guadagno atteso? Si noti che mentre nel caso precedente, poichè veniva forzato a ogni esecuzione il caso pessimo, in ogni partita l'algoritmo scopriva tutte e tre le carte. Ora c'è la possibilità che l'algoritmo scopra la carta vincente anche al primo o al secondo tentativo. Assumendo che la posizione della carta vincente sia scelta uniformemente fra le tre possibili, si ha che in media l'algoritmo indovinerà al primo tentativo $1/3$ delle volte e al secondo tentativo un altro $1/3$ delle volte (e ovviamente per il restante $1/3$ al terzo tentativo). Quindi il costo di $m$ partite è di $\frac{m}{3} + \frac{2m}{3} + \frac{3m}{3} = 2m$. Poichè le vincite ammontano a $2m$, il guadagno atteso è 0.

Dunque, con l'algoritmo deterministico si ha una perdita netta di 1€ a partita, mentre con un algoritmo randomizzato si va in pari[1].

[1] Questo è un buon motivo per non giocare a questo gioco: se chi distribuisce le carte è onesto dobbiamo aspettarci di non guadagnare nulla.

### 4.1.2 *Un esempio più complesso*

Come altro esempio consideriamo il problema dell'ordinamento e analizziamo il comportamento dell'algoritmo QuickSort. Tale algoritmo divide il problema in due sottoproblemi in base a un elemento perno. Se la scelta di questo perno divide il problema, ad ogni chiamata ricosiva, in due sottoproblemi di pari grandezza, l'algoritmo risulta più efficiente e ordina l'input in $O(n \log n)$. Se invece la scelta del perno genera una suddivisione in due problemi molto sbilanciati allora l'algoritmo necessita di tempo $\Theta(n^2)$. Quindi la scelta del perno gioca un ruolo fondamentale. Usando delle scelte deterministiche, qualunque sia la scelta (es. il primo elemento, oppure l'ultimo, oppure quello centrale) sarà possibile fornire una sequenza di input che causa sempre un perno inefficiente. Se invece la scelta del perno viene fatta in modo casuale, allora non si potrà a priori determinare un input per cui l'algoritmo è inefficiente. Ovviamente l'algoritmo potrà comunque essere inefficiente, e magari adesso potrà esserlo su degli input per i quali la versione deterministica si comporta meglio, ma il punto cruciale è

che adesso a causare il comportamento inefficiente sono esclusivamente le scelte casuali. E siccome l'algoritmo si comporta bene mediamente, il comportamento atteso della versione randomizzata è quello efficiente. L'altro punto importante da notare e che è possibile fare una scelta casuale del perno in quanto l'algoritmo QuickSort funziona per qualunque scelta del perno. Quindi possiamo sceglierlo casualmente senza nessun problema. Fra poco vedremo più in dettaglio la versione randomizzata di QuickSort.

### *4.1.3 Vantaggi e svantaggi*

Dunque, l'uso di scelte casuali toglie la possibilità a chi fornisce l'input di generare sempre il caso pessimo. Questo non è l'unico vantaggio che deriva dall'uso della casualità negli algoritmi. Ad esempio, nei sistemi distribuiti, l'uso della casualità da parte dei singoli processi può ridurre la quantità di informazioni da comunicare. La randomizzazione in questi casi è un utilissimo strumento per poter superare i problemi dovuti alla *simmetria* delle parti in gioco.

Come spesso accade c'è un rovescio della medaglia: il vantaggio ottenuto con la randomizzazione lo si paga con uno svantaggio che ovviamente deriva anche esso dall'uso della randomizzazione. Una classificazione tipica che si fa degli algoritmi randomizzati è in base alla natura dello svantaggio e gli algoritmi vengono partizionati in due classi.

- Algoritmi che possono sbagliare, cioè dare una risposta errata[2]. Lo svantaggio qui è evidente.
- Algoritmi che non sbagliano; lo svantaggio è che l'algoritmo può non essere così efficiente come ci si aspetta e che addirittura in alcuni casi potrebbe non fornire la risposta continuando l'esecuzione all'infinito (se la fornisce essa è corretta)[3].

[2] Questo tipo di algoritmo randomizzato viene chiamato di tipo *Monte Carlo*.

[3] Questo tipo di algoritmo randomizzato viene chiamato di tipo *Las Vegas*.

Ovviamente affinchè l'algoritmo sia utile è necessario che in entrambi i casi la probabilità con cui lo svantaggio si manifesta deve essere sufficientemente bassa.

L'uso della casualità negli algoritmi richiede nozioni di probabilità. In alcuni casi gli algoritmi randomizzati sono basati effettivamente su conoscenze di argomenti probabilistici molto complessi. Tuttavia spesso è sufficiente un minimo di tali nozioni per poter comprendere molti algoritmi randomizzati, alcuni anche di fondamentale importanza.

## *4.2 Randomness*

Prima di procedere con lo studio degli algoritmi randomizzati, facciamo una breve, informale riflessione sui valori casuali di cui gli algoritmi hanno bisogno. Cosa è un valore casuale? Un valore casuale è un valore che non è predicibile. Supponiamo di considerare singoli bit (se ci servono numeri più grandi possiamo usare sequenze di bit). Un bit $b$ è casuale se a priori non possiamo dire nulla sul suo valore e nel momento in cui lo dobbiamo usare il bit $b$ varrà 0 con probabilità $1/2$ e ovviamente 1 con la stessa probabilità. Più in generale una variabile $X$ che può assumere $n$ valori è casuale se a priori non possiamo dire nulla sul suo valore e nel momento in cui la dobbiamo

usare ognuno degli $n$ possibili valori ha la stessa probabilità, $1/n$, di essere quello assunto dalla variabile. In altre parole la distribuzione di probabilità sui possibili valori deve essere quella uniforme[4]. I computer sono dotati di generatori di valori casuali, ma tali generatori non sono veramente casuali! La cosa non dovrebbe sorprendere perchè, per fortuna, tutto ciò che un computer fa è deterministico, quindi non c'è spazio per l'imponderabile. I generatori presenti sui computer *simulano*, in vari modi, la generazione di numeri casuali. In molti casi questa simulazione fornisce delle proprietà sufficienti (i numeri sembrano abbastanza casuali e, per esempio, possono essere usati per implementare un videogioco) in altri casi le proprietà garantite non sono sufficienti (ad esempio per applicazioni crittografiche per le quali è fondamentale avere dei numeri veramente casuali).

[4] Più in generale una variabile casuale non deve necessariamente avere un distribuzione uniforme. Per i nostri scopi considereremo solo variabili casuali uniformemente distribuite.

Il problema della generazione di numeri casuali è un problema di estrema importanza; il lettore potrà approfondire questo aspetto su altri testi. Per i nostri scopi assumeremo di avere a disposizione un generatore di numeri veramente casuali. Più formalmente assumeremo che gli algoritmi randomizzati abbiano accesso a una funzione $\texttt{random}(n)$ che restituisce un valore casuale uniformemente distribuito fra 1 e $n$. Gli algoritmi potranno sfruttare questa funzione dalla quale potranno attingere numeri casuali. Nello pseudocodice utilizzeremo la notazione

$$x \leftarrow \texttt{random}(n)$$

per indicare che stiamo utilizzando il generatore di numeri casuali che assegna ad $x$ un valore casuale uniformemente distribuito fra 1 e $n$. Per comodità estenderemo tale notiazione ad insiemi: dato un insieme $S$ di $n$ elementi la notazione

$$x \leftarrow \texttt{random}(S)$$

indicherà la scelta casuale di uno degli elementi di $S$. Questo equivale a scgeliere un valore casuale $i = \texttt{random}(n)$ e poi selezionare l'$i$-esimo elemento dell'insieme.

## 4.3 *Cenni di probabilità*

Se lanciamo un dado, la probabilità che esca un numero più grande di 4 è $1/3$. Questo perchè ci sono 6 possibili risultati del lancio, un numero fra 1 e 6 e solo 2 di questi risultati ci danno un numero più grande di 4. La probabilità è il numero di caso favorevoli, 2 in questo caso, diviso il numero di casi totali, 6 in questo caso. Più genericamente considereremo un esperimento, nell'esempio precedente era il lancio di un dado, che ha un certo numero di possibili risultati, che chiameremo eventi elementari, che nel caso del dado sono i 6 possibili numeri. Un evento (non elementare) invece è un sottoinsieme deell'insieme di tutti i possibili eventi. Nell'esempio fatto precedentemente l'evento "uscita di un numero maggiore di 4" è dato dall'insieme $\{5, 6\}$.

Procedendo un po' più formamlmente definiamo uno spazio di probabilità come un insieme $\Sigma = \{e_1, e_2, ...\}$, finito o infinito, di eventi elementari che descrivono i possibili risultati di un esperimento. Un evento $X$ (non elementare) è un sottoinsieme di $\Sigma$.

Nell'esempio del dado $\Sigma = \{1, 2, 3, 4, 5, 6\}$. Esempi di eventi sono $X = \{e_3\}$, $X' = \{1, 3, 5\}$, $X = \{5, 6\}$; l'evento $X$ è l'uscita del numero 3, l'evento $X'$ è l'uscita di un

numero dispari, l'evento $X''$ è l'uscita di un numero maggiore di 4. Un esempio di spazio di probabilità infinito è relativo al seguente esperimento: si lanci una monetina fina a ottenere "croce". Indicando con "T" (testa) e "C" (croce) i due possibili risultati di un singolo lancio si ha che $\Sigma = \{C, TC, TTC, TTTC, TTTTC, \ldots\}$. Un esempio di evento è "il numero di T è dispari", cioè $X = \{TC, TTTC, TTTTTC, \ldots\}$.

**Lemma 4.3.1** *Consideriamo un esperimento casuale in cui la probabilità di successo dell'esperimento è $p$. Se ripetiamo l'esperimento e ogni ripetizione è indipendente dalle precedenti, allora il numero atteso di ripetizione per avere il primo successo è $1/p$.*

## 4.4 Problemi di ordinamento

Iniziamo con due problemi affini: l'ordinamento di un insieme e la ricerca del $k$-esimo elemento di un insieme non ordinato. Il secondo di questi problemi può essere facilmente ridotto al primo: per trovare il $k$-esimo elemento di un insieme non ordinato basta ordinarlo e prendere l'elemento nella $k$-esima posizione.

### 4.4.1 QuickSort randomizzato

Iniziamo dal problema dell'ordinamento e consideriamo l'algoritmo QuickSort. Esso prende in input una sequenza di elementi $a_1, a_2, \ldots, a_n$ sui quali è definita una relazione di ordine totale $\leq$ e fornisce in output gli elementi ordinati, cioè trova una permutazione $a'_1, a'_2, \ldots, a'_n$ degli elementi tali che $a'_i \leq a'_j$ per $i < j$. Per semplicità assumeremo che tutti gli elementi siano distinti: modificare l'algoritmo per gestire anche elementi uguali è semplice e non richiede nessuna nuova idea ma solo delle accortezze non difficili da implementare. QuickSort opera nel seguente modo: seleziona uno specifico elemento della sequenza di input; poichè l'algoritmo funziona qualunque sia l'elemento scelto, solitamente si sceglie il primo elemento $a_1$. Tale elemento, detto *perno*, viene utilizzato per suddividere la sequenza di input in due parti $S^<$ e $S^>$, la prima contenente tutti gli elementi più piccoli e la seconda tutti gli elementi più grandi lasciando fuori dalla divisione il perno. Quindi QuickSort potrà essere usato ricorsivamente[5] su $S^<$ e $S^>$ per ottenere l'insieme ordinato che è dato dall'output relativo a $S^<$, seguito dal perno $a_1$, seguito dall'output relativo a $S^>$.

[5] Volendo considerare la possibilità di elementi uguali, essi possono essere trattati come il perno, cioè esclusi dalla ricorsione e poi dati in output insieme al perno.

La complessità di QuickSort è data dalla seguente relazione di ricorrenza:

$$T(n) = T(|S^<|) + T(|S^>|) + O(n),$$

dove il termine $O(n)$ è quello necessario a effettuare la partizione. La soluzione di questa relazione di ricorrenza dipende da come viene diviso l'insieme di input nei due sottoinsiemi. Il caso migliore si ha quando i due sottoinsiemi $S^<$ e $S^>$ hanno la stessa grandezza, circa $n/2$, e quindi la soluzione della ricorrenza è $\Theta(n \log n)$. Il caso peggiore, che misura la complessità dell'algoritmo, si ha invece quando i due sottoinsiemi sono molto sbilanciati, ad esempio uno è vuoto e l'altro contiene $n - 1$ elementi. In questo caso la soluzione della ricorrenza è $\Theta(n^2)$. Quindi la complessità di QuickSort è $O(n^2)$. É facile costruire sequenze di input che causano il caso pessimo:

un input già ordinato (o quasi) sia in senso ascendente che discendente. Pertanto un "avversario" in grado di decidere l'input può causare sempre il caso pessimo.

Un approccio randomizzato può migliorare QuickSort. L'approccio randomizzato si basa sul fatto che l'algoritmo funziona qualunque sia il perno scelto. Nella versione deterministica si deve necessariamente scegliere una determinata posizione ed usare l'elemento in quella posizione come perno. Questo dà la possibilità a chi sceglie l'input di sceglierlo in modo tale che il perno sia sempre il minimo o il massimo (o quasi) dell'insieme causando una partizione complemetamente (o quasi completamente) sbilanciata. Poichè, come detto, l'algoritmo funziona qualunque sia il perno, avendo a disposizione un generatore di numeri casuali, anzichè fissare la posizione del perno, cosa necessaria per un algoritmo deterministico, possiamo sceglierlo in maniera casuale selezionandolo durante l'esecuzione dell'algoritmo. Pertanto la versione randomizzata differisce da quella deterministica solo per il fatto che il perno viene scelto in maniera casuale.

**Algorithm 16:** RandomQuickSort

```
RandomQuickSort(S)
if |S| sufficientemente piccolo, per esempio 1,2,3 then
    Ordina S con forza bruta
    Fornisci in output gli elementi di S
else
    i ← random(n)
    /* a_i è il perno scelto in modo casuale                    */
    for ogni a_j ∈ S do
        Se a_j < a_i allora S^< = S^< ∪ {a_j}
        Se a_j > a_i allora S^> = S^> ∪ {a_j}
    Chiama ricorsivamente RandomQuickSort(S^<)
    Chiama ricorsivamente RandomQuickSort(S^>)
    Fornisci in output gli elementi di S^<, seguiti da a_i, seguito dagli elementi di
     S^>.
```

Poichè RandomQuickSort differisce da QuickSort solo per la scelta del perno, la relazione di ricorrenza che descrive la complessità dell'algoritmo è la stessa. Quindi sia la complessità del caso migliore che quella del caso peggiore sono le stesse.

Cosa abbiamo guadagnato quindi? La differenza fra i due algoritmi è che per RandomQuickSort il caso pessimo non può più essere determinato dall'input ma può scaturire solo dalla scelte casuali del perno. In altre parole per avere il caso pessimo dobbiamo essere veramente sfortunati (è in gioco la casualità!). Poichè le scelte del perno sono casuali la complessità attesa di RandomQuickSort è quella del caso medio, che risulta essere uguale al caso ottimo, cioè $O(n \log n)$.

L'analisi formale non è semplicissima, pertanto procediamo con delle osservazioni che ci forniscono un'idea sul comportamento atteso dell'algoritmo. Successivamente procederemo con l'analisi formale.

Prima di tutto osserviamo che il tempo di cui necessita l'algoritmo, escludendo

le chiamate ricorsive, è praticamente quello necessario per operare la partizione in quanto tutte le altre operazioni possono essere eseguite in tempo costante e che il tempo necessario per la partizione è lineare nel numero $n$ di elementi da considerare. Non dovrebbe essere difficile convincersi che il tempo necessario è proporzionale al numero di confronti da fare che è esattamente $n-1$ (per operare la partizione è sufficiente confrontare ogni elemento con il perno). Quindi, ignorando il $-1$, stimeremo con $cn$ il tempo necessaro a effettuare la partizione, per una qualche costante $c$.

Se siamo veramente fortunati, il perno divide sempre l'insieme di input in due parti uguali, generando la relazione di ricorrenza

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

la cui soluzione è $O(n \log n)$, mentre se siamo veramente sfortunati la partizione produce un insieme vuoto e uno di $n-1$ elementi, generando la relazione di ricorrenza

$$T(n) = T(n-1) + cn$$

la cui soluzione è $O(n^2)$. Questi sono i due casi estremi che corrispondono al caso migliore e al caso pessimo. Cosa succede in mezzo? Supponiamo che la partizione generi sempre due insiemi leggermente sbilanciati, ad esempio con $1/4$ e $3/4$ degli elementi. La relazione di ricorrenza diventa

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + cn.$$

La soluzione di questa relazione è ancora $O(n \log n)$. Proviamo a sbilanciare maggiormente i due insiemi della ricorsione, ad esempio $1/10$ e $9/10$. La relazione di ricorrenza diventa

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + cn.$$

Nonostante questo sbilanciamento più pronunciato, la soluzione è ancora $O(n \log n)$. In realtà fino a che lo sbilanciamento prevede una frazione (funzione di $n$) da una parte e il resto dall'altro, per quanto piccola possa essere tale frazione, la soluzione alla relazione di ricorrenza sarà sempre $O(n \log n)$.

La soluzione diventerà quadratica solo quando la partizione inserirà in uno dei due sottoinsiemi un numero di elementi più piccolo di *una costante*. Ad esempio se la partizione produce sempre un insieme di 10 elementi ed un insieme di $n-10$ elementi, allora la relazione di ricorrenza diventa

$$T(n) = T(10) + T(n-10) + cn$$

la cui soluzione è $O(n^2)$. Chiaramente anche in questo caso, simmetricamente a quanto detto prima, partizioni leggermente più bilanciate, ad esempio con 100 e $n-100$ elementi o 3000 e $n-3000$ elementi, generano sempre una relazione di ricorrenza la cui soluzione è $O(n^2)$.

Un altro caso interessante da analizzare è il seguente. Supponiamo che la fortuna ci sia amica a corrente alternata: in una iterazione siamo fortunati con una partizione

completamente bilanciata mentre alla successiva siamo sfortunati con una partizione completamente sbilanciata e così via, alternando fortuna a sfortuna. Cosa succede al tempo di esecuzione? Sostituendo $T(n)$ con $F(n)$ e $S(n)$, rispettivamente per il caso fortunato e quello sfortunato, avremo che

$$F(n) = 2S(n/2) + cn,$$

$$S(n) = F(n-1) + cn.$$

Quindi si ha che

$$\begin{aligned} F(n) &= 2S(n/2) + cn \\ &= 2\left(F\left(\frac{n}{2} - 1\right) + cn/2\right) + cn \\ &= 2F\left(\frac{n}{2} - 1\right) + cn + cn \\ &= O(n \log n) \end{aligned}$$

e in modo simile si può vedere che $S(n) = O(n \log n)$.

Dunque l'intuizione ci dice che nella maggior parte dei casi il tempo di esecuzione di RandomQuickSort è dato da una relazione di ricorrenza la cui soluzione è $O(n \log n)$ per cui non dovrebbe sorprendere il fatto che la complessità (attesa) di RandomQuickSort sia $O(n \log n)$. Vediamo questo fatto in maniera più formale.

Sia $T(n)$ una variabile casuale che descrive il tempo di esecuzione di RandomQuickSort su un input di $n$ elementi. Per $k = 0, 1, \ldots, n-1$, sia $X_k$ una variabile (casuale) indicatrice definita da

$$X_k = \begin{cases} 1 \text{ se la partizione è } k : n-k-1 \\ 0 \text{ altrimenti.} \end{cases}$$

Poichè il perno è scelto uniformemente a caso, tutti le possibili partizioni $k : n-k-1$ hanno la stessa probabilità di verificarsi, e qiondi si ha che

$$E[X_k] = \Pr\{X_k = 1\} = 1/n.$$

Quindi si ha che

$$T(n) = \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + n).$$

Pertanto

$$\begin{aligned}
E[T(n)] &= \sum_{k=0}^{n-1} E[X_k(T(k)+T(n-k-1)+n)] \\
&\quad \textit{(per la linearità di E)} \\
&= \sum_{k=0}^{n-1} E[X_k]\cdot E[(T(k)+T(n-k-1)+n)] \\
&\quad \textit{(per l'indipendenza delle scelte casuali)} \\
&= \frac{1}{n}\sum_{k=0}^{n-1} E[(T(k)+T(n-k-1)+n)] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} n \\
&= n + \frac{2}{n}\sum_{k=0}^{n-1} E[(T(k)] \\
&\quad \textit{(perchè le due sommatorie sono uguali)}
\end{aligned}$$

A questo punto abbiamo una relazione di ricorrenza per $E[T(n)]$ che possiamo risolvere per esempio con il metodo della sostituzione. Con tale metodo dobbiamo provare a "indovinare" una soluzione e verificare se la scelta è corretta. Per quanto detto prima possiamo ragionevolmente tentare con $E[T(n)] \leq an\log n$, per una qualche costante $a$.

$$\begin{aligned}
E[T(n)] &= n + \frac{2}{n}\sum_{k=0}^{n-1} E[(T(k)] \\
&\leq n + \frac{2}{n}\sum_{k=0}^{n-1} ak\log k \\
&= n + \frac{2a}{n}\sum_{k=1}^{n-1} k\log k \\
&\leq n + \frac{2a}{n}\int_{x=1}^{n} x\log x \\
&\quad \left(\text{ricordando che } \int x\log x = \frac{x^2\log x}{2} - \frac{x^2}{4}\right) \\
&= n + \frac{2a}{n}\left[\frac{x^2\log x}{2} - \frac{x^2}{4}\right]_1^n \\
&\leq n + \frac{2a}{n}\left(\frac{n^2\log n}{2} - \frac{n^2}{4} + \frac{1}{4}\right) \\
&= n + an\log n - \frac{an}{2} + \frac{a}{2n}.
\end{aligned}$$

A questo punto basta scegliere la costante $a$ in modo tale che la relazione sia verificata a partire da un determinato valore di $n$ in poi. Scgliendo $a = 2$ la relazione è sempre

verificata, per ogni $n \geq 1$.

Una versione alternativa di RandomQuickSort, è quella in cui il perno viene forzato a essere un elemento che fa ottenere sempre un buon bilanciamento. L'algoritmo è equivalente, richiede qualche piccolo sforzo in più per la sua implementazione, ma è più facile da analizzare. Pertanto è interessante presentarlo ed analizzarlo. Diremo che un perno è *centrale* se ci sono almeno un quarto degli elementi che sono più piccoli del perno ed almeno un quarto degli elementi che sono più grandi del perno, cioè entrambi $S^<$ e $S^>$ hanno cardinalità almeno $n/4$. In RandomQuickSortBil forzeremo il perno a essere un elemento centrale.

**Algorithm 17:** RandomQuickSortBil

```
RandomQuickSortBil(S)
if |S| sufficientemente piccolo, per esempio 1,2,3 then
    Ordina S con forza bruta
    Fornisci in output gli elementi di S
else
    while Non troviamo un perno centrale do
        i ← random(n)
        /* a_i è il perno scelto in modo casuale                    */
        for ogni a_j ∈ S do
            Se a_j < a_i allora S^< = S^< ∪ {a_j}
            Se a_j > a_i allora S^> = S^> ∪ {a_j}
        if |S^<| ≥ |S|/4 e |S^>| ≥ |S|/4 then
            abbiamo trovato un perno centrale (a_i)
    Chiama ricorsivamente RandomQuickSortBil(S^<)
    Chiama ricorsivamente RandomQuickSortBil(S^>)
    Fornisci in output gli elementi di S^<, seguiti da a_i, seguito dagli elementi di
     S^>.
```

Con RandomQuickSortBil siamo sempre "fortunati" nella ricorsione in quanto il caso pessimo che si verifica è uno sbilanciamento di $n/4 : 3n/4$. Come abbiamo già visto questo sbilanciamento genera comunque una ricorrenza la cui soluzione è $O(n \log n)$. Tuttavia questa partizione fortunata viene pagata con il ciclo **while** che cerca un perno centrale: la ricorsione verrà fatta solo dopo aver trovato un perno centrale, il che significa che non sappiamo quante partizioni dobbiamo provare prima di trovarne una bilanciata. Tuttavia possiamo facilmente stimare il numero medio di tentativi che dobbiamo fare per trovare un perno centrale. Infatti, poichè il perno è scelto uniformemente a caso, si ha che la probabilità di trovare un perno centrale è di $1/2$ in quanto ci sono esattamente $n/2$ elementi che sono centrali (sono gli elementi dalla posizione $\frac{n}{4}$ alla posizione $\frac{3}{4}n$ nella sequenza ordinata).

Quindi per il Lemma 4.3.1 si ha che il numero atteso di iterazioni nel ciclo **while** prima di trovare un elemento centrale è 2. Questo significa che il tempo necessario al partizionamento, in media, raddoppia. Pertanto la relazione di ricorrenza del tempo di

esecuzione atteso di RandomQuickSortBil è

$$E[T(n)] = E[T(n/4)] + E[T(3n/4)] + 2cn,$$

e pertanto $E[T(n)] = O(n \log n)$.

### 4.4.2 *k-esimo ordine statistico*

Consideriamo adesso un problema collegato a quello dell'ordinamento: la ricerca del $k$-esimo ordine statistico. Dato un insieme di $n$ elementi $S = \{a_1, a_2, \ldots, a_n\}$, il $k$-esimo ordine statistico è l'elemento che si troverrebbe nella posizione $k$ se gli elementi fossero elencati in ordine crescente. Come per il caso di QuickSort, supporremo che gli $a_i$ siano tutti diversi tra loro; anche in questo caso la presenza di elementi uguali non è un problema, ma assumendo che gli elementi siano tutti diversi semplifichiamo l'analisi; le idee fondamentali che introdurremo per risolvere il problema continuano a essere valide anche se i numeri non sono tutti diversi.

Il *minimo* dell'insieme $S$ è il primo ordine statistico, mentre il *massimo* è l'ultimo ($n$-esimo) ordine statistico. La *mediana* dell'insieme $S$ è l'ordine statistico che si trova al centro della lista ordinata. Per $n$ dispari, la mediana è il $k$-esimo ordine statistico con $k = (n+1)/2$. Un valore di $n$ pari crea un problema per la definizione di centro in quanto, tolta la mediana, i due insiemi non possono avere la stessa cardinalità. Quindi uno dei due insieme dovrà avere un elemento in più e questo da spazio a due possibili mediane, cioè il $k$-esimo ordine statistico $k = n/2$ oppure con $k = (n/2) + 1$; una qualsiasi delle due definizioni va bene.

Dovrebbe essere ovvio che è facile trovare il $k$-esimo ordine statistico in $O(n \log n)$: basta ordinare $S$ e prendere l'elemento in posizione $k$. La domanda che ci poniamo è: è davvero necessario ordinare gli elementi per risolvere il problema? In altre parole è possibile progettare un algoritmo che trova il $k$-esimo ordine statistico in meno di $O(n \log n)$? Vedremo un algoritmo randomizzato[6] che risolve il problema in $O(n)$; ovviamente tale tempo è quello atteso, le scelte casuali potrebbero comunque causare il caso pessimo che richiede $O(n \log n)$. L'algoritmo RandomSelect è un algoritmo randomizzato che risolve il problema del $k$-esimo ordine statistico. RandomSelect$(S, 1)$ corrisponde al minimo, RandomSelect$(S, n)$ al massimo e la mediana è data da RandomSelect$(S, n/2)$ per $n$ pari o da RandomSelect$(S, (n+1)/2)$ per $n$ dispari.

La struttura di base dell'algoritmo RandomSelect è la seguente. Scegliamo un elemento $a_i \in S$ che chiameremo perno; la scelta del perno è fatta in modo casuale. Quindi partizioneremo i restanti elementi dell'insieme $S$ in due sottoinsiemi $S^< = \{a_j | a_j < a_i\}$ e $S^> = \{a_j | a_j > a_i\}$. In base alle cardinalità di tali insiemi potremo determinare quale dei due contiene il $k$-esimo elemento (che potrebbe anche essere proprio $a_i$) e poi procedere ricorsivamente se non lo abbiamo ancora trovato.

[6] Esiste anche un algoritmo deterministico che risolve il problema in $O(n)$. Tuttavia quello randomizzato è molto più semplice.

**Algorithm 18:** RANDOMSELECT

```
RandomSelect(S, k)
i ← random(n)
/* a_i è il perno scelto in modo casuale                    */
for ogni a_j ∈ S do
    Se a_j < a_i allora S^< = S^< ∪ {a_j}
    Se a_j > a_i allora S^> = S^> ∪ {a_j}
if |S^<| = k − 1 then
    Il perno a_i è il valore cercato
if |S^<| ≥ k − 1 then
    /* L'elemento cercato si trova in S^<                   */
    Chiama ricorsivamente RandomSelect(S^<, k)
if |S^<| < k − 1 then
    /* L'elemento cercato si trova in S^>                   */
    ℓ = |S^<|
    Chiama ricorsivamente RandomSelect(S^>, k − 1 − ℓ)
```

**Lemma 4.4.1** *L'algoritmo* RANDOMSELECT *restituisce il k-esimo ordine statistico.*

DIMOSTRAZIONE. Si noti che se $|S| = 1$ allora si deve avere $k = 1$ e l'algoritmo restituisce l'unico elemento di $S$. Se $|S| > 1$, l'algoritmo ricorsivamente cerca l'elemento in un insieme più piccolo aggiustando oppurtunamente il valore di $k$. Il fatto che il perno venga scelto in modo casuale non ha nessuna influenza sulla correttezza dell'algoritmo. Pertanto l'algoritmo termina e restituisce la risposta corretta. □

Quale è il tempo di esecuzione di RANDOMSELECT? Ovviamente dipende da quanto sono i grandi i sottoproblemi delle chiamate ricorsive. Se per esempio il perno fosse sempre la mediana dell'insieme su cui si effettua la chiamata ricorsiva allora si avrebbe $T(n) = T(n/2) + cn$. La soluzione di questa relazione di ricorrenza è $T(n) = O(n)$.

Se invece il perno fosse sempre il minimo (o il massimo), allora la relazione di ricorrenza sarebbe $T(n) = T(n-1) + cn$, la cui soluzione è $T(n) = \Theta(n^2)$.

Chiaramente è difficile scegliere la mediana come perno visto che uno degli obiettivi di RANDOMSELECT è proprio quello di trovare la mediana. Tuttavia anche un elemento che è vicino al centro comporta una relazione di ricorrenza la cui soluzione è $O(n)$. Infatti supponiamo che la grandezza dell'insieme su cui si effettua la chiamata ricorsiva sia tale da garantire che ci siano almeno $\epsilon n$ elementi sia più piccoli che più grandi del perno, per una qualsiasi costante $\epsilon > 0$. Allora si ha che l'insieme su cui si chiama ricorsivamente l'algoritmo non può avere più di $(1-\epsilon)n$ elementi. Quindi il tempo di esecuzione soddisfa $T(n) \leq T((1-\epsilon)n) + cn$. Ricordando la serie geometrica $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$, si ha che la soluzione di questa ricorrenza è

$$\begin{aligned} T(n) &\leq T((1-\epsilon)n) + cn \\ &\leq T((1-\epsilon)^2 n) + c(1-\epsilon) + cn \\ &\leq \dots \end{aligned}$$

$$\begin{aligned} &= cn\left(1 + (1-\epsilon) + (1-\epsilon)^2 + \ldots\right) \\ &\leq \tfrac{1}{\epsilon} cn. \end{aligned}$$

Quindi, i casi che generano tempi di esecuzione non lineari sono quelli in cui il perno è estremamente vicino al minimo o al massimo. Questo suggerisce che la scelta di un perno casuale nella maggior parte dei casi funziona bene.

Per analizzare l'algoritmo in modo più formale utilizzeremo la seguente terminologia: diremo che l'algoritmo è nella *fase* $j$, quando il numero di elementi dell'insieme in considerazione, che si riduce a ogni chiamata ricorsiva, è maggiore di $n(\frac{3}{4})^{j+1}$ ma minore o uguale a $n(\frac{3}{4})^j$. Per valutare la complessità dell'algoritmo daremo un limite al tempo speso nella fase $j$. Inoltre diremo che un elemento è *centrale* rispetto all'insieme in considerazione, se ci sono almeno un quarto di elementi più piccoli ed almeno un quarto di elementi più grandi.

Dunque, se il perno è un elemento centrale, nella successiva chiamata ricorsiva il numero di elementi in considerazione diminuirà di almeno un quarto e pertanto la fase corrente terminerà. La probabilità che il perno sia centrale è $1/2$ in quanto metà degli elementi sono centrali. Quindi per il Lemma 4.3.1 si ha che il numero atteso di iterazioni prima di scegliere un perno centrale è 2. Quindi in ogni fase $j$ il numero atteso di iterazioni eseguite nella fase $j$ è 2.

Per concludere l'analisi, denotiamo con $X$ la variabile casuale che descrive il numero di passi eseguiti dall'algoritmo, e denotiamo con $X_j$ il numero di passi eseguiti dall'algoritmo nella fase $j$. Quindi $X = X_0 + X_1 + X_2 + \ldots$. Quando l'algoritmo si trova nella fase $j$ l'insieme preso in considerazione contiene al massimo $n(\frac{3}{4})^j$ elementi, poichè il lavoro svolto dall'algoritmo in una singola iterazione è proporzionale a tale numero, si ha che per ogni iterazione durante la fase $j$, l'algoritmo spende al massimo tempo $cn(\frac{3}{4})^j$, per una costante $c$. Sappiamo già che il numero atteso di iterazioni per ogni fase è 2 quindi si ha che $E[X_j] \leq 2cn(\frac{3}{4})^j$. Per la linearità della media, si ha che $E[X] = \sum_j E[X_j]$ e pertanto

$$E[X] = \sum_j E[X_j] \leq \sum_j 2cn\left(\frac{3}{4}\right)^j = 2cn\sum_j\left(\frac{3}{4}\right)^j \leq 8cn = O(n).$$

## 4.5 *Taglio minimo globale*

Dato un grafo non direzionato $G = (V, E)$, definiamo un *taglio* di $G$ una partizione di $V$ in due insiemi non vuoti $A$ e $B$. Per un taglio $(A, B)$ di $G$, la grandezza di $(A, B)$ è il numero di archi che attraversano il taglio, cioè che hanno un vertice in $A$ e l'altro in $B$. Un taglio minimo globale è un taglio di grandezza minima.

L'aggettivo globale sta a enfatizzare il fatto che il taglio può essere qualunque partizione di $V$ in due sottoinsiemi, non c'è nessuna sorgente e nessun pozzo, come invece avviene per i problemi di flusso. L'unica restrizione è che gli insiemi $A$ e $B$ non devono essere vuoti, altrimenti avremmo un taglio di grandezza 0 ed il problema non avrebbe senso.

Il taglio minimo globale può essere visto come un parametro di *robustezza* del grafo.

Infatti è il più piccolo numero di archi che si deve cancellare per disconnettere il grafo stesso.

Ovviamente il problema è in qualche modo legato ai problemi di flusso per i quali sappiamo che il taglio minimo gioca un ruolo fondamentale; il teorema del flusso massimo e taglio minimo ci dice che il massimo flusso che può arrivare dalla sorgente alla destinazione è uguale alla somma delle capacità del taglio minimo che separa la sorgente dalla destinazione. Quindi non dovrebbe sorprendere il fatto che possiamo usare l'algoritmo per il calcolo del massimo flusso, e quindi del taglio minimo, per risolvere anche il problema del taglio minimo globale.

**Lemma 4.5.1** *Esiste un algoritmo polinomiale per trovare un taglio minimo globale di un grafo non direzionato $G$.*

Dimostrazione. Sia $G = (V, E)$ il grafo. Possiamo sfruttare l'algoritmo per il calcolo del massimo flusso che di fatto trova anche il taglio minimo. Dobbiamo però prestare attenzione a due differenze fra i problemi.

La prima è dovuta al fatto che il massimo flusso gestisce delle capacità su degli archi direzionati. Questo è una caratteristica in più che possiamo facilmente gestire e sfruttare. Creiamo un nuovo grafo $G' = (V', E')$ con $v' = v$ e per ogni arco $(u, v) \in E$, inseriamo in $E'$ sia l'arco $(u, v)$ che l'arco $(v, u)$ e a entrambi assegniamo capacità 1. Questo fa sì che il valore del flusso massimo sia proprio il numero di archi che attraversano il taglio e quindi la grandezza del taglio nella definizione del problema del taglio minimo globale.

La seconda differenza è che abbiamo bisogno di una sorgente e di un pozzo per poter calcolare il flusso massimo. Supponiamo di scegliere una qualsiasi coppia di nodi $a, b$ come sorgente e pozzo. A questo punto usando l'algoritmo per il calcolo del massimo flusso individueremo il taglio minimo fra tutti i tagli che separano $a$ da $b$. Chiaramente questo potrebbe non essere il taglio minimo globale, per il quale magari $a$ e $b$ sono nello stesso insieme. Tuttavia il nodo $a$ deve necessariamente essere separato da un qualche altro nodo! Quindi se ripetiamo l'esecuzione dell'algoritmo per $n - 1$ volte, usando ogni volta $a$ come sorgente e tutti gli altri $n - 1$ nodi come pozzo, individueremo il taglio minimo globale che è semplicimente il taglio minimo fra tutti gli $n - 1$ trovati nelle ripetizioni dell'algoritmo di massimo flusso.

La complessità totale è data da $n - 1 = O(n)$ moltiplicato la complessità dell'algoritmo di massimo flusso. Essendo quest'ultima polinomiale, anche la complessità del calcolo del taglio minimo globale è polinomiale. □

Quanto appena detto farebbe supporre che il calcolo del taglio minimo globale è più difficile del problema del massimo flusso in quanto per risolvere il primo dobbiamo risolvere $n - 1$ istanze del secondo. Non è così, nel senso che si può risolvere il problema del taglio minimo globale in maniera più efficiente sfruttando la randomizzazione.

### *4.5.1 Algoritmo di contrazione degli archi*

Descriveremo l'algoritmo nella sua forma più semplice. Questa forma, sebbene sia comunque polinomiale, non è quella più efficiente; successive ottimizzazioni hanno

reso l'algoritmo più efficiente. Tuttavia questa forma ci permette di capire facilmente l'idea di base.

L'algoritmo di contrazione opera collassando i nodi: due nodi vengono fusi insieme e diventano un solo nodo. Questo crea archi multipli fra nodi collassati insieme. Quindi per gestire questa possibilità considereremo il grafo $G = (V, E)$ come un *multigrafo*: in un multigrafo è possibile avere più di un arco fra due nodi. Quindi l'insieme $E$ è un multi-insieme in cui un elemento $(u, v)$ può comparire più di una volta. Inizialmente il grafo $G$ è il grafo di input, quindi non è un multigrafo. Durante l'esecuzione dell'algoritmo però i nodi verrano collassati e questo potrà creare un multigrafo.

Ad esempio, consideramo la Figura 4.3. Il grafo iniziale è un grafo normale con 4 nodi uniti da alcuni archi. Collassando il nodo $a$ con il nodo $b$, poichè sia $a$ che $b$ avevano un arco con $d$, nel nuovo grafo il nodo $\{a, b\}$ avrà due archi che lo collegano a $d$.

Se da questo nuovo grafo si collassa il nodo $\{a, b\}$ con il nodo $d$ creando il nodo $\{a, b, d\}$, questo nodo avrà due archi con il nodo $c$. Anche in questo caso archi che per effetto del collassamento di due nodi diventerebbero degli archi che collegano un nodo a sè stesso sono stati rimossi. Ad esempio, nel collassare $a$ e $b$ l'arco $(a, b)$ viene rimosso; nel collassare $\{a, b\}$ con $d$ i due archi $(\{a, b\}, d)$ vengono rimossi. I nodi creati nei collassamenti verranno chiamati *supernodi*. Un supernodo $w$ può essere identificato con l'insieme dei nodi $S(w)$ che sono stati collassati per ottenerlo.

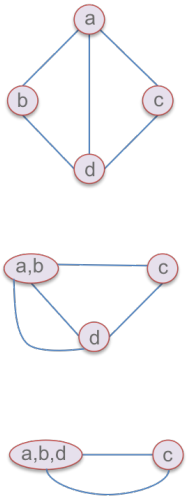



Figura 4.3: Contrazione di nodi e multigrafo

L'algoritmo di contrazione procede collassando nodi su archi scelti a caso fino a che il multigrafo contiene due soli nodi $v_1$ e $v_2$. A questo punto l'algoritmo termina e produce come output i due insieme $S(v_1)$ e $S(v_2)$.

**Algorithm 19:** EdgeContraction$(G = (V, E))$

```
Per ogni nodo v, S(v) = {v}
if V ha due soli nodi v1 e v2 then
    Fornisci in output S(v1) e S(v2)
else
    (u, v) ← random(E)
    Sia G' il grafo ottenuto collassando u e v nel nodo z
    S(z) = S(u) ∪ S(v)
    EdgeContraction(G')
```

Analizziamo l'algoritmo. È possibile implementare l'algoritmo in tempo $O(n^2)$. Tuttavia una singola esecuzione dell'algoritmo non sarà sufficiente in quanto la probabilità di trovare una taglio minimo è bassa ma sufficientemente alta da poter essere sfruttata ripetendo varie volte l'algoritmo. Quanto è grande questa probabilità? Vediamo.

Prima di tutto osserviamo che che EdgeContraction restituisce un taglio. Infatti i due supernodi finali rappresentano due sottoinsiemi di $V$ che sono una partizione di $V$ visto che tutti i nodi sono stati collassati o nell'uno o nell'altro. Tale taglio è il risultato di scelte casuali, dipende infatti dalla sequenza di archi scelti per collassare i nodi. Quindi è possibile, se siamo fortunati, che il taglio fornito dall'algoritmo sia proprio un taglio minimo. Ovviamente è anche possibile, e più probabile, che venga

restituito un taglio non minimale. Ma quale è la probabilità di ognuno di queste due possibilità? Si potrebbe pensare che la probabilità che il taglio restituito sia minimo è estremamente piccola. Ovviamente non è alta, ma è sufficientemente non piccola (è solo polinomialmente piccola) da poter ripetere l'algoritmo un numero polinomiale di volte e scegliendo il taglio più piccolo fra tutti quelli trovati riusciamo a trovare un taglio minimo globale con alta probabilità.

**Lemma 4.5.2** *L'algoritmo* EdgeContraction *produce un taglio minimo globale con probabilità almeno* $1/\binom{n}{2}$.

Dimostrazione. Sia $(A, B)$ un taglio minimo globale e sia $k$ la sua grandezza. In altre parole l'insieme di archi con un vertice in $A$ ed un vertice in $B$, che denoteremo con $F$, ha cardinalità $k$.

Sia $(A', B')$ il taglio restituito dall'algoritmo. Vogliamo provare un limite inferiore di $1/\binom{n}{2}$ alla probabilità che $(A', B') = (A, B)$.

L'algoritmo non troverà il taglio $(A, B)$ se un arco di $F$ viene selezionato per collassare i suoi due nodi. In tale caso entrambi i nodi finiranno nello stesso supernodo e quindi saranno o entrambi in $A'$ oppure entrambi in $B'$. D'altra parte se nessun arco di $F$ viene selezionato durante l'algoritmo allora si ha che $A' = A$ e $B' = B$.

Poichè la grandezza del taglio minimo globale è $k$, ogni nodo ha almeno $k$ vertici a esso incidenti. Infatti se così non fosse, dato un nodo $v$ con meno di $k$ archi incidenti, il taglio $(\{v\}, V \setminus \{v\})$ sarebbe un taglio con grandezza minore del taglio minimo, che è un assurdo. Quindi possiamo dedurre che il numero di archi nel grafo iniziale è $|E| \geq \frac{1}{2}nk$.

Definiamo $\beta_i$ come il seguente evento: un arco di $F$ viene contratto alla $i$-esima iterazione. L'evento $\alpha_i$ è definito come il complemento di $\beta_i$: nessun arco di $F$ viene contratto alla $i$-esima iterazione dell'algoritmo. Ovviamente si ha che $Pr[\alpha_i] = 1 - Pr[\beta_i]$.

Dunque si ha

$$Pr[\beta_1] \leq \frac{k}{\frac{1}{2}nk} = \frac{2}{n}$$

e pertanto

$$Pr[\alpha_1] \geq 1 - \frac{2}{n}.$$

Per la nostra analisi servirà valutare le probabilità

$$Pr[\alpha_{j+1} | \alpha_1 \cap \alpha_2 \ldots \cap \alpha_j].$$

Consideriamo $Pr[\alpha_2|\alpha_1]$. La condizione $\alpha_1$ significa che nella prima iterazione non è stato selezionato un arco di $F$, quindi gli archi di $F$ sono ancora $k$, mentre i nodi del grafo sono $n - 1$. Pertanto si ha che

$$Pr[\beta_2|\alpha_1] \leq \frac{k}{\frac{1}{2}k(n-1)} = \frac{2}{n-1}$$

e pertanto

$$Pr[\alpha_2|\alpha_1] \geq 1 - \frac{2}{n-1}.$$

Analogamente per le successive iterazioni. Più precisamente dopo ogni contrazione il numero di nodi del grafo diminuisce di un'unità. Quindi dopo $j$ iterazioni, il grafo $G'$ ha esattamente $n - j$ nodi e pertanto il numero di archi in $G'$ è almeno $\frac{1}{2}k(n-j)$. Possiamo quindi generalizzare quanto detto poc'anzi sulla probabilità che un arco di $F$ venga contratto. Quando si opera sul grafo $G'$, all'iterazione $(j+1)$-esima, assumendo che nessun arco di $F$ sia stato contratto nelle precedenti iterazioni, si ha che la probabilità che un arco di $F$ venga contratto è al massimo

$$Pr[\beta_{j+1}|\alpha_1 \cap \alpha_2 \ldots \cap \alpha_j] \leq \frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j}$$

e quindi si ha che

$$Pr[\alpha_{j+1}|\alpha_1 \cap \alpha_2 \ldots \cap \alpha_j] \geq 1 - \frac{2}{n-j}.$$

Come abbiamo già osservato, se durante le $n-2$ iterazioni dell'algoritmo nessun arco di $F$ viene selezionato allora si ha che $A' = A$ e $B' = B$. Quindi ci interessa studiare la quantità

$$Pr[A' = A] = Pr[\alpha_1 \cap \alpha_2 \ldots \cap \alpha_{n-2}].$$

Sfruttando la formula per la probabilità condizionata[7] (usandola $n-2$ volte con $Y = \alpha_i$ per $i = 1, \ldots n-2$) si ha che

[7] $Pr[X|Y] = \frac{Pr[X \cap Y]}{Pr[Y]}$

$$\begin{aligned}
Pr[A' = A] &= Pr[\alpha_1 \cap \alpha_2 \ldots \cap \alpha_{n-2}] \\
&= Pr[\alpha_1] \cdot Pr[\alpha_2 \cap \alpha_3 \ldots \cap \alpha_{n-2}|\alpha_1] \\
&= Pr[\alpha_1] \cdot Pr[\alpha_2|\alpha_1] \cdot Pr[\alpha_3 \cap \ldots \cap \alpha_{n-2}|\alpha_1 \cap \alpha_2] \\
&= \ldots \\
&= Pr[\alpha_1] \cdot Pr[\alpha_2|\alpha_1] \cdot Pr[\alpha_3|\alpha_1 \cap \alpha_2] \cdot \ldots \cdot Pr[\alpha_{n-2}|\alpha_1 \cap \alpha_2 \ldots \cap \alpha_{n-3}] \\
&\geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \ldots \left(1 - \frac{2}{n-j}\right) \ldots \left(1 - \frac{2}{3}\right) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right) \ldots \left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&= \frac{2}{n(n-1)} \\
&= \binom{n}{2}^{-1}.
\end{aligned}$$

□

Il lemma precedente ci dice che in ogni singola esecuzione la probabilità che l'algoritmo EdgeContraction trovi un taglio minimo globale è almeno $\binom{n}{2}^{-1}$, o equivalentemente che la probabilità che l'algoritmo non trovi un taglio minimo globale è al massimo $1 - \binom{n}{2}^{-1}$.

Sebbene la probabilità di insuccesso con una singola esecuzione sia alta è sufficientemente più piccola di 1, e pertanto ripetendo l'algoritmo abbastanza volte, la probabilità che un taglio minimo non venga mai trovato si riduce a valori più interessanti.

Quando ripetiamo l'esecuzione dell'algoritmo per un numero $x$ di volte, poichè ogni esecuzione è indipendente dalle altre, la probabilità di non trovare un taglio minimo in

nessuna delle $x$ esecuzioni è

$$\left(1 - 1/\binom{n}{2}\right)^x.$$

Questa funzione tende a 0 al tendere di $x$ a $\infty$, pertanto possiamo rendere piccola a piacere la probabilità di non trovare un taglio minimo. Ad esempio, se $n = 50$, la probabilità di trovare un taglio minimo con una singola esecuzione è almeno $2/2450 \simeq 0.0008$ e quindi la probabilità di non trovare un trovare un taglio minino in una singola esecuzione è al massimo 0.9992. Ripetendo l'algoritmo $n = 50$ volte, la probabilità di non trovare il taglio minimo è al massimo $0.9992^{50} =\simeq 0.96$; ripetendo l'algoritmo $\binom{n}{2} = 2450$ volte, la probabilità di non trovare il taglio minimo è al massimo $0.9998^{2450} =\simeq 0.14$; ripetendo l'algoritmo $\binom{n}{2} \log n \simeq 2450 \cdot 5.63 \simeq 13793$ volte, la probabilità di non trovare il taglio minimo è al massimo $0.9992^{13793} =\simeq 0.00001$, cioè lo 0.001%.

Tuttavia non possiamo esagerare con il numero di ripetizioni altrimenti l'intero processo diventa inefficiente. Il numero di ripetizioni dell'algoritmo deve essere al massimo polinomiale; $x = \binom{n}{2}$ e $x = \binom{n}{2} \log n$ sono polinomiali in $n$. Analizziamole più analiticamente.

Se $x = \binom{n}{2} = n(n-1)/2$ si ha che la probabilità di insuccesso è $\leq 1/e \simeq 0.37$. Infatti per un tale numero di ripetizioni la probabilità di insuccesso è

$$\left(1 - 1/\binom{n}{2}\right)^{\binom{n}{2}} \tag{4.1}$$

e la funzione $\left(1 - \frac{1}{x}\right)^x$ è una funzione crescente che converge da $1/4$ a $1/e \simeq 0,3678 < 0,37$ al crescere di $x$. Quindi la probabilità di trovare un taglio minimo è almeno $0,73$.

Per avere una probabilità di successo ancora più alta, possiamo usare $x = \binom{n}{2} \log n$, nel qual caso la probabilità di insuccesso è

$$\left(1 - 1/\binom{n}{2}\right)^{\binom{n}{2} \log n} \tag{4.2}$$

e tale funzione vale al massimo $e^{-\log n} = 1/n$. La Figura 4.4 mostra le funzioni (4.1), linea blu, e (4.2), linea rossa.

Per entrambe le scelte di $x$ l'algoritmo è ancora polinomiale. Scegliendo $x = \binom{n}{2} \log n = O(n^2 \log n)$, poichè ogni singola esecuzione dell'algoritmo costa $O(n^2)$ il tempo totale è $O(n^4 \log n)$. Notiamo che rispetto all'algoritmo che risolve il problema usando un algoritmo per il massimo flusso, EdgeContraction ha un tempo di esecuzione peggiore. Infatti il massimo flusso può essere risolto in $O(n^3)$ e quindi ripentendolo $n - 1$ per risolvere il taglio minimo globale avremmo un tempo di esecuzione di $O(n^4)$.

Come abbiamo già detto, esistono versioni ottimizzate che hanno tempi di esecuzione considerevolmente migliori. Una variante semplice è quella di Karger-Stein che risolve il problema con un tempo di esecuzione di $O(n^2 \log n)$ che però fornisce una probabilità di successo più grande, $\Omega(1/\log n)$, per la quale servono solo $O(\log^2 n)$ ripetizioni per avere una ragionevole probabilità di successo ($\geq 1 - 1/poly(n)$). Quindi con questa versione ottimizzata riusciamo a trovare in $O(n^2 \log^3 n)$ tempo il taglio minimo globale con probabilità $\geq 1 - 1/poly(n)$.

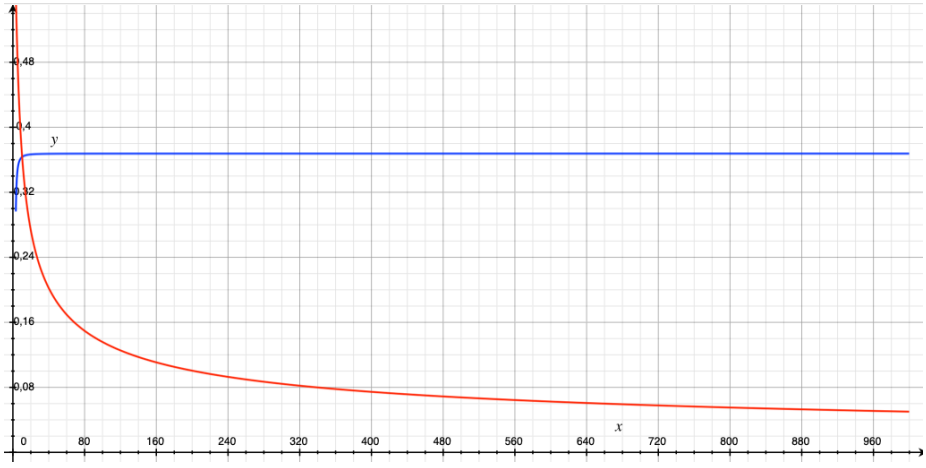



Figura 4.4: Probabilità di insuccesso al crescere di $n$

### 4.5.2 *Numero di tagli minimi*

L'analisi dell'algoritmo EdgeContraction permette di rispondere anche a un'altra domanda: quanti tagli minimi ci sono?

In una rete di flusso è facile vedere che il numero di tagli sorgente-pozzo può essere esponenziale. Ad esempio consideriamo il grafo con nodi $V = \{s, t, v_1, v_2, \ldots, v_n\}$ e archi con capacità 1, $(s, v_i)$ e $(v_i, t)$. Tutti i tagli $s$-$t$ devono avere grandezza $n$ e un qualsiasi sottoinsieme $S$ di $\{v_1, v_2, \ldots, v_n\}$ determina un taglio minimo $(\{s\} \cup S, \{t\} \cup V \setminus S)$. Si veda la Figura 4.5. Quindi esistono $2^n$ tagli minimi.

La situazione è diversa se prendiamo un grafo non direzionato. Se consideriamo un ciclo di $n$ nodi, i tagli minimi hanno grandezza 2 e ce ne sono $\binom{n}{2}$ che si ottengono tagliando due archi qualsiasi, come mostrato nella Figura 4.6. È possibile trovare grafi con un numero di tagli minimi più grande? La risposta è no, e deriva dalla prova del Lemma 4.5.2.

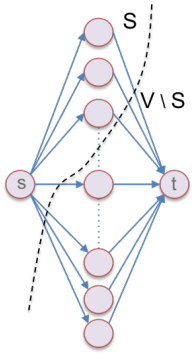



Figura 4.5: Taglio in un grafo $s$-$t$

**Lemma 4.5.3** *In un grafo non direzionato con n nodi, ci sono al massimo $\binom{n}{2}$ tagli minimi globali*

Dimostrazione. Sia $G$ il grafo e siano $C_1, \ldots, C_r$ tutti i tagli minimi globali. Sia $\alpha_i$ l'evento che $C_i$ sia restituito dall'algoritmo EdgeContraction, e sia $\alpha = \bigcup_{i=1}^{r} \alpha_i$.

Nella prova del Lemma 4.5.2 abbiamo visto che $Pr[\alpha_i] \geq 1/\binom{n}{2}$. Poichè gli eventi $\alpha_i$ sono disgiunti, e poichè in ogni singola esecuzione dell'algoritmo viene prodotto un solo taglio, si ha che la probabilità dell'unione degli eventi è uguale alla somma delle probabilità dei singoli eventi e quindi

$$Pr[\alpha] = Pr[\bigcup_{i=1}^{r} \alpha_i] = \sum_{i=1}^{r} Pr[\alpha_i] \geq r/\binom{n}{2}.$$

Ovviamente si ha anche che $Pr[\alpha] \leq 1$ e quindi possiamo concludere che $r \leq \binom{n}{2}$. □

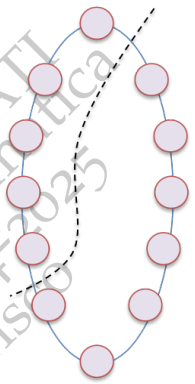


Figura 4.6: Taglio in un ciclo

## 4.6 *Un algoritmo randomizzato per* MAX-3SAT.

Nel capitolo 1 abbiamo studiato il problema 3SAT ed abbiamo stabilito che è un problema $\mathcal{NP}$-completo. Ricordiamo il problema: abbiamo un insieme di $k$ clausole $C_1, \ldots, C_k$, ognuna è la disgiunzione di 3 letterali deifiniti su $n$ variabili $x_1, \ldots, x_n$. Il problema 3SAT è quello di stabilire se esiste un assegnamento di valori alle variabili in modo tale da rendere vere tutte le clausole. Un modo diverso di approcciare questo problema è quello di vederlo come un problema di ottimizzazione: trovare un assegnamento che renda vere il massimo numero possibile di clausole. Chiaramente anche questo problema è difficile, e di fatto è un problema NP-hard, in quanto è chiaro che 3SAT può essere ridotto MAX-3SAT. Il problema MAX-3SAT si presta a essere l'oggetto di un algoritmo di approssimazione.

Anzi, questo è uno di quei casi dove l'algoritmo è di una semplicità estrema per cui non specifichiamo nemmeno lo pseudocodice: l'algoritmo assegna a ogni singola variabile un valore a caso, 0 o 1 con probabilità ciascuno 1/2. Quale è il numero atteso di clausole soddisfatte?

**Lemma 4.6.1** *La soluzione fornita da un assegnamento casuale delle variabili ha un valore che dista dall'ottimo di un fattore moltiplicativo pari a* 7/8*.*

DIMOSTRAZIONE. Sia $Z$ una variabile casuale che denota il numero di clausole soddisfatte. Possiamo decomporre tale variabile nella somma di variabili $Z_i$ dove ogni $Z_i = 1$ se la clausola $C_i$ è soddisfatta, 0 altrimenti. Quindi si ha che $Z = Z_1 + Z_2 + \ldots + Z_k$. Il valore atteso di $Z_i$, $E[Z_i]$ è dato dalla probabilità che la clausola $C_i$ sia vera. Questa probabilità è facile da calcolare. Infatti $C_i$ è la disgiunzione di 3 letterali ed è vera se almeno uno dei 3 letterali ha il valore vero. Poichè tutte le variabili hanno valore vero con probabilità 1/2 anche tutti i letterali, indipendentemente dal fatto che rappresentano una variabile o la sua negazione, hanno probabilità 1/2 di essere veri e la stessa di essere falsi. L'unico caso in cui la clausola è falsa è quello in cui tutti i letterali sono falsi. Ciò avviene con probabilità 1/8. Quindi la clausola $C_i$ è vera con probabilità 7/8, quindi $E[Z_i] = 7/8$. Usando la linearità del valore atteso si ha che $E[Z] = E[Z_1] + E[Z_1] + \ldots + E[Z_k] = \frac{7}{8}k$.

D'altra parte l'assegnamento ottimale può soddisfare al massimo $k$ clausole semplicemente perché non ce ne sono di più. □

La prova che abbiamo appena fornito in realtà ci dice anche di più di quello che abbiamo asserito nell'enunciato del lemma. Infatti poichè il valore atteso del numero di clausole soddisfatte per un assegnamento casuale è 7/8 di quello ottimale, è necessario che esista almeno un assegnamento per il quale effettivamente almeno 7/8 delle clausole sono soddisfatte (altrimenti il valore medio dovrebbe essere più basso). Pertanto si che:

**Lemma 4.6.2** *Per una qualsiasi istanza di* 3SAT *esiste un assegnamento delle variabili che rende vere almeno* 7/8 *delle clausole.*

L'asserzione del precedente lemma è abbastanza sorprendente in quanto non ha nulla a che vedere con la randomizzazione. Ma l'abbiamo ottenuta utilizzando l'analisi di un

algoritmo randomizzato. Questo tipo di risultato è abbastanza comune nell'area della combinatorica: si prova l'esistenza di oggetti con determinate caratteristiche mostrando che una costruzione casuale li produce con una probabilità non nulla. In questi casi si parla di *metodo probabilistico*.

Un'altra osservazione riguardo la precedente asserzione: ogni istanza di 3SAT con al massimo 7 clausole è soddisfattibile. Infatti per il lemma precedente esiste un assegnamento in cui almeno 7/8 delle clausole sono vere. Ma se l'istanza ha $k \leq 7$ clausole, si ha che $\frac{7}{8}k > k - 1$ e quindi per quel particolare assegnamento tutte le clausole sono vere.

Sappiamo dunque che un assegnamento casuale produce un assegnamento che soddisfa mediamente un numero di clausole pari a $\frac{7}{8}k$. Tuttavia non è per nulla garantito che assegnando casualmente i valori alle variabili vengano soddisfatte tante clausole; $\frac{7}{8}k$ è un valore medio. Se un assegnamento non soddisfa almeno $\frac{7}{8}k$ clausole possiamo ripetere l'assegnamento fino a quando non ne troviamo uno (sappiamo che esiste) che soddisfi tante clausole. Ma quante volte dobbiamo ripetere l'assegnamento per avere almeno $\frac{7}{8}k$ clausole vere?

**Lemma 4.6.3** *Per trovare un assegnamento che soddisfi almeno $\frac{7}{8}k$ clausole, il numero medio di ripetizioni che dobbiamo utilizzare è $8k$.*

DIMOSTRAZIONE. Consideriamo come esperimento l'assegnazione dei valori alle variabili e come successo il fatto che ci siano almeno $\frac{7}{8}k$ clausole vere.

Indichiamo con $p_j$ la probabilità che esattamente $j$ clausole siano vere, per $j = 0, 1, \ldots, k$ si ha che il numero medio di clausole vere è

$$0 \cdot p_0 + 1 \cdot p_1 + \ldots k \cdot p_k = \sum_{j=0}^{k} j p_j.$$

Sappiamo già che tale numero è $\frac{7}{8}k$, pertanto si ha che

$$\sum_{j=0}^{k} j p_j = \frac{7}{8}k.$$

D'altra parte, la probabilità di successo dell'esperimento è data da

$$p = \sum_{j \geq \frac{7}{8}k} p_j.$$

Definiamo $k'$ come l'intero più grande tale che $k' < \frac{7}{8}k$. Prima di proseguire analizziamo la relazione fra $k$ e $k'$. Nella seguenmte tabella sono riportati i valori per alcuni $k$ piccoli.

| $k$ | $\frac{7}{8}k$ | $k'$ | $\frac{7}{8}k - k'$ |
|---|---|---|---|
| 0 | 0 | $-1$ | 8/8 |
| 1 | 7/7 | 0 | 7/8 |
| 2 | 14/8 | $1 = 8/8$ | 6/8 |
| 3 | 21/8 | $2 = 16/8$ | 5/8 |
| 4 | 28/8 | $3 = 24/8$ | 4/8 |
| 5 | 35/8 | $4 = 32/8$ | 3/8 |
| 6 | 42/8 | $5 = 40/8$ | 2/8 |
| 7 | 49/8 | $6 = 48/8$ | 1/8 |
| 8 | 56/8 | $6 = 48/8$ | 8/8 |
| 9 | 63/8 | $7 = 56/8$ | 7/8 |
| ... | ... | ... | ... |
| 14 | 98/8 | $12 = 96/8$ | 2/8 |
| 15 | 105/8 | $13 = 104/8$ | 1/8 |
| 16 | 112/8 | $13 = 104/8$ | 8/8 |
| ... | ... | ... | ... |

Non è difficile dimostrare che $\frac{1}{8} \leq \frac{7}{8}k - k'$. Un semplice studio della funzione porta a dimostrare che la funzione ha l'andamento mostrato nella Figura 4.7.

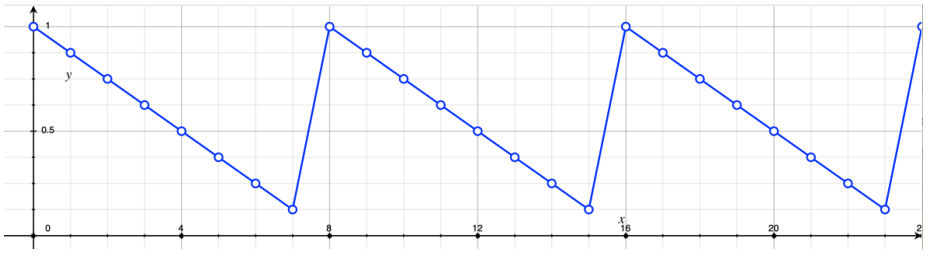



Figura 4.7: Funzione $\frac{7}{8}k - k'$

Si ha che:

$$\begin{aligned}
\frac{7}{8}k &= \sum_{j=0}^{k} jp_j \\
&= \sum_{j<7k/8} jp_j + \sum_{j\geq 7k/8} jp_j \\
&\leq \sum_{j<7k/8} k'p_j + \sum_{j\geq 7k/8} kp_j \\
&= k' \sum_{j<7k/8} p_j + k \sum_{j\geq 7k/8} p_j \\
&= k'(1-p) + kp \\
&\leq k' + kp
\end{aligned}$$

Quindi

$$\frac{7}{8}k - k' \leq kp.$$

Ricordando che $\frac{1}{8} \leq \frac{7}{8}k - k'$, pertanto

$$\frac{1}{8} \leq kp$$

e quindi

$$p \geq \frac{1}{8k}.$$

Dal Lemma 4.3.1 si ha che il numero di ripetizioni prima di avere successo è $1/p$, quindi è $\leq 8k$. □

## 4.7 *Note bibliografiche*

Fonti consultate: KT2014 [20], A2016 [2]. L'algoritmo EdgeContraction è dovuto a Karger [18] mentre la versione ottimizzata a Karger e Stein [19].

## 4.8 *Esercizi*

1. Nell'esempio del gioco delle 3 carte abbiamo usato un algoritmo randomizzato che gira sempre tutte e tre le carte. Il gioco però ammette anche che si giri una sola carta o due carte, il che permetterebbe di risparmiare sulla cifra necessaria per giocare. Ricordando che si paga 1€per ogni giocata e che la vincita vale 2€, perchè abbiamo usato un algoritmo che gira tutte e tre le carte?

2. Consideriamo il gioco delle 4 carte che è come quello delle 3 carte, ma con una carta in più. Anche in questo caso si paga 1€ per girare una singola carta e si vincono 2€ quando si scopre la carta vincente. Supponendo di dover decidere di fissare un numero di carte da scoprire in ogni partita, quale è la strategia migliore e perchè?

3. Consideriamo un ulteriore variante: questa volta giochiamo con 4 carte, ed una vincita di 3€, mentre il costo per girare una carta è sempre di 1€. Cosa cambia? Quale è la strategia migliore?

4. Consideriamo (per l'ultima volta!!!!) il gioco delle 3 carte e questa volta generalizziamo ad $n$ carte e giochiamo $m$ partite. Il costo per scoprire una singola carta è di 1€ e la vincita per l'individuazione della carta vincente è di $v$€, con $v$ che può assumere i valori da 2 a $n$. Ora le strategie possibili sono $n$: girare $k$ carte, dove $k$ può essere un valore fra 1 e $n$. Quale è la strategia migliore?

5. Fornire un esempio di input per il quale QuickSort genera un'alternanza fra caso sfortunato (partizione completamente sbilanciata) e caso frotunato (partizione completamente bilanciata) nella sequenza di chiamate ricorsive. Più precisamente nel primo livello dell'albero della ricorsione si è fortunati, nel secondo livello si è sfortunati (su tutte le chiamate ricorsive), nel terzo livello si è fortunati (su tutte le chiamate ricorsive), e così via. Motivare la risposta.

6. Fornire un algoritmo RandomSelectBil, simile a RandomSelect, sfruttando la stessa idea con la quale si è progettato RandomQuickSortBil partendo da RandomQuickSort. Analizzare il tempo di esecuzione.

7. Nell'algoritmo RandomQuickSortBil un perno è considerato centrale se lo sbilanciamento massimo è di 1/4 (nessuna delle due parti può essere più piccola di 1/4).

Questo significa che metà degli elementi sono centrali. Cosa succede se cambiamo la definizione di perno centrale in modo tale che lo sbilancimaneto massimo sia di 3/8? Quale è la complessità di tempo dell'algorimto?

8. Un'azienda con $n$ dipendenti deve pianificare l'esecuzione di un insieme di $m$ lavori $w_1, \ldots, w_m$, ognuno dei quali deve essere svolto da due dipendenti. L'assegnazione dei lavori ai dipendenti è fissata, cioè per ogni lavoro $w_i$, sono stati assegnati due dipendenti $d_1(w_i)$ e $d_2(w_i)$. Non c'è un vincolo sul numero di lavori a cui un dipendente può essere assegnato. Per svolgere i lavori sono di aiuto 3 specifiche competenze: se il lavoro viene svolto da due dipendenti con competenze diverse il tempo necessario a svolgerlo è minore. Poichè l'azienda deve investire denaro per far acquisire a ognuno dei dipendenti una delle 3 competenze, deve decidere quale competenza far acquisire a ognuno dei dipendenti. Ovviamente l'obiettivo è quello di massimizzare il numero di lavori che verranno svolti da due dipendenti con competenze diverse.

   Si fornisca un algoritmo randomizzato che produce una soluzione $(2/3)$-approssimata. Fornire l'analisi.

9. Consideriamo il problema MAXSAT che generalizza il problema MAX-3SAT studiato in questo capitolo: massimizzare il numero di clausole vere in una formula $\phi$; le clausole della formula possono avere un numero qualsiasi di letterali. Per MAXSAT abbiamo visto un algoritmo randomizzato che forniva un'approssimazione di 7/8. Analizzare il comportamento dell'algoritmo per MAXSAT.

10. Consideriamo una variante dell'algoritmo randomizzato per il problema MAX-3SAT studiato in questo capitolo. L'algoritmo sceglie il valore di ogni variabile assegnado `falso` con probabilità 1/4 e `vero` con probabilità 3/4. Analizzare il comportamento dell'algoritmo proposto.

11. Consideriamo il problema dell'individuazione di un taglio minimo che divide una sorgente $s$ da un destinazione $t$. Potremmo utilizzare l'algoritmo EdgeContraction se si fa in modo che $s$ e $t$ non vengano mai contratti. Basterà, dopo ogni contrazione, rimuovere tutti gli archi che connettono $s$ e $t$. Analizzare questo algoritmo e dire se può essere utile usandolo ripetutamente come EdgeContraction.

# 5
# *Algoritmi Online*

## *5.1 Introduzione*

Normalmente un algoritmo viene progettato per risolvere un problema per il quale l'input è completamente disponibile quando l'esecuzione dell'algoritmo inizia. Per completamente disponibile intendiamo che è disponibile nella sua interezza. Ci sono molte situazioni però dove l'input viene fornito *durante* l'esecuzione e quindi non è completamente noto all'inizio dell'esecuzione, ed è necessario fornire un output (parziale) man mano che si ricevono ulteriori pezzi dell'input. In tali situazioni si parla di problemi e algoritmi *online*. Quindi un algoritmo online riceve una serie di richieste (ognuna è un pezzo dell'input) ad ognuna delle quali deve fornire una risposta (ognuna è un pezzo dell'output) senza conoscere le richieste successive.

Ad esempio, supponiamo che l'input sia una sequenza di interi $x_1, x_2, \ldots, x_n$ che vengono forniti uno per volta. Quindi possiamo immaginare che l'algoritmo proceda per iterazioni, e alla $i$-esima iterazione viene fornito il valore di $x_i$. Immaginiamo di voler trovare il massimo fra gli $x_i$. Un algoritmo offline esaminerebbe tutti gli $x_i$ e individuerebbe il massimo. Nella versione online del problema, però, dobbiamo stabilire se l'elemento $x_i$ è il massimo alla $i$-esima iterazione, quindi senza conoscere i successivi elementi della sequenza di input.

Dovrebbe essere evidente che un problema online è più difficile del relativo problema offline: in quest ultimo caso possiamo prendere delle decisioni "conoscendo il futuro", mentre nel primo caso dobbiamo prenderle senza sapere quali saranno le successive richieste. Quindi è ovvio che un algoritmo offline sarà sempre migliore di un algoritmo online: se per un algoritmo offline proprio non riusciamo a fare meglio possiamo fare esattamente quello che fa un algoritmo online; un algoritmo online, invece, non può fare quello che fa un algoritmo offline in quanto ha meno informazioni a disposizione.

Per valutare quanto è buono un algoritmo online si utilizza l'*analisi competitiva*: cioè si valuta la soluzione trovata dall'algoritmo online confrontandola con quella di un algoritmo offline ottimo. Un algoritmo offline ottimo è un algoritmo che trova la soluzione ottima al problema conoscendo tutto l'input in partenza. Un algoritmo online è tanto più competitivo, quanto più esso approssima la soluzione ottima. In particolare diremo che un algoritmo online è $c$-competitivo se il valore della soluzione che produce si differenzia al massimo per un fattore moltiplicativo $c$ dal valore della soluzione

ottima.

Il fattore di competitività per gli algoritmi online è l'analogo di quello di approssimazione per gli algoritmi approssimati. La differenza è che per l'analisi dell'approssimazione confrontiamo il comportamento di due algoritmi, quello approssimato e quello ottimo, sullo stesso problema, mentre per l'analisi della competitività confrontiamo il comportamento di algoritmi che operano su due problemi diversi, quello online e quello offline. Il fattore di competitività considera il caso pessimo; nella sezione 5.7 vedremo un'approccio alternativo che permette di considerare il caso medio.

Formalmente un problema online può essere descritto nel seguente modo. L'input da fornire a un algoritmo $A$ è dato da una sequenza di richieste $R = r_1, r_2, r_3, \ldots$. Le richieste devono essere soddisfatte nell'ordine in cui vengono ricevute: questo significa che l'algoritmo per ogni richiesta $r_i$, dovrà fornire un output $o_i$ senza conoscere $r_j$ per $j > i$. Produrre l'output $o_i$ ha un costo $Costo_A(o_i)$. Il costo totale dell'algoritmo $Costo_A(R) = \sum_i Costo_A(o_i)$ è la somma dei costi di tutti gli output.

Data una sequenza $R$ di richieste denotiamo on $\textsc{opt}(R)$ il valore della soluzione fornita da un algoritmo offline ottimo. Sia $A$ un algoritmo online e consideriamo problemi di minimizzazione (per questo parliamo di costi). L'algoritmo $A$ è detto $c$-competitivo se esiste una costante $a$ tale che

$$Costo_A(R) \leq c \cdot Costo_{\text{OPT}}(R) + a,$$

per tutte le possibili sequenze di input $R$.

## 5.2 *Il problema dell'affitto degli sci*

Iniziamo con un esempio semplice: il problema dell'affitto degli sci. Supponiamo di voler andare a sciare ma di non avere gli sci. Possiamo o fittarli, pagando 1 per ogni volta che andiamo a sciare, opppure comprarli, pagando $s$. Se sapessimo a priori il numero $n$ di volte che andremo a sciare potremmo facilmente decidere se conviene fittare gli sci o se conviene comprarli: se $n < s$ conviene fittarli se $n > s$ conviene comprarli (se $n = s$ le due scelte sono equivalenti). Nella versione online di questo problema non conosciamo a priori $n$, ma ogni volta dovremo decidere se fittare gli sci o comprarli (ovviamente dopo averli comprati non ci porremo più la domanda). In questo problema la richiesta $r_i$ è sempre la stessa ("procurati gli sci") mentre i possibili output dell'algoritmo sono 3: (1) fitta, (2) compra, (3) usa gli sci comprati in precedenza. Il terzo output è possibile solo dopo aver comprato gli sci. Gli output hanno costo, rispettivamente 1, $s$ e 0. Chiaramente un qualsiasi algoritmo sensato non farà altro che far fittare gli sci per le prime $k$ volte (con $k \geq 0$), comprare gli sci alla $k$-esima volta, e usare gli sci comprati dalla $(k+1)$-esima volta. Quindi il costo può essere o solo $k$, quando $k < n$, oppure $k - 1 + s$, quando $k \geq n$.

In altre parole per stabilire quale algoritmo online usare dobbiamo solo decidere un valore per $k$.

Consideriamo una sequenza di $n$ richieste. Il costo ottimale dell'algoritmo offline è $\min\{n, s\}$.

Il costo dell'algoritmo online è $n$ se $n < k$, e $k - 1 + s$ se $n \geq k$.

Consideriamo l'algoritmo online con $k = s$. Dobbiamo confrontare tale costo con quello dell'algoritmo offline ottimo. Analizziamo il fattore di competitività separando i due casi

- Caso $n < s$. In questo caso entrambi gli algoritmi hanno un costo pari a $n$. Quindi l'algoritmo online è ottimo!

- Caso $n \geq s$. In questo caso l'algoritmo offline ottimo ha costo pari a $s$, mentre l'algoritmo online ha costo pari a $2s - 1$, quindi ha un fattore di competitività $2\frac{s-1}{s}$ che tende a 2 al tendere di $s$ a $\infty$.

Questa strategia è la migliore possibile (Esercizio 1). Quindi, in pratica, il migliore algoritmo online è quello che fa fittare gli sci fino a che non raggiunge la cifra che sarebbe bastata a comprarli. A quel punto decide che era meglio comprarli e quindi li fa comprare. Anche se non ci saranno più utilizzi si è speso al massimo il doppio di quello che si sarebbe potuto spendere con il senno di poi.

## 5.3 *Problema del paging*

Un esempio classico di problema online è il problema della gestione delle pagine di memoria in un sistema operativo che supporta la memoria virtuale. Con il meccanismo della memoria virtuale si mette a disposizione dei programmi degli utenti una quantità di memoria RAM, divisa in cosiddette *pagine*, più grande di quella fisicamente disponibile. Questo è possibile in quanto non tutte le pagine di memoria devono essere fisicamente presenti in RAM in ogni singolo istante; pertanto il sistema operativo sfrutta l'hard disk per memorizzare le pagine che non sono necessarie facendo posto in RAM per quelle necessarie. Le pagine che devono essere presenti in RAM dipendono dai programmi che vengono eseguiti dall'utente e quindi non si può sapere a priori quali saranno. Tuttavia ogni qualvolta il sistema operativo è costretto a cancellare una pagina dalla RAM (che verrà salvata nell'hard disk per il suo successivo recupero) deve decidere quale pagina deve essere cancellata. Chiaramente lo *scambio* delle pagine fra RAM e hard disk costa tempo, pertanto la strategia dovrebbe minimizzare il numero di scambi delle pagine. Conoscendo a priori l'intera sequenza di pagine necessarie rende il problema più facile. Tuttavia questo è un problema online in quanto successive pagine da caricare in RAM saranno note solo durante l'esecuzione perchè possono dipendere dall'interazione con l'utilizzatore.

Assumiamo che la memoria RAM possa contenere, simultaneamente, $k$ pagine e che la memoria di massa (hard disk) possa invece contenere $N$ pagine, con $k < N$. Ogni richiesta $r_i$ è una specifica pagina che si vuole nella memoria RAM. Per soddisfare la richiesta $r_i$ l'algoritmo deve far in modo che la pagina $r_i$ sia presente in memoria. Se la pagina richiesta è già in memoria allora e non si deve far nulla. Se invece la pagina richiesta non è in RAM, allora occorre leggerla dalla memoria di massa e scriverla in una delle pagine della memoria RAM. Questa situazione viene detta in gergo *page fault*. Un page fault comporta un costo, che quantificheremo in un'unità, e richiede la scelta di una pagina che dovrà essere cancellata dalla RAM per far posto alla pagina $r_i$.

Consideriamo un esempio con $k = 4$, $N = 8$, e

$$\sigma = 1, 6, 1, 3, 4, 2, 1, 3, 2, 4, 5, 6, 8, 2, 5, 6, 4, 2.$$

Assumendo che la memoria cache sia inizialmente vuota,

$$C = \langle \cdot, \cdot, \cdot, \cdot \rangle,$$

le prime $k$ diverse pagine richieste genereranno $k$ page faults, ma non ci sarà bisogno di eliminare nessuna pagina dalla cache in quanto abbiamo $k$ posizioni libere. Nel nostro esempio le prime due richieste $1, 6$ generano due page fault e la cache sarà

$$C = \langle 1, 6, \cdot, \cdot \rangle.$$

La terza richiesta è per una pagina già presente quindi non ci sarà un page fault. Le successive due richieste per 3 e 4 generano altri due page fault che faranno riempiere la cache

$$C = \langle 1, 6, 3, 4 \rangle.$$

Denotiamo con $C_0$ tale cache. Nella Figura 5.1 sono riportati i successivi cambiamenti.

Quando la cache è piena per gestire un page fault dovremo togliere una pagina presente dalla cache per far posto alla pagina richiesta. Ad esempio la successiva richiesta per la pagina 2 genera un page fault in quanto 2 non è in $C$ e per far posto alla pagina 2 dovremo togliere una delle pagine presenti in $C$. Un algoritmo di paging deve decidere quale pagina togliere. Per questo esempio utilizziamo una regola qualsiasi, come ad esempio, scegliere la pagina nella posizione $i$-esima, partendo da $i = 1$, e incrementando $i$ di 1 (modulo $k$) ad ogni page fault. Questa regola, chiaramente non ha nessuno criterio che la giustifichi e quindi la stiamo usando semplicemente per capire cosa deve fare un algoritmo di paging. Indicheremo con una sottolineatura la posizione che sarà usata per inserire il prossimo elemento e quindi cancellare quella presente in tale posizione:

$$C_0 = \langle \underline{1}, 6, 3, 4 \rangle.$$

Dunque, la pagina 2 verrà messa nella prima posizione, il che significa che la pagina 1 viene cancellata, e quindi si avrà $C_1$. Procedendo con la sequenza di richieste troviamo una richiesta per 1, che non è più nella cache quindi genera un page fault. La pagina 1 verrà caricata e messa nella posizione 2 cancellando la pagina 6; la cache diventa $C_2$. Le successive tre richieste, per $3, 2, 4$, non generano nessun page fault. Mentre le tre richieste che troviamo dopo, per $5, 6, 8$ generano 3 page faults e il corrispondente contenuto della cache sarà $C_3, C_4$ e $C_5$. La successiva richiesta è per la pagina 2 che è stata cancellata e quindi anche questa richiesta genera un page fault e ora la cache diventerà $C_6$. Le successive richieste per $5, 6$ non generano nessun page fault, mentre la richiesta per 4 sì, e la cache diventerà $C_7$. L'ultima richiesta per 2 non genera page fault. Il costo totale è, dunque, di 4 page fault iniziali per riempire la cache e poi di 7 page fault, quindi 11 in totale.

L'obiettivo di un algoritmo di paging è quello di minimizzare tale costo totale. L'algoritmo che abbiamo usato è abbastanza "stupido". Un algoritmo offline può

$C_0 = \langle \underline{1}, 6, 3, 4 \rangle$
$\sigma_6 = 2$, page fault
$C_1 = \langle 2, \underline{6}, 3, 4 \rangle$
$\sigma_7 = 1$, page fault
$C_2 = \langle 2, 1, \underline{3}, 4 \rangle$
$\sigma_{11} = 5$, page fault
$C_3 = \langle 2, 1, 5, \underline{4} \rangle$
$\sigma_{12} = 6$, page fault
$C_4 = \langle \underline{2}, 1, 5, 6 \rangle$
$\sigma_{13} = 8$, page fault
$C_5 = \langle 8, \underline{1}, 5, 6 \rangle$
$\sigma_{14} = 2$, page fault
$C_6 = \langle 8, 2, \underline{5}, 6 \rangle$
$\sigma_{17} = 4$, page fault
$C_7 = \langle 8, 2, 4, \underline{6} \rangle$

Figura 5.1: Esempio di esecuzione

$C = \langle 1, 6, 3, 4 \rangle$.
$\sigma_6 = 2$, page fault
$C = \langle 1, \underline{6}, 3, 4 \rangle$.
$C = \langle 1, 2, 3, 4 \rangle$.
$\sigma_{11} = 5$, page fault
$C = \langle \underline{1}, 2, 3, 4 \rangle$.
$C = \langle 5, 2, 3, 4 \rangle$.
$\sigma_{12} = 6$, page fault
$C = \langle 5, 2, \underline{3}, 4 \rangle$.
$C = \langle 5, 2, 6, 4 \rangle$.
$\sigma_{13} = 8$, page fault
$C = \langle 5, \underline{2}, 6, 4 \rangle$.
$C = \langle 5, 8, 6, 4 \rangle$.
$\sigma_{17} = 4$, page fault
$C = \langle 5, \underline{8}, 2, 6 \rangle$.
$C = \langle 5, 4, 6, 2 \rangle$.

Figura 5.2: Esecuzione di LFD

operare in modo più intelligente guardando le richieste future e decidendo in base a tali richieste quale pagine eliminare: non conviene eliminare pagine che saranno richieste nell'immediato futuro. Pertanto una possibile strategia è quella che decide di eliminare la pagina che sarà richiesta più tardi. Tale strategia è denominata LFD.

- LFD: (Longest Forward Distance) ad ogni page-fault espelli la pagina la cui prossima richiesta è più lontano nel futuro.

La Figura 5.2 mostra il comportamento di LFD sulla sequenza di input usata per l'esempio precedente. Il numero totale di page faults, $4 + 6 = 10$, è diminuito. Quindi, almeno in questo caso LFD è migliore. In realtà LFD è un algoritmo ottimo.

**Teorema 5.3.1** *L'algoritmo* LFD *è un algoritmo ottimo per il problema del paging.*

Omettiamo la prova del precedente teorema. Il lettore interessato potrà trovarla, ad esempio, in S2012 [29]. L'algoritmo LFD non è un algoritmo online: ha bisogno di esaminare l'intero input per poter decidere quale pagina cancellare. Gli algoritmi online possono utilizzare solo la porzione di input nota fino al momento della decisione. I seguenti algoritmi, invece, sono online.

- LRU: (Least Recently Used) quando si verifica un page fault, selezionare come pagina da cancellare quella che è stata richiesta meno recentemente.
- FIFO: (First-In-First-Out) quando si verifica un page fault, selezionare come pagina da cancellare quella che sta da più tempo nella memoria RAM.
- LIFO: (Last-In-First-Out) quando si verifica un page fault, selezionare come pagina da cancellare quella che sta da meno tempo nella memoria RAM.

### 5.3.1 *Algoritmo* LRU

**Teorema 5.3.2** *L'algoritmo LRU è k-competitivo.*

DIMOSTRAZIONE. Proveremo che per una qualunque sequenza $\sigma = \sigma_1\sigma_2 \dots \sigma_m$ di richieste, risulta che

$$Costo_{\text{LRU}}(\sigma) \leq k \times Costo_{\text{OPT}}(\sigma).$$

Partizioniamo la sequenza di richieste $\sigma$ in fasi

$$\sigma = F_0F_1 \dots F_i \dots F_s$$

dove la fase $F_0$ è composta da una sottosequenza di richieste di $\sigma$ su cui l'algoritmo LRU ha *al più* $k$ page faults e le successive fasi $F_i$ sono composte da sottosequenze con *esattamente* $k$ page faults. La suddivisione di $\sigma$ in fasi è effettuabile esaminando la sequenza partendo dalla fine.

Se riuscissimo a dimostrare che in ogni fase l'algoritmo ottimo ha almeno un page fault avremmo dimostrato il teorema in quanto si avrebbe che

$$Costo_{\text{OPT}} \geq s + 1$$

e quindi che

$$\begin{aligned} Costo_{\text{LRU}}(\sigma) &= \text{numero di fault che LRU genera su } \sigma \\ &\leq k \times \text{numero di fasi} \\ &= k \times (s+1) \\ &\leq k \times Costo_{\text{OPT}}. \end{aligned}$$

E ciò dimostrerebbe il teorema.

Per dimostrare che l'algoritmo ottimo avrà almeno un page fault in ognuna delle fasi procediamo come segue. Per un confronto equo, assumiamo che la cache iniziale sia la stessa sia per l'algoritmo LRU che per quello ottimo off-line. Quindi la prima richiesta di $\sigma$ che provoca un page fault lo provoca sia per LRU che per quello ottimo (a meno che l'algoritmo ottimo non decida di caricare la pagina prima anche in assenza di un page fault ma questo equivale a pagare il costo di un page fault prima del page fault, quindi il discorso funzionerebbe comunque). Ovviamente ciò avviene nella fase $F_0$. Quindi abbiamo provato che nella fase $F_0$ l'algoritmo ottimo paga il costo di almeno un page fault. Adesso proveremo che per ogni fase $F_i$, $i \geq 1$, un qualunque algoritmo (e quindi anche quello ottimo) deve necessariamente avere almeno un page fault. Consideriamo la fase $F_i$:

$$F_i = \sigma_{t_i}\sigma_{t_i+1} \ldots \sigma_{t_{i+1}-1}$$

dove $t_i$ è l'indice della prima richiesta della fase $i$. Sia $P$ l'ultima pagina richiesta nella fase $F_{i-1}$. Essendo l'ultima pagina caricata nella fase precedente, $P$ è sicuramente presente nella memoria all'inizio di $F_i$.

Dimostriamo adesso che $F_i$ contiene almeno $k$ richieste a pagine diverse da $P$. Sappiamo che in $F_i$ ci sono $k$ fault per l'algoritmo LRU. Questo significa che ci sono state $k$ richieste a pagine non presenti nella memoria di LRU. Siano $A_1, A_2, \ldots, A_k$ tali pagine.

Consideriamo ora queste due possibilità:

1. non ci sono duplicati in $\{P, A_1, A_2, \ldots, A_k\}$.

   Questo è il caso più semplice in quanto $A_1, A_2, \ldots, A_k$ sono tutte diverse e sono diverse anche da $P$. Poichè le pagine $A_i$ sono pagine richieste nella fase $F_i$ abbiamo dimostrato l'asserzione.

2. ci sono duplicati in $\{P, A_1, A_2, \ldots, A_k\}$.

   Consideriamo due sottocasi:

   - $P$ è duplicato; cioè $P = A_i$ per un qualche $i$
     Poiché $P$ è la pagina più recente all'inizio della fase, l'unico modo per avere un page fault con $A_i = P$, è quello di epurare $P$ dalla cache prima della richiesta $A_i$. Ma questo significa che $P$ deve diventare la pagina più "vecchia" nella cache, e per questo servono $k-1$ richieste diverse da $P$ e poi ci deve essere un page fault che toglie $P$ e quindi serve una ulteriore richiesta per una pagina non presente nella cache e diversa da $P$. Quindi anche in questo caso abbiamo che nella fase $F_i$ ci sono $k$ richieste per pagine diverse tra loro e diverse da $P$ (in questo caso potrebbero non essere $A_1, A_2, \ldots, A_k$, ma altre pagine presenti in cache).

**Esempio:** $k = 3$, $\sigma = \underbrace{1,2,3}_{F_0},\underbrace{1,2,4,5,\dots}_{F_1}$ Per avere un fault sulla pagina $P = 3$ è necessario prima epurarla dalla cache e per fare questo deve diventare la pagina più vecchia; servono almeno $k-1$ richieste diverse da $P$, nell'esempio 1 e 2. Poi occorre una ulteriore richiesta diversa da $P$ e dalle pagine in memoria per generare il page fault che toglie $P$ dalla cache, nell'esempio 5. Le pagine $1, 2, 5$ sono $k$ pagine diverse da $P$ richieste nella fase.

- $P$ non è duplicato; quindi $P \neq Q = A_i = A_j$ per due indici $i < j$.
  In realtà questo caso è molto simile al precedente, forse anche più semplice. Infatti per avere una pagina $Q = A_i = A_j$, $i < j$, duplicata fra le pagine che generano un page fault è necessario che fra il primo page fault $A_i$ e il secondo page fault $A_j$ ci siano altri $k$ page fault; e poichè $P$ è in memoria, questi page fault devono essere tutti per pagine diverse da $P$; prima di essere epurato dalla memoria $P$ deve diventare la pagina più vecchia e non può farlo prima di $k$ richieste a pagine diverse da $P$.

  **Esempio:** $k = 3$, $\sigma = \underbrace{1,2,3}_{F_0},\underbrace{4,2,3,1,4}_{F_1}$. Per avere un doppio fault sulla pagina $Q = 4$ è necessario avere almeno $k = 3$ richieste per pagine diverse da $P = 3$, nell'esempio $1, 2, 4$.

Quindi, in ogni caso, si ha che la fase $F_i$ contiene almeno $k$ richieste a pagine distinte e diverse da $P$.

A questo punto è facile concludere in quanto se nella fase $F_i$ ci sono state richieste per $k$ pagine diverse tra loro e diverse da $P$ poichè $P$ è in memoria all'inizio della fase le $k$ richieste distinte necessariamente generano almeno un page fault. E questo è vero per un qualsiasi algoritmo, anche per quello ottimo. Pertanto il costo dell'algoritmo ottimo deve essere pari ad almeno il numero di fasi, cioè almeno $s + 1$. □

### 5.3.2 *Algoritmo* FIFO

Abbiamo dunque dimostrato che LRU è $k$-competitivo. In maniera analoga si può dimostrare che l'algoritmo FIFO è $k$-competitivo (Esercizio 5).

### 5.3.3 *Algoritmo* LIFO

Si può anche dimostrare (Esercizio 8) che l'algoritmo LIFO *non* è $c$-competitivo per nessun valore di $c$. In altre parole è sempre possibile trovare una sequenza di input per la quale l'algoritmo LIFO genera un numero di page faults più grande di $c \cdot Costo_{OPT}$, per qualunque valore di $c$.

### 5.3.4 *Ottimalità algoritmi online*

Gli algoritmi LRU e FIFO sono algoritmi online ottimi, nel senso che non esistono algoritmi con competitività migliore. Infatti si ha il seguente risultato.

**Teorema 5.3.3** *Non esistono algoritmi per il problema del paging che siano $k'$-competitivi per un qualsiasi $k' < k$.*

DIMOSTRAZIONE. Per assurdo supponiamo che esista un algoritmo ALG $k'$-competitivo con $k' < k$ e consideriamo un insieme di pagine $\{P_1, \ldots, P_k, P_{k+1}\}$. Iniziamo con il provare che l'algoritmo LFD, per una qualsiasi sequenza $\sigma = \sigma_1\sigma_2\ldots$ di pagine nell'insieme $\{P_1, \ldots, P_k, P_{k+1}\}$ ha un costo che di

$$Costo_{\text{LFD}}(\sigma) \leq \frac{|\sigma|}{k} \tag{5.1}$$

dove $|\sigma|$ è il numero di richieste della sequenza. Notiamo innanzitutto che ogni richiesta $\sigma_i$ si riferisce o a una pagina già nella memoria cache o all'unica pagina correntemente al di fuori della memoria cache. Ciò a causa dell'ipotesi che $\sigma$ è una sequenza di richieste a pagine nell'insieme $\{P_1, \ldots, P_k, P_{k+1}\}$. Supponiamo ora che a un certo istante, in corrispondenza alla richiesta $\sigma_i$, l'algoritmo LFD espella la pagina $P$. Ricordiamo che l'algoritmo LFD espelle sempre la pagina che verrà richiesta più lontanamente nel futuro, tra tutte le pagine attualmente risiedenti nella memoria cache. Quindi, se $P$ viene espulsa da LFD in conseguenza della richiesta $\sigma_i$ vi saranno, dopo la richiesta $\sigma_i$, necessariamente almeno $k-1$ successive richieste alle altre pagine diverse da $P$, che per ipotesi stanno già in memoria, e quindi non creano alcun page fault. In altri termini, per ogni $k$ richieste consecutive della sequenza $\sigma$, possiamo avere al più un page fault. Ovvero, il numero di page fault di LFD è minore o al massimo uguale a $|\sigma|/k$, e quindi la (5.1) è dimostrata.

Consideriamo ora l'algoritmo online ALG. Costruiamo la seguente sequenza di richieste $\sigma = \sigma_1 \ldots \sigma_n$. Le prime $k$ richieste $\sigma_1, \ldots, \sigma_k$ sono per $k$ diverse pagine e senza perdere in generalità assumiamo che non ci siano page faults (siamo stati così fortunati da trovare nella memoria le $k$ pagine richieste; visto che stiamo provando un lower bound questo non è un problema nel senso che se ci fossero dei page fault andrebbero ad avvalorare la tesi). La successiva richiesta $\sigma_{k+1}$ sarà per una pagina non presente nella cache. Questo genera per l'algoritmo ALG, qualunque esso sia, un page fault. Sia $P_j$ la pagina che ALG decide di espellere. La successiva richiesta $\sigma_{k+2}$ sarà per $P_j$ ed ovviamente genererà un page fault. Sia $P'_j$ la pagina che ALG decide di espellere per questo nuovo page fault. La successiva richiesta $\sigma_{k+3}$ sarà per $P'_j$. E così via. Cioè ogni nuova richiesta della sequenza sarà per la pagina appena espulsa.

Questa particolare sequenza che abbiamo costruito genera $n-k$ page faults (per le prime $k$ richieste non ci sono page-fault).

Si ha che

$$Costo_{\text{ALG}}(\sigma) = n - k.$$

Poichè abbiamo assunto per assurdo che l'algortimo ALG è $k'$-competitivo per un $k' < k$ si ha che

$$Costo_{\text{ALG}}(\sigma) \leq k' \cdot Costo_{\text{OPT}} + a$$

per una qualche costante $a$ ed una qualsiasi sequenza di input $\sigma$. D'altra parte abbiamo appena visto che $Costo_{\text{ALG}}(\sigma) = n - k$.

D'altra parte, sfruttando l'equazione (5.1) si avrebbe che[1]

$$Costo_{\text{OPT}} \leq Costo_{\text{LFD}} \leq \frac{n}{k}.$$

[1] In realtà noi sappiamo che LFD è ottimo, ma ai fini di questa dimostrazione possiamo ignorare questa conoscenza e sfruttare l'ovvia relazione $Costo_{\text{OPT}} \leq Costo_{\text{LFD}}$.

Pertanto se ALG fosse $k'$-competitivo si avrebbe che

$$\begin{aligned} n-k &= Costo_{\text{ALG}} \\ &\leq k' \cdot Costo_{\text{OPT}}(\sigma) + a \\ &\leq k' \cdot \frac{n}{k} + a. \end{aligned}$$

il che implicherebbe la disuguaglianza

$$n - k \leq k' \cdot \left(\frac{n}{k}\right) + a$$

che è impossibile per valori di $n$ sufficientemente grandi. □

## 5.4 *Aggiornamento di liste*

Consideriamo il seguente problema. Dobbiamo mantenere una lista (non ordinata) di elementi; gli elementi possono essere inseriti o cancellati e possiamo ricevere richieste per recuperare un elemento presente nella lista. Più formalmente abbiamo una sequenza di richieste $\sigma = \sigma_1, \ldots, \sigma_m$ ognuna delle quali è una delle seguenti operazioni.

1. *inserimento* di un elemento nella lista;
2. *accesso* a un elemento della lista;
3. *cancellazione* di un elemento dalla lista.

Per soddisfare la richiesta $\sigma_i$, un algoritmo online per questo problema deve prima di tutto attraversare la lista partendo dalla prima posizione fino a trovare l'elemento (se esiste). Il costo per fare ciò è proporzionale alla posizione $j$ dell'elemento nella lista (se non c'è è proporzionale alla lunghezza $n$ della lista). L'operazione di inserimento è consentita solo quando l'elemento non c'è, e l'elemento viene inserito in fondo alla lista. L'algoritmo può riorganizzare la lista in qualsiasi momento operando scambi di posizione fra elementi consecutivi[2]. Ogni scambio ha un costo pari a 1. Permetteremo anche di spostare un elemento, per il quale si è appena eseguita un'operazione di accesso, in una qualsiasi posizione più vicina alla testa della lista senza costi aggiuntivi (vedi Esercizio 9). Nel prosieguo useremo l'espressione "*scambi pagati*" per far riferimento agli scambi che fanno incorrere l'algoritmo nel costo dello scambio.

Dato un algoritmo $A$ il costo di $A(\sigma)$ su una sequenza di richieste $\sigma$ è dato dalla somma dei costi delle singole richieste: $Costo(A(\sigma)) = \sum_{i=1}^{m} Costo(\sigma_i)$, dove il $Costo(\sigma_i)$ è dato dalla posizione $j$ dell'elemento cercato più il costo degli scambi pagati operati dall'algoritmo. L'obiettivo è quello di avere un algoritmo online $A$ che minimizza tale costo $Costo(A(\sigma))$.

[2] L'esercizio 7 chiede di mostrare che questa non è una limitazione in quanto usando solo scambi fra elementi consecutivi possiamo ottenere un qualsiasi ordinamento.

### 5.4.1 *Strategia* MTF

La strategia MTF è quella più utilizzata: ogni volta che si accede a un elemento lo si sposta in testa alla lista (Move-To-Front).

**Teorema 5.4.1** *La strategia* MTF *è 2-competitiva.*

DIMOSTRAZIONE. Sia $\sigma_1, \ldots, \sigma_m$ una qualunque sequenza di input. Per provare che MTF è 2-competitivo dobbiamo provare che $Costo(\text{MTF}(\sigma)) \leq 2Costo(\text{OPT}(\sigma))$.

Per confrontare il comportamento di MTF rispetto al comportamento di OPT assumeremo che entrambi gli algoritmi partono dalla stessa lista. Quindi possiamo immaginare di eseguire parallelamente i due algoritmi e osservare le modifiche che fanno alla propria lista. Poichè gli algoritmi potrebbero comportarsi in modo diverso è possibile che durante la loro esecuzione gli elementi della lista assumano posizioni diverse in uno rispetto all'altro. Se due elementi $x$ e $y$ presenti nella lista hanno posizioni relative invertite, cioè $x$ compare prima di $y$ nella lista di MTF mentre nella lista di OPT $x$ compare dopo di $y$, diremo che $x$ e $y$ formano una *inversione*.

Indicheremo con $Costo(\text{MTF}(\sigma_i))$ il costo in cui incorre MTF nel servire la richiesta $\sigma_i$ e analogamente con $Costo(\text{OPT}(\sigma_i))$ il costo in cui incorre OPT nel servire la richiesta $\sigma_i$. Definiamo ora la funzione $\Phi(i)$ come il numero di inversioni che si hanno dopo aver servito la richiesta $\sigma_i$. Inizialmente non ci sono inversioni (gli algoritmi partono dalla stessa lista), quindi $\Phi(0) = 0$.

Proveremo che

$$\forall i, Costo(\text{MTF}(\sigma_i)) + \Phi(i) - \Phi(i-1) \leq 2Costo(\text{OPT}(\sigma_i)) - 1. \tag{5.2}$$

Prima di provare l'equazione (5.2), osserviamo che essa implica il teorema. Infatti si ha che

$$\begin{aligned} Costo(\text{MTF}(\sigma)) &= \sum_{i=1}^{m} Costo(\text{MTF}(\sigma_i)) \\ &\leq \sum_{i=1}^{m} (2Costo(\text{OPT}(\sigma_i)) - 1 - \Phi(i) + \Phi(i-1)) \\ &= 2Costo(\text{OPT}(\sigma)) - m - \Phi(m) + \Phi(0) \\ &\leq 2Costo(\text{OPT}(\sigma)) \end{aligned}$$

dove l'ultima disuguaglianza è dovuta al fatto che $\Phi(0) = 0$ e $\Phi(m) \geq 0$. Dunque la (5.2) implica il teorema. Procediamo a provare che essa è vera.

Consideriamo la richiesta $\sigma_i$ per un qualunque $i$. Essa può essere una di tre operazioni: inserimento, cancellazione o accesso. Consideriamo prima il caso in cui $\sigma_i$ è un richiesta di accesso a un elemento $k_i$ della lista. Denotiamo con $t_i$ la posizione dell'elemento $k_i$ nella lista di MTF prima che la richiesta $\sigma_i$ venga servita e analogamente, denotiamo con $s_i$ la posizione dell'elemento $k_i$ nella lista di OPT prima che la richiesta $\sigma_i$ venga servita.

Sia inoltre $P_i$ il numero di scambi pagati in cui incorre OPT per servire $\sigma_i$. Si ricordi che MTF non usa scambi pagati ma solo spostamenti senza costi. Pertanto si ha che

$$Costo(\text{MTF}(\sigma_i)) = t_i \text{ e che } Costo(\text{OPT}(\sigma_i)) = s_i + P_i.$$

Distiguiamo ora due possibili casi.

- Caso 1: $t_i > s_i$.
  Considereremo due sottocasi: $P_i = 0$ e $P_i > 0$. Consideriamo prima $P_i = 0$. Questo

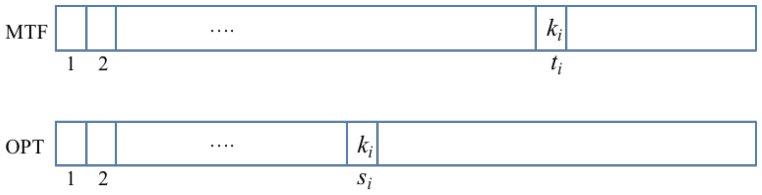



Figura 5.3: Posizioni di $k_i$ nelle due liste (caso $t_i > s_i$).

significa che OPT non effettua scambi pagati, pertanto l'unico elemento che può cambiare di posizione è $k_i$. Ci chiediamo, quante inversioni ci sono prima di servire $\sigma_i$? Prima che $k_i$ cambi posizione nella lista di MTF, poichè $t_i > s_i$, ci sono *almeno* $t_i - s_i$ inversioni in cui in elemento $x$ si trova dopo $k_i$ nella lista di OPT e prima di $k_i$ nella lista di MTF. Sia $X$ il numero di inversioni rimanenti, cioè quelle che non coinvolgono $k_i$. Abbiamo dunque che

$$\Phi(i-1) \geq t_i - s_i + X.$$

Per definizione MTF sposta $k_i$ in testa alla lista, mentre OPT potrà spostare $k_i$ in una qualunque delle posizioni $1, 2, \ldots, s_i - 1$ oppure lasciarlo nella posizione $s_i$. Pertanto dopo aver servito la richiesta $\sigma_i$ ci saranno *al massimo* $s_i - 1$ inversioni che coinvolgono l'elemento $k_i$. Il numero $X$ di inversioni che non coinvolgono $k_i$ non è cambiato in quanto l'unico elemento che è stato spostato è $k_i$ quindi la posizione relativa degli altri elementi non è cambiata. Quindi

$$\Phi(i) \leq s_i - 1 + X.$$

Pertanto abbiamo che

$$\Phi(i) - \Phi(i-1) \leq (s_i - 1 + X) - (t_i - s_i + X) = 2s_i - t_i - 1.$$

Ricordando che $Costo(\text{MTF}(\sigma_i)) = t_i$, e che $P_i = 0$ quindi $Costo(\text{OPT}(\sigma_i)) = s_i$, si ha che

$$\begin{aligned} Costo(\text{MTF}(\sigma_i)) + \Phi(i) - \Phi(i-1) &= t_i + \Phi(i) - \Phi(i-1) \\ &\leq t_i + (2s_i - t_i - 1) \\ &= 2s_i - 1 \\ &= 2Costo(\text{OPT}(\sigma_i)) - 1 \end{aligned}$$

e quindi la (5.2) è provata quando $P_i = 0$. Ora consideriamo $P_i > 0$. Ognuno degli $P_i$ scambi pagati può far aumentare la differenza $\Phi(i) - \Phi(i-1)$ di al più 1, in quanto uno scambio di due elementi consecutivi può creare una sola nuova inversione. Pertanto abbiamo che

$$\Phi(i) - \Phi(i-1) \leq 2s_i - t_i - 1 + P_i.$$

Quindi si ha che

$$\begin{aligned} Costo(\text{MTF}(\sigma_i)) + \Phi(i) - \Phi(i-1) &= t_i + \Phi(i) - \Phi(i-1) \\ &\leq t_i + (2s_i - t_i - 1) + P_i \\ &= 2s_i + P_i - 1 \\ &\leq 2s_i + 2P_i - 1 \\ &= 2Costo(\text{OPT}(\sigma_i)) - 1. \end{aligned}$$

Dunque per il caso $t_i > s_i$ la (5.2) vale sempre.

- Caso 2: $t_i \leq s_i$.
  Possiamo procedere con un ragionamento analogo. Prima di servire la richiesta $\sigma_i$ ci sono *almeno* $s_i - t_i$ inversioni che coinvolgono $k_i$ e quindi, denotando con $X$ quelle che non coinvolgono $k_i$ si ha che $\Phi(i-1) \geq s_i - t_i + X$. Analogamente al caso precedente, dopo aver servito la richiesta $\sigma_i$ si ha che MTF ha spostato $k_i$ in testa alla lista mentre OPT può aver spostato $k_i$ in una qualsiasi delle posizioni $1, 2, \ldots, s_i - 1$ e quindi ci possono ora essere *al massimo* $s_i - 1$ inversioni che coinvolgono $k_i$. Pertanto, come nel caso precedente, si ha che $\Phi(i) \leq s_i - 1 + X$. Considerando anche gli scambi pagati di OPT potremo avere $P_i$ ulteriori inversioni in più, pertanto

  $$\Phi(i) - \Phi(i-1) \leq (s_i - 1 + X) - (s_i - t_i + X) + P_i = t_i - 1 + P_i.$$

  E quindi

  $$\begin{aligned} Costo(\text{MTF}(\sigma_i)) + \Phi(i) - \Phi(i-1) &\leq t_i + t_i - 1 + P_i \\ &= 2t_i + P_i - 1 \\ &\leq 2s_i + P_i - 1 \\ &\leq 2s_i + 2P_i - 1 \\ &= 2Costo(\text{OPT}(\sigma_i)) - 1. \end{aligned}$$

  Dunque anche per il caso $t_i \leq s_i$ la (5.2) è vera.

  Per concludere la prova del teorema, occorre considerare i casi in cui $\sigma_i$ è un inserimento o una cancellazione. Consideriamo il caso in cui la richiesta è un inserimento. Detta $n$ la lunghezza della lista, si ha che

  $$Costo(\text{MTF}(\sigma_i)) = n + 1 \text{ e } Costo(\text{OPT}(\sigma_i)) = n + 1.$$

  Poichè l'inserimento di un elemento in fondo alla lista non crea nessuna nuova inversione si ha che $\Phi(i) - \Phi(i-1) = 0$, quindi se la (5.2) valeva prima di $\sigma_i$, vale anche dopo.

  Rimane da considerare il caso della cancellazione. Questo caso è molto simile a quello dell'accesso. Basta osservare che una cancellazione non crea nuove inversioni ma può solo eliminarne alcune. Quindi, analogamente a quanto fatto per l'accesso, dette $t_i$ e $s_i$ le posizioni dell'elemento da cancellare rispettivamente nelle liste di MTF e OPT, potremo procedere esattamente come nel caso dell'accesso.

□

### 5.4.2 *Limite competitività*

La strategia MTF è, in pratica, la migliore possibile, nel senso che si può provare che non esistono algoritmi online con un fattore di competitività sostanzialmente migliore di 2. Per provare questa asserzione analizzeremo il costo di un generico algoritmo online e mostreremo che ci sono casi in cui l'algortimo deve necessariamente avere un costo che è quasi il doppio del costo di OPT.

Consideriamo un generico algoritmo online $\mathcal{A}$. Sia $\sigma = \sigma_1 \dots \sigma_m$ la sequenza di richieste che richiede sempre l'ultimo elemento della lista mantenuta da $\mathcal{A}$. Quindi, qualunque sia $\mathcal{A}$, il suo costo è

$$Costo(\mathcal{A}(\sigma)) = nm$$

dove $n$ è la lunghezza della lista. Detto in altre parole, abbiamo osservato che per ogni algoritmo online esiste una particolare sequenza di richieste che causa un costo di $nm$.

Per confrontare tale costo con quello di un algoritmo ottimo considereremo il seguente algoritmo offline $\mathcal{B}$. Eseguiamo una fase preliminare in cui si analizza la sequenza $\sigma$ per calcolare le frequenze con cui gli elementi compaiono; siano $f_1 \geq f_2, \dots \geq f_n$ tali frequenze ordinate dall'elemento più frequente a quello meno frequente. Riorganizziamo la lista ordinando gli elementi in base a tali frequenze, cioè mettendo nella prima posizione l'elemento più frequente e nell'ultima quello meno frequente. Per effettuare l'ordinamento sono sufficenti al più $n(n-1)/2$ scambi, ad esempio usando BubbleSort. Dopo l'ordinamento si potranno servire le $m$ richieste, che faranno incorrere in un costo pari a $\sum_{i=1}^{n} i f_i$ e quindi si ha che

$$Costo(\mathcal{B}(\sigma)) \leq \frac{n(n+1)}{2} + \sum_{i=1}^{n} i f_i.$$

Osserviamo ora che la condizione $f_1 \geq f_2, \dots \geq f_n$, implica

$$\sum_{i=1}^{n} i f_i \leq \frac{m(n+1)}{2}. \tag{5.3}$$

La prova dell'equazione (5.3) è lasciata come esercizio (vedi Esercizio 10). Dunque si ha che

$$Costo(\text{OPT}(\sigma)) \leq Costo(\mathcal{B}(\sigma)) \leq \frac{n(n-1)}{2} + \frac{m(n+1)}{2}$$

e quindi che

$$\frac{m(n+1)}{2} \geq Costo(\text{OPT}(\sigma)) - \frac{n(n-1)}{2}$$

e pertanto

$$\begin{aligned} Costo(\mathcal{A}(\sigma)) &= mn \\ &= \frac{2n}{(n+1)} \cdot \frac{m(n+1)}{2} \\ &\geq \frac{2n}{(n+1)} \left[ Costo(\text{OPT}(\sigma)) - \frac{n(n-1)}{2} \right] \\ &= \left(2 - \frac{2}{n+1}\right) Costo(\text{OPT}(\sigma)) - \frac{n}{n+1} n(n-1) \\ &\geq \left(2 - \frac{2}{n+1}\right) Costo(\text{OPT}(\sigma)) - n(n-1) \end{aligned}$$

Osserviamo ora che il termine $\frac{2}{n+1}$ può essere reso piccolo a piacere considerando $n$ grandi e che scegliendo $m$ molto più grande di $n$, $m$ diventa il fattore predominante nel costo degli algoritmi. Dunque per un $n$ grande e un $m$ molto più grande si ha che il costo di $\mathcal{A}$, cioè di un qualsiasi algoritmo online, deve essere più grande del (o più precisamente, di quasi il) doppio del costo di un algoritmo ottimo.

## 5.5 *Problema load balancing*

Il problema del load balancing che abbiamo visto nella sezione 3.3 può essere anche un problema online, quando l'input viene fornito un compito alla volta e bisogna assegnare il compito a una delle macchine prima di vedere il compito successivo. L'algoritmo ApproxLoadInOrder è in realtà un algoritmo online! Il fattore di approssimazione è l'equivalente del fattore di competitività, quindi abbiamo già visto un algoritmo online 2-competitivo per questo problema.

## 5.6 *Problema Bin Packing*

La sequenza di richieste $a_1, a_2, \ldots, a_m$ rappresenta una sequenza di oggetti di peso $a_i$. Gli oggetti devono essere inseriti in dei contenitori $B_1, B_2, \ldots, B_b$ con $b \geq m$; ogni contenitore $B_i$ ha capacità $c$, quindi la somma dei pesi degli oggetti assegnati a $B_i$ non può superare $c$. Gli oggetti vanno assegnati in maniera online e si vuole minimizzare il numero totale di contenitori. Assumeremo che ogni singolo oggetto possa essere messo da solo in un qualsiasi conenitore (cioè che $\max_i a_i \leq c$). Infine, quando si decide di usare un nuovo contenitore, occorre "chiudere" il precedente, il che significa che non potremo più inserire altri oggetti in quel contenitore; questo vincolo, ad esempio, codifica il fatto che i contenitori debbano essere spediti e quindi "chiuderli" equivale a spedirli.

Come per il problema del load balancing, anche per questo problema l'approccio greedy fornisce un algoritmo sufficientemente buono. L'algoritmo è il seguente.

**Algorithm 20:** NextFit

```
j ← 1
for i ← 1 to m do
    if a_i non può essere inserito in B_j then
        j ← j + 1
    Inserisci a_i in B_j
```

Questo algoritmo fornisce un fattore di competitività pari a 2. Ossserviamo che l'algoritmo ottimo non può usare meno di $s = \frac{\sum_i^m a_i}{c}$ contenitori: questo caso limite si ha quando tutti i contenitori usati sono completamente pieni. L'algoritmo NextFit usa al massimo $s$ contenitori. Infatti per ogni coppia di contenitori consecutivi, si ha che il carico totale dei due contenitori è superiore a $c$, in quanto si passa al secondo contenitore proprio perchè il carico del primo contenitore più l'oggetto considerato fa diventare il peso totale maggiore di $c$. Dunque l'algoritmo NextFit non userà mai più di $2s$ contenitori.

## 5.7 *Analisi probabilistica*

Gli algoritmi online che abbiamo visto in precedenza sono stati valutati in base alla loro competitività, cioè confrontandoli con i corrispondenti algoritmi offline ottimi. Questo tipo di analisi considera il caso pessimo ed è quella più utilizzata. Tuttavia ci sono dei casi nei quali vogliamo valutare l'algoritmo con un'analisi probabilistica facendo delle assunzioni sulla distribuzione dell'input. Questo tipo di analisi fornisce una misura del comportamento dell'algoritmo nel caso medio. L'analisi può essere difficile da fare in generale, ma se possiamo fare delle forti assunzioni sulla distribuzione dell'input, come ad asempio assumere che tutti i possibili input hanno la stessa probabilità di verificarsi, allora l'analisi può essere sufficientemente facile. Inoltre ci possono essere dei problemi per i quali nel caso pessimo, che è quello considerato dall'analisi competitiva, l'algoritmo online è estremamente poco competitivo mentre potrebbe comportarsi bene in media; in questi casi può avere più senso fornire un'analisi probabilistica. Per esemplificare l'analisi probabilistica riprendiamo il problema del massimo usato come esempio nell'introduzione al capitolo.

Potremmo formularlo nella seguente versione alternativa. Supponiamo di dover vendere una casa e supponiamo che $n$ persone hanno contattatto l'agenzia di vendita per un appuntamento per presentarci un'offerta di acquisto. Siamo vincolati ad incontrare i potenziali acquirenti uno per volta e dopo avere ricevuto l'offerta dobbiamo decidere se declinarla e quindi passare alla prossima oppure accettarla e quindi non considerare più le prossime offerte (che ancora non conosciamo). Ovviamente il nostro obiettivo è quello di vendere al maggior offerente! La strategia online deve selezionare un elemento $x_j$, sulla base della conoscenza delle prime $j$ offerte $x_1, x_2, \ldots, x_j$ e senza conoscere $x_{j+1}, \ldots, x_n$. Abbiamo già osservato che qualunque strategia online potrebbe non portare all'individuazione del massimo, che può essere determinato solo nel caso offline. Per facilitare l'analisi assumeremo che le offerte siano ordinate in modo casuale,

cioè che i possibili ordinamenti delle offerte hanno tutti la stessa probabilità di essere la sequenza delle offerte. Questa è un'assunzione abbastanza ragionevole se le offerte vengono fatte in modo indipendente.

Consideriamo ora i seguenti algoritmi, parametrizzati dall'indice $r$, $0 \leq r \leq n$: rifiutiamo incondizionatamente le prime $r-1$ offerte, ma ne calcoliamo il massimo $M_{r-1}$; accettiamo la prima delle successive offerte che è maggiore o uguale a $M_{r-1}$ e se tutte le successive offerte sono minori di $M_{r-1}$ accettiamo l'ultima offerta.

Facciamo un esempio, supponiamo che l'insieme delle offerte sia $\{10, 12, 18, 20\}$. Ci sono 24 possibili sequenze di questi 4 elementi:

| # | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 10 | 12 | 18 | 20 | ($M_0 = 0$) 10 | ($M_1 = 10$) 12 | ($M_2 = 12$) 18 | ($M_3 = 18$) 20 |
| 2 | 10 | 12 | 20 | 18 | ($M_0 = 0$) 10 | ($M_1 = 10$) 12 | ($M_2 = 12$) 20 | ($M_3 = 20$) 18 |
| 3 | 10 | 18 | 12 | 20 | ($M_0 = 0$) 10 | ($M_1 = 10$) 18 | ($M_2 = 18$) 20 | ($M_3 = 18$) 20 |
| 4 | 10 | 18 | 20 | 12 | ($M_0 = 0$) 10 | ($M_1 = 10$) 18 | ($M_2 = 18$) 20 | ($M_3 = 20$) 12 |
| 5 | 10 | 20 | 12 | 18 | ($M_0 = 0$) 10 | ($M_1 = 10$) 20 | ($M_2 = 20$) 18 | ($M_3 = 20$) 18 |
| 6 | 10 | 20 | 18 | 12 | ($M_0 = 0$) 10 | ($M_1 = 10$) 20 | ($M_2 = 20$) 12 | ($M_3 = 20$) 12 |
| 7 | 12 | 10 | 18 | 20 | ($M_0 = 0$) 12 | ($M_1 = 12$) 18 | ($M_2 = 12$) 18 | ($M_3 = 18$) 20 |
| 8 | 12 | 10 | 20 | 18 | ($M_0 = 0$) 12 | ($M_1 = 12$) 20 | ($M_2 = 12$) 20 | ($M_3 = 20$) 18 |
| 9 | 12 | 18 | 10 | 20 | ($M_0 = 0$) 12 | ($M_1 = 12$) 18 | ($M_2 = 18$) 20 | ($M_3 = 18$) 20 |
| 10 | 12 | 18 | 20 | 10 | ($M_0 = 0$) 12 | ($M_1 = 12$) 18 | ($M_2 = 18$) 20 | ($M_3 = 20$) 10 |
| 11 | 12 | 20 | 10 | 18 | ($M_0 = 0$) 12 | ($M_1 = 12$) 20 | ($M_2 = 20$) 18 | ($M_3 = 20$) 18 |
| 12 | 12 | 20 | 18 | 10 | ($M_0 = 0$) 12 | ($M_1 = 12$) 20 | ($M_2 = 20$) 10 | ($M_3 = 20$) 10 |
| 13 | 18 | 10 | 12 | 20 | ($M_0 = 0$) 18 | ($M_1 = 18$) 20 | ($M_2 = 18$) 20 | ($M_3 = 18$) 20 |
| 14 | 18 | 10 | 20 | 12 | ($M_0 = 0$) 18 | ($M_1 = 18$) 20 | ($M_2 = 18$) 20 | ($M_3 = 20$) 12 |
| 15 | 18 | 12 | 10 | 20 | ($M_0 = 0$) 18 | ($M_1 = 18$) 20 | ($M_2 = 18$) 20 | ($M_3 = 18$) 20 |
| 16 | 18 | 12 | 20 | 10 | ($M_0 = 0$) 18 | ($M_1 = 18$) 20 | ($M_2 = 18$) 20 | ($M_3 = 20$) 10 |
| 17 | 18 | 20 | 10 | 12 | ($M_0 = 0$) 18 | ($M_1 = 18$) 20 | ($M_2 = 20$) 12 | ($M_3 = 20$) 12 |
| 18 | 18 | 20 | 12 | 10 | ($M_0 = 0$) 18 | ($M_1 = 18$) 20 | ($M_2 = 20$) 10 | ($M_3 = 20$) 10 |
| 19 | 20 | 10 | 12 | 18 | ($M_0 = 0$) 20 | ($M_1 = 20$) 18 | ($M_2 = 20$) 18 | ($M_3 = 20$) 18 |
| 20 | 20 | 10 | 18 | 12 | ($M_0 = 0$) 20 | ($M_1 = 20$) 12 | ($M_2 = 20$) 12 | ($M_3 = 20$) 12 |
| 21 | 20 | 12 | 10 | 18 | ($M_0 = 0$) 20 | ($M_1 = 20$) 18 | ($M_2 = 20$) 18 | ($M_3 = 20$) 18 |
| 22 | 20 | 12 | 18 | 10 | ($M_0 = 0$) 20 | ($M_1 = 20$) 10 | ($M_2 = 20$) 10 | ($M_3 = 20$) 10 |
| 23 | 20 | 18 | 10 | 12 | ($M_0 = 0$) 20 | ($M_1 = 20$) 12 | ($M_2 = 20$) 12 | ($M_3 = 20$) 12 |
| 24 | 20 | 18 | 12 | 10 | ($M_0 = 0$) 20 | ($M_1 = 20$) 10 | ($M_2 = 20$) 10 | ($M_3 = 20$) 10 |

Poichè ci sono 4 elementi, i possibili valori di $r$ sono 4, cioè ci sono 4 possibili algoritmi online $A_r$, ognuno dei quali seleziona un elemento dall'$r$-esimo all'ultimo in base alla strategia descritta. Consideriamo ad esempio l'input $(18, 20, 10, 12)$. L'algoritmo $A_1$, per il quale si ha che $M_0 = 0$, seleziona il primo elemento 18 in quanto questo valore è maggiore di $M_0$. L'algoritmo $A_2$, per il quale si ha che $M_1 = 18$, seleziona il secondo elemento 20 in quanto questo valore è maggiore di $M_1$. L'algoritmo $A_3$, per il quale si ha che $M_2 = 20$, seleziona l'ultimo elemento 12 in quanto i valori del terzo e quarto elemento sono più piccoli di $M_2$. Analogamente l'algoritmo $A_4$ seleziona anch'esso l'ultimo elemento.

Analizzando gli output forniti dai 4 algoritmi, si ha che $A_1$ e $A_4$ forniscono la soluzione ottima, cioè il massimo, in 6 casi su 24, quindi forniscono la soluzione ottima con probabilità 0.25, mentre $A_2$ ed $A_3$ in 11 casi su 24, quindi con una probabilità di circa 0.458. Per cui, per il caso di $n = 4$, l'algoritmo migliore è $A_2$ (o anche $A_3$). La seguente tabella riporta le probabilità di ottenere l'ottimo dei miglior algoritmi per i $n \leq 9$.

| $n$ | $r$ | $p$ |
|---|---|---|
| 3 | 2 | 0.5 |
| 4 | 2 | $\simeq 0.458$ |
| 5 | 3 | $\simeq 0.433$ |
| 6 | 3 | $\simeq 0.428$ |
| 7 | 3 | $\simeq 0.414$ |
| 8 | 4 | $\simeq 0.410$ |
| 9 | 4 | $\simeq 0.406$ |

Procedendo più analiticamente, possiamo calcolare la probabilità $P_r$ che l'algoritmo $A_r$, per un dato valore di $r$ calcoli il massimo:

$$\begin{aligned}
P_r &= \sum_{i=1}^{n} Pr[A_r \text{ seleziona l'offerta } i \wedge \text{l'offerta } i \text{ è il massimo}] \\
&= \sum_{i=r}^{n} Pr[A_r \text{ seleziona l'offerta } i \wedge \text{l'offerta } i \text{ è il massimo}] \\
&\quad \text{(perchè per } i < r, A_r \text{ non può selezionare il massimo)} \\
&= \sum_{i=r}^{n} Pr[A_r \text{ seleziona l'offerta } i | \text{l'offerta } i \text{ è il massimo}] \cdot Pr[\text{l'offerta } i \text{ è il massimo}] \\
&= \frac{1}{n} \sum_{i=r}^{n} Pr[A_r \text{ seleziona l'offerta } i | \text{l'offerta } i \text{ è il massimo}] \\
&= \frac{1}{n} \sum_{i=r}^{n} \frac{r-1}{i-1} \\
&= \frac{r-1}{n} \sum_{i=r}^{n} \frac{1}{i-1}.
\end{aligned}$$

Si noti che $Pr[A_r \text{ seleziona l'offerta } i \mid \text{l'offerta } i \text{ è il massimo}]=\frac{r-1}{i-1}$ perché, dato che l'offerta $i$ è il massimo, essa verrà selezionata se e solo se non ne viene selezionata una prima, cioè se e solo se il massimo fra le prime $i-1$ offerte si trova tra le prime $r-1$ offerte. Più formalmente se $M_{i-1}$, cioè il massimo fra i primi $i-1$ elementi, che è anche il valore più grande dopo il massimo (che per ipotesi si trova nella posizione $i$), si trova nelle prime $r-1$ posizioni, allora l'algoritmo selezionerà il massimo in quanto i valori nelle posizioni $r, r+1, \ldots, i-1$ saranno tutti più piccoli di $M_{r-1} = M_{i-1}$. Se invece $M_{i-1}$ si trova dopo la posizione $r-1$, allora l'algoritmo non selezionerà il massimo in quanto esiste almeno un valore ($M_{i-1}$) prima di arrivare al massimo, più grande di $M_{r-1}$. Poichè $M_{i-1}$ può trovarsi in una qualsiasi delle prime $i-1$ posizioni, ci sono $r-1$ casi favorevoli su $i-1$ casi totali, come mostrato nella figura 5.4.

Facciamo un esempio con $r = 5$, $i = 9$ per chiarire meglio. Consideriamo la sequenza

$$2, 15, 7, 8, 6, 4, 12, 13, 18, 10.$$

In questo caso il massimo, 18, si trova nella posizione $i = 9$, $M_{i-1} = M_{r-1} = 15$. Pertanto $A_r$ scarterà $6, 4, 12$ e 13 perchè tutti minori di 15 e selezionerà il 18 che è maggiore di 15. Questo è un caso favorevole in cui l'algoritmo sceglie il massimo.

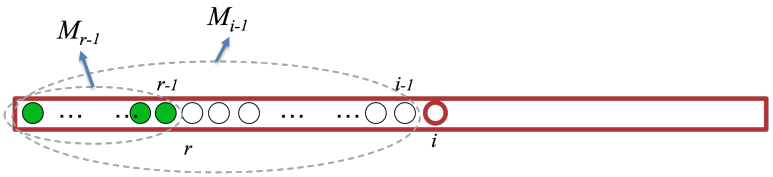



Figura 5.4: Se $M_{i-1}$ si trova nelle prime $r-1$ posizioni, $A_r$ seleziona $x_i$

Consideriamo adesso la sequenza

$$2, 12, 7, 8, 6, 4, 15, 13, 18, 10.$$

In questo caso $M_{i-1} = 15$, e $M_{r-1} = 12$. Pertanto $A_r$ scarterà 6 e 4 perchè tutti minori di 12 ma selezionerà il 15 che è maggiore di 12. Questo è un caso sfavorevole in cui l'algoritmo non sceglie il massimo.

La funzione $\frac{r-1}{n} \sum_{i=r}^{n} \frac{1}{i-1}$ è massimizzata per $r = n/e$.

## 5.8 Note bibliografiche

Il meteriale presentato in questo capitolo è tratto da varie fonti.

## 5.9 Esercizi

1. Consideriamo il problema dell'affitto degli sci per il quale abbiamo analizzato l'algoritmo online con $k = s$. Mostrare che le strategie con $k > s$ e $k < s$ portano ad algoritmi online peggiori rispetto a quello per $k = s$.

2. Consideriamo il seguente problema: abbiamo investito una somma di denaro comprando $N$ azioni di una società quotata in borsa. Ora dobbiamo rivenderle e, semplificando molto ciò che accade in realtà, assumiamo che ogni giorno il valore delle azioni cambi, quindi abbiamo una sequenza $v_1, v_2, \ldots, v_n$ di valori. Per ogni giorno $i = 1, 2, \ldots, n$ dobbiamo decidere quante delle azioni rimanenti vendere. Fornire e analizzare un algoritmo online per tale problema.

3. Nella prova della $k$-competitività dell'algoritmo LRU abbiamo dimostrato che una generica fase $F_i, i > 0$, quindi dalla seconda in poi, è necessario avere almeno $k$ richieste a $k$ pagine $\{A_1, A_2, \ldots, A_k\}$ non presenti nella memoria. La prova richiedeva di dimostrare che ci sono sempre $k$ richieste a pagine diverse da $P$, dove $P$ è l'ultima pagina chiesta nella fase precedente $F_{i-1}$. Il caso in cui $\{P, A_1, A_2, \ldots, A_k\}$ non contiene duplicati è semplice. Il caso in cui invece $\{P, A_1, A_2, \ldots, A_k\}$ contiene duplicati è diviso in due sottocasi:

   (a) $P$ è duplicato, cioè $P = A_i$ per un qualche $i$.

   (b) $P$ non è duplicato, quindi $P \neq Q = A_i = A_j$, per $i \neq j$.

   Si faccia un esempio di queste due situazioni. Si consideri $k = 4$.

4. Si consideri il seguente algoritmo di paging FWF (Flush When Full): quando c'è un page fault, se c'è un posto vuoto nella cache lo si utilizza, altrimenti si svuota la cache generando $k$ posti liberi. Si commenti la strategia FWF e si fornisca un'analisi dell'algoritmo.

5. Provare che la strategia FIFO è $k$-competitiva.

6. L'algoritmo FIFO soffre della seguente anomalia: esistono delle sequenze di input per le quali il numero di page fault con una cache più grande è maggiore di quello che si avrebbe con una cache più piccola. Fornire una sequenza di input in cui tale anomalia si verifica passando da una cache di grandezza $k = 3$ and una cache di grandezza $k = 4$.

7. Mostrare che data una lista di elementi, è possibile ottenere un qualsiasi ordinamento effetuando esclusivamente scambi di elementi in posizioni adiacenti.

8. Provare che la strategia LIFO non è $c$-competitiva per nessun valore di $c$.

9. Per gli algoritmi di gestione di una lista online abbiamo considerato delle operazioni senza costo: quando si accede a un elemento possiamo spostarlo in una qualsiasi posizione più vicina alla testa della lista. Si dia una giustificazione. Perchè non permettiamo di spostarlo verso la coda della lista?

10. Siano $f_1 \geq f_2 \geq \ldots \geq f_n$ delle frequenze di $n$ elementi tali che $\sum_{i=1}^{n} f_i = m$. Per semplicità assumiamo che $m$ sia un multiplo di $n$. Provare che $\sum_{i=1}^{n} if_i \leq \frac{m(n+1)}{2}$.

11. Si consideri il problema del bin packing in una versione che non richiede di "chiudere" un contenitore prima di usarne un altro. Si dia un algoritmo online per questa versione che si comporti meglio di NEXTFIT.

# 6
# *Algoritmi Distribuiti*

In questo capitolo ci occuperemo di algoritmi distribuiti in cui abbiamo varie entità che cooperano per la risoluzione di un problema. Come per gli altri capitoli, la trattazione è da intendersi solo introduttiva in quanto anche in questo caso l'argomento può essere l'oggetto di interi corsi. Pertanto focalizzeremo l'attenzione su un singolo problema distribuito che per vari motivi è quello più rappresentativo: il problema del *consenso*.

## *6.1 Introduzione*

Gli algoritmi che abbiamo studiato finora sono algoritmi sequenziali, cioè algoritmi che assumono che ci sia una solo unità logica centrale (CPU) che esegue tutte le operazioni. Questa è la situazione tipica che si ha quando il programma viene implementato per un computer con una sola CPU. Il progresso della tecnologia ha, da un lato, portato alla possibilità di usare più di una CPU su un singolo computer, dall'altro, alla possibilità di interconnettere fra di loro tramite una rete molti computer in modo tale da farli comunicare. Queste due soluzioni tecnologiche portano a un approccio algoritmico sostanzialmente diverso da quello usato per gli algoritmi sequenziali. Ovviamente l'obiettivo non cambia: sviluppare algoritmi efficienti per risolvere i problemi. Tuttavia la presenza di più unità logiche (siano esse le CPU di un singolo computer, che comunicano tramite memoria condivisa, siano esse vari nodi di una rete che comunicano tramite scambio di messaggi), il dover gestire la comunicazione fra di esse e la possibilità che si verifichino dei guasti, rendono i problemi più difficili. Il vantaggio è quello di sfruttare la *parallelizzazione* della computazione: in ogni singolo istante ci sono più unità che con la propria capacità computazionale contribuiscono a risolvere il problema. In alcuni casi vogliamo sfruttare la parallelizzazione semplicemente per avere degli algoritmi più veloci. In altri casi la parallelizzazione o più precisamente la *distribuzione* del problema su più nodi, è un fatto intrinseco del problema. Si pensi ad esempio al problema della ricerca dei percorsi migliori per far comunicare i computer connessi ad Internet: in quel caso le informazioni di input, e anche l'output, sono distribuite fra i router della rete che devono comunicare fra di loro, scambiarsi tali informazioni e ognuno di essi deve calcolare un output locale.

Storicamente si è usato l'aggettivo *parallelo* per indicare un algoritmo progettato per un computer con più processori e l'aggettivo *distribuito* per indicare un algoritmo

progettato per nodi di una rete. La differenza sostanziale è nella tipologia di comunicazione. Gli algoritmi paralleli sfruttano la memoria condivisa fra i processori per far comunicare i vari nodi. Gli algoritmi distribuiti non hanno una tale possibilità e devono necessariamente usare la spedizione di messaggi per far comunicare i nodi del sistema. Sebbene le due tipologie di comunicazione abbiano caratteristiche diverse (ad es., il passaggio di informazioni tramite memoria condivisa è immediato, mentre quello tramite messaggi comporta una latenza che dipende dalla rete), di cui si deve tenere conto nella progettazione degli algoritmi, concettualmente svolgono la stessa funzione: permettono la comunicazione. In questa parte del corso parleremo di algoritmi ditribuiti e li classificheremo in base a varie caratteristiche, fra queste la tipologia di comunicazione che include la memoria condivisa, quindi di fatto la nostra definizione di algoritmo ditribuito includerà gli algoritmi paralleli. È d'obbligo notare che la nostra trattazione sarà necessariamente sommaria: sia gli algoritmi distribuiti in senso stretto che quelli paralleli sono l'argomento di interi corsi di studio, anche di carattere avanzato. Qui ci limiteremo a una introduzione a tali argomenti.

### *6.1.1 Definizione di sistema/algoritmo distribuito*

Che gli algoritmi distribuiti siano più ostici di quelli sequenziali lo si capisce immediatamente dalla definizione. Di fatto non esiste una singola definizione universalmente utilizzata, come per gli algoritmi sequenziali per i quali, a meno di differenze classificabili come sottigliezze linguistiche, la definizione è unica. Anche per le "variazioni sul tema" studiate nei capitoli precedenti, la definizione è chiara e precisa. Non vale lo stesso per il caso degli algoritmi distribuiti. A titolo di esempio riportiamo di seguito le definizioni utilizzate in alcuni libri di testo:

- (Lynch [26]) "Gli algoritmi distribuiti sono algoritmi progettati per funzionare su molti processori interconessi fra di loro. Parti di un algoritmo distribuito operano simultaneamente e indipendentemente e ognuna ha a disposizione informazioni limitate. Gli algoritmi devono funzionare correttamente anche se i singoli processori e i canali di comunicazione operano a velocità diverse e sono soggetti a guasti."
- (Attiya-Welch [3]) "Un sistema distribuito è un insieme di unità di computazione che comunicano fra di loro. Questa definizione generale include un ampio insieme di moderni sistemi di computazione, dai chip VLSI a Internet, passando per multiprocessori con memoria condivisa e cluster di workstations."
- (Guerraoui-Rodrigues [17]) "Computare in modo distribuito ha a che fare on la progettazione di algoritmi per un insieme di processi con l'obiettivo di cooperare. Oltre all'esecuzione simultanea, alcuni dei processi di un sistema distribuito possono fermarsi, ad esempio perché si rompono oppure perché vengono disconnessi, mentre altri rimangono attivi e continuano ad operare."
- (Tel [31]) "Per sistema distribuito intendiamo tutte le applicazioni dove vari computer o processori cooperano in qualche modo. Questa definizione include sia reti locali che reti estese, ma anche computer multiprocessore in cui ogni processore ha la sua unità di controllo ed un suo sistema di processi che cooperano."

- (Birman [5]) "Un sistema di computazione distribuita è un insieme di programmi, eseguiti su uno o più computer che coordinano le proprie azioni scambiando messaggi. Una rete di computer è una collezione di computer interconnessi da hardware che supporta lo scambio di messaggi."
- (Tanenbaum [30]) "Un sistema distributo è una collezione di computer indipendenti che appare all'utente come un sistema singolo."

Senza la pretesa, ovviamente, di avere elencato tutte le definizioni utilizzate (esistono molti altri libri su sistemi e algoritmi distribuiti e paralleli), quelle citate fanno rendere conto della difficoltà: ci sono molti aspetti da tenere in considerazione. Pertanto al posto di fornire una nuova definizione, ci limitiano ad elencare gli aspetti importanti che si evincono dalla definizioni citate:

- molte unità di computazione, che chiameremo in modo generico *nodi*, *processi* o *processori*;
- esecuzione simultanea e indipendente delle istruzioni (programma; ogni singola unità può avere il suo programma, anche se nella maggior parte dei casi il programma è lo stesso per ogni unità);
- informazioni limitate disponibili ad ogni unità di computazione;
- diversi modi di comunicare (memoria condivisa, messaggi);
- diverse velocità, sia per l'esecuzione dei programmi sia per la comunicazione;
- possibilità che ci siano dei guasti, sia per le unità di computazione sia per i canali di comunicazione.

Chiudendo il discorso sulla "definizione" di sistema distribuito, prima di proseguire, citiamo una definizione scherzosa ma molto veritiera dovuta a Leslie Lamport che l'ha utilizzata in un email nel 1987 [21]:

> *Un sistema distribuito è un sistema nel quale il guasto di un computer, di cui non sapevi nemmeno dell'esistenza, rende il tuo computer inutilizzabile.*

I sistemi/algoritmi distribuiti vengono solitamente classificati in base alle seguenti caratteristiche:

- tipo di comunicazione: scambio di messaggi e memoria condivisa;
- sincronia dei processori: sistemi sincroni e sistemi asincroni (e sistemi parzialmente sincroni)
- vari tipi di guasti: crash o comportamento arbitrario dei processori, perdita, duplicazione e riordino dei messaggi spediti.

Focalizzeremo l'attenzione sulla comunicazione con scambio di messaggi. Di fatto tutto ciò che diremo può essere applicato anche al caso della memoria condivisa in quanto avendo a disposizione una memoria condivisa è facile "simulare" la spedizione

e la ricezione di un messaggio (senza nemmeno doversi preoccupare di eventuali errori di comunicazione). Dunque la memoria condivisa è più potente dello scambio dei messaggi. Utilizzando la terminologia introdotta poc'anzi possiamo dire che un algoritmo distribuito può essere implementato anche su un sistema parallelo, mentre un algoritmo parallelo potrebbe non essere implementabile in un sistema distribuito. Il Capitolo 17 di L96 [26] discute della relazione fra questi due modelli. Senza scendere ulteriormente in dettagli, nel prosieguo considereremo algoritmi che comunicano tramite scambio di messaggi.

Inizieremo con il descrivere algoritmi per sistemi sincroni e poi tratteremo i sistemi asincroni. La nostra trattazione sarà basata su un singolo problema, il problema del *consenso*, e descriveremo sia algortimi che risultati di impossibilità. Informalmente, il problema del consenso consiste nel far prendere a tutti i processori la stessa decisione (definiremo formalmente il problema nel prosieguo) ed è di fondamentale importanza in quanto può essere considerato come la base per la coordinazione necessaria in un sistema distribuito.

### 6.1.2 *Formalismo e misure per l'analisi*

Analizzare un algoritmo distribuito può essere molto subdolo. All'apparenza le asserzioni che si fanno sui sistemi distributi sono semplici, tuttavia i problemi possono nascondere delle difficoltà che li rendono complicati. Per poter analizzare gli algoritmi e provare dei risultati è necessario fornire un formalismo che permetta un ragionametno rigoroso. Più preciso è il formalismo più rigoroso sarà il ragionamento. Tuttavia un formalismo troppo rigoroso porta a ragionamenti molto più dettagliati che possono essere fonte di errori se non li si affronta con adeguata attenzione. Quindi, come per la definizione di sistema distribuito, esistono varie scuole di pensiero: c'è chi opta per un formalismo molto rigoroso, come ad esempio l'IOA (automi di Input/Output) di Lynch [26], e chi per formalismi meno pesanti, come ad esempio uno pseducodice. Entrambi gli approcci hanno vantaggi e svantaggi e probabilmente la scelta migliore la si può fare in funzione del problema da affrontare. Per i nostri scopi, e anche per il tempo a disposizione, opteremo per una descrizione informale dei problemi e degli algoritmi tramite pseudocodice, in linea con quanto visto finora nel corso.

La principale differenza nella descrizione di un algoritmo distribuito rispetto a un algoritmo sequenziale è dovuta al fatto che un algoritmo distributo, per un insieme di $n$ processori, è di fatto un insieme di $n$ algoritmi, uno per ogni processore. Nella maggior parte dei casi gli $n$ algoritmi sono copie dello stesso algoritmo. L'algoritmo può avere comportamenti diversi da processore a processore sfruttando delle informazioni locali, come ad esempio un identificativo del processore. Ogni processore del sistema $P = \{1, 2, \ldots, n\}$ sarà specificato tramite un indice $i \in P$, e quando faremo riferimento a un qualunque aspetto relativo al processore $i$, lo specificheremo usando $i$ come pedice. Ad esempio se vogliamo specificare che una variabile, ad esempio *status*, appartiene al processore $i$, scriveremo $status_i$.

Quando ragioneremo su un algoritmo distribuito, vorremo tipicamente provare delle proprietà dette di

- *Safety:* non succedono mai cose cattive. Questo corrisponde alla correttezza di un algoritmo sequenziale. Ad esempio, nel problema del consenso una proprietà di safety è che due processori non facciano scelte diverse.
- *Liveness:* che prima o poi qualcosa (di buono) accade. Questo corrisponde alla proprietà di terminazione degli algoritmi sequenziali. Ad esempio, nel problema del consenso, si richiede che prima o poi i processori facciano una scelta.

Per valutare la bontà di un algoritmo saremo interessati fondamentalmente a due misure: la complessità di tempo e la complessità di comunicazione. La seconda è più facile da definire in quanto basterà contare il numero di messaggi, o se necessario il numero di bit, spediti. Per quanto riguarda il tempo può essere più difficile definire una misura in quanto essa dipende dal tipo di sincronia presente nel sistema. Per sistemi sincroni sarà il numero di *round* utilizzati dall'algoritmo (descriveremo fra poco i sistemi sincroni). Per i sistemi asincroni è molto più complicato in quanto ogni componente del sistema misura il tempo in modo diverso, per cui è necessario utilizzare un'osservazione esterna del tempo.

### *6.1.3 Guasti*

Un altro aspetto importante è la resilienza ai guasti. Questo è un aspetto che è totalmente assente per un algoritmo sequenziale: un algoritmo sequenziale non è resiliente ai guasti in quanto l'esecuzione su un computer di un programma non prevede "guasti". Certamente, anche per un singolo computer ci possono essere dei guasti, ed è per questo, ad esempio, che facciamo delle copie di backup dei file, ma non sono contemplati guasti che riguardano l'esecuzione di un programma. In un sistema distribuito, invece, un guasto è un naturale evento che può verificarsi durante l'esecuzione di un programma distribuito. Se si assume che non ci siano guasti, progettare algoritmi distribuiti è relativamente semplice. La possibilità di avere dei guasti rende molto più complesse sia la progettazione che l'analisi degli algoritmi.

### *6.1.4 Sincronia*

**Sistemi sincroni.** In un sistema sincrono i processori eseguono i programmi in perfetta sincronia e anche i canali di comunicazione operano in sincronia con i processori. Pertanto l'esecuzione di un algoritmo distribuito in un sistema sincrono procede per iterazioni, in gergo chiamate *round*. In ogni round ogni singolo processore esegue un numero fissato di istruzioni, può spedire un messaggio e tutti i messaggi spediti vengono ricevuti dai destinatari che potranno utilizzarli nel round successivo. L'esecuzione di un algoritmo in un sistema sincrono può essere dunque descritta da una sequenza

$$C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots$$

dove $C_i$ è una *configurazione*, cioè l'insieme degli stati dei singoli processori dopo $i$ round, $M_i$ è l'insieme di di messaggi spediti durante il round $i$, mentre $N_i$ è l'insieme di messaggi ricevuti nel round $i$. I due insiemi potrebbero essere diversi a causa di guasti del sistema.

Considereremo principalmente sistemi sincroni. Verso la fine del capitolo tratteremo brevemente sistemi asincroni e parzialmente sincroni.

**Sistemi asincroni.** In un sistema asincrono non esiste il concetto di tempo: ogni processore opera alla sua velocità e non c'è relazione fra le velocità dei processori. Analogamente non esiste un limite di tempo per la consegna di un messaggio: ogni messaggio rimane nel canale di comunicazione per un tempo indeterminato prima di esssere consegnato.

**Sistemi parzialmente sincroni.** La parziale sincronia impone dei vincoli sul tempo necessario ad ogni singolo processore per eseguire una istruzione e al tempo necessario a consegnare un messaggio. Pertanto in un sistema parzialmente sincrono potremo fare delle assunzioni sulle velocità dei singolo processori e sul tempo di consegna di un messaggio.

**Relazione.** Un sistema sincrono è un caso particolare di un sistema asincrono, quindi da questo punto di vista progettare un un algoritmo per un sistema sincrono è più facile. Infatti in un sistema sincrono tutti i processori eseguono un'istruzione nello stesso istante e tutti i messaggi vengono immediatamente consegnati. Un algoritmo progettato per un sistema sincrono potrebbe non funzionare (anzi, nella maggior parte dei casi non funziona) in un sistema asincrono. Mentre un algoritmo progettato per un sistema asincrono funzionerà anche in un sistema sincrono. Simmetricamente un risultato di impossibilità per i sistemi sincroni è valido anche per i sistemi asincroni: se non è possibile risolvere il problema in un sistema più facile da gestire, a maggior ragione non è possibile risolvere in un sistema più difficile da gestire.

## *6.2 Problema del consenso*

Il *problema del consenso* è l'astrazione di molti problemi di coordinazione. Dovrebbe essere abbastanza evidente che coordinare le azioni in un sistema distribuito è alla base di qualunque computazione distribuita. Coordinarsi significa mettersi d'accordo, prendere una stessa decisione. Il problema del consenso formalizza queste necessità. La definzione formale del problema ammette delle varianti. Assumeremo che i processori abbiano un input, un valore iniziale, e che debbano produrre un output. Gli output devono essere tutti uguali, questo rappresenta il mettersi d'accordo e devono anche in qualche modo essere funzione dell'input. Quest'ultimo requisito è abbastanza naturale e dal punto di vista formale serve ad evitare soluzioni precostruite che chiaramente non avrebbero senso nella realtà (ad esempio, dare in output sempre 0).

Problemi di consenso possono essere individuati in una infinità di situazioni reali. Ad esempio due processori potrebbero dover mettersi d'accordo sul se annullare o rendere definitiva una transazione in un database distribuito. Oppure due processori potrebbero doversi mettere d'accordo su delle letture indipendenti fatte a dei sensori, ad esempio in un aereo per misurare l'altitudine.

Il problema CONSENSO viene definito come segue. La seguente definizione è generica: in funzione delle particolari caratteristiche del sistema considerato (es. tipo di guasti) la definizione potrà subire delle piccole variazioni; ad esempio non avrebbe senso richiedere che un processore che si guasta in modo bizantino debba soddisfare le proprietà richieste in quanto per definizione di guasto bizantino un processore guasto può comportarsi in maniera arbitraria. Pertanto diamo qui una definizione generale del problema per poi istanziarla in modo più preciso caso per caso.

**Problema** CONSENSO**:** dato un sistema distribuito con $n$ processori $p_1, \ldots, p_n$, che iniziano la computazione con un valore di input $v_i \in V$, $i = 1, \ldots, n$, dove $V$ è l'insieme di tutti i possibili valori di input, i processori devono stabilire un valore di output, una *decisione*, in modo tale che 3 proprietà vengano soddisfatte:

- Accordo: Tutte le decisioni sono uguali.
- Validità: Se tutti i valori iniziali sono uguali, cioè se $v_i = v$ per tutti i processori, allora la decisione deve essere $v$.
- Terminazione: Tutti i processori decidono.

Osserviamo che se il sistema è *affidabile*, cioè non ci sono guasti né dei processori né dei canali di comunicazione, i problemi di consenso sono facili da risolvere: basta scambiarsi gli input ed usare una regola comune per decidere l'output in funzione dell'input. Quindi nel prosieguo considererm il problema del consenso in vari scenari, in ognuno dei quali instanzieremo una specifica *inaffidabilità* del sistema.

## 6.3 *Consenso sincrono con perdita di messaggi*

In questa sezione considereremo la seguente definizione del problema del consenso, detta anche problema dell'attacco coordinato. Uno dei primi risultati relativi al problema del consenso, introduce il problema in un ipotetico scenario di guerra, chiamandolo il *problema dei due generali*, conosciuto anche come il problema dell'attacco coordinato. Immaginiamo una guerra antica in cui si usavano ancora i messaggeri per comunicare, e consideriamo una situazione in cui i generali di due armate di un esercito, devono prendere una decisione sul se attaccare o meno il nemico. I generali possono comunicare solo con dei messaggeri che però possono essere catturati dal nemico. Quindi la comunicazione non è affidabile. Nel caso del problema dei due generali ci sono solo 2 processori, più in generale il numero di processori è arbitrario.

Formalmente abbiamo il seguente problema del consenso nella versione "Attacco Coordinato".

**Problema** CONSENSOAC**:** In un sistema distribuito di $n$ processori, che comunicano tramite scambio di messaggi su un canale che può perdere i messaggi, ogni processore ha in input un bit[1], cioè $V = \{0, 1\}$, e deve decidere il valore di un bit di output. Le 3 proprietà da soddsifare sono

[1] Il valore del bit di input rappresenta la propria opinione sul se attaccare, 1, o non attaccare, 0.

- Accordo: Tutte le decisioni sono uguali (quindi o tutti danno in output 0 oppure tutti danno in output 1).
- Validità: Se tutti i processori iniziano con 0 allora l'output deve essere 0. Se tutti i processori iniziano con 1 e tutti i messaggi vengono consegnati, allora l'output deve essere 1.
- Terminazione: Tutti i processori decidono.

### *6.3.1 Sistema deterministico*

In un sistema sincrono, deterministico, e comunicazione non affidabile il problema ConsensoAC non può essere risolto.

**Teorema 6.3.1** *Consideriamo un sistema distribuito con due processori* 1 *e* 2*, connessi da un canale di comunicazione. Non esiste un algoritmo che possa risolvere il problema* ConsensoAC*.*

Dimostrazione. Per contraddizione, assumiamo che esista un algoritmo $A$ che risolve il problema. Una qualunque esecuzione dell'algoritmo dipende esclusivamente dall'input dei due processori e dallo specifico pattern di consegna dei messaggi dell'esecuzione. Senza perdere in generalità assumiamo che l'algoritmo $A$ faccia spedire ad entrambi i processori un messaggio in ogni round, in quanto se così non fosse potremmo usare dei messaggi fittizi.

Sia $\alpha$ l'esecuzione in cui i due processori hanno entrambi come input 1 e in cui tutti i messaggi vengono consegnati, cioè non ci sono guasti sui canali di comunicazione. Dalla proprietà di terminazione si ha che i processori devono decidere e dalla proprietà di validità si ha che tale decisione deve essere 1. Sia $r$ il round entro il quale entrambi decidono. Sia $\alpha_0$ l'esecuzione che differisce da $\alpha$ solo per il fatto che tutti i messaggi dopo i primi $r$ round vengono persi. Quindi $\alpha_0$ è identica ad $\alpha$ per i primi $r$, ma diversa dal round $r+1$ in poi in quanto i messaggi spediti nel round $r$ non verranno consegnati. Poichè le due esecuzioni sono indistiguibili fino al round $r+1$ (la perdita dei messaggi spediti al round $r$ potrà essere rilevata dai processori solo al round $r+1$), sia $p_1$ che $p_2$ decideranno 1 anche in $\alpha_1$, visto che in $\alpha$ decidono 1 prima del round $r+1$.

A questo punto consideriamo una sequenza $\alpha_1, \alpha_2, \ldots$ di esecuzioni costruite a partire da $\alpha_0$ in cui ogni esecuzione successiva è uguale alla precendete a parte che l'ultimo messaggio non viene consegnato. Più precisamente, sia $\alpha_1$ l'esecuzione indentica ad $\alpha_0$ tranne che l'ultimo messaggio da $p_1$ a $p_2$ viene perso, e sia $\alpha_2$ l'esecuzione indentica ad $\alpha_1$ tranne che l'ultimo messaggio da $p_2$ a $p_1$ viene perso. La Figura 6.1 mostra graficamente le due esecuzioni.

Consideriamo $\alpha_1$. Essa, per il processore $p_1$ è indistinguibile da $\alpha_0$. Pertanto $p_1$, che decide 1 in $\alpha_0$, deciderà 1 anche in $\alpha_1$. Per la proprietà di accordo, anche il processore $p_2$ deciderà 1 in $\alpha_1$.

Consideriamo ora $\alpha_2$. Essa, per il processore $p_2$ è indistinguibile da $\alpha_1$. Pertanto $p_2$, che decide 1 in $\alpha_1$, deciderà 1 anche in $\alpha_2$. Per la proprietà di accordo, anche il processore $p_1$ deciderà 1 in $\alpha_2$.

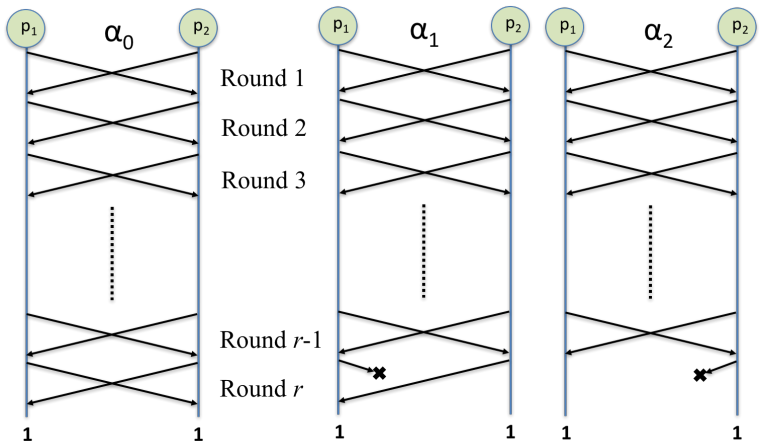



Figura 6.1: Esecuzioni $\alpha_0, \alpha_1$ e $\alpha_2$

Le esecuzioni $\alpha_3$ e $\alpha_4$ sono definite in modo analogo, e per un ragionamento simile si ha che $p_1$ e $p_2$ decidono 1 anche in $\alpha_3$ e $\alpha_4$. Procedento in questo modo si arriverà a una esecuzione $\alpha'$ in cui nessun messaggio viene consegnato ma i processori comunque decidono 1 (ricordiamo che l'input dei processori è sempre 1 in tutte le esecuzioni finora considerate).

A questo punto consideriamo l'esecuzione $\alpha''$, identica ad $\alpha'$, quindi senza messaggi consegnati, ma in cui $p_2$ ha come input 0. Per $p_1$ non c'è differenza fra $\alpha'$ e $\alpha''$, quindi $p_1$ decide comunque 1 in $\alpha''$. Di conseguenza anche $p_2$ decide 1 in $\alpha''$. Sia $\alpha'''$ l'esecuzione identica ad $\alpha''$, con la differenza del valore di input di $p_1$ che è 0. Per il processore $p_2$ non c'è differenza fra $\alpha''$ e $\alpha'''$, quindi $p_1$ decide 1 anche in $\alpha'''$. Di conseguenza, per la proprietà di accordo, anche $p_1$ decide 1 in $\alpha'''$. Questo però è una violazione della proprietà di validità in quanto in $\alpha'''$ entrambi i processori hanno come input 0 ma decidono 1. Pertanto aver assunto che esiste un algoritmo $A$ che risolve il problema ha portato a un contraddizione. □

Il Teorema 6.3.1 dimostra una chiara limitazione: c'è poco da fare se la comunicazione è inaffidabile. Ovviamente in pratica è molto difficile perdere tutti i messaggi. Inoltre possiamo anche o rilassare i requisiti del problema oppure rendere più potente il modello di computazione. Ad esempio possiamo utilizzare la randomizzazione.

### 6.3.2 *Algoritmo randomizzato*

Consideriamo un sistema di $n$ processori che possono utilizzare delle scelte casuali. Ogni processore inizia con un valore di input in $\{0,1\}$. Assumeremo anche che l'algoritmo deve terminare l'esecuzione in un numero fissato $r \geq 1$ di round; in altre parole ogni processore deve decidere al più tardi durante il round $r$. Assumeremo che ogni processore ha una variabile *write-once* con valori in $\{0,1\}$, la cui scrittura rappresenta l'azione della decisione. Ad ogni round verrà spedito un messaggio da ogni processore (possiamo usare dei messaggi fittizi se non c'è niente da spedire), ed un qualsiasi numero di messaggi può non arrivare a destinazione. Per poter gestire le scelte casuali la proprietà di accordo verrà leggermente rilassata:

- Accordo: Tutte le decisioni sono uguali (quindi o tutti danno in output 0 oppure tutti danno in output 1) con una probabilità pari ad almeno $1 - \epsilon$, dove $\epsilon$ è un numero positivo piccolo a piacere.

Per poter formalizzare l'algoritmo e la sua analisi definiamo un *pattern di comunicazione* come un sottoinsieme dell'insieme

$$\{(i, j, k) : (i, j) \text{ è un canale di comunicazione, e } 1 \leq k\}.$$

Ogni elemento $(i, j, k)$ di un pattern di comunicazione rappresenta la spedizione di un messaggio da $i$ a $j$ durante il round $k$. Un pattern di comunicazione $\gamma$ è detto *ammissibile* se per ogni $(i, j, k) \in \gamma$ si ha che $k \leq r$.

Per analizzare l'algoritmo assumeremo che esiste un avversario che sceglie quali messaggi far arrivare a destinazione e quali no. Pertanto l'avversario sceglierà uno specifico pattern di comunicazione ammissibile; ovviamente l'avversario potrà scegliere anche l'input dei processori. Non ha, invece, controllo sulle scelte casuali dei processori.

Per una fissata strategia (pattern di comunicazione) dell'avversario, le scelte casuali effettuate dai processori determinano in modo univoco una particolare esecuzione. Quindi le scelte probabilistiche determinano una distribuzione di probabilità sull'insieme delle possibili esecuzioni.

Per semplicità di esposizione assumeremo che il grafo delle comunicazioni è completo: ogni processore può spedire messaggi ad ogni altro processore. L'algoritmo si può facilmente estendere al caso di grafi di comunicazione non completi. Durante l'esecuzione dell'algoritmo, ogni processore tiene traccia di ciò che conosce riguardo ai valori iniziali degli altri processi.

Definiamo un ordine parziale $\leq_\gamma$ relativo al pattern di comunicazione $\gamma$, sulle coppie $(i, k)$, dove $i$ è un processore e $k \geq 0$ un intero che rappresenta il numero di un round. Quando diremo "al tempo $k$", intenderemo l'istante di tempo esattamente dopo l'esecuzione di $k$ round, e quindi prima dell'inizio del $(k + 1)$-esimo round. L'ordine parziale $\leq_\gamma$ rappresenta il flusso di informazione convogliato dai messaggi ed è definito come segue:

- $(i, k) \leq_\gamma (i, k')$ per ogni $i$ e tutti i $k, k'$ tali che $0 \leq k \leq k'$.
- Se $(i, j, k) \in \gamma$ allora $(i, k - 1) \leq_\gamma (j, k)$.
- Se $(i, k) \leq_\gamma (i', k')$ e $(i', k') \leq_\gamma (i'', k'')$ allora $(i, k) \leq_\gamma (i'', k'')$.

Il primo caso descrive il flusso di informazione di un singolo processore: la conoscenza di un processore può solo aumentare con il procedere dell'esecuzione. Il secondo caso descrive il flusso di informazione convogliato da un messaggio: se $j$ riceve un messaggio da $i$ al round $k$ allora ciò che conosce $i$ al round $k - 1$ sarà conosciuto da $j$. Questo ovviamente significa che quando un processore spedisce un messaggio include tutta la sua conoscenza. Infine il terzo caso è semplicemente la proprietà transitiva.

Per un pattern di comunicazione ammissible $\gamma$, definiamo il *livello di informazione*, $livello_\gamma(i, k)$ per un qualsiasi processore $i$ e un tempo $k, 0 \leq k \leq r$, in modo ricorsivo come segue:

- $k = 0$. Allora $livello_\gamma(i,k) = 0$.
- $k > 0$ e c'è un qualche $j \neq i$ tale che $(j,0) \not\leq_\gamma (i,k)$. Allora $livello_\gamma(i,k) = 0$.
- $k > 0$ e $(j,0) \leq_\gamma (i,k)$ per ogni $j \neq i$. Sia per ogni $j \neq i$, $l_j = \max\{livello_\gamma(j,k') | (j,k') \leq_\gamma (i,k)\}$. Si noti che $l_j$ è il più alto livello che $i$ sa che $j$ ha raggiunto. Si noti inoltre che $0 \leq l_j \leq k-1, \forall j$. Allora $livello_\gamma(i,k) = 1 + \min\{l_j | j \neq i\}$.

Informalmente, ogni processore inizia con un livello di informazione pari a 0. Quando riceve messaggi da tutti gli altri processori passa al livello di informazione 1; quando sa che tutti gli altri processori hanno raggiunto il livello 1 passa al livello 2, e così via.

La Figura 6.2 mostra un esempio per il pattern di comunicazione

$$\gamma = \{(1,2,1),(1,2,2),(2,1,2),(1,2,3),(2,1,4),(1,2,5),(2,1,5),(1,2,6)\}.$$

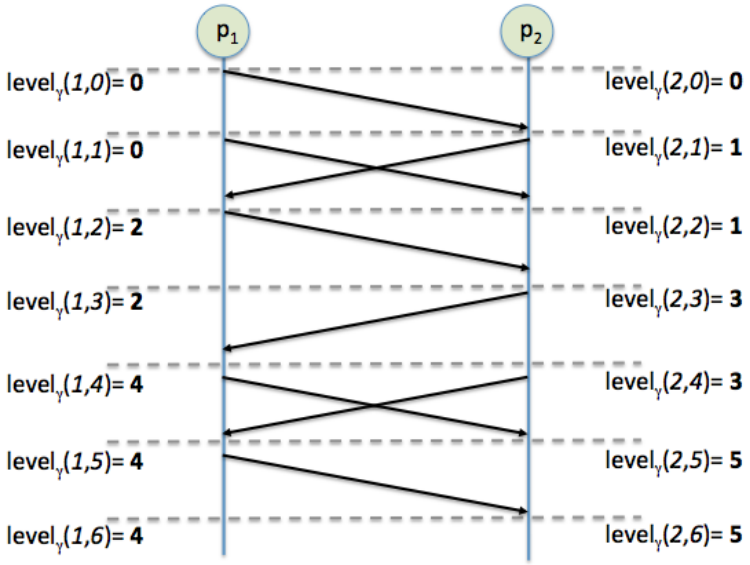



Figura 6.2: Livello di informazione

**Lemma 6.3.2** *Per un qualsiasi pattern di comunicazione ammissibile $\gamma$, un qualsiasi $k$, $0 \leq k \leq r$, e qualsiasi $i$ e $j$, si ha che $|livello_\gamma(i,k) - livello_\gamma(j,k)| \leq 1$.*

Dimostrazione. Il livello di comunicazione di un processore $i$ può essere incrementato a un valore $s+1$ solo quando il processore $i$ viene a sapere che il livello di comunicazione di $j$ è $s$. E viceversa. Quindi non è possibile che i livelli di comunicazione di due processori differiscano per più di 1. □

**Lemma 6.3.3** *Se $\gamma$ è il pattern di comunicazione "completo", cioè che contiene tutte le possibili triple $(i,j,k)$, $1 \leq k \leq r$, allora $livello_\gamma(i,k) = k$, per tutti i valori di $i$ e $k$.*

Dimostrazione. Se nessun messaggio si perde, il livello di comunicazione viene incrementato di 1 ad ogni round. □

A questo punto siamo pronti a descrivere l'algoritmo che chiameremo RANDOMATTACK. Ogni processore $i$ mantiene traccia del suo livello di informazione in una variabile *level*. Inoltre, il processore 1 sceglie un valore casuale *key*, un intero nell'intervallo $[1, r]$; questo valore sarà aggiunto a tutti i messaggi spediti. Anche i valori iniziali di tutti i processori verranno aggiunti a tutti i messaggi spediti. Dopo $r$ round, ogni processore deciderà usando la seguente regola: se il valore del livello di informazione *level* è maggiore o uguale a *key*, allora la decisione è 1, altrimenti la decisione è 0.

**Algorithm 21:** RANDOMATTACK$_i$

**Stato:**
$rounds \in \mathbb{N}$, inizialmente 0
$decision \in \{unknown, 0, 1\}$, inizialmente *unknown*
$key \in [1, r] \cup undef$, inizialmente *undef*
for every $j, 1 \leq j \leq n$
  $val(j) \in \{undef, 0, 1\}$, inizialmente *undef* per $j \neq i$ e input di $i$ per $j = i$
  $level(j) \in [-1, r]$, inizialmente -1 per $j \neq i$ e 0 per $j = i$

**Randomizzazione:**
Se $i = 1$ e $rounds = 0$ allora $key := random([1, r])$

**Generazione messaggi:**
Spedisci $(L, V, key)$ dove $L$ è il vettore dei livelli $level(j)$ e $V$ è il vettore dei valori $val(j)$

**Computazione:**
$rounds := rounds + 1$
Sia $(L_j, V_j, k_j)$ il messaggio ricevuto da $j$, per ogni $j$ da cui si riceve un messaggio
**if** *per un qualche* $j$, $k_j \neq undef$ **then**
  $key := k_j$
**for** *tutti i* $j \neq i$ **do**
  Se per qualche $i'$, $V_{i'}(j) \neq undef$ allora $val(j) := V_{i'}(j)$
  Se per qualche $i'$, $L_{i'}(j) > level(j)$ allora $level(j) := \max_{i'}\{L_{i'}(j)\}$
$level(i) := 1 + \min\{level(j) | j \neq i\}$
**if** $rounds = r$ **then**
  **if** $key \neq undef$ *and* $level(i) \geq key$ *and* $\forall j, val(j) = 1$ **then**
    $decision := 1$
  **else**
    $decision := 0$

**Teorema 6.3.4** RANDOMATTACK *risolve il problema* CONSENSOAC *con una probabilità di errore di* $\epsilon = 1/r$.

DIMOSTRAZIONE. Osserviamo prima che l'algoritmo calcola correttamente i livelli di

informazione. Cioè in ogni esecuzione con un pattern di comunicazione ammissibile $\gamma$, per ogni $k$, $0 \leq k \leq r$, e per ogni $i$, dopo $k$ round si ha che il valore di $level(i)$ è effettivamente uguale a $livello_\gamma(i,k)$. Inoltre, osserviamo che dopo $k$ round, se $level(i)_i \geq 1$, allora sono definiti sia $key_i$ che $val(j)_i$ per tutti gli $i$ e tali valori sono uguali, rispettivamente, al valore casuale scelto del processore 1 nel primo round e ai valori iniziali dei processori.

L'algoritmo soddisfa la proprietà di terminazione: tutti i processori decidono nel round $r$. Proviamo che l'algoritmo soddisfa la proprietà di validità. Se tutti i processori iniziano con 0, la decisione sarà 0 in quanto per decidere 1 è necessario che tutti i $val(j)$ siano 1. Se tutti i processori iniziano con 1 e tutti i messaggi vengono consegnati, dal Lemma 6.3.3 e dal fatto che l'algoritmo calcola correttamente i livelli di informazione si ha che per ogni $i$, $level(i)_i = r$ nel momento in cui viene presa la decisione; poiché $r \geq 1$, si ha anche che sono definiti sia $key_i$ che $val(j)_i$ per tutti gli $i$ ed essendo $key \leq r$, la decisione è 1.

Rimane da provare che l'algoritmo soddisfa la proprietà di accordo con una certa probabilità. Per ogni $i$, sia $\ell_i$ il valore di $level(i)_i$ nel momento in cui il processore $i$ prende la sua decisione (nel round $r$). Il Lemma 6.3.2 ci dice che gli $\ell_i$ distano al massimo 1. Cominciamo con il distinguere due possibili casi: (1) almeno un processore inizia con 0 e (2) tutti i processori iniziano con 1. Nel primo di questi due casi tutti i processori decidono 0 quindi non c'è errore. L'errore è possibile solo quando tutti i processori hanno come input 1, perché questo è l'unico caso in cui l'istruzione **if** per la decisione può far eseguire ad alcuni processori la parte **then** e ad altri la parte **else**, in funzione dei valori di *key* e degli $\ell_i$.

Se il valore scelto per *key* è maggiore di $\max\{\ell_i\}$, allora tutti i processori decidono 0 e non c'è nessun errore. Se invece $key \leq \min\{\ell_i\}$, allora tutti decidono 1 e non c'è nessun errore. Pertanto l'unico caso in cui c'è disaccordo sulla decisione, cioè alcuni processori decidono 1 e altri 0 si ha quando $key = \max\{\ell_i\}$. La probabilità di un tale evento è $1/r$, visto che $\max\{\ell_i\}$ è determinato dall'avversario ed è un valore fra 0 e $r$ e *key* è scelto casualmente in modo uniforme fra 0 e $r$. □

## 6.4 *Consenso sincrono, deterministico*

In questa sezione considereremo il problema in sistemi sincroni deterministici (cioè senza l'uso della randomizzazione) e considereremo sia guasti di tipo stop, sia guasti di tipo bizantino.

### 6.4.1 *Con guasti stop, algoritmo* EIGStop

Consideriamo ora il problema del consenso in un sistema distribuito con la possibilità che i processori possano guastarsi. Il grafo di comunicazione è completo e la comunicazione è affidabile. Se un processore si guasta durante la spedizione di un messaggio in broadcast, solo un sottoinsieme (che potrebbe anche essere vuoto) dei destinatari riceverà il messaggio. I possibili valori di input e quelli di output sono presi da un insieme $V$.

La definizione formale del problema, che chiameremo ConsensoGS, consenso per guasti stop, è la seguente.

**Problema** ConsensoGS**:** Consideriamo un sistema distribuito di $n$ processori, che comunicano tramite scambio di messaggi su un canale affidabile. I processori possono rompersi con dei guasti stop. Ogni processore ha un valore di input $v_i \in V$ e i processori devono decidere un valore di output in modo tale da soddisfare le seguenti 3 proprietà.

- (Accordo) Tutte le decisioni sono uguali.

- (Validità) Se tutti i processori iniziano con $v \in V$ allora l'output deve essere $v$.

- (Terminazione) Tutti i processori che non si guastano, decidono.

Assumeremo che ci sia un limite $f$, $0 \leq f \leq n$, al numero di guasti che si possono verificare in ogni singola esecuzione. I casi $f = 0$ e $f = n$, vengono solitamente indicati con *failure-free* e *wait-free*:

- *failure-free:* $f = 0$, non ci sono guasti quindi tutti i processori decidono.

- *wait-free:* $f = n$, tutti i processori si possono rompere. Tuttavia, ogni processore non guasto deciderà, indipendentemente dai guasti degli altri processori.

Per questo problema, poichè i processori si guastano semplicemente fermandosi, è possibile dare un algoritmo molto semplice, che chiameremo FloodSet. L'algoritmo "inonda" la rete con l'insieme dei valori di input e poi utilizza una regola prestabilita per decidere.

Sia $V$ l'insieme dei possibili valori inziali e sia $v_0$ un fissato elemento di $V$. Sia $f$ il numero massimo di guasti. Ogni processore manterrà nella variabile $W$ l'insieme dei valori di input di cui conosce l'esistenza. Inizialmente per ogni processore $i$ si avrà $W_i = \{v_i\}$. In ogni round tutti processori spediranno a tutti gli altri processori l'insieme $W$ e il processore $i$ ricevendo $W_j$ aggiornerà il proprio insieme ponendo $W_i = W_i \cup W_j$. Al round $f + 1$, ogni processore deciderà usando la seguente regola: se $|W_i| = 1$ allora il processore $i$ decide $v$, dove $v$ è l'unico elemento di $W_i$; se invece $|W_i| > 1$ allora il processore $i$ deciderà $v_0$.

**Algorithm 22:** FLOODSET$_i$

**Stato:**
*rounds* $\in \mathbb{N}$, inizialmente 0
*decision* $\in V \cup \{unknown\}$, inizialmente *unknown*
$W \subseteq V$, inizialmente $\{v_i\}$

**Generazione messaggi:**
**if** *rounds* $\leq f$ **then**
  spedisci $W$ agli altri processori

**Computazione:**
*rounds* := *rounds*+1
Sia $X_j$ il messaggio in arrivo da $j$, per ogni $j$ da cui un messaggo arriva
$W = W \cup \bigcup_j X_j$
**if** *rounds* $= f + 1$ **then**
  **if** $|W| = 1$ **then**
    *decision* $:= v$, dove $W = \{v\}$ **else**
      *decision* $:= v_0$

La Figura 22 mostra l'algoritmo FLOODSET. Non forniamo una prova formale della correttezza dell'algoritmo, ma dovrebbe essere abbastanza facile convincersi che dopo $f + 1$ rounds tutti i processori hanno il medesimo valore per $W$ e quindi fanno tutti la stessa scelta raggiungendo il consenso. Infatti diversi valori di $W$ sono possibili solo quando ci sono dei guasti ma poichè l'algoritmo viene eseguito per $f + 1$ round, sicuramente ci sarà un round senza guasti che garantirà l'uniformità della conoscenza di $W$.

Per comprendere meglio consideriamo il seguente esempio che è un caso limite in cui in ogni round un singolo processore si rompe facendo sì che la conoscenza non sia uniforme. Abbiamo un sistema di 5 processori e $f = 3$. La conoscenza iniziale dei processori è

$$\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}$$

Nel primo round $p_1$ spedisce il proprio valore solo a $p_2$ e poi si guasta. Quindi la conoscenza dei processori diventa:

$$guasto, \{v_1, v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\},$$

Nel secondo round $p_2$ spedisce la propria conoscenza solo a $p_3$ e poi si guasta. Quindi la conoscenza dei processori diventa:

$$guasto, guasto, \{v_1, v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\},$$

Nel terzo round $p_3$ spedisce la propria conoscenza solo a $p_4$ e poi si guasta. Quindi la conoscenza dei processori diventa:

$$guasto, guasto, guasto, \{v_1, v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\},$$

Nel quarto round non ci possono essere più guasti per cui $p_4$ invierà il valore $v_1$ anche a $p_5$ e pertanto alla fine del round la conoscenza sarà:

$$guasto, guasto, guasto, \{v_1, v_2, v_3, v_4, v_5\}, \{v_1, v_2, v_3, v_4, v_5\}.$$

In pratica, quello che succede è che al primo round senza guasti la conoscenza diventa uniforme. Dopo $f+1$ coi deve necessariamente essere un tale round. L'esempio fatto prima è un caso limite in cui il round senza guasti è il round $f+1$.

È immediato vedere che l'algoritmo termina dopo $f+1$ round, e che il numero di messaggi spediti è $O((f+1)n^2))$. Occorre osservare che ogni messaggio deve specificare un insieme che può contenere fino a $n$ elementi, quindi la reale complessità dei messaggi, supponendo di poter usare un numero fissato $b$ di bit per rappresentare ogni valore, diventa $O((f+1)n^3b)$, in quanto gli insiemi contenuti nei messaggi possono avere fino a $n$ valori e per ognuno dei valori abbiamo bisogno di $b$ bit.

Vediamo adesso un approccio al problema ConsensoGS un po' più complicato. Chiameremo EIGStop l'algoritmo risultante. Questo approccio è molto oneroso per il caso dei guasti stop, per il quale l'algoritmo FloodSet risolve il problema in modo molto più efficiente. Tuttavia EIGStop sarà un utilissimo punto di partenza per risolvere il problema nel caso dei guasti bizantini. L'idea di base è che ogni processore comunichi a ogni altro processore, in ogni round, non solo tutto quello che sa, ma anche come lo ha saputo. All'inizio della computazione ogni processore conosce solo il proprio valore iniziale e quindi nel primo round può comunicare questo valore a tutti gli altri processori. Nel secondo round ogni processore, oltre al proprio valore iniziale, conosce anche i valori iniziali degli altri processori (almeno di quelli dai quali è arrivato un messaggio, visto che i processori possono rompersi). Tale informazione è arrivata direttamente dai processori che l'hanno prodotta: è il processore $i$ a comiunicare $v_i$, il proprio valore iniziale, a tutti gli altri processori. Nei round successivi le informazioni possono essere propagate indirettamente nel senso che un processore potrà dire di aver saputo da un altro processore che altri processori avevano quei particolari valori iniziali. Ad esempio nel terzo round il processore $k$ potrà sapere dal processore $j$ che il valore iniziale del processore $i$ è $v_i$. Nei round successivi la catena di "questo processore mi ha detto che" si allunga. Questo approccio è molto robusto in quanto l'informazione si diffonde nel modo più ampio e completo possibile; ovviamente è estremamente oneroso in termini di numero e grandezza dei messaggi.

Per formalizzare l'algoritmo faremo ricorso a una struttura dati che chiameremo albero EIG (Exponential Information Gathering), riportato nella Figura 6.3. Ogni processore manterrà una sua copia di questa struttura dati che verrà riempita con le informazioni che arriveranno dagli altri processori. Per rendere uniforme la descrizione assumeremo che un processore invii un messaggio anche a sé stesso; in pratica questo non avviene ma è ovviamente molto semplice simulare la spedizione di un tale messaggio.

La radice dell'albero del processore $i$, etichettata con $\lambda$, conterrà il valore iniziale $v_i$ del processore $i$. I nodi del primo livello conterranno i valori iniziali degli altri processori così come ricevuti direttamente dagli altri processori, cioè il nodo con etichetta $j$ conterrà il valore $v_j$ che $i$ ha ricevuto da $j$. I nodi del secondo livello, invece, conterranno tutte le

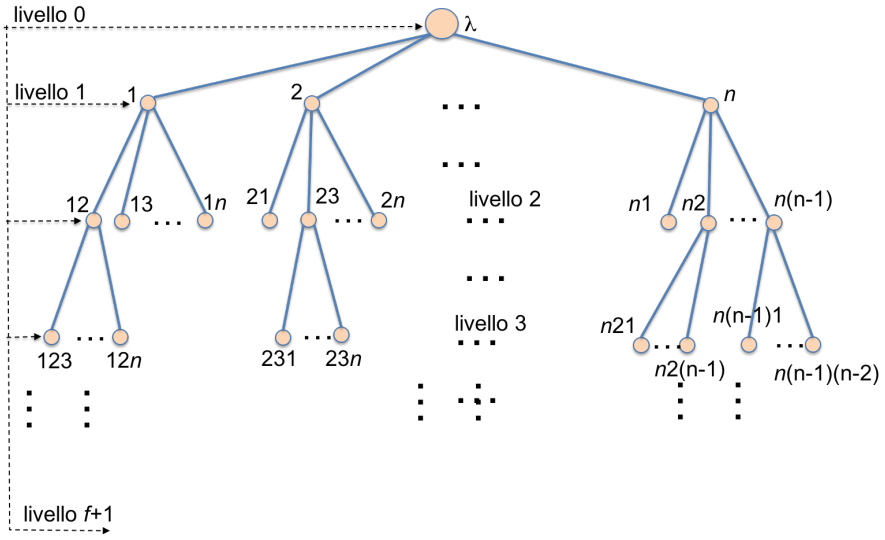



Figura 6.3: Albero EIG per $n$ processori e $f$ guasti.

informazioni ricevute indirettamente tramite un altro processore; l'etichetta del nodo specifica il "cammino" dell'informazione. Ad esempio il nodo con etichetta 12 conterrà il valore iniziale del processore 1 così come questo è stato comunicato prima da 1 a 2 e poi da 2 al processore $i$. In generale un nodo con etichetta $j_1 j_2 \dots j_k$ nell'albero del processore $i$ al livello $k$, conterrà il valore iniziale del processore $j_1$ che il processore $i$ ha ricevuto dal processore $j_k$ che a sua volta lo ha ricevuto dal processore $j_{k-1}$ che a sua volta lo ha ricevuto dal processore $j_{k-2}$ e così via fino al processore $j_1$ che lo ha inviato a $j_2$ nel primo round. La Figura 6.4 mostra l'albero EIG per $n = 4$ e $f = 2$.

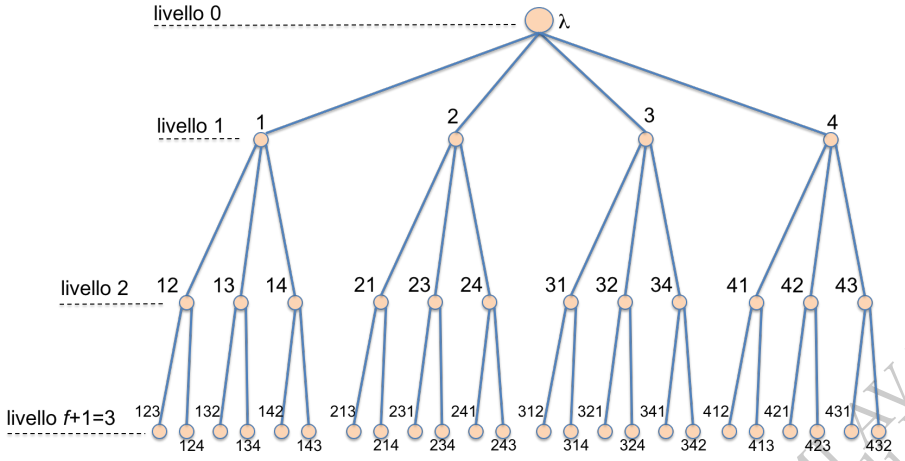



Figura 6.4: Albero EIG per $n = 4$ processori e $f = 2$ guasti.

L'algoritmo EIGSTOP, Algoritmo 23, funziona nel seguente modo. Per ogni etichetta $x$ nell'albero EIG, ogni processore memorizza il valore corrispondente a tale etichetta nella variabile $val(x)$. All'inizio della computazione il processore $i$ assegna il valore $v_i$ alla variabile $val(\lambda)$.

Nel primo round il processore $i$ invia $val(\lambda)$ a tutti gli altri processori e alla ricezione

di un valore $v$ da $j$ esegue l'istruzione $val(j) := v$.

Nei round successivi, cioè in ogni round $k$, con $2 \leq k \leq f+1$, il processore $i$ spedisce a tutti gli altri processori il messaggio $(x, v)$, dove $x$ è un'etichetta del livello $k-1$ dell'albero EIG tale che $x$ non contiene $i$ e $v = val(x)$. Quindi il processore $i$, alla ricezione dei messaggi inviati dagli altri processori mnemorizza le informazioni in essi contenute. Cioè, se dal processore $j$ arriva il messaggio $(x, v)$, con $x$ sequenza di indici che non contiene $j$, allora $i$ esegue l'istruzione $val(xj) = v$.

Alla fine del round $f+1$, il processore $i$ applica la regola di decisione: sia $W$ l'insieme dei valori che sono memorizzati in tutti i nodi dell'albero EIG; se $W$ ha cardinalità 1 allora il processore decide sull'unico valore presente in $W$, altrimenti decide un valore prestabilito $v_0$.

**Algorithm 23:** EIGSTOP$_i$

**Stato:**

Per ogni etichetta $x$ dell'albero

$val$(x) $\in V \cup \{\perp\}$, inizialmente $v_i$ per $x = \lambda$ e $\perp$ altrimenti

**Generazione messaggi:**

Round 1:

Invia $val(\lambda)$ a tutti (incluso $i$ stesso)

Round $k$, $2 \leq k \leq f+1$:

Invia $(x, val(x))$ a tutti, per tutte le etichette $x$ del livello $k-1$ con $i \notin x$.

**Computazione:**

Round 1:

**if** $v \in V$ arriva da $j$ **then** $val(j) := v$ **else** $val(j) := \perp$

Round $k$, $2 \leq k \leq f+1$:

**if** $(x, v)$ arriva da $j$, con $j \notin v$ **then** $val(xj) := v$ **else** $val(xj) := \perp$

**if** *siamo al round* $f+1$ **then**

Sia $W$ l'insieme dei valori $val(j)$, eslcuso $\perp$

**if** $W = \{v\}$ **then** decidi $v$ **else** decidi $v_0$

Non dovrebbe essere difficile vedere che i valori memorizzati nell'albero EIG sono quelli spiegati prima, cioè se il valore $v$ viene memorizzato nel nodo con etichetta $i_1 \dots i_k$, con $1 \leq k \leq f+1$, allora significa che il processore $i_k$ ha comunicato nel round $k$ al processore $i$ che il processore $i_{k-1}$ ha comunicato nel round $k-1$ al processore $i_k$ che il processore $i_{k-2}$ ha comunicato nel round $k-2$ al processore $i_{k-1}$ che ... il processore $i_1$ ha comunicato nel round 1 al processore $i_2$ che il valore iniziale di $i_1$ è $v$. Inoltre se il valore assegnato a un nodo con etichetta $i_1 \dots i_k$ è `null`, allora vuol dire che nella catena di messaggi da $i_1$ a $i$ a un certo punto nessun messaggio è stato spedito da $i_j$ a $i_{j+1}$, $j = 1, \dots, k-1$ nel round $j$ o da $i_k$ ad $i$ nel round $k$.

Facciamo un esempio per rendere chiaro l'algoritmo. Consideriamo un sistema con $n = 3$ processori ed $f = 1$. La Figura 6.5 mostra gli alberi EIG dei 3 processori quando non si verificano guasti. I valori iniziali $v_1, v_2$ e $v_3$ dei 3 processori vengono correttamente propagati su tutti i messaggi e alla fine del secondo round tutti i processori

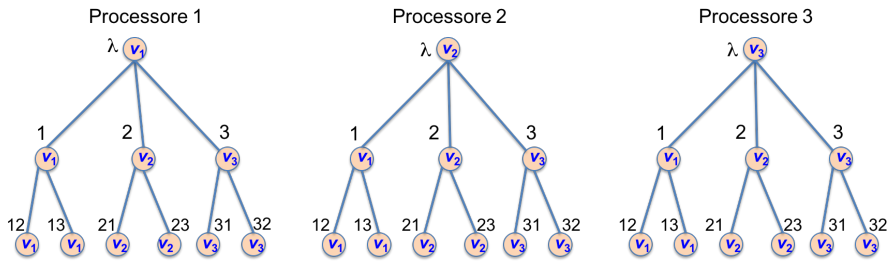



Figura 6.5: Alberi EIG con valori per un'esecuzione con $n = 3$ e $f = 1$.

avranno $W = \{v_1, v_2, v_3\}$.

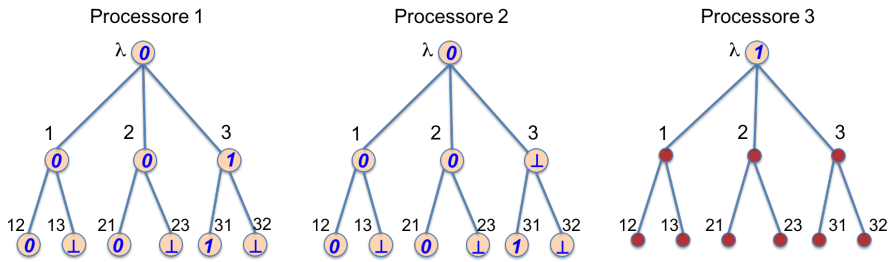



Figura 6.6: Alberi EIG con valori per $n = 3$ e $f = 1$, con un'esecuzione in cui il processore 3 si guasta subito.

La Figura 6.6 mostra gli alberi EIG dei 3 processori in una esecuzione in cui il processore 3 si guasta subito dopo aver spedito il suo valore iniziale. I valori iniziali dei processori sono $0, 0$ e 1. Alla fine del secondo round i processori 1 e 2 avranno $W = \{0, 1\}$.

Proviamo adesso che l'algoritmo è corretto. Iniziamo con il provare che se due processori non si guastano, allora nel momento della decisione avranno lo stesso valore per $W$.

**Lemma 6.4.1** *Se i processori $i$ e $j$ sono entrambi non guasti, allora dopo aver ricevuto i messaggi del round $f + 1$, si ha $W_i = W_j$.*

Dimostrazione. Forniamo una prova informale; il lettore interessato può trovare una prova più dettagliata in L96 [26]. Sserviamo innanzitutto che l'insieme $W$ è l'insieme dei valori iniziali di cui il processore è venuto a conoscenza durante l'algoritmo. Inoltre, poichè i processori $i$ e $j$ sono entrambi non guasti ogni messaggio che inviano arriva all'altro. Ovviamente sia $i$ che $j$ conosceranno il valore iniziale dell'altro in quanto entrambi lo inviano nel primo round. Che possiamo dire della loro conoscenza del valore iniziale $v_k$ di un altro processore $k$? Se $k$ non è guasto, allora sicuramente $v_k$ comparirà nell'albero di $i$ e $j$ già dal primo round. Se $k$ invece si guasta potrebbe non comparire subito negli alberi di $i$ e $j$. Infatti $k$ potrebbe essere riuscito a inviare un messaggio a un altro processore $k'$ che successivamente si è guastato, che a sua volta, nel round successivo è riuscito a inviare la conoscenza di $v_k$ a un solo altro processore $k''$, e così via. Tuttavia, poichè ci sono $f$ guasti questa sequenza di scenari con informazioni discordanti (solo un processore, o più in generale alcuni processori, conoscono il valore $v_k$) non può durare più di $f$ round. Per cui, o il valore di $v_k$ viene "perso" nel senso che

nessuno dei processori non guasti lo riceve mai, oppure, mal che vada, nel round $f+1$ verrà comunicato a tutti i processori non guasti, quindi inclusi $i$ e $j$. □

In realtà quanto appena detto è una reiterazione dell'argomentazione usata per FloodSet. Infatti EIGStop, relativamente ai valori iniziali degli altri processori, mantiene esattamente le stesse informazioni di FloodSet. Proviamo ora che l'algoritmo è corretto.

**Teorema 6.4.2** *L'algoritmo* EIGStop *permette di raggiungere il consenso in un sistema sincrono con un massimo di $f$ guasti stop.*

Dimostrazione. La proprietà di terminazione è ovviamente verificata: tutti i processori producono un output al round $f+1$.

Vediamo la propietà di validità. Supponiamo che tutti i valori iniziali siano pari a $v$. Allora $v$ è l'unico valore che può essere inserito in $W$. Poichè ogni processore non guasto inserisce $v$ nel proprio insieme $W$, si ha che nel round $f+1$, tutti i processori hanno $W = \{v\}$. Quindi la decisione sarà $v$.

Infine la proprietà di accordo deriva direttamente dal Lemma 6.4.1 che garantisce che $W_i = W_j$ per tutti i processori non guasti. □

L'algoritmo EIGStop termina in $f+1$ round, quindi la complessità di tempo è $O(f+1)$ ed utilizza $O((f+1)n^2)$ messaggi; tuttavia i messaggi hanno lunghezza variabile in quanto ogni processore deve comunicare ad ogni altro processore molte coppie di etichette e valori; le etichette sono costituite da tutti i possibili cammini tramite i quali si è ricevuto il valore e hanno lunghezza crescente con il numero di round. Pertanto, il numero di *bit* necessari per spedire i messaggi è esponenziale nel numero di guasti: $O(n^{f+1}b)$, dove $b$ è il numero di bit necessari a rappresentare un indice. Un miglioramento si può ottenere osservando che la conoscenza esatta di $W$ è necessaria solo quando $|W| = 1$, nel qual caso occorre sapere quale è il singolo valore presente in $W$ per poter decidere. Pertanto si può usare una variante dell'algoritmo in cui ogni processore effettua solo 2 spedizioni: una al round 1 con il proprio valore iniziale e la seconda al round $r$, con $2 \leq r \leq f+1$, quando il processore viene a conoscenza di un'altro valore iniziale diverso dal suo valore iniziale. I dettagli vengono lasciati come esercizio.

**Commento.** L'algoritmo EIGStop funziona per guasti stop. Lo abbiamo introdotto in quanto ci sarà utile per fonire un nuovo algoritmo, apportando delle modifiche a EIGStop, capace di funzionare anche con guasti bizantini. Tuttavia, EIGStop funziona anche con una interessante restrizione dei guasti bizantini: se la comunicazione può essere autenticata, usando ad esempio la firma digitale, allora l'algoritmo EIGStop può tollerare anche guasti bizantini.

### 6.4.2 *Con guasti bizantini, algoritmo* EIGByz

Consideriamo adesso il caso di guasti bizantini. Un processore guasto può esibire un comportamento arbitrario. Per guasti reali questo significa tipicamente un comportamento casuale. I guasti bizantini modellano però un ben più problematico scenario: un processore potrebbe essere violato da hackers che controllano il suo comportamento in

modo arbitrario. In questa ipotesi il comportamento può disturbare la computazione in maniera intelligente per scopi fraudolenti. In altre parole, da un guasto bizantino ci si deve aspettare il peggio.

Prima di procedere dobbiamo modificare leggermente le proprietà di accordo e validità richieste dal problema del consenso.

**Problema** ConsensoGB**:** Consideriamo un sistema distribuito di $n$ processori, che comunicano tramite scambio di messaggi su un canale affidabile. I processori possono rompersi con dei guasti bizantini. Ogni processore ha un valore di input $v_i \in V$ e i processori devono decidere un valore di output in modo tale da soddisfare le seguenti 3 proprietà:

- (Accordo) Tutti i processori non guasti danno in output lo stesso valore.
- (Validità) Se tutti i processori non guasti iniziano con uno stesso valore $v \in V$, allora $v$ è l'unico possibile output.
- (Terminazione) Tutti i processori non guasti decidono.

Le modifiche introdotte nel problema ConsensoGB riguardano il fatto che le proprietà di accordo e validità devono essere soddisfatte solo dai processori non guasti: poichè i guasti sono bizantini, non c'è modo di "forzare" un processore guasto a fare delle scelte né tantomeno a fargli rivelare il vero valore di input.

**Algoritmo** EIGByz**.** L'algoritmo EIGStop che abbiamo visto in precedenza può essere adattato al caso di guasti bizantini nel seguente modo. I processori propagano i valori inziali per $f+1$ rounds come in EIGStop. Messaggi non conformi vengono semplicemente ignorati e il destinatario si comporta come se il messaggio non fosse mai arrivato. Alla fine degli $f+1$ rounds i valori dell'albero EIG che sono nulli vengono rimpiazzati dal valore di default $v_0$. Per prendere una decisione, ogni processo visita il proprio albero EIG partendo dalle foglie per arrivare alla radice e durante la visita calcola un nuovo valore, *newval*, per ogni nodo dell'albero come segue. Per ogni foglia con etichetta $x$, $newval(x) := val(x)$. Per ogni nodo interno con etichetta $x$, $newval(x)$ è definito come il *newval* di una maggioranza stretta dei figli del nodo $x$, cioè si esaminano i valori *newval* dei figli di $x$, e se c'è un valore $v$ presente in una maggioranza stretta si pone $newval(x) := v$. Se non esiste un valore in una maggioranza stretta, allora si pone $newval(x) := v_0$.

La Figura 6.7 mostra un esempio. I valori *newval* delle foglie sono uguali ai valori *val* calcolati da EIGStop quando non sono $\perp$; i valori $\perp$ vengono sostituiti da $v_0$. Per i nodi interni invece vengono calcolati con la regola descritta sopra. Ad esempio per il nodo $x_1$ ci sono 4 figli con $newval = 1$ e 3 figli con $newval = 0$, quindi $newval(x_1) = 1$. Per i nodi $x_2$ e $x_3$ invece c'è una maggioranza di figli con $newval = 0$ quindi si ha che $newval(x_2) = 0$ e $newval(x_3) = 0$. Procedendo verso la radice, il nodo $y$ ha una maggioranza di figli con $newval = 0$ quindi si ha che $newval(y) = 0$. La decisione finale è $newval(\lambda)$, cioè il *newval* della radice dell'albero EIG.

L'algoritmo EIGByz funziona se $n > 3f$. La prova di correttezza dell'algoritmo EIGByz si basa sul fatto che, poichè $n > 3f$, $f$ processori guasti non possono produrre

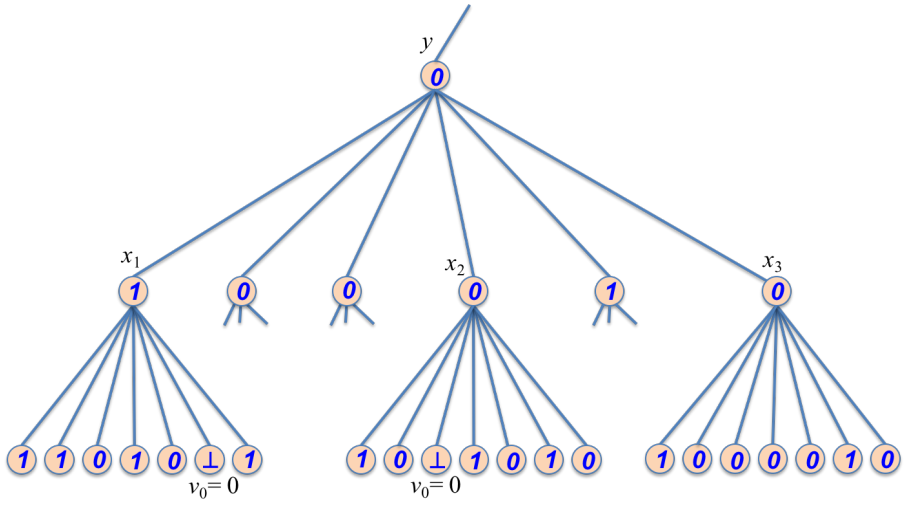



Figura 6.7: Valori *newval* in EIGBYZ.

troppi valori discordanti e quindi le informazioni diffuse dagli $n - f$ processori non guasti sono sufficienti a prendere la decisione corretta (dettagli sulla prova di correttezza possono essere trovati in L96 [26]). L'algoritmo EIGBYZ ha la stessa complessità di EIGSTOP: $f + 1$ rounds, $O((f + 1)n^2)$ messaggi e $O(n^{f+1}b)$ bit di comunicazione.

### 6.4.3 *Limite inferiore $n > 3f$.*

Abbiamo visto che l'algoritmo EIGBYZ risolve il problema del consenso per guasti bizantini. Esistono anche altri algoritmi come l'algoritmo TURPINCOAN, che funziona per il caso in cui l'insieme di valori iniziali è $V = \{0, 1\}$. Un altro algoritmo usa l'algoritmo TURPINCOAN come subroutine per risolvere il problema per un insieme di valori iniziali arbitrario (si veda il Capitolo 6 di L96 [26] per ulteriori dettagli su questi algoritmi). Tutti questi algoritmi hanno una caratteristica in comune: richiedono che il numero di processori $n$ del sistema sia strettamente più grande del triplo del numero $f$ di possibili guasti, cioè deve essere $n > 3f$.

Non è un caso. Infatti per $n \leq 3f$ il problema non può essere risolto. Intuitivamente questo significa che $f$ guasti bizantini, quindi $f$ processori che si comportano in modo da disturbare la computazione possono "imbrogliare" fino a $2f$ processori che funzionano bene. Prima di presentare la prova, diamo un esempio che ci dà una intuzione della limitazione.

Consideriamo il caso particolore di un sistema con $n = 3$ processori e $f = 1$. Supponiamo, per semplicità, che i processori decidano dopo 2 round e che nel primo round ogni processore spedisca il proprio valore iniziale mentre nel secondo round ogni processore spedisce il valore ricevuto nel primo round, sepcificando anche da chi lo ha ricevuto. Queste limitazioni non sono "forti" in quanto i processori si stanno scambiando tutte le informazioni a loro disposizione e per fare ciò bastano due round (in ogni caso con questo esempio stiamo solo dando un'intuizione del perché il problema non può essere risolto quando $n \leq 3f$). Consideriamo le seguenti esecuzioni:

1. Esecuzione $\alpha_1$. I valori iniziali dei 3 processori $p_1, p_2$ e $p_3$ sono, rispettivamente, 1, 1 e 0. I processori $p_1$ e $p_2$ funzionano correttamente mentre $p_3$ è guasto. Nel primo round tutti i processori si comportano seguendo l'algoritmo e quindi spediscono il proprio valore iniziale. Nel secondo round i processori non guasti, $p_1$ e $p_2$, spediscono correttamente ciò che hanno appreso nel primo round, mentre il processore guasto, $p_3$ si comporta in modo fraudolento e spedisce un messaggio corretto a $p_1$ ed un messaggio *falso* a $p_1$. La Figura 6.8 mostra i messaggi spediti. Per la proprietà di validità i processori corretti devono decidere 1.

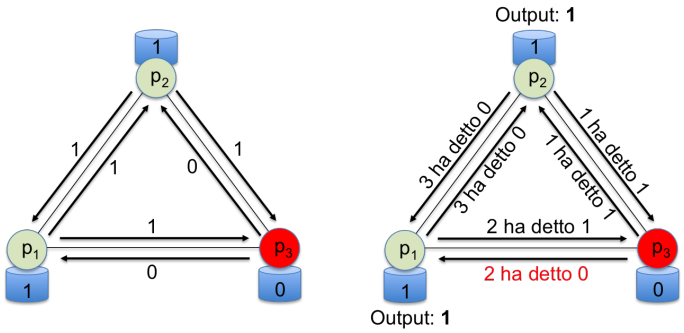



Figura 6.8: Esecuzione $\alpha_1$

2. Esecuzione $\alpha_2$. Questa esecuzione è simile all'esecuzione $\alpha_1$: questa volta è il processore $p_1$ ad essere guasto e i valori iniziali sono 0 per $p_2$ e $p_3$ e 1 per $p_1$. Il processore guasto si comporta come nel caso precedente: nel primo round spedisce correttamente il proprio valore iniziale mentre nel secondo round manda un messaggio veritiero a $p_2$ ed un messaggio falso a $p_3$. La Figura 6.9 mostra i messaggi spediti. Per la proprietà di validità i processori corretti quest volta devono decidere 0.

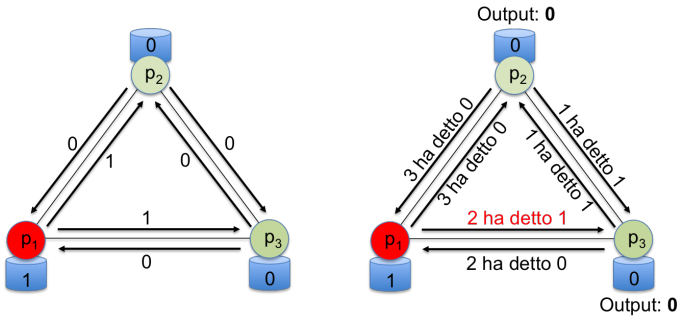



Figura 6.9: Esecuzione $\alpha_2$

3. Esecuzione $\alpha_3$. Consideriamo infine una terza esecuzione $\alpha_3$ in cui il processore $p_2$ è guasto, mentre $p_1$ e $p_3$ funzionano correttamente. I valori iniziali di $p_1$ e $p_3$ sono rispettivamente 1 e 0. Questa volta il processore guasto spedisce messaggi contradditori nel primo round (a $p_1$ invia 0 e a $p_3$ invia 1) e messaggi veritieri nel secondo round. La Figura 6.10 mostra i messaggi spediti.

Le esecuzioni $\alpha_1$ e $\alpha_3$ sono indistinguibili per il processore $p_1$: esso infatti riceve e spedisce gli stessi messaggi nelle due esecuzioni. Poichè $p_1$ decide 1 in $\alpha_1$, concludiamo che deve decidere 1 anche in $\alpha_3$.

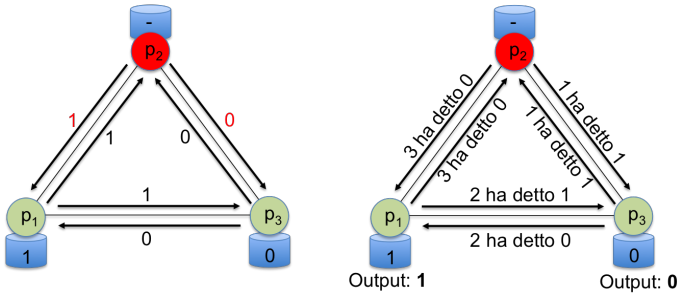



Figura 6.10: Esecuzione $\alpha_3$

Analogamente, le esecuzioni $\alpha_2$ e $\alpha_3$ sono indistinguibili per il processore $p_3$: esso infatti riceve e spedisce gli stessi messaggi nelle due esecuzioni. Poichè $p_3$ decide 0 in $\alpha_0$, concludiamo che deve decidere 0 anche in $\alpha_3$.

Questo significa che la proprietà di accordo è violata nell'esecuzione $\alpha_3$.

L'esempio che abbiamo fornito non costituisce una prova della limitazione $n > 3f$, ma fornisce solo l'intuizione del perché tale limitazione è necessaria. Non è una prova in quanto abbiamo stabilito uno specifico tipo di algoritmo: utilizza solo 2 round e manda uno specifico pattern di messaggi. Quindi, da quanto detto, possiamo concludere solo che non esistono algoritmi di quel tipo.

Il seguente lemma generalizza l'argomentazione usata nell'esempio fornendo quindi una prova del fatto che con soli 3 processori anche un solo guasto rendo il problema non risolvibile.

**Lemma 6.4.3** *Tre processori non possono risolvere il problema del consenso se uno di essi può comportarsi in modo bizantino.*

DIMOSTRAZIONE. Per contraddizione assumiamo che esista un algortimo A che risolve il problema in un sistema con tre processori, 1, 2 e 3 nell'ipotesi che uno di essi possa essere guasto in modo bizantino. Nella prova useremo 6 processori che verranno utilizzati in gruppi di 3 per poter far girare l'algoritmo $A$. Consideriamo l'esecuzione dell'algoritmo $A$ nei seguenti 3 casi:

1. Processori $p_1, p_2, p_3$ con $p_1$ guasto e input di $p_2$ e $p_3$ pari a 0. Chiamiamo $\alpha_1$ l'esecuzione che ne deriva.
2. Processori $p'_1, p'_2, p'_3$ con $p'_3$ guasto e input di $p'_1$ e $p'_2$ pari a 1. Chiamiamo $\alpha_2$ l'esecuzione che ne deriva.
3. Processori $p'_1, p_2, p_3$ con $p_2$ guasto, input di $p'_1$ pari a 1 e input di $p_3$ pari a 0. Chiamiamo $\alpha_2$ l'esecuzione che ne deriva.

Poichè l'algoritmo $A$ risolve il problema anche se un processore è guasto, si ha che in $\alpha_1$, per la condizione di validità, i processori $p_2$ e $p_3$ decidono 0 e in $\alpha_2$, sempre per la condizione di validità, i processori $p'_1$ e $p'_2$ decidono 1.

Nell'esecuzione $\alpha_3$ non sappiamo quale valore viene deciso in quanto i processori partono da input diversi quindi non possiamo sfruttare la condizione di validità. Tuttavia per la proprietà di accordo devono decidere o entrambi 0 o entrambi 1; mostreremo ora che questo non è possibile.

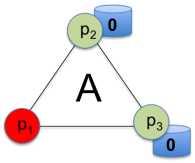



Figura 6.11: Esecuzione $\alpha_1$

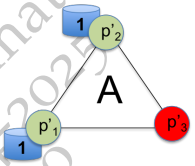



Figura 6.12: Esecuzione $\alpha_2$

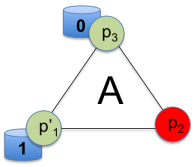



Figura 6.13: Esecuzione $\alpha_3$

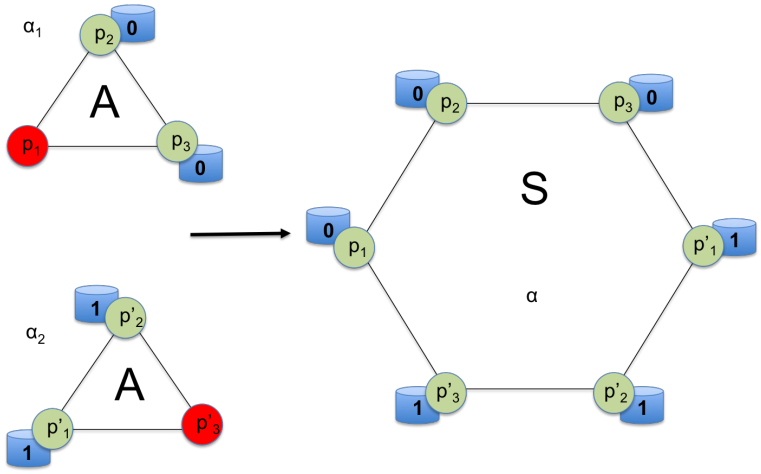



Figura 6.14: Sistema $S$ per il Lemma 6.4.3

Costruiamo un sistema $S$ usando due "copie" di $A$, come mostrato nella Figura 6.14. Nel sistema $S$ i processori si comportano seguendo l'algoritmo $A$ ma i processori $p_1$ e $p_3$ comunicheranno con gli omologhi dell'altra copia: l'algoritmo deve continuare a funzionare in quanto abbiamo supposto che possa tollerare un guasto bizantino e nel sistema $S$ è come se ognuna delle due copie di $A$ si ritrovi in una situazione reale in cui un processore ($p_1$ per una copia e $p_3$ per l'altra) sia guasto.

Osservazione: cosa è $S$? Non è un sistema dove risolviamo il problema del consenso usando l'algoritmo $A$; infatti $A$ risolve il problema del consensi in un sistema con 3 processori non con 6. $S$ è semplicemente un sistema distribuito sincrono con 6 processori in cui ogni processore si comporta seguendo l'algoritmo $A$ nel modo descritto in precedenza. Il sistema $S$ non risolve il problema del consenso! Cosa fa il sistema $S$ non ci interessa nemmeno. Assumeremo che in $S$ non si verifichino guasti e chiameremo $\alpha$ l'esecuzione dell'algoritmo $A$ in $S$.

Per i processori $p_2$ e $p_3$ l'esecuzione $\alpha_1$ è indistinguibile da $\alpha$, mentre per i processori $p'_1$ e $p'_2$ l'esecuzione $\alpha_2$ è indistingubile da $\alpha$. Poichè in $\alpha_1$ i processori $p_2$ e $p_3$ decidono 0 anche in $\alpha$ $p_2$ e $p_3$ decidono 0. Analogamente, poichè in $\alpha_2$ i processori $p'_1$ e $p'_2$ decidono 1 anche in $\alpha$ $p'_1$ e $p'_2$ decidono 1.

Infine consideriamo le esecuzioni $\alpha$ e $\alpha_3$ dal punto di vista dei processori $p'_1$ e $p_3$. Per tali processori $\alpha$ e $\alpha_3$ sono indistinguibili e poichè in $\alpha$ decidono rispettivamente 0 e 1 si ha che anche in $\alpha_3$ i processori $p'_1$ e $p_3$ decidono rispettivamente 0 e 1. Questo significa che l'algoritmo $A$ non risolve il problema del consenso nell'esecuziuone $\alpha_3$. □

Il lemma precedente mostra che non è possibile risolvere il problema se $f \geq n/3$ nel caso specifico di $n = 3$. Ovviamente questo lascerebbe la possibilità che per valori maggiori di $n$ la situazioni cambi. Il seguente teorema mostra che il risultato è valido per ogni $n$.

**Teorema 6.4.4** *Il problema del consenso in un sistema di n processori non può essere risolto se si possono verificare f guasti bizantini con* $n \leq 3f$.

Dimostrazione. Il caso $n = 2$ è semplice in quanto, informalmente, ogni processore

deve decidere da solo perché l'altro protrebbe essere guasto. Più formalmente potremmo procedere per assurdo assumendo che esista un algoritmo $A$. Consideriamo l'esecuzione di $A$ quando $p_1$ e $p_2$ inziano con rispettivamente 0 e 1 e nessuno dei due è guasto. Chiamiamo $\alpha$ tale esecuzione. In $\alpha$ $A$ dovrà far prendere una decisione; senza perdere in generalità assumimamo che sia 0; se fosse 1 si potremmo precedere in modo simmetrico (e la prova potrebbe essere estesa anche al caso non binario). Ora consideriamo una nuova esecuzione $\alpha'$ di $A$ in cui $p_1$ è guasto e $p_2$ funziona correttamente e ha come input 1. Il processore bizantino $p_1$ si comporta esattamente come in $\alpha$. Dunque $p_2$ non vede nessuna differenza fra $\alpha$ e $\alpha'$ pertanto, visto che in $\alpha$ ha deciso 0 farà lo stesso in $\alpha'$ violando la proprietà di validità. Pertanto l'algoritmo $A$ non può esistere e quindi il problema non può essere risolto.

Consideriamo ora il caso $3 \leq n \leq 3f$. Anche in questo caso procediamo per contraddizione assumendo che esista un algoritmo $A$ che risolve il problema. Mostreremo come trasformare $A$ in un algoritmo $B$ per un sistema con 3 processori $b_1, b_2$ e $b_3$ in cui uno può essere guasto. Ognuno dei 3 processori usati in $B$ simulerà circa un terzo dei processori usati in $A$. Cioè partizioniamo gli $n$ processori in 3 insiemi $I_1, I_2$ e $I_3$ ognuno di grandezza al più $f$. Il processore $b_i$ simulerà l'insieme $I_i$ nel modo seguente: il processore $b_i$ tiene traccia degli stati di tutti i processori in $I_i$, simulando tutti i passi e le spedizioni dei messaggi dei processori in $I_i$. I messaggi saranno spediti al processore che simula il destinatario. Se uno qualsiasi dei processori in $I_i$ decide un valore $v$ allora anche $b_i$ decide $v$ (se ci sono più valori allora $b_i$ può decidere uno qualsiasi di tali valori).

L'algoritmo $B$ risolve il problema del consenso bizantino in un sistema con 3 processori. Consideriamo infatti un'esecuzione $\alpha$ in cui al massimo un processore $b_i$ è guasto e sia $\alpha'$ la corrispondente esecuzione di $A$ nel sistema simulato. Poichè $b_i$ simula al massimo $f$ processori, si ha che in $\alpha'$ ci sono al massimo $f$ guasti. Poichè $A$ risolve il problema del consenso bizantino con al massimo $f$ guasti, si ha che in $\alpha'$ vengono soddisfate le proprietà di accordo, validità e terminazione. Le stesse valgono anche in $\alpha$. Infatti sia $b_i$ un processore non guasto pertanto $b_i$ simula i processori $I_i$ che essendo non guasti, decideranno; quindi anche $b_i$ decide.

Per la proprietà di validità, assumiamo che tutti i processori non guasti di $B$ abbiano come input lo stesso valore $v$. Ovviamente anche i processori di $A$ iniziano con lo stesso valore $v$. Pertanto per la validità in $\alpha'$ si ha che tutti processori non guasti di $A$ decidono $v$. Questo significa anche che tutti i processori non guasti di $B$ decidono $v$.

Infine per la proprietà di accordo, siano $b_i$ e $b_j$ due processori non guasti di $B$. Tali processori simulano solo processori non guasti di $A$. Questo significa che i processori in $I_i$ e in $I_j$ decidono lo stesso valore. Quindi anche $b_i$ e $b_j$ decidono lo stesso valore.

Pertanto siamo riusciti a costruire un algoritmo che risolve il problema del consenso nel caso di 3 processori ed un guasto bizantino. Ma questo contraddice il Lemma 6.4.3.
□

## 6.5 *Consenso in sistemi asincroni con guasti stop*

Consideriamo adesso il problema del consenso in sistemi asincroni. In un sistema asincrono i processori possono avere velocità diverse e, almeno dal punto di vista teorico,

estremamente diverse al punto che diventa difficile, anzi impossibile, distinguere un processore molto lento da un processore guasto. Analogamente anche i tempi di consegna dei messaggi sono variabili e non possiamo sapere se un messaggio è stato perso oppure è ancora in transito. In un sistema asincrono, in un certo senso, il tempo non esiste, o più precisamente non può essere misurato. In realtà, ogni componente del sistema può misurare il tempo, ma lo fa in modo indipendente e quindi ogni componente misura il tempo in modo diverso. L'assunzione è che queste diversità possono essere enormi, fino al caso limite di infinito che corrisponde a componenti guaste. Ovviamente questo crea una grossa difficoltà in quanto gli algoritmi dovranno rinunciare ad usare informazioni relative alla misura del tempo. Si noti come questo sia in contrapposizione a ciò che succede in un sistema sincrono, nel quale è come se esistesse una misurazione del tempo uguale per tutte le componenti (processori e canali) e ogni conponente fosse in grado di eseguire delle istruzioni in perfetta sincronia, cioè nello stesso istante, con tutte le altre componenti del sistema.

Consideriamo guasti di tipo stop (porblema ConsensoGS). Ricordiamo la definizione del problema. *Accordo*: in una qualsiasi esecuzione, tutti le decisioni sono uguali. *Validità*: in una qualsiasi esecuzione, se tutti i valori iniziali sono uguali a $v$, allora $v$ è l'unico possibile valore per le decisioni. *Terminazione*: in una qualsiasi esecuzione in cui si possono verificare al massimo $f$ guasti stop, tutti i processori non guasti decidono.

### 6.5.1 *Impossibilità per algoritmi deterministici*

Il seguente risultato è una delle pietre miliari nella teoria dei sistemi distribuiti, ed è noto come teorema dell'impossibilità FLP, dai nomi[2] dei ricercatori che lo hanno dimostrato.

[2] Fischer, Lynch, Patterson, 1985 [13].

**Teorema 6.5.1** *Non è possibile risolvere il problema del consenso in un sistema distribuito asincrono anche in presenza della possibilità di un solo guasto.*

Prima di dimostrare il teorema, specifichiamo meglio il formalismo che utilizzeremo.

Assumiamo che $V = \{0, 1\}$. Consideriamo un sistema con $n$ processori, $p_1, p_2, \ldots, p_n$. Ogni processore inizia la computazione con un bit di input e deve produrre un bit di output in accordo alle 3 proprietà della definzione del problema del consenso.

I processori comunicano inviando dei messaggi. Possiamo modellare la comunicazione con un buffer globale e affidabile: i messaggi non vengono persi, anche se ogni messaggio può impiegare un tempo arbitrario per arrivare alla destinazione. Il buffer quindi è semplicemente un multinsieme di coppie

$$\mathit{buffer} = \{(p, m) |\ m \text{ è un messaggio per } p\}$$

dove $m$ è un messaggio e $p$ è un processore, quello che riceverà il messaggio. L'invio e la ricezione dei messaggio sono modellati da due operazioni:

- $\mathit{send}(m, p)$: eseguita da un processore $q$, indica la spedizione del messaggio $m$ destinato a $p$. L'effetto è quello di inserire l'elemento $(m, p)$ nel *buffer*.

- *receive*$(p)$: eseguita da un processore $p$, indica la ricezione di un messaggio da parte di $p$. L'effetto è quello di cancellare un elemento arbitrario $(m, p)$ dal *buffer* e di consegnare $m$ a $p$. Come caso particolare questa operazione può restituire il valore speciale `null` che indica l'assenza di un messaggio per $p$. Questo modella l'asincronia del sistema: anche se $p$ chiede di ricevere un messaggio, il messaggio potrebbe non esistere, oppure essere ancora "in transito".

Una *configurazione* del sistema è lo stato di ogni processore più il contenuto del *buffer*. Nella configurazione iniziale, il *buffer* è vuoto e ogni processore ha il bit iniziale settato.

La computazione procede in *passi* che portano il sistema da una configurazione a quella successiva. Ogni passo viene eseguito da un processore $p$ che può utilizzare l'operazione *receive*$(p)$ per ricevere un messaggio $m \in M \cup \{\texttt{null}\}$, eseguire una computazione locale cambiando il proprio stato ed eventualmente spedire con l'operazione *send*$(m, q)$ un numero finito di messaggi agli altri processori.

Un *evento* $e = (m, p)$ rappresenta la ricezione del messaggio $m$ da parte di $p$. Una *esecuzione parziale* è una sequenza $\sigma$, finita o infinita, di eventi a partire da un configurazione $c$. Indicheremo con $\sigma(c)$ la configurazione risultante e diremo che $\sigma$ è *applicabile* a $c$. Una *esecuzione* è una esecuzione parziale che inizia da una configurazione iniziale.

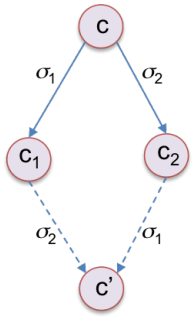


Figura 6.15: Lemma 6.5.2

**Lemma 6.5.2** *Consideriamo una configurazione $c$ e le esecuzioni parziali $\sigma_1$ e $\sigma_2$ applicabili a $c$. Sia $c_1 = \sigma_1(c)$ e $c_2 = \sigma_2(c)$. Se l'insieme dei processori che eseguono passi in $\sigma_1$ e $\sigma_2$ sono disgiunti, allora $\sigma_2(c_1) = \sigma_1(c_2)$.*

DIMOSTRAZIONE. Il lemma dice che applicando entrambe le esecuzioni parziali si arriva sempre alla stessa configurazione indipentemente dall'ordine in cui le si applica. (Figura 6.15). Per provarlo, notiamo che poichè sia $\sigma_1$ che $\sigma_2$ sono entrambe esecuzioni parziali applicabili in modo indipendente a $c$ e l'insieme dei processori che eseguono passi in $\sigma_1$ che $\sigma_2$ sono disgiunti, non è possibile "spedire" un messaggio in un passo di $\sigma_1$ e riceverlo in un passo di $\sigma_2$ o viceversa.

Dunque, poichè gli insiemi di processori sono disgiunti, si ha che i passi applicati nelle due esecuzioni non interagiscono fra loro.

Questo significa che in $c_1$ è possibile applicare $\sigma_2$ e in $c_2$ è possibile applicare $\sigma_1$; inoltre, poichè gli insiemi di processori sono disgiunti, eseguire prima $\sigma_1$ e poi $\sigma_2$ non crea nessuna differenza rispetto a eseguire prima $\sigma_2$ e poi $\sigma_1$ (l'ordine è rilevante solo se uno stesso processore esegue passi sia in $\sigma_1$ che $\sigma_2$). □

Diremo che una configurazione è 0-*valente* se da quella configurazione tutte le possibili esecuzioni parziali portano a un risultato finale di 0. Analogamente diremo che una configurazione è 1-*valente* se da quella configurazione tutte le possibili esecuzioni parziali portano a un risultato finale di 1. Inoltre diremo che una configurazione è *univalente* se è 0-valente oppure 1-valente. Una configurazione *bivalente* è una configurazione che può portare sia a una decisione di 0 sia a una decisione di 1. Si noti che quando si raggiunge una configurazione univalente, di fatto si è raggiunto la decisione finale, quindi in pratica l'algoritmo, almeno nella sua sostanza, è terminato. Da una configurazione bivalente invece sono necessari altri passi per poter raggiungere una decisione.

Per provare che il problema non può essere risolto, assumeremo che, per assurdo, esiste un algoritmo $A$ che risolve il problema. Mostreremo che esistono dei casi in cui $A$ non termina mai. Per fare ciò mostreremo che esiste una configurazione iniziale bivalente e che da tale configurazione è possibile eseguire passi senza mai raggiungere una configurazione univalente.

**Lemma 6.5.3** *Esiste una configurazione iniziale bivalente.*

Dimostrazione. Assumiamo per assurdo che tutte le configurazioni iniziali siano univalenti. Consideriamo le seguenti configurazioni iniziali $c_i$, per $i = 0, 1, 2, \ldots, n$. Nella configurazione $c_0$ tutti gli input sono 0. Per la condizione di validità $c_0$ deve essere 0-valente. La configurazione $c_i$ differisce da $c_{i-1}$ solo per l'input del processore $i$-esimo: tale input è 0 in $c_{i-1}$ e 1 in $c_i$. Pertanto $c_n$ è una configurazione inizale in cui tutti gli input sono 1. Per la condizione di validità $c_n$ deve essere 1-valente.

Poichè $c_0$ è 0-valente e $c_n$ è 1-valente, deve esistere un indice $i \in [0, n-1]$ tale che $c_i$ è 0-valente e $c_{i+1}$ è 1-valente.

Consideriamo l'esecuzione $\sigma_i$ che inizia da $c_i$ nella quale il processore $p_{i+1}$ si guasta immmediatamente senza eseguire nessun passo. L'input di $p_{i+1}$, dunque, non potrà essere noto a nessuno.

Consideriamo anche l'eseuzione $\sigma_{i+1}$ che inizia da $c_{i+1}$ nella quale il processore $p_{i+1}$ si guasta immmediatamente senza eseguire nessun passo. Anche in questo caso l'input di $p_{i+1}$ non potrà essere noto a nessuno. Tale bit è però l'unica differenza fra $\sigma_i$ e $\sigma_{i+1}$; il fatto che nessuno lo conosca significa che le due esecuzioni sono indistinguibile per tutti i processori (tranne che per $p_{i+1}$, che però è guasto).

Pertanto tutti i processori non guasti decideranno lo stesso valore sia $\sigma_i$ che in $\sigma_{i+1}$. Ma questo non è possibile in quanto $\sigma_i$ è 0-valente e $\sigma_{i+1}$ è 1-valente. □

**Lemma 6.5.4** *Consideriamo una configurazione $c$ raggiungibile e bivalente e sia $e = (p, m)$ un evento applicabile in $c$. Sia $C$ l'insieme di configurazioni raggiungibili da $c$ senza applicare $e$. Sia $D$ l'insieme di configurazioni raggiungibili da una qualsisi configurazione di $C$ applicando $e$ (si veda la Figura 6.16). L'insieme $D$ contiene una configurazione bivalente.*

Dimostrazione. Procediamo per assurdo assumendo che $D$ non contenga nessuna configurazione bivalente, quindi che $D$ contenga solo configurazioni univalenti.

Osserviamo che se l'insieme $C$ non cotiene altre configurazioni oltre a $c$, si avrebbe che l'unic evento applicabile a $c$ è l'evento $e$. Ma poichè possiamo far rompere il processore $p$ che è l'unico che può applicare $e$, $c$, avremmo trovato una configurazione che non permette di proseguire l'esecuzione. Essendo $c$ bivalente si avrebbe un'esecuzione in cui non si decide. Per cui, in questo, caso il problema non è risolvibile. Pertanto consideriamo il caso in cui oltre a $e$ ci sono altri eventi $e' \neq e$ applicabili a $C$.

Poichè $c$ è bivalente, esistono due configurazioni $\overline{c_0}$ e $\overline{c_1}$, raggiungibili da $c$, rispettivamente 0-valente e 1-valente. Da ciò segue anche che esistono due configurazioni $c_0 \in C$ e $d_0 = e(c_0)$ con $d_0$ 0-valente. Infatti se $\overline{c_0} \in C$ allora basta prendere $c_0 = \overline{c_0}$ e si ottiene il risultato voluto. Consideriamo il caso $\overline{c_0} \notin C$. Poichè siamo arrivati a $\overline{c_0}$ partendo da $c$, visto che $\overline{c_0}$ non è in $C$, abbiamo usato $e$ per raggiungere $\overline{c_0}$. Questo implica che siamo passati per una configurazione $c_0 \in C$ tale che $d_0 = e(c_0) \in D$. Ma

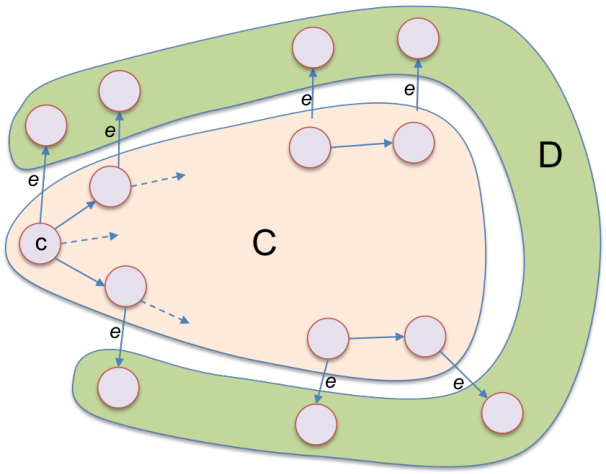



Figura 6.16: Lemma 6.5.4. Le esecuzioni da $c$ a $C$ non contengono $e = (p, m)$.

sappiamo anche che $D$ contiene solo configurazioni univalenti; pertanto poichè da $d_0$ raggiungiamo $\overline{c_0}$ che è 0-valente, anche $d_0$ deve essere 0-valente.

In modo analogo possiamo provare che esistono due configurazioni $c_1 \in C$ e $d_1 = e(c_1)$ con $d_1$ 1-valente.

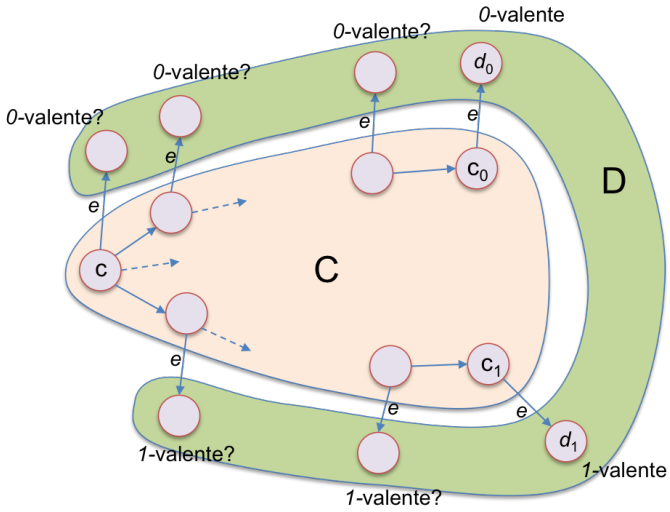



Figura 6.17: Configurazioni $c_0, d_0, c_1$ e $d_1$.

Dunque, per ipotesi (assurda) $D$ contiene solo configurazioni univalenti e abbiamo visto che non possono essere tutte dello stesso tipo, cioè $D$ contiene sia configurazioni 0-valenti che configurazioni 1-valenti. Il prossimo passo è quello di mostrare che esistono due configurazioni di $C$, le chiameremo $C_0$ e $C_1$, con $C_1 = e'(C_0)$, dove $e' = (p', m') \neq e$, e tali che $D_0 = e(C_0)$ e $D_1 = e(C_1)$ sono, rispettivamente, 0-valente e 1-valente[3]. Per provare tale affermazione sfruttiamo il fatto che $D$ contiene sia una configurazione $d_0$ 0-valente sia una configurazione $d_1$ 1-valente. Entrambe $d_0$ e $d_1$ sono raggiungibili da $c$, come mostrato nella Figura 6.17. Chiamiamo $\sigma_0$ il cammino da $c$ a $c_1$ e $\sigma_1$ il cammino da $c$ a $c_1$.

Partendo dal fatto che $d_1$ è 1-valente, e considerando il cammino da $c$ a $c_1$ fatto da

[3] Qui il ruolo di 0 e 1 potrebbe essere invertito. Il punto chiave è che si passa da una configurazione univalente di un valore (0 o 1) a un configurazione univalente del valore negato.

configurazioni successive in $C$, ci chiediamo se il nodo di $D$ corrispondente al nodo di $C$ che precede $c_1$ è 0-valente oppure 1-valente. Se è 0-valente abbiamo trovato la coppia di nodi $C_0$ e $C_1$. Se è 1-valente ripetiamo il procedimento con il nodo procedente nel cammino da $c$ a $c_1$.

Possiamo fare lo stesso, in modo simmetrico, partendo da $d_0$. Poichè andando a ritroso su $\sigma_0$ e $\sigma_1$ arriviamo in entrambi i casi a $c$, dovremo necessariamente trovare (su almeno uno fra $\sigma_1$ e a $\sigma_2$) due configurazioni $C_0$ e $C_1$ per le quali le corrispondenti configurazioni in $D$ passano da 0-valente a 1-valente (o da 1-valente a 0-valente). Senza perdere in generalità assumiamo che si passi da 0-valente a 1-valente: nel caso simmetrico si procede nello stesso modo scambiando i ruoli di 0 e 1. La Figura 6.18 mostra le configurazioni $C_0, C_1, D_0$ e $D_1$.

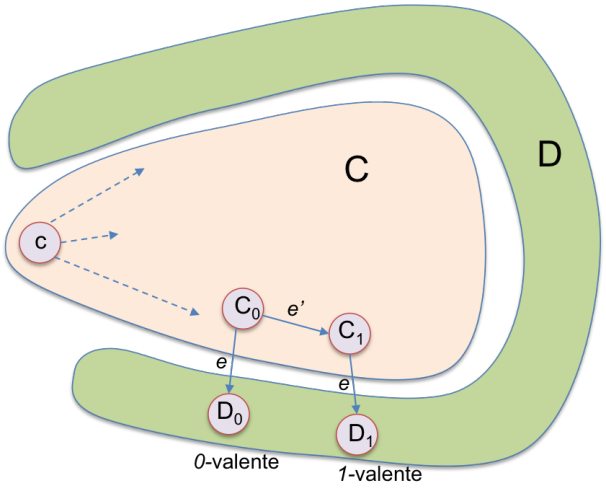



Figura 6.18: Configurazioni $C_0, D_0, C_1$ e $D_1$.

Sia $p'$ il processore coinvolto nel passo $e'$.

Se fosse $p' \neq p$, potremmo applicare $e'$ a $D_0$ e per il Lemma 6.5.2 si avrebbe che $e'(D_0) = D_1$, come mostrato nella Figura 6.19. Ma questo è un assurdo in quanto si passerebbe da una configurazione 0-valente a una configurazione 1-valente.

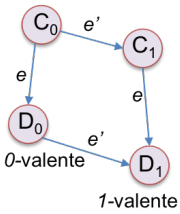



Figura 6.19: Situazione impossibile per il caso $p' \neq p$.

Pertanto deve necessariamente essere $p' = p$. Ma anche in questo caso arriviamo a un assurdo. Infatti, possiamo considerare l'esecuzione parziale $\sigma$ da $C_0$ nella quale $p$ non esegue più nessun passo e che arriva in una configurazione finale, cioè nella quale si prende una decisione, in altre parole una configurazione univalente, $f = \sigma(C_0)$. Questo è un punto cruciale della prova: qui invochiamo il fatto che l'algoritmo deve funzionare anche se un solo processore si guasta. Quindi l'esistenza dell'esecuzione $\sigma$ è garantita dal fatto che da $C_0$ dobbiamo arrivare a una decisione anche se il processore $p$ si guasta e quindi non esegue nessun passo.

Per il Lemma 6.5.2 possiamo applicare $\sigma$ anche a $D_0$ e $D_1$ e parimenti l'evento $e$ ad $f$ e anche la sequenza di eventi $(e', e)$ sempre a $f$, raggiungendo le configurazioni $E_0$ ed $E_1$, come mostrato nella Figura 6.20. Tale situazione è impossibile in quanto si avrebbe una configurazione univalente, la configurazione $f$, che può essere trasformata sia in 0-valente (configurazione $E_0$) che in 1-valente (configurazione $E_1$).

Pertanto $D$ deve necessariamente contenere una configurazione bivalente. □

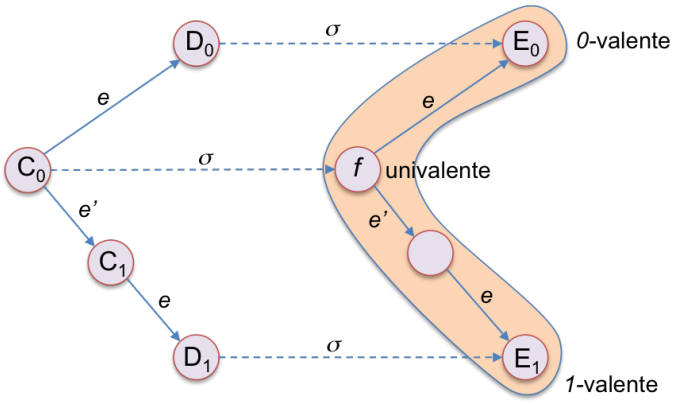



Figura 6.20: Situazione impossibile per il caso $p' = p$.

### 6.5.2 *Algoritmo randomizzato*

Poichè il problema del consenso non è risolvibile in sistemi asincroni con algoritmi deterministici, spostiamo l'attenzione su una soluzione randomizzata. Per poter utilizzare la randomizzazione rilassiamo leggermente il problema cambiando la condizione di terminazione nel seguente modo:

- (Terminazione) Tutti i processi non guasti decidono entro il tempo $t$ con probabilità almeno $1 - \epsilon(t)$,

dove $\epsilon(t)$ è una funzione che diventa sempre più piccola al crescere di $t$.

Una soluzione semplice è l'algoritmo BENOR. Tale algoritmo richiede che $n > 2f$ e $V = \{0, 1\}$.

*Algoritmo* BENOR. Ogni processore mantiene due variabili locali $x$ e $y$. Inizialmente la $x$ contiene il valore di input del processore e la $y$ è `null`. Ogni processore $p_i$ esegue una serie di *fasi* numerate $1, 2, \ldots$. Il processore continua ad eseguire l'algoritmo anche dopo aver preso la decisione finale. In ogni fase $s$, il processore $p_i$ esegue 2 rounds:

Round 1: Spedisce in broadcast il messaggio $(\mathit{first}, s, x)$. Aspetta l'arrivo di almeno $n - f$ messaggi della forma $(\mathit{first}, s, v)$. Se almeno $\lfloor \frac{n}{2} \rfloor + 1$ hanno tutti lo stesso valore $v$ allora $y := v$, altrimenti $y :=$ `null`.

Round 2: Spedisce in broadcast il messaggio $(\mathit{second}, s, y)$. Aspetta l'arrivo di almeno $n - f$ messaggi della forma $(\mathit{second}, s, v)$.

(a) Se c'è un messaggio con un valore $v \neq$ `null` allora $x := v$;

(b) Se almeno $f + 1$ messaggi contengono uno stesso valore $v$ allora il processore decide $v$

(c) Altrimenti, si sceglie uniformemente a caso 0 o 1 come valore di $x$.

Osserviamo che il passo *a* del round 2 è ben definito in quanto si può provare che i valori $y$ sono o tutti appartenenti a $\{0, \texttt{null}\}$ oppure tutti appartenenti a $\{1, \texttt{null}\}$. Quindi se esistono più messagi *second* per valori diversi da `null` essi contengono tutti lo

stesso valore. Osserviamo anche che la condizione $n > 2f$ è necessaria per assicurarsi di avere un valore di $y$ diverso da `null` in una fase in cui tutti i processori iniziano con lo stesso valore di $x$.

**Lemma 6.5.5** *Per qualsiasi fase, si ha che $y_i \in \{0, \texttt{null}\}$ per tutti i processori $i$, oppure che $y_i \in \{1, \texttt{null}\}$.*

Dimostrazione. Affinchè un processore $i$ esegua $y := v$ è necessario che il processore riceva un messaggio *first* per $v$ da più di $n/2$ processori. Poichè i processori sono $n$ non possono esserci due valori per i quali ciò succede. □

**Lemma 6.5.6** *In una fase in cui tutti i processori $i$ non guasti iniziano con lo stesso valore $x_i = v$, tutti i processori non guasti prenderanno una decisione.*

Dimostrazione. Poichè $n > 2f$ ci sono almeno $n/2 + 1$ processori non guasti che spediranno il valore $x_i = v$ nei messaggi *first*. Quindi ogni processore non guasto riceverà almeno $n/2 + 1$ messaggi e quindi avrà $y_i = v$. Nel secondo round tutti i processori non guasti spediranno tale valore nei messaggi *second* e poichè $n/2 + 1 \geq f + 1$ tutti i processori non guasti riceveranno almeno $f + 1$ messaggi e quindi decideranno $v$. □

**Lemma 6.5.7** *L'algoritmo* BenOr *soddisfa la proprietà di validità.*

Dimostrazione. La prorietà di validita richiede che se tutti i valori iniziali sono $v$ allora $v$ deve essere la decisione. Quindi assumiamo che tutti i valori di input siano uguali a $v$. Quindi tutti i processori avranno $x = v$; pertanto i messaggi *first* avranno tutti il valore $v$ come terza componente. Di conseguenza l'unico valore possibile per i messagi *second* è $v$. Dunque l'unico valore possibile per un processo che decide è $v$. □

**Lemma 6.5.8** *L'algoritmo* BenOr *soddisfa la proprietà di accordo.*

Dimostrazione. La proprietà di accordo richiede che non ci siano due decisioni diverse. Sia $p_i$ il primo processore che decide e sia $s$ la fase nella quale il processore decide e $v$ il valore. Poichè $p_i$ è il primo processore che decide non ci sono altri processori che decidono in una fase $s'$, $s' < s$. Poichè $p_i$ decide nella fase $s$, dal codice dell'algoritmo si ha che $p_i$ riceve nella fase $s$ almeno $f + 1$ messaggi della forma $(\mathit{second}, s, v)$. Questo implica che un qualsiasi processore $p_j$, $j \neq i$, che completa la fase $s$ riceve almeno 1 di questi messaggi, visto che $p_j$ riceverà un messaggio da tutti i processori che hanno spedito un messaggio a $p_i$ con al massimo l'eccezione dei processori che si guastano che sono al più $f$. Dal Lemma 6.5.5 sappiamo che $p_i$ riceverà messaggi *second* solo per questo valore e quindi non può decidere su un altro valore.

Inoltre $p_j$, avendo ricevuto almeno un messaggio *second* per $v$, imposterà il valore della variabile $x$ a $v$. Questo è vero per tutti i processori che completano la fase $s$. Pertanto tutti i propcessori che iniziano la fase $s + 1$ (e anche quelle successive) avranno $x = v$ e possiamo concludere, con una giustificazione simile a quella del lemma precedente, che l'unico valore sul quale i processori possono decidere è $v$. □

**Lemma 6.5.9** *L'algoritmo* BenOr *soddisfa la proprietà di terminazione: tutti i processori non guasti decidono nelle prime $s+1$ fasi con probabilità almeno $1-\left(1-\frac{1}{2^n}\right)^s$.*

Dimostrazione. Per $s=0$ la prova è banale vista che la probabilità con cui i processori devono decidere diventa 0. Consideriamo quindi il caso $s \geq 1$.

Dal Lemma 6.5.6 si ha che che se tutti i processori $i$ iniziano la fase $s$ con lo stesso valore di $x_i = v$, allora la fase avrà successo e i processori prenderanno una decisione. Quindi ragioniamo su come vengono scelti i valori per la fase $s$. Tali valori vengono stabiliti nel round 2 della fase $s-1$. Dal codice dell'algoritmo si ha che il valore di $x$ può essere stabilito in due modi: o copiando il valore di un messaggio di tipo *second*, oppure scegliendo un valore casuale.

Per il Lemma 6.5.5 si ha che i valori spediti nei messaggio di tipo *second* non sono mai discordanti, pertanto nel round 2 della fase $s-1$ alcuni valori $x_i$ saranno stabiliti in base ai messaggi di tipo *second* e saranno o tutti 0 o tutti 1 e i restanti $x_i$ saranno stabiliti a caso. Si noti che sono possibili anche i casi estremi in cui tutti i valori $x_i$ sono stabiliti a caso oppure sono tutti stabiliti in base al valore dei messaggi di tipo *second*. In ogni caso c'è la possibilità che siano tutti uguali.

Quale è la probabilità che siano tutti uguali? La probabilità di tale evento dipende da quanti valori vengono scelti a caso: più sono i valori scelti a caso e minore sarà tale probabilità. Quindi il caso "peggiore" è quello in cui tutti i valori sono scelti a caso e in tale caso la probabilità che tutti i valori siano uguali è $2/2^n = 1/2^{n-1}$.

Dunque la probabilità di terminare l'algoritmo nella fase $s$ è almeno $1/2^{n-1}$, pertanto la probabilità che *non* venga raggiunta la decisione nella fase $s$ è al massimo $\left(1-\frac{1}{2^n}\right)$. Tale affermazione è vera per ogni fase e la probabilità è indipendente dalle fasi precedenti in quanto dipende solo dalle scelte casuali fatte per stabilire i valori $x_i$.

Quindi la probabilità che non venga raggiunta nessuna decisione in $s$ fasi è al massimo $\left(1-\frac{1}{2^{n-1}}\right)^s$. Per cui con probabilità almeno $1-\left(1-\frac{1}{2^{n-1}}\right)^s$ tutti i processori non guasti decidono nelle prime $s+1$ fasi. □

Si noti che il teorema fornisce una stima del "tempo" necessario a far diventare $\epsilon$ piccolo in termini di numero di fasi. Poichè siamo in un sistema asicnrono per avere una valutazione che sia una funzione del tempo reale è necessario fare delle assunzioni su quanto può durare una fase.

L'algoritmo di Ben-Or può essere adattato per far fronte a guasti di tipo bizantino ma per funzionare richiede che la percentuale di processori che possono guastarsi sia minore, più precisamente deve essere $n > 5f$.

## 6.6 Il problema del consenso in sistemi reali

Abbiamo visto che il problema del consenso è estremamente semplice da risolvere quando tutto funziona bene mentre può essere irrisolvibile in presenza di guasti. Lo abbiamo affrontato considerando varie assunzioni sia sulla sincronia del sistema sia sul tipo di guasti. Per quanto riguarda la sincronia siamo passati da un estremo, sistemi totalmente sincroni, all'altro, sistemi totalmente asincroni. La sincronia gioca un ruolo fondamentale per la progettazione di algoritmi distribuiti. I sistemi totalmente sincroni

sono molto più facili da gestire; tuttavia in pratica è molto difficile avere tali sistemi. È molto più comodo avere algoritmi che funzionano per sistemi totalmente asincroni in quanto ciò facilita l'implementazione fisica del sistema distribuito; purtroppo progettare algorimi che funzionino anche in assenza di sincronia è più difficile e a volte impossibile come abbiamo visto per il problema del consenso. Nella realtà i sistemi distribuiti sono "parzialmente" sincroni: i processori eseguono ogni passo in al massimo $\ell$ unità di tempo e i messaggi vengono consegnati in al massimo $d$ unità di tempo. Questo tipo di sincronia è più facile da implementare rispetto alla sincronia totale. In questi sistemi diventa possibile risolvere il problema del consenso. Infatti l'assunzione di parziale sincronia permette di adattare algoritmi progettati per sistemi sincroni ai sistemi parzialmente sincroni (si veda il Capitolo 25 di [26] per approfondimenti).

Nel seguito presenteremo un algortimo, chiamato PAXOS, per il consenso in sistemi parzialmente sincroni. In realtà l'algoritmo è molto più robusto in quanto la parziale sincronia è sfruttata molto poco. In pratica l'algoritmo garantisce le proprietà di safety anche se il sistema è completamente asincrono, anche se i messaggi si perdono e anche se i processori si guastano con guasti di tipo stop. Le proprietà di liveness invece sono garantite solo se per un periodo sufficientemente lungo il sistema non presenta guasti e la parziale sincronia è rispettata. Pertanto PAXOS è un algoritmo molto importante dal punto di vista pratico.

Le assunzioni riguardo al sistema sono: sistema asincrono con scambi di messaggi; per garantire la terminazione è necessario che il sistema sia parzialmente sincrono per un periodo sufficentemente lungo. Ogni nodo opera a una propria velocità, può fermarsi per un guasto stop ma può anche essere riparato. I messaggi possono richiedere un tempo arbitrariamente lungo per essere consegnati, possono essere persi, duplicati ma non alterati.

### 6.6.1 *Overview*

L'idea di base dell'algoritmo PAXOS è quella di proporre dei valori fino a che uno di essi viene accettato da una maggioranza dei processi. Il valore accettato da una maggioranza dei processi è il valore della decisione finale. Un qualsiasi nodo/processo può proporre un valore. I nodi che propongono valori verranno detti *leader*. Per proporre un valore un proponente inizia un round logico. I round sono numerati con una numerazione che permette un ordine totale (un esempio è dato da una coppia formata da un numero progressivo e l'ID del nodo). Informalmente i passi di un round sono i seguenti.

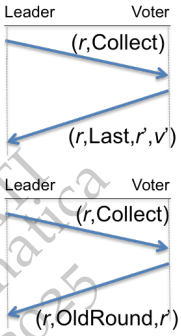



Figura 6.21: Possibili risposte al messaggio Collect

1. Il proponente (leader) spedisce un messaggio di "Collect" a tutti gli altri nodi. Il messaggio Collect serve a dichiarare l'intenzione di iniziare un round, di cui deve specificare il numero, e allo stesso tempo chiede informazioni riguardo ai round precedenti in cui i nodi sono stati coinvolti.

2. I nodi (voter) che ricevono un messaggio Collect, rispondono con un messaggio "Last", fornendo informazioni riguardo ai precedenti round, in particolare l'ultimo (cioè con il numero più alto) round in cui il nodo è stato coinvolto. Con tale risposta il nodo fa anche la promessa di non accettare nessun valore per un round con un

numero minore. Se il nodo ha promesso a qualche altro proponente di non accettare il round proposto, allora la risposta sarà un messaggio di "OldRound". La Figura 6.21 mostra le possibili risposte al messaggio Collect.

3. Quando il proponente ha ricevuto messaggi "Last" da almeno una maggioranza dei processori, decide il valore da proporre nel round e invia un messaggio di "Begin" per il round specificando il valore proposto. Il valore da proporre dipende dalle informazioni ricevute: se qualche processo già conosce una decisione allora il valore proposto sarà la decisione già presa, altrimenti sarà il valore proposto nel round più recente fra quelli conosciuti o quello iniziale del proponente.

4. I nodi che ricevono il messaggio "Begin" e non hanno nessun vincolo di rifiuto per il numero di round a cui si riferisce il messaggio, accetteranno il valore proposto spedendo un messaggio di "Accept". Nel caso siano vincolati a un round con un numero più alto spediranno un messaggio di "OldRound". La Figura 6.22 mostra le possibili risposte al messaggio Begin.

5. Se il proponente riceve almeno una maggioranza di "Accept", allora può decidere sul valore proposto nel round.

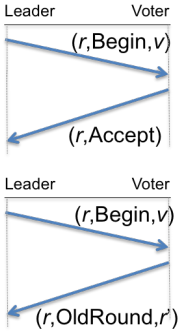



Figura 6.22: Possibili risposte al messaggio Begin

A questo punto la decisione è stata presa, ma è conosciuta solo dal proponente del round. Occorre informare gli altri nodi. Il proponente potrà farlo spedendo ulteriori messaggi "Success", come mostrato nella Figura 6.23. In realtà la cosa importante è che non è più possibile scegliere un nuovo valore. Infatti la regola per la scelta del valore da proporre e il fatto che due maggioranze hanno sempre un nodo in comune garantisce che se un valore è stato accettato in un round con un numero più piccolo allora tale valore è l'unico possibile per round con numeri più grandi.

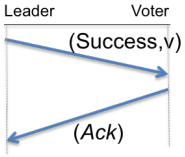



Figura 6.23: Messaggio di successo

Si noti che l'algoritmo prevede la possibilità che più leader decidano contemporaneamente di iniziare un nuovo round. I messaggi pertanto saranno sempre etichettati con il numero del round in modo tale che non ci sia confusione. I numeri dei round sono unici e sono formati da un valore intero che viene incrementato dal leader del round e dall'identificatore del leader stesso. Ad esempio il numero di round $r = (87, 3)$ è il numero di round scelto dal processo 3 con progressivo 87. Si noti che con tale numerazione per ogni coppia di numeri di round $r$ e $r'$, con $r \neq r'$ si ha che o $r < r'$ oppure $r' < r$, semplicemente definendo l'ordine $<$ con $(x, i) < (y, j)$ se e solo se $x < y$ oppure $x = y$ e $i < j$. I leader sceglieranno dei numeri di round sempre crescenti rispetto a quelli di cui vengono a conoscenza.

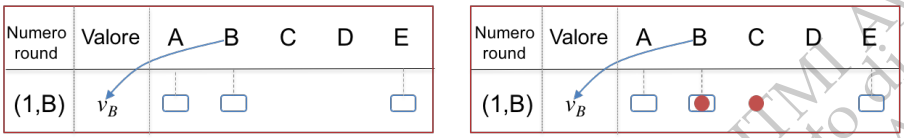



Figura 6.24: Esempio di esecuzione: Round (1,B). I riquadri vuoti indicano l'invio di un messaggio Last, i riquadri pieni l'invio di un messaggio Accept.

La Figura 6.24 mostra un esempio di esecuzione dell'algoritmo Paxos. In questo esempio il nodo $B$ ha inziato il round $(1, B)$ e i nodi $A, B$ e $E$ hanno risposto con un messaggio Last, promettendo quindi di non accettare successivi round con numero

inferiore. Poichè $B$ non è a conscenza di nessun altro round precedente è libero di scegliere il proprio valore iniziale come possibile decisione da prendere nel round. I processori $B$ e $C$ hanno anche accettato il valore proposto da $B$ nel round $(1, B)$. Questo round non ha permesso il raggiungimento di una decisione in quanto non c'è stata una maggioranza dei processori che ha accettato il valore proposto.

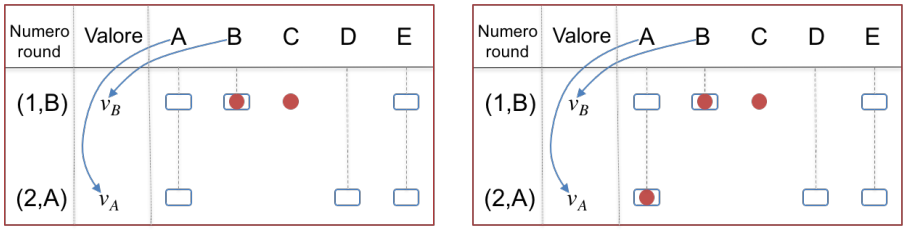



Figura 6.25: Esempio di esecuzione: Round (2,A).

Successivamente il processore $A$ decide di iniziare un nuovo round $(2, A)$, come mostrato nella Figura 6.25. In questo round i processori $A, D$ e $E$ inviano un Last message impegnandosi a non partecipare in nessun round con numero inferiore. Il valore proposto in questo round viene accettato solo da $A$, quindi anche in questo round non viene presa nessuna decisione.

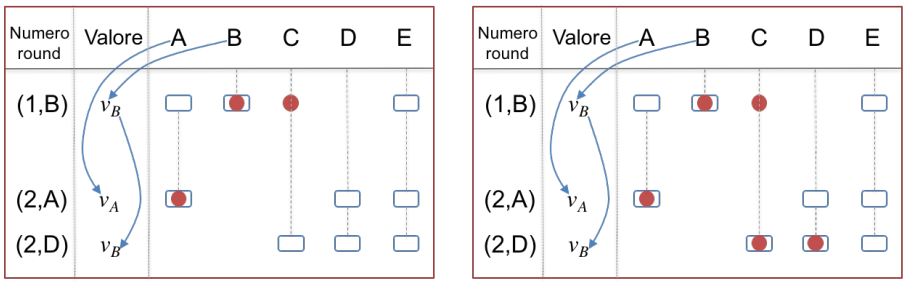



Figura 6.26: Esempio di esecuzione: Round (2,D).

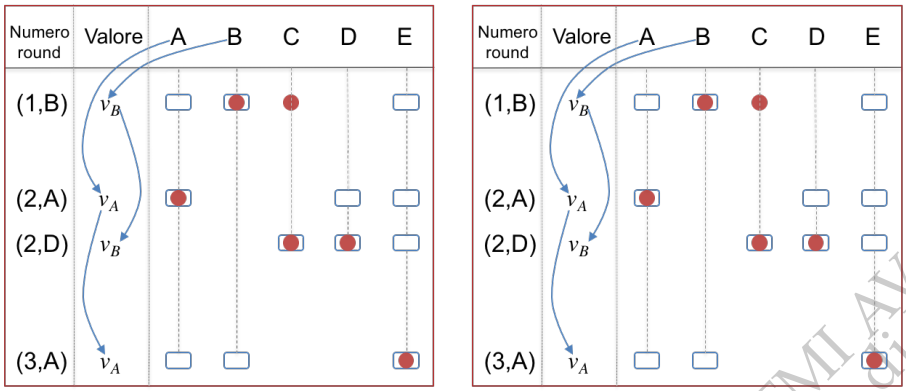



Figura 6.27: Esempio di esecuzione: Round (3,A).

Le Figure 6.26 e 6.27 mostrano altri due round che non portano a una decisione. Per questi due round è fondamentale osservare che il valore proposto è quello del round precedente con numero più alto fra quelli conosciuti. Ad esempio nel round $(2, D)$ il leader $D$ propone il valore $v_b$ in quanto c'è il processore $C$, che ha spedito un

messaggio Last, che è a conoscenza del fatto che nel round $(1, B)$ è stato proposto il valore $v_B$. Analogamente nel round $(3, A)$ viene proposto il valore $v_A$ in quanto fra la maggioranza dei processori che hanno inviato un Last messagge non c'è nessuno che conosce il valore proposto nel round $(2, D)$, mentre il processore $A$ conosce il valore proposto nel round $(2, A)$ che è il più recente fra quelli conosciuti.

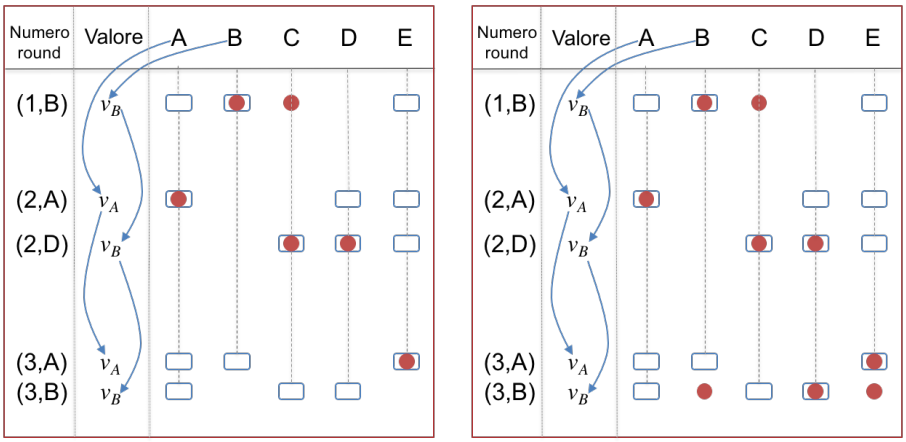



Figura 6.28: Esempio di esecuzione: Round (3,B).

La Figura 6.28 mostra il successivo round che permette di decidere sul valore $v_B$. Dal momento in cui il round $(3, B)$ ha ottenuto una maggioranza di Accept, successivi round con numeri inferiori non potranno mai avere una maggioranza (quindi non potranno nemmeno iniziare) mentre round con numeri superiori potranno proporre solo il valore $v_B$. Il fatto che il valore di un round che permette al leader di decidere deve essere l'unico valore possibile per i round successivi è cruciale ovviamente per la proprietà di accordo. Questa è la parte delicata di tutto l'algoritmo e l'intersezione garantita dall'uso delle maggioranze permette di mantenere la consistenza delle scelte dei valori da proporre nei round.

**Algorithm 24:** PAXOSLEADER$_i$

```
counter = 1
while decido di iniziare un nuovo round do
    counter++
    r = (counter, i)
    Spedisci (r, Collect)
    Ricevi messaggi (r, Last, r', v')
    Aggiorna counter al valore più grande
    Wait: una maggioranza di (r, Last, r', v')
          (Se troppa attesa, inizia un nuovo round)
    Scegli v come il v' nel messaggio Last con r' più grande
    Se non c'è un tale v' prendi il valore iniziale di i
    Spedisci (r, Begin, v)
    Ricevi messaggi (r, Accept)
    Wait: una maggioranza di (r, Accept)
          (Se troppa attesa, inizia un nuovo round)
    Decidi sul valore v di r
    Spedisci (Success, v)
    Ricevi messaggi (Ack)
```

### 6.6.2 *Pseudocodice*

La Figura 24 mostra una descrizione con pseudocodice (molto informale) del comportamento dei leader. Un nodo può può iniziare un nuovo round in qualsiasi momento, diventando così leader di quel round. Nella prima fase di un round il leader richiede informazioni a tutti gli altri nodi inviando il messaggio di Collect. La scelta del numero del round è importante. Il leader sceglie un numero di round che è maggiore di tutti i numeri di round di cui è a conoscenza. Pertanto, ogni volta che un leader riceve informazioni riguardanti altri round prende nota dei numeri di round in modo da tale poter scegliere un numero di round maggiore di tutti quelli già usati. I messaggi Last contengono anche i valori proposti nei round precedenti. Grazie a queste informazioni il leader potrà scegliere come valore da proporre nel proprio round il valore che corrisponde al numero di round più grande contenuto nel messaggi Last. Si noti che il leader sceglie tale valore solo dopo aver ricevuto messaggi Last da almeno una maggioranza dei nodi. Nel caso in cui nessuno dei messaggi Last contiene un round precedente (all'inizio nessun nodo ha partecipato a nessun round) il leader proporrà il proprio valore di input. A questo punto il leader può chiedere di accettare il valore porposto e a tal fine spedisce a tutti il messaggio Begin in cui viene specificato il valore del round. Se riceve una maggioranza di Accept il leader dichiara successo nel round e quindi può decidere sul valore proposto nel round.

L'algoritmo ammette la possibilità che più leader inizino contemporaneamente dei round. Questo significa che durante l'esecuzione un leader può ricevere messaggi da altri leader. In questo caso, poichè le attività di un leader possono interferire con quelle

degli altri leader, impedendo il raggiungimento di una decisione, è opportuno non proseguire con il round iniziato ma "lasciare il passo" al round con il numero più alto. Come vedremo la terminazione è garantita solo quando per un periodo sufficientemente lungo un solo processore si comporta da leader dando la possibilità a tale leader di terminare con successo un round.

**Algorithm 25:** PAXOSVOTER$_i$

```
lastR = (0, i)
lastV = input_i
Commit = (0, i)
while 1 do
    if Ricevo (r, Collect) then
        if r ≥ Commit then
            Spedisci (r, Last, lastR, lastV)
            Commit = r
        else
            Spedisci (r, OldRound, Commit)
    if Ricevo (r, Begin, v) then
        if r ≥ Commit then
            Spedisci (r, Accept)
            lastR = r
            lastV = v
        else
            Spedisci (r, OldRound, Commit)
```

La Figura 25 mostra una descrizione con pseudocodice del comportamento dei votanti (tutti i nodi che ricevono i messaggi spediti dai leader sono votanti). Un nodo accetterà i valori proposti a meno che non è stato già coinvolto in round precedenti. A tal fine ogni votante manterrà delle informazi riguardo a quanto fatto in precedenza, in particolare la variabile *Commit* contiene l'ultimo round per il quale il nodo ha risposto positivamente al messaggio di Collect, mentre le variabili *lastR* e *lastV* contengono rispettivamente il numero di round e il valore dell'ultimo round in cui il nodo ha accettato il valore proposto. Pertanto, quando arriva un nuovo messaggio di Collect per un round $r$, il votante controllerà se $r \geq Commit$, cioè se il nuovo round è sufficientemente "nuovo". Se lo è allora il votante spedirà le informazioni necessarie con un messaggio Last e contemporaneamente aggiornerà il valore di *Commit*. Se il round non è sufficientemente nuovo allora la risposta sarà un messaggio OldRound. Analogamente alla ricezione di un messaggio Begin, il votante risponderà con un Accept solo se il round proposto è sufficientemente nuovo.

### 6.6.3 *Accordo e validità*

L'algoritmo garantisce la proprietà di validità in quanto tutti i valori che vengono proposti sono sempre uguali a uno degli input. Provare che la proprietà di accordo è soddisfatta è un po' più complicato. Una prova formale va al di là degli obiettivi di questo corso. Forniamo delle argomentazioni informali.

Poichè durante l'esecuzione dell'algoritmo è possibile che dei leader inizino dei round che non saranno mai completati per il fatto che una maggioranza dei processori partecipa a round con numeri più alti, classificheremo tali round come *bloccati*. Quindi un round bloccato è un round $r$ per il quale una maggioranza dei nodi ha fatto un commit per un round con un numero più grande di $r$ e pertanto $r$ non potrà procedere.

Inoltre diremo che un round $r$ è *ancorato* se ogni altro round $r' < r$ è o bloccato oppure ha lo stesso valore di $r$.

Osserviamo che se il nodo $i$ ha partecipato ad un round $r_1$ e spedisce un messaggio Collect per il round $r_2 > r_1$, allora il nodo $i$ non parteciperà a nessun round $r$, $r_1 < r < r_2$.

**Lemma 6.6.1** *Ogni round non bloccato è ancorato.*

Dimostrazione. **(sketch)** Procediamo per induzione sulla lunghezza dell'esecuzione. Quindi supponiamo che l'asserzione è vera in uno stato $s$ e dobbiamo provare che è vera nello stato $s'$ a cui si arriva da $s$ eseguendo uno step dell'algoritmo. L'unico passo che può rendere falsa l'asserzione del lemma è la scelta di un nuovo valore per un round: se il valore non fosse uguale a quelli scelti per tutti i round precedenti non bloccati il lemma sarebbe falso. Sia $r$ il numero del round. Poichè il valore viene scelto quando una maggioranza dei nodi ha inviato un messaggio Last si ha che quella maggioranza non parteciperà a round compresi fra $r'$ e $r$ dove $r'$ è il valore più alto ricevuto nei messaggi Last. Si noti che poichè stiamo scegliendo il valore di $r$ il round $r'$ è esso stesso ancorato (per l'ipotesi induttiva). Pertanto anche $r$ è ancorato. □

### 6.6.4 *Terminazione*

L'algoritmo Paxos non garantisce terminazione in presenza di leader multipli: ognuno di essi potrebbe iniziare un nuovo round impedendo a quelli precedenti di arrivare a completamento. Tuttavia questo è possibile solo in presenza di leader multipli. Se la situazione si stabilizza ed esiste un solo leader esso potrà portare a completamento un round. Infatti dal momento in cui il leader è unico un solo round iniziato dall'unico leader potrebbe essere bloccato da round iniziati precedentemente da nodi che erano leader (in quanto i numeri di quei round potrebbero essere più grande del round iniziato dal leader). Un secondo round iniziato dall'unico leader sarà portato a termine senza ostacoli.

Pertanto per garantire la terminazione dell'algoritmo è necessario garantire l'unicità di un leader e la possibilità per questo leader di comunicare con una maggioranza dei nodi per un periodo sufficientemente lungo.

Si noti anche che dopo la decisione presa dal leader è necessario che anche gli altri nodi vengano informati della decisione: è sufficiente che il leader invii un messaggio a

tutti gli altri nodi e si accerti, tramite un riscontro, che il messaggio sia stato ricevuto (si veda la Figura 6.23).

## 6.7 Note bibliografiche

Molto del materiale presentato in questo capitolo è tratto da L1996 [26]. Il problema dei due generali, e la prova di impossibilità per il caso di sistemi deterministici con perdita di messaggi è stato presentato in [16]; l'algoritmo randomizzato è stato presentato in [32]. I risultati presentati per sistemi con guasti bizantini sono stati studiati in [27] e [24]. La prova di impossibilità per sistemi asincroni con guasti stop è dovuta a Fischer, Lynch e Paterson [13]. L'algoritmo di Ben-Or è stato presentato in [4]. L'algoritmo Paxos è dovuto a Lamport [22]; la sua iniziale stesura "archeologica" è stata rivisitata da De Prisco, Lampson e Lynch [10, 11]. L'algoritmo Paxos può essere esteso per gestire anche guasti bizantini [23].

## 6.8 Esercizi

1. Provare che il problema dell'attacco coordinato con guasti sui canali non è risolvibile per un qualsiasi fissato $n$, $n > 2$.

2. Si consideri il problema dell'attacco coordinato con guasti sui canali, con la seguente variante sulla condizione di accordo: se c'è qualche processo che decide 1 allora ci devono essere almeno due processi che decidono 1 (in altre parole l'unico caso che vogliamo evitare è che un generale decida di attaccare da solo). Per $n = 2$ questa condizione è equivalente a quella originale, ma per $n \geq 3$ no. Questo problema è risolvibile? Se sì, si dia un algoritmo, se no si fornisca una prova di impossibilità.

3. Si consideri il problema dell'attacco coordinato con guasti sui canali in un sistema sincrono per il caso semplice di 2 processi $p_1$ e $p_2$ connessi da un canale di comunicazione. Il canale si comporta in maniera randomizzata e in ogni round, in maniera indipendente dagli altri round, con probabilità $p$ entrambi i messaggi vengono consegnati e con probabilità $1 - p$ vengono entrambi perso.

   Progettare un algoritmo e valutare con quale probabilità soddisfa ciascuna delle 3 proprietà richieste dal problema. Si cerchi di ottenere un algoritmo che rende quanto più alte possibile le probabilità di soddisfare le proprietà.

4. Si consideri il problema dell'attacco coordinato con guasti sui canali in un sistema sincrono per il caso semplice di 2 processi $p_1$ e $p_2$ connessi da un canale di comunicazione. Il canale si comporta in maniera randomizzata e in ogni round, in maniera indipendente dagli altri round, il messaggio spedito da $p_1$ a $p_2$ viene consegnato con probabilità $q_1$, e quindi perso con probabilità $1 - q_1$ mentre quello spedito da $p_2$ a $p_1$ viene consegnato con probabilità $q_2$ e quindi perso con probabilità $1 - q_2$.

   Progettare un algoritmo e valutare con quale probabilità soddisfa ciascuna delle 3 proprietà richieste dal problema. Si cerchi di ottenere un algoritmo che rende quanto più alte possibile le probabilità di soddisfare le proprietà.

5. Si consideri l'algoritmo FloodSet in un sistema distribuito sincrono di 4 processori $p_1, p_2, p_3, p_4$ che iniziano la computazione con i valori di input $1, 0, 0, 0$. Si consideri un'esecuzione $\alpha$ in cui il processore si ferma (guasto stop) nel primo round dopo aver spedito il proprio messaggio solo a $p_2$, mentre $p_2$ si ferma nel secondo round dopo aver spedito il messaggio a $p_1$ e $p_3$ (quindi non a $p_4$). Si descriva l'esecuzione dell'algoritmo specificando l'insieme di valori di input noti a ognuno dei processori in ognuno dei round; si assuma che l'algoritmo venga eseguito per il massimo numero di round.

6. Perché l'algoritmo FloodSet stabilisce l'output al round $f + 1$? Cosa succede se cambiamo l'algoritmo facendo prendere la decisione al round $f$? E se la prendiamo al round $f + 2$? Stabilire, in entrambi i casi, se l'algoritmo continua a funzionare fornendo una spiegazione, se funziona, o un controesempio, se non funziona.

7. Si consideri la seguente variante dell'algoritmo FloodSet: al posto dell'insieme $W$ ogni processore tiene traccia solo del minimo valore visto, *minv*. Quindi inizialmente $minv_i = v_i$ per tutti i processori $p_i$, dove $v_i$ è l'input di $p_i$. Alla ricezione dei messaggi $x_j$, il processore $p_i$ aggiorna $minv = \min\{minv, x_j\}$, per tutti i processori $p_j$ dai quali $p_i$ riceve messaggi. Alla fine del round $f + 1$ $p_i$ decide dando in output *minv*. Questo algoritmo funziona? Se sì dare una prova, se no, fornire un controesempio.

8. Si consideri l'algoritmo EIGstop. Sappiamo che esso può gestire guasti di tipo stop ma non guasti di tipo bizantino. Si fornisca un esempio in cui guasti bizantini causano una violazione della proprietà di validità.

9. Si consideri l'algoritmo EIGbyz in un sistema con 7 processori. Scegliere dei valori di input per tutti i processori. Scegliere 2 processori che saranno guasti durante l'esecuzione dell'algoritmo: essi spediranno dei valori casuali al posto di quelli stabiliti dall'algoritmo: lanciare una monetina per stabilire i valori. Mostrare l'esecuzione dell'algoritmo per 3 round (uno in più del numero di guasti) e verificare che l'algoritmo funziona correttamente.

10. Si consideri l'algoritmo EIGbyz. Si costruiscano delle esecuizioni in cui l'algoritmo fornisce un risultato errato nei seguenti casi:

    - 7 nodi, 2 guasti e 2 round.
    - 6 nodi, 2 guasti e 3 round.

11. Progettare un algoritmo per il problema del consenso in un sistema sincrono con al massimo $f$ guasti stop in modo tale che venga soddisfatta anche la seguente proprietà (detta di *early stopping*): se in una esecuzione si verificano solo $f' < f$ guasti allora il tempo necessario a decidere è proporzionale a $f'$ e non a $f$.

12. Si risolva l'esercizio precedente anche per il caso di guasti di tipo bizantino.

13. Si consideri il problema del consenso in sistemi asicroni con guasti stop. Si fornisca una variante dell'algoritmo di BenOr in cui i processori non guasti possono, a un certo punto, fermarsi.

14. Si consideri la segfuente variante dell'algoritmo di BenOr.

*Algoritmo* ByzBenOr. Ogni processore mantiene due variabili locali $x$ e $y$. Inizialmente la $x$ contiene il valore di input del processore e la $y$ è `null`. Ogni processore $p_i$ esegue una serie di *fasi* numerate $1, 2, \ldots$. Il processore continua ad eseguire l'algoritmo anche dopo aver preso la decisione finale. In ogni fase $s$, il processore $p_i$ esegue 2 rounds:

Round 1: Spedisce in broadcast il messaggio $(\mathit{first}, s, x)$. Aspetta l'arrivo di almeno $n - f$ messaggi della forma $(\mathit{first}, s, v)$. Se almeno $n - 2f$ hanno tutti lo stesso valore $v$ allora $y := v$, altrimenti $y :=$ `null`. Si noti che l'assunzione $n > 5f$ implica che non possono esserci due valori $v$ in almeno $n - 2f$ messaggi.

Round 2: Spedisce in broadcast il messaggio $(\mathit{second}, s, y)$. Aspetta l'arrivo di almeno $n - f$ messaggi della forma $(\mathit{second}, s, v)$.

i. Se almeno $n - 2f$ hanno lo stesso $v \neq \mathit{null}$ allora $x = v$ e decide $v$ se non ha già deciso.

ii. Se almeno $n - 4f$ messaggi contengono uno stesso valore $v$ allora $x = v$ ma non si prende una decisione.

iii. Altrimenti, si sceglie uniformemente a caso 0 o 1 come valore di $x$.

Si provi che ByzBenOr risolve il problema in sistemi asincroni con guasti bizantini nel caso $n > 5f$.

15. Si consideri l'algoritmo Paxos in un sistema con 5 processori e si descriva una esecuzione in cui occorrono 3 round per arrivare alla decisione.

# *Bibliografia*


[1] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. *Annals of mathematics*, 160(2):781–793, 2002.

[2] J. Aspnes. Notes on randomized algorithms cpsc 469/569: Fall 2016, 2016. www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf.

[3] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[4] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.

[5] K. P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[6] C. J. Colbourn. The complexity of completing partial latin square. *Discrete Applied Mathematics*, 8(1):25–30, 1984.

[7] S. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[8] T. H. Cormen, R. L. Rivest, C. E. Leiserson, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[9] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2008.

[10] R. De Prisco. *Revisting the Paxos Algorithm*. PhD thesis, Massachusetts Institute of Technology, 1997.

[11] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the paxos algorithm. *Theoretical Computer Science*, 243(1):35 – 91, 2000.

[12] M. du Sautoy. *L'equazione da un milione di dollari*. RCS Libri, Titolo originale: The number misteries. A Mathematical Odyssey Through Every Day Life., 2010.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[14] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[15] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[16] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.

[17] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[18] D. R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In V. Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 21–30. ACM/SIAM, 1993.

[19] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996.

[20] J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson Education Limited, Harlow, Essex, GB, 2014.

[21] L. Lamport. Definizione di un sistema distribuito. http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt, 1987.

[22] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[23] L. Lamport. Byzantizing paxos by refinement. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 211–224, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[25] L. A. Levin. Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. In russo.

[26] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[27] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[28] D. Poole and A. Mackworth. *Artificial Intelligence*. Cambridge University Press, 2nd edition, 2017. Disponibile online in versione HTML: `http://artint.info/index.html`.

[29] M. Soltys. *An Introduction to the Analysis of Algorithms*. World Scientific, 2st edition, 2012.

[30] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[31] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2001.

[32] G. Varghese and N. A. Lynch. A tradeoff between safety and liveness for randomized coordinated attack. *Inf. Comput.*, 128(1):57–71, July 1996.

[33] V. V. Vazirani. *Approximation Algorithms*. Springer, 1st edition, 2001.