Algoritmi greedy
II parte

Progettazione di Algoritmi a.a. 2019-20
Matricole congrue a 1
Docente: Annalisa De Bonis

17

17

Partizionamento di intervalli

In questo caso disponiamo di più risorse identiche tra di loro e vogliamo che vengano svolte tutte le attività in modo tale da usare il minor numero di risorse e tenendo conto del fatto che due attività non possono usufruire della stessa risorsa allo stesso tempo.

- Durante il suo svolgimento, ciascuna attività necessita di un'unica risorsa.
- Una risorsa puo` essere allocata ad al più un'attività alla volta

Nell'interval scheduling avevamo un'unica risorsa e volevamo che venissero svolte il massimo numero di attività

Un'istanza del problema (Input) consiste in un insieme di n intervalli $[s_1,f_1],\ldots,[s_n,f_n]$ che rappresentano gli intervalli durante i quali si svolgono le n attività.
Obiettivo: far svolgere le n attività utilizzando il minor numero possible di risorse e in modo che ciascuna risorsa utilizzata venga allocata ad al piùun'attività alla volta.

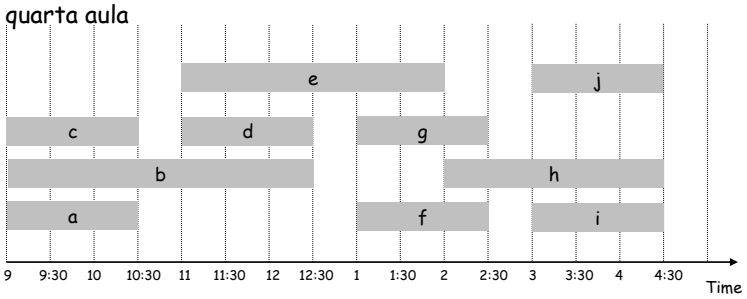
18

Partizionamento di intervalli

Esempio. Attività = lezioni da svolgere; Risorse= aule

- La lezione j comincia ad s_j e finisce ad f_j.
- Obiettivo: trovare il minor numero di aule che permetta di far svolgere tutte le lezioni in modo che non ci siano due lezioni che si svolgono contemporaneamente nella stessa aula.

Esempio: Questo schedule usa 4 aule (una per livello) per 10 lezioni: e, j nella prima aula; c,d,g nella seconda aula; b, h nella terza aula; a, f, i nella quarta aula

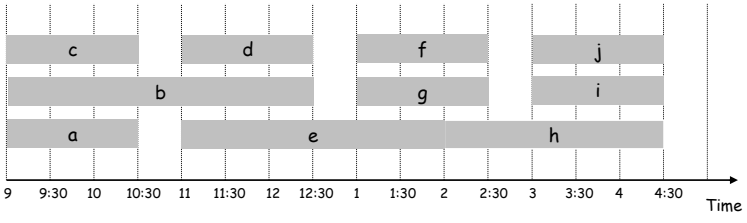


19

Partizionamento di intervalli

Esempio. Questo schedule usa solo 3 aule per le stesse attività: {c,d,f,j},{b,g,i},{a,e,h} .

Si noti che la disposizione delle lezioni lungo l'asse delle ascisse è fissato dall'input mentre la disposizione lungo l'asse delle y è fissato dall'algoritmo.



20

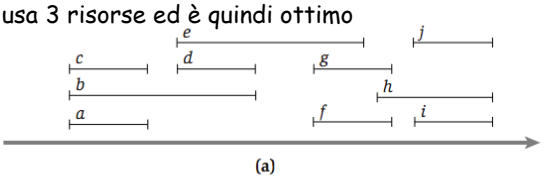
Partizionamento di intervalli: limite inferiore alla soluzione ottima

Def. Immaginiamo di disporre gli intervalli lungo l'asse delle ordinate in modo da non avere due intervalli che si sovrappongono alla stessa altezza (un qualsiasi schedule fa al nostro caso) . La profondità di un insieme di intervalli è il numero massimo di intervalli intersecabili con una singola linea verticale che si muove lungo l'asse delle ascisse.

Osservazione. Numero di risorse necessarie ≥ profondità.

Esempio. L'insieme di intervalli in figura (a) ha profondità 3. Lo schedule in figura (b) usa 3 risorse ed è quindi ottimo

Una risorsa per livello. Per un totale di 4 risorse



(a)

Una risorsa per livello. Per un totale di 3 risorse

(b)

21

21

Partizionamento di intervalli: soluzione ottima

Domanda. E' sempre possibile trovare uno schedule pari alla profondità dell'insieme di intervalli?

Osservazione. Se così fosse allora il problema del partizionamento si ridurrebbe a constatare quanti intervalli si sovrappongono in un certo punto. Questa è una caratteristica locale per cui un algoritmo greedy potrebbe essere la scelta migliore.

Nel seguito vedremo un algoritmo greedy che trova una soluzione che usa un numero di risorse pari alla profondità e che quindi è ottimo

Idea dell'algoritmo applicata all'esempio delle lezioni:

- Considera le lezioni in ordine non decrescente dei tempi di inizio
- Ogni volta che esamina una lezione controlla se le può essere allocata una delle aule già utilizzate per qualcuna delle lezioni esaminate in precedenza. In caso contrario alloca una nuova aula.

22

Partizionamento di intervalli: Algoritmo greedy

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0

for j = 1 to n {
   if (interval j can be assigned an already allocated resources v)
      assign resource v to interval j
   else
      allocate a new resource d + 1
      assign the new resource d+1 to interval j
      d ← d + 1
}
```

- Osservazione. L'algoritmo greedy non assegna mai la stessa risorsa a due intervalli incompatibili

- Dimostreremo che alla j-esima iterazione del for il valore di d è pari alla profondità dell'insieme ordinato di intervalli {1,2,…,j}
- Il valore finale di d è quindi pari alla profondità dell'insieme di intervalli {1,…,n}

23

Partizionamento di intervalli: Ottimalità soluzione greedy

Lemma. Alla j-esima iterazione del for, il valore di d in quella iterazione è pari alla profondità dell'insieme di intervalli {1,2,…,j}.

Dim.

Caso in cui alla j-esima iterazione del for viene allocata una nuova risorsa

- Supponiamo che alla j-esima iterazione del ciclo di for d venga incrementato.
- La risorsa d è stata allocata perchè ciascuna delle altre d-1 risorse già allocate è assegnata al tempo s_j ad un intervallo che non è ancora terminato.
  → $f_i > s_j$ per ciascuna attività i che impegna una delle d-1 risorse
- **Siccome l'algoritmo considera gli intervalli in ordine non decrescente dei tempi di inizio**, i d-1 intervalli a cui sono assegnate le d-1 risorse iniziano non più tardi di s_j → $s_i \le s_j$ e $f_i > s_j$, per ciascuna attività i che impegna una delle d-1 risorse
- Di conseguenza, questi d-1 intervalli e l'intervallo $[s_j,f_j]$ si sovrappongono al tempo s_j (sono cioè tutti intersecabili da una retta verticale che passa per s_j). Per definizione di profondità, d ≤ profondità di {1,2,…,j}
- Abbiamo osservato che il numero di risorse allocate per intervallo è maggiore uguale della sua profondità per cui d ≥ profondità di {1,2,…,j}
- Dagli ultimi due punti segue che d = profondità di {1,2,…,j}

24

24

Partizionamento di intervalli: Ottimalità soluzione greedy

Caso in cui alla j-esima iterazione del for non viene allocata una nuova risorsa:

- Sia j' l'ultima iterazione prima della j-esima in cui viene allocata una nuova risorsa.
- Per quanto dimostrato nella slide precedente, d= profondità {1,2,…,j'}.
- Siccome {1,2,…,j'} è contenuto in {1,2,…,j} allora si ha che profondità {1,2,…,j'} ≤ profondità di {1,2,…,j} e quindi per il punto precedente d ≤ profondità di {1,2,…,j}.
- Usiamo di nuovo il fatto che il numero di risorse allocate per un insieme di intervalli è maggiore o uguale della profondità dell'insieme. Si ha quindi d ≥profondità di {1,2,…,j}.
- Gli ultimi due punti implicano che d=profondità di {1,2,…,j}.

25

25

Partizionamento di intervalli: Ottimalità soluzione greedy

Teorema. L'algoritmo greedy usa esattamente un numero di risorse pari alla profondità dell'insieme di intervalli {1,2,…,n}

Dim. Per il lemma precedente, quando j=n il numero di risorse allocate è uguale alla profondità dell'intervallo {1,2,…,n}

26

26

Partizionamento di intervalli: Analisi Algoritmo greedy

Implementazione. $O(n \log n)$.

- Per ogni risorsa p, manteniamo il tempo di fine più grande tra quelli degli intervalli assegnati fino a quel momento a p. Indichiamo questo tempo con k_p
- Usiamo una coda a priorità di coppie della forma (p, k_p), dove p è una risorsa già allocata e k_p è l'istante fino al quale è occupata.
  - In questo modo l'elemento con chiave minima indica la risorsa v che si rende disponibile per prima
- Se $s_j \geq k_v$ allora all'intervallo j può essere allocata la risorsa v. In questo caso cancelliamo v dalla coda e la reinseriamo con chiave $k_v = f_j$. In caso contrario allochiamo una nuova risorsa e la inseriamo nella coda associandole la chiave f_j
- Se usiamo una coda a priorità implementata con heap ogni operazione sulla coda richiede log m, dove m è la profondità dell'insieme di intervalli. Poiché vengono fatte $O(n)$ operazioni di questo tipo, il costo complessivo del for è $O(n \log m)$.
- A questo va aggiunto il costo dell'ordinamento che è $O(n \log n)$. Siccome $m \leq n$ il costo dell'algoritmo è $O(n \log n)$

27

27

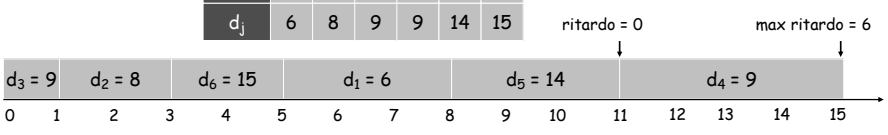
Scheduling per Minimizzare i Ritardi

Problema della minimizzazione dei ritardi.

- Una singola risorsa in grado di elaborare un unico job.
- Il job j richiede t_j unità di tempo e deve essere terminato entro il tempo d_j (scadenza).
- Considerazione: Se j comincia al tempo s_j allora finisce al tempo $f_j = s_j + t_j$.
- Def. Ritardo del job j è definito come $\ell_j = \max \{ 0, f_j - d_j \}$.
- Obiettivo: trovare uno scheduling di tutti i job che minimizzi il ritardo massimo $L = \max \ell_j$.

Esempio:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



PROGETTAZIONE DI ALGORITMI A.A. 2019-20
A.DE BONIS

28

28

Minimizzare il ritardo: Algoritmo Greedy

Schema greedy. Considera i job in un certo ordine.

- [Shortest processing time first] Considera i job in ordine non decrescente dei tempi di elaborazione t_j.
- [Earliest deadline first] Considera i job in ordine non decrescente dei tempi entro i quali devono essere ultimati d_j.
- [Smallest slack] Considera i job in ordine non decrescente degli scarti $d_j - t_j$.

Minimizzare il ritardo: Algoritmo Greedy

- [Shortest processing time first] Considera i job in ordine non decrescente dei tempi di elaborazione t_j.

	1	2
t_j	1	10
d_j	100	10

controesempio

Viene eseguito prima il job 1. Ritardo massimo è 11-10=1. Se avessimo eseguito prima il job 2 avremmo avuto ℓ_1 = max{0, 10-10}=0 e ℓ_2 = max{0,11-100}=0 per cui il ritardo massimo sarebbe stato 0.

- [Smallest slack] Considera i job in ordine non decrescente degli scarti $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

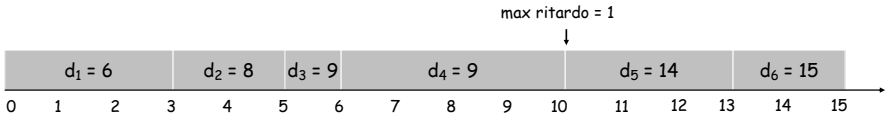
controesempio

Viene eseguito prima il job 2. Ritardo massimo è 11-2=9. Se avessimo eseguito prima il job 1 il ritardo massimo sarebbe stato 11-10=1

Minimizzare il ritardo: Algoritmo Greedy

Algoritmo greedy. Earliest deadline first: Considera i job in ordine non decrescente dei tempi d_j entro i quali devono essere ultimati.

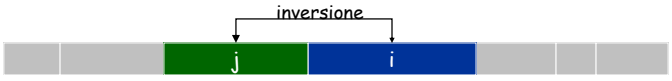
```
Sort n jobs by deadline so that d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervals [s1,f1],...,[sn, fn ]
```



31

Minimizzare il ritardo: Inversioni

Def. Un' inversione in uno scheduling S è una coppia di job i e j tali che: $d_i < d_j$ ma j viene eseguito prima di i.



32

Ottimalità soluzione greedy

La dimostrazione dell'ottimalità si basa sulle seguenti osservazioni che andremo poi a dimostrare

1. La soluzione greedy ha le seguenti due proprietà:
   a. Nessun idle time. Non ci sono momenti in cui la risorsa non è utilizzata
   b. Nessuna inversione. Se un job j ha scadenza maggiore di quella di un job i allora viene eseguito dopo i

2. Tutte le soluzioni che hanno in comune con la soluzione greedy le caratteristiche a e b, hanno lo stesso ritardo massimo della soluzione greedy.

3. Ogni soluzione ottima può essere trasformata in un'altra soluzione ottima per cui valgono la a e la b

Si noti che la 3 implica che esiste una soluzione ottima che soddisfa la a e la b e, per la 2, questa soluzione ha lo stesso ritardo massimo della soluzione greedy che quindi e` a sua volta ottima.

Dimostrazioni delle osservazioni 1. 2. e 3.

1. La soluzione greedy ha le seguenti due proprietà:

   a. Nessun idle time. Non ci sono momenti in cui la risorsa non è utilizzata
   b. Nessuna inversione. Se un job j ha scadenza maggiore di quella di un job i allora viene eseguito dopo i

Dim.

Il punto a discende dal fatto che ciascun job comincia nello stesso istante in cui finisce quello precedente.

Il punto b discende dal fatto che i job sono esaminati in base all'ordine non decrescente delle scadenze.

Dimostrazioni delle osservazioni 1. 2. e 3.

Prima di dimostrare il punto 2 consideriamo i seguenti fatti

Fatto I. In uno scheduling con le caratteristiche a e b i job con una stessa scadenza d sono disposti uno di seguito all'altro.

Dim.

- Consideriamo i e j con $d_i=d_j=d$ e assumiamo senza perdere di generalità (da ora in poi s.p.d.g.) che i venga eseguito prima di j .
- Supponiamo **per assurdo** che tra i e j venga eseguito il job q con $d_i \neq d_q$.
- Se $d < d_q$ allora la coppia j, q è un'inversione. Se $d > d_q$ allora la coppia i, q è un'inversione. Ciò contraddice la proprietà b.

Ne consegue che tra due job con una stessa scadenza d non vengono eseguiti job con scadenza diversa da d e poiché lo scheduling non ha idle time, i job con una stessa scadenza vengono eseguiti uno di seguito all'altro.

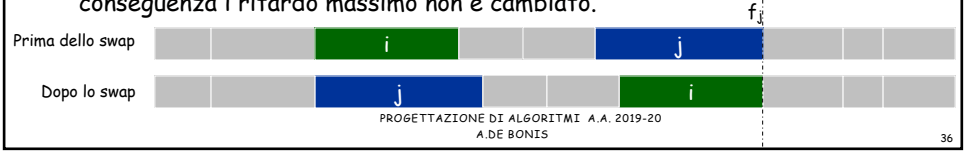
35

Dimostrazioni delle osservazioni 1. 2. e 3.

Fatto II. Se in uno scheduling con le caratteristiche a e b scambiamo due job con la stessa scadenza, il ritardo massimo non cambia.

Dim.

- Consideriamo due job i e j con $d_i=d_j$ e supponiamo s.p.d.g. che i preceda j in S.
- Per il fatto I, tra i e j vengono eseguiti solo job con la stessa scadenza di i e j. Ovviamente il ritardo di j è maggiore del ritardo di i e dei ritardi di tutti i job eseguiti tra i e j perché j finisce dopo tutti questi job e ha la loro stessa scadenza.
- Se scambiamo i con j in S otteniamo che il ritardo di j non può essere aumentato mentre quello di i è diventato uguale a quello che aveva prima j in quanto i finisce nello stesso istante in cui finiva prima j e la scadenza di i è la stessa di j. Il ritardo dei job compresi tra i e j potrebbe essere aumentato ma non può superare il ritardo che aveva prima j. Di conseguenza i ritardo massimo non è cambiato.



36

Dimostrazioni delle osservazioni 1. 2. e 3.

2. Tutte le soluzioni che hanno in comune con la soluzione greedy le caratteristiche a e b, hanno lo stesso ritardo massimo della soluzione greedy

Dim.

- Dimostriamo che dati due scheduling S ed S' di n job entrambi aventi le caratteristiche a e b, S può essere trasformato in S' senza che il suo ritardo massimo risulti modificato.
- Osserviamo che S ed S' possono differire solo per il modo in cui sono disposti tra di loro job con la stessa scadenza altrimenti o S o S' conterrebbero un'inversione.
- Di conseguenza S può essere trasformato in S' scambiando tra di loro di posto coppie di job con la stessa scadenza.
- Per il fatto II, scambiando coppie di job con la stessa scadenza il ritardo max non cambia. Di conseguenza possiamo trasformare S in S' senza che cambi il ritardo max. In altre parole S ed S' hanno lo stesso ritardo max.
- Prendendo S uguale ad un qualsiasi scheduling con le caratteristiche a e b ed S' uguale allo scheduling greedy si ottiene la tesi.

37

37

Dimostrazioni delle osservazioni 1. 2. e 3.

Prima di dimostrare il punto 3 consideriamo i seguenti fatti.

Fatto III. Una soluzione ottima può essere trasformata in una soluzione soluzione con nessun tempo di inattività (idle time).
Dim. Se tra il momento in cui finisce l'elaborazione di un job e quello in cui inizia il successivo vi è un gap, basta shiftare all'indietro l'inizio del job successivo in modo che cominci non appena finisce il precedente. Ovviamenti i ritardi dei job non aumentano dopo ogni shift

Esempio: Soluzione ottima con idle time

Esempio: Soluzione ottima con nessun idle time

.

38

Dimostrazioni delle osservazioni 1. 2. e 3.

Fatto IV. Se uno scheduling privo di idle time ha un'inversione allora esso ha una coppia di job invertiti che cominciano uno dopo l'altro.

Dim.

- Consideriamo tutte le coppie di job i e j tali che $d_i < d_j$ e j viene eseguito prima di i nello scheduling. Supponiamo per assurdo tutte le coppie di questo tipo siano separate da un job.
- Tra tutte le coppie siffatte prendiamo quella più vicina nello scheduling. Deve esistere un job $k \neq i$ eseguito subito dopo j che non forma un'inversione né con i né con j altrimenti i e j non formerebbero l'inversione più vicina.
- Deve quindi essere $d_j \leq d_k$ e $d_k \leq d_i$. Le due diseguaglianze implicano $d_j \leq d_i$ il che contraddice il fatto che la coppia i, j sia un'inversione.

Abbiamo dimostrato che se uno scheduling contiene inversioni allora deve esistere una coppia di job invertiti tra i quali non viene eseguito nessun altro job. Siccome lo scheduling considerato non contiene idle time allora questi due job invertiti devono cominciare uno dopo l'altro

39

Dimostrazioni delle osservazioni 1. 2. e 3.

Fatto V. Scambiare due job adiacenti invertiti i e j riduce il numero totale di inversioni di uno e non fa aumentare il ritardo massimo.

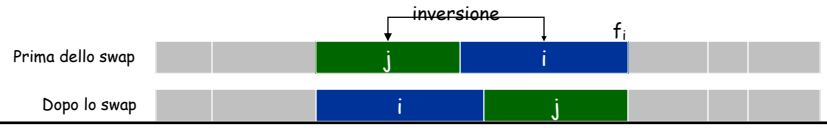
Dim. Supponiamo $d_i < d_j$ e che j precede i nello scheduling.

Siano $\ell_1, \ldots, \ell_n$ i ritardi degli n job e siano $\ell'_1, \ldots, \ell'_n$ i ritardi degli n job dopo aver scambiato i e j di posto. Si ha che

- $\ell'_k = \ell_k$ per tutti i $k \neq i, j$
- $\ell'_i < \ell_i$ perchè viene anticipata la sua esecuzione.
- Vediamo se il ritardo di j è aumentato al punto da far aumentare il ritardo max. Ci basta considerare il caso in cui $\ell'_j > 0$ altrimenti vuol dire che il ritardo di j non è aumentato. Si ha quindi

$$\begin{aligned} \ell'_j &= f'_j - d_j && \text{(per la definizione di ritardo)} \\ &= f_i - d_j && \text{(dopo lo swap, j finisce al tempo } f_i\text{)} \\ &< f_i - d_i && \text{(in quanto } d_i < d_j\text{)} \\ &\leq \ell_i && \text{(per la definizione di ritardo)} \end{aligned}$$

per cui il max ritardo non è aumentato



40

Dimostrazioni delle osservazioni 1. 2. e 3.

- 3. Ogni soluzione ottima S può essere trasformata in un'altra soluzione ottima per cui valgono la a e la b

- Dim.
- Il fatto III implica che la soluzione ottima S può essere trasformata in una soluzione ottima S' per cui non ci sono idle time.
- Il fatto IV implica che se la soluzione ottima S' contiene inversioni allora S' contiene una coppia di job **adiacenti** invertiti.
  - Il fatto V implica che se scambiamo le posizioni di questi due job invertiti adiacenti il ritardo massimo non cambia per cui otteniamo ancora una soluzione ottima con un numero inferiore di inversioni.
- Quindi se S' contiene inversioni, possiamo scambiare di posto coppie di job adiacenti invertiti fino a che non ci sono più inversioni e la soluzione S'' così ottenuta sarà a sua volta ottima.