Programmazione dinamica (II parte)

Progettazione di Algoritmi a.a. 2017-18

Matricole congrue a 1

Docente: Annalisa De Bonis

Esercizio 27 cap. 6

- I proprietari di una pompa di carburante devono confrontarsi con la seguente situazione:
- Hanno un grande serbatoio che immagazzina gas; il serbatoio puo` immagazzinare fino ad L galloni alla volta.
- Ordinare carburante e` molto costoso e per questo essi vogliono farlo raramente. Per ciascun ordine pagano un prezzo fisso P in aggiunta al costo della carburante.
- Immagazzinare un gallone di carburante per un giorno in piu` costa c dollari per cui ordinare carburante troppo in anticipo aumenta i costi di immagazzinamento.
- I proprietari del distributore stanno progettando di chiudere per una settimana durante l'inverno e vogliono che per allora il serbatoio sia vuoto.
- In base all'esperienza degli anni precedenti, essi sanno esattamente di quanto carburante hanno bisogno. Assumendo che chiuderanno dopo n giorni e che hanno bisogno di g_i galloni per ciascun giorno i=1,...n e che al giorno 0 il serbatoio e` vuoto, dare un algoritmo per decidere in quali giorni devono effettuare gli ordini e la quantita` di carburante che devono ordinare in modo da minimizzare il costo.

Esercizio 27 cap. 6: Soluzione

- Supponiamo che il giorno 1 i proprietari ordinino carburante per i primi i-1 giorni: $g_1+g_2+\ldots+g_{i-1}$
- Il costo di questa operazione si traduce in un costo fisso di P piu` un costo di $c(g_2+2g_3+3g_4\ldots+(i-2)g_{i-1})$
- Sia OPT(d) il costo della soluzione ottima per il periodo che va dal giorno d al giorno n partendo con il serbatoio vuoto
- La soluzione ottima da d ad n include sicuramente un ordine al giorno d per un certo quantitativo di carburante. Sia f il giorno in cui verra` fatto il prossimo ordine.
  - La quantita ordinata il giorno d deve essere $g_d+g_{d+1}+\ldots+g_{f-1.}$ ($g_d+g_{d+1}+\ldots+g_{f-1}\leq L$)
  - Il costo connesso a questo ordine e` $P+c(g_{d+1}+2g_{d+2}+3g_{d+3}\ldots+(f-1-d)g_{f-1})$
  - Il costo della soluzione ottima da d ad n se il secondo ordine in questo intervallo avviene al tempo f e` $P+\sum_{i=d}^{f-1} c(i-d)g_i + OPT(f)$

$$OPT(d) = P + \min_{f>d:\sum_{i=d}^{f-1} g_i \leq L} \sum_{i=d}^{f-1} c(i-d)g_i + OPT(f)$$

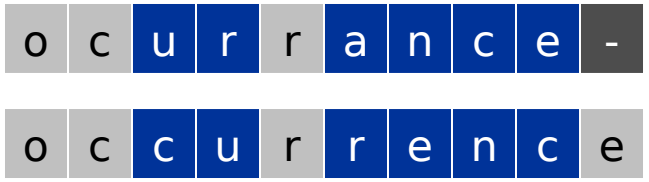
Esercizio 27 cap. 6: Soluzione

Possiamo scrivere un algoritmo iterativo che computa le soluzioni a partire da d = n fino a d=1 e memorizza le soluzioni in un array

```
Input: n, L, g1,…,gn
A[d,f]  //contiene la somma dei gi per i=d,…,f
S[d,f]   //contiene la somma dei gi(i-d)per i=d,…f
M[d] //contiene soluzione ottima da d ad n

For d = 1 to n
  A[d,d]=gd
  S[d,d]=0

For d = n to 1{
   min= large_value                          O(n2)
   For  x=d to n  {     //x=f-1
      If A[d,x-1]+gx<=L{
        A[d,x]=A[d,x-1]+gx
        S[d,x] =S[d,x-1] + (x-d)gx
        If P+S[d,x]+ M[x+1]<min
           min= P+c*S[d,x]+ M[x+1]
        M[d]=min} //fine if esterno
      Else  break }//ha computato l'ottimo per giorni da d ad n
  } //fine for esterno
return M[1]
```

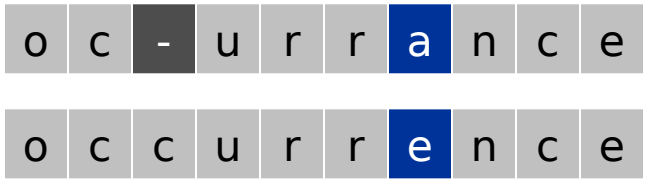
Allineamento di sequenze

- Quanto sono simili le due stringhe seguenti ?
  - **ocurrance**
  - **occurrence**



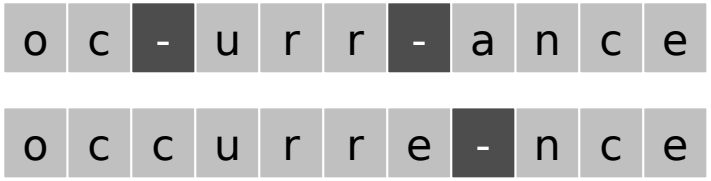
6 mismatch, 1 gap

- Allineamo i caratteri
- Ci sono diversi modi per fare questo allineamento



1 mismatch, 1 gap

- Qual e` migliore?

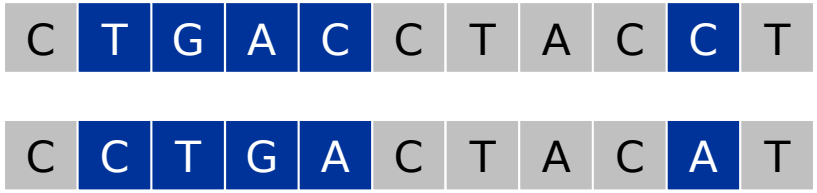


0 mismatch, 3 gap

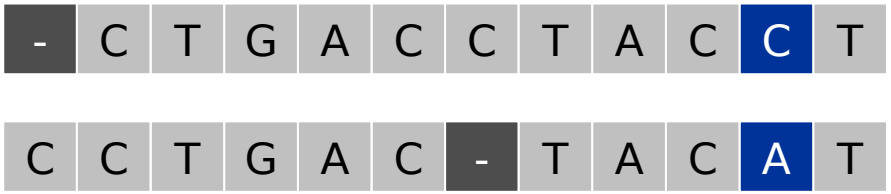
Edit Distance

- Applicazioni.
  - Base per il comando Unix diff.
  - Riconoscimento del linguaggio.
  - Biologia computazionale.

- Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]
  - Gap penalty δ;
  - Mismatch penalty α_{pq}. Si assume $\alpha_{pp}=0$
  - Costo = somma delle due penalita`



$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$



$2\delta + \alpha_{CA}$

Applicazione del problema dell'allineamento di sequenze

- I problemi su stringhe sorgono naturalmente in biologia: il genoma di un organismo e` suddiviso in molecole di DNA chiamate cromosomi, ciascuno dei quali serve come dispositivo di immagazzinamento chimico.

- Di fatto, si puo` pensare ad esso come ad un enorme nastro contenente una stringa sull'alfabeto $\{A,C,G,T\}$. La stringa di simboli codifica le istruzioni per costruire molecole di proteine: usando un meccanismo chimico per leggere porzioni di cromosomi, una cellula puo` costruire proteine che controllano il suo metabolismo.

Continua nella
prossima slide

Applicazione del problema dell'allineamento di sequenze

- Perche` le somiglianze tra stringhe sono rilevanti in questo scenario?
- Le sequenze di simboli nel genoma di un organismo determinano le proprieta` dell'organismo.
- Esempio. Supponiamo di avere due ceppi di batteri X e Y che sono strettamente connessi dal punto di vista evolutivo.
- Supponiamo di aver determinato che una certa sottostringa nel DNA di X sia la codifica di una certa tossina.
- Se scopriamo una sottostringa molto simile nel DNA di Y, possiamo ipotizzare che questa porzione del DNA di Y codifichi un tipo di tossina molto simile a quella codificata nel DNA di X.
- Esperimenti possono quindi essere effettuati per convalidare questa ipotesi.
- Questo e` un tipico esempio di come la computazione venga usata in biologia computazionale per prendere decisioni circa gli esperimenti biologici.

Allineamento di sequenze

- Abbiamo bisogno di un modo per allineare i caratteri di due stringhe
  - Formiamo un insieme di coppie di caratteri, dove ciascuna coppia e` formata da un carattere della prima stringa e uno della seconda stringa.

- Def. insieme di coppie e` un matching se ogni elemento appartiene ad una sola coppia

- Def. Le coppie (x_i,y_j) e $(x_{i'},y_{j'})$ si incrociano se $i < i'$ ma $j > j'$.

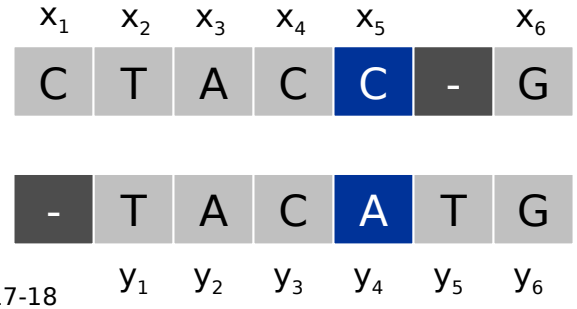
Def. Un allineamento M e` un insieme di coppie (x_i,y_j) tali che

- M e` un matching
- M non contiene coppie che si incrociano

Esempio: CTACCG vs. TACATG

Soluzione:

$M = (x_2,y_1), (x_3,y_2), (x_4,y_3), (x_5,y_4), (x_6,y_6)$.



Allineamento di sequenze

- Obiettivo: Date due stringhe $X = x_1\, x_2 \ldots x_m$ e $Y = y_1\, y_2 \ldots y_n$ trova l'allineamento di minimo costo.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

- Affermazione.
- Dato un allineamento M di due stringhe $X = x_1\, x_2 \ldots x_m$ e $Y = y_1\, y_2 \ldots y_n$, se in M non c'e` la coppia $(x_m, y_n)$ allora o $x_m$ non e` accoppiato in M o y_n non e` accoppiato in M.
- Dim. Supponiamo che x_m e y_n sono entrambi accoppiati ma non tra di loro. Supponiamo che x_m sia accoppiato con y_j e y_n sia accoppiato con x_i. In altre parole M contiene le coppie (x_m, y_j) e (x_i, y_n). Siccome $i<m$ ma $n>j$ allora si ha un incrocio e cio` contraddice il fatto che M e` allineamento.

Allineamento di sequenze: struttura del problema

- Def. OPT(i, j) = costo dell'allineamento ottimo (di costo minimo) delle due stringhe $x_1\ x_2\ \ldots\ x_i$ and $y_1\ y_2\ \ldots\ y_j$.
  - Caso 1: in OPT x_i e y_j sono accoppiati.
    - OPT(i, j) = Costo dell'eventuale mismatch tra x_i e y_j + costo dell'allineamento ottimo di $x_1\ x_2\ \ldots\ x_{i-1}$ and $y_1\ y_2\ \ldots\ y_{j-1}$
  - Caso 2a: in OPT x_i non e` accoppiato.
    - OPT(i, j) = Costo del gap x_i + costo dell'allineamento ottimo di $x_1\ x_2\ \ldots\ x_{i-1}$ e $y_1\ y_2\ \ldots\ y_j$
  - Case 2b: in OPT l y_j non e` accoppiato.
    - OPT(i, j) = Costo del gap y_j + costo dell'allineamento ottimo di $x_1\ x_2\ \ldots\ x_i$ e $y_1\ y_2\ \ldots\ y_{j-1}$

$$OPT(i,j) = \begin{cases} j\delta & \text{se } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{altrimenti} \\ i\delta & \text{se } j = 0 \end{cases}$$

Allineamento di sequenze: algoritmo

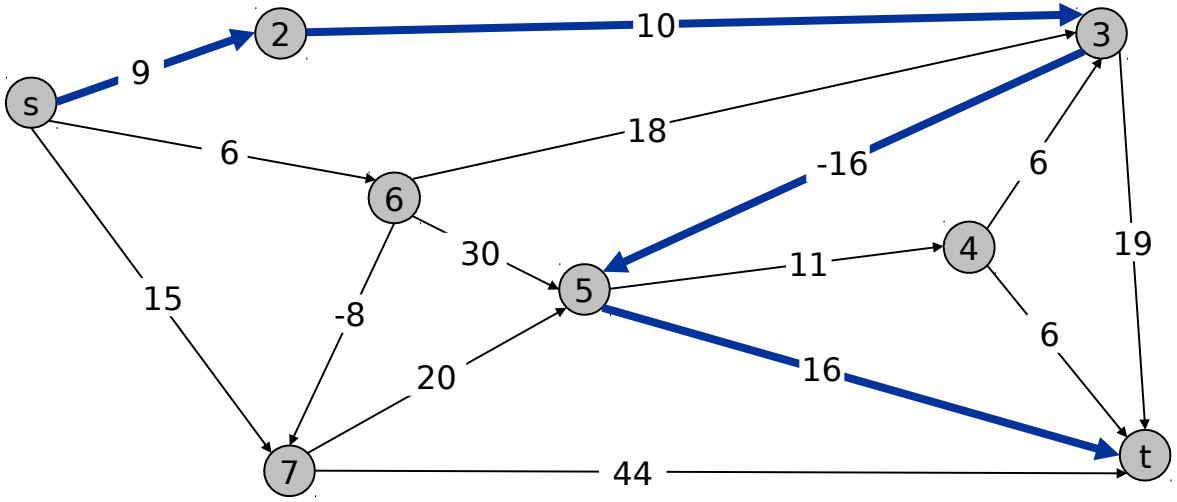
```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[i, 0] = iδ
   for j = 0 to n
      M[0, j] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

- Analisi. Tempo e spazio $\Theta(mn)$.
- Parole inglesi: m, n $\leq$ 10.
- Applicazioni di biologia computazionale: m = n = 100,000.
- Quindi mxn=10 miliardi . OK per il tempo ma non per lo spazio (10GB)

Cammini minimi

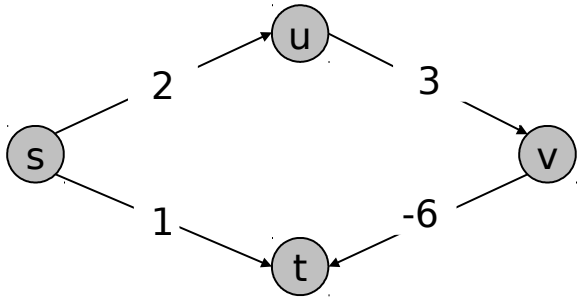
- Problema del percorso piu` corto. Dato un grafo direzionato G = (V, E), con pesi degli archi $c_{vw}$, trovare il percorso piu` corto da s a t.

- Esempio. I nodi rappresentano agenti finanziari e c_{vw} e` il costo di una transazione che consiste nel comprare dall'agente v e vendere immediatamente a w.

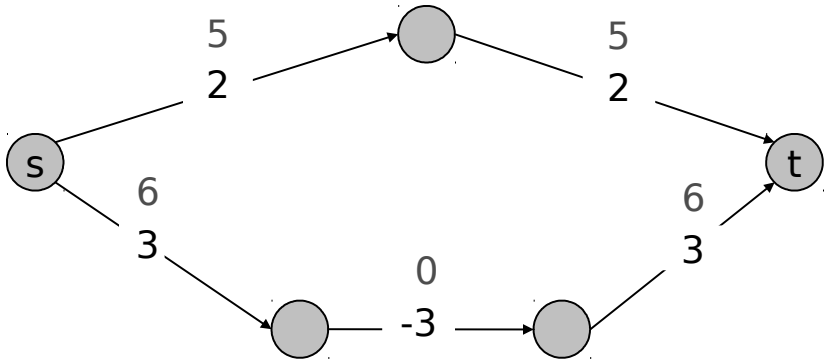


Cammini minimi in presenza di archi con costo negativo

- Dijkstra. Puo` fallire se ci sono archi di costo negativo

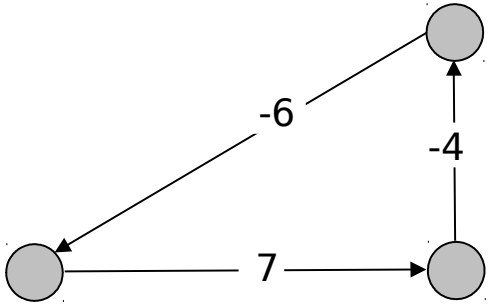


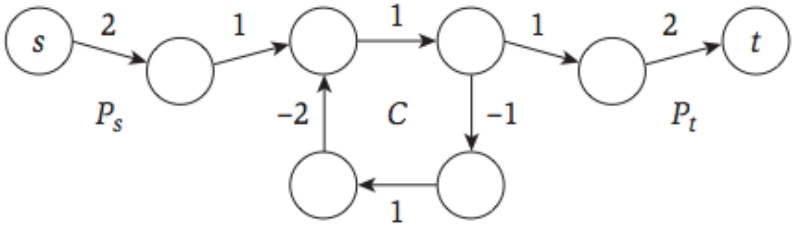
- Re-weighting. Aggiungere una costante positiva ai pesi degli archi potrebbe non funzionare.



Cammini minimi in presenza di archi con costo negativo

- Ciclo di costo negativo.

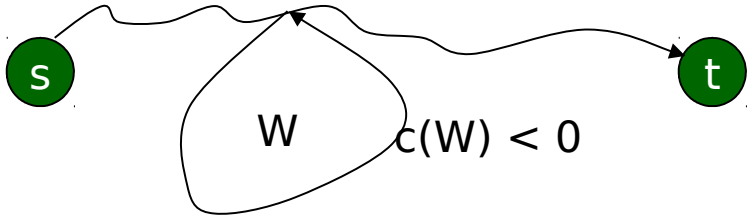




Cammini minimi in presenza di archi con costo negativo

Osservazione. Se qualche percorso da s a t contiene un ciclo di costo negativo allora non esiste un percorso minimo da s a t. In caso contrario esiste un percorso minimo da s a t che e` semplice (nessun nodo compare due volte sul percorso).

- Dim. Sia P un percorso da s a t con un ciclo C di costo negativo –c. Ogni volta che attraversiamo il ciclo riduciamo il costo del percorso di un valore pari a c. Cio` rende impossibile definire il costo del percorso minimo perche` dato un percorso riusciamo sempre a trovarne uno di costo minore attraversando il ciclo C . Sia ora P un percorso da s a t privo di cicli di costo negativo. Supponiamo che un certo vertice v appaia almeno due volte in P. C'e` quindi in P un ciclo che contiene v e che per l' ipotesi deve avere costo **non negativo**. In questo caso potremmo rimuovere le porzioni di P tra due occorrenze consecutive di v in P senza far aumentare il costo del percorso.



Cammini minimi: Programmazione dinamica

Def. OPT(i, v) = lunghezza del cammino piu` corto P per andare da v a t che consiste di al piu` i archi

Per computare OPT(i,v) quando i>0 e v≠t, osserviamo che

- il percorso ottimo P deve contenere almeno un arco (che ha come origine v).
- se (v, w) e` il primo arco di P allora P e` formato da (v, w) e dal percorso piu` corto da w a t di al piu` i-1 archi

$$OPT(i,v) = \min_{(v,w)\in E} \left\{ OPT(i-1,\ w) + c_{vw} \right\}$$

$$\text{OPT(i,v)} = \begin{cases} 0 & \text{se v=t} \\ \infty & \text{se } i = 0 \text{ e } v \neq t \\ \min_{(v,w)\in E} \{OPT(i-1, w) + c_{vw}\} & \text{altrimenti} \end{cases}$$

Cammini minimi: Programmazione dinamica

$$OPT(i,v)=\begin{cases}0 & \text{se } v=t\\ \infty & \text{se } i=0 \text{ e } v\neq t\\ \min_{(v,w)\in E}\{OPT(i-1,w)+c_{vw}\} & \text{altrimenti}\end{cases}$$

Dove si usa l'osservazione di prima sul fatto che in assenza di cicli negativi il percorso minimo e` semplice?

Ecco dove….

Affermazione. Se non ci sono cicli di costo negativo allora OPT(n-1, v) = lunghezza del percorso piu` corto da v a t.

Dim. Dall'osservazione precedente se non ci sono cicli negativi allora il percorso piu` corto da v a t e`semplice e di conseguenza contiene al piu` n-1 archi

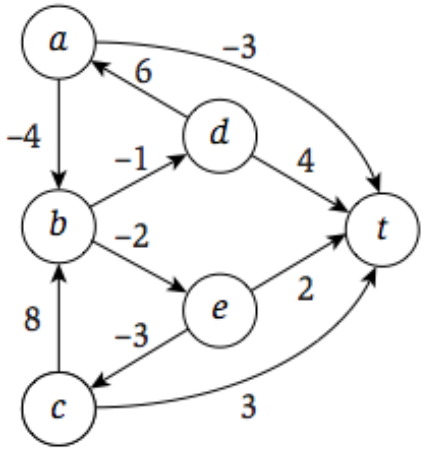
Algoritmo di Bellman-Ford per i cammini minimi

```
Shortest-Path(G, t) {
   foreach node v ∈ V
      M[0, v] ← ∞

   for i = 0 to n-1
      M[i, t] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← ∞
         foreach edge (v, w) ∈ E
            M[i, v] ← min {M[i,v], M[i-1, w] + c_vw }
}
```

- Analisi. Tempo $\Theta(mn)$, spazio $\Theta(n^2)$.

- Trovare il percorso minimo. Memorizzare un successore per ogni entrata della matrice

Bellman-Ford: esempio



(a)

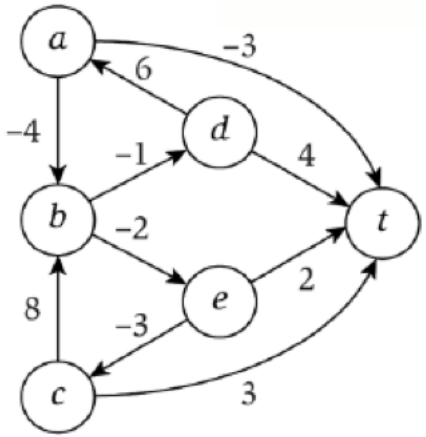
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

In questa tabella qui l'elemento di indice (u,i) corrisponde al valore M(i,u)

Bellman-Ford: esempio

$$OPT(i,v)=\begin{cases} 0 & \text{se v=t} \\ \infty & \text{se } i = 0 \text{ e } v \neq t \\ \min_{(v,w)\in E}\{OPT(i-1,w)+c_{vw}\} & \text{altrimenti} \end{cases}$$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

- OPT(5,a)=min{c_{ab}+OPT(4,b) , c_{at}+OPT(4,t)}=min{-4-2, -3+0}=-6
- OPT(4,b)=min{c_{bd}+OPT(3,d),c_{be}+OPT(3,e)}=min{-1+4,-2+0}=-2
- OPT(4,t)=0
- OPT(3,d)=min{c_{dt}+OPT(2,t),c_{da}+OPT(2,a)}={4+0, 6+3}=4
- OPT(3,e)=min{c_{ec}+ OPT(2,c), c_{et}+ OPT(2,t)}=min{-3+3, 2+0}= 0
- OPT(2,t)=OPT(1,t)=0
- OPT(2,c)={c_{cb}+ OPT(1,b), c_{ct}+ OPT(1,t)}=min {8+∞, 3+0}=3
- OPT(2,a)={c_{ab}+ OPT(1,b), c_{at}+ OPT(1,a)}=min {8+∞, 3+0}=3
- OPT(1,b)=min{c_{bd}+ OPT(0,d), c_{be}+ OPT(0,e)}={-1+∞ ,-2+∞}=∞
- OPT(1,a)=min{c_{ab}+ OPT(0,b), c_{at}+ OPT(0,t)}={-4+∞,-3+0}=-3
- OPT(0,d)=OPT(0,b)=OPT(0,e)=∞

Bellman-Ford: esempio

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

OPT(5,a)=min{c_{ab}+OPT(4,b) , c_{at}+OPT(4,t)}=min{-4-2, -3+0}=-6

OPT(4,b)=min{c_{bd}+OPT(3,d),c_{be}+OPT(3,e)}=min{-1+4,-2+0}=-2

OPT(4,t)=0

OPT(3,d)=min{c_{dt}+OPT(2,t),c_{da}+OPT(2,a)}={4+0, 6+3}=4

OPT(3,e)=min{c_{ec}+ OPT(2,c), c_{et}+ OPT(2,t)}=min{-3+3, 2+0}= 0

OPT(2,t)=OPT(1,t)=0

OPT(2,c)={c_{cb}+ OPT(1,b), c_{ct}+ OPT(1,t)}=min {8+∞, 3+0}=3

OPT(2,a)={c_{ab}+ OPT(1,b), c_{at}+ OPT(1,a)}=min {8+∞, 3+0}=3

- OPT(1,b)=min{c_{bd}+ OPT(0,d), c_{be}+ OPT(0,e)}={-1+∞ ,-2+∞}=∞
- OPT(1,a)=min{c_{ab}+ OPT(0,b), c_{at}+ OPT(0,t)}={-4+∞,-3+0}=-3
- OPT(0,d)=OPT(0,b)=OPT(0,e)=∞

Percorso ottimo da a a t: a b e c t

Ho ottenuto questo percorso partendo da M[a,5]=OPT(5,a) e osservando che e` uguale a $c_{ab}$+M[b,4] → il percorso e` formato da a e b seguiti dal percorso ottimo per andare da b a t attraversando al piu` 4 archi

Per ottenere il percorso ottimo per andare da b a t attraversando al piu` 4 archi osserviamo che M[b,4] e` uguale a c_{be}+M[e,3] → il percorso e` formato da b ed e seguiti dal percorso ottimo per andare da e a t attraversando al piu` 3 archi.

Per ottenere il percorso ottimo per andare da e a t attraversando al piu` 3 osserviamo che M[e,3] e` uguale a c_{ec}+M[c,2] → il percorso e` formato da e e c seguiti dal percorso ottimo per andare da c a t attraversando al piu` 2 archi

Per ottenere il percorso ottimo per andare da c a t attraversando al piu` 2 osserviamo che M[c,2] e` uguale a c_{ct}+M[t,1] → il percorso e` formato da c e t seguiti dal percorso ottimo per andare da t a t attraversando al piu` 1 arco. Ho finito perche' sono arrivata in t.

Miglioramento dell'algoritmo

- Usiamo un array unidimensionale M:
  - M[v] = percorso da v a t piu` corto che abbiamo trovato fino a questo momento

$$M[v] = \min(M[v], \min_{w \in V}(c_{vw} + M[w])).$$

- Scrivere un algoritmo che usa l'array M nel modo illustrato.

Miglioramento dell'algoritmo

- Teorema. Durante l'algoritmo, M[v] e` la lunghezza di un certo percorso da v a t, e dopo i aggiornamenti il valore di M[v] **non e` piu` grande della lunghezza del percorso da v a t che usa al piu` i archi**

- Conseguenze sullo spazio
  - Memoria: O(n).

  - Tempo:
  - il tempo e` sempre O(nm) nel caso pessimo pero` in pratica l'algoritmo si comporta meglio.
    - Possiamo interrompere le iterazioni non appena accade che durante una certa iterazione i nessun valore M[v] cambia

Computazione del cammino minimo

- Per ogni vertice v memorizziamo in first[v] il primo nodo che segue v lungo il percorso da v a t di costo M[v].
- First[v] viene aggiornato ogno volta che M[v] viene aggiornato. Se M[v] viene posto uguale a c_{vw}+M[w] allora si pone first[v]=w.

Minimum Coin Change Problem

- Dato un insieme di monete con valori $v_1<v_2<v_3<...<v_n$ e una banconota di valore V, fornire una formula per calcolare il minimo numero di monete richieste per cambiare la banconota. Assumiamo $v_1=1$ in modo che il problema ammetta una soluzione.

- Ad esempio: Banconota di 6 euro

  Valori monete: 1,2,4

- Possiamo cambiare la banconota in 5 modi:

- {1,1,1,1,1,1} {1,1,1,1,2}, {1,1,2,2},{1,1,4}, {2,4}

- La soluzione che include meno monete e` quindi {2,4}

Minimum Coin Change Problem

OPT(i,v)= minimo numero di monete di valore $v_1,\ldots,v_i$ per cambiare una banconota di valore v

- Se $v_i \leq v$ bisogna considerare sia il caso la soluzione include monete di valore v_i sia il caso in cui non le contiene:
  - Numero monete nella soluzione ottima tra quelle che contengono una moneta di valore v_i e` = 1+ numero monete nella soluzione ottima per l'importo $v-v_i$ quando si possono utilizzare monete di valore $v_1,\ldots,v_i$
  - Numero monete nella soluzione ottima tra quelle che non contengono una moneta di valore v_i e` = numero monete nella soluzione ottima per l'importo v quando si possono utilizzare monete di valore $v_1,\ldots,v_{i-1}$

    → $OPT(i,v)= \min\{OPT(i,v-v_i)+1, OPT(i-1,v)\}$

- Se v=0 allora OPT(i,v)=0 per ogni i
- Se i=1 allora OPT(i,v)=v per ogni v

Minimum Coin Change problem

MinCoinChange(d, n, V)

For i = 1 to n

M[i, 0] ← 0 //importo da cambiare = 0 → soluzione contiene 0 monete

For v = 1 to V

M[1, v] ← v //si possono usare solo monete da un euro

For i = 1 to n

For v= 1 to V

if $v < v_i$

then M[i, v] =M[i-1,v]

else

M[i, v] := min{1+M[i,v-v_i],M[i-1,v]}

Minimum Coin Change problem

Esercizio: scrivere l'algoritmo che costruisce la soluzione ottima del minimum change coin problem.

Coin change problem

- Dato un insieme infinito di monete C di n diversi valori v_1, …, v_n ed una certa banconota di valore V, fornire una strategia per trovare in quanti modi possiamo usare le monete in C per cambiare la banconota.

- Ad esempio: Banconota di 6 euro

  Valori monete: 1,2,4

- Possiamo cambiare la banconota in 5 modi:

- {1,1,1,1,1,1} {1,1,1,1,2}, {1,1,2,2},{1,1,4}, {2,4}

Coin change problem

N(i,v)=numero di modi in cui possiamo cambiare v con monete di valore $v_1,\ldots,v_i$

- Se $v_i \leq v$ allora la soluzione puo` includere o meno una moneta di valore v_i
    - Dobbiamo sommare il numero di soluzioni che includono monete di valore v_i al numero di soluzioni che non includono monete di valore v_i
    - $N(i,v)= N(i,v-v_i)+ N(i-1, v)$
- Se il valore v_i e` maggiore dell'importo da coprire allora
    - Le soluzioni possibili sono solo quelle che non includono monete di valore v_i
    - $N(i,v)=N(i-1, v)$

Sottosequenza comune piu` lunga

Def. Dati una sequenza di caratteri $x=x_1,x_2,\ldots,x_m$ ed un insieme di indici $\{k_1,k_2,\ldots,k_t\}$ tali che $1 \le k_1<k_2<\ldots<k_t \le m$, la sequenza formata dai caratteri di x in posizione $k_1,k_2,\ldots,k_t$ viene detta sottosequenza di x (N.B. I caratteri della sottosequenza non devono essere necessariamente consecuti in x).

Problema: Date due sequenze $x=x_1,x_2,\ldots,x_m$ e $y=y_1,\ldots,y_n$, vogliamo trovare la sottosequenza piu` lunga comune ad entrambe le sequenze.

Esempio: x=BACBDAB e y=BDCABA,
BCAB e` una sottosequenza comune a x e y di lunghezza massima.
I caratteri della sequenza BCAB appaiono nelle posizioni 1, 3, 6, 7 in x e nelle posizioni 1, 3, 4, 5 in y.
BDAB e` un'altra una sottosequenza comune a x e y di lunghezza massima.

Approccio brute force: Per ogni sottosequenza di x controlla se la sottosequenza compare in y.
Ci sono 2^m sottosequenze di x per cui l'algoritmo sarebbe esponenziale.

Sottosequenza comune piu` lunga

Input: $x=x_1,x_2,\ldots,x_m$ e $y=y_1,\ldots,y_n$

Sia OPT(i,j) la lunghezza della sottosequenza piu` lunga comune a $x_1,\ldots,x_i$ e $y_1,\ldots,y_j$.

Per calcolare OPT(i,j) consideriamo i 3 seguenti casi:

- Se $x_i = y_j$ allora la sottosequenza comune piu` lunga termina con x_i
  - In questo caso la soluzione ottima e` formata dalla sottosequenza piu` lunga comune a $x_1,\ldots,x_{i-1}$ e $y_1,\ldots,y_{j-1}$ sequita dal carattere $x_i = y_j$

- Se $x_i \neq y_j$ e la sottosequenza comune piu` lunga termina con un simbolo diverso da $x_i$ allora la soluzione ottima e` data dalla soluzione ottima per $x_1,\ldots,x_{i-1}$ e $y_1,\ldots,y_j$

- Se $x_i \neq y_j$ e la sottosequenza comune piu` lunga termina con un simbolo diverso da $y_j$ allora la soluzione ottima e` data dalla soluzione ottima per $x_1,\ldots,x_i$ e $y_1,\ldots,y_{j-1}$

Sottosequenza comune piu` lunga

Quindi si ha che per i>0 e j>0

$$OPT(i,j) = \begin{cases} OPT(i,j)=0 & \text{se } i=0 \text{ o } j=0 \\ OPT(i-1,j-1)+1 & \text{se } i>0,\ j>0 \text{ e } x_i = y_j \\ \max\{OPT(i-1,j), OPT(i,j-1)\} & \text{altrimenti} \end{cases}$$

Sottosequenza comune piu` lunga: algoritmo

```
ComputaLunghezzaLCS(X,Y,i,j)
1. m ← lunghezza di X
2. n ← lunghezza di Y
3. For i=1 to m
4.       M[i,0] ← 0
5.   For j=0 to n
6.       M[0,j] ← 0
7. For i=1 to m
8.      For j=0 to n
9.           If x_i=y
10.           Then M[i,j] ← 1+M[i-1,j-1]
11.                   b[i,j]="\"
12.          Else if  M[i-1,j]>M[i,j-1]
13.           Then M[i,j] ← M[i-1,j]
14.                   b[i,j]="↑"
15.          Else  M[i,j] ← M[i,j-1]
16.                   b[i,j]="← "
```

L'algoritmo oltre a computare i valori M[i,j]=OPT(i,j), memorizza nelle entrate b[i,j] della matrice b delle frecce in modo che successivamente la sottosequenza comune piu` lunga possa essere ricostruita agevolmente (si veda algoritmo nella slide successiva).

L’algortimo che stampa la sottosequenza comune piu` lunga

```
Stampa-LCS(b,X,i,j)
1. If i=0 or j=0
2.     Then return
3. If b[i,j]=”\”
4.   Then Stampa-LCS(b,X,i-1,j-1)
5.              print(x_i )
6. Else if b[i,j]=”↑”
7.    Then Stampa-LCS(b,X,i-1,j)
8. Else Stampa-LCS(i,j-1)
```