Programmazione dinamica (DP)

→ tecnica di risoluzione di problemi basata sulla memorizzazione tabellare

Ogni problema di DP necessita di due regole:

- regola ricorsiva (e.g. esplorare ricors. altre soluzioni)
- regola di riempimento della tabella (e.g. usare le altre regole per riempire la tabella)

Esempio (Knapsack problem) Si deve riempire uno zaino per ottenere il valore massimo sapendo che si può scegliere tra n oggetti, ciascuno con il proprio peso e valore — lo zaino ha una capacità max di peso

Creo una tabella V 2-dim dove

$V[i][j]$ ≐ "max valore in uno zaino di cap. j usando i primi $0, \dots, i$ oggetti"

Allora a ogni step:

```
V[i][J] = V[i-1][J],        "se lo prendo"
if (J ≥ W[i]) {                  6
    V[i][J] = max { V[i-1][J-W[i]] +
            + value[i], V[i-1][J] }
}
```

↳ "se non lo prendo"

Esempio (Longest common subsequence)

Si definisce sottoseq comune di due stringhe due sottostringhe (viste come array di char) ordinatamente uguali.

Usiamo ancora una tabella 2-dim. L dove:

$L[i][J]$ = "lunghezza sottoseq più lunga comune $A[0-i]$ e $B[0-J]$"

↝ $L[0][J] = L[i][0] = 0$ (confronto con ε)

Il passo induttivo è il seguente:

si itera su $i = 0 \dots m$ ($m = |A|$), $J = 0 \dots n$ ($n = |B|$)

- $A[i] = B[J] \rightsquigarrow L[i][J] = L[i-1][J-1] + 1$
- $A[i] \neq B[J] \rightsquigarrow L[i][J] = \max\{L[i-1][J], L[i][J-1]\}$

"aggiungi un char" → "ad A" o B"

→ Per stampare una LCS si può fare backtracking a partire da L:

- si va al contrario
- se $A[i] = B[J]$, si aggiunge $A[i]$ allo stack di char
  $i--$ $J--$
- se $A[i] \neq B[J]$:
  - se $C[i][J-1] = C[i][J]$ allora $B[J]$ non compare nella sottostringa e quindi $J--$
  - altrimenti $i--$ ($A[i]$ non compare nella sottostr.)

Tenendo conto di tutte le scelte si possono ottenere tutte le sottostr. più lunghe, ma richiede tempo esponenziale.

MST (Minimum - cost spanning tree)

→ spanning tree = albero di ricoprimento (G connesso)

Si dice SPANNING TREE di un grafo G un albero (V, T) t.c.: (G connesso (altrim non esiste))

a) $T \subseteq E$
b) (V, T) è ciclico
c) $|T| = |V| - 1$ (tocca tutti i nodi con le due cose ↑ un num minimo sono equiv. per connessione) di archi

→ esempi di spanning tree sono i DFS tree, i BFS tree e gli SPT.

→ il numero di spanning tree può essere molto grande. Considerando K_n — il grafo completo di n nodi — ci sono $\frac{n!}{n} = (n-1)!$ (a meno di rotazione) cammini spanning.

↝ il polinomio di Tutte contiene informazioni sul numero di spanning tree.

Si definisce COSTO DELLO SPANNING TREE T:

$$\sum_{(i,j)\in T} w(i,j)$$

(nel caso di G pesato)

L'obiettivo è trovare un MST: uno ST con costo minimo.

Algoritmo di Jarnik - Prim

↝ trova un MST per G non orient pesato; ispirato a Dijkstra

↝ esempio di algoritmo greedy

Al posto di fare una PQ sulle dist dalla radice, le si fa direttamente sui pesi (in qst senso è' greedy!

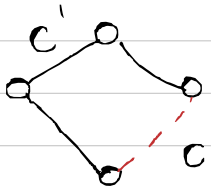
↝ $O(m \cdot \log n)$ — come Dijkstra

L'algoritmo di Jarnik-Prim si basa su due regole che forniscono cond. di ottimalità.

a) REGOLA DEL CICLO $(T = MST(G))$

Per ogni arco $c \in E \setminus T$, se c' è un qualsiasi arco che farebbe parte di un ciclo in $T \cup \{c\}$, allora $w(c) \geq w(c')$

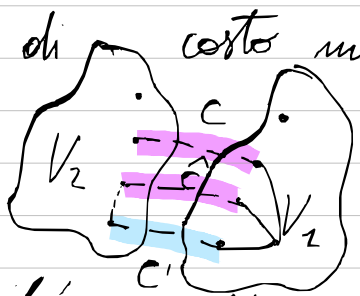
Altrimenti $T' = T \cup \{c\} \setminus \{c'\}$ avrebbe costo minore e T non sarebbe MST.



b) REGOLA DEL TAGLIO

Sia $E(V_1, V_2) = \{(i,j) \in E \mid i \in V_1, j \in V_2\}$ (con $V = V_1 \dot\cup V_2$) un qls taglio di G. Allora $\exists\, c \in T \cap E(V_1, V_2)$ t.c. $w(c) \leq w(c') \quad \forall\, c' \in E(V_1, V_2)$.

Supp. che per $c \in T \cap E(V_1, V_2)$ di costo minimo $\exists\, c' \in E(V_1, V_2)$ t.c. $w(c') < w(c)$. Se $\hat{c} \in T \cap E(V_1, V_2)$ è un arco che crea un ciclo in $T \cup \{c'\}$, allora — poiché $w(\hat{c}) \geq w(c)$ — $w(c') < w(\hat{c})$ e dunque $T \cup \{c'\} \setminus \{\hat{c}\}$ è un SP di costo minore di T, $\lightning$. Dunque c è l'arco desiderato.



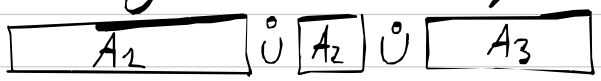
Algoritmo di Kruskal

→ algoritmo greedy per trovare un MST.

Per implementarlo serve una nuova struttura dati:

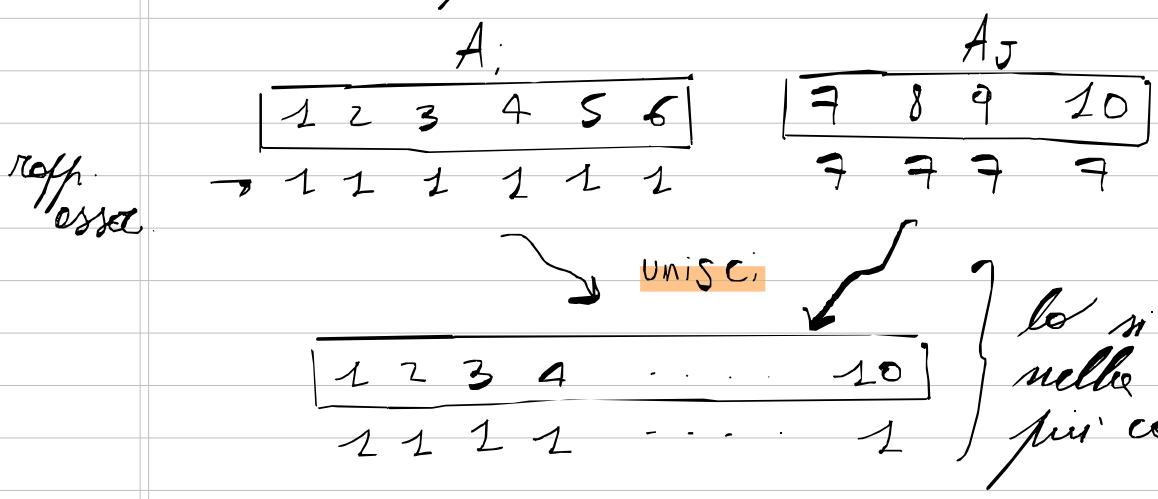
UNION-FIND

→ immagazzina le partiz. di un insieme



- unisci (A_i, A_j) — unisce due elem. della partiz.
- appartieni (A_i, A_j) — verifica se $A_i = A_j$

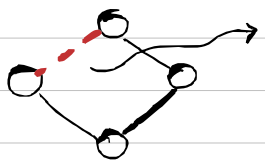
L'idea è quella di mantenere una lista di rappres. per ogni el. della partiz.:



Opportuni devo allora solo controllare che i rappr siano gli stessi ($O(1)$)

Idea L'algoritmo di Kruskal innanzitutto ordina gli archi per peso e poi aggiunge gli archi che non formeranno cicli al tree corrente

→ in un grafo non orientato aggiungere un arco (i,j) crea un ciclo se raggiungibili (i) = = raggiungibili (j).



→ creerebbe un ciclo

Quindi possiamo tenere traccia dei raggiungibili con una union-find e aggiungere (i,j) se appartiene (i,j) è poiso.

→ aggiunti $n-1$ archi è possibile già fermare l'algoritmo (l'albero è già di ricoprimento).

Prop. Vengono effettuati al più $n-1$ unioni e in totale richiedono $O(m \cdot \log(n))$ tempo.

Dim Che vengono effettuati al più $n-1$ unioni è ovvio — altrimenti non si otterrebbe un albero.

Un elem. cambia rappr. in unioni sì l'altro insieme ha taglia maggiore di quello a cui appartiene. Se quindi si sta facendo $S_u \cup S_v$ e $u \in S_v$, $|S_u| = x$, allora $|S_v| \geq x \rightsquigarrow |S_u \cup S_v| \geq 2x$

Per induzione (partendo dai singoletti e "raddoppiando" a ogni "unisci") si ricava alla fine che $2^i \leq n$ dove i è il n° di volte che si cambiano rappr. (ossia n° di unioni) $\rightsquigarrow i \leq \log_2(n)$.

per un singolo elemento

Quindi ricomponendo $O(\log(n))$ per tutti gli elem. si ottiene $O(n \cdot \log(n))$. ■

$\rightsquigarrow$ poiché il sorting iniziale richiede $O(|E| \log(|E|)) = O(|E| \log(|V|))$ e dunque Kruskal da questa $\frac{V}{2} \leq |E| \leq |V|^2$ siccome il grafo è conn. è dominato complessità