

Greedy

Applicazione: problemi di ottimizzazione

Approccio: calcolo della soluzione ottima attraverso una sequenza di scelte localmente ottime

Caratteristiche: semplici da scrivere ed efficienti

Questioni: dimostrare la correttezza di un algoritmo greedy e capire quali problemi sono affrontabili tramite questa strategia

Greedy Algorithmi

Soluzione ottima

Top-down

pochi sottoprobl. da risolvere

efficiente e semplice da scrivere

meno applicabile

Dynamic Programming

valore ottimo + soluzione ottima

bottom-up

tanti sottoproblemi da risolvere

meno efficiente e più complicato da scrivere

più applicabile

Algoritmo greedy: Algoritmo goloso: prendo sempre il meglio che passa per ogni iterazione.

Greedy (Input):

Soluzione = \emptyset

$O(1)$

<calcolo n parametri>

$O(n)$

ordina per parametro

$O(n \log n)$

for i=1 to n

$O(n)$

if Input_i può essere aggiunto

$S = S \cup \{Input_i\}$

$O(1)$

return S

Tempo algoritmo greedy: $O(n \log n)$

Matroidi

Sistema di indipendenza: una coppia (S, F) dove S insieme finito di elementi e F famiglia di sottinsiemi di S è un sistema di indipendenza se preso $A \in F$, allora in qualsiasi $B \subseteq A$ $\exists B' \in F$ tale che $|B'| = |A|$ e $B' \subseteq B$.

Matroide: struttura combinatoria su cui è associato un algoritmo greedy. Un S è matroide se per qualsiasi $A, B \in F$ t.c. $|B| = |A| + 1$ allora esiste un elemento $b \in B - A$ t.c. $A \cup \{b\} \in F$.

Matroide Grafico: Dato un grafo $G = (V, E)$ non orientato e连通的, $M_G = (S, F)$ con S insieme E degli archi e F tutti i sottinsiemi di S aciclici, è il matroide grafico di G .

Estensione: Dato $M = (S, F)$ matroide, $s \in S$ è detto estensione di $A \in F$ se $A \cup \{s\} \in F$.

Massimale: Dato $M = (S, F)$ matroide, $A \in F$ è massimale se non ha estensioni.

Matroide pesato: matroide $M = (S, F)$ a cui viene associata una funzione peso $W: S \rightarrow \mathbb{R}^+$.

Teorema di Rado: la coppia (S, F) è matroide sse per ogni funzione peso W , l'algoritmo greedy standard fornisce la soluzione ottima.

MST

Input: grafo connesso non orientato pesato $G = (V, E)$ con $W: E \rightarrow \mathbb{R}^+$

Output: $T \subseteq E$ aciclico t.c.

1. $\forall v \in V, \exists (u, v) \in E$
2. $W(T) = \sum_{(u, v) \in T} W(u, v)$ è minimo

Generic MST: algoritmo greedy.

$$A = \emptyset$$

while $A \neq \text{MST}$

(?) trova Arco (u, v) da aggiungere

$$A = A \cup \{u, v\}$$

return A

Osservazione: come faccio a sapere quale arco posso aggiungere?

Taglio: Partizione di V in V' e $V - V'$

Arco che attraversa il taglio: arco $(u, v) \in E$ t.c. u appartiene a V' e v appartiene a $V - V'$

Taglio che rispetta un insieme: un taglio rispetta insieme $A \subseteq E$ se nessun arco di A attraversa il taglio

Arco leggero: arco di peso minimo che attraversa il taglio

Teorema dell'arco sicuro: Dati un grafo connesso non orientato e pesato $G = (V, E)$, un sottoinsieme A dell'insieme T di archi di un MST e un qualsiasi taglio che rispetta A , allora un arco leggero (u, v) del taglio è sicuro per A , cioè $A \cup \{(u, v)\} \subseteq T$.

Idea algoritmo Kruskal: A è una foresta \rightarrow vertici sono presi da G (i vertici V)

Ad ogni passo aggiunge un arco sicuro $z = (u, v)$ dove z è un arco di peso minimo che collega due componenti distinte. (si può dimostrare che z è arco sicuro)

MST

Kruskal: Greedy + Insiemi disgiunti.

MST-Kruskal(G, W):

$$A = \emptyset$$

$O(1)$

$\forall v \in V \text{ makeSet}(v)$

$O(|V|)$

$E' = \text{sort}(\omega(E))$

$O(|V| \cdot \log |V|)$

$\forall z(u, v) \in E'$

$O(|E| \cdot \alpha)$

if $\text{findset}(u) \neq \text{findset}(v)$

$$A = A \cup \{(u, v)\}$$

$\text{Union}(u, v)$

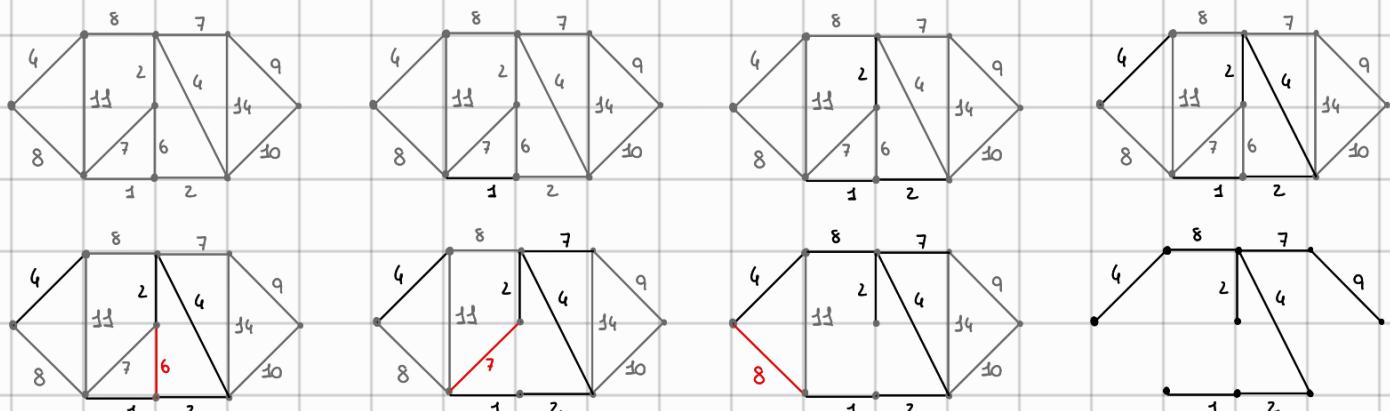
Return(A)

$O(1)$

$\text{findset} + \text{makeset} + \text{union}$

Tempo: $O(|E| \log |E|) + O(m \cdot \alpha)$ ma $m = |V| + |E|$ e $\alpha \leq \log |V|$, poiché G è connesso $|E| \geq |V| - 1$

$$= O(|E| \log |E|) + O(|E| \log |E|) = O(|E| \log |E|)$$



Idea algoritmo Prim: viene mantenuta una coda Q di min priority che all'inizio contiene tutti i vertici del grafo. Ad ogni passo Q:

- contiene i vertici che non appartengono alla componente C

- permette di estrarre vertice v tc (u, v) è l'arco di peso minimo che collega un vertice u in C con uno non ancora in C

Due campi per ogni vertice v del grafo:

v.Key: minimo valore del peso degli archi (u, v) incidenti in V (inizialmente ∞)

v.tc: vertice u tc (u, v) è l'arco di peso v.Key (inizialmente av 0)

MST

Iterazione Prim: ad ogni passo viene estratto da Q il vertice v con $\min v.Key$. L'arco (v, π, v) è un nuovo arco di MST e si aggiunge a C . Per ogni $v \in \text{Adj}(v)$, se $\text{lav} v.Key < W(v, v)$ si aggiornano $v.Key$ e $v.\pi$. L'algoritmo termina quando Q è vuoto.

Algoritmo Prim:

PRIM-MST(G, W, r)

foreach $v \in V$:

$v.Key = \infty$

$v.\pi = \text{NIL}$

$r.Key = 0$

aggiungi tutti i vertici di V alla coda con priorità Q .

while $Q \neq \emptyset$

$v = \text{estrai vertice da } Q$.

foreach $v \in \text{Adj}(v)$

if $v \in Q$ and $W(v, v) < v.Key$

$v.Key = W(v, v)$

$v.\pi = v$

$O(|V|)$

$O(|V|)$

$O(\log |V|)$

$O(|E|)$

$O(\log |V|)$

Tempo: $O(|V| \log |V|) + O(|E| \log |V|) \rightarrow O(|E| \log |E|)$

Dijkstra

Cammini minimi da sorgente unica: potremmo ottimizzare FW: non siamo interessati a tutti i cammini da tutte le sorgenti a tutte le destinazioni.

input: grafo $G = (V, E, W)$ orientato e pesato:

$W: E \rightarrow \mathbb{R}^+$ tc $w(i,j) = w_{ij}$ = peso dell'arco (i,j)

un vertice $s \in V$ vertice sorgente

output: $\forall v \in V \setminus \{s\}$, trovare il cammino di peso minimo che inizia con s e termina in v

Sottostruttura ottima: Se cammino $P = \langle v_1, \dots, v_K \rangle$ è minima, allora sono minimi tutti i sottocammini

$P_{ij} = \langle v_i, \dots, v_j \rangle$ tc $1 \leq i < j \leq K$. In particolare è minima il cammino il sottocammino $\langle v_1, \dots, v_{K-1} \rangle$ dove v_{K-1} è il predecessore di v_K nel cammino minimo P .

Limite superiore: dato un qualsiasi arco (u,v) si ha $\delta(s,v) \leq \delta(s,u) + w(u,v)$ dove:

$\delta(s,v)$ peso del cammino minimo da s al secondo vertice v dell'arco

$\delta(s,u)$ peso del cammino minimo da s al primo vertice u dell'arco

$w(u,v)$ peso dell'arco (u,v)

Tecnica del rilassamento: si aggiungono ad ogni vertice v i due attributi $v.d$ e $v.r$ tali che $v.d$ è limite superiore per $\delta(s,v)$ e $v.r$ vertice u tc $(u,v) \in E$.

Inizializzando $v.d = \infty$, $v.s = 0$ e $v.r = \text{NIL}$, terminiamo con $v.d = \delta(s,v)$ peso cammino minimo da s a v , $v.r = u$, predecessore di v nel cammino minimo da s a v .

Algoritmo di Dijkstra: una coda Q di min-priority contiene tutti i vertici del grafo

Ad ogni passo:

Q contiene i vertici v tc $v.d \neq \delta(s,v)$

Quando $v.d = \delta(s,v)$, viene eseguito il rilassamento di ogni arco (v,u) uscente da v .

l'algoritmo termina quando Q è vuota.

Alla fine tramite i valori dei campi dei predecessori si ricostruisce il cammino minimo da sorgente s ad un determinato vertice v .

Dijkstra

Init source(G, s) Relax(u, v, w)
 for $v \in V$ if $d(v) > d(u) + w(u, v)$
 $d(v) = \infty$ $d(v) = d(u) + w(u, v)$
 $\pi(v) = \text{NIL}$ $\pi(v) = u$
 $\pi(s) = \text{nil}$
 $d(s) = 0$

Dijkstra: $G = (V, E)$ $w: E \rightarrow \mathbb{R}^+$

init-source(G, s)

$\Theta(|V|)$

$S = \emptyset$

$H = v$ // metto tutti i vertici nella coda con priorità

$\Theta(|V|)$

while $H \neq \emptyset$

$u = \text{extract_min}(H)$

$O((|V| + |E|) \log |V|)$

$S = S \cup \{u\}$

for all $v \in \text{Adj}(u)$

$O(\log |V|)$

Relax(u, v, w)

BFS

- BFS: visita in ampiezza di un grafo non orientato G , a partire da una sorgente s .
- visita tutti e soli i vertici raggiungibili da s
 - ogni vertice viene visitato al più una volta
 - permette di trovare la distanza da s di tutti i vertici raggiungibili

Vertice: ogni vertice ha associato:

- $\text{color}[v] = \{\text{W}, \text{G}, \text{B}\}$
- $\text{distance}[v] = \{0, 1, \dots, \infty\}$
- $\pi[v] = \{\text{NIL}, s \in S\}$

Coda: un vertice viene inserito nella coda Q quando visitato. Quando tutti i suoi adiacenti sono visitati, diventa nero. Termina quando Q vuota.

Operazioni: 3 operazioni sulla coda Q :

- $\text{head}(Q)$ restituisce la testa
- $\text{enqueue}(Q, v)$ inserisce v in coda
- $\text{dequeue}(Q, v)$ elimina il vertice in testa.

Raggiungibile: $v \in V$ è raggiungibile da $u \in V$ sse \exists un cammino da u a v .

Cammino: un cammino da u a v è una sequenza finita di vertici u_0, u_1, \dots, u_k dove $u_0 = u$, $u_k = v$ e $V_i \in \{0, \dots, K-1\}$ $(u_i, u_{i+1}) \in E$

Distanza: la distanza di un vertice v da un vertice u è il numero di archi su un cammino minimo da u a v

BFS

Algoritmo BFS

 $BFS(G, s)$ $\forall v \in V \setminus \{s\}$ $color[v] = W$ $P[v] = NIL$ $d[v] = \infty$ $color[s] = G$ $P[s] = NIL$ $d[s] = 0$ $ENQUEUE(Q, s)$ while $Q \neq \emptyset$ $v = DEQUEUE(Q)$ $\forall v \in \text{Adj}[v]$ if $color[v] == W$ $color[v] = G$ $P[v] = v$ $d[v] = d[u] + 1$ $ENQUEUE(Q, v)$ $color[v] = B$ Tempo esecuzione: $O(|V| + |E|)$ BFS Tree: $T = (V_T, E_T)$ dove: $V_T = \{v \in V \mid color[v] = B\}$ $E_T = \{(\pi[v], v) \mid v \in V_T\}$

DFS: visita profondità di un grafo G.

sceglie un vertice s come sorgente

visita il 1° adiacente a s, poi il primo adiacente a questo ecc...

ogni volta che non ci sono adiacenti da visitare risale al predecessore

quando una sorgente non ha più vertici da visitare, si sceglie una nuova sorgente

la visita termina quando non ci sono più vertici disponibili.

Vertici: i vertici hanno, oltre al colore e π , come in BFS:

d = vettore tempi scoperto

f = vettore tempi completamento

Teorema delle parentesi: i tempi di scoperta non si accavallano. DFS di $G = (V, E) \rightarrow \forall u, v \in V$

$$A = [d[u], f[u]], B = [d[v], f[v]] \text{ allora } A \cap B = \emptyset \vee A \subseteq B \vee B \subseteq A$$

Classificazione degli archi:

W: Tree-edge: archi che appartengono alla foresta DFS. Lati che portano a scoprire nuovi vertici

G: back-edge: archi che non appartengono alla foresta DFS. Vanno da un vertice v ad un antenato di v nell'albero DFS

B: forward-edge: archi che non appartengono alla foresta DFS che vanno da un vertice v ad un suo successore

cross-edge: tutti gli altri archi, si trovano tramite teorema parentesi. (disgiunti)

Osservazione: B è valido solo se grafo orientato, altrimenti avremo solo T e B

DFS

Algoritmo DFS.

DFS(G)

for $v \in V$
 $\text{color}(v) = W$
 $p[v] = \text{NIL}$

$\text{time} = 0$

for all $v \in V$

if $\text{color}[v] == W$
DFS_VISIT(v)

DFS_VISIT(G, u)

$\text{time}++$
 $d[u] = \text{time}$
 $\text{color}[u] = G$
for all $w \in \text{Adj}[u]$
if $\text{color}[w] == W$
 $p[w] = u$

DFS_VISIT(G, w)

$\text{color}[u] = B$
 $\text{time}++$
 $f[u] = \text{time}$

ciclo adj

Tempo esecuzione: $O(|V| + |E|)$

Ordinamento topologico: Si consideri un grafo $G = (V, E)$ orientato aciclico. L'ordinamento topologico è: $T = \langle v_1, \dots, v_n \rangle$ tc ogni arco (v_i, v_j) si ha che $v_i < v_j$