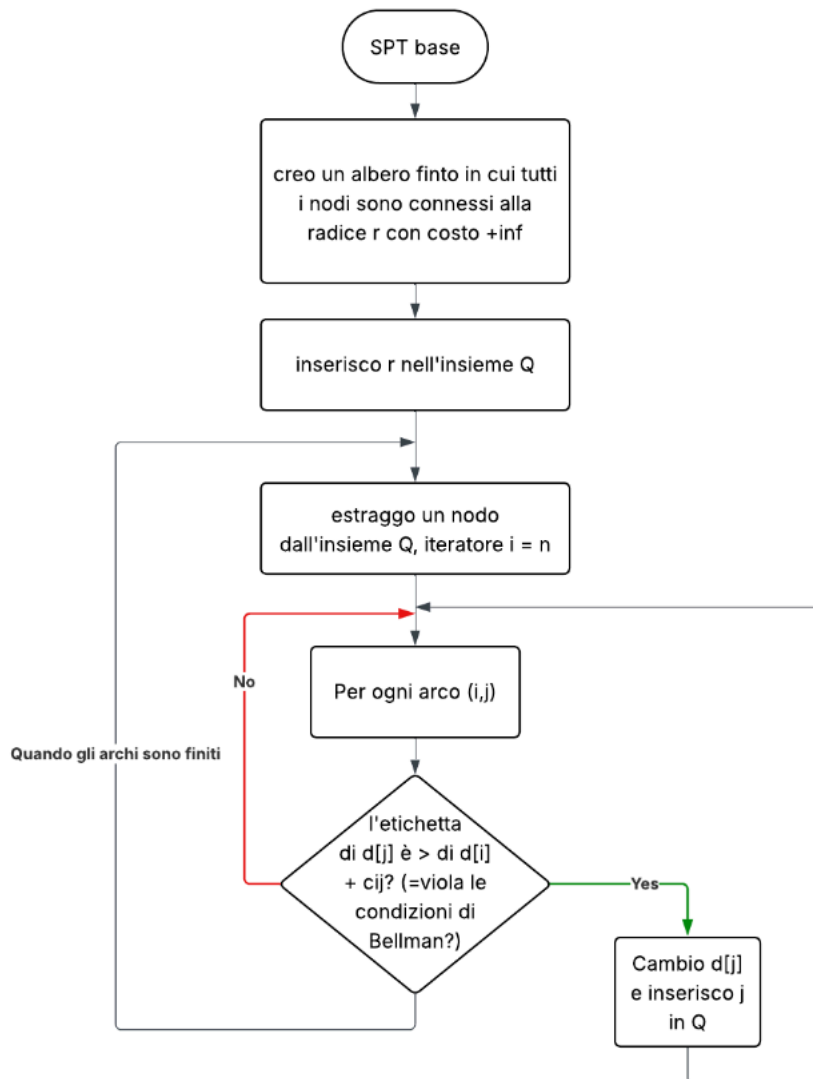


Secondo compito

SPT (Shortest Path Tree) - problema dell'albero di cammini di costo minimo



```

procedure (p, d) = SPT (G, c, r) {
  foreach( i ∈ N ) do { p[i] = r; d[i] = M; }
  d[r] = 0; Q = { r };
  do { select i from Q; Q = Q \ { i };
      foreach( (i, j) ∈ FS(i) ) do
        if( d[i] + cij < d[j] ) then {
          d[j] = d[i] + cij; p[j] = i; Q = Q ∪ { j };
        }
      } while( Q ≠ ∅ );
}
  
```

Complessità varianti SPT

Algoritmo	Implementazione di Q	Tipo di grafo	Complessità asintotica	Comportamento su grafi sparsi (m ≈ n)	Comportamento su grafi densi (m ≈ n ²)	Comportamento su cicli di costo negativo	Note
SPT.S	Q con coda di priorità ma lista	Costi non	O(n ²)	⚠ Non ottimale — ogni	✅ Ottimo su grafi densi	❌ Non corretto: SPT.S	Algoritmo basato sul

Algoritmo	Implementazione di q	Tipo di grafo	Complessità asintotica	Comportamento su grafi sparsi ($m \approx n$)	Comportamento su grafi densi ($m \approx n^2$)	Comportamento su cicli di costo negativo	Note
	non ordinata	negativi		estrazione costa $O(n)$, anche se pochi archi.		fallisce con archi negativi.	teorema di Dijkstra. Ogni estrazione costa $O(n)$.
SPT.S con heap binario	Coda di priorità (heap binario = albero binario)	Costi non negativi	$O(m \log n)$	✅ Eccellente su grafi sparsi — il $\log n$ compensa il basso numero di archi.	⚠️ Peggiora su grafi densi $\rightarrow (m \log n \approx n^2 \log n)$, quindi più lento della versione $O(n^2)$.	❌ Non corretto SPT.S fallisce con archi negativi.	Stesse note di SPT.S.
SPT.L.queue (Alg. Bellman)	Lista FIFO (coda)	Costi qualsiasi	$O(mn)$	⚠️ Accettabile se pochi archi e nessun ciclo negativo.	❌ Molto lento su grafi densi $\rightarrow (O(n^3))$.	⚠️ Può rilevare cicli negativi con opportuni controlli.	Se non esistono cicli orientati di costo negativo, nessun nodo può entrare in coda più di $n-1$ volte
SPT.L.stack	Lista LIFO (pila)	Costi qualsiasi	$O(mn)$	⚠️ Simile a L.queue — discreto su grafi piccoli e sparsi.	❌ Molto lento su grafi densi $\rightarrow (O(n^3))$.	⚠️ Può rilevare cicli negativi con opportuni controlli.	Stesso ordine della versione con coda.
SPT.L.deque	Lista doppia (deque)	Costi qualsiasi	$O(n \cdot 2^n)$ nel caso peggiore	✅ Buone prestazioni su grafi sparsi .	💀 Non usato: anche se teoricamente per creare il caso pessimo ci vogliono istanze specifiche e particolari	⚠️ Non adatto: cicli negativi possono causare loop infiniti. Abbiamo infatti perso la proprietà della coda	Al caso medio si comporta meglio degli altri algoritmi
SPT.L.2queue (doppia coda)	Lista con due code (Q' e Q'')	Costi qualsiasi	$O(mn^2)$	⚠️ Funziona ma lento , anche su grafi sparsi.	❌ Molto inefficiente su grafi densi $\rightarrow (O(n^4))$.	⚠️ Può rilevare cicli negativi ma lentamente.	Variante migliorativa in pratica ma non teoricamente efficiente.
SPT.Acyclic	Ordinamento topologico = non ha bisogno di Q perché segue in ordine crescente la numerazione dei nodi	Grafi aciclici	$O(m)$ e l'algoritmo di numerazione sempre $O(m)$	✅ Eccellente su grafi sparsi (lineare nel numero di archi).	✅ Eccellente nei grafi densi, resta lineare $\rightarrow (O(n^2))$ al massimo.	✅ Non ci sono cicli \rightarrow sempre corretto.	Esegue un solo passaggio sugli archi. Un grafo è aciclico \Leftrightarrow è ben numerato.

Legenda

m = numero di archi

n = numero di nodi

SPT.L \rightarrow L = lista di elementi non ordinati

SPT.S \rightarrow S = shortest, ordinati in base all'etichetta di ordine crescente.

Lemma del teorema di Dijkstra

(applicabile su SPT.S).

$\forall (i, j) \in A$, se $c_{ij} \geq 0$

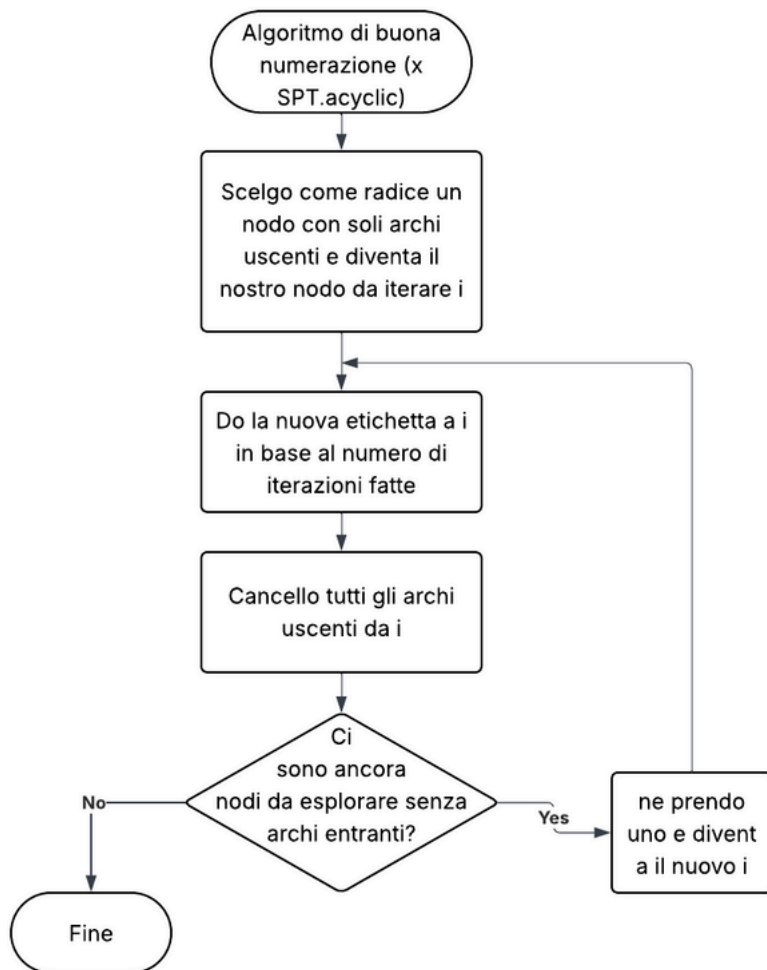
allora definisco i^k come iteratore degli elementi in Q e $d^k[\]$ come vettore delle etichette e $d^k[i^k] \leq d^{k+1}[i^{k+1}] \forall k$.

Traduzione: Le etichette degli elementi che escono da Q sono debolmente crescenti.

Teorema di Dijkstra

Ogni nodo con costi non negativi in Q esce dalla coda al più una volta.

Algoritmo di buona numerazione



Casi particolari

Problema del cammino di costo minimo

Noi non abbiamo bisogno dell'intero albero di cammini minimi ma abbiamo bisogno del cammino minimo da s a t. Allora possiamo usare l'algoritmo SPT.S e fermarci alla fine dell'esplorazione di t.

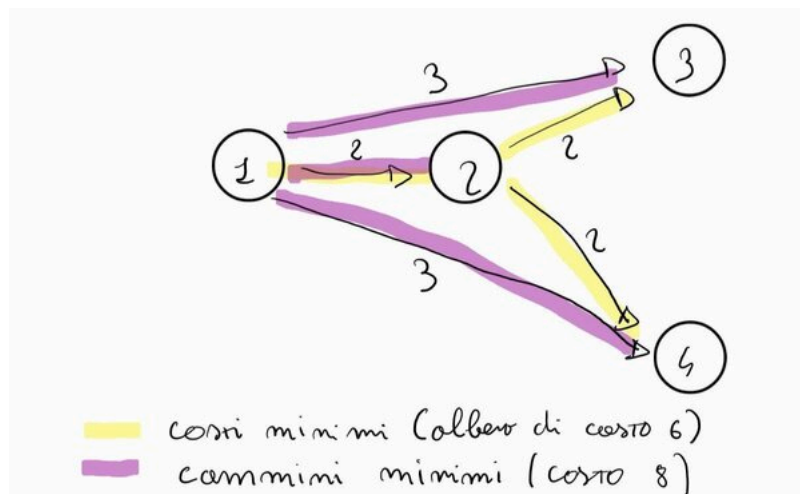
Problema dei cammini di costo minimo con più radici

Se abbiamo più candidati di radici, invece di fare una visita all'albero per ogni radice per capire quale radice ci conviene di più possiamo creare un nodo chiamato superradice che si collega a tutti i candidati con archi di costo zero e con una singola visita verrà fuori l'albero dei cammini di costo minimo.

Problema dell'albero di copertura di costo minimo

Si differenzia dal problema dei cammini perché in questo problema vogliamo trovare un albero che connetta tutti gli archi con minimo costo (quindi la somma degli archi presi di tutto l'albero), mentre il problema dei cammini vogliamo trovare il cammino più breve da s a t (e quindi la somma degli archi da s a t).

Esempio



Algoritmo Greedy-MST

Algoritmo per il problema dell'albero di copertura di costo minimo.

```

procedure  $S = Greedy-MST( G, c )$  {
   $S = \emptyset; R = \emptyset;$ 
  do applica Inserzione o Cancellazione
  while(  $S \cup R \subsetneq E$  and  $|S| < n - 1;$  )
}
```

L'algoritmo si basa sul prendere degli archi e controllare se sono da aggiungere nel nostro albero o rifiutarli. Abbiamo quindi due insiemi, uno per gli archi rifiutati, R, e uno per gli archi accettati, S.

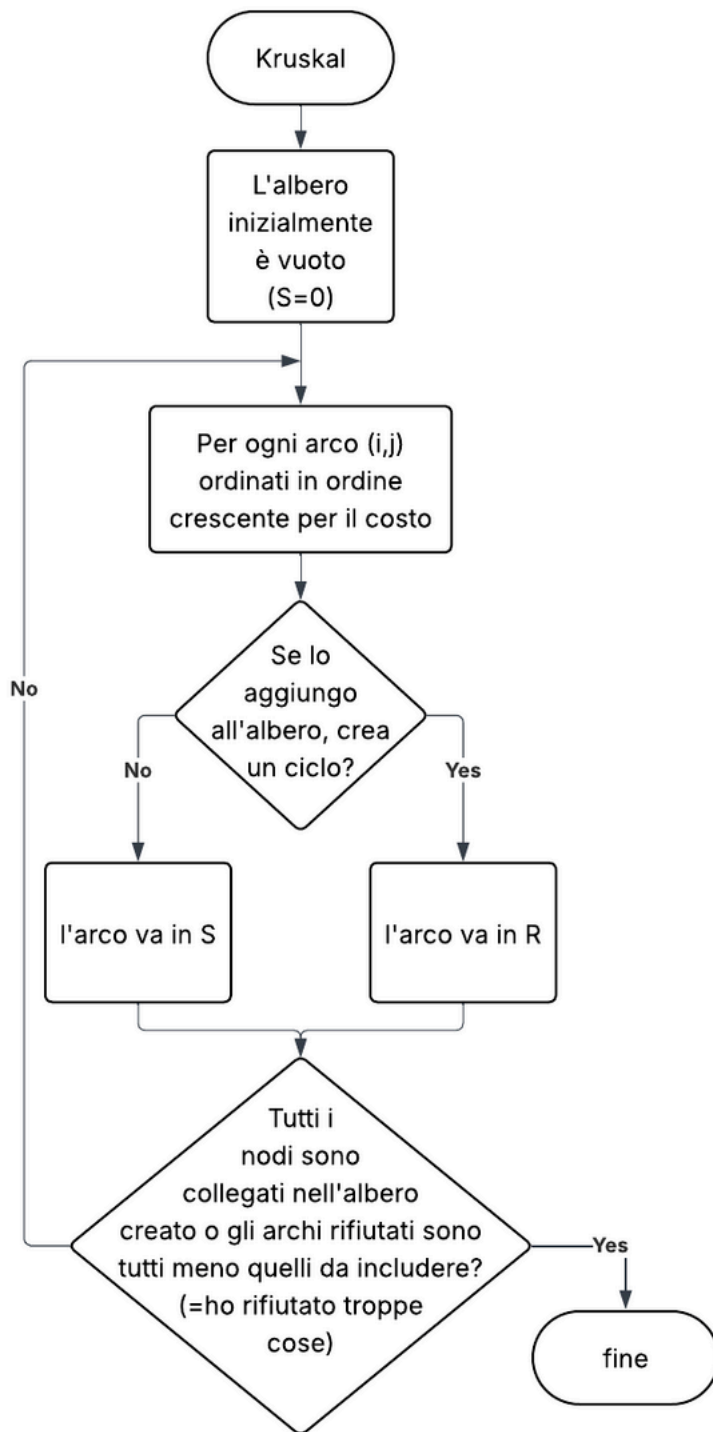
$$S \cap R = \emptyset$$

Teorema. La proprietà soprastante continua a valere per ogni iterazione e scelta effettuata dall'algoritmo.

Il do while termina quando o ho cancellato troppi archi e quelli restanti sono per forza dell'insieme di S, oppure ho trovato già l'albero e ho connesso tutti i nodi

L'algoritmo greedy ha una forma molto generica ed è per questo che da questa base nascono due sue implementazioni: l'algoritmo di Kruskal e l'algoritmo di Prim.

L'algoritmo di Kruskal

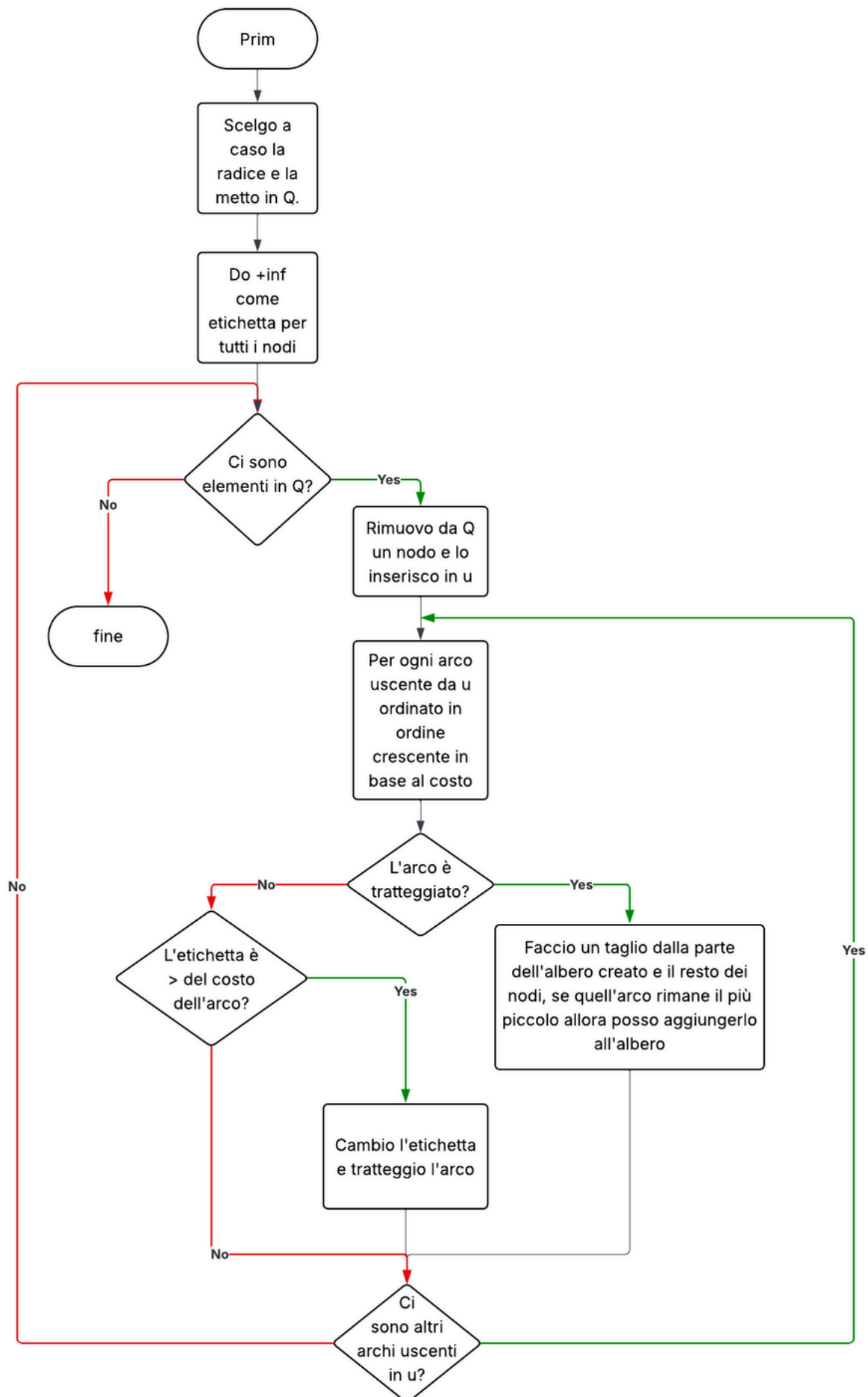


La sua complessità al caso pessimo è $\sigma(m * \log * m)$

```

procedure  $S = \text{Kruskal}(G, c)$  {
   $S = \emptyset$ ;  $R = \emptyset$ ;  $X = \text{Sort}(A)$ ;
  do { estrai da  $X$  il primo lato  $\{u, v\}$ ;
    if ( $\text{Component}(E, u, v)$ )
      then  $R = R \cup \{\{u, v\}\}$ ; /* cancellazione */
    else  $S = S \cup \{\{u, v\}\}$ ; /* inserzione */
  } while ( $|S| < n - 1$ );
}
  
```

Algoritmo di Prim



Q è una coda di priorità

L'algoritmo assomiglia a SPT-S

```

procedure  $p = \text{Prim}(G, c, r)$  {
  foreach ( $i \in V$ ) do {  $p[i] = r$ ;  $d[i] = M$ ; }
   $d[r] = -M$ ;  $Q = \{r\}$ ;
  do { seleziona  $u$  in  $Q$  tale che  $d[u] = \min\{d[j] : j \in Q\}$ ;
     $d[u] = -M$ ;  $Q = Q \setminus \{u\}$ ;
    foreach ( $\{u, v\} \in S(u)$ ) do
      if ( $c_{uv} < d[v]$ ) then {  $d[v] = c_{uv}$ ;  $p[v] = u$ ;
        if ( $v \notin Q$ ) then  $Q = Q \cup \{v\}$ ;
      }
  } while ( $Q \neq \emptyset$ );
}

```

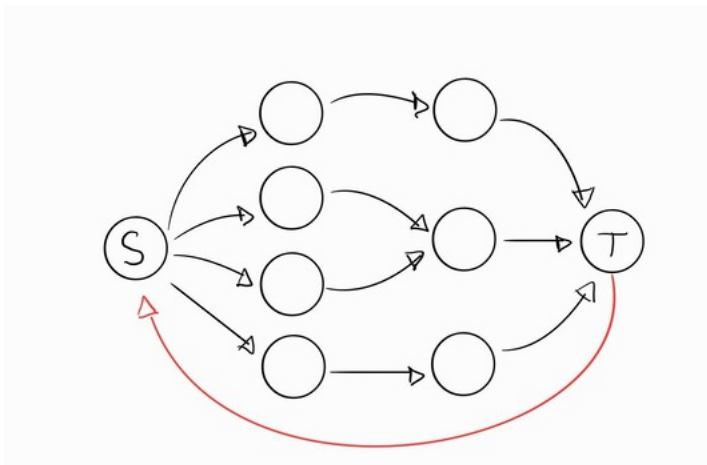
L'algoritmo di Prim può essere implementato in due modi esattamente come in SPT-S: heap-bilanciato e coda.

L'heap bilanciato ha la stessa complessità dell'algoritmo di Kruskal e quindi non cambia niente, mentre la coda ha al caso peggio $\sigma(n^2)$ rendendo l'algoritmo più efficace rispetto all'algoritmo di Kruskal in caso di grafi densi.

Problema di flusso massimo

E' un problema decisionale ("Decido quanto deve passare in un arco"), noi lo abbiamo trasformato in un problema di ottimizzazione, cioè "Quante unità di flusso riesco a mandare al massimo all'arco $x \rightarrow y$?"

Un sottoproblema del problema di flusso massimo è il **problema di circolazione**, in cui abbiamo tutti i nodi con deficit 0, e ogni unità di flusso che entra in un nodo, esce



Cammino aumentante

Un cammino aumentante è un cammino che va da S a T non orientato in cui viene inviato nel cammino quanto più flusso possibile.

A livello grafico si vede come una freccia al contrario dell'orientamento iniziale nel **grafico residuo**. Significa che ci sono degli archi non ancora saturi.

Per risolvere il problema del flusso massimo basta inviare quanto più flusso si può e verificare se la nostra soluzione è ottima cercando un **cammino aumentante**

Teorema

Se \exists un cammino aumentante allora la soluzione proposta non è ottima.

Cammini aumentanti in formule

Abbiamo un cammino aumentante P , uno scalare $\sigma \in \mathbb{R}$.

$x' = x \oplus \sigma P \rightarrow$ significa che partendo da un cammino, se esiste un cammino aumentante allora possiamo aggiungerci uno scalare θ e creare un cammino nuovo x' . Una volta terminato verifichiamo se x' è sol ottima

$0 \leq x'_{ij} \leq u_{ij} \rightarrow$ non stiamo aumentando il flusso oltre la sua capacità.

Definizione formale di Operazione di composizione tra flussi

$$X'_{ij} = \begin{cases} X_{ij} + \sigma & (i, j) \in P^+ \\ X_{ij} - \sigma & (i, j) \in P^- \\ X_{ij} & (i, j) \notin P \end{cases}$$

Dove P^+ è il cammino con archi concordi,

P^- è il cammino con archi discordi (ovvero orientati al contrario del grafo originale).

$\sigma(\bar{P}, x)$ = capacità del cammino rispetto al flusso, calcolata come la minima delle capacità residue concorde e discordi.

Teorema

\exists cammino aumentante $\Leftrightarrow \sigma(\bar{P}, x) > 0$

Algoritmo di risoluzione

L'algoritmo di risoluzione del problema di flusso massimo si basa sul creare il grafo residuo, se esiste un cammino da s a t, allora non è la soluzione ottima e posso aumentare il flusso.

Tagli saturi

Se abbiamo nel nostro grafo residuo un taglio saturo, ovvero un taglio che isola i nodi in due insiemi, allora quella è la soluzione ottima.

Questo quindi, risolve anche il problema in cui si vuole minimizzare la capacità del taglio da s a t per poterlo rendere saturo (problema del taglio saturo di costo minimo).

Complessità

Caso pessimo

$\sigma(n * m * v)$

v = massimo delle capacità sugli archi.

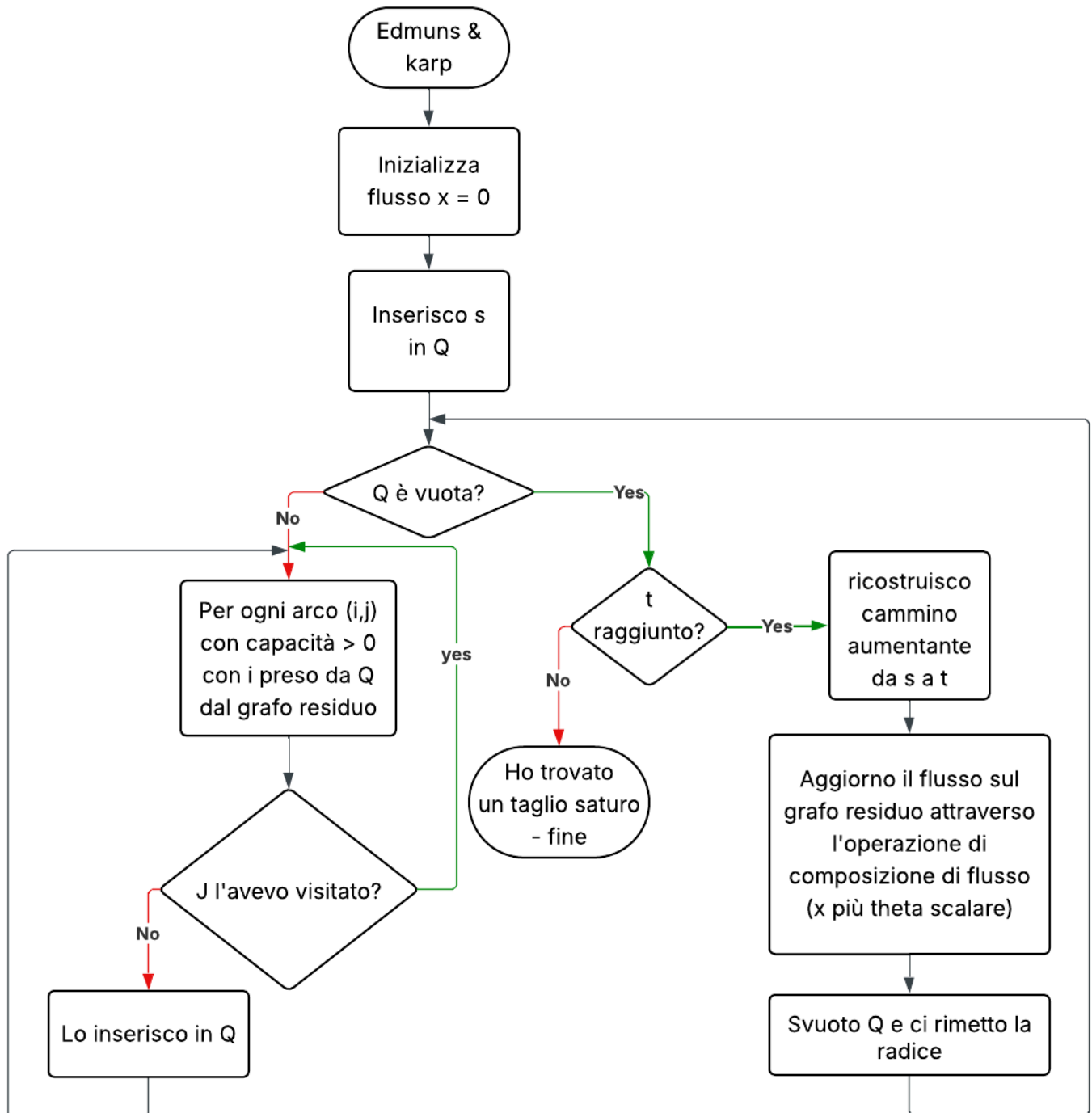
Complessità pseudopolinomiale, ovvero, se siamo sfortunati potremmo avere un algoritmo estremamente complesso anche su grafi piccoli

Caso medio

$\sigma(m^2 \log n)$

Algoritmo Edmuns & Karp

E' un algoritmo con complessità polinomiale $\sigma(n * m^2)$ che effettua una visita "a ventaglio" grazie all'implementazione di Q come una fila.



Particolarità algoritmo

- man mano che vado avanti trovo cammini sempre più lunghi
- complessità polinomiale $\sigma(n * m^2) \rightarrow$ lascia a desiderare nel caso di grafi densi ($\sigma(n^5)$)

Molteplici radici

Come nel problema di flusso minimo, se abbiamo più radici candidate basta connetterle a una super radice fittizia che stavolta avrà capacità $+\infty$ e uguale nel caso di più pozzi.

Problema di accoppiamento di massima cardinalità

Ciascun nodo dell'insieme di sotto può essere accoppiato con un solo nodo dell'insieme di sopra.

Può essere risolto come un problema di flusso massimo connettendo l'insieme di sotto a un nodo s e l'insieme di sopra a un nodo t, e tutti gli archi hanno capacità 1.

Ci convinciamo della correttezza della soluzione perché abbiamo connesso s a i nodi dell'insieme di sotto e uguale per t esattamente una volta.

Quindi, la complessità che sarebbe la pseudopolinomiale $\sigma(n * m * v)$ dato che tutti gli archi hanno capacità 1 diventa polinomiale ovvero $\sigma(n * m)$

Problema di flusso di costo minimo

Dobbiamo mandare tutto il flusso dalle sorgenti ai pozzi, spendendo il meno possibile.

Questo tipo di problemi potrebbe non avere soluzione dato che gli archi hanno una capacità max che potrebbe non essere sufficiente per fare defluire tutto il flusso.

Questi tipi di problemi hanno sorgenti e pozzi multiple, per sapere se esiste una soluzione ammissibile possiamo usare l'algoritmo dei flussi max applicato a più radici e più pozzi, ignorando i costi sugli archi.

Una volta che sappiamo che esiste una soluzione ammissibile, possiamo usare gli algoritmi fatti ad hoc per questo tipo di problema.

Pseudoflusso

Lo pseudoflusso è un vettore attaccato agli archi che assomiglia a un flusso ma che non per forza lo è. Esso infatti, rispetta i vincoli di capacità su ogni arco.

$$x = [x_{ij}]_{(i,j) \in A} \quad \text{dove } G = (N, A)$$

$$0 \leq x_{ij} \leq u_{ij} \quad \forall (i, j) \in A$$

Sbilanciamento

Lo sbilanciamento ϱ di uno pseudoflusso x del nodo i è:

$$\sum_{(j,i) \in BS(i)} x_{ji} - \sum_{(i,j) \in FS(i)} x_{ij} - b_i$$

ovvero: flusso entrante del nodo i - flusso uscente da i - deficit del nodo i

$\varrho_x(i)$:

se è 0 si dice **bilanciato**,

se è > 0 si dice **sorgente**,

se < 0 si dice **pozzo**.

$\varrho_{x'}(i)$:

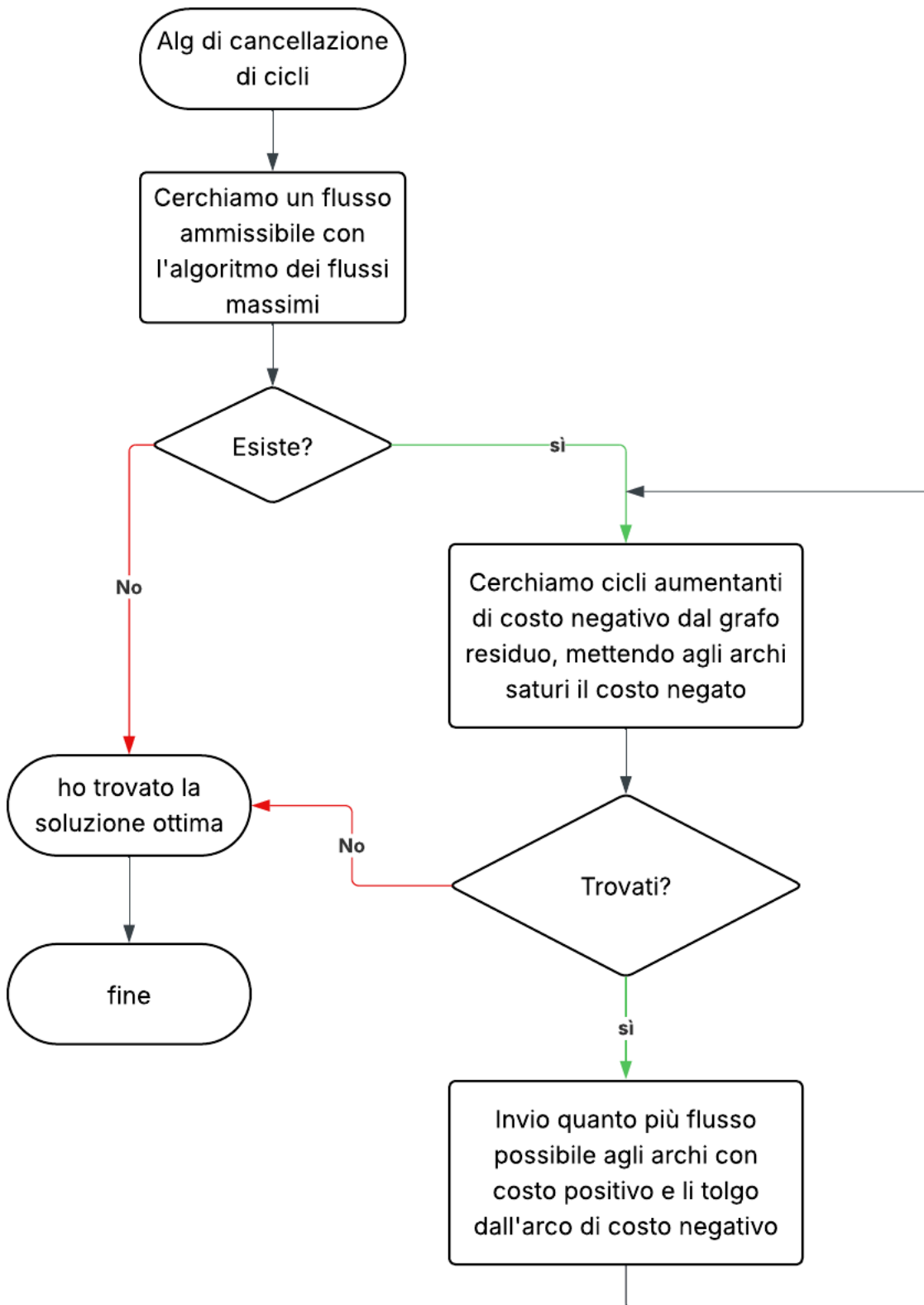
se è $\varrho_x(i) - \theta$ allora $i = s$,

se è $\varrho_x(i) + \theta$ allora $i = t$,

se è $\varrho_x(i)$ allora $i \notin \{s, t\}$.

Uno pseudoflusso se ha sbilanciamento 0 è un flusso.

Algoritmo di cancellazione di cicli



```

procedure (  $x, caso$  ) = Cancella-Cicli(  $G, c, b, u$  ) {
    (  $x, caso$  ) = Flusso-Ammissibile(  $G, b, u$  );
    if(  $caso == \text{"vuoto"}$  ) then return;
    for( ; ; ) {
        (  $C, \theta$  ) = Trova-Ciclo(  $G, c, u, x$  );
        if(  $\theta == 0$  ) then break; // CICLO NON TROVATO
         $x = \text{Cambia-Flusso}( x, C, \theta );$ 
    }
}

```

L'algoritmo termina se $u \in N$, b e c interi.

La complessità è pari a $\sigma(\bar{c} - \bar{c} * m * U * n * m)$ dove c sotto è il minimo dei costi, c sopra è il max dei costi e U è la massima capacità.

Teorema di composizione degli pseudoflussi

Dati due pseudoflussi x' e x'' sullo stesso grafo, dal primo pseudoflusso con un num di passaggi finito posso arrivare al secondo pseudoflusso in maniera polinomiale, usando i cammini per rettificare la differenza di sbilanciamento tra i due pseudoflussi.

Questo teorema serve per dimostrare la correttezza dell'algoritmo di cancellazione dei cicli

Corollario

X è un flusso ottimo $\Leftrightarrow \nexists$ cicli aumentanti di costo negativo

X è uno pseudoflusso minimale $\Leftrightarrow \nexists$ cicli aumentanti di costo negativo

Algoritmo dei cammini minimi successivi

Si cercano cammini aumentanti da s a t nel grafo residuo, più precisamente cammini minimi.

Non sempre è possibile risolvere il problema di flusso di costo minimo con questo algoritmo, perché non sempre esistono i cammini minimi \Leftrightarrow Non esistono quando esiste un ciclo orientato di costo negativo.

Lo pseudoflusso ottenuto con questa procedura non per forza rispecchia i vincoli di conservazione del flusso e viene chiamato **pseudoflusso minimale**.

In questo algoritmo quindi partiamo sempre non da un flusso ammissibile, ma da uno pseudoflusso minimale.

```

procedure (  $x, caso$  ) = Cammini-Minimi-Successivi(  $G, c, b, u$  ) {
     $x = \text{Inizializza}( c, u ); caso = \text{"ottimo"};$ 
    while(  $g(x) \neq 0$  ) do
        (  $p, d$  ) = Trova-Cammino-Minimo(  $G_x, O_x, D_x$  );
         $t = \text{argmin } \{ d[i] : i \in D_x \};$ 
        if(  $d[t] == \infty$  ) then {  $caso = \text{"vuoto"};$  break; }
         $x = \text{Aumenta-Flusso}( x, p, t );$ 
    }
}

```

Inizializza(c,u) = inizializza lo pseudoflusso rendendolo minimale.

