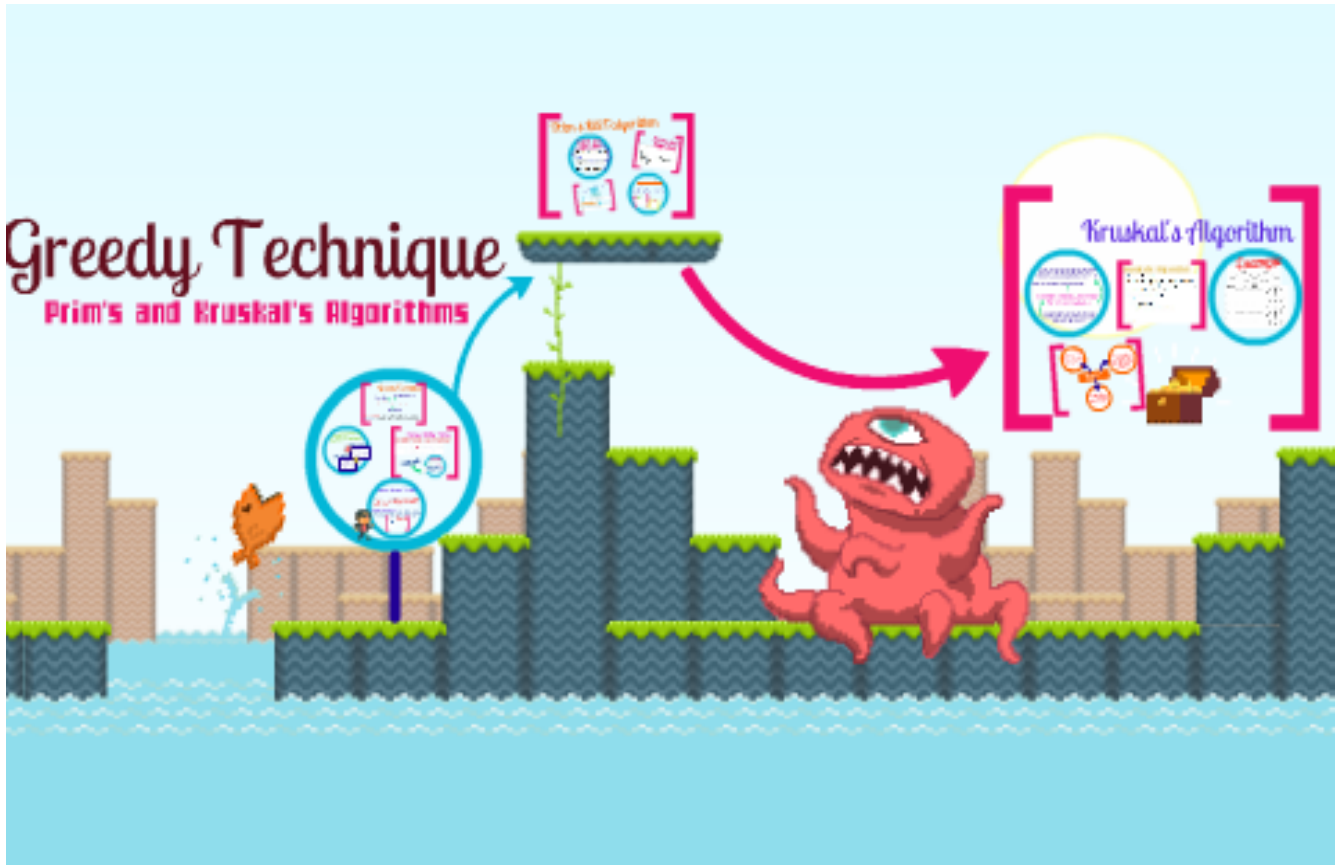
Algoritmi e strutture dati

Grafi e alberi di copertura minima



Menú di questa lezione

In questa lezione introduciamo il concetto copertura minima di un grafo pesato e indiretto, e vediamo due soluzioni al problema del calcolo dell'albero di copertura minima.

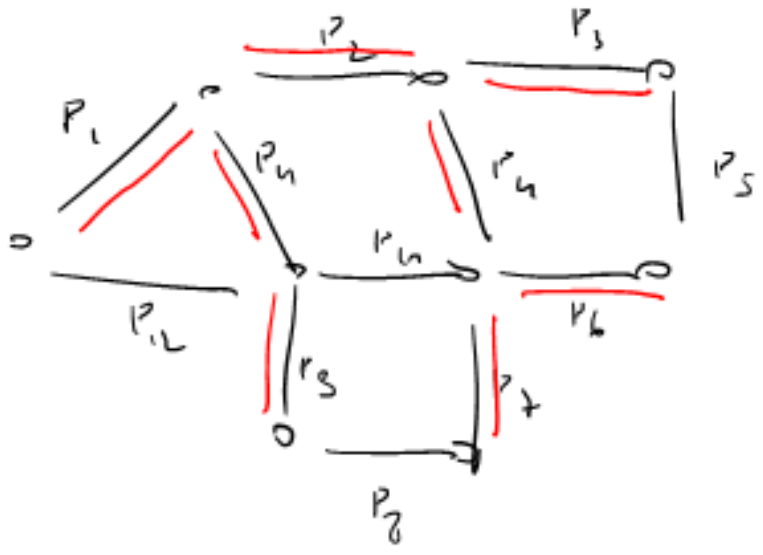
Alberi di copertura minimi

Ci concentriamo adesso sui grafi pesati indiretti connessi. Sia quindi $G = (V, E, W)$. Un grafo indiretto pesato può essere una buona rappresentazione di situazioni reali, come ad esempio una rete di connessioni informatiche tra computer, dove il peso di ogni arco rappresenta il costo della connessione. Un altro esempio può essere quello di una rete di distribuzione, dove i trasporti possono essere bidirezionali, ed il peso di ogni arco rappresenta il costo del trasporto. In una situazione come quelle descritte possiamo domandarci qual è il costo di visitare ogni vertice, ed, in particolare, se c'è una scelta di archi ottima, che minimizza il costo.

Alberi di copertura minimi

Definiamo un **albero di copertura minimo** (o **MST**) come un sottoinsieme di archi che forma un albero, copre tutti i vertici, e la cui somma dei pesi è minima. Quindi, così definito, è facile capire che calcolare un albero di copertura minimo di un grafo indiretto pesato è archeotipale di numerose applicazioni. Le più dirette ed immediate sono tutte casi particolari del problema generale della progettazione di una rete. Questa può essere telefonica, elettrica, idraulica, televisiva, di computer, oppure stradale, ed il problema della progettazione è sempre legato a trovare il costo minimo della rete che copre tutti i punti (nodi) che sono previsti.

Esempio di Coperture minime



Non è un albero indotto

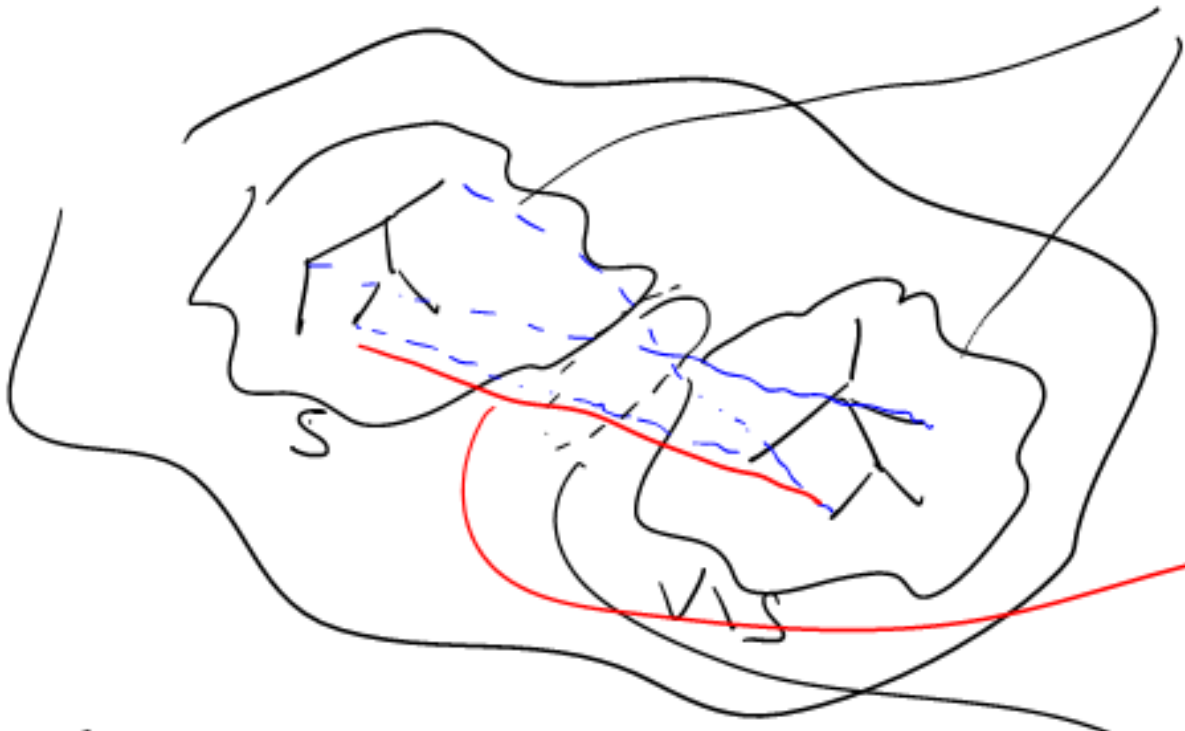
Alberi di copertura minimi

La strategia generale per risolvere questo problema è di tipo **greedy**.
Questo significa che ad ogni passo faremo la scelta **localmente** migliore, e vogliamo in questo modo ottenere il risultato **globalmente** migliore.
Questo che vediamo è un esempio, tra molti possibli, di applicazione di questa strategia di programmazione, che va vista, epistemologicamente, sullo stesso piano della strategia che abbiamo chiamato divide and conquer (introdotta con *MergeSort*).

Alberi di copertura minimi

Come abbiamo fatto in altre occasioni, prima di costruire un algoritmo per il calcolo dell'MST di un grafo indiretto pesato, studiamo alcune proprietà dello stesso che ci aiuteranno sia nella fase della progettazione che nella fase della dimostrazione di correttezza. Cominciamo con osservare che un MST A è un insieme di archi tali che, per definizione, formano un albero (indiretto) che tocca tutti i nodi del grafo. Quindi, eliminando qualunque arco da un albero di copertura (anche uno non minimo) si ottengono due alberi sconnessi (nel caso limite, uno dei due è composto da un solo nodo). In generale, qualunque partizione di V in due sottoinsiemi S e $V \setminus S$ viene chiamata **taglio del grafo** (semplicemente, **taglio**). Costruire un MST T significa considerare un taglio ed aggiungere progressivamente un arco, partendo dal taglio piú semplice che comprende un solo nodo in S (qualsiasi).

Concetto di taglio



Stessi punti all'esterno in entrambi

Arco che attraversa un punto

Arco sicuro

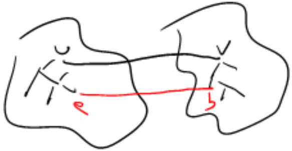
$(S, V \setminus S)$ è un taglio di G

Archi da attraversare per il taglio

IDEA: Scelgo sempre un arco sicuro tra quelli che attraversano un taglio

<u>Dim di correttezza</u>:

Supponiamo che T sia MST che contiene (u,v)



l'arco sicuro

chiamiamo (a,b) sicuro scelto.

Consideriamo l'albero $T' = (T \setminus \{(u,v)\}) \cup \{(a,b)\}$

T' è albero, copre tutti, e peso minore di T

$\Rightarrow$ T non è MST.

Alberi di copertura minimi

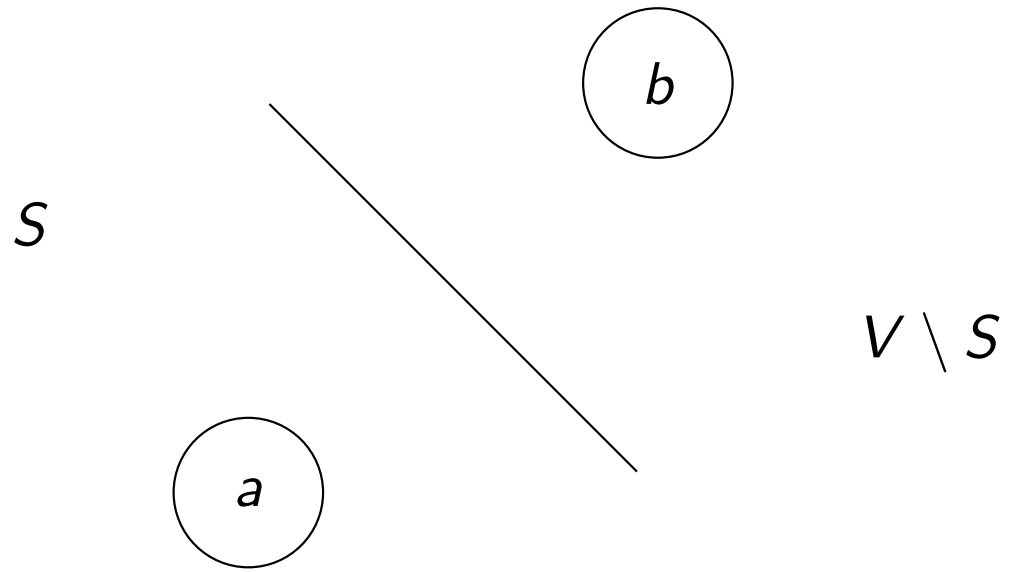


Figura: Rappresentazione di un taglio: a sinistra, un gruppo di nodi di G tra cui anche a, a destra tutti gli altri, tra cui anche b.

Alberi di copertura minimi

Consideriamo dunque un taglio qualsiasi $(S, V \setminus S)$ tale che per una certa coppia di nodi a, b, $a \in S$ e $b \notin S$. Poichè vogliamo costruire un MST, dovremo scegliere, in qualche momento, un arco che connetta qualche nodo di S con qualche nodo di $V \setminus S$: se non lo facessimo, non potremmo ottenere mai un MST (l'albero finale sarebbe sconnesso). Immaginiamo adesso che tra tutti gli archi con questa proprietà, l'arco (a, b) sia quello di peso minimo. Vogliamo mostrare che **l'arco** (a, b) **deve essere scelto**, cioè deve essere parte di qualche MST che copra G.

Alberi di copertura minimi

Per vedere ciò, immaginiamo di non scegliere (a, b) nella costruzione di un MST, per assurdo. Per la proprietà detta precedentemente, a fine operazioni ci sarà certamente un arco, chiamiamolo (u, v), che connette qualche nodo di S con qualche nodo di $V \setminus S$:

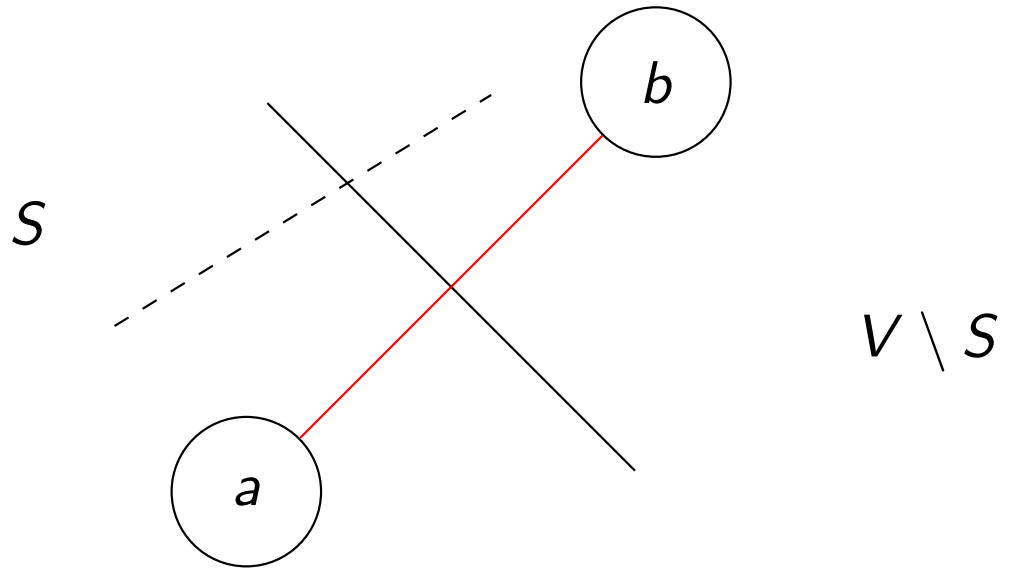


Figura: L'arco rosso è un arco di peso minimo che connette nodi di S con nodi di $V \setminus S$; l'arco tratteggiato è un altro con la stessa proprietà, ma non di peso minimo.

Alberi di copertura minimi

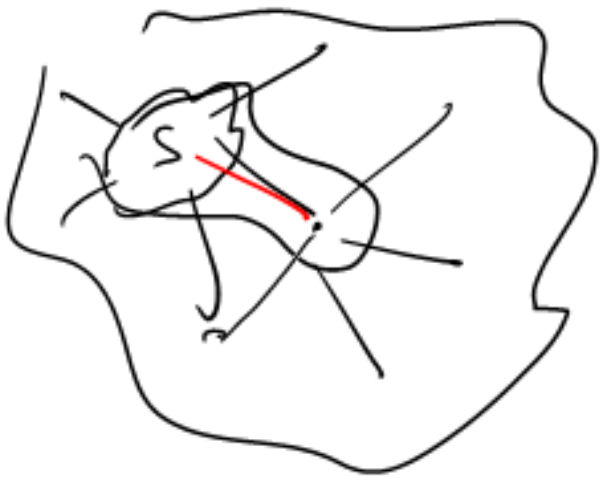
Sia T, nella nostra ipotesi assurda, l'MST ottenuto scegliendo (u, v) invece di (a, b). In questa situazione, possiamo costruire un nuovo albero, chiamiamolo T' semplicemente così: $T' = (T \setminus \{(u, v)\}) \cup \{(a, b)\}$. Chiaramente, se T era un albero di copertura, lo è anche T'; inoltre, T' pesa meno di T. Pertanto T' **non poteva essere** un MST. Abbiamo dimostrato la seguente proprietà: considerata qualunque situazione di costruzione di un MST, cioè qualunque taglio, il prossimo passo per la costruzione è scegliere sempre l'arco di peso minimo che lo attraversa. Chiamiamo questo arco **sicuro** (per il taglio).

Alberi di copertura minimi

La proprietà vista prima è, in pratica, un algoritmo di costruzione di un MST, che non rimane che codificare. Inoltre ci permette di fare le seguenti considerazioni. In prima istanza, se tutti i pesi di archi di G sono diversi tra loro, allora l'MST è unico: infatti, per ogni taglio ci sarebbe sempre una sola scelta. In maniera simile, se così non fosse, cioè se ci fossero archi di peso uguale, allora l'MST potrebbe non essere unico. L'algoritmo di Prim è un modo efficiente di codificare quanto visto.

Alberi di copertura minimi: algoritmo di Prim

L'idea alla base dell'algoritmo di Prim è quella di partire da un vertice qualsiasi, e, ad ogni passo, aggiungere un arco (ed un vertice) in modo che l'arco aggiunto sia un arco sicuro (come precedentemente definito). La corretta struttura dati in questo caso deve permettere di mantenere un insieme di vertici in maniera da poter facilmente individuare ed estrarre, tra questi, quello che comporta una spesa minima in termini del peso dell'arco che viene scelto per aggiungere quel vertice. È chiaro che abbiamo bisogno di una coda di priorità. Robert Prim pubblicò il suo algoritmo nel 1959, ma era già noto dal 1930, sembra.



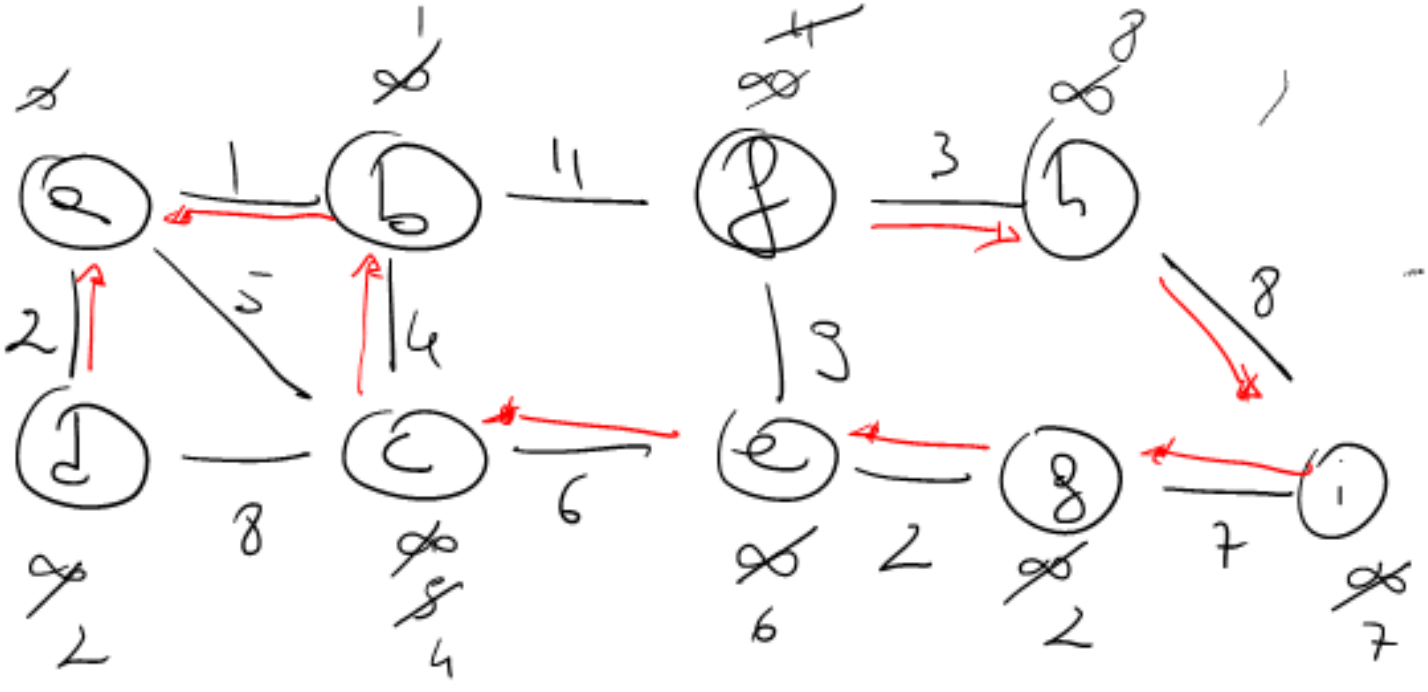
Alberi di copertura minimi: algoritmo di Prim

Ogni vertice di G viene arricchito con due campi: $v.key$ (il peso minimo, inizialmente ∞, tra gli archi che connettono qualche vertice di T con v), e $v.\pi$ (il padre di v, inzialmente *Nil*, nel MST risultante). Inizialmente tutti i vertici si trovano nella coda di prioritá Q semi-ordinata su $v.key$, dove, sempre inizialmente, tutti gli elementi sono a ∞. La radice r è data esplicitamente, e si cerca un MST radicato in r. Ad ogni scelta, i pesi degli elementi in Q (la coda di priorità) vengono modificati in base al principio che abbiamo spiegato, il vertice viene estratto da Q e inserito in T (T viene mantenuto in maniera virtuale: costruiamo l'albero modificando i predecessori $v.\pi$) e ripetiamo finché Q si svuota, e tutti i vertici sono stati coperti.

Alberi di copertura minimi: algoritmo di Prim

```
proc MST-Prim(G, w, r)
  for (v ∈ G.V)
    do
    { v.key = ∞
    { v.π = Nil
  r.key = 0
  Q = G.V
  while (Q ≠ ∅)
    do
    { u = ExtractMin(Q)
    { for (v ∈ G.Adj[u])
      do
      { if ((v ∈ Q) and (W(u, v) < v.key))
        do
        { v.π = u
        { v.key = W(u, v)
```

Esercizio 128:



Peso totale: 33

ID

Correttezza di *MST-Prim*

Per mostrare che *MST-Prim* è **corretto**, definiamo T come l'insieme di tutte le coppie $(v.\pi, v)$ tali che $v.\pi$ è definito e $v \notin Q$. Adesso mostriamo che vale la seguente **invariante**: T è sempre sottoinsieme di qualche MST.

- Nel **caso base**, T è vuoto e l'invariante è rispettata in maniera triviale (l'insieme vuoto è sottoinsieme di qualunque insieme, quindi anche ti tutti gli MST di G).
- Supponiamo adesso che l'invariante valga per T ad un certo punto della computazione (**caso induttivo**), e consideriamo l'insieme T' ottenuto dopo una esecuzione del ciclo **while**. Succede che $T' = T \cup \{(v.\pi, v)\}$, e v è il vertice tale che $v.key$ è il minore tra quelli ancora nella coda Q. Vogliamo mostrare che T' è ancora sottoinsieme di qualche MST. Sia S l'insieme di tutti e soli i vertici coperti da archi di T. Chiaramente, $(S, V \setminus S)$ è un taglio di G, e chiaramente, l'arco $(v.\pi, v)$ è un arco sicuro per il taglio. Pertanto, T' è ancora sottoinsieme di qualche MST.

Complessità di *MST-Prim*

Calcolare la **complessità** di *MST-Prim* non è del tutto banale, sebbene si tratti di un algoritmo iterativo. Il problema nasce dal computo delle operazioni sulla coda Q. Se assumiamo che i vertici siano denotati con interi per esempio da 1 a $|V|$, allora, come sappiamo, possiamo implementare la coda di priorità in almeno due modi diversi. Il primo consiste nel usare un semplice **array senza ordine**. Questo significa che la costruzione della coda costa $\Theta(|V|)$, l'estrazione del minimo $\Theta(|V|)$, e il decremento $\Theta(1)$. Immaginiamo che il grafo sia denso. Allora avremo: $\Theta(|V|)$ (inizializzazione), $\Theta(|V|)$ (costruzione della coda), $\Theta(|V|^2)$ (per le estrazioni del minimo, analisi aggregata), e $\Theta(|E|) = \Theta(|V|^2)$ (per i decrementi, analisi aggregata), per un totale di $\Theta(|V|^2)$. Osserviamo che, con questa implementazione, anche se il grafo fosse sparso, non cambierebbe nulla dal punto di vista asintotico.

Complessità di *MST-Prim*

In alternativa possiamo usare una **heap binaria** per implementare la coda, il che ci permetterà di avere un vantaggio, in caso di grafi sparsi. Avremo che la costruzione della coda costa sempre $\Theta(|V|)$, l'estrazione del minimo costa $\Theta(log(|V|))$, e il decremento costa $\Theta(log(|V|))$, se abbiamo reaizzato un'implementazione attenta. Quindi, con un grafo sparso, avremo: $\Theta(|V|)$ (inizializzazione), $\Theta(|V|)$ (costruzione della coda), $\Theta(|V| \cdot log(|V|))$ (per le estrazioni del minimo, analisi aggregata), e $\Theta(|E| \cdot log(|V|))$ (per i decrementi, analisi aggregata), per un totale di $\Theta(|E| \cdot log(|V|))$ (anche nei grafi sparsi ci sono più archi che vertici - altrimenti avremmo vertici sconnessi non coinvolti nel problema). Cosa accadrebbe con questa implementazione in caso di grafi densi?

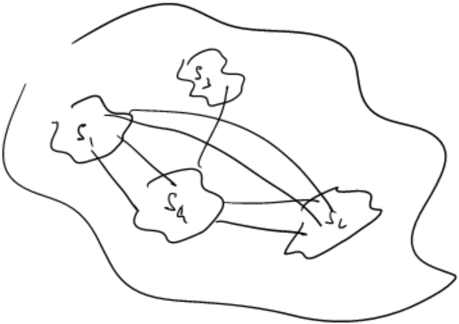
Complessità di *MST-Prim*

Il caso peggiore dunque si verifica con un grafo denso, e, per minimizzare il danno, si preferisce l'implementazione con una coda senza struttura. Poichè esistono modi diversi da quelli che noi abbiamo visto per implementare le code, non possiamo dare una risposta definitiva alla domanda sulla complessità del problema. I grafi si considerano non densi anche quando $|E|$ cresce rispetto a $|V|$ ma non si avvicina asintoticamente a $|V|^2$; anche in questo caso, la seconda scelta (heap binarie) è migliore della prima.

Alberi di copertura minimi: algoritmo di Kruskal

Una valida alternativa a *MST-Prim* è l'algoritmo noto come algoritmo di Kruskal, che utilizza una generalizzazione del concetto di taglio e del concetto di arco sicuro per il taglio al fine di ottenere un albero di copertura minimo. Joseph Kruskal lo propose nel 1956.

[illegible]


S_1

Alberi di copertura minimi: algoritmo di Kruskal

L'idea di *MST-Kruskal* è che possiamo ordinare gli archi in ordine crescente di peso, e, analizzandoli uno ad uno in questo ordine, stabilire se inserirlo come parte dell'albero di copertura minimo oppure no. Quale sarebbe la ragione di non farlo, ad un certo punto dell'esecuzione? Semplicemente, un arco (u, v) (che è chiaramente l'arco di peso minimo non ancora considerato, visto che li stiamo analizzando in ordine) **non** è parte di nessun MST se u e v sono già connessi da qualche altro arco precedentemente scelto. Come prima, sia T l'insieme di archi che abbiamo scelto fino ad un certo punto. T, a differenza del caso di *MST-Prim*, non è necessariamente un albero in ogni momento, ma lo sarà certamente alla fine della computazione. Dati gli archi di T, e dato l'insieme V di vertici, diciamo che un sottoinsieme S di V è T-**connesso** se, considerando solo archi in T, è un albero, ed è massimale.

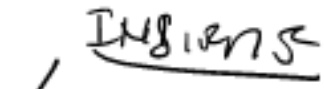
Alberi di copertura minimi: algoritmo di Kruskal

Dato T ad un certo punto della computazione, identifichiamo tutte le componenti T-connesse di V: $S_1, S_2, \ldots, S_n$. La tupla $(S_1, S_2, \ldots, S_n)$ è certamente una partizione di V (perchè anche i singoletti sono T-connessi), e generalizza il concetto di taglio visto prima. Lo chiamiamo **taglio generalizzato**. Per la stessa ragione per cui questa strategia funzionava per *MST-Prim*, possiamo affermare che **un arco di peso minimo tra quelli non ancora considerati** che attraversa un taglio generalizzato è un arco sicuro. Passare da un insieme T ad un insieme T' scegliendo un arco di peso minimo tra quelli non ancora considerati, assumendo che attraversi il taglio, garantisce che, se T era sottoinsieme di qualche MST, allora lo sarà anche T'.

Alberi di copertura minimi: algoritmo di Kruskal

Come facciamo a garantire che un arco scelto attraversi il taglio? La definizione è semplice: un arco (u, v) attraversa un taglio generalizzato se u e v appartengono a diverse componenti T-connesse. Quindi abbiamo bisogno di una struttura dati per insiemi disgiunti come quelle che abbiamo visto in una lezione passata. In particolare, per noi adesso, le operazioni avranno la seguente semantica. L'operazione di *MakeSet* costruisce un nuovo insieme (che per noi sarà una componente T-connessa), quella di *FindSet* stabilisce se due elementi appartengono allo stesso insieme (alla stessa componente T-connessa), e la *Union* unisce due insiemi in uno solo (unisce due componenti T-connesse in una sola, come conseguenza di aver scelto un arco).

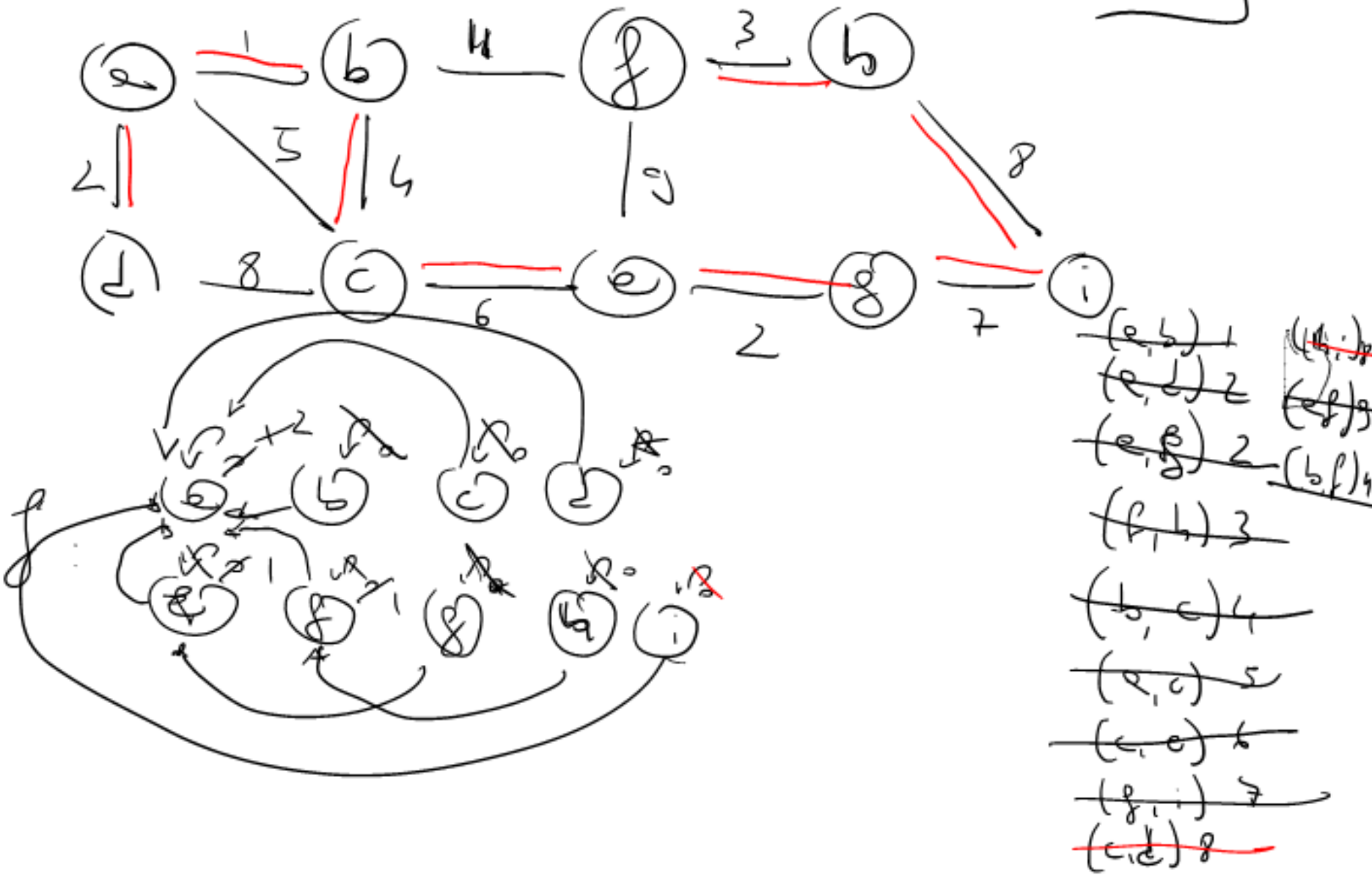
Alberi di copertura minimi: algoritmo di Kruskal



```
proc MST-Kruskal (G, w)
  T = ∅
  for (v ∈ G.V)
    do MakeSet(v)
  SortNoDecreasing(G.E)
  for ((u, v) ∈ G.E − in order)
    do
      if (FindSet(u) ≠ FindSet(v))
        then
          T = T ∪ {(u, v)}
          Union(u, v)
  return T
```

Ejercicio 134

1C



Correttezza di *MST-Kruskal*

Per mostrare che *MST-Kruskal* è **corretto**, osserviamo che tutti gli insiemi $S_1, S_2, \ldots, S_{|V|}$ che vengono creati da *MakeSet* nella prima operazione costituiscono un taglio generalizzato, considerato che T è vuoto. A questo punto, dimostriamo la seguente **invariante**: ogni arco che viene aggiunto nel ciclo è un arco sicuro:

- Poichè gli archi vengono ordinati in maniera crescente, l'invariante è vera all'inizio (**caso base**): il primo arco scelto è un arco di peso minimo, e certamente attraversa il taglio.
- Supponiamo adesso che l'invariante sia vera fino ad una certa esecuzione (**caso induttivo**). Sia $(S_1, \ldots, S_n)$, con $n \leq |V|$, il taglio generalizzato corrente, e sia (u, v) l'arco considerato. Per definizione di *FindSet*, u e v appartengono a due diverse componenti T-connesse, quindi attraversa il taglio. Ogni arco (u', v') di peso minore di (u, v) è giá stato considerato prima di (u, v), e attraversava il taglio quando era stato considerato (dunque, dopo *Union*, non lo attraversa piú) oppure non lo attraversava. Quindi (u', v') non attraversa il taglio, e (u, v) è un arco di peso minimo che attraversa il taglio.

Complessità di *MST-Kruskal*

Come per *MST-Prim*, per calcolare la **complessitá** di *MST-Kruskal* dobbiamo distinguere tra grafi sparsi e grafi densi; inoltre, molto dipende dall'implementazione delle operazioni di insiemi disgiunti. Noi abbiamo visto tre implementazioni possibili, per un totale di sei casi. Poichè pero, in questo caso, le scelte non si influenzano tra loro, possiamo limitarci a scegliere l'implementazione più efficiente, con foreste di alberi, union-by-rank, e compressione del percorso. Se il grafo è denso, allora abbiamo: inizializzazione ($O(1)$), ordinamento, $(\Theta(|E| \cdot log(|E|)) = \Theta(|V|^2 \cdot log(|E|)) = \Theta(|V|^2 \cdot log(|V|))$, piú $O((|V| + |E|)$ diverse operazioni su insiemi, di cui $O(|V|)$ sono *MakeSet* (per un totale di $O((|V| + |E|) \cdot \alpha(|V|))$=$O(|V| + |E|)$). In totale, $\Theta(|V|^2 \cdot log(|V|))$. Se invece il grafo è sparso abbiamo: inizializzazione ($O(1)$), ordinamento: ($\Theta(|E| \cdot log(|E|))$, e $O((|V| + |E|)$ diverse operazioni su insiemi, di cui $O(|V|)$ sono *MakeSet* (per un totale di $O((|V| + |E|) \cdot \alpha(|V|))$=$O(|V| + |E|)$). Totale, ($\Theta(|E| \cdot log(|E|))$.

Conclusione

Gli alberi di copertura minima ci introducono ai problemi di **ottimizzazione**, dove la risposta è una quantitá (e una scelta, di vertici, di archi, eccetera). Questi costituiscono una categoria a parte nel mondo dell'algoritimica, che studieremo in maniera particolare nel corso di linguaggi formali, calcolabilitá, e complessitá.