Progettazione di algoritmi

Discussione dell'esercizio [MST]

Sia G un grafo pesato e connesso e sia T un MST di G.

a. Se incrementiamo di una stessa costante c il peso degli archi di G, il nuovo peso di un albero di copertura S diventa $p'(S) = p(S) + (n - 1)c$. Quindi T è ancora un MST dato che $p'(T) = p(T) + (n - 1)c \leq p(S) + (n - 1)c = p'(S)$, per ogni albero di copertura S.

b. Un arco di peso minimo di G è un arco $\{u, v\}$ tale che $p(u, v) \leq p(x, y)$ per ogni arco $\{x, y\}$ di G. Si osservi che possono esserci tanti archi di peso minimo. Supponiamo per assurdo che T non contenga nessun arco di peso minimo. Questo significa che per ogni arco $\{w, z\}$ in T esiste un arco $\{x, y\}$ di G tale che $p(x, y) < p(w, z)$. Sia allora $\{u, v\}$ un arco di peso minimo di G. Chiaramente, $\{u, v\}$ non appartiene a T e quindi $T \cup \{u, v\}$ ha un ciclo C che contiene l'arco $\{u, v\}$. Siccome T non contiene archi di peso minimo, tutti gli archi di C eccetto $\{u, v\}$, essendo archi di T, devono avere un peso strettamente maggiore a $p(u, v)$. Sia $\{w, z\}$ uno di questi. Definiamo $T' = (T - \{w, z\}) \cup \{u, v\}$. Chiaramente T' è ancora un albero di copertura perché è connesso, copre n nodi ed ha esattamente $n - 1$ archi. Inoltre, risulta $p(T') = p(T) - p(w, z) + p(u, v) < p(T)$ (dato che $p(u, v) < p(w, z)$), in cotraddizione con l'ipotesi che T sia un MST. Quindi T deve necessariamente contenere almeno un arco di peso minimo.

c. Assumiamo che gli archi di G abbiano tutti pesi distinti. Supponiamo per assurdo che ci siano due MST differenti A e B di G. Sia allora $\{u, v\}$ l'arco di peso minimo tra quelli che appartengono alla differenza simmetrica di A e B, cioè $A \Delta B = (A - B) \cup (B - A)$. Supponiamo senza perdità di generalità che $\{u, v\}$ è in $A - B$. L'aggiunta di $\{u, v\}$ a B produce un ciclo C. Gli archi in $C - \{u, v\}$ appartengono tutti a B ma non possono appartenere anche a A, altrimenti A conterrebbe un ciclo. Quindi c'è almeno un arco $\{w, z\}$ che è in C (e quindi in B) ma non è in A. Ne segue che $\{w, z\}$ è in $A \Delta B$ e allora $p(w, z) < p(u, v)$ (perché $\{u, v\}$ ha peso minimo tra tutti gli archi in $A \Delta B$ e tutti i pesi sono diversi). Perciò $B' = (B - \{w, z\}) \cup \{u, v\}$ è un albero di copertura con peso strettamente inferiore a quello di B contraddicendo l'ipotesi che B fosse un MST. Dunque, se tutti i pesi degli archi sono distinti, c'è un unico MST.

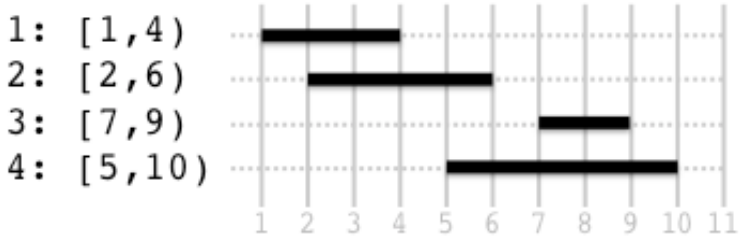
d. Effettivamente gli algoritmi di Prim e Kruskal funzionano correttamente anche quando G contiene archi di peso negativo. Questo può anche essere ricavato da quanto provato nel punto (a).

Discussione dell'esercizio [aule]

Dobbiamo risolvere un problema che è una variazione del problema Selezione Attività. Invece di massimizzare il numero di attività che si possono svolgere in un'unica aula, dobbiamo minimizzare il numero di aule che occorrono per svolgere tutte le attività. Una prima idea potrebbe essere quella di sfruttare l'algoritmo TERMINA_PRIMA che ci permette di massimizzare il numero di attività da svolgere in un'aula:

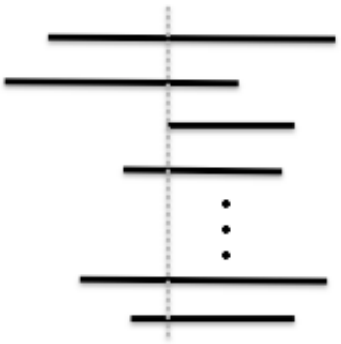
```
AULE_TERMINA_PRIMA(A: insieme delle n attività)
    AULE <- lista vuota        /* lista delle aule e delle relative attività
*/
    WHILE A non è vuoto DO
        S <- TERMINA_PRIMA(A)  /* seleziona le attività da svolgere nella
prossima aula */
        A <- A - S
        AULE.append(S)
    RETURN AULE
```

L'algoritmo ritorna una lista il cui *h*-esimo elemento *S* è l'insieme delle attività da svolgere nell'*h*-esima aula. La lunghezza della lista è il numero di aule utilizzate. Ma l'algoritmo non è corretto, infatti ecco un controesempio:



L'algoritmo AULE_TERMINA_PRIMA usa tre aule: {1,3}, {2}, {4}. Invece la soluzione ottima usa solo due aule: {1,4}, {2,3}.

Dobbiamo trovare un altro algoritmo. Osserviamo che un algoritmo corretto deve garantire che ogniqualvolta usa una nuova aula, è proprio necessario usare un'aula in più. Cosa significa questo? Significa che se l'algoritmo già usa *k* - 1 aule e a un certo punto introduce una nuova aula, allora le attività già considerate non possono essere svolte in *k* - 1 aule. Quindi devono esistere *k* attività che sono tra loro incompatibili, cioè due qualsiasi di queste attività si sovrappongono. Osserviamo che se *I* è un insieme di attività fra loro incompatibili,



il tempo d'inizio dell'attività che inizia più tardi tra le attività di *I* è contenuto in tutte le attività di *I*. Inoltre, se *i* è un'attività che è incompatibile con le attività di un insieme *A* e le attività in *A* non iniziano dopo *i*, allora le attività $A \cup \{i\}$ sono incompatibili tra loro, perché il tempo d'inizio di *i* è contenuto in tutte le attività. Questo ci suggerisce di considerare le attività in ordine di tempo d'inizio e per ogni attività se non la possiamo svolgere in una delle aule già

usate introdurremo una nuova aula.

```
AULE_INIZIA_PRIMA(A: insieme delle n attività)
    AULE <- lista vuota
    WHILE A non è vuoto DO
        Sia i un'attività in A che inizia prima
        IF esiste un'aula j tale che i è compatibile con le attività in
AULE(j) THEN
            AULE(j) <- AULE(j) ∪ {i}
        ELSE
            AULE.append({i})
        A <- A - {i}
    RETURN AULE
```

La soluzione ritornata è ammissibile. Infatti le attività assegnate ad un'aula sono fra loro compatibili e ogni attività è assegnata ad un'aula. Per dimostrare che l'algoritmo usa un numero minimo di aule potremmo adottare lo schema generale per gli agoritmi greedy ma in questo caso la dimostrazione sarebbe piuttosto lunga. Invece c'è una dimostrazione molto più semplice basata sull'argomento che ha ispirato l'algoritmo. Sia k il numero di aule usate dall'algoritmo. Consideriamo l'iterazione del `WHILE` nella quale l'ultima delle k aule è stata introdotta. Sia i l'attività esaminata in quell'iterazione. Siccome i ha aperto una nuova aula, i non poteva essere svolta in nessuna delle precedenti k - 1 aule. Questo implica che esistono k - 1 attività $j_1,\ldots,j_{k-1}$ una per ognuna delle k - 1 aule, tali che i è incompatibile con ognuna di queste. Siccome le attività $j_1,\ldots,j_{k-1}$ sono state considerate prima di i, i tempi d'inizio di esse sono minori o uguali al tempo d'inizio di i. Quindi il tempo d'inizio di i è contenuto negli intervalli di tutte le attività $j_1,\ldots,j_{k-1}$. Ne segue che le k attività i, $j_1,\ldots,j_{k-1}$ sono fra loro incompatibili e quindi sono necessarie k aule.

Per quanto riguarda un'implementazione efficiente, si noti che le attività possono essere ordinate per tempi d'inizio in tempo $O(n\log n)$ prima del `WHILE` , che diventerà un `FOR` che scorre l'array ordinato delle attività. Poi si usa un min-heap per mantenere le aule organizzate rispetto ai loro tempi di fine. Cosi' l'operazione di lettura dell'aula con il minor tempo di fine, cioè quella in cui eventualmente svolgere l'attività corrente, ha tempo O(1). Inoltre, l'aggiornamento dell'heap a seguito dell'aggiunta dell'attività corrente prende tempo limitato da $O(\log n)$. Quindi questa implementazione ha complessità $O(n\log n)$.

Algoritmo di Kruskal

Torniamo al problema di trovare un MST. L'algoritmo di Kruskal parte dall'insieme vuoto e ad ogni passo aggiunge all'insieme un arco di peso minimo tra tutti quelli che possono essere aggiunti senza creare cicli.

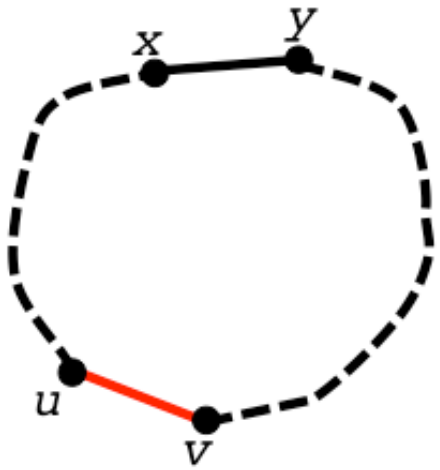
```
KRUSKAL(G: grafo non diretto, pesato e connesso)
    SOL <- insieme vuoto
    WHILE esiste un arco che può essere aggiunto a SOL senza creare cicli DO
        Sia {u, v} un arco di peso minimo fra tutti quelli che possono
```

```
essere aggiunti a SOL
        SOL <- SOL ∪ {{u, v}}
    RETURN SOL
```

Prima di tutto sinceriamoci che l'algoritmo termina sempre. Ad ogni iterazione del `WHILE` un nuovo arco viene aggiunto a `SOL` e la condizione del `WHILE` diventa falsa quando non ci sono più archi che possono essere aggiunti a `SOL` senza creare cicli, quindi vengono eseguite al più n-1 iterazioni (un grafo con al più n nodi e aciclico ha al più n-1 archi). Siccome il grafo è connesso, il numero di iterazioni è esattamente n-1, dato che se fossero di meno il sottografo indotto dagli archi in `SOL` non sarebbe connesso e quindi esisterebbe almeno un arco che si può aggiungere senza creare cicli. Sia SOL_h il valore di `SOL` al termine della h-esima iterazione e sia SOL_0 il valore iniziale. Dimostriamo che ogni soluzione parziale è estendibile ad una soluzione ottima.

> **Per ogni $h = 0,1,\dots n\text{-}1$ esiste una soluzione ottima SOL^* che estende SOL_h.**

Dimostrazione Procediamo per induzione su h. Per $h = 0$ è banalmente vero. Sia ora $h \geq 0$ (e $h \leq n$-1), assumendo la tesi vera per h la dimostriamo per $h + 1$. Quindi per ipotesi induttiva esiste una soluzione ottima SOL^* che estende SOL_h. Se SOL^* estende anche SOL_{h+1}, abbiamo fatto. Altrimenti, l'arco $\{u, v\}$ aggiunto a SOL_h nella $(h + 1)$-esima iterazione non appartiene a SOL^*. Cerchiamo di trasformare SOL^* in un'altra soluzione ottima che contiene SOL_{h+1}. Siccome gli archi di SOL^* formano un albero di copertura, in $SOL^* \cup \{u, v\}$ l'arco $\{u, v\}$ determina un ciclo C. L'arco $\{u, v\}$ non crea cicli con SOL_h per cui in C deve esserci almeno un altro arco $\{x, y\}$ che non appartiene a SOL_h.



Inoltre, l'arco $\{x, y\}$ non crea cicli con SOL_h dato che appartiene a SOL^* e SOL^* contiene SOL_h e non ha cicli. Quindi l'arco $\{x, y\}$ era tra gli archi che potevano essere scelti durante l'$(h + 1)$-esima iterazione. È stato scelto $\{u, v\}$ per cui deve essere che $p\{u, v\} \leq p\{x, y\}$. Definiamo $SOL^{\#} = (SOL^* - \{\{x, y\}\}) \cup \{\{u, v\}\}$. Chiaramente, $SOL^{\#}$ è una soluzione ottima che estende SOL_{h+1}.

Adesso sappiamo che la soluzione finale SOL_{n-1} prodotta dall'algoritmo di Kruskal è contenuta in una soluzione ottima SOL^*. Ma sia SOL_{n-1} che la soluzione SOL^* hanno lo stesso numero di archi, cioè n-1, quindi sono uguali.

Implementazione efficiente dell'algoritmo di Kruskal

Ora che sappiamo che l'algoritmo di Kruskal è corretto, consideriamo possibili implementazioni efficienti. Osserviamo subito che quando in una iterazione un arco è esaminato e viene scartato perché crea cicli non verrà successivamente riesaminato. Allora potrebbe risultare conveniente ordinare preliminarmente gli archi del grafo in ordine di peso non decrescente. Fatto ciò, per ogni arco che si presenta tramite questo ordine, si deve controllare se produce o meno cicli con gli archi in `SOL`. Per effettuare tale controllo osserviamo che `SOL` determina delle componenti connesse che coprono tutti i nodi del grafo e un arco non crea cicli con `SOL` se e solo se gli estremi dell'arco appartengono a componenti connesse differenti. Per mantenere l'informazione di queste componenti possiamo usare un vettore `CC` tale che, per ogni nodo `i`, `CC[i]` è l'etichetta che identifica la componente a cui il nodo `i` appartiene.

```
KRUSKAL(G: grafo non diretto, pesato e connesso)
    SOL <- lista vuota
    E <- array che contiene gli archi di G, per ogni arco i: E[i].u, E[i].v,
E[i].p
    Ordina l'array E rispetto ai pesi in modo non descrescente
    CC <- array delle componenti
    FOR ogni nodo u DO
        CC[u] <- u      /* Inizialmente, ogni nodo è una componente a sè
stante */
    FOR i = 1 TO m DO
        {u, v} <- {E[i].u, E[i].v}
        IF CC[u] <> CC[v] THEN         /* Se l'arco non crea cicli */
            SOL.append({u, v)          /* aggiungilo alla soluzione */
            c <- CC[v]
            FOR ogni nodo w DO         /* Fondi le due componenti */
                IF CC[w] = c THEN
                    CC[w] = CC[u]
    RETURN SOL
```

L'ordinamento degli archi richiede O($m$log$m$) tramite un algoritmo classico di ordinamento.

Ogni iterazione del `FOR` sugli archi richiede un tempo che dipende dal fatto se l'arco crea o meno cicli. Se crea cicli l'iterazione richiede O(1) ma se invece non crea cicli e quindi l'arco viene aggiunto l'iterazione richiede O(n). Per fortuna però le iterazioni più onerose non sono così tante. Infatti, ce ne saranno tante quanti sono gli archi che vengono aggiunti a `SOL`, cioè, n–1. Quindi, il `FOR` sugli archi richiede O($m + n^2$) = O(n^2) e l'implementazione ha complessità O($m\log m + n^2$) = O($m\log n + n^2$). Quando il grafo non è denso questa è una complessità superiore a quella che abbiamo visto per l'algoritmo di Prim O(($n + m$)log$n$). Per ottenere una complessità di quest'ordine anche per l'algoritmo di Kruskal si può usare una struttura dati, chiamata *merge-find-set*, che permette di gestire efficientemente una collezione di insiemi disgiunti (le componenti) rispetto alle operazioni di *find* (determinare la componente di un nodo) e *merge* (fusione di due componenti).

Esercizio [MST dinamico]

Dato un grafo pesato G, sia T un suo MST. Supponiamo che un nuovo arco $\{u, v\}$ di peso p venga aggiunto al grafo G. Si vuole trovare il nuovo MST T', senza ricalcolarlo daccapo. L'algoritmo deve essere corretto e avere complessità O(n), assumendo che G e T sono rappresentati tramite liste di adiacenza. E se l'arco invece di essere aggiunto viene eliminato?