Progettazione di Algoritmi - lezione 11

Discussione dell'esercizio [rifornimenti]

Un algoritmo greedy per questo problema è quello che si presenta naturalmente: facciamo rifornimento solo quando è necessario, cioè, solo quando l'autonomia residua non ci permette di arrivare al prossimo distributore.

```
MIN_RIF(d: array delle n distanze, k: autonomia)
    R <- array che seleziona i distributori in cui fare rifornimento, inizializzato a 0
    aut <- 0
    FOR i = 1 TO n DO
        IF aut < d[i] THEN         /* Se non possiamo arrivare al prossimo distributore, */
            aut <- k               /* facciamo rifornimento */
            R[i] <- 1              /* nel distributore i */
        aut <- aut - d[i]      /* Consumo per il tratto di strada da i a i+1 */
    RETURN R
```

Osserviamo che la soluzione prodotta dall'algoritmo è sempre *ammissibile*, nel senso che i rifornimenti selezionati permettono di percorrere l'intero viaggio. Infatti, per ogni tratto di strada tra due distributori i e i+1, lo `IF` garantisce che se l'autonomia residua non è sufficiente per percorrere quel tratto, viene effettuato un rifornimento al distributore i.

Denotiamo con R_i l'array `R` al termine dell'i-esima iterazione del `FOR` e con R_0 l'array iniziale. Sapendo che la soluzione prodotta è ammissibile, per dimostrare la correttezza è sufficiente mostrare che ogni soluzione parziale R_i è estendibile a una soluzione ottima. Così la soluzione finale R_n è necessariamente ottima.

> **Per ogni $i = 0,1,\ldots n$ esiste una soluzione ottima R^* che estende R_i, cioè $R_i[j] = R^*[j]$ per $j = 1,\ldots i$.**

Dimostrazione Procediamo per induzione su i. Per $i = 0$ non c'è nulla da provare. Supponiamo che sia vera per i e proviamola per $i + 1$. Siccome per $j = 1,\ldots i$ $R_{i+1}[j] = R_i[j]$ e per ipotesi induttiva $R_i[j] = R^*[j]$, se risulta $R_{i+1}[i+1] = R^*[i+1]$ abbiamo fatto. Altrimenti, consideriamo i due possibili casi.

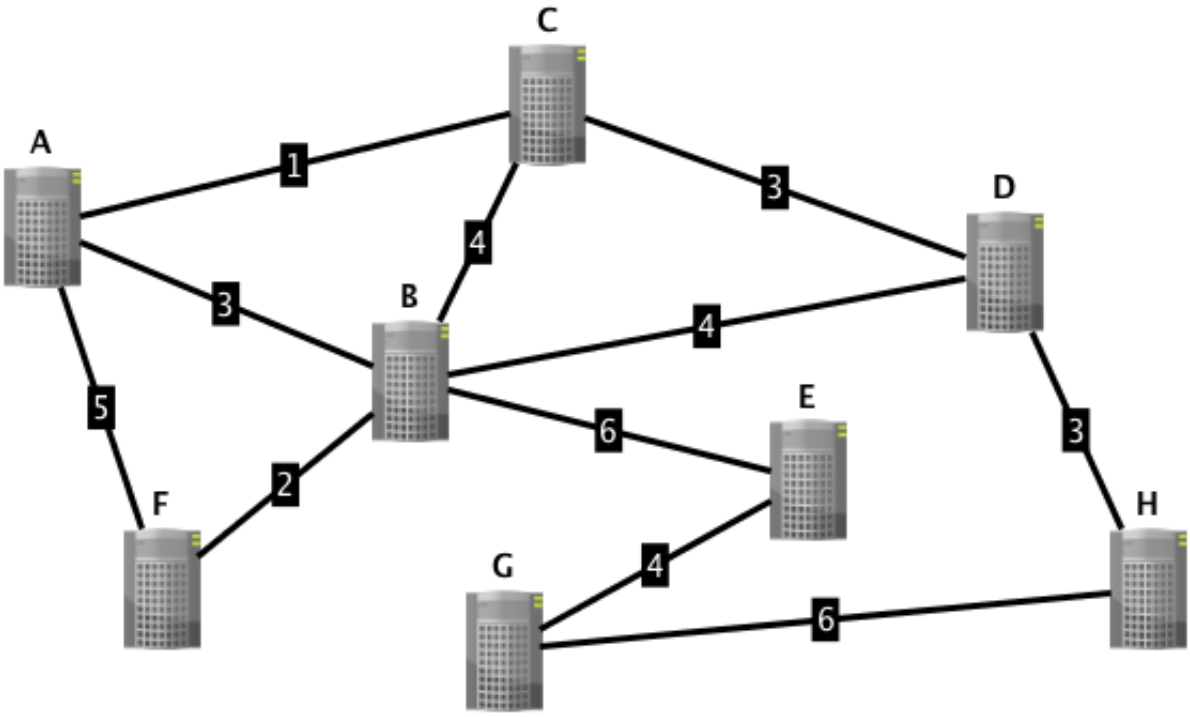
- $R_{i+1}[i+1] = 1$ e $R^*[i+1] = 0$. Questo caso non può accadere perché, per l'ipotesi induttiva, R^* seleziona gli stessi distributori di R_{i+1}, fino al distributore i, e il fatto che $R_{i+1}[i+1]$ seleziona il distributore $i+1$ implica che l'autonomia residua al distributore $i+1$ non permetteva di percorrere il tratto da $i+1$ a $i+2$. Quindi R^* non può non selezionare $i+1$.
- $R_{i+1}[i+1] = 0$ e $R^*[i+1] = 1$. Osserviamo che non può essere $i+1 = n$ perchè altrimenti R^* non sarebbe una soluzione ottima (avrebbe un rifornimento in più rispetto ad una soluzione ammissibile data da $R_{i+1} = R_n$). Quindi deve essere $i+1 < n$. Definiamo $R^\#$ tale che $R^\#[j] = R^*[j]$ per ogni $j \neq i+1, i+2$ e $R^\#[i+1] = 0$, $R^\#[i+2] = 1$. Chiaramente $R^\#$ estende R_{i+1}. Il numero di rifornimenti selezionati da $R^\#$ non è superiore a quello di R^*. Inoltre $R^\#$ è ammissibile. Infatti i rifornimenti fino a $i+1$ sono gli stessi di quelli selezionati da R_{i+1} e quindi permettono di arrivare al distributore $i+2$, poi $R^\#$ seleziona $i+2$ e i successivi rifornimenti sono uguali a quelli di R^* per cui permettono di percorrere tutti i tratti successivi al distributore $i+2$. Quindi $R^\#$ è una soluzione

ottima che estende R_{i+1}.

Per quanto riguarda un'implementazione efficiente, osserviamo che la descrizione dell'algoritmo che abbiamo dato è già un'implementazione efficiente in quanto ha complessità $O(n)$.

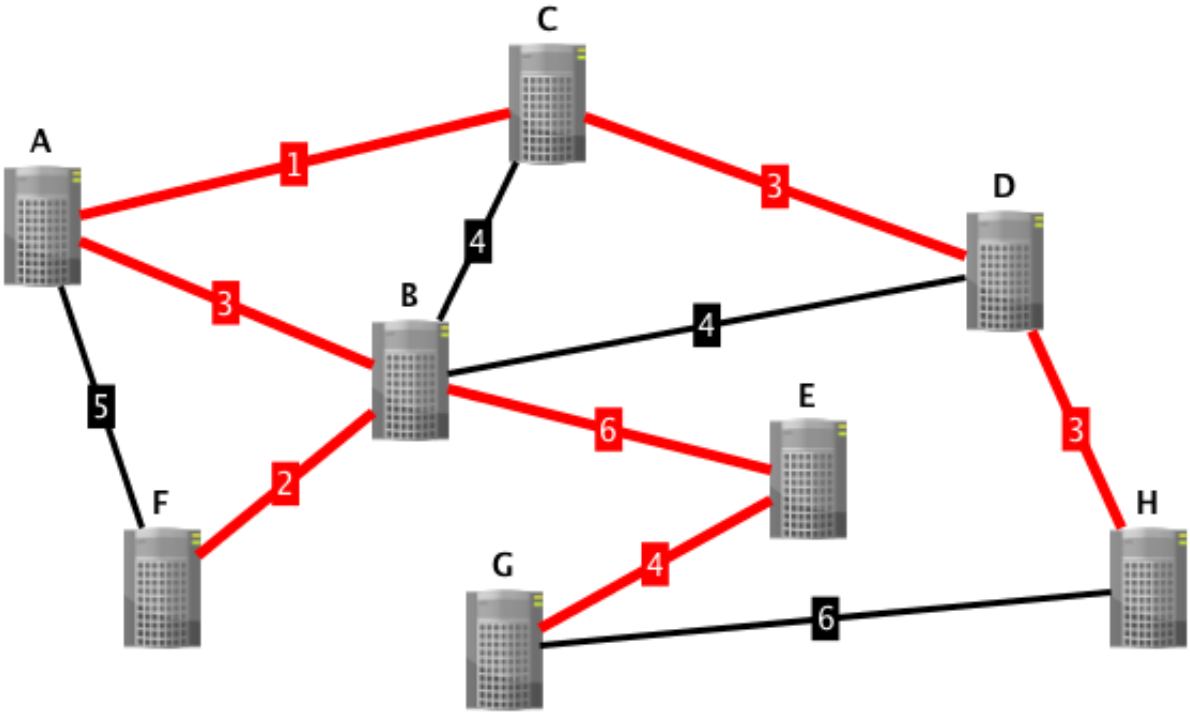
Minimo albero di copertura

Consideriamo un insieme di computer (server e/o router) che devono essere connessi tramite cavi a formare una rete in modo che ogni computer possa comunicare con ogni altro o tramite un cavo che li collega direttamente o passando per altri computer delle rete. Ogni possibile collegamento tramite cavo ha un costo. Un esempio è mostrato nella figura seguente:



Quindi vogliamo installare alcuni dei collegamenti possibili in modo tale da garantire la connessione della rete e al contempo minimizzare il costo totale.

Il problema può essere rappresentato tramite un grafo *G* non diretto e pesato i cui nodi sono i computer, gli archi sono i possibili collegamenti e i relativi pesi sono i costi. Il requisito della connessione è tradotto nel requisito che l'insieme degli archi da selezionare *T* deve connettere tutti i nodi. Siccome vogliamo minimizzare il costo totale, l'insieme *T* non deve contenere cicli, perchè se ci fossero potremmo eliminare almeno un arco senza perdere la connessione e riducendo però il costo. Quindi *T* deve essere un sottografo connesso ed aciclico, cioè un albero, che tocca tutti i nodi di *G*. Un sottografo con queste proprietà è detto **albero di copertura** di *G* (in inglese *spanning tree*). Si osservi che un albero di copertura può essere trovato con una qualsiasi visita (DFS o BFS) del grafo. Però per risolvere il nostro problema dobbiamo trovare un albero di copertura *T* che abbia costo totale (cioè, la somma dei pesi degli archi di *T*) minimo tra i costi di tutti gli alberi di copertura di *G*. Un albero di copertura di costo minimo è detto **minimo albero di copertura** (inglese *minimimum spanning tree*, abbreviato *MST*) di *G*. Quindi il nostro problema consiste nel trovare un minimo albero di copertura, o più in breve, un MST di *G*. Ad esempio, un MST per il grafo della figura precedente è quello evidenziato in rosso nella figura qui sotto:



Non è difficile pensare ad algoritmi greedy per risolvere questo problema. Viene infatti subito in mente di partire dall'insieme vuoto e ad ogni passo aggiungere all'insieme un arco di peso minimo tra tutti quelli che non producono cicli con quelli già scelti. Procedendo in questo modo siamo intuitivamente sicuri che, se il grafo di input è connesso, alla fine, quando cioè non possiamo aggiungere altri archi, l'insieme determinerà un albero di copertura. Rimane allora da dimostrare che ha peso minimo. Quello che abbiamo appena delineato è conosciuto come algoritmo di Kruskal, dal nome del ricercatore, Joseph Kruskal, che lo propose nel 1956. Studieremo l'algoritmo di Kruskal dopo aver studiato un altro algoritmo greedy per trovare un MST che è meno intuitivo che possa essere corretto ma è più facile da implementare in modo efficiente.

Algoritmo di Prim

L'algoritmo di Prim (dal nome del ricercatore, Robert C. Prim, che lo propose nel 1957) restringe, rispetto all'algoritmo di Kruskal, gli archi che possono essere scelti ad ogni passo. L'algoritmo parte da un nodo qualsiasi e ad ogni passo estende l'albero finora costruito con un arco che aggiunge un nuovo nodo all'albero. L'arco scelto è uno di peso minimo tra tutti quelli che possono estendere l'albero.

```
PRIM(G: grafo non diretto, pesato e connesso)
    SOL <- insieme vuoto
    Scegli un nodo s di G
    C <- {s}
    WHILE C ≠ V DO      /* Finché l'albero non copre tutti i nodi di G */
        Sia {u, v} un arco di peso minimo fra tutti quelli con u in C e v non in C
        SOL <- SOL ∪ {{u, v}}
        C <- C ∪ {v}
    RETURN SOL
```

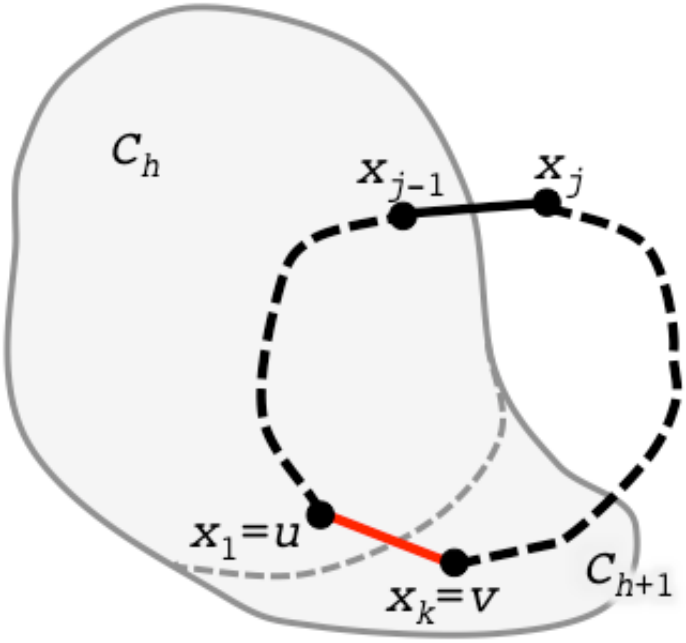
La primissima cosa da osservare è che tutti i passi dell'algoritmo possono essere effettivamente eseguiti. Infatti, siccome il grafo di input *G* è connesso, la condizione del `WHILE`, `C ≠ V`, garantisce che c'è almeno un arco tra un nodo in `C` e un nodo non in `C` e quindi ad ogni iterazione del `WHILE` un arco {*u*, *v*} esiste sempre.

Ora dobbiamo mostrare che l'algoritmo termina sempre, cioè, il `WHILE` esegue sempre un numero finito di iterazioni. Ciò equivale a dimostrare che esiste sempre un'iterazione in cui la condizione del `WHILE` risulta falsa, cioè, un'iterazione in cui risulta `C = V`. Ad ogni iterazione del `WHILE` viene aggiunto un nuovo nodo a `C`, quindi vengono eseguite esattamente *n*–1 iterazioni.

Siano SOL_h e C_h i valori di, rispettivamente, `SOL` e `C` al termine dell'h-esima iterazione del `WHILE` e siano SOL_0 e C_0 i valori iniziali. Proviamo che ogni soluzione parziale può essere estesa ad una soluzione ottima:

Per ogni $h = 0,1,\ldots, n-1$, esiste una soluzione ottima SOL^* che estende SOL_h.

Dimostrazione Per induzione su h. Per $h = 0$, SOL_0 è l'insieme vuoto, quindi l'asserto è vero. Sia $h \geq 0$, vogliamo provare che se l'asserto è vero per h (ipotesi induttiva) allora è vero anche per $h + 1$ (sempre che $h+1 \leq n-1$). 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` durante l'$(h+1)$-esima iterazione non appartiene a SOL^*. Vogliamo trasformare SOL^* in un'altra soluzione ottima che però estende SOL_{h+1}. Siccome SOL^* è un albero di copertura, l'arco $\{u, v\}$ determina un ciclo in $SOL^* \cup \{u, v\}$. Vogliamo far vedere che c'è un arco di questo ciclo che possiamo sostituire con l'arco $\{u, v\}$. L'arco che cerchiamo deve non appartenere a SOL_{h+1} e avere peso maggiore od uguale a $p\{u, v\}$. L'arco $\{u, v\}$ è stato scelto come un arco di peso minimo fra tutti gli archi con u in C_h e v non in C_h. Quindi se troviamo un altro arco del ciclo con questa proprietà abbiamo fatto. Siano $x_1, x_2,\ldots x_k$ i nodi del ciclo presi nell'ordine tale che $x_1 = u$ e $x_k = v$:



Partendo da x_1 percorriamo i nodi del ciclo in tale ordine fino a che troviamo il primo nodo che non appartiene a C_h. Sia x_j questo nodo, allora l'arco $\{x_{j-1}, x_j\}$ è un arco del ciclo diverso da $\{u, v\}$ tale che x_{j-1} è in C_h e x_j non è in C_h. Definiamo $SOL^{\#} = (SOL^* - \{\{x_{j-1}, x_j\}\}) \cup \{\{u, v\}\}$. Siccome $p\{x_{j-1}, x_j\} \geq p\{u, v\}$, allora $SOL^{\#}$ è una soluzione ottima che estende SOL_{h+1} e questo completa la dimostrazione.

Ora sappiamo che esiste una soluzione ottima SOL^* che contiene la soluzione finale SOL_{n-1} prodotta dall'algoritmo. Essendo SOL^* un albero di copertura deve avere esattamente $n-1$ archi. Siccome anche SOL_{n-1} ha esattamente $n-1$ archi, SOL_{n-1} coincide con SOL^*. Dunque l'algoritmo di Prim è corretto per il problema MST.

Implementazione efficiente dell'algoritmo di Prim

Ad ogni iterazione l'algoritmo di Prim deve trovare un arco di peso minimo che estende l'albero finora costruito. Questa operazione è molto simile a quella effettuata in un'iterazione dell'algoritmo di Dijkstra. Perciò viene naturale pensare di usare un min-heap per mantenere i nodi che possono estendere l'albero parziale con le priorità che sono i costi di aggiunta dei nodi. Il costo di un nodo è il minimo peso di un arco che connette il nodo all'albero parziale, se non ci sono archi che lo connettono, il costo è infinito. In questo modo, scegliere il nodo con costo minimo equivale a scegliere l'arco di peso minimo che estende l'albero. Per mantenere l'albero usiamo un vettore dei padri `P` la cui radice sarà inizializzata con un nodo *s* scelto in modo arbitrario.

```
PRIM(G: grafo non diretto, pesato e connesso)
    P: vettore dei padri, inizializzato a 0
    P[s] <- s         /* s è un nodo qualsiasi di G */
    H <- min-heap dei nodi coi costi inizializzati a ∞ eccetto quello di s inizializzato a 0
    WHILE H non è vuoto DO
        u <- H.get_min()               /* Estrae il nodo con costo minimo */
        FOR ogni adiacente v di u DO
            IF v in H AND p{u, v} < H.costo(v) THEN
                P[v] <- u
                H.decrease(v, p{u, v})  /* Aggiorna il costo di v e l'heap H */
    RETURN P
```

Il costo di tutte le inizializzazioni è O(*n*). Il numero di iterazioni del `WHILE` è esattamente *n* e ogni estrazione dall'heap del nodo con costo minimo costa O(log*n*). Poi sono controllati tutti gli adiacenti del nodo estratto e ognuno di tali controlli costa al più O(log*n*), essendo dominato dal costo dell'operazione di aggiornamento del costo dell'adiacente. Saranno effettuati due controlli per ogni arco, una volta per uno degli estremi e una volta per l'altro estremo. Il costo totale di tutti questi controlli è perciò O(*m*log*n*). Quindi la complessità di questa implementazione è O((*n* + *m*)log*n*).

Anche per l'algoritmo di Prim, similmente all'algoritmo di Dijkstra, c'è un'implementazione più semplice che usa un array per mantenere i costi dei nodi invece che un heap. L'estrazione del nodo di costo minimo richiede O(*n*) e ogni controllo di un adiacente richiede O(1). Quindi la complessità di questa implementazione è O(n^2). È dunque conveniente, rispetto a quella che usa un heap, solo quando il grafo è molto denso.

Esercizio [MST]

Sia *G* un grafo pesato e connesso. Sia *T* un MST per *G*. Dimostrare oppure fornire un controesempio per ognuna delle seguenti affermazioni:

a. *T* è ancora un MST anche per il grafo che si ottiene da *G* incrementando di una stessa costante *c* il peso degli archi.
b. *T* deve contenere un arco di peso minimo.
c. Se gli archi di *G* hanno tutti pesi distinti, allora l'MST è unico.
d. L'algoritmo di Prim funziona correttamente anche quando il grafo contiene archi di peso negativo.