1. Minimi alberi di supporto

Un problema di grandissima rilevanza pratica e teorica è costituito dalla ricerca di un albero che connetta tutti i nodi di un grafo e abbia la somma dei costi dei suoi archi minima. Questo problema prende il nome di *minimo albero di supporto* (*minimal spanning tree*).

Si può dimostrare che il problema si risolve facilmente con il seguente algoritmo detto di tipo *greedy*: si ordinano gli archi per costi crescenti e poi si costruisce l'albero inserendo un arco alla volta (secondo l'ordine) eliminando gli archi il cui inserimento creerebbe un ciclo.

Per implementare un tale algoritmo si tratta di capire come determinare, per ogni arco che si vorrebbe aggiungere, se si genera un circuito oppure no. Gli archi aggiunti di volta in volta formano un certo numero di insiemi sconnessi. Un arco da inserire genera un circuito se e solo se i suoi estremi appartengono alla stessa componente connessa. Se questo non avviene allora l'arco viene aggiunto e le due componenti connesse vengono fuse in una componente connessa più grande.

A questo scopo bisogna realizzare una struttura dati che permetta di: 1) identificare una componente connessa, 2) determinare rapidamente a quale componente appartiene un nodo, 3) creare una nuova componente dall'unione di due o più componenti. La struttura dati che realizza efficientemente queste operazioni viene detta *Union-Find*.

In una struttura Union-Find ogni insieme di nodi viene rappresentato come un albero con radice in cui ogni nodo dell'albero è un elemento del sottoinsieme e la radice è un elemento del sottoinsieme che funge da 'rappresentante' e che identifica il sottoinsieme. Ogni nodo dell'albero punta al suo nodo genitore mentre la radice punta a se stessa. Quindi la determinazione del sottoinsieme cui appartiene un nodo assegnato richiede un numero di passi pari alla profondità dell'albero.

Fondere assieme due sottoinsiemi disgiunti, cioè due alberi, richiede una complessità costante, in quanto basta 'dirottare' il puntatore di una radice verso l'altra radice. Nel caso gli alberi abbiano profondità diversa è ovviamente conveniente mantenere come radice quella dell'albero più profondo e quindi la profondità del nuovo albero è uguale alla maggiore delle due profondità. Nel caso le profondità siano uguali è indifferente quale radice mantenere come tale, però questa volta la profondità del nuovo albero è di una unità più elevata di quella degli alberi originari. Dimostriamo ora per induzione che, partendo da alberi costituiti da un singolo elemento e operando in questo modo, la profondità di un albero qualsiasi è $O(\log_2 n)$.

Sia $p(T)$ la profondità di un albero T. Se T è costituito da un singolo elemento allora $p(T) = 0$ e si ha $p(T) = \log_2 |T| = 0$. Quindi la proprietà è verificata se $|T| = 1$. Dati due alberi T_1 e T_2 per i quali la proprietà sia verificata, la loro fusione genera un albero T_3 per il quale si ha, se $p(T_1) \neq p(T_2)$ (sia $p(T_1) > p(T_2)$),

$$p(T_3) = p(T_1) \leq \log_2 |T_1| \leq \log_2(|T_1| + |T_2|) = \log_2 |T_3|$$

e, se $p(T_1) = p(T_2)$ (sia $|T_1| \leq |T_2|$),

$$p(T_3) = p(T_1) + 1 \leq \log_2 |T_1| + \log_2 2 = \log_2 2\,|T_1| \leq \log_2(|T_1| + |T_2|) = \log_2 |T_3|$$

e quindi la proprietà è ancora verificata.

Quindi la fase di costruzione dell'albero ha una complessità di $O(m \log n)$, che è pari alla complessità $O(m \log m) = O(m \log n)$ di ordinare preventivamente gli archi.

Quindi, usando questo metodo il problema si risolve in tempo $O(m \log n)$. La complessità della costruzione dell'albero si può ulteriormente abbassare ad una funzione 'quasi' lineare in m usando degli accorgimenti. Questo algoritmo viene generalmente citato come *algoritmo di Kruskal*, in quanto fu presentato per la prima volta da Kruskal (1956) (senza tuttavia la speciale struttura dati).

Vi sono altri algoritmi per il problema del minimo albero di supporto. Consideriamo un taglio qualsiasi nel grafo e l'arco $\hat{e}$ (o gli archi) di minimo costo del taglio. Supponiamo che un minimo albero di supporto

T non contenga $\hat{e}$. Si aggiunga allora $\hat{e}$ all'albero, generando così un circuito. Almeno un altro arco $\tilde{e}$ del circuito deve appartenere al taglio. Se si rimuove $\tilde{e}$ da $T \cup \hat{e}$ si ottiene un altro albero T' il cui costo deve essere inferiore a quello di T per l'ipotesi sui costi di $\hat{e}$ e $\tilde{e}$. Ma questo contraddice l'ottimalità di T.

Quindi, se si costruisce un albero prendendo per ogni taglio l'arco di costo minimo, certamente si ottiene un minimo albero di supporto. Quest'idea può essere realizzata algoritmicamente in vari modi. Due algoritmi in particolare meritano di essere menzionati e cioè l'algoritmo di Prim e l'algoritmo di Borůvka. La storia del problema del minimo albero di supporto è abbastanza interessante: sembra che il primo ad occuparsi del problema sia stato Borůvka in connessione con la costruzione della rete elettrica nella Moravia meridionale negli anni Venti. Questi articoli furono poi seguiti da un lavoro di Jarník (1930) dove di fatto si anticipa l'algoritmo di Prim. Questi risultati tuttavia rimasero ignorati a lungo tempo a causa della loro scarsa accessibilità. Negli anni Cinquanta il problema fu riesaminato e si ebbero gli articoli di Kruskal (1956) e di Prim (1957) (che comunque citano Borůvka (1926)) e di Dijkstra (1959), che indipendentemente presentò l'algoritmo di Prim insieme con il suo famoso algoritmo del cammino minimo.

L'*algoritmo di Prim* si basa sull'idea di aggiungere un arco ad un albero che supporta in modo ottimo un sottoinsieme S di nodi. In base alle precedenti considerazioni l'arco viene scelto come l'arco di minimo costo del taglio generato da S. L'insieme S viene quindi aggiornato aggiungendovi l'altro estremo dell'arco e la procedura si ripete finché S contiene tutti i nodi. La procedura viene inizializzata prendendo $S = \{s\}$ con s nodo arbitrario. Per realizzare efficientemente la procedura conviene trovare l'arco di minimo costo sfruttando l'informazione ottenuta nei precedenti passi. Dato $S \subset N$ sia $T(S)$ il minimo albero di supporto su S e per ogni $j \notin S$ si definisca

$$\begin{aligned} \rho_j(S) &:= \min\{w(e) : e \in Q(S) \cap Q(\{j\})\} \\ e_j(S) &:= \{e \in Q(S) \cap Q(\{j\}) : w(e) = \rho_j(S)\} \end{aligned}$$

e sia

$$k(S) := \operatorname*{argmin}_j \rho_j(S) \tag{1}$$

Allora

$$\begin{gathered} T(S \cup \{k(S)\}) := T(S) \cup \left\{e_{k(S)}(S)\right\} \\ \rho_j(S \cup \{k(S)\}) := \min\{w(e) : e \in Q(S \cup \{k(S)\}) \cap Q(\{j\})\} = \\ \min\left\{\min\{w(e) : e \in Q(S) \cap Q(\{j\})\} \ ; \ w((kj))\right\} = \min\left\{\rho_j(S) \ ; \ w((kj))\right\} \\ e_j(S \cup \{k(S)\}) := \begin{cases} e_j(S) & \text{se } \rho_j(S \cup \{k(S)\}) = \rho_j(S) \\ (k,j) & \text{se } \rho_j(S \cup \{k(S)\}) = w((kj)) \end{cases} \end{gathered} \tag{2}$$

L'aggiornamento (2) costa globalmente $O(m)$ mentre il calcolo del minimo in (1) costa $O(n)$ ad ogni iterazione e va ripetuto n volte. Complessivamente quindi l'algoritmo ha complessità $O(n^2)$. Si può notare la strettissima parentela dell'algoritmo di Prim con quello di Dijkstra. Come per quell'algoritmo il valore di complessità $O(n^2)$ non può essere abbassato per grafi densi ($m = \Omega(n^2)$) e in questi casi l'algoritmo di Prim è preferibile a quello di Kruskal che richiede un tempo $O(m \log n) = O(n^2 \log n)$.

Per grafi sparsi (cioè $m = O(n)$) è più conveniente usare una struttura a 'heap' per i valori $\rho_j(S)$. In questo modo il calcolo (1) richiede tempo costante. Tuttavia bisogna aggiornare lo 'heap' ad ogni aggiornamento (2) e ad ogni rimozione della radice dello 'heap'. Quindi discende una complessità globale $O(m \log n)$ pari a quella dell'algoritmo di Kruskal.

L'*algoritmo di Borůvka* opera come l'algoritmo di Prim cominciando però la costruzione dell'albero a partire da tutti i nodi contemporaneamente anziché da un solo nodo. Preventivamente si assegnino in modo arbitrario etichette da 1 a $|E|$ a tutti gli archi. Ad un passo generico dell'algoritmo è disponibile una foresta F formata da un certo numero di alberi $T_1, \ldots, T_p$ che sono minimi alberi di supporto per i rispettivi insiemi

di nodi $S_1, \ldots, S_p$. Sia $\hat{e}_i$ l'arco di costo minimo e, a parità di costo, di etichetta minima fra gli archi del taglio $Q(S_i)$. Si aggiorni $F := \cup_i T_i \cup_i \{\hat{e}_i\}$. Se F è un albero l'algoritmo è terminato altrimenti, si ripete l'iterazione. Siccome ad ogni iterazione il numero di componenti connesse almeno si dimezza, il numero di iterazioni è $O(\log n)$. Tuttavia è necessario un lavoro alquanto complesso di aggiornamento dati e strutture e alla fine si perviene comunque alla medesima complessità dell'algoritmo di Kruskal, cioè $O(m \log n)$, oppure con una più raffinata gestione dei dati a $O(m \log \log n)$. Il vantaggio di questo algoritmo risiede nella possibilità di parallelizzare il calcolo, altrimenti la difficoltà implementativa rispetto agli algoritmi di Kruskal e Prim ne sconsiglia l'uso. In figura 1 è rappresentata l'evoluzione dell'algoritmo di Borůvka su un grafo euclideo (costi degli archi pari alla distanza geometrica fra i nodi) completo con 200 nodi. Quattro iterazioni sono sufficienti a trovare il minimo albero di supporto.

Esercizio. Perché si sono introdotte le etichette nella valutazione dei costi degli archi nell'algoritmo di Borůvka? ∎