Azzolini Riccardo 2019-05-20

Algoritmi greedy

1 Algoritmo di Prim

L'**algoritmo di Prim** è un algoritmo greedy che calcola l'albero di copertura di peso minimo (MST) di un grafo, ma, a differenza dell'algoritmo di Kruskal, non opera sul matroide grafico.

- *Input*: un grafo $G = \langle V, E \rangle$ non orientato, connesso e pesato e una funzione peso $w : E \to \mathbb{Q}$.
- *Output*: un MST di costo minimo per G.

La soluzione parziale costruita dall'algoritmo di Prim è un albero. La costruzione inizia da un nodo sorgente $s \in V$, che inizialmente è l'unico appartenente all'albero. A ogni passo, si seleziona poi il lato meno costoso tra quelli che "escono" dall'albero (cioè quelli che collegano un nodo dell'albero a uno che non vi appartiene), e lo si aggiunge alla soluzione. L'algoritmo termina quando l'albero copre tutti i nodi di G.

L'implementazione dell'algoritmo fa uso di:

- l'insieme S di nodi appartenenti all'albero;
- l'insieme R dei nodi "da raggiungere", cioè che non appartengono ancora all'albero;
- il vettore dist, che associa a ogni nodo da raggiungere la sua distanza, cioè il peso del lato più corto che lo collega a un nodo dell'albero (se esiste):

$$\forall v \in R, \quad \text{dist}[v] = \begin{cases} \min\{w(\{v, z\}) \mid z \in S\} & \text{se } \{z \in S \mid \{v, z\} \in E\} \neq \varnothing \\ \infty & \text{altrimenti} \end{cases}$$

- l'insieme $D \subseteq R$ dei nodi "di frontiera", cioè quelli che sono collegati ai nodi dell'albero da un qualche lato:

$$D = \{v \in R \mid \text{dist}[v] \neq \infty\}$$

- il vettore vicino, che per ogni $v \in R$ indica da quale nodo parte il lato migliore per raggiungerlo (se esiste):

$$\forall v \in R, \quad \text{vicino}[v] = \begin{cases} z \in S & \text{se } w(\{v, z\}) = \text{dist}[v] \\ \perp & \text{se dist}[v] = \infty \end{cases}$$

```
set Prim(<V, E>, w, s) {
    set R = V \ {s};
    for (v in R) dist[v] = MAX_INT;
    set D = emptyset();
    for (v in L(s)) {
        vicino[v] = s;
        dist[v] = w({s, v});
        add(D, v);
    }

    set T = emptyset();
    while (!is_empty(D)) {
        v = nearest(D);
        delete(D, v); delete(R, v);
        add(T, {v, vicino[v]});
        for (z in L(v)) {
            if (z in R && w({z, v}) < dist[z]) {
                if (dist[z] == MAX_INT) add(D, z);
                dist[z] = w({z, v});
                vicino[z] = v;
            }
        }
    }
    return T;
}
```

1. Si inizializza l'insieme `R` dei nodi da raggiungere, che all'inizio contiene tutti i nodi del grafo tranne il nodo sorgente `s`.
2. Viene inizializzato il vettore delle distanze, impostando a `MAX_INT` (che rappresenta ∞) la distanza di tutti i nodi da raggiungere.
3. Si inizializzano i dati relativi ai nodi di frontiera, che all'inizio sono quelli adiacenti a `s` (`L(s)` è la lista di adiacenza di `s`).
4. Finché l'insieme dei nodi di frontiera (`D`) non è vuoto, il ciclo while:
   a) seleziona il nodo più vicino tra quelli di frontiera (`v`);[1]

[1] A parità di distanza, il nodo scelto dipende dall'implementazione.

b) cancella `v` dagli insiemi dei nodi di frontiera e da raggiungere (`R`);

c) aggiunge all'insieme `T` di lati dell'albero il lato (più corto) che collega `v` all'albero;

d) aggiorna la frontiera in base alla lista di adiacenza di `v`:

- se un nodo `z` adiacente a `v` non è da raggiungere, o se è già noto un lato più corto che lo collega alla soluzione, non si aggiornano le informazioni relative a `z`;
- altrimenti, `z` può essere un nodo che prima non apparteneva alla frontiera (quando `dist[z] == MAX_INT`), oppure un nodo già della frontiera la cui distanza diminuisce se lo si raggiunge mediante il lato `{z, v}` (cioè `w({z, v}) < dist[z]`).

5. Infine, vengono restituiti i lati dell'albero costruito.

1.1 Complessità

Le strutture dati più idonee sono:

- uno heap rovesciato per l'insieme `D` dei nodi di frontiera, nel quale la priorità di ciascun nodo è la sua distanza, in modo da poter ottenere in modo efficiente il nodo più vicino;
- un vettore di booleani per l'insieme `R` dei nodi da raggiungere, poiché l'uso di un vettore consente di verificare e modificare l'appartenenza di un nodo a `R` in tempo costante;
- una lista per memorizzare i lati dell'albero (insieme `T`), dato che bisogna effettuare solo operazioni di inserimento su di essa.

Dato un grafo con $n = \#V$ nodi e $m = \#E$ lati,

- ogni lista di adiacenza è attraversata una e una sola volta, quindi vengono incontrati in totale $O(m)$ nodi;
- per ogni nodo incontrato si effettuano alcune operazioni di costo costante e altre di costo $O(\log n)$ (quelle sullo heap, cioè `delete(D, v)`, `add(D, z)`, e l'aggiornamento della distanza, dopo il quale può essere necessario riposizionare il nodo nello heap).

Di conseguenza, il tempo di calcolo totale è $O(m \log n)$.

Osservazione: Rispetto all'algoritmo di Kruskal, che ha complessità $O(m \log m)$, l'ordine di grandezza non cambia, perché

$$m = O(n^2) \implies O(m \log m) = O(m \log n^2) = O(m \cdot 2 \log n) = O(m \log n)$$

L'algoritmo di Prim ha quindi costo minore rispetto a quello di Kruskal, ma solo per una costante moltiplicativa, e non a livello asintotico.

1.2 Correttezza

Siccome la soluzione parziale è un insieme T di lati di un albero, e non tutti i sottoinsiemi di T formano a loro volta degli alberi, non si ha un sistema di indipendenza, e quindi neanche un matroide. Di conseguenza, è necessario verificare la correttezza dell'algoritmo di Prim.

Teorema: Siano

- $G = \langle V, E \rangle$ un grafo non orientato e connesso;
- $w : E \to \mathbb{Q}$ una funzione peso;
- T un MST di G;
- U un sottoalbero di T (la soluzione parziale, a partire dal solo nodo sorgente, che è un sottoalbero di qualsiasi albero di copertura);
- S l'insieme dei nodi di U;
- $\{a, b\} = \min_w\{\{x, y\} \in E \mid x \in S, y \notin S\}$ è il meno costoso tra i lati che "escono" dall'albero (quelli che collegano un nodo appartenente a U con uno che non vi appartiene), cioè il prossimo lato da aggiungere per espandere la soluzione parziale.

Allora, esiste un MST $T' = \langle V, E' \rangle$ di G tale che U è un sottoalbero di T' e $\{a, b\} \in E'$, quindi la soluzione parziale rimane sottoalbero di un qualche MST (ma non necessariamente sempre dello stesso) all'aggiunta di $\{a, b\}$.

2 Algoritmo di Dijkstra

Problema: Dati un grafo pesato orientato $\langle V, E \rangle$ e una funzione peso $w : E \to \mathbb{Q}^+$,[2] calcolare i cammini minimi tra un nodo sorgente $s \in V$ e tutti gli altri.

L'algoritmo di Dijkstra risolve questo problema calcolando due vettori,

$$\text{pcam}[v] = \begin{cases} c & \text{se } c \text{ è il costo del cammino più corto da } s \text{ a } v \\ \infty & \text{se non esiste un cammino da } s \text{ a } v \end{cases}$$

$$\text{pred}[v] = \begin{cases} u & \text{se } u \text{ è il nodo che precede } v \text{ nel cammino più corto da } s \text{ a } v \\ \perp & \text{se non esiste un cammino da } s \text{ a } v \end{cases}$$

[2] Se gli archi potessero avere peso negativo, eventuali cicli si potrebbero percorrere ripetutamente per ridurre all'infinito il costo di un cammino, quindi la soluzione non sarebbe ben definita.

e partizionando i nodi in due insiemi,

- S: la soluzione parziale già calcolata;
- R: i nodi ancora da raggiungere.

```
void Dijkstra(<V, E>, w, s) {
    set R = V \ {s};
    for (v in R) pcam[v] = MAX_INT;
    set D = emptyset();
    for (v in L(s)) {
        pred[v] = s;
        pcam[v] = w((s, v));
        add(D, v);
    }

    while (!is_empty(D)) {
        v = nearest(D);
        delete(D, v); delete(R, v);
        for (z in L(v)) {
            if (z in R && pcam[v] + w((v, z)) < pcam[z]) {
                if (pcam[z] == MAX_INT) add(D, z);
                pcam[z] = pcam[v] + w((v, z));
                pred[z] = v;
            }
        }
    }
}
```

L'algoritmo è molto simile a quello di Prim, e ha la stessa complessità, $O(m \log n)$. La differenza principale è che nel ciclo `while`, invece di aggiornare la frontiera, vengono aggiornati i vettori `pcam` e `pred`: quando `pcam[v] + w((v, z)) < pcam[z]`, significa che esiste un cammino da `s` a `z` (passante per il lato `(v, z)`) che è più corto di quelli trovati in precedenza.

Osservazione: Le informazioni contenute in `pred` e `pcam` diventano sicuramente corrette solo quando i nodi vengono aggiunti alla soluzione parziale, perché quelle relative ai nodi in frontiera sono ancora in fase di calcolo.