Metodo di costruzione di un albero di copertura minimo

Laboratorio di Algoritmi e Strutture Dati - Lezione 6

Giuditta Franco

Dipartimento di Informatica,
Università di Verona

26 Febbraio 2008

Talk Outline

1. Algoritmi Greedy
2. “Minimal Spanning Tree” risolto con l’algoritmo (greedy) di Kruskal
3. Cenno alle “HashMap”, e metodi per implementare l’algoritmo di Kruskal.
4. ADT Grafo
5. Strutture Dati per Grafi

Introduzione agli algoritmi greedy

Gli algoritmi per la risoluzione di problemi di ottimizzazione sono costituiti da sequenze di passi elementari che presentano, ad ogni passo, un certo numero di scelte possibili.

Gli **algoritmi greedy** adottano la strategia di prendere la decisione che, al momento, appare come la migliore - una decisione che *localmente* è ottima, ma non è detto che porterà ad una soluzione ottima del problema nella sua globalità.

In generale, non sempre esiste un algoritmo greedy che determina la soluzione ottima di qualsiasi problema, ma gli algoritmi greedy sono vantaggiosi per risolvere alcune classi di problemi.

Proprietà della strategia greedy

1. Proprietà della **scelta greedy**: la modalità di scelta può dipendere dalle decisioni prese fino a quel punto ma non da quelle future o dalle soluzioni dei sottoproblemi (*top-down* versus *bottom-up* della programmazione dinamica). Si può ottenere una sol ottima globale prendendo delle decisioni che sono ottimi locali.

2. Un problema presenta una **sottostruttura ottima** se una soluzione ottima del problema contiene al suo interno una soluzione ottima dei sottoproblemi.

Introduzione al problema MST

Si supponga di voler connettere tutti i computer in un nuovo ufficio usando il minor numero di cavi possibile. Si modelli il problema usando un **grafo pesato G** i cui vertici rappresentano i computer e i cui lati rappresentano tutte le possibile coppie (u, v) di computer, in cui il peso $w(u, v)$ del lato (u, v) sia uguale alla quantità di cavo necessaria per connettere il computer v al computer u.

Si cerca di trovare un **albero T** che contenga tutti i vertici di G (**spanning**) e abbia, tra tutti gli alberi ricoprenti, il minimo costo complessivo (**minimum spanning tree**).

Definizione del problema MST

Dato un grafo pesato non orientato G, si cerchi un albero T che contenga tutti i vertici di G e che minimizzi:

$$w(T) = \sum_{(v,u) \text{ in T}} w(v,u).$$

Il problema si risolve con l'**algorimo di Kruskal** e quello di Prim-Jarník.

Proprietà: Se V_1 e V_2 formano una partizione di un grafo pesato connesso, ed f è un arco con costo minore tra quelli che hanno un vertice in V_1 e l'altro in V_2, allora esiste un minimun spanning tree T in cui f è uno dei suoi lati.

OSS: Se i pesi sono distinti, il MST è unico.

Algoritmo di Kruskal (G, w)

1. $A \leftarrow \emptyset$
2. **for** ogni vertice $v \in V(G)$
3. **do** *MakeSet*(v)
4. ordina gli archi di $E(G)$ per peso w non decrescente
5. **for** ogni arco $(u, v) \in E$ (in ordine di peso non decr)
6. **do if** *FindSet*$(u) \neq$ *FindSet*(v)
7. **then** $A \leftarrow A \cup \{(u, v)\}$
8. *Union*(u, v)
9. **return** A

Instrs 1-3 inizializzano A con l'insieme vuoto e creano $|V|$ alberi, uno per ogni vertice. Istr 4 ordina gli archi per peso non decrescente. Instrs 5-7 aggiungono ad A gli archi che non appartengono allo stesso albero, e 8 fonde i vertici dei due alberi (*FindSet*(u) e *FindSet*(v))..

Implementazione dell'algoritmo di Kruskal

L'implementazione dell'algoritmo di Kruskal sarà compito dell'esercitazione.

Cosa serve sapere: ADT Grafo, le strutture dati usate per implementare grafi, e i metodi $MakeSet(v)$, $FindSet(v)$, e $Union(v, u)$.

L'algoritmo usa una struttura dati per mantenere diversi insiemi disgiunti di elementi.

Interface Disjoint

```
public interface DisjointSet {

/**
* makeSet(x) crea un nuovo insieme singleton contenente
* l'oggetto x (che ne diviene il rappresentante). Dato che
* gli insiemi devono essere disgiunti, si richiede che x non
* sia già in un insieme. */

void makeSet(Object x);

/**
* findSet(x) restituisce un riferimento al rappresentante
* dell'insieme che contiene l'oggetto x, se esiste; null
* altrimenti */

Object findSet(Object x);
```

Interface Disjoint (II)

```
/**
* union(x,y) esegue l'unione dei due insiemi
* rappresentanti da x e y in un nuovo insieme che avrà
* come rappresentante o x o y. */

void union(Object x, Object y);

/**
* size() restituisce il numero di insiemi che compongono
la partizione */

int size(); }
```

Implementazione dei metodi

```
/**
* Un insieme disgiunto è rappresentato da un HashMap
* of Node. Si utilizza HashMap in quanto è possibile
* inserire e ricercare gli oggetti utilizzandoli come key
* della hash table.

* Volendo implementare la classe mediante foreste di
* alberi, la classe Node contiene l'oggetto, l'oggetto padre
* (è un albero particolare) e il rank.
* Il rank approssima il logaritmo della dimensione del
* sottoalbero ed è un upper bound all'altezza del nodo
*nell'albero. */

public class ForestDisjointSet implements DisjointSet {
```

Cenni alle HashMap (I)

Un'HashMap è un vettore con, al posto dell'indice sequenziale, una chiave essa stessa oggetto.

È quindi una generalizzazione di un array, e le differenze sono

	Array	*HashMap*
Chiavi	interi	Tipi e classi
Valori	Tipi e classi	Tipi e classi
Ordinam	Ordinato	Non ordinato
Getter	array[indice]	hashmap.get(objChiave)
Setter	array.push(objVal)	hashmap.put(objCh, objVal)

Cenni alle HashMap (II)

Un'istanza di HashMap ha due parametri: la **capacità iniziale** e il **"load factor"**. La capacità (iniziale) è la dimensione dell'Hash Table (quando viene creata). Il "load factor" misura di quanto si può riempire l'Hash Table prima che ne venga automaticamente aumentata la dimensione: quando il numero di elementi nell'hash table supera il prodotto dell'load factor per la capacità, allora questa viene circa duplicata (da metodi rehash).

In generale, un'hashmap vuota di default ha una capacità iniziale di 16 e un load factor di 0.75, che è un buon tradeoff tra i costi di tempo e di spazio.

public HashMap()

Implementazione del metodo size

```
public class ForestDisjointSet implements DisjointSet{

HashMap insiemi;
int nInsiemi;

public ForestDisjointSet(){
insiemi = new HashMap();
nInsiemi = 0;}

public int size(){
return nInsiemi;
}
```

Implementazione del metodo MakeSet

```
/**
* makeSet(Object x) crea un nuovo insieme singleton.
* Ricordarsi che tutti gli elementi devono essere inseriti la
* prima volta come insiemi singoletti. */

public void makeSet(Object x){

if (!insiemi.containsKey(x)) {
Node nodo = new Node(x, 0);
insiemi.put(x, nodo);
nInsiemi++;} }

// public boolean containsKey(Object key)
// Returns true if this map contains (a mapping for) the
specified key
```

Implementazione della classe Node

```
/**
* La classe Node rappresenta un nodo dell'albero
* utilizzato per rappresentare un insieme.
* DA NOTARE: in questo albero OGNI nodo punta al
* proprio padre */

class Node {

Object padre;
int rank;

Node() {
padre = null;
rank = 0; }

Node(Object p, int r) {  padre = p; rank = r; } }
```

Implementazione del metodo FindSet

```
/**
* findSet(Object x) restituisce il rappresentante
* dell'insieme che contiene l'oggetto x. Questa procedura
* implementa l'euristica di "path compression" che
* modifica (durante la ricerca) il padre di ciascun nodo di
* un albero, in modo tale che punti direttamente alla
* radice dell'albero. In questo modo si può dimostrare
* che il tempo di computazione per n operazioni di findSet
* e union si riduce considerevolmente. */

public Object findSet(Object x) {
Node nodo = (Node) insiemi.get(x); // recupero il nodo
nell'insieme
...
```

Implementazione del metodo union

```
/**
* union(Object x, Object y) esegue l'unione dei due
* insiemi che contengono gli oggetti x e y. * /

public void union(Object x, Object y){

link(findSet(x),findSet(y)); }

/**
* link(x, y) fonde i due alberi con radice x e y. La fusione è
* realizzata secondo l'euristica "union by rank".
* L'idea è far diventare l'albero con meno nodi figlio della
* radice dell'altro. Il rank deve essere aggiustato solo nel
* caso in cui i due alberi hanno il medesimo rank (in
* questo caso infatti l'albero risultante è più alto di 1) */

private void link(Object x, Object y) {
...
```

ADT Grafo

```
public class Grafo {
HashMap nodi;
int nArchi;

public Grafo() {
nodi = new HashMap();
nArchi = 0;
}

public int nodesNumber() {
return nodi.size();
}

public int edgesNumber() {

return nArchi;
}
```

Metodi di ADT Grafo

add(x) aggiunge un nodo al grafo con valore x se non esiste, nulla altrimenti. L'aggiunta di un nodo significa aggiungere la coppia (x, lista) nella HashMap dove lista è una HashSet nuovo vuoto.

```
public void add(Object x);

public void remove(Object x);

public boolean add(Object x, Object y, Object value);
```

// $add(x, y, v)$ aggiunge un arco tra i nodi x, y con peso v

Strutture Dati per Grafi

Di solito i grafi si rappresentano tramite **lista di lati**, **lista delle adiacenze**, e **matrice delle adiacenze**.

Le prime due memorizzano solo i lati presenti nel grafo, la terza memorizza un segnaposto per ogni coppia di vertici nel grafo ($m + n$ versus n^2 l'ordine della complessità dello spazio occupato).

Nella lista di lati, V è una collezione di oggetti vertice, memorizzati in un array list o in una lista a nodi.

Nella lista delle adiacenze, la più efficiente, un oggetto vertice mantiene un riferimento ad una collezione I(v) (**collezione di incidenza**), i cui elementi memorizzano riferimenti ai lati che sono incidenti in v, mentre un oggetto lato (con vertici terminali v e w) mantiene riferimenti agli elementi associati al lato nelle collezioni di incidenza ($I(v)$ e $I(w)$).

Compito dell'esercitazione

- Implementare l'ADT Grafo con metodi per restituire una collezione iterabile di: tutti i vertici, tutti i lati, e tutti i lati incidenti ad un nodo dato, e anche un metodo per verificare se due vertici sono adiacenti.
- Implementare l'algoritmo di Kruskal, utilizzando i metodi visti a lezione.