Appunti Algoritmi e Strutture Dati

Luca Seggiani

7 Maggio 2024

1 Grafi

Grafi orientati
Un grafo orientato $g(N, A)$ è un'insieme di N nodi e di $A \subseteq N \times N$ archi. Un'arco è una coppia ordinata di nodi. Se $(p, q) \subseteq A$, diciamo che p è predecessore di q, e che q è successore di p. Definiamo poi $n = |N|$ numero dei nodi e $m = |A|$ numero degli archi. Un grafo orientato con n nodi ha al massimo n^2 archi. Un **cammino** su un grafo orientato è una sequenza di nodi $(n_1, n_2..., n_k), k \geq 1$ tale che esiste un arco da n_i a n_{i+1} per ogni $1 \leq i \leq k$. La lunghezza del cammino è data dal numero degli archi. Un ciclo è un cammino che comincia e finisce sullo stesso nodo. Un grafo dove non esistono cammini ciclici si chiama aciclico.

Rappresentazioni dei grafi
Possiamo rappresentare i grafi in più modi:

- **Liste di adiacenza**
  Una lista di adiacenza è un modo di rappresentare i grafi attraverso una lista di nodi accessibili da ogni nodo, cioè:

```
struct Node {
  int nodeNumber;
  Node* next;
};
Node *graph[N]; //radice del grafo
```

  Si definisce un'array con un dimensione uguale al numero N di nodi. Ogni elemento dell'array è a questo punto uno struct che rappresenta un nodo e i suoi successori.

- **Matrici di adiacenza**
  Un'altro tipo di rappresentazione è attraverso le matrici di adiacenza. Si realizza una matrice quadrata $n \times n$ che rappresenta il prodotto

cartesiano dei nodi. L'elemento della matrice i, j sarà a questo punto 1 se esiste un arco dal nodo i al nodo j e 0 altrimenti: questo tipo di struttura dovrebbe essere già stata studiata ad algebra lineare. Si ricorda che la potenza n-esima di tale matrice rappresenta per ogni coppia di nodi il numero dei cammini di lunghezza n fra di essi.

Vediamo come gestire le informazioni immagazzinate dai grafi, attraverso le **etichette**:

- **Liste di adiacenza etichettate**
  Assegnamo un'etichetta sia ai nodi che agli archi:

```
struct Node {
  int nodeNumber;
  ArcType arcLabel;
  Node* next
};
Node* graph;
NodeType nodeLabels[N];
```

  Potremo allora immagazzinare un valore di tipo ArcType per ogni arco, e un valore di tipo NodeType per ogni nodo. Il NodeType di un nodo si potrà trovare per corrispondenza diretta con gli indici del vettore di liste, mentre l'ArcType sarà associato ad ogni elemento della lista di adiacenze del nodo.

- **Matrici di adiacenza etichettata**
  Per le matrici di adiacenza etichettate, possiamo usare un'approccio identico per i nodi, e indicare l'ArcType direttamente sulla matrice.

Visità in profondità

Vediamo come effettuare una visita su una struttura dati grafo: la visita in profondità. Nella visita in profondità gli archi vengono esplorati a partire dall'ultimo nodo esaminato che abbia ancora degli archi non esplorati uscenti da esso. Questo comportamento viene implementato scorrendo tutti i nodi, marchiando man di mano quelli che sono già stati visitati. I nodi appena visitati vengono "scorsi" fino in fondo dove non si trovano nodi già marchiati, e solo dopo si passa al nodo successivo. Definiamo la classe grafo basata sulle liste di adiacenza:

```
class Graph {
  struct Node {
    int nodeNumber;
    Node* next;
  };
  Node* graph[N];
```

```
  NodeType nodeLabels [N];
  int mark[N];
};
```

Il vettore mark[] rappresenta il marchio dato ai nodi visitati. A questo punto la visita in profondità potrà essere implementata come:

```
void nodeVisit(int i) {
  mark[i] = 1;
  //esamina nodeLabels[i]
  Node* q; int j;
  for(q = graph[i]; g; g = g->next) {
    j = g->nodeNumber;
      if(!mark[j]) nodeVisit(j);
  }
}

public:
void depthVisit() {
  for(int i = 0; i < N; i++)
    mark[i] = 0;
  for(i = 0; i < N; i++) {
    if(!mark[i])
      nodeVisit[i];
  }

}
```

Grafi non orientati

Un grafo non orientato $g(N, A)$ è una struttura non dissimile dal grafo orientato, ma dove le coppie che rappresentano gli archi non sono ordinate. Se $(p, q) \subseteq A$, diciamo che p è adiacente a q e viceversa. Un grafo non orientato con n nodi ha al massimo $\frac{n(n-1)}{2}$ archi. Un **cammino** su un grafo non orientato è una sequenza di nodi $(n_1, n_2..., n_k), k \geq 1$ tale che esiste un arco fra n_i a n_{i+1} per ogni $1 \leq i \leq k$. La lunghezza del cammino è data dal numero degli archi. Un ciclo è un cammino che comincia e finisce sullo stesso nodo, e non ha ripetizioni, eccetto per l'ultimo nodo. Un grafo non orientato è connesso se esiste sempre un cammino fra due nodi qualsiasi del grafo.

Rappresentazione dei grafi non orientati

Un grafo non orientato può essere rappresentato attraverso le tecniche viste per i grafi orientiati, assumendo per ogni arco dal nodo a al nodo b, un'arco inverso dal nodo b al nodo a. Naturalmente ogni arco del grafo non orientato sarà rappresentato due volte (nel caso della matrice di adiacenza, essa sarà simmetrica).

Multi-grafi orientati e non

Un multigrafo $m(N, A)$ è un'insieme di N nodi e di A insiemi di archi su

qualsiasi nodo. In sostanza, un multi-grafo elimina la dipendenza fra il numero di nodi e il numero di archi. I multi-grafi possono essere orientati e non, a seconda se le coppie che formano gli archi sono ordinate o no.

Minimo albero di copertura
Sappiamo che un grafo non orientato è connesso se esiste un cammino fra due nodi qualsiasi del grafo. Una **componente connessa** è un sottografo connesso del grafo. La componente connessa *massimale* è una componente connessa che non è contenuta in nessun'altra componenete connessa, ovvero una componente connessa che non è connessa a nessun'altro nodo tramite un'arco addizionale. A questo punto, un **albero di copertura** è un insieme di componenti connesse massimali acicliche, e il minimo albero di copertura è l'albero di copertura che ha somma dei pesi degli archi minima.

Algoritmo di Kruskal per il minimo albero di copertura
L'algoritmo di Kruskal trova il minimo albero di copertura di un grafo attraverso un'algoritmo greedy. Il processo dell'algoritmo si può ridurre a:

- Ordina gli archi del grafo in ordine crescente;
- Scorri l'elenco degli archi. Per ogni arco a:
  - Se a connette due componenti non connesse, scegli a e unifica le componenti.

Algoritmo di Dijkstra
L'algoritmo di Dijkstra serve a trovare i cammini minimi da un nodo di partenza a tutti gli altri nodi, su un grafo con archi pesati. Si tratta di un algoritmo greedy. Si adoperano due tabelle, che chiameremo *dist* (distanza) e *pred* (predecessore) con n elementi pari al numero dei nodi. Per ogni nodo a, dist(a) contiene in ogni momento la lunghezza di un cammino dal nodo iniziale ad a e pred(a) il predecessore di a in questo cammino. I nodi sono divisi in due insiemi: quelli già sistemati, ovvero per i quali i valori delle tabelle *dist* e *pred* sono già stati decisi definitivamente, e quelli da sistemare (che chiameremo insieme Q). Si noti che anche i nodi nell'insieme Q hanno valori in *dist* e *pred*, relativi al percorso migliore trovato fino a quel momento. Il processo può essere descritto come:

- Considera come "sistemato" il nodo con *dist* minore in Q e rimuovilo;
- Aggiorna *dist* e *pred* per ogni nodo immediatamente successore del nodo appena rimosso;
- Ripeti finche Q non è vuoto.

In pseudocodice molto approssimativo, possiamo dire:

```
Q = N;
per ogni nodo p diverso da p_0 {
  dist(p) = infty, pred(p) = vuoto;
}
dist(p_0) = 0;
while(Q contiene piu di un nodo) {
  estrai da Q il nodo p con minima dist(p);
  per ogni nodo q successore di p {
    lpq = lunghezza di arco (p, q);
    if(dist(p) + lpq < dist(q)) {
      dist(q) = dist(p) + lpq;
      pred(q) = p;
      reinserisci q in Q;
    }
  }
}
```

La struttura dati più adatta per Q è chiaramente un min-heap: vogliamo ad ogni iterazione restituire l'elemento minore.