Secondo progetto in itinere del corso di Ingegneria degli algoritmi a.a. 2017-2018

Andrea Graziani - matricola 0189326

20 gennaio 2018

Indice

1 Grafi e gradi di visibilità

1.1 Definizioni preliminari

Prima di descrivere e analizzare un algoritmo capace di calcolare il *grado di visibilità* di un nodo è essenziale dapprima precisare i termini che useremo nella nostra trattazione.

Definizione 1 *Sia dato un grafo diretto $G = (V, E)$ e un arco (u, v) con $u, v \in V$, si dice che il nodo u è **predecessore** del nodo v e che il nodo v è il **successore** del nodo u.*[1]

Definizione 2 *Un **cammino** in un grafo G da un vertice x ad un vertice y è dato da una sequenza di vertici $\langle u_0, u_1, u_2, ..., u_n \rangle$ con $v_0 = x$ e $v_n = y$, tale che, per $1 \leq i \leq n$, l'arco (u_{i-1}, u_i) appartiene a G.*[2]

Definizione 3 *Se esiste un cammino da un vertice x ad un vertice y di G, diremo anche che y è **raggiungibile** da x, ovvero che y è un **discendente** di x e che x è un **antenato** di y.*[3]

Ora abbiamo tutti gli strumenti per poter introdurre il concetto di visibilità.

Definizione 4 *Sia dato un grafo diretto aciclico non pesato $G = (V, E)$ in cui ad ogni nodo n sia associato un valore numerico intero `value`. Un nodo n_2 si definisce **visibile** da un nodo n_1 se e solo se valgono **entrambe** le seguenti condizioni:*

1. $n_1 \neq n_2$
2. *n_2 è **successore** di n_1 oppure un suo **discendente**; in quest'ultimo caso il cammino $\langle n_1, u_1, u_2, ...u_k, n_2 \rangle$ deve essere tale che $k \leq n_2$`.value`.*

Definizione 5 *Il **grado di visibilità** di un nodo n é pari al numero di nodi visibili da n.*

1.2 Rappresentazione del grafo

Come vedremo anche in base a valutazioni sperimentali, la scelta della struttura dati per rappresentare il grafo risulta essere cruciale in quanto essa influenza in modo preponderante le prestazioni dell'algoritmo che abbiamo utilizzato per calcolare il grado di visibilità di un nodo appartenente al grafo.

Nel nostro caso, dovendo privilegiare l'operazione che ci consenta di trovare tutti i successori di un generico vertice v all'interno di un grafo, la rappresentazione più idonea è risultata essere quella basata su *liste di adiacenza*. Secondo tale rappresentazione, ogni vertice v ha una lista contenente i suoi vertici adiacenti, ovvero tutti i vertici u per cui esiste un arco (v, u)[4]; avendo a che fare con grafi orientati, usando la definizione 1, si può affermare che la lista di adiacenza di un generico nodo v contiene tutti i suoi successori.

[1] Cfr. http://www.di.unito.it/ locatell/didattica/ro1/grafi-sl-bf.pdf, pp. 4

[2] Cfr. Camil Demetrescu & Irene Finocchi & Giuseppe F. Italiano - *Algoritmi e strutture dati, Seconda edizione*, McGraw-Hill, pp. 318

[3] *ibid.*

[4] Cfr. *ivi*, pp. 322-323

1.2.1 Vantaggi

Rispetto ad altre forme di rappresentazione, l'utilizzo delle liste di adiacenza risulta essere particolarmente vantaggiosa per lo scopo della nostra applicazione per due motivi:

Facile implementazione Implementare l'operazione che consenta di trovare i successori di un generico vertice v è molto semplice in quanto è sufficiente scandirne la lista di adiacenza.

Efficienza Trovare i successori di un generico nodo v risulta essere un operazione molto efficiente: il costo è pari infatti a $\delta(v)$, dove $\delta(v)$ rappresenta il grado del vertice v.[5] Anche le operazioni di inserimento di un nuovo arco e di un nuovo vertice sono molto efficienti: inserire un arco (u, v) implica inserire il vertice v all'interno della lista di adiacenza di u e ciò ha costo pari a $O(1)$.[6]

1.3 Calcolo del grado di visibilità

1.3.1 Calcolo del grado di visibilità di un singolo vertice

Descriviamo ora un algoritmo capace di calcolare il grado di visibilità di un generico vertice v di un grafo sfruttando adeguatamente le caratteristiche della forma di rappresentazione dei grafi mediante liste di adiacenza. Supponiamo quindi di avere un grafo $G = (V, E)$ nell'ipotesi descritte nella definizione 4 e di voler calcolare il grado di visibilità di un generico nodo $u \in V$.

Durante la k-esima iterazione, l'algoritmo trova e visita tutti i discendenti v di u per cui esiste un cammino orientato $\langle u, u_1, u_2, ...u_n, v\rangle$ tale che $n = (k-1)$; in altri termini, l'algoritmo trova ad ogni iterazione k tutti i discendenti v il cui cammino da u in v abbia un numero di nodi intermedi pari a $(k-1)$. Ad esempio, durante la seconda iterazione, l'algoritmo trova e visita tutti i nodi discendenti di u il cui cammino dal nodo u abbia un solo nodo intermedio.

Tuttavia verranno visitati e analizzati esclusivamente quei discendenti che risultino contrassegnati come *non visitati*; durante la suddetta operazione verranno verificate le condizioni di visibilità descritte nella definizione 4 aggiornando opportunamente il valore del grado di visibilità del nodo u. Tutti i nodi analizzati verranno quindi contrassegnati come *visitati* e inseriti all'intero di una coda C.

Alla $(k+1)$-esima iterazione, l'algoritmo provvederà a trovare e visitare tutti i successori dei nodi presenti nella coda C procedendo secondo le modalità precedentemente descritte. Qualora la coda C risultasse vuota, l'algoritmo, non avendo trovato altri discendenti di u, termina la sua esecuzione rimuovendo il flag di *visitato* a tutti i nodi analizzati durante l'intera computazione e restituendo il valore del grado di visibilità calcolato.

Detto questo è importante ora chiarire alcuni aspetti dell'algoritmo. Siano u e v due nodi appartenenti a G tali che v sia discendente di u. Possiamo affermare che:

1. L'ipotesi di aciclicità garantisce la non esistenza di cicli all'interno del grafo per cui non può esistere un cammino $\langle u, u_1, u_2, ...u_n, v\rangle$ tale che $u = v$.

[5] Cfr. *ivi*, pp. 323, Tabella 12.2
[6] *ibid.*

Ciò garantisce che la prima condizione di visibilità descritta all'interno della definizione 4 sia *sempre* verificata. Come si può facilmente intuire, l'assenza di cicli permette inoltre all'algoritmo di terminare.

2. Dal momento che può esistere più di un cammino tra i vertici u e v, corriamo il rischio di visitare più di una volta uno stesso nodo ottenendo perciò risultati erronei. L'uso di un apposito flag consente di risolvere il problema permettendoci di distinguere i vertici già visitati.

3. Affinché l'algoritmo possa essere correttamente eseguito, l'uso dei flag richiede tuttavia una fase di inizializzazione durante la quale tutti i nodi del grafo devono essere contrassegnati come non visitati. Per comprendere meglio le conseguenze della precedente constatazione facciamo un esempio: supponiamo che $E = \emptyset$, ovvero il grafo G non ha spigoli; in tal caso questa operazione di inizializzazione sarebbe completamente inutile e costosa poiché l'algoritmo non è in grado di trovare alcun discendete di u. Più in generale, in assenza di ipotesi sul numero di componenti connesse all'interno del grafo, probabilmente l'algoritmo visiterà solo un sottoinsieme $Q \subseteq V$ di vertici. Fatte queste considerazioni, per evitare di aggiornare inutilmente i flag di tutti i nodi a ogni esecuzione, si è preferito utilizzare una lista di supporto H all'interno della quale inserire tutti i nodi analizzati dall'algoritmo; quest'ultimo, al termine della computazione, aggiorna il flag solo per i nodi visitati, cioè per quelli contenuti all'interno di H.

L'algoritmo `calcNodeVisibilityDegree` usato per il calcolo del grado di visibilità di un generico nodo è riportato in figura **algorithm 1**.

1.3.2 Calcolo del vertice con massimo grado di visibilità

A questo punto trovare il nodo con massimo grado di visibilità è molto semplice in quanto richiede il calcolo del grado di visibilità di ogni nodo del grafo per poi isolare quello con grado di visibilità massimo mediante $(n - 1)$ confronti. Lo pseudo-codice dell'algoritmo, denominato `getMaxVisibilityDegreeNode`, è riportato in **algorithm 2**.

Algorithm 1 calcNodeVisibilityDegree

```
1:  function calcNodeVisibilityDegree(node)
2:      visibilityDegree ← 0
3:      supportList ← Nuova lista
4:      successorsNodeQueue ← Nuova coda
5:      intermediateNodes ← 0
6:      Inserisci node all'interno della coda successorsNodeQueue
7:      while not successorsNodeQueue.isEmpty() do
8:          successorsNodeQueueSize ← successorsNodeQueue.getSize()
9:          for successorsNodeQueueSize times do
10:             myNode ← successorsNodeQueue.dequeue()
11:             successors ← Estrai i successori di myNode
12:             for all node in successors do
13:                 if not node.isVisited then
14:                     node.isVisited ← True
15:                     if (intermediateNodes ≤ node.value) then
16:                         Incrementa visibilityDegree
17:                     Inserisci node in coda a successorsNodeQueue
18:                     Inserisci node in coda a supportList
19:         Incrementa intermediateNodes
20:     for all node in supportList do
21:         node.isVisited ← False
22:     return visibilityDegree
```

Algorithm 2 getMaxVisibilityDegreeNode

```
1:  function getMaxVisibilityDegreeNode(grafo G con n nodi)
2:      maxVisibilityNodeIdentifier ← 0
3:      maxVisibilityValue ← calcNodeVisibilityDegree(0)
4:      for nodeID in [1,n] do
5:          currentVisibilityValue ← calcNodeVisibilityDegree(nodeID)
6:          if currentVisibilityValue ≥ maxVisibilityValue then
7:              maxVisibilityNodeIdentifier ← nodeID
8:              maxVisibilityValue ← currentVisibilityValue
```

1.4 Costo computazionale

Sia dato un grafo diretto aciclico non pesato $G = (V, E)$ tale che $|V| = n$ e $|E| = x$. Supponiamo inoltre che il grafo sia connesso, ovvero esiste un cammino tra ogni coppia di vertici appartenenti al grafo G; siamo ovviamente nella situazione peggiore poiché in queste condizioni l'algoritmo dovrà tener conto di ogni vertice. Analizziamo quindi il tempo necessario per eseguire il calcolo del grado di visibilità di un generico nodo $u \in V$.

Durante la sua esecuzione, ad un certo punto l'algoritmo dovrà trovare tutti i discendenti di un generico nodo x a sua volta discendente di u; ciò richiede la scansione della lista di adiacenza di x e questa operazione richiede tempo $O(\delta(x))$. Successivamente occorre verificare le condizioni di visibilità per ogni discendente di x trovato e, se supponiamo che l'operazione di verifica della visibilità di un nodo possa essere svolta in tempo costante, anche questa operazione ha un costo pari a $O(\delta(x))$. È facile perciò convincersi che, in generale, elaborare un generico nodo x richieda tempo pari $O(2\delta(x))$ nel caso peggiore.

Pur essendo tuttavia possibile che alla k-esima iterazione l'algoritmo analizzi più di un vertice per volta, in base alle ipotesi è pur certo che il numero totale di nodi che verranno visitati sia pari n; di conseguenza possiamo affermare che il costo dell'algoritmo sia pari a $O(\sum_{v \in V} 2\delta(v))$ nel caso peggiore. Poiché per un qualunque grafo è vero che $\sum_{v \in V} \delta(v) = 2|E|$, possiamo concludere che il costo dell'esecuzione dell'algoritmo è pari a $O(4|E|)$ nel caso peggiore[7].

Trovare il nodo avente grado di visibilità massimo richiede innanzitutto il calcolo del grado di visibilità di ogni nodo del grafo: questa operazione richiede tempo pari a n volte il tempo richiesto per calcolare il grado di visibilità di un singolo nodo, ovvero $O(4n|E|)$. Isolare nodo avente grado di visibilità massimo viene fatto attraverso $(n-1)$ confronti[8]; il costo totale dell'algoritmo è quindi pari a $O(4n|E| + n - 1)$.

1.4.1 Costo computazionale con altre forme di rappresentazione

Per sottolineare l'importanza della forma di rappresentazione del grafo, abbiamo voluto testare lo stesso algoritmo anche su grafi che utilizzassero differenti strutture dati. In particolare abbiamo voluto rappresentare il nostro grafo mediante una *matrice di adiacenza* di cui forniamo la definizione:

Definizione 6 *Sia M una matrice, di dimensione $n \times n$, le cui righe e colonne sono indicizzate dai vertici del grafo. La* ***matrice di adiacenza*** *M è definita nel modo seguente:*

$$M[u, v] = \begin{cases} 1, & \textit{se } (u, v) \textit{ è un arco del grafo } G, \\ 0, & \textit{altrimenti.} \end{cases}$$

Sia dato un grafo diretto aciclico non pesato $G = (V, E)$ tale che $|V| = n$. Nonostante i diversi vantaggi computazionali, questa forma di rappresentazione non è adatta per la nostra applicazione poiché trovare tutti i successori di un

[7] Da ciò deduciamo che il tempo di esecuzione dell'algoritmo sia proporzionale al numero degli spigoli; questo risultato è confermato anche dai nostri risultati sperimentali: infatti, qualora applicassimo l'algoritmo a un grafo con n vertici senza spigoli, esso termina immediatamente a prescindere dal numero di vertici.

[8] Cfr. *ivi*, pp. 122-123

Tabella 1: Analisi dell'algoritmo `calcNodeVisibilityDegree`

Forma di rappresentazione	Tempo (s)
Matrice di adiacenza	0.00003538099917932414
Lista di adiacenza	0.00001901199721032754

Tabella 2: Analisi dell'algoritmo `getMaxVisibilityDegreeNodeID`

Forma di rappresentazione	Tempo (s)
Matrice di adiacenza	0.00013200300236348994
Lista di adiacenza	0.00006565399962710217

generico nodo v risulta molto costoso in quanto è necessario esaminare tutte le posizioni $M[v, \cdot]$ della matrice, ovvero scandire un'intera riga, anche nel caso in cui ci sono pochi vicini di v; di conseguenza il tempo richiesto per trovare tutti i successori di un nodo è sempre pari $O(n)$[9]; ciò implica un tempo quadratico per il calcolo del grado di visibilità di un nodo. Per rendersi conto delle differenze tra le due forme di rappresentazione, potete osservare i risultati sperimentali riassunti all'interno della tabella 1 e 2.

2 Implementazione

Da un punto di vista implementativo, la realizzazione dell'applicazione è stata effettuata utilizzando le seguenti classi:

`AcyclicDirectGraph` una classe astratta usata per modellare un comune grafo aciclico orientato.

`AdjacencyListGraph` figlia della classe precedente, modella un grafo rappresentato mediante liste di adiacenza.

`AdjacencyMatrixGraph` figlia della classe `AcyclicDirectGraph`, modella un grafo rappresentato mediante matrice di adiacenza.

`GraphNode` modella un nodo di un comune grafo orientato.

`GraphNode` questa classe, figlia di quella precedente, modella un nodo di un grafo rappresentato mediante liste di adiacenza.

`Queue` questa classe è stata utilizzata per modellare una coda.

Per ovvi motivi di spazio, omettiamo in questa sede una descrizione dettagliata delle varie classi e dei loro metodi in ogni caso già presente all'interno del codice sorgente. Ricordiamo che per ottenere informazioni dettagliate

[9] Cfr. *ivi*, pp. 324-325

sull'interfaccia di tutti i metodi definiti nelle varie classi è sufficiente digitare `help(<nome della classe>)` dalla console dell'interprete *Python* dopo aver eseguito l'importazione della classe desiderata.