Briscese Filippo Maria 0228612
Ingegneria degli Algoritmi A.A. 2017-2018 - Prova Pratica: Seconda Prova in Itinere

Traccia 1 - *Grado di visibilità*

Descrizione dell'algoritmo

Per risolvere la richiesta della traccia ho modificato una classe tra quelle mostrate in aula, `GraphAdjacencyList`, implementando nuovi metodi e attributi, e modificando, dove necessario, alcuni metodi.

La richiesta viene risolta dal metodo `maxGradoVisibilita`, che esegue il metodo `gradoVisibilita` per ogni nodo del grafo e ne ritorna quello con il grado di visibilità maggiore. Qui di seguito è illustrato il funzionamento di `gradoVisibilita`.

» `gradoVisibilita(rootID)` : Calcola il grado di visibilità del nodo in input, effettuando una sorta di visita in profondità del grafo; i criteri per la visita sono:
- Un nodo viene visitato solo se, nel momento dell'analisi, si giunge a questo percorrendo il cammino minimo che lo unisce al nodo di partenza.
- Se il nodo che sto analizzando non è adiacente al nodo di partenza, ed il suo valore è minore del valore massimo tra i nodi incontrati nel tragitto percorso fino al nodo in analisi, non continuare più in profondità.

Se il nodo in analisi rispetta la definizione di nodo visibile, viene aggiunto in una lista, `visibili`, dove sono salvati gli id dei nodi visibili dal nodo di partenza (in questa lista è presente anche il nodo di partenza stesso). Infine viene restituito il numero di nodi all'interno di `visibili` meno 1, vista la presenza del nodo di partenza in essa.
Per verificare le distanze tra i nodi legge nell'attributo del grafo `distanze`, che viene aggiornato prima di eseguire la visita sopra descritta chiamando il metodo `distanzeDaNodo(rootID)`; questo metodo esegue una visita in ampiezza del grafo e, visto che la traccia chiede che gli archi del grafo abbiano costo unitario, aggiorna l'attributo `distanze` del grafo tenendo conto che la distanza tra due nodi è pari al "livello" in cui si trova il nodo in analisi rispetto alla radice nell'albero BFS.

Due parole infine sul nuovo attributo del grafo, `distanze`: esso è un dizionario di dizionari, ovvero nel dizionario `distanze` ogni id dei nodi del grafo viene usato come chiave per un dizionario nel quale gli id dei nodi sono usati come chiavi per le distanze

tra i nodi con id le due chiavi precedenti. Probabilmente per capire cosa e come è stato implementato per l'attributo `distanze` è più facile leggere questo esempio: `{ IDu:{ IDv: distanza(u,v)}` .

Ho deciso di usare un dizionario di dizionari (tipo `dictionary`, built-in di Python) per non avere le preoccupazioni sugli indici che avrei avuto se avessi scelto di usare liste di liste, tutto questo perché in un grafo orientato, presi due nodi u e v, non necessariamente è possibile raggiungere uno partendo dall'altro (es. $u, v \in V$, v non raggiungibile da u. Nel caso delle liste avrei dovuto settare `distanze[u][v] = float("inf")`, invece con i dizionari evito direttamente di inserire la coppia `(v:distanza)` nel dizionario assegnato alla chiave u nel dizionario `distanze`).

Analisi tempi teorici

- `gradoVisibilita`
  Questo metodo non è altro che una particolare visita di un grafo, effettuata dopo una visita BFS dello stesso grafo (eseguendo `distanzeDaNodo`). Visto che il grafo è implementato per liste di adiacenza, come abbiamo visto a lezione, una visita costa $O(m + n)$; in particolare effettuare due visite costa ugualmente $O(m + n)$.
- `maxGradoVisibilita`
  Esegue `gradoVisibilita` per ogni nodo del grafo, il suo costo sarà dunque $n \cdot O(m+n) = O(m \cdot n + n^2)$.

Analisi tempi sperimentali

# nodi	10	25	50	100	150	200	250	300	350
Tempo	0.00164	0.01446	0.08360	0.73011	3.06304	7.90510	18.47217	39,28023	71.57239

Tabella 1. *Tempo medio di esecuzione di* `maxGradoVisibilita` *al variare del numero dei nodi del grafo. Tempi in secondi.*

Per calcolare i tempi sperimentali su un caso più generico possibile ho implementato `randomGrafoAciclico(n)`, un metodo che crea un grafo diretto aciclico connesso con n nodi ed un numero casuale di archi. L'idea dietro a questo metodo è che presa la lista dei nodi in un grafo senza archi, è possibile rendere il grafo diretto aciclico evitando che, preso un nodo alla posizione i nella lista, esista un arco dal nodo in posizione i a un qualsiasi nodo in posizione $j < i$. La ricerca della creazione di un caso

più casuale possibile si concretizza quindi nell'implementazione, qui sono descritte in breve le operazioni che si possono leggere nel codice di questo metodo:

1. Crea n nodi, assegnando ad ognuno un valore casuale in $[0, 999]$.
2. Per ogni nodo: stabilisci quali sono i suoi "successivi", scegli un numero casuale tra 0 e il numero di successivi per sapere quanti nodi "puntare", tra i successivi scegli casualmente tanti nodi quanti il numero casuale di "puntati" ottenuto prima, infine crea gli archi.
3. Prima di concludere controlla che non ci siano nodi "isolati" intersecando l'insieme (nel codice classe `set` built-in di Python) dei nodi "pozzo" (cioè nodi con soli archi entranti) e l'insieme dei nodi "sorgente" (cioè nodi con soli archi uscenti); se esiste qualche nodo in questa intersezione; prendine uno alla volta, scegli un nodo a caso tra i suoi "successivi" e crea un arco; continua fino a quando l'intersezione è vuota, infine ritorna il grafo.

È importante però mettere bene in chiaro che questo metodo non crea un grafo esattamente come quello ipotizzato nella traccia, infatti il grafo casuale creato non necessariamente collegherà due suoi nodi tra loro con un unico cammino minimo; ciò comunque non influisce sulla correttezza dell'algoritmo da me proposto in `maxGradoVisibilita`, che, nel caso due nodi siano collegati tra loro da più di un cammino minimo, deciderà di considerare un nodo come visibile da un altro nodo se tra tutti i cammini minimi che li collegano, è possibile vederlo attraversandone almeno uno (naturalmente in caso di più cammini minimi che rispettino le condizioni dette, il nodo non sarà considerato visibile più volte, infatti verrà sempre eseguito il controllo di esistenza del nodo in analisi nell'insieme `explored` proprio della visita). I tempi di esecuzione di `randomGrafoAciclico` non ci interessano e pertanto non verranno considerati.

I tempi (si veda *Tabella 1*) per un grafo con n nodi, sono stati dunque presi usando il modulo `time` per misurare le durate di 10 esecuzioni di `maxGradoVisibilita` su 20 grafi diversi creati con `randomGrafoAciclico`.

Mettendo i dati in *Tabella 1* su un grafico si ottiene quanto è possibile vedere in *Figura 1*. I risultati sperimentali non tradiscono i tempi teorici; in *Figura 1* si può notare che i tempi medi quasi si posano sulla curva verde disegnata da una funzione di terzo grado[1]. Questo è facilmente comprensibile cercando di ottenere il valore m in

[1] : Funzione ottenuta usando "fit" seguito da tutte le coppie (#nodi, tempo medio) su WolframAlpha.com , comando che cerca di calcolare una curva che passa per i punti in input.
Il risultato ottenuto è $f: y = 3.4765 \cdot (10^{-6})\ x^3 - 0.00082x^2 + 0.0679x - 1.08822$.

funzione di n : in un generico grafo il valore m è compreso tra $(n-1)$ e $(n(n-1))/2$, e per come è implementata la funzione `randomGrafoAciclico` è estremamente probabile che il valore di m sia un $O(n^2)$, piuttosto che $O(n)$, questo perché per ottenere $O(n)$, nel passo 2 di `randomGrafoAciclico` scritto sopra, il valore casuale corrispondente al numero di "successivi" da puntare dovrà essere costante e piccolo durante tutta l'esecuzione di questo passo, cosa estremamente difficile che accada e che diventa sempre più improbabile all'aumentare del numero dei nodi del grafo. Quanto appena trattato rende possibile concettualizzare il costo di `maxGradoVisibilita` in $O(n^2 + n \cdot n^2) = O(n^3)$.

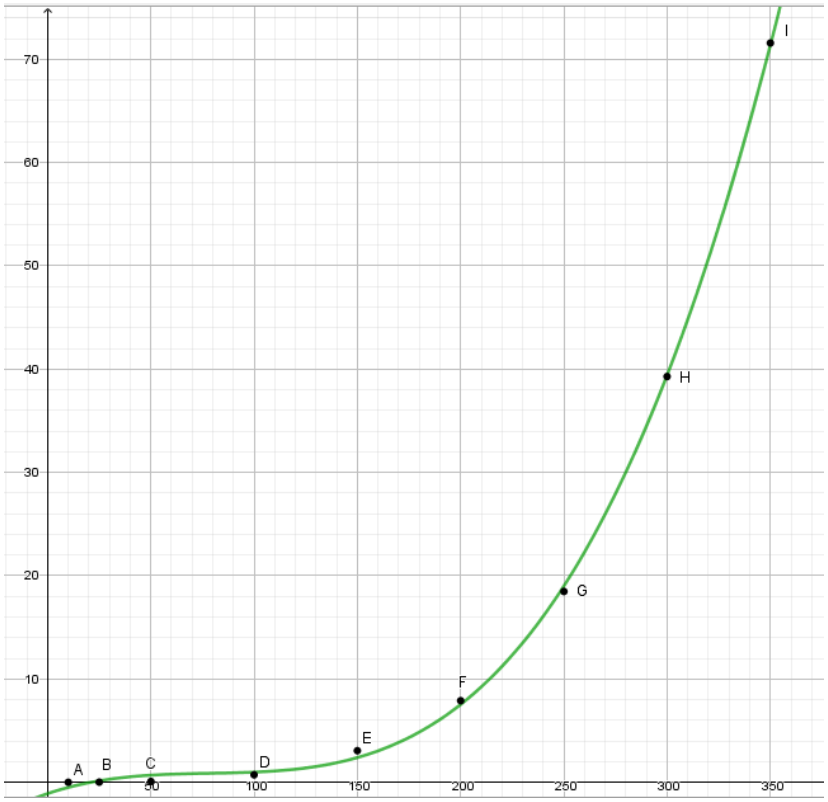

Figura 1. *Risultati sperimentali su un grafo, insieme ad un polinomio di terzo grado* $y = ax^3 + bx^2 + cx + d$.

Note finali:

- Tutto il lavoro è stato svolto su questo hardware: CPU Intel Z3735F, RAM 2GB, SO Windows 10, Pycharm17.2.3.
- Nel cercare di creare per i test un caso più casuale possibile che rientrasse nei casi della traccia, ho provato svariate idee matematiche e di programmazione, ma risultavano tutte troppo complicate oppure non casuali poiché creavano grafi con strutture determinate in partenza. Qui risiede la decisione di fare un passo indietro e di usare per i test grafi come descritto in precedenza, chiudendo un occhio sull'unicità dei cammini minimi tra nodi, e mettendo in chiaro il comportamento dell'algoritmo, risultando in un lavoro estremamente più semplice e proporzionato a quello che eravamo interessati. Probabilmente però l'algoritmo avrà prestazioni migliori su grafi che rientrano esattamente nel caso definito dalla traccia.
- Alla riga 374 del file `NewGraph_AdjacencyList.py` ho scritto che un grafo è diretto aciclico se la matrice della sua rappresentazione per matrice di adiacenza (con i nodi ordinati sulle righe e sulle colonne secondo l'ordine topologico) è triangolare superiore. Ho scritto in questo modo perché spiegarlo attraverso liste di adiacenza risultava più lungo e meno chiaro. Inoltre questa "idea", che riflettendo velocemente sembra vera, non è stata da me dimostrata; non ho cercato se questo fosse un risultato già dimostrato da qualche parte, e non so se esista una dimostrazione... Magari averi potuto evitare tutto questo discorso dicendo semplicemente: *«... la dimostrazione è lasciata al lettore per esercizio»*.