Esercitazione 1

Problema 1. In un corso di laurea sono previsti un certo numero di esami obbligatori. Esistono inoltre dei vincoli di propedeuticità: se un esame A è propedeutico a un esame B allora B deve essere sostenuto in una sessione di esami successiva a quella in cui è stato sostenuto A. Se due o più esami non violano alcun vincolo di propedeuticità allora possono essere sostenuti tutti all'interno della stessa sessione . Supponendo di superare ogni esame al primo tentativo fornire un algoritmo lineare che calcoli il minimo numero di sessioni necessarie per sostenere tutti gli esami. Per ogni sessione d'esami, fornire inoltre l'insieme di esami da sostenere.

Suggerimento: Considerare il grafo delle propedeuticità e le sue proprietà strutturali.

Soluzione problema 1. Dato in input un insieme di esami $C = \{c_1, ..., c_n\}$ e un insieme di propedeuticità $P = \{p_1, ..., p_m\}$, dove ogni generico elemento p_i è una coppia di esami (c_h, c_k) e indica l'esistenza di una propedeuticità tra c_h e c_k, si chiede di trovare un insieme (ordinato) di sessioni $S = \{S_1, ..., S_t\}$. Ogni elemento S_i conterrà il sottoinsieme di esami che saranno sostenuti nella sessione i-esima. Per ogni propedeuticità (c_h, c_k) si ha il vincolo di collocare c_h in un S_i e c_k in un S_j tali che $i < j$. Si vuole inoltre che $|S|$ sia minima.
L'idea è quella di modellare l'insieme degli esami e delle propedeuticità con un grafo diretto (grafo delle propedeuticità) e sfruttare eventuali proprietà di tale grafo per trovare una soluzione al problema. In particolare l'insieme degli esami costituirà i nodi del nostro grafo, mentre ci sarà un arco diretto (u, v) se e soltanto se esiste la coppia di esami relativa in P. Il semplice algoritmo seguente costruisce il grafo dati C e P.

Algorithm 1: *buildGraph*

```
   Data: C, P
   Result: G = (V, E)
1  begin
2      V ⟵ C
3      E ⟵ ∅
4      forall (u, v) ∈ P do
5          E ⟵ E ∪ (u, v)
```

La prima osservazione importante è che –per fortuna degli studenti– il grafo ottenuto non può avere cicli (o non sarebbe possibile laurearsi), ovvero esso è un DAG. Una volta ottenuto il grafo delle propedeuticità, l'algoritmo individua tutti i nodi sorgente del grafo (nodi con grado entrante zero), assegna gli esami alla prima sessione e rimuove tali nodi e tutti i loro archi uscenti dal grafo. È facile osservare che rimuovendo un sottoinsieme di nodi e i loro archi uscenti da un DAG il grafo risultante continua ad essere un DAG. Quindi, l'algoritmo individuerà le nuove sorgenti, assegnerà tali esami alla seconda sessione e come prima tali nodi saranno rimossi dal grafo. In generale l'algoritmo durante l'iterazione i-esima individua tutte le sorgenti, assegna gli esami alla sessione i-esima e rimuove tali nodi e i lori archi uscenti dal grafo. L'algoritmo termina quando tutti i nodi sono stati rimossi.

Affinché ad ogni iterazione le sorgenti siano facilmente individuabili verranno calcolati preventivamente gli *in_degree* (gradi entranti) di ogni nodo con la seguente procedura:

Algorithm 2: *compute_in_degree*

```
   Data: G = (V, E)
1  begin
2      forall u ∈ V do
3          in_degree(u) ⟵ 0
4      forall u ∈ V do
5          forall (u, v) ∈ E do
6              in_degree(v) ⟵ in_degree(v) + 1
```

Ora descriviamo l'algoritmo che calcola gli esami da dare in ogni sessione. Riportiamo un possibile pseudo-codice:

Algorithm 3: *findSessions*

```
   Data: C, P
   Result: S
1  begin
2      S ⟵ ∅
3      G = (V, E) ⟵ buildGraph(C, P)
4      calculate_in_degree(G)
5      i ⟵ 1
6      S_i ⟵ ∅
7      forall u ∈ V do
8          if in_degree(u) == 0 then
9              S_i ⟵ S_i ∪ {u}
10     S ⟵ S ∪ {S_i}
11     while G is not empty do
12         S_{i+1} ⟵ ∅
13         forall u ∈ S_i do
14             forall (u, v) ∈ E do
15                 E ⟵ E\{(u, v)}
16                 in_degree(v) ⟵ in_degree(v) − 1
17                 if in_degree(v) == 0 then
18                     S_{i+1} ⟵ S_{i+1} ∪ {v}
19             V ⟵ V\{u}
20         S ⟵ S ∪ {S_{i+1}}
21         i ⟵ i + 1
22     return S
```

Andiamo ora ad argomentare sulla complessità dell'algoritmo proposto, nel caso in cui per rappresentare il grafo venga usata una lista di adiacenza.

Dati gli insiemi C, P in input di dimensione n e m rispettivamente, il grafo delle propedeuticità avrà n nodi e m archi e quindi la sua dimensione in memoria sarà $O(m+n)$. La costruzione della lista di adiacenza (algoritmo *buildGraph*) ha complessità temporale $O(m+n)$, mentre per il calcolo iniziale degli *in_degree* (procedura *calculate_in_degree*) è sufficiente una scansione della lista di adiacenza, cosa che richiede tempo $O(m+n)$. La determinazione dell'insieme S_1 (righe 7-9) costa tempo $O(n)$. Per determinare le restanti sessioni d'esame (righe 11-21) si osservi che ogni nodo del grafo viene aggiunto una ed una sola volta ad un qualche S_i, e quindi viene valutato una ed una sola volta nel ciclo 13-19. Inoltre, per ogni nodo viene letta la sua lista di adiacenza al fine di rimuovere gli archi uscenti e decrementare l'*in_degree* dei vicini corrispondenti, cosa che richiede tempo proporzionale al grado uscente del nodo. Da ciò segue che la complessità è ancora pari ad una scansione della lista di adiacenza: $O(m+n)$ (si ricorda che la somma dei gradi uscenti di tutti i nodi è pari a m).

In definitiva è stato quindi trovato un algoritmo di complessità totale $O(m+n)$, quindi lineare nella dimensione degli insiemi in input (come richiesto dall'esercizio).

Problema 2. Su un grafo non orientato e non pesato $G = (V, E)$ si possono muovere due robot telecomandati a distanza. All'inizio i due robot sono posizionati su due nodi del grafo, diciamo s_1 ed s_2. In ogni istante di tempo è possibile effettuare la seguente mossa: ordinare ad uno dei due robot di spostarsi dal nodo su cui è a un nodo adiacente (percorrendo un arco del grafo). L'obiettivo è portare il robot che si trova su s_1 in un certo nodo t_1 ed il robot che si trova su s_2 in un certo nodo t_2. Le antenne dei robot, però, soffrono di problemi di interferenze: se i robot finiscono troppo vicini l'uno con l'altro non riescono più a ricevere il segnale e quindi a muoversi. Per questo motivo si vuole che i robot in ogni istante di tempo siano sempre a distanza reciproca (nel grafo) di almeno k, dove k è un parametro del problema. Progettare un algoritmo che trovi il numero minimo di mosse che porta i robot nelle posizioni desiderate.

Soluzione Esercizio 2 L'idea è quella di servirci di un grafo ausiliario che riesca a modellare nella sua topologia le posizioni ammissibili (a distanza almeno k) che i robot possono avere sul grafo originale, e le possibili mosse effettuabili dai robot ancora una volta non violando il vincolo sulla distanza.

Chiameremo tale grafo come *grafo delle configurazioni* e sarà indicato con $G' = (V', E')$. Nel grafo delle configurazioni i nodi sono una coppia *ordinata* $\langle u, v\rangle$ dove u e v sono nodi del grafo G, un nodo $\langle u, v\rangle$ appartiene a V' se e soltanto se $d_G(u, v) \geq k$, dove $d_G(u, v)$ denota la distanza in G fra u e v.

Formalmente G' è definito come segue :

- $V' = \{\langle u, v\rangle : u, v \in V \text{ e } d_G(u, v) \geq k\}$
- $E' = \{(\langle u, v\rangle, \langle x, y\rangle) : (u = x \text{ e } (v, y) \in E) \text{ o } (v = y \text{ e } (u, x) \in E)$

Intuitivamente i nodi in V' rappresentano tutti i possibili modi in cui i robot possono trovarsi senza violare il vincolo relativo all'interferenza delle antenne: uno nodo $\langle x, y\rangle$ indica che il robot 1 si trova su x e il robot 2 su y. Gli archi invece, modellano le possibili mosse: c'è un arco fra due configurazioni se è possibile passare da una all'altra spostando uno dei due robot lungo un arco.

Una volta comprese le proprietà di G' è facile convincerci che per trovare la sequenza di spostamenti minimi che risolva il problema basta calcolare un cammino minimo tra $\langle s_1, s_2\rangle$ e $\langle t_1, t_2\rangle$ in G'. Naturalmente si sta supponendo che le posizioni iniziali e target rispettino il vincolo sulla distanza, ciò fa si che i nodi in questione si trovino in V'. Nel caso in cui in G' non esista alcun cammino tra il nodo sorgente e il nodo destinazione, non è possibile trovare una sequenza di mosse dei robot che risolvano il problema.

Ma quanto è complicato costruire il grafo G'? Quale sarà la sua dimensione in relazione alla dimensione di G? Quanto costerà calcolare il cammino minimo richiesto su G'?.

Diamo innanzitutto una stima sulla dimensione di G' :

Abbiamo visto che i nodi di G' corrispondono a una coppia ordinata di elementi in V, da ciò deriva che $|V'| \leq n^2$ e quindi $|V'| = O(n^2)$.

Per quanto riguarda gli archi, con una prima stima grossolana possiamo dire che $|E'| = O(n^4)$ in quanto al più gli archi di un grafo possono essere quadratici rispetto ai nodi. Tuttavia è possibile fare un analisi più raffinata che ci da un bound migliore. Denotiamo con $\delta_H(x)$ il grado di un generico nodo x in un generico grafo H. Abbiamo:

$$|E'| \leq \sum_{\langle u,v\rangle \in V'} \delta_{G'}(\langle u, v\rangle) \leq \sum_{\langle u,v\rangle \in V'} (\delta_G(u) + \delta_G(v)) = \sum_{\langle u,v\rangle \in V'} \delta_G(u) + \sum_{\langle u,v\rangle \in V'} \delta_G(v) \leq$$
$$\sum_{u,v \in V} \delta_G(u) + \sum_{u,v \in V} \delta_G(v) \leq n \sum_{u \in V} \delta_G(u) + n \sum_{v \in V} \delta_G(v) \leq 2nm + 2nm = O(nm)$$

Andiamo ora a vedere la complessità che ha la costruzione del grafo G' e il calcolo del cammino minimo su di esso. Per costruire il grafo è necessario calcolare le distanze tra tutte le coppie di nodi in V in modo da poter aggiungere a V' solo quelle a distanza almeno k. Ricordando che tramite la visita BFS è possibile calcolare le distanze rispetto ad un nodo sorgente, per trovare le distanze fra tutte le coppie è necessario effettuare n visite BFS (ogni volta utilizzando un nodo diverso come sorgente). Avendo una visita costo $O(m)$ (utilizzando liste di adiacenza) avremo che il costo di costruzione del grafo è $O(nm)$. Una volta costruito G' per trovare il cammino minimo da $\langle s_1, s_2\rangle$ a $\langle t_1, t_2\rangle$ utilizzeremo una visita BFS su G' utilizzando come sorgente il nodo $\langle s_1, s_2\rangle$. Tale computazione avrà costo $O(nm)$. In definita in tempo totale $O(nm)$ è possibile trovare la sequenza di mosse per i robot.

Problema 3 È dato un grafo non orientato $G = (V, E)$. Si vogliono posizionare delle monete sui vertici di G. Per posizionare una moneta, questa deve prima essere piazzata su di un vertice x di G non ancora occupato e successivamente spostata su un vertice adiacente ad x, anch'esso non occupato. Progettare un algoritmo lineare che trovi il modo di posizionare il maggior numero di monete possibili sui vertici di G.

Suggerimento: Pensare al caso in cui G è un albero ed estendere la soluzione trovata a grafi generici.

Soluzione Problema 3 Una prima domanda che possiamo farci è : Quale è il numero massimo di monete che posso posizionare su un grafo con n nodi? Un upper bound banale è $n - 1$: ogni mossa mi richiede che due nodi del grafo siano liberi (e adiacenti); quindi non potrei mai posizionare l'n-esima moneta.

Ma esistono istanze di grafi su cui è possibile posizionare $n-1$ monete? Utilizzando il suggerimento mostriamo come su alberi radicati è *sempre* possibile posizionare $n - 1$ monete.

L'idea è quella di cominciare a posizionare monete sulle foglie, posizionandole dapprima sui loro padri e facendole scorrere su ogni foglia. Una volta che le foglie saranno tutte coperte si andranno a posizionare le monete sui loro padri seguendo la stessa procedura. L'algorimo viene iterato fino al posizionamento delle monete sui nodi del livello 1. Solo la radice non potrà essere coperta da una moneta, in quanto non ha per definizione un padre su cui posizionare dapprima la moneta, e ha tutti i figli già coperti. Tale procedura può essere implementata tramite una visita in profondità (postordine) dell'albero, di seguito è riportato l'algoritmo:

Algorithm 4: *placeCoinsOnTree*

```
   Data: Tree T, Node r
1  begin
2      if r is a not a leaf then
3          forall child v of r do
4              placeCoinsOnTree(T,v)
5          forall child v of r do
6              place coin on r and move it to v
```

Come si può estendere tale algoritmo a grafi generici? Semplice: si può prima effettuare una qualsiasi visita sul grafo ottenendo così un albero di copertura, e poi si può utilizzare l'algoritmo appena visto su tale albero. L'algoritmo finale è quindi il seguente :

Algorithm 5: *placeCoinsOnGraph*

```
   Data: G = (V, E)
1  begin
2      Let v be any node of V
3      T ⟵ BFS(G, v)
4      placeCoinsOnTree(T,v)
```

Per quanto riguarda la complessità si deve prima pagare una visita del grafo, la cui compessità è $O(m+n)$ (se si utilizzano le liste di adiacenza). Poi, il posizionamento delle monete sull'albero ha complessità pari a una visita DFS dell'albero, quindi $O(n)$. Restringendoci a istanze in cui G è connesso ($m \geq n-1$), si ha che la complessità totale dell'algoritmo è $O(m)$.