UNIVERSITÀ DEGLI STUDI DI PERUGIA
Dipartimento di Matematica e Informatica

ARTIFICIAL INTELLIGENT SYSTEMS : INTELLIGENT APPLICATION DEVELOPMENT

Cammino lungo (ricerca in profondità)

Studente:

Fabrizio Fagiolo

Professore:

Prof. Stefano Marcugini

Anno accademico 2021/2022

Indice

1 Obbiettivo

Nel seguente progetto si va ad implementare un grafo G=(V,E) un grafo, in cui ogni arco ha associata una lunghezza e sia K un intero.
Dati due vertici determinare, se esiste un cammino di lunghezza almeno K.
Il problema verrà risolto utilizzando una **ricerca in profondità**.

2 Algoritmo utilizzato

La ricerca in profondità, è un tipo di algoritmo **non informato**.
La ricerca parte dalla radice ed espande il successore della radice andando al livello più profondo dell'albero, fino a che il nodo non ha successori. Quando il nodo è espanso completamente, viene rimosso dalla pila e la ricerca torna indietro al nodo successivo piu' superficiale che ha ancora successori non esplorati.
Caratteristiche algoritmo:

Completo?

L'algoritmo è completo solo se lo spazio degli stati è **finito**.
Altrimenti se si presenta uno spazio degli stati infinito l'algoritmo **non arriverà mai ad una soluzione.**

Ottimo?

Non è ottimo in quanto andando ad espandere il successore della radice ed i suoi successori in profondità, se la soluzione si trova nel primo successore il cammino sarà ottimo in tutti gli altri casi no.

Complessità in tempo?

Limitata dallo spazio degli stati.
$O(b^m)$ dove m è la profondità massima di 1° nodo.

Complessità in spazio?

O(bm) se lo spazio è lineare.
Con fattore di ramificazione b e profondità massima m. Una volta che un nodo è stato espanso può essere rimosso non appena i suoi discendenti sono stati esplorati completamente.

3 Definizione problema

3.1 Implementazione grafi

Per risolvere il problema definito precedentemente, sono andato a costruire 3 grafi.
I grafi all'interno del progetto sono stati costruiti mediante una funzione che, dato un nodo, restituisce la lista dei suoi successori all'interno del grafo utilizzando il **pattern matching**.
Nel caso in cui venga passato un valore per un nodo che non è stato definito all'interno della funzione (dei successori), tramite la variabile **muta ("_")** sarà possibile dare come risultato una **lista vuota**.
É stato inoltre definito un nuovo tipo chiamato "graph" tramite un **costruttore polimorfo** di nome "Graph" avente tipo 'a $->$ 'a list.
Il costruttore viene utilizzato per distinguere il tipo dei grafi dal tipo 'a $->$ 'a list.
Per calcolare la lunghezza dei cammini, la lista degli archi con il rispettivo peso è stata definita mediante una lista di triple di elementi interi, in cui una generica tripla ha il seguente significato:

1. nodo partenza
2. peso arco
3. nodo arrivo

Si assumono anche che i pesi siano non negativi e diversi da zero.

I tre grafi sui quali verrà applicata la funzione proposta per la ricerca di un cammino di lunghezza almeno K sono chiamati grafo1, grafo2 e grafo3.
Avremmo quindi i seguenti tipi:

La lista degli archi ha tipo:

(int * int * int) list, ovvero una lista composta da terne di elementi interi.

La funzione che restituisce i successori di un nodo all'interno del grafo:

ha tipo int $->$ int list $=<$ fun $>$, ovvero una funzione che dato un intero ritorna una lista di interi

I grafi hanno tipo:

int graph = Graph <fun>.
Sono costituiti da il costruttore "Graph" e la funzione dei successori.

Eccezioni:

Sono anche presenti 2 eccezioni che sono:

- CamminoNonTrovato
- Errore

che verrannò invocate quando opportuno.

```
exception CamminoNonTrovato;;
exception Errore;;
```

Tipo grafo:

Definizione del nuovo tipo chiamato "graph" tramite un costruttore polimorfo di nome "Graph" avente tipo: 'a − > 'a list.

```
type 'a graph = Graph of ('a -> 'a list);;
```

3.2 Grafo1

```
let pesi1 = [(0,1,1);(0,1,5);(1,1,3);(2,1,0);(3,1,4);(3,1,2)];;

let f1 = function
    0 -> [1;2]
  | 1 -> [3]
  | 2 -> [3]
  | 3 -> [4]
  | 4 -> []
  | _ -> [];;

let grafo1 = Graph f1;;
```

Figura 1: Codice definizione grafo1

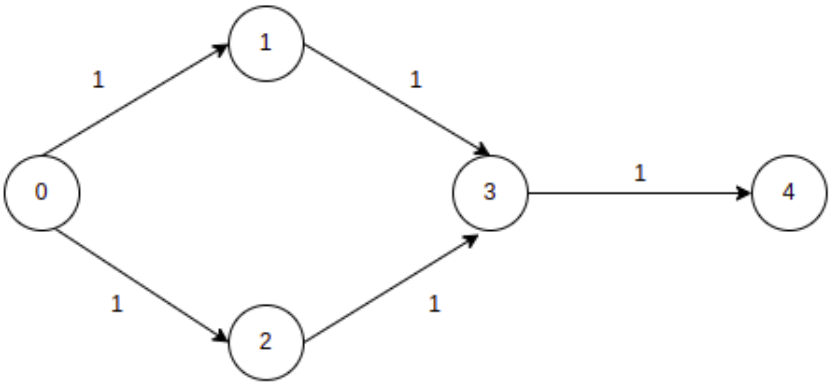

Figura 2: Grafo1

3.3 Grafo2

```
let pesi2 =[(0,1,1);(0,1,5);(1,1,3);(2,1,0);(3,1,4);(3,1,2)];;

let f2 = function
    0-> [1;5]
  | 1 -> [3]
  | 2 -> [0]
  | 3 -> [4;2]
  | 4 -> []
  | 5 -> []
  | _ -> [];;

let grafo2 = Graph f2;;
```

Figura 3: Codice definizione grafo2

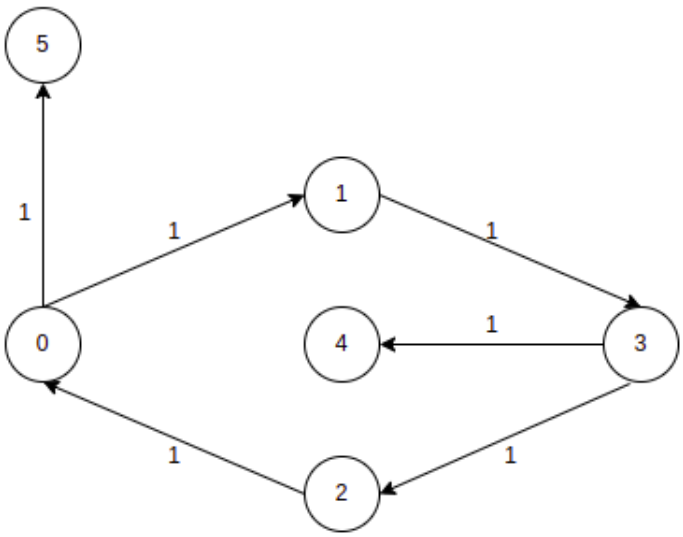

Figura 4: Grafo2

3.4 Grafo3

```
let pesi3 = [(0,2,1);(1,3,3);(2,1,1);(3,2,4);(3,1,2);(4,1,5)];;

let f3 = function
    0-> [1]
  | 1 -> [3]
  | 2 -> [1]
  | 3 -> [4;2]
  | 4 -> [5]
  | 5 -> []
  | _ -> [];;

let grafo3 = Graph f3;;
```

Figura 5: Codice definizione grafo3

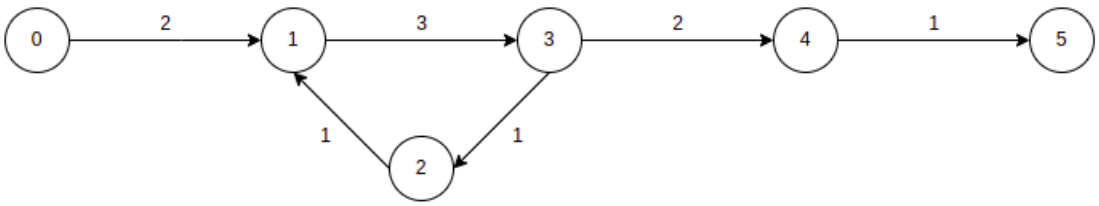

Figura 6: Grafo3

4 Funzioni Utilizzate

4.1 Funzione cerca

Funzione cerca: prende due coppie di interi, e una lista di lunghezze, e restituisce la lunghezza associata all'arco che ha come estremi proprio i due nodi.
Viene utilizzata all'interno della funzione **costo_cammino**.

'a $->$ 'b $->$ ('a $*$ int $*$ 'b) list $->$ int

```
let rec cerca x y= function
    []->0
  |(a,b,c)::resto -> if((a=x && c=y))then b else cerca x y resto;;
```

La funzione ritorna **il valore 0** se la lista è vuota.
Altrimenti nel caso in cui il valore **a sia uguale al valore x** dove x è inserito come primo parametro e **c sia uguale a y** dove y è passato come parametro, **ritorna il valore b.**
Altrimenti richiama la funzione cerca con x y e la lista resto.

4.2 Funzione costo_cammino

Funzione costo_cammino: prende come parametro una lista che in questo caso rappresenta un cammino "ipotetico" ed i pesi definiti precedentemente, grazie alla funzione cammino_ottimo è in grado di calcolare il costo del cammino passato come parametro basandosi sui pesi.

'a list $->$ ('a $*$ int $*$ 'a) list $->$ int

```
let costo_cammino cammino pesi=
  let rec cammino_ottimo costo=function
      []->raise Errore
    |x::y::rest -> cammino_ottimo ( costo + cerca x y pesi ) (y::rest)
    |_::[]->costo
  in cammino_ottimo 0 cammino;;
```

4.3 Funzione stampalista

Funzione stampalista: è una funzione di debug stampa semplicemente la lista del cammino ogni volta che questa viene chiamata.

int list $->$ unit

```
let rec stampaLista = function [] -> print_newline()
                          | x::rest -> print_int(x); print_string("; "); stampalista rest;;
```

4.4 Funzione dfs

1. **Funzione dfs:** questa funzione effettua la ricerca in profondità andandogli a passare come parametro il nodo da dove si vuole iniziare la ricerca, il nodo dove si vuole terminare la ricerca ed il grafo su cui si vuole effettuare la ricerca.

   'a $->$ 'a $->$ 'a graph $->$ 'a list

```
let dfs inizio fine (Graph succ) =
  let estendia cammino = (*stampalista cammino;*)
    List.map (function x -> x::cammino)
      (List.filter (function x -> not (List.mem x cammino)) (succ (List.hd cammino)))
  in let rec search_auxa fine = function
        [] -> raise CamminoNonTrovato
      | cammino::rest ->
          if ((fine = List.hd cammino) ) then List.rev cammino
          else search_auxa fine ((estendia cammino) @ rest)
  in search_auxa fine [[inizio]];;
```

   Utilizza la funzione **estendia cammino** che mappa tutte le x presenti in cammino e va a filtrare le x che non sono membre di cammino.

   La funzione **estendia** va a filtrare anche i successori della testa della lista cammino.

   La funzione ausiliaria **search_aux** prende il parametro fine e se la lista è vuota ritorna l'eccezione **CamminoNonTrovato**, altrimenti per ogni elemento della lista controlla se il parametro fine è uguale al primo elemento in testa alla lista cammino, se lo è allora il cammino viene restituito andandolo a stampare all'inverso per ottenere l'ordine giusto.

   Altrimenti i suoi figli vengono aggiunti in testa alla lista utilizzata per la ricerca.

   La lista è quindi utilizzata come Stack (politica LIFO) per una ricerca in profondità.

4.5 Funzione dfs_bool

2. **Funzione dfs_bool:** esegue una ricerca in profondità andando ad analizzare tutti i nodi del grafo. Restituerà un valore booleano dopo aver visitato l'intero grafo. Se è presente il nodo finale all'interno del grafo ritornerà il valore **true** sennò ritornerà la lista vuota ed il valore **false**.

Anche qui andremo a passare come parametri alla funzione il nodo di inizio del grafo il nodo finale ed il grafo.

'a $->$ 'a $->$ 'a graph $->$ bool

```
let dfs_bool inizio fine (Graph succ) =
  let estendia cammino = (*stampalista cammino;*)
    List.map (function x -> x::cammino)
      (List.filter (function x -> not (List.mem x cammino)) (succ (List.hd cammino)))
  in let rec search_auxa fine = function
        [] -> false
      | cammino::rest ->
          if ((fine = List.hd cammino) ) then true
          else search_auxa fine ((estendia cammino) @ rest)
  in search_auxa fine [[inizio]];;
```

4.6 Funzione dfs_cerca

3. **Funzione dfs_cerca:** esegue una ricerca in profondità considerando anche i cicli, non appena soddisfa il k, controlla se il cammino è esattamente la soluzione andando a verificare anche il valore booleano della funzione **dfs_bool**, altrimenti, ritorna alla fine chiamando la funzione ausiliaria **search_aux**

$$\text{int} -> \text{int} -> \text{int} -> (\text{int} * \text{int} * \text{int})\ \text{list} -> \text{int graph} -> \text{int list}$$

```
let dfs_cerca inizio fine k pesi (Graph succ)=
  let estendi cammino = stampalista cammino;
    List.map (function x -> x::cammino)
      (succ (List.hd cammino))
  in let rec search_aux fine = function
        [] -> raise CamminoNonTrovato
      | cammino::rest ->
          if ((costo_cammino (List.rev cammino) pesi >=k) && (List.hd cammino = fine) )then List.rev cammino
          else if((costo_cammino (List.rev cammino) pesi) >=k) && (dfs_bool (List.hd cammino) fine (Graph succ))
          then (List.rev (List.tl cammino)) @ dfs (List.hd cammino) (fine) (Graph succ)
          else if((costo_cammino (List.rev cammino) pesi) < k)
          then search_aux fine ((estendi cammino) @ rest)
          else search_aux fine rest
  in search_aux fine [[inizio]];;
```

Utilizza la funzione ausiliaria estendi cammino che mappa tutte le x presenti nella lista cammino e i successori in testa a cammino.

Utilizza poi la funzione ausiliaria **search_aux** che prende la fine come parametro. Se questa la lista è vuota, presenta solleva l'eccezione **CamminoNonTrovato**.

Altrimenti per ogni elemento della lista cammino per prima cosa vado a controllare se il costo del percorso è >= k e l'ultimo nodo è il nodo finale, se così è restituisco quel percorso, andando a fare il reverse della lista per stampare nell'ordine giusto il cammino.

Altrimenti se il costo totale del percorso è >= k ma l'ultimo nodo non è il nodo finale, trova un percorso che colleghi l'ultimo nodo di questo percorso al nodo finale usando la dfs. Ovviamente si va anche a controllare che il percorso dal nodo iniziale al nodo finale esista usando la funzione **dfs_bool**. Se questo esiste si effetuano i seguenti passaggi:

(a) Si prende la lista contenente il cammino trovato, si rimuove la testa e successivamente viene stampata la lista nell'ordine corretto facendo il reverse della stessa. In questo caso la lista conterrà il cammino dal

nodo iniziale al penultimo nodo trovato (siccome la testa viene rimossa precedentemente).

(b) Successivamente si genera un'altra lista contente il cammino dal nodo iniziale della lista precedente (ovvero quello rimosso) al nodo finale. La lista verrà ovviamente rovesciata per stamparla nel senso giusto.

(c) Per ottenere il cammino finale si andranno a concatenare le 2 liste create.

Altrimenti, se il costo non è ancora k si richiama la funzione **estendi cammino** e si cerca un altro percorso nella lista 'rest'.

Il cammino finale sarà quindi composto dalla concatenazione di un cammino che permette di raggiungere **un costo** $>=$ **k** ed un **cammino che termina sul nodo finale.**

Nel caso in cui il percorso non esista solleva l'eccezione **CamminoNonTrovato**.

4.7 Funzione trova_cammino

4. **Funzione trova_cammino:** è una funzione che serve a richiamare le funzioni precedenti (**dfs_cerca**). E che stampa i risultati ottenuti.

5. int $->$ int $->$ int $->$ (int $*$ int $*$ int) list $->$ int graph $->$ unit

```
let trova_cammino inizio fine k pesi (Graph succ)=
  let risultato = (dfs_cerca inizio fine k pesi (Graph succ)) in
  print_string("il risultato trovato è il seguente = "); stampalista risultato ;
  print_string("il costo del cammino è il seguente = "); print_int (costo_cammino risultato pesi); print_newline();;
```

5 Risultati

In questo capitolo saranno mostrati i risultati ottenuti dalle varie prove effettuate.

5.1 Esempi grafo1

Nella sezione seguente tutti gli esempi saranno effettuati sul grafo "grafo1" il quale è un **DAG** (grafo diretto aciclico), i cui archi sono rappresentati dalla lista chiamata "pesi1".

1. Caso analizzato DAG

   La figura 7 rappresenta la ricerca di un cammino che parte dal nodo 0, termina sul nodo 4 e che abbia costo almeno 3. In questo caso, quando si visita il nodo finale, il costo del cammino è >= k e quindi viene sollevato il primo if della funzione **dfs_cerca.**

   Il cammino trovato è [4; 3; 1; 0].

   Viene stampata la lista in maniera inversa e successivamente viene stampato anche il suo costo.

```
#trova_cammino 0 4 3 pesi1 grafo1 ;;
0;
1; 0;
3; 1; 0;
il risultato trovato è il seguente = 0; 1; 3; 4;
il costo del cammino è il seguente = 3
```

Figura 7: Primo esempio sul grafo1

2. Caso analizzato DAG

   Nella figura 8 è mostrata la ricerca di un cammino che parte dal nodo 0 al nodo 4 e che costi almeno 15.

   Questo cammino con questo determinato costo, nel grafo1 **non esiste**, quindi viene sollevata l'eccezione "CamminoNonTrovato".

   Com'è possibile vedere dallo screen viene effettuata la ricerca su tutto il grafo prima di lanciare l'eccezione.

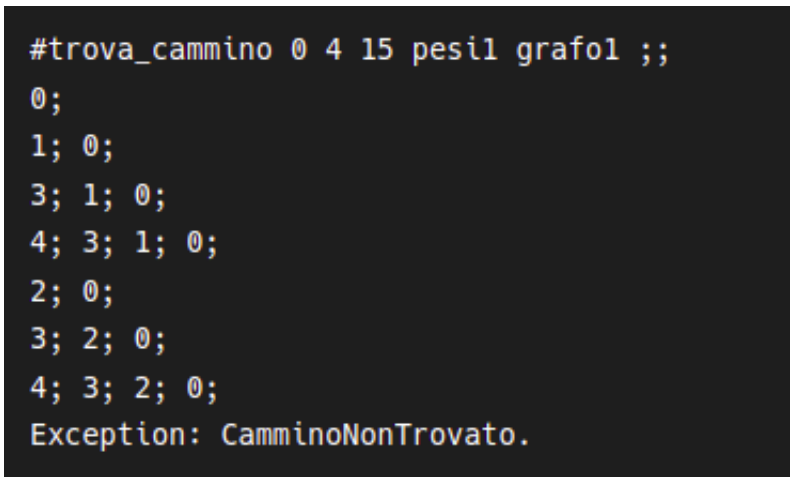

Figura 8: Secondo esempio sul grafo1

3. Caso analizzato DAG

   Nella figura 9 è mostrata la ricerca di un cammino che parte dal nodo 0 al nodo 20 (il quale non è presente nel grafo) e che costi almeno 20.

   Com'è possibile vedere dallo screen viene effettuata la ricerca su tutto il grafo prima di lanciare l'eccezione.

```
#trova_cammino 0 20 20 pesi1 grafo1 ;;
0;
1; 0;
3; 1; 0;
4; 3; 1; 0;
2; 0;
3; 2; 0;
4; 3; 2; 0;
Exception: CamminoNonTrovato.
```

Figura 9: Terzo esempio sul grafo1

5.2 Esempi grafo2

Nella sezione seguente tutti gli esempi saranno effettuati sul grafo "grafo2", i cui archi sono rappresentati dalla lista chiamata "pesi2".
In questo caso il grafo analizzato presenta **un ciclo**.

1. Caso grafi ciclico

   La figura 10 rappresenta la ricerca di un cammino che parte dal nodo 0, termina sul nodo 4 e che abbia costo almeno 3. In questo caso come precedentemente nel dag, quando si visita il nodo finale, quindi il costo del cammino è >= k e quindi viene sollevato il primo if della funzione **dfs_cerca**.

   Il cammino trovato è [4; 3; 1; 0].

   Viene stampata la lista in maniera inversa e successivamente viene stampato anche il suo costo.

```
#trova_cammino 0 4 3 pesi2 grafo2 ;;
0;
1; 0;
3; 1; 0;
il risultato trovato è il seguente = 0; 1; 3; 4;
il costo del cammino è il seguente = 3
```

Figura 10: Primo esempio sul grafo2

2. Caso grafo ciclico

   Nella figura 11 è mostrata la ricerca di un cammino che parte dal nodo 0 al nodo 5, che ha un costo almeno pari a 8. In questo caso il cammino che si ottiene per arrivare ad un costo >= k **non termina esattamente sul nodo finale**, viene quindi sollevato il secondo if della funzione **dfs_cerca**.

   La lista L analizzata che ha un costo maggiore o uguale a k è la seguente:

   L = [0; 2; 3; 1; 0; 2; 3; 1; 0]. Il nodo in testa a questa lista, ovvero 0, non è il nodo finale, quindi si effettuano i seguenti passaggi:

   (a) Si prende la lista L senza l'elemento in testa (in questo caso il nodo 0) viene rovesciata per ottenere un cammino che parte dal nodo iniziale e termina nel penultimo nodo trovato. Questo cammino verrà chiamato A. In questo caso abbiamo A = [0; 1; 3; 2; 0; 1; 3; 2]

   (b) Si cerca un cammino che parte dall'elemento in testa alla lista L (in questo caso 0) e che termina nel nodo finale. Si rovescia questa lista e che chiamiamo B. In questo caso abbiamo B = [0; 5]

   (c) Per ottenere il cammino finale si va a concatenare A e B (A @ B).

   Il cammino finale sarà quindi composto dalla concatenazione di un cammino che permette di raggiungere un costo >= k ed un cammino che termina sul nodo finale.

   Il cammino trovato in questo caso è: [0; 1; 3; 2; 0; 1; 3; 2; 0; 5] che ha un costo pari a 9.

```
#trova_cammino 0 5 8 pesi2 grafo2 ;;
0;
1; 0;
3; 1; 0;
4; 3; 1; 0;
2; 3; 1; 0;
0; 2; 3; 1; 0;
1; 0; 2; 3; 1; 0;
3; 1; 0; 2; 3; 1; 0;
4; 3; 1; 0; 2; 3; 1; 0;
2; 3; 1; 0; 2; 3; 1; 0;
il risultato trovato è il seguente = 0; 1; 3; 2; 0; 1; 3; 2; 0; 5;
il costo del cammino è il seguente = 9
```

Figura 11: Secondo esempio sul grafo2

3. Caso grafo ciclico

   Nel caso in cui venga passato all'algoritmo un nodo non esistente nel grafo, l'algoritmo solleverà l'eccezione **CamminoNonTrovato** dopo aver visitato tutto il grafo con i passi indicati.

```
#trova_cammino 0 29 28 pesi2 grafo2 ;;
0;
1; 0;
3; 1; 0;
4; 3; 1; 0;
2; 3; 1; 0;
0; 2; 3; 1; 0;
1; 0; 2; 3; 1; 0;
3; 1; 0; 2; 3; 1; 0;
4; 3; 1; 0; 2; 3; 1; 0;
2; 3; 1; 0; 2; 3; 1; 0;
0; 2; 3; 1; 0; 2; 3; 1; 0;
1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
4; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
4; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
4; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
4; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
5; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
5; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
5; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
5; 0; 2; 3; 1; 0; 2; 3; 1; 0; 2; 3; 1; 0;
5; 0; 2; 3; 1; 0; 2; 3; 1; 0;
5; 0; 2; 3; 1; 0;
5; 0;
Exception: CamminoNonTrovato.
```

5.3 Esempi grafo3

Nella sezione seguente tutti gli esempi saranno effettuati sul grafo "grafo3", i cui archi sono rappresentati dalla lista chiamata "pesi3".
In questo caso **gli archi non presentano tutti lo stesso peso** ed il grafo come nel caso precedente, presenta **un ciclo**.

1. Caso grafo archi con pesi diversi

   La figura 13 rappresenta la ricerca di un cammino che parte dal nodo 0, termina sul nodo 3 che ha a costo almeno 5. In questo caso quando si visita il nodo finale **il costo del cammino è >= k** e quindi viene sollevato il primo if della funzione dfs_cerca.

   Il cammino trovato è [3; 1; 0].

   Viene stampata la lista in maniera inversa e successivamente viene stampato anche il suo costo.

```
#trova_cammino 0 3 5 pesi3 grafo3 ;;
0;
1; 0;
il risultato trovato è il seguente = 0; 1; 3;
il costo del cammino è il seguente = 5
```

Figura 12: Primo esempio sul grafo3

2. Caso grafo archi con pesi diversi

   Nella figura 14 è mostrata la ricerca di un cammino che parte dal nodo 0 al nodo 5, che ha un costo almeno pari a 10. In questo caso il cammino che si ottiene per arrivare ad un costo >= k **non termina esattamente sul nodo finale**, viene quindi sollevato il secondo if della funzione **dfs_cerca**.

   La lista L' analizzata che ha un costo maggiore o uguale a k è la seguente:

   L' = [3; 1; 2; 3; 1; 0]. Il nodo in testa a questa lista, ovvero 3, non è il nodo finale, quindi si effettuano i seguenti passaggi:

   (a) Si prende la lista L' senza l'elemento in testa (in questo caso il nodo 3) viene rovesciata per ottenere un cammino che parte dal nodo iniziale e termina nel penultimo nodo trovato. Questo cammino verrà chiamato A. In questo caso abbiamo A = [0; 1; 3; 2; 1]

   (b) Si cerca un cammino che parte dall'elemento in testa alla lista L' (in questo caso 3) e che termina nel nodo finale. Si rovescia questa lista e che chiamiamo B. In questo caso abbiamo B = [3; 4; 5]

   (c) Per ottenere il cammino finale si va a concatenare A e B (A @ B).

   Il cammino finale sarà quindi composto dalla concatenazione di un cammino che permette di raggiungere un costo >= k ed un cammino che termina sul nodo finale.

   Il cammino trovato in questo caso è: [0; 1; 3; 2; 1; 3; 4; 5] che ha un costo pari a 13.

```
#trova_cammino 0 5 10 pesi3 grafo3 ;;
0;
1; 0;
3; 1; 0;
4; 3; 1; 0;
5; 4; 3; 1; 0;
2; 3; 1; 0;
1; 2; 3; 1; 0;
il risultato trovato è il seguente = 0; 1; 3; 2; 1; 3; 4; 5;
il costo del cammino è il seguente = 13
```

Figura 13: secondo esempio sul grafo3

3. Caso archi con pesi diversi

   Nel caso in cui venga passato all'algoritmo un nodo non esistente nel grafo, l'algoritmo solleverà l'eccezione **CamminoNonTrovato** dopo aver visitato tutto il grafo con i passi indicati.

```
#trova_cammino 0 23 44 pesi3 grafo3 ;;
0;
1; 0;
3; 1; 0;
4; 3; 1; 0;
5; 4; 3; 1; 0;
2; 3; 1; 0;
1; 2; 3; 1; 0;
3; 1; 2; 3; 1; 0;
4; 3; 1; 2; 3; 1; 0;
5; 4; 3; 1; 2; 3; 1; 0;
2; 3; 1; 2; 3; 1; 0;
1; 2; 3; 1; 2; 3; 1; 0;
3; 1; 2; 3; 1; 2; 3; 1; 0;
4; 3; 1; 2; 3; 1; 2; 3; 1; 0;
5; 4; 3; 1; 2; 3; 1; 2; 3; 1; 0;
2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
4; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
5; 4; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
4; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
5; 4; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
4; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
5; 4; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
4; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
5; 4; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
4; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
5; 4; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 0;
Exception: CamminoNonTrovato.
```

6 Conclusioni

Il programma implementato è quindi in grado di trovare una soluzione se il nodo finale è presente nel grafo.
Come detto precedentemente se il nodo finale e il costo del cammino sarà >= k si troverà nel successore iniziale della radice per trovare la soluzione l'algoritmo impiegherà pochissimi passi per restituire il cammino. Nel caso in cui questo non si verificasse, l'algoritmo impiegherà piu' tempo per trovare il **cammino soluzione**.
Come visto in precedenza invece l'algoritmo restituirà l'eccezione **CamminoNonTrovato** nel caso in cui andremo a cercare un nodo non presente nel grafo.