Università di Perugia
Dipartimento di Matematica e Informatica

Relazione per progetto di Artificial Intelligent Systems - Intelligent Application Development

Il problema del postino di campagna: ricerca in ampiezza

Professore
Prof.re Stefano Marcugini

Studente
Chiara Luchini

Anno Accademico 2020-2021

Indice

Elenco dei codici sorgente

Elenco delle figure

1 Introduzione

L'obiettivo di questo progetto è di creare un codice che permetta di trovare, dato un grafo G con V vertici ed E archi ai quali viene associata una lunghezza, un circuito o cammino chiuso che parte da un vertice specifico v_0 definito dall'utente. Questo cammino chiuso deve inoltre verificare determinate proprietà, ovvero che contenga tutti gli archi del sottoinsieme E' e che abbia una lunghezza totale minore di un intero K. Per trovare il circuito si utilizza una ricerca in ampiezza, inoltre si suppone che il grafo G sia un grafo indiretto.
Nelle prossime sezioni riporterò le strutture dati utilizzate e le funzioni implementate, inserendo anche degli esempi per una miglior comprensione e verifica del codice.

2 Strutture dati

Le strutture dati utilizzate sono le seguenti:

```
type 'a graph =  {nodi : 'a list; archi : ('a*'a*'a) list; lunghezza: 'a list;};;
type 'a grafo_suc = Graph_suc of ('a -> 'a list );;
```

Listing 1: Strutture dati utilizzate.

Nel primo caso ho definito una tipo di dato per il grafo G, il quale è costituito da tre campi diversi ovvero:

- **nodi**: questa è una lista di interi che identifica i nodi del grafo G;
- **archi**: è una lista di triple costituita da tre valori interi che corrispondono rispettivamente ai nodi che compongono l'arco e la lunghezza di quest'ultimo. Ad esempio l'arco (1,2,5) indica l'arco costituito dai nodi (1,5) con lunghezza associata pari a 2, quindi in generale i valori esterni corrispondono ai nodi e quello interno alla lunghezza;
- **lunghezza**: è una lista di interi, come la lista di nodi, corrisponde a tutte le lunghezze degli archi.

Ho suddiviso in questo modo il tipo grafo poichè risulta più semplice accedere a determinati campi, ad esempio se ho bisogno di lavorare con i nodi mi basterà utilizzarli tramite `grafo.nodi`.

```
let grafo1 =
    {nodi = [1; 2; 3; 4; 5 ; 6 ];
     archi = [(1,2,5); (1,4,3); (2,3,4); (2,10,3);
                (3,6,4); (4,5,6); (5,7,6)];
     lunghezza =[2;4;3;5;6]};;
```

Listing 2: Esempio definizione grafo.

Il secondo tipo definito in 1 viene usato per rappresentare il grafo tramite una funzione "successori", ovvero ad ogni nodo viene associata la lista di nodi ad esso vicini. Questa funzione viene impiegata spesso per la risoluzione di vari problemi sui grafi.

```
(** funzione che permette di trovare tutti i vicini di un nodo *)
let ricerca_vicini grafo x =
    let rec ricerca vicini = function
        []-> vicini
        |(a,b,c)::rest -> if a=x then ricerca (c::vicini) rest
            else if x=c then ricerca (a::vicini) rest
            else ricerca vicini rest
    in ricerca [] grafo;;
```

Listing 3: Funzione ricerca vicini.

Inoltre ho creato questa funzione `ricerca_vicini` per ricercare tutti i vicini o "successori" di un determinato nodo. Essa prende in input gli archi del grafo e il nodo del quale dobbiamo cercare i vicini e restituisce in output una lista di nodi successori.

```
let fun_suc1 = function
      1 -> ricerca_vicini grafo1.archi 1
    | 2 -> ricerca_vicini grafo1.archi 2
    | 3 -> ricerca_vicini grafo1.archi 3
    | 4 -> ricerca_vicini grafo1.archi 4
    | 5 -> ricerca_vicini grafo1.archi 5
    | 6 -> ricerca_vicini grafo1.archi 6
    |_ -> [];;

let g_s1 = Graph_suc fun_suc1;;
```

Listing 4: Funzione successori.

3 Funzioni ausiliare

In questa Sezione riporto alcune delle funzioni ausiliare utilizzate con annessa spiegazione.

```
exception Numero_negativo;;

(* restituisce il valore passato k passato in input dall'utente *)
let get_k  =
    print_string ("Inserire il numero K:");
    let s = read_int() in
    if s > 0 then s else raise Numero_negativo;;

(* stampa il cammino *)
let rec stampalista = function
    [] -> print_newline()
    | x::rest -> print_int(x); print_string(";"); stampalista rest;;

(**stampa il cammino come lista di archi *)
let rec stampaarchi = function
    [] -> print_newline()
    | (x,y)::rest -> Printf.printf "(%i,%i)" x y; print_string(";");
                     stampaarchi rest;;
```

Listing 5: Funzioni get_k, stampalista e stampaarchi.

Queste tre semplici funzioni permettono rispettivamente di prendere il valore K passato in input dall'utente, di stampare una lista nel nostro caso la lista che identifica il cammino e di stampare una lista di tuple.

```
(* ricava la lunghezza associata all'arco passato (x,y) *)
let rec valore_arco x y = function
    [] -> 0
    | (a,b,c)::rest -> if (x=a && y=c) ||
                         (x=c && y=a)then b else valore_arco x y rest;;
(* calcola il costo del cammino *)
let costo_del_cammino cammino archi =
    let rec sum costo = function
        []-> raise Errore
        |_::[]-> costo
        |x::y::rest -> sum ( costo + valore_arco x y archi) (y::rest)
in sum 0 cammino ;;
```

Listing 6: Funzioni valore_arco e costo_del_cammino.

Le prima funzione ricorsiva restituisce il valore associato all'arco con nodi x e y, ovvero la lunghezza di quest'ultimo. Questa può ritornare due valori:

- 0 : nel caso in cui non è stato trovato l'arco con nodi x e y, ovvero (x,y) o (y,x);
- b : nel caso in cui l'arco è stato trovato.

La seconda invece calcola il costo del cammino totale, essa prende in input il cammino trovato e gli archi del grafo in questione. Essa possiede diversi valori di ritorno:

- raise Errore : solleva l'eccezione Errore quando abbiamo passato un cammino vuoto;
- costo : nel caso in cui abbiamo calcolato il valore di tutto il cammino;
- `sum (costo + valore_arco x y archi) (y::rest)`: caso ricorsivo in cui stiamo ancora calcolando il costo del cammino.

```
(** trasforma la lista cammino in una lista di archi *)
let trasforma_cammino cammino =
    let rec crea acc = function
        [] -> []
        |_::[] -> acc
        | x::(y :: _ as rest)-> crea ((x,y) :: acc) rest
    in List.rev (crea [] cammino);;

(*controlla se tutti gli archi stanno nel cammino *)
let controllo_lista lista cammino =
    let rec controllo = function
        []-> true
        | (x,y)::rest -> if List.mem (x,y) cammino ||
                            List.mem (y,x) cammino then controllo rest
                else false
    in controllo lista;;

(**controlla il numero delle occorrenze del nodo v_0 *)
let scan_occorrenze elem lista =
    List.fold_left (fun acc y -> if elem = y then acc + 1 else acc) 0 lista;;
```

Listing 7: Funzioni trasforma_cammino, controllo_lista e scan_occorrenze.

La funzione `trasforma_cammino` trasforma il cammino passato in input come una lista di archi. Quest'ultima è stata implementata data la necessità di controllare se all'interno del cammino trovato sono presenti tutti gli archi del sottoinsieme E'. La suddetta verifica viene effettuata tramite la funzione `controllo_lista` la quale si assicura della presenza di tutti gli archi passati dall'utente. Questo controllo viene fatto tramite la funzione `List.mem` che verifica se l'arco (x,y) o (y,x) del sottoinsieme E' è presente all'interno del cammino trovato. L'ultima funzione permette di calcolare il numero di occorrenze di un determinato elemento passato in input `elem` all'interno di una lista `lista` utilizzando una variabile `acc` come contatore .

```
exception Arco_NotFound;;

(* restituisce gli archi del grafo come una coppia *)
let modifica_archi_grafo archi =
    let rec modifica mod_archi = function
        []-> mod_archi
        | (x,y,z)::rest -> modifica ((x,z) :: mod_archi) rest
    in modifica [] archi;;

(* controlla se tutti gli archi scelti dall'utente sono nel grafo *)
let controllo_archi lista archi =
    let rec controllo = function
        [] -> lista
        |(x,y)::rest ->  if List.mem (x,y) (modifica_archi_grafo archi) ||
                         List.mem (y,x) (modifica_archi_grafo archi)
                            then controllo rest
        else raise Arco_NotFound
in controllo lista;;
```

Listing 8: Funzioni modifica_archi_grafo e controllo_archi.

Le ultime due funzioni ausiliare vengono impiegate per accettarsi che l'utente abbia inserito tutti archi esistenti all'interno del sottoinsieme E'. Nel dettaglio, la funzione `modifica_archi` serve per trasformare la lista di archi del grafo, costituita da triple, in una lista di coppie di nodi al fine di poter verificare con la funzione `controllo_archi` se tutte le coppie di archi inserite dall'utente effettivamente esistono nel grafo originale G. Questo esame viene fatto sempre mediante la funzione `List.mem` sulla lista degli archi modificata, in caso negativo ovvero l'utente ha inserito un arco non esistente, viene sollevata l'eccezione `Arco_NotFound`.

4 Ricerca in ampiezza

4.1 Prima soluzione bfs_circuit

La ricerca in ampiezza è stata implementata utilizzando la funzione di ricerca di un cammino proposta dal Prof.re Marcugini nel slide "Grafi e algoritmi di ricerca", adattandola al problema in questione. Infatti come possiamo vedere in 9 nel caso

ricorsivo abbiamo un'espressione condizionale del tipo `if...then...else` la quale controlla:

1. **controllo_lista lista_archi (trasforma_cammino cammino)**: se è stato trovato un cammino che abbia tutti gli archi di E';
2. **costo_del_cammino cammino grafo** $<$ **k** : se il costo del cammino è minore dell'intero K;
3. **List.hd cammino** $=$ **inizio** : se il nodo finale del cammino è uguale a quello di inizio, ovvero se partendo da un generico nodo v_0 si ritorna in quest'ultimo;
4. **scan_occorrenze inizio cammino** $=$ **2** : se il numero di volte in cui compare il nodo v_0 nel cammino è esattamente due, una all'inizio e alla fine.

```
exception NotFound;;

(**BFS CIRCUITO*)
let bfs_circuit inizio k grafo lista_archi (Graph_suc succ) =
    let estendi cammino = stampalista cammino;
        List.map (function x -> x::cammino)
        (List.filter (function x -> if x = inizio then true
                                    else not (List.mem x cammino))
        (succ (List.hd cammino)))
    in let rec ricerca  = function
            [] -> raise NotFound
            | cammino::rest ->
                (*se il costo del cammino è minore di k e contiene tutti
                gli archi passati allora ritorna il cammino*)
                if ((costo_del_cammino cammino grafo < k) &&
                    (List.hd cammino = inizio) &&
                    (scan_occorrenze inizio cammino = 2) &&
                    (controllo_lista lista_archi (trasforma_cammino cammino)))
                    then cammino
                else ricerca (rest @ (estendi cammino))
in ricerca [[inizio]];;
```

Listing 9: Funzione bfs_circuit.

Inoltre nel nostro caso il cammino da cercare è un circuito perciò sicuramente all'interno del cammino da trovare dovrà essere presente il vertice di partenza v_0 sia all'inizio che alla fine. Ma durante l'estensione del cammino non è possibile ricercare possibili successori tra quelli già inseriti in quest'ultimo, questo avviene grazie al comando

```
(List.filter (function x -> not (List.mem x cammino))
```

Poiché nel nostro caso abbiamo la necessità di usare il nodo iniziale due volte, essendo un circuito, ho modificato questo comando permettendo all'algoritmo di scegliere più volte il nodo v_0. In seguito ho controllato anche il numero di occorrenze del nodo v_0, poiché modificando quest'ultimo comando c'era la possibilità che venisse scelto come cammino uno che passava più volte per il nodo iniziale. Detto ciò gli unici cammini scelti come risultato finale saranno quelli che, oltre a verificare le proprietà richieste, possiedono il nodo iniziale esattamente due volte `(scan_occorrenze inizio cammino = 2)` e categoricamente all'inizio e alla fine `(List.hd cammino = inizio)`.

4.2 Seconda soluzione bfs_circuit

```
exception NotFound;;

(**BFS CIRCUITO*)
let bfs_circuit2 inizio k grafo lista_archi (Graph_suc succ) =
    let estendi 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 ricerca  = function
            [] -> raise NotFound
            | cammino::rest ->
                (*se il costo del cammino è minore di k e contiene tutti
                gli archi passati allora ritorna il cammino*)
                if ((costo_del_cammino ([inizio] @ cammino) grafo < k) &&
                (List.mem inizio (succ (List.hd cammino))) &&
                (controllo_lista lista_archi
                        (trasforma_cammino ([inizio] @ cammino))))
                    then  [inizio] @ cammino
                else ricerca (rest @ (estendi cammino))
in ricerca [[inizio]];;
```

Listing 10: Funzione bfs_circuit2.

Questa seconda soluzione differisce dalla prima poichè al posto di modificare la funzione

```
(List.filter (function x -> not (List.mem x cammino))
```

e controllare le occorrenze del nodo iniziale presenti nel cammino, ho inserito delle espressioni diverse all'interno dell'`if...then...else` le quali controllano:

1. **controllo_lista lista_archi (trasforma_cammino ([inizio] @ cammino))**: se è stato trovato un cammino con il nodo iniziale che abbia tutti gli archi di E';
2. **costo_del_cammino ([inizio] @ cammino) grafo < k** : se il costo del cammino trovato unito al nodo iniziale è minore dell'intero K;

3. **List.mem inizio (succ (List.hd cammino))** : se nei successori del nodo finale del cammino è presente il nodo iniziale allora abbiamo trovato un circuito.

```
(*RICERCA CAMMINO E STAMPA*)
let find_path start k grafo lista_archi (Graph_suc succ)=
    let result = (bfs_circuit start k grafo lista_archi (Graph_suc succ))
    in print_string("Cammino finale:");
    stampalista (List.rev result) ;
    print_string("Cammino finale archi:");
    stampaarchi (trasforma_cammino (List.rev result));
    print_string("Il peso del cammino è:");
    print_int (costo_del_cammino result grafo);
    print_newline();;
```

Listing 11: Funzione find_path.

```
(*RICERCA CAMMINO E STAMPA*)
let find_path2 start k grafo lista_archi (Graph_suc succ)=
    let result = (bfs_circuit2 start k grafo lista_archi (Graph_suc succ))
    in print_string("Cammino finale bfs_circuit2:");
    stampalista (List.rev result) ;
    print_string("Cammino finale bfs_circuit2 archi:");
    stampaarchi (trasforma_cammino (List.rev result));
    print_string("Il peso del cammino con bfs_circuit 2 è:");
    print_int (costo_del_cammino result grafo);
    print_newline();;
```

Listing 12: Funzione find_path2.

Quest'ultima funzione è stata implementata per richiamare la funzione di ricerca in ampiezza `bfs_circuit` e per stampare il risultato trovato sul terminale.

```
let lista_archi1= controllo_archi e1 grafo1.archi;;

find_path 2 50 grafo1.archi lista_archi1 g_s1;;
```

Listing 13: Richiamo funzione controllo_archi e find_path.

Prima di cercare il circuito richiamo la funzione `controllo_archi` in modo da verificare se gli archi passati dall'utente esistono all'interno del grafo.

5 Esempi

In questa Sezione riporto alcuni esempi utilizzati durante il debug del codice.

5.1 Primo grafo

Il primo grafo utilizzato è definito come di seguito:

```
let grafo1 =
    {nodi = [1;2;3;4;5;6];
     archi = [(1,2,5);(1,4,3);(2,3,4);(2,1,3);(3,6,4);(4,5,6);(5,5,3);(5,7,6)];
     lunghezza =[1;2;4;3;5;6;7]};;

let e1 = [(1,5);(5,6);(6,4)];;
let lista_archi1 = controllo_archi e1 grafo1.archi;;
find_path 2 50 grafo1.archi lista_archi1 g_s1;;

find_path2 2 50 grafo1.archi lista_archi1 g_s1;;
```

Listing 14: Grafo 1.

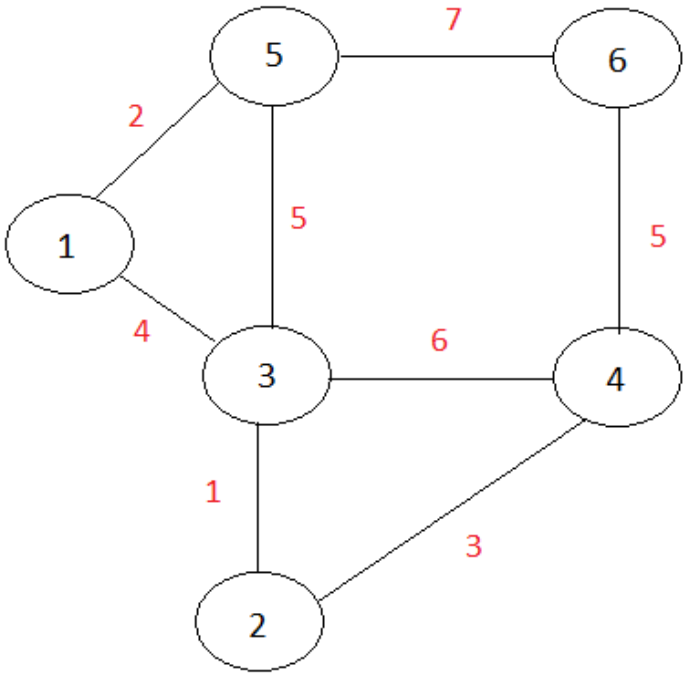

Figura 1: Immagine grafo 1.

5.1.1 Soluzione bfs_circuit1

In questo caso la ricerca in ampiezza riesce a trovare un cammino con gli archi e1 che parte dal nodo 2 con una lunghezza minore di 50, il risultato è:

```
Cammino finale: 2;3;1;5;6;4;2;
Cammino finale archi: (2,3);(3,1);(1,5);(5,6);(6,4);(4,2);
Il peso del cammino è: 22
```

Listing 15: Risultato esempio bfs_circuit grafo 1.

5.1.2 Soluzione bfs_circuit2

In entrambi i casi le soluzioni riportate sono le stesse.

```
Cammino finale bfs_circuit2:2;3;1;5;6;4;2;
Cammino finale bfs_circuit2 archi:(2,3);(3,1);(1,5);(5,6);(6,4);(4,2);
Il peso del cammino con bfs_circuit 2 è:22
```

Listing 16: Risultato esempio bfs_circuit2 grafo 1.

5.2 Secondo grafo

Il secondo grafo viene definito come di seguito:

```
let grafo2 =
    {nodi = [1; 2; 3; 4; 5 ; 6; 7];
     archi = [(1,3,2);(1,9,5);(2,2,3);(5,4,6);(3,1,6);(3,7,4);(4,4,6);(6,1,2)];
     lunghezza =[1;2;3;4;7;9]};;

let e2 = [(1,2);(2,3);(1,5);(3,4)];;
let lista_archi2 = controllo_archi e2 grafo2.archi;;
find_path 3 30 grafo2.archi lista_archi2 g_s2;;
find_path2 3 30 grafo2.archi lista_archi2 g_s2;;
```

Listing 17: Grafo 2.

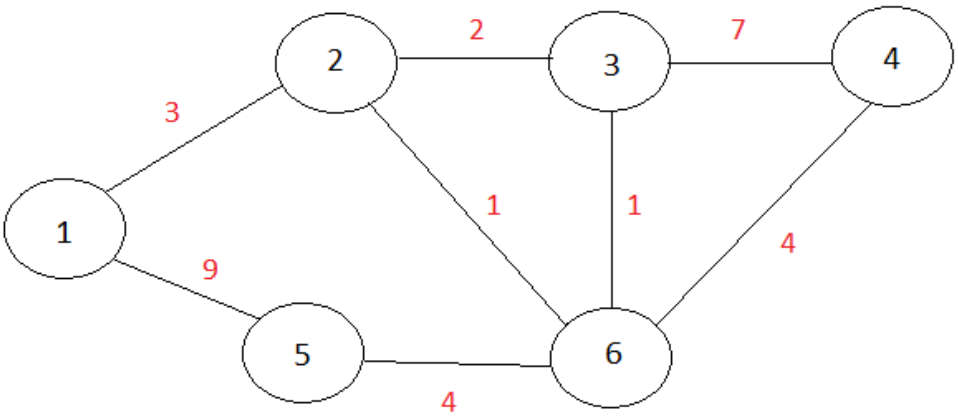

Figura 2: Immagine grafo 2.

5.2.1 Soluzione bfs_circuit

Anche in questo caso la ricerca in ampiezza riesce a trovare un circuito con lunghezza minore di 30 che parte dal nodo 3, il risultato è il seguente:

```
Cammino finale: 3;4;6;5;1;2;3;
Cammino finale archi: (3,4);(4,6);(6,5);(5,1);(1,2);(2,3);
Il peso del cammino è:29
```

Listing 18: Risultato esempio bfs_circuit grafo 2.

5.2.2 Soluzione bfs_circuit2

In entrambi i casi le soluzioni riportate sono le stesse.

```
Cammino finale bfs_circuit2:3;4;6;5;1;2;3;
Cammino finale bfs_circuit2 archi:(3,4);(4,6);(6,5);(5,1);(1,2);(2,3);
Il peso del cammino con bfs_circuit 2 è:29
```

Listing 19: Risultato esempio bfs_circuit2 grafo 2.

5.3 Terzo grafo

Definizione del terzo grafo:

```
let grafo3 =
    {nodi = [1; 2; 3; 4; 5 ; 6; 7; 8; 9];
     archi = [(1,4,2);(1,2,3);(1,5,4);(2,3,5);(2,1,6);(3,7,6);(3,3,7);
     (4,6,9);(5,5,6);(7,8,8);(8,2,9)];
     lunghezza =[1;4;3;5;7;8;6;2]};;

let e3 = [(1,2);(8,9)];;
let lista_archi3= controllo_archi e3 grafo3.archi;;
find_path 5 70 grafo3.archi lista_archi3 g_s3;;
find_path2 5 70 grafo3.archi lista_archi3 g_s3;;
```

Listing 20: Grafo 3.

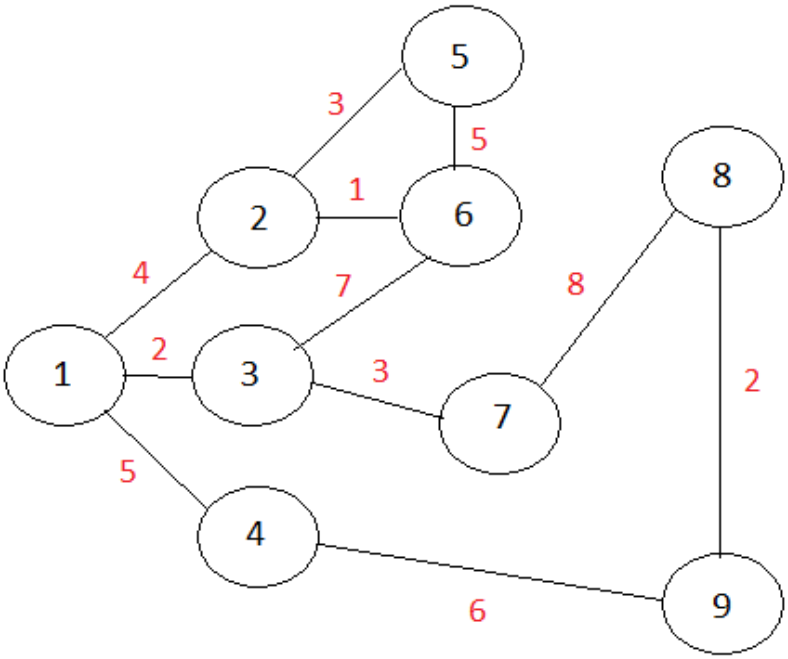

Figura 3: Immagine grafo 3.

5.3.1 Soluzione bfs_circuit

Anche in questo caso la ricerca in ampiezza riesce a trovare un circuito con lunghezza minore di 70 che parte dal nodo 5, il risultato è il seguente:

```
Cammino finale:5;6;3;7;8;9;4;1;2;5;
Cammino finale archi:(5,6);(6,3);(3,7);(7,8);(8,9);(9,4);(4,1);(1,2);(2,5);
Il peso del cammino è:43
```

Listing 21: Risultato esempio bfs_circuit grafo 3.

5.3.2 Soluzione bfs_circuit2

In entrambi i casi le soluzioni riportate sono le stesse.

```
Cammino finale bfs_circuit2:5;6;3;7;8;9;4;1;2;5;
Cammino finale bfs_circuit2 archi:(5,6);(6,3);(3,7);(7,8);(8,9);
                                  (9,4);(4,1);(1,2);(2,5);
Il peso del cammino con bfs_circuit 2 è:43
```

Listing 22: Risultato esempio bfs_circuit2 grafo 3.

5.4 Quarto grafo

Quest'ultimo grafo ha una forma particolare proprio per controllare se effettivamente il codice riportava i risultati aspettati. Esso è definito in questo modo:

```
let grafo4 =
    {nodi = [1; 2; 3; 4; 5 ; 6; 7; 8; 9;10;11];
     archi = [(1,1,2);(2,1,3);(3,1,4);(4,1,5);(5,1,6);(6,1,1);
              (1,1,7);(1,1,11);(7,1,8);(8,1,9);(9,1,10);(10,1,11)];
     lunghezza =[1]};;
```

Listing 23: Grafo 4.

Il grafo in questione ha una forma ad "infinito", poiché volevo verificare se effettivamente il programma non trovava un cammino quando passavo come archi uno appartenente alla "partizione" di destra e uno a quella di sinistra.

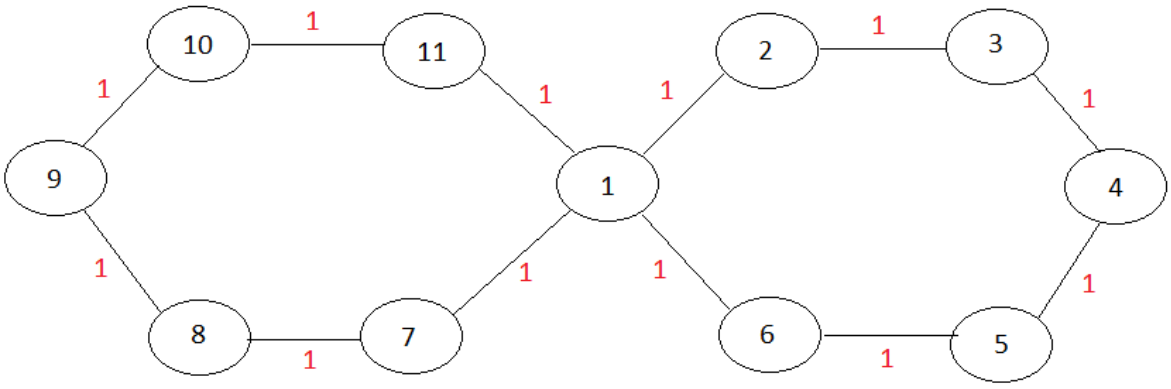

Figura 4: Immagine grafo 4.

In questo caso riporto due esempi per mostrare il funzionamento del programma con diversi sottoinsiemi E':

1. `let e4 = [(2,3)]`: il programma dovrebbe effettivamente trovare un cammino visto gli archi richiesti;
2. `let e4 = [(2,3);(8,9)]`: dovrebbe sollevare l'eccezione `NotFound`.

5.4.1 Risultati esempio 1

Soluzione bfs_circuit Come previsto il programma riesce effettivamente a trovare un cammino, ciò che ci interessa maggiormente è il secondo caso.

```
Cammino finale:1;6;5;4;3;2;1;
Cammino finale archi:(1,6);(6,5);(5,4);(4,3);(3,2);(2,1);
Il peso del cammino è:6
```

Listing 24: Risultato esempio 1 bfs_circuit grafo 4.

Soluzione bfs_circuit2 Il risultato ottenuto con la prima funzione di ricerca è uguale a quello ottenuto dalla seconda.

```
Cammino finale bfs_circuit2:1;6;5;4;3;2;1;
Cammino finale bfs_circuit2 archi:(1,6);(6,5);(5,4);(4,3);(3,2);(2,1);
Il peso del cammino con bfs_circuit 2 è:6
```

Listing 25: Risultato esempio 1 bfs_circuit grafo 4.

5.4.2 Risultati esempio 2

Soluzione bfs_circuit e bfs_circuit2 Le nostre `bfs_circuit` e `bfs_circuit2` funziona riportando il risultato atteso, ovvero l'eccezione `NotFound` essendo che per includere entrambi gli archi avrebbe dovuto passare di nuovo per il nodo iniziale avendo così un numero di occorrenze maggiori di due.

```
(*prima funzione ricerca*)
3;4;5;6;1;7;8;9;10;11;1;2;1;
3;4;5;6;1;11;10;9;8;7;1;2;1;
Fatal error: exception Postino.NotFound

(*seconda funzione ricerca*)
11;10;9;8;7;1;
2;3;4;5;6;1;
6;5;4;3;2;1;
Fatal error: exception Postino.NotFound
```

Listing 26: Risultato esempio 2 grafo 4.