A.D. 1308
unipg
UNIVERSITÀ DEGLI STUDI
DI PERUGIA

Esame IA Development System

Relazione Progetto:

Problema della Cricca
(ricerca in ampiezza)

Studente: Ludovico Guercio
Matricola: 340036

Dipartimento di Matematica e Informatica,
Università degli Studi di Perugia

Indice

1 Obiettivo

l'obbiettivo di questo progetto è realizzare un programma che risolva il problema della cricca(clique) attraverso la ricerca in ampiezza. Una cricca in un grafo non orientato G è un insieme V di vertici tale che, per ogni coppia di vertici in V, esiste un arco che li collega. Equivalentemente, si potrebbe dire che il sottografo indotto da V è un grafo completo. La dimensione di una cricca è il numero di vertici che contiene. Si scriva un programma che, dato un grafo ed un intero N , restituisca, se esiste, una cricca di dimensione N.

2 Algoritmo BFS

L'algoritmo utilizzato è la BFS (breadth-first-search), ossia di ricerca in ampiezza. La BFS permette di trovare un cammino all'interno di un grafo, partendo da un nodo inziale fino ad un nodo obiettivo e connesso al nodo iniziale. L'algoritmo fa parte della famiglia della ricerca non informata.
Si inizia visitando il nodo iniziale, poi i suoi successori, e poi tutti i successori dei successori. I nodi successori da visitare sono gestiti tramite una coda, in questo modo la ricerca è sistematica riuscendo così a visitare tutti i nodi del grafo se l'obbiettivo non è stato trovato nella ricerca parziale.

2.1 Complessità

L'algoritmo è **ottimale e completo**. L'ottimalità è data dal fatto che la prima soluzione che si trova è anche la migliore. L'algoritmo è buono da utilizzare soprattutto quando il costo per ogni arco è uguale; la presenza di una soluzione alternativa alla stessa profondità avrà costo di cammino uguale, ma tempo di ricerca superiore perchè non è la prima soluzione trovabile.

L'espansione dei nodi per livello di profondità garantisce che l'algoritmo sia completo.

La complessità in termini di tempo e spazio coincide:
dato b il fattore di ramificazione, e d la profondità della soluzione, allora il numero massimo di nodi espansi nel peggiore dei casi sarà:

$$O(b)^d$$

3 Struttura del programma

Per cercare una cricca di N dimensioni, visita i nodi nel cammino del grafo e controlla se ognuno di essi possa far parte della cricca creata nel cammino. Dal nodo iniziale, si va a costruire la potenziale cricca di N dimensioni: se non si determina nessuna cricca di quelle dimensioni, il programma esegue la ricerca partendo da un nodo diverso da quello precedente.
I grafi utilizzati sono non orientati e senza pesi, dunque un nodo che ha un arco verso un altro nodo hanno la proprietà di essere successori di entrambi, ossia "vicini".

3.1 Definizione grafo

Per la definizione dei grafi, ho scelto sua rappresentazione mediante lista di archi. Dunque, è stato definito un tipo "graph", composto da elementi di tipo `'a` generico, applicando il costruttore "Grafo" ad una lista di coppie di elementi(nodi) di tipo `'a`.

```
type 'a graph = Grafo of ('a * 'a) list;;
```

Con questa definizione si può definire un grafo che è una lista contenente tutte le coppie di nodi che sono collegate da un arco.

3.2 Funzioni

Il programma è composto da diverse funzioni di utilità; queste serviranno poi per effettuare la ricerca della cricca nei grafi stessi.

Funzione **nodi**:

```
(*'a graph -> 'a list*)
let nodi (Gr arcs) =
  let setadd x set = if List.mem x set then set
                     else x::set
        in let rec aux = function
              [] -> []
              | (x,y)::resto -> setadd x (setadd y (aux resto))
        in aux arcs
;;
```

La funzione **nodi** prende in input un grafo e ritorna la lista dei suoi elementi(nodi) distinti. Utilizza al suo interno le funzioni ausiliari `setadd` e `aux`.

Funzione **successori**: funzione che serve per riportare tutti i nodi successivi di un nodo del grafo, ossia tutti quei nodi per il quale esiste un arco che li collega; prende in input un grafo e un nodo, ritorna una lista contenente nodi successori di quel nodo.

```
(*'a graph -> 'a -> 'a list*)
let rec successori (Gr arcs) n =
  match arcs with
  [] -> []
  | (x,y)::resto -> if x = n then y::successori (Gr resto) n
                    else if y = n then x::successori (Gr resto) n
                    else successori (Gr resto) n
;;
```

L'ordine dei nodi successori dipende dalla definizione di un grafo specifico.

Funzione **vicini**: `vicini` permette di verificare se due nodi all'interno del grafo hanno un arco che li collegano.

```
(*'a graph -> 'a -> 'a -> bool*)
let rec vicini (Gr arcs) n1 n2 =
match arcs with
[] -> false
|(x,y)::resto -> if x == n1 then y == n2 ||vicini (Gr resto) n1 n2
        else if x == n2 then y == n1 ||vicini (Gr resto) n1 n2
        else vicini (Gr resto) n1 n2
;;
```

vicini prende come input un grafo e una coppia di nodi, ritorna `true` se esiste un arco che li collega, cioè che i nodi in input sono coppie di elementi nel grafo, altrimenti `false`. La funzione `successori` e `vicini` sono definite per grafi non orientati, infatti, identificano la presenza di un arco controllando i valori in entrambe le posizioni delle coppia.

Funzione **massimale**: è la funzione obbiettivo che viene utilizzata all'interno della funzione **bfs**. Il suo scopo è controllare ogni volta se una cricca è massimale, ossia che quella cricca rappresenta un sottografo completo non estendibile. Tale cricca non può essere inclusa in una cricca più grande, quindi nessun nodo adiacente può crescere la sua dimensione.

```
(*'a graph -> 'a list -> 'a -> bool*)

let rec massimale grafo cricca m =
  match cricca with
  [] -> false
  | n::resto -> if not(vicini grafo n m) then true
                else massimale grafo resto m
;;
```

La funzione prende un grafo, la lista dei nodi della cricca, e un nodo. Controlla, utilizzando la funzione **vicini**, che il nodo in input in analisi non ha un arco che colleghi quel nodo appartenente alla lista della cricca del grafo, quindi impossibilitando la crescita della dimensione. Se è vero, la cricca non è espandibile, dunque ritorna `true`; `false altrimenti`.

Funzione **stampalista**:

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

stampalista permette di effettuare una stampa più pulita riportando una lista sottoforma di testo (utile per il debug).

3.3 Eccezioni

Nel programma sono definite utilizzata due tipi di eccezioni:

- `exception NessunaCricca`: richiamata quando durante una ricerca vengono esaurite le liste utilizzate per cercare una cricca, quindi non trovando il risultato voluto
- `exception NessunNodoInit`: richiamata in **cerca_dfs** quando la lista dei nodi iniziali è vuota
- `exception DimesioneErrata`: richiamata nella funzione **cerca_cricca** quando si passa un parametro non valido per k

3.4 Funzione BFS

La funzione **bfs** è la funzione centrale del programma, permette di effettuare la ricerca in ampiezza in un grafo e verificare la presenza di una cricca di dimensione fissata. La ricerca tiene anche conto dei cicli in modo da non ricontrollare i nodi già visitare.

```
(*'a graph -> int -> 'a list -> 'a list*)
let bfs (Grafo arcs) k inizio =
  let rec cerca visitati cricca = function
    [] -> raise NessunaCricca
    | n::resto -> if (List.length cricca) = k then cricca
                  else if List.mem n visitati then cerca visitati cricca resto
                  else if massimale (Grafo arcs) cricca n
                    then cerca (n::visitati) cricca (resto @ (successori (Grafo arcs) n))
                  else cerca (n::visitati) (n::cricca) (resto @ (successori (Grafo arcs) n))
  in cerca [] [] [inizio]
;;
```

Partendo da un nodo iniziale, si visita in ampiezza i nodi nel cammino del grafo, ossia i successori dei nodi nel cammino. La ricerca viene effettuata tramite una funzione ausiliare **cerca**, che prende due parametri `visitati, cricca`. Se la lista dei nodi del cammino è vuota ritorna l'eccezione `NessunaCricca`. Altrimenti per ogni elemento(nodo) `cricca` vengono effettuati dei controlli:

- si verifica la terminazione della ricerca, controllando se la lunghezza della lista `cricca` coincide con il valore k(ossia la dimensione N cercata)
- se il nodo è già visitato, allora viene richiamata la funzione ausiliare `cerca` che riprende la ricerca con il resto dei nodi del cammino
- se cricca è massimale, si prosegue sempre con `cerca` aggiungendo però il nodo adiacente in analisi alla lista `visitati`; altrimenti , se la cricca non è massimale, il nodo viene aggiunto alla cricca e ai visitati e si prosegue la ricerca

Il proseguimento della ricerca dei nodi è fatto in ampiezza, quindi aggiungendo i nodi `successori` in fondo alla lista dei nodi del cammino.

A scopo di testing, ho definito un'altra funzione uguale a **bfs**, chiamata **stampabfs**, la quale al suo interno utilizza la funzione **stampalista** per riportare durante l'esecuzione l'aggiornamento della lista `visitati` e `cricca` e dei nodi del cammino.

Funzione **cerca_cricca_da**: utilizzata a scopo di test, dato un grafo, un valore intero k (dimensioni della cricca) e un valore di inizio, applica la funzione **bfs** se k > 2. La funzione, dunque, cerca una cricca nel grafo, eseguendo una visita in ampiezza su di esso.

```
(*'a graph -> int -> 'a -> 'a list*)
let cerca_cricca_da (Gr arcs) k inizio =
  if k < 2 then raise ErroreDimensione
  else bfs (Gr arcs) k inizio
;;
```

Funzione **cerca_bfs**: permette di applicare la funzione **bfs** utilizzando un grafo, un valore k e una lista di nodi iniziali; riporta la lista invertita contenente gli elementi che formano la cricca cercata.

```
(*'a graph -> int -> 'a list -> 'a list*)
let rec cerca_bfs grafo k = function
  [] -> raise NessunNodInit
  | n::resto -> try List.rev (bfs grafo k n)
                with NessunaCricca -> cerca_bfs grafo k resto
;;
```

La lista di nodi iniziali serve per poter cercare una possibile cricca partendo da diversi nodi del grafo.
Se la lista dei nodi iniziali è vuota, non è possibile avviare la ricerca e dunque solleva l'eccezione `NessunNodoInit`. Altrimenti partendo da un nodo, tramite il costrutto `try-with` prova a cercare una cricca utilizzando la funzione `bfs`. Se la cricca non è stata trovata, si effettua nuovamente `cerca_bfs` passandogli il resto della lista

Funzione **cerca_cricca**: è la funzione main del programma; cerca una cricca di dimensioni N all'interno del grafo. Prende come parametri un grafo e un valore k; applica la funzione `cerca_bfs` che prende anche come parametro la lista di tutti i nodi del grafo.

```
let cerca_cricca grafo k =
  if k < 2 then raise ErroreDimensione;;
  else cerca_bfs grafo k (nodi grafo)
;;
```

Se il valore k della dimensione della cricca è minore di 2, allora ritorna l'eccezione **ErroreDimensione**.

4 Esempi esecuzione

In questa ultima sezione sono riporta i risutalti dei test del programma effettuati su alcuni grafi. La funzione **stampalista** utilizzata dentro **stampabfs**, permette di fare da debug e stampa in ordine la lista `visitati` e la lista `cricca`.

Il programma può essere eseguito chiamando la funzione **cerca_cricca**:

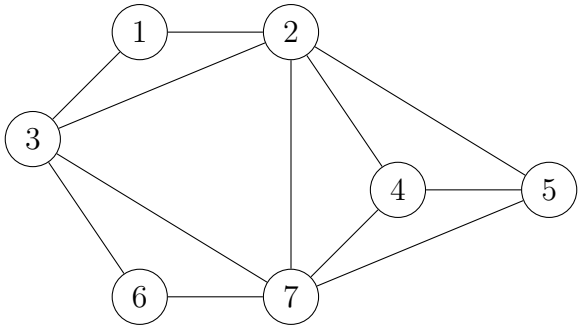
```
cerca_cricca mioGrafo 6;;
```

Prende come argomenti un grafo di tipo `int graph` e un valore k di tipo`int` che rappresenti la dimensione della cricca voluta.

4.1 1° esempio

Il primo esempio su cui è stato testato il programma è un grafo formato da 7 nodi con archi non orientati; il grafo è planare, ossia contiene archi che non si intersecano per collegare i nodi.

```
let grafo1 = Gr [(1,2);(1,3);
                 (2,3);(2,4);(2,5);(2,7);
                 (3,6);(3,7);
                 (4,5);(4,7);
                 (5,7);
                 (6,7)];;
```



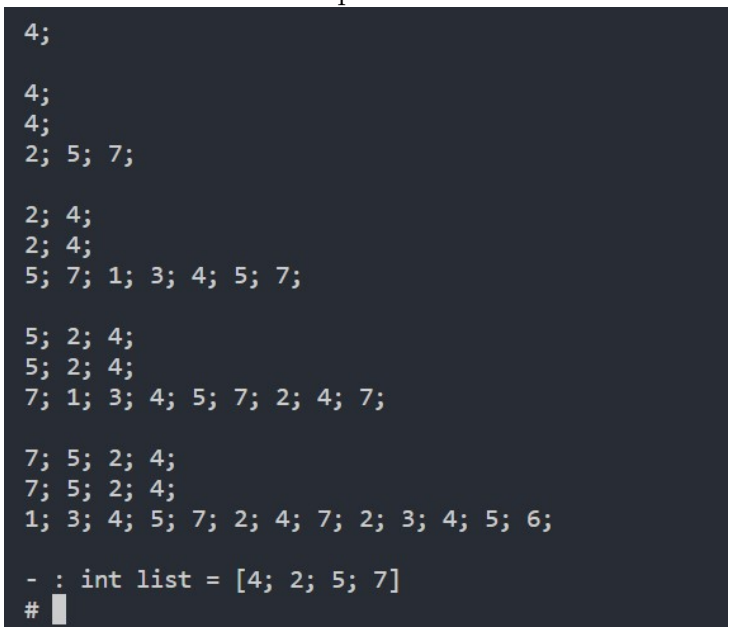
In questo grafo ci sono diverse cricche di dimensione 3, mentre solamente una cricca di dimensione 4, che corrisponde anche alla cricca massima del grafo.

Esempi ricerca di una cricca di dimensione 4:

```
cerca_cricca grafo1 4;;


1;

1;
1;
2; 3;

2; 1;
2; 1;
3; 1; 3; 4; 5; 7;

3; 2; 1;
3; 2; 1;
1; 3; 4; 5; 7; 1; 2; 6; 7;

3; 2; 1;
3; 2; 1;
3; 4; 5; 7; 1; 2; 6; 7;
```

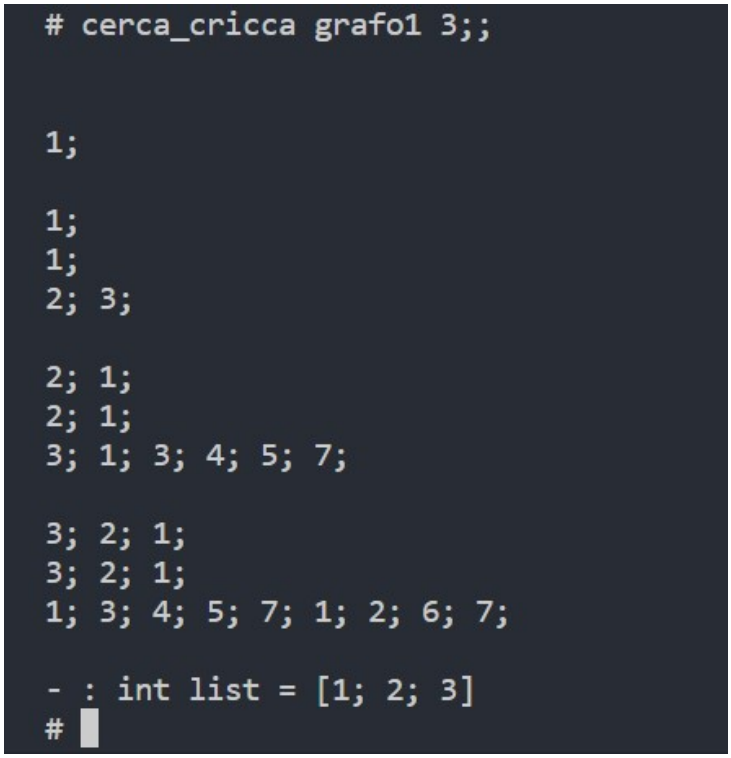
```
2;

2;
2;
1; 3; 4; 5; 7;

1; 2;
1; 2;
3; 4; 5; 7; 2; 3;

3; 1; 2;
3; 1; 2;
4; 5; 7; 2; 3; 1; 2; 6; 7;

4; 3; 1; 2;
3; 1; 2;
5; 7; 2; 3; 1; 2; 6; 7; 2; 5; 7;

5; 4; 3; 1; 2;
3; 1; 2;
7; 2; 3; 1; 2; 6; 7; 2; 5; 7; 2; 4; 7;
```

La ricerca prosegue e si trova solamente il primo risultato iniziando la ricerca dal nodo 4:



Il programma arriva a trovare la cricca di dimensioni 4 soltanto partendo dal nodo 4: questo perchè nella ricerca partendo da nodi come 2 o 3, questi prendono subito come loro successori il nodo 1 perchè il loro primo successore nella definizione del grafo1. Di conseguenza si forma subito una cricca in quel cammino che è di dimensione 3 e non espandibile dal nodo 4.

In modo analogo se si effettua una ricerca della cricca di dimensione 2, oppure di 3, si otterrà subito il risultato:



Con l'utilizzo della funzione **cerca_cricca_da** si può verificare che **bfs** può trovare diverse cricche partendo da altri nodi:

```
# cerca_cricca_da grafo1 3 6;;


6;

6;
6;
3; 7;

3; 6;
3; 6;
7; 1; 2; 6; 7;

7; 3; 6;
7; 3; 6;
1; 2; 6; 7; 2; 3; 4; 5; 6;

- : int list = [7; 3; 6]
#
```

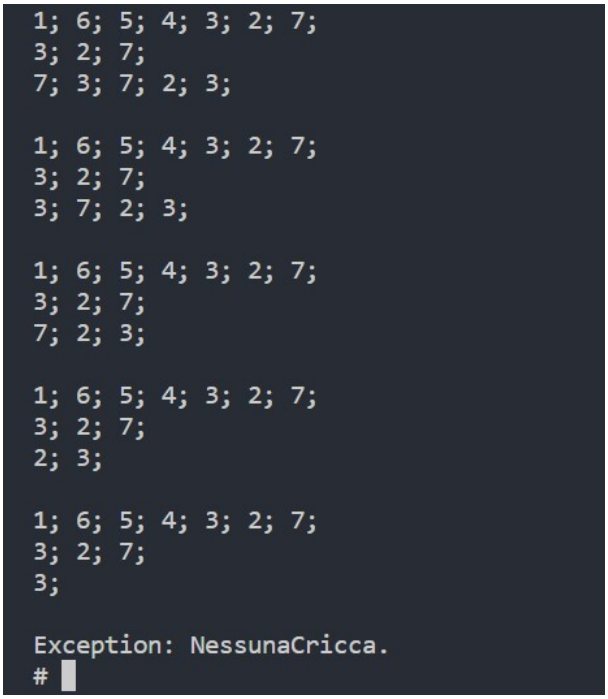
Esempio in cui la ricerca fallisce non trovando nessuna cricca:

```
# cerca_cricca grafo1 5;;


1;

1;
1;
2; 3;

2; 1;
2; 1;
3; 1; 3; 4; 5; 7;

3; 2; 1;
3; 2; 1;
1; 3; 4; 5; 7; 1; 2; 6; 7;

3; 2; 1;
3; 2; 1;
3; 4; 5; 7; 1; 2; 6; 7;

3; 2; 1;
3; 2; 1;
4; 5; 7; 1; 2; 6; 7;

4; 3; 2; 1;
3; 2; 1;
5; 7; 1; 2; 6; 7; 2; 5; 7;
```

```
4;

4;
4;
2; 5; 7;

2; 4;
2; 4;
5; 7; 1; 3; 4; 5; 7;

5; 2; 4;
5; 2; 4;
7; 1; 3; 4; 5; 7; 2; 4; 7;

7; 5; 2; 4;
7; 5; 2; 4;
1; 3; 4; 5; 7; 2; 4; 7; 2; 3; 4; 5; 6;

1; 7; 5; 2; 4;
7; 5; 2; 4;
3; 4; 5; 7; 2; 4; 7; 2; 3; 4; 5; 6; 2; 3;

3; 1; 7; 5; 2; 4;
7; 5; 2; 4;
4; 5; 7; 2; 4; 7; 2; 3; 4; 5; 6; 2; 3; 1; 2; 6; 7;
```

```
5;
5;
2; 4; 7;

2; 5;
2; 5;
4; 7; 1; 3; 4; 5; 7;

4; 2; 5;
4; 2; 5;
7; 1; 3; 4; 5; 7; 2; 5; 7;

7; 4; 2; 5;
7; 4; 2; 5;
1; 3; 4; 5; 7; 2; 5; 7; 2; 3; 4; 5; 6;

1; 7; 4; 2; 5;
7; 4; 2; 5;
3; 4; 5; 7; 2; 5; 7; 2; 3; 4; 5; 6; 2; 3;
```

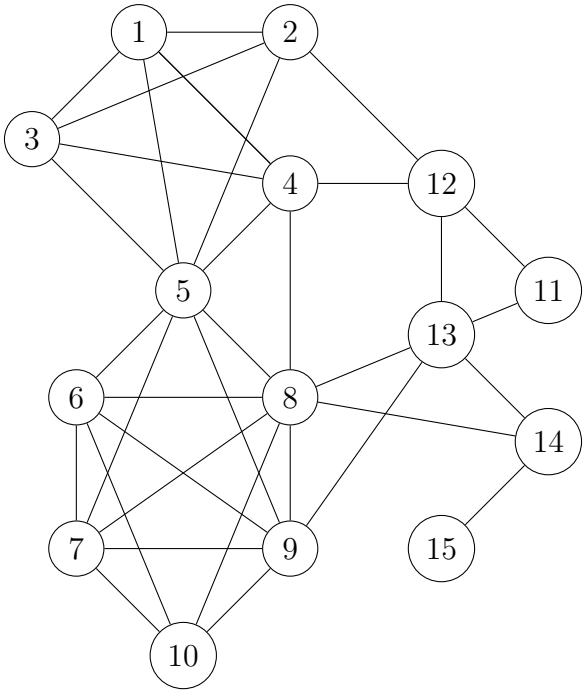
```
7;

7;
7;
2; 3; 4; 5; 6;

2; 7;
2; 7;
3; 4; 5; 6; 1; 3; 4; 5; 7;

3; 2; 7;
3; 2; 7;
4; 5; 6; 1; 3; 4; 5; 7; 1; 2; 6; 7;

4; 3; 2; 7;
3; 2; 7;
5; 6; 1; 3; 4; 5; 7; 1; 2; 6; 7; 2; 5; 7;
```



4.2 2° esempio

Nel secondo esempio si è applicato il programma ad un grafo più grande, di 15 nodi, contenente maggiori cricche di più grande dimensione.

```
let grafo2 = Grafo [(1,2);(1,3);(1,4);(1,5);
                    (2,3);(2,5);(2,12);
                    (3,4);(3,5);
                    (4,5);(4,12);
                    (5,6);(5,7);(5,8);(5,9);(5,10);
                    (6,7);(6,8);(6,9);(6,10);
                    (7,8);(7,9);(7,10);
                    (8,4);(8,9);(8,10);(8,13);
                    (9,10);(9,13);
                    (11,12);(11,13);
                    (12,13);
                    (13,14);
                    (14,8);(14,15)];;
```



La ricerca di una cricca di N-dimensioni in questo grafo viene fatta con successo data anche la sua maggior dimensione.

Ad esempio, la ricerca di una cricca di dimensione 6 risulta più lunga perché si visitano molti più nodi e di conseguenza anche la lista dei cammini aumenta di dimensioni:

```
3;

3;
3;
1; 2; 4; 5;

1; 3;
1; 3;
2; 4; 5; 2; 3; 4; 5;

2; 1; 3;
2; 1; 3;
4; 5; 2; 3; 4; 5; 1; 3; 5; 12;

4; 2; 1; 3;
2; 1; 3;
5; 2; 3; 4; 5; 1; 3; 5; 12; 1; 3; 5; 12; 8;
```

```
7; 6; 8; 12; 5; 4; 2; 1; 3;
5; 2; 1; 3;
9; 10; 2; 4; 11; 13; 5; 6; 7; 4; 9; 10; 13; 14; 5; 7; 8; 9; 10; 5; 6; 8; 9; 10;

9; 7; 6; 8; 12; 5; 4; 2; 1; 3;
5; 2; 1; 3;
10; 2; 4; 11; 13; 5; 6; 7; 4; 9; 10; 13; 14; 5; 7; 8; 9; 10; 5; 6; 8; 9; 10; 5; 6; 7; 8; 10; 13;

10; 9; 7; 6; 8; 12; 5; 4; 2; 1; 3;
5; 2; 1; 3;
2; 4; 11; 13; 5; 6; 7; 4; 9; 10; 13; 14; 5; 7; 8; 9; 10; 5; 6; 8; 9; 10; 5; 6; 7; 8; 10; 13; 5; 6; 7; 8; 9;

10; 9; 7; 6; 8; 12; 5; 4; 2; 1; 3;
5; 2; 1; 3;
4; 11; 13; 5; 6; 7; 4; 9; 10; 13; 14; 5; 7; 8; 9; 10; 5; 6; 8; 9; 10; 5; 6; 7; 8; 10; 13; 5; 6; 7; 8; 9;

10; 9; 7; 6; 8; 12; 5; 4; 2; 1; 3;
5; 2; 1; 3;
11; 13; 5; 6; 7; 4; 9; 10; 13; 14; 5; 7; 8; 9; 10; 5; 6; 8; 9; 10; 5; 6; 7; 8; 10; 13; 5; 6; 7; 8; 9;

11; 10; 9; 7; 6; 8; 12; 5; 4; 2; 1; 3;
5; 2; 1; 3;
13; 5; 6; 7; 4; 9; 10; 13; 14; 5; 7; 8; 9; 10; 5; 6; 8; 9; 10; 5; 6; 7; 8; 10; 13; 5; 6; 7; 8; 9; 12; 13;
```

```
6;
6;
5; 7; 8; 9; 10;

5; 6;
5; 6;
7; 8; 9; 10; 1; 2; 3; 4; 6; 7; 8; 9; 10;

7; 5; 6;
7; 5; 6;
8; 9; 10; 1; 2; 3; 4; 6; 7; 8; 9; 10; 5; 6; 8; 9; 10;

8; 7; 5; 6;
8; 7; 5; 6;
9; 10; 1; 2; 3; 4; 6; 7; 8; 9; 10; 5; 6; 8; 9; 10; 5; 6; 7; 4; 9; 10; 13; 14;

9; 8; 7; 5; 6;
9; 8; 7; 5; 6;
10; 1; 2; 3; 4; 6; 7; 8; 9; 10; 5; 6; 8; 9; 10; 5; 6; 7; 4; 9; 10; 13; 14; 5; 6; 7; 8; 10; 13;

10; 9; 8; 7; 5; 6;
10; 9; 8; 7; 5; 6;
1; 2; 3; 4; 6; 7; 8; 9; 10; 5; 6; 8; 9; 10; 5; 6; 7; 4; 9; 10; 13; 14; 5; 6; 7; 8; 10; 13; 5; 6; 7; 8; 9;

- : int list = [6; 5; 7; 8; 9; 10]
#
```

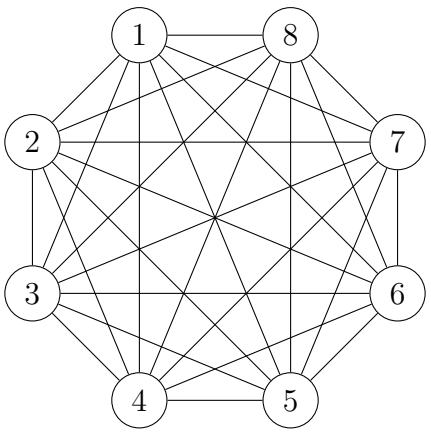
La cricca di dimensione 6 viene trovata partendo dal nodo "6"

Si potrebbero evidenziare problemi con grafi ancora molto più grandi dove la prima soluzione non è immediata nei nodi del cammino. In questi casi l'algoritmo BFS non è prestazionale in quanto visiterebbe ogni nodo per ogni profondità.

4.3 3° esempio

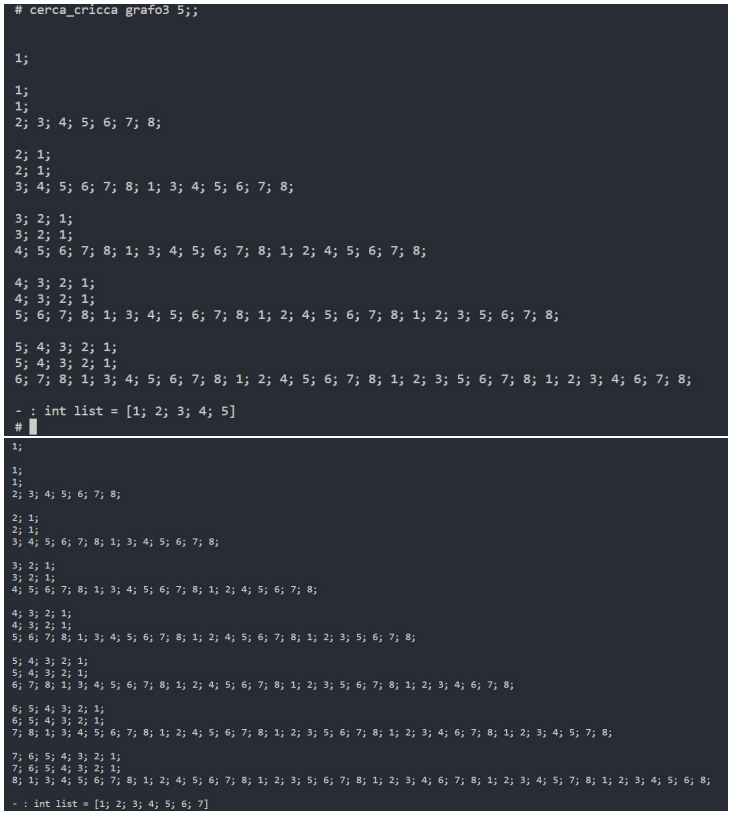
Il terzo grafo su cui è stato eseguito il programma, è un grafo completo composto da 8 nodi. In un grafo completo tutti i nodi hanno un arco che li collega con tutti gli altri nodi distinti nel grafo.

```
let grafo2 = Gr [(1,2);(1,3);(1,4);(1,5);(1,6);(1,7);(1,8);
                 (2,3);(2,4);(2,5);(2,6);(2,7);(2,8);
                 (3,4);(3,5);(3,6);(3,7);(3,8);
                 (4,5);(4,6);(4,7);(4,8);
                 (5,6);(5,7);(5,8);
                 (6,7);(6,8);
                 (7,8)];;
```



La cricca di dimensione massima è uguale al numero di nodi totali del grafo (nell'esempio di dimensione 8).

La ricerca di una qualsiasi cricca di dimensione massima di 8 è immediata a partire dal qualsiasi nodo del grafo.



4.4 Conclusioni

Il programma definito per ricercare una cricca di dimensioni k può trovare le soluzioni quando:

- `k` <= del numero totale dei nodi di un grafo G
- `k` > 2, ossia la dimensione minimi da una cricca
- `I` sottografi di G sono grafi completi

Il programma trova la soluzione nell'immediato quando è possibile determinare che i nodi sono vicini direttamente nei primi successori. La soluzione, se esiste, verrà trovata in un tempo maggiore quando il programma dovrà visitare un maggior numero di nodi del grafo, o addirittura tutti.
Difatto, controllare che un sottografo può formare una cricca di N vertici, ossia sia un sottografo completo, impiegherà un tempo di costo **polinomiale** data la conoscenza della dimensione N.

Il programma, invece, non troverà soluzioni nel caso:

- Non ci siano sottografi completi
- La dimensione di k < 2