**TRACCIA**

Problema della cricca (ricerca 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. Si risolva il problema utilizzando un algoritmo di ricerca in ampiezza.

**INTRODUZIONE**

Un grafo è una coppia ordinata G = (V, E) , dove V è l'insieme dei nodi ed E l'insieme degli archi, tali che gli elementi di E siano coppie di elementi di V . Se le coppie di nodi sono ordinate, il grafo é detto orientato, se non sono ordinate é detto non orientato.

Due vertici u, v connessi da un arco e prendono nome di "'estremi dell'arco'"; l'arco e viene anche identificato con la coppia formata dai suoi estremi (u, v).

In [teoria dei grafi](https://it.wikipedia.org/wiki/Teoria_dei_grafi), una cricca è un insieme V di vertici in un [grafo](https://it.wikipedia.org/wiki/Grafo) non [orientato](https://it.wikipedia.org/wiki/Grafo_orientato) G, tale che, per ogni coppia di vertici in V, esiste un arco che li collega. La dimensione di una cricca è definita come il numero di [vertici](https://it.wikipedia.org/wiki/Vertice_(teoria_dei_grafi)) che contiene.

Il problema di trovare, se esiste, una cricca di una dimensione fissata all'interno di un grafo è detto [problema della cricca](https://it.wikipedia.org/wiki/Problema_della_cricca), ed è un problema [NP-completo](https://it.wikipedia.org/wiki/NP-completo).

L'idea intuitiva per la risoluzione del problema è quella di visitare in ampiezza il grafo dato, a partire da ogni nodo. Ad ogni passo ricorsivo si ha un nodo corrente e una cricca temporanea e si deve controllare se tale nodo può espandere tale cricca entrando a far parte di essa.

Nel linguaggio OBJECTIVE CAML un grafo è rappresentabile tramite le liste, in particolare si può pensare di definire un grafo come la lista degli archi presenti nel grafo oppure con la lista dei successori di ogni nodo del grafo.

Per la risoluzione del problema della cricca si è pensato di utilizzare la seconda tipologia di costrutto, quindi ogni elemento della lista è costituito da una coppia dove il primo elemento è un nodo e il secondo elemento della coppia è la lista dei nodi raggiungibili da quello specifico nodo, ovvero ogni elemento è nella forma (x, [y,z,..]) con x,y,z,.. nodi.

**let** grafo **=** **[(**1**,[**2**;**3**;**5**]);** **(**5**,[**2**;**3**;**4**;**1**;**7**])** **;(**7**,[**5**;**2**;**3**;**4**]);** **(**2**,[**4**;**1**;**5**;**3**;**7**]);** **(**3**,[**2**;**4**;**1**;**5**;**7**]);** **(**4**,[**2**;**3**;**5**;**7**])** **];;**

(MA perchè non abbiamo usato il costrutto Gr????)

Passiamo alla spiegazione delle varie funzioni implementate per questo problema.

La funzione "cricca" rappresenta, insieme alla sua funzione ausiliaria, la procedura principale dell'algoritmo:

**- funzione** cricca

**let** cricca graph k**=**

**let** **rec** aux coda graph **=**

**match** coda **with**

[]**->**[]

**|** **(**x**,**cricca**)::**rest**->** **if((**contiene **(**succNodo graph x**)** cricca**)** **&&** **((**List.length **(**x**::**cricca**))==**k**))** **then** cricca**@[**x**]**

**else** **if** **((**contiene **(**succNodo graph x**)** cricca**)** **&&** **((**List.length **(**x**::**cricca**))<**k**))** **then** **(**aux **(**rest**@(** succNodoConCricca **(**succNodo graph x**)** **(**cricca**@[**x**])))** graph **)**

**else** **(**aux rest graph**)**

**in** aux **(**grafoAdjToGrafoAdjCricca graph**)** graph**;;**

La funzione ricorsiva ausiliaria "aux" si aspetta come input un grafo (il grafo iniziale rappresentato tramite lista di adiacenze) e una lista di coppie sul quale viene fatto pattern matching. Il primo elemento della coppia rappresenta un nodo e il secondo una lista rappresentante la *cricca temporanea*.

Questi due input saranno costituiti al primo passo ricorsivo da:

-il grafo iniziale

- una lista formata da una coppia per ogni nodo del grafo di origine, i cui elementi sono per l'appunto il nodo e una *cricca temporanea* vuota.

Ad ogni passo ricorsivo si considera la prima coppia della coda e si controlla se ogni nodo della cricca temporanea appartiene alla lista delle adiacenze del nodo corrente.

In caso negativo si va a esaminare ricorsivamente la prossima coppia della coda

Altrimenti il nodo corrente può entrare a far parte della cricca temporanea poiché sappiamo con certezza che esiste un arco che lo collega ad ogni altro nodo di tale cricca.

Ora che la cricca temporanea è stata espansa, se la sua dimensione ha raggiunto il valore k richiesto, l'algoritmo termina restituendo in output la cricca trovata.

Altrimenti si vanno a formare tutte le coppie composte da ogni nodo adiacente al nodo appena computato e la nuova cricca temporanea ottenuta, vengono accodate alla lista "coda" e si continua a scorrere tale coda ricorsivamente.

- funzione listaNodiGrafo:

**let** **rec** listaNodiGrafo grafo**=**

**match** grafo **with**

[]**->**[]

**|** **(**x**,**adj**)::**rest**->** x**::(**listaNodiGrafo rest**);;**

La funzione listaNodiGrafo è una funzione ricorsiva definita mediante pattern matching, la quale prende in input un grafo e restituisce la lista di tutti i suoi nodi. Il caso base è dato dalla lista vuota e restituisce la lista vuota. Altrimenti, se il matching ha successo, il primo elemento delle coppie del grafo viene messo in testa ad una lista che si crea tramite le chiamate ricorsive sulla restante parte del grafo .

**let** **rec** grafoAdjToGrafoAdjCricca grafo**=**

**match** grafo **with**

[]**->**[]

**|** **(**x**,**adj**)::**rest**->** **(**x**,**[]**)::(**grafoAdjToGrafoAdjCricca rest**);;**

La funzione grafoAdjToGrafoAdjCricca è una funzione anch'essa ricorsiva che, a partire dalla lista rappresentante il grafo delle adiacenze, restituisce una nuova lista costituita da coppie, dove il primo elemento rappresenta un nodo del grafo iniziale e il secondo elemento della coppia è una lista vuota. Tale lista servirà poi nella funzione principale per costruire la k-cricca, nel caso esistesse.

**- funzione contiene**

**let** **rec** contiene adjNodo cricca**=**

**match** cricca **with**

[]**->**true

**|** x**::**rest**->(**List.mem x adjNodo **&&** **(**contiene adjNodo rest**));;**

tale funzione è una funzione booleana che controlla se tutti gli elementi di una lista (chiamata " adjNodo") sono contenuti anche nell'altra lista passata come parametro.

In questa funzione viene utilizzata la funzione "mem" della classe List, nativa di tale linguaggio di programmazione, che controlla se un elemento appartiene ad una lista.

- funzione succNodo

**let** **rec** succNodo graph n**=**

**match** graph **with**

[]**->**[]

**|** **(**x**,**s**)::**rest**->**

**if(**x**==**n**)**

**then** s

**else** **(**succNodo rest n**);;**

La funzione "succNodo" restituisce i successori del nodo con valore "n" in un grafo rappresentato con la lista delle adiacenze.

- funzione succNodoConCricca

**let** **rec** succNodoConCricca succ cricca**=**

**match** succ **with**

[]**->**[]

**|** x**::**rest**->** **(**x**,**cricca**)::(**succNodoConCricca rest cricca**);;**

Questa funzione prende in input due liste e crea una nuova lista di coppie di nodi e liste.