UNIVERSITÀ DEGLI STUDI DI PERUGIA

**Dipartimento di Matematica e Informatica**

**Relazione**

**Problema della cricca**

**Linguaggio: OCaml**

**Lorenzo Franco Ranucci**

**Sommario**

Questa relazione spiega una delle possibili soluzioni per il problema della cricca di dimensione *k*, tramite lo sviluppo di un algoritmo scritto in linguaggio OCaml.

*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

*Objective Caml*, conosciuto anche con il nome di *OCaml* o *O'Caml*, è un avanzato linguaggio di programmazione funzionale (e non solo…) appartenente alla famiglia di linguaggi di programmazione ML. Questo linguaggio è stato utilizzato per la risoluzione del problema della cricca.

Il *problema della cricca* si riferisce a qualsiasi problema della teoria dei grafi che riguarda un un sottografo connesso di un grafo dato. Assume un ruolo importante in informatica, basti pensare a un social network come un grafo i cui nodi rappresentino gli utentiegli archi la loro reciproca conoscenza. Trovare il più grande gruppo di persone che si conoscano l'una con l'altra si riduce a risolvere il problema della cricca massima.

L'algoritmo

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. Non appena una cricca temporanea raggiunge la dimensione *k* ricercata, tale cricca viene restituita come risultato dell'algoritmo. Se nessuna cricca raggiunge la dimensione voluta, si restituisce una lista vuota.

Un grafo viene generalmente rappresentato tramite due diverse strutture dati entrambe riguardanti 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. Tali rappresentazioni oltre che essere equivalenti, possono essere facilmente convertite l'una nell'altra.

Per la risoluzione del problema della cricca si è scelto 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.

Es.:

**let** graph **=** **[(**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**])** **];;**

Passiamo ora alla spiegazione delle varie funzioni implementate per questo problema.

La funzione *clique* rappresenta, insieme alla sua funzione ausiliaria, la procedura principale dell'algoritmo:

**let** clique graph k**=**

**let** **rec** aux queue graph **=**

**match** queue **with**

[]**->**[]

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

**if((**contains **(**succNode graph x**)** clique**)** **&&** **((**List.length **(**x**::**clique**))==**k**))** **then** clique**@[**x**]**

**else** **if** **((**contains **(**succNode graph x**)** clique**)**

**&&** **((**List.length **(**x**::**clique**))<**k**))**

**then** **(**aux **(**rest**@(** succNodeClique **(**succNode graph x**)** **(**clique**@[**x**])))** graph**)**

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

**in** aux **(**queueForCliqueFromAdjGraph graph k**)** 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, esse vengono accodate a *queue* e si continua a scorrere tale coda ricorsivamente.Vediamo adesso le varie funzioni secondarie utilizzate dalla funzione di cui sopra.

**let** **rec** queueForCliqueFromAdjGraph graph k**=**

**match** graph **with**

[]**->**[]

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

**if(**List.length adj **>=** k**)**

**then(**x**,**[]**)::(**queueForCliqueFromAdjGraph rest k**)**

**else** queueForCliqueFromAdjGraph rest k**;;**

La funzione *queueForCliqueFromAdjGraph* è una funzione utilizzata per inizializzare la coda necessaria alla funzione principale, a partire da un grafo rappresentato con lista di adiacenze.

La funzione scorre ricorsivamente il grafo ricevuto in input e restituisce una lista di coppie in cui il primo elemento rappresenta un nodo e il secondo una lista vuota che verrà utilizzata come *cricca temporanea*. Tutti i nodi che hanno una lista di adiacenti minore di *k* vengono scartati poiché sicuramente non fanno parte di una cricca di tale dimensione.

**let rec** succNode graph n=  
 **match** graph **with**  
 []->[]  
 **|** (x,s)::rest->  
 **if**(x==n) **then** s  
 **else** (succNode rest n);;

La funzione *succNode* restituisce i successori del nodo *n* in un grafo rappresentato con la lista delle adiacenze.

**let rec** succNodeClique succ clique=  
 **match** succ **with**  
 []->[]   
 **|** x::rest-> (x,clique)::(succNodeClique rest clique);;

*succNodeClique* viene utilizzata per aggiungere alla coda della funzione principale tutti i successori del nodo che è stato appena inserito in una *cricca temporanea*. La funzione riceve in input i successori di tale nodo e accoppia ciascuno alla *cricca temporanea*, anch'essa passata in input. Tutte le coppie vengono aggiunte ad una lista che viene restituita in output.

**let rec** contains adjNode clique=  
 **match** clique **with**  
 []->true  
  **|** x::rest->(List.mem x adjNode && (contains adjNode rest));;

La funzione *contains* viene sfruttata per controllare se tutti i nodi di una *cricca temporanea* appartengono alla lista delle adiacenze di un nodo. Questo ci permette di capire se il nodo può essere aggiunto a tale cricca.