Problema dei K cammini

Università degli Studi di Napoli Federico II

Dipartimento di ingegneria elettrica e delle tecnologie dell'informazione, classe delle lauree in scienze e tecnologie informatiche

Autore: Michele Maione

Certificato da: Dr. Vadim Malvone

Approvato da: Prof. Dr. Aniello Murano

Presentato: 15/07/2015

Ultima modifica: 01/10/2015

Tavola dei contenuti

Lista delle figure

Lista delle tavole

Appendici

Lista delle abbreviazioni e dei simboli

ABR	Albero Binario di Ricerca
ASD	Algoritmi e strutture dati
BFS	Breadth-First Search (ricerca in ampiezza)
CFC	Componente Fortemente Connessa
DFS	Depth-First Search (Ricerca in profondità)
LASD	Laboratorio di algoritmi e strutture dati

1 Abstract

È necessario fare una piccola introduzione dei grafi. I grafi sono strutture matematiche discrete. Sono utilizzati in aree come topologia, teoria degli automi, funzioni speciali, geometria dei poliedri. Essi inoltre sono alla base di modelli di sistemi e processi studiati nell'ingegneria, nella chimica, nella biologia molecolare, nella ricerca operativa, nella organizzazione aziendale, nella geografia (sistemi fluviali, reti stradali, trasporti), nella linguistica strutturale, nella storia (alberi genealogici, filologia dei testi).[1]

Stiamo progettando degli algoritmi ed implementando delle soluzioni software per risolvere i seguenti problemi sui grafi orientati:

1) Creare un grafo orientato casuale; riportare il tempo di esecuzione.
2) Verificare se in un grafo orientato esistono k cammini tra un nodo sorgente e un insieme di nodi destinazione; riportare il tempo di esecuzione.
3) Rappresentare graficamente il grafo.

2 Introduzione

Nel linguaggio scientifico, un grafo è una struttura relazionale formata da un insieme finito di oggetti detti nodi, e da un insieme di relazioni tra coppie di nodi detti archi. Per indicare un grafo viene utilizzata una notazione del tipo: G(V, E), dove V indica l'insieme dei nodi e E l'insieme degli archi. Se vengono indicati con i e j due nodi del grafo, un arco che congiunge i a j viene spesso indicato con la notazione (i, j).[2]

2.1 Reali impieghi dei grafi

Pragmaticamente possiamo utilizzare i grafi per rappresentare le seguenti realtà:

- una rete telefonica, in cui i nodi rappresentano gli utenti e i punti di smistamento, e gli archi i cavi di collegamento tra i nodi
- un circuito elettrico in cui i nodi rappresentano i punti di giunzione e gli archi i collegamenti elettrici
- una rete stradale in cui i nodi rappresentano gli incroci e gli archi le strade
- un impianto idraulico in cui i nodi rappresentano le giunzioni e gli archi i tubi
- una rete di elaboratori in cui i nodi rappresentano gli elaboratori connessi in rete e gli archi le linee di comunicazione tra di essi
- un programma di calcolo in cui i nodi rappresentano le istruzioni ed esiste un arco tra due nodi se le relative istruzioni possono essere eseguite nel programma in successione
- una struttura dati in cui i nodi rappresentano i dati e gli archi i legami tra i dati realizzati tramite puntatori[3]

2.2 Origine storica

L'origine storica della teoria dei grafi è in genere fatta risalire a una memoria di Eulero del 1736, nella quale veniva formulato il famoso problema dei sette ponti di Königsberg: attraverso Königsberg scorre il fiume Pregel e, in mezzo al fiume, vi sono due isole (che indichiamo con A e B e le due rive del fiume con C e D), collegate fra loro da un ponte; inoltre quattro ponti (due per parte) collegano l'isola A alle due rive e due ponti (uno per parte) collegano l'isola B alle due rive; il problema consiste nel decidere se sia possibile, partendo da un qualsiasi punto, camminare attraverso ogni ponte esattamente una volta, ritornando al punto di partenza. Eulero, per dimostrare che il problema non ammette soluzione, rimpiazzò ogni riva del fiume e ogni isola con un nodo di un

grafo, e ogni ponte con un arco fra i nodi corrispondenti alle due entità collegate dal ponte. In questo modo, dimostrare che il problema non ammette soluzione equivale a dimostrare che il grafo non ammette un ciclo che passa esattamente una volta su ogni arco. Il teorema di Eulero afferma che condizione necessaria e sufficiente affinché un grafo sia percorribile nel modo richiesto è che esista un percorso fra ogni coppia di nodi e tutti i nodi siano toccati da un numero pari di archi. Si può vedere nel caso in esame che la prima condizione è verificata, ma i quattro nodi sono tutti toccati da un numero dispari di archi. Il teorema fornisce un criterio generale per decidere se un qualsiasi grafo è percorribile nel modo richiesto.[4]

3 Analisi

In questa fase analizzeremo separatamente i 3 problemi e stenderemo le fondamenta, gli algoritmi, che codificheremo per ottenere delle funzionalità del programma.

3.1 Grafo casuale

In teoria dei grafi un grafo casuale è un grafo generato da un procedimento aleatorio. Dato un grafo con N nodi, il numero di archi possibili equivale a N^2.

La costruzione di un grafo casuale può essere fatta semplicemente identificando N nodi e collegandoli tra di loro con una probabilità $p \rightarrow [0,1]$. Quindi il numero medio di archi sarà dato da $n = pN^2$.

Paul Erdős e Alfréd Rényi studiarono diverse proprietà dei grafi notando come queste non emergano gradualmente ma non appena venga superata un certa soglia di p. Studiarono quindi la p critica per cui il grafo acquisisce una proprietà q; una proprietà importante fu la topologia. Loro scoprirono che se il numero di archi inserito era basso il grafo era costruito da diverse componenti non connesse tra loro e connettenti $O(\log N)$ nodi. Ma per $n = N/2$ i cluster iniziano a connettersi e si ottiene un cluster gigante.[5]

3.2 K cammini

Verificare se esistono k percorsi differenti tra un vertice u e un gruppo di vertici (v_1, v_2, v_3, ... v_j), può essere eseguito creando un vertice w che riceva i vertici (v_1, v_2, v_3, ... v_j), ed eseguire l'algoritmo per trovare se esistono k percorsi differenti tra u e w.

Trovare k percorsi differenti è un estensione del problema dell'esistenza di un cammino. Un cammino dal nodo a al nodo z di V è un cammino p = (a, v_1, v_2, v_3, ... v_j, z). Quindi tra u e w possono esserci $\sum_{k=1}^{n-2} \binom{n-2}{k} k! = \Omega(2^n)$ cammini.

3.2.1 Algoritmi notevoli di un problema simile: i k cammini minimi

Un problema simile a quello dei k cammini è quello dei k cammini minimi. Questo problema è stato ampiamente studiato negli anni, e sono stati formulati molti algoritmi per risolvere il problema. Anche se il problema dei k cammini minimi è un estensione del nostro è importante conoscere ciò che è stato fatto; di seguito una lista dei più importanti algoritmi per la risoluzione dei k cammini minimi:[6]

Anno	*Autore*	*Titolo*
1971	Jin Y. Yen	Finding the K Shortest Loopless Paths in a Network
1972	M. T. Ardon et al.	A recursive algorithm for generating circuits and related subgraphs
1973	P. M. Camerini et al.	The k shortest spanning trees of a graph
1975	K. Aihara	An approach to enumerating elementary paths and cutsets by Gaussian elimination method
1976	T. D. Am et al.	An algorithm for generating all the paths between two vertices in a digraph and its application
1988	Ravindra K. Ahuja et al.	Faster algorithms for the shortest path problem
1989	S. Anily et al.	Ranking the best binary trees
1990	Ravindra K. Ahuja et al.	Faster algorithms for the shortest path problem
1993	El-Amin et al.	An expert system for transmission line route selection
1997	David Eppstein	Finding the k Shortest Paths
1997	Ingo Althöfer	On the K-best mode in computer chess: measuring the similarity of move proposals
1999	Ingo Althöfer	Decision Support Systems With Multiple Choice Structure
2011	Husain Aljazzar, Stefan Leue	K*: A heuristic search algorithm for finding the k shortest paths

Tabella 3.1: Algoritmi per la soluzione del problema dei k cammini minimi

3.2.2 Soluzione

Il National Institute of Standards and Technology, un'agenzia del governo degli Stati Uniti d'America che si occupa della gestione delle tecnologie, ha chiamato questo problema "tutti i cammini semplici"[1], e, secondo il professore Paul E. Black[2], si risolve utilizzando una versione modificata della DFS: la ricerca può evitare di ripetere i vertici contrassegnandoli come visitati durante la ricorsione, quindi rimuovere il contrassegno poco prima di ritornare dalla chiamata ricorsiva. In questo modo otteniamo, nel peggiore dei casi, una complessità temporale di $\Omega(V+E)$ e una complessità spaziale di $O(V)$.[7]

3.3 Rappresentazione grafica del grafo

Per rappresentare graficamente il grafo è stata utilizzata la libreria Graphviz[3] della AT&T Labs Research, rilasciata sotto licenza Eclipse Public License.

[1] http://xlinux.nist.gov/dads/HTML/allSimplePaths.html
[2] http://hissa.nist.gov/~black
[3] http://www.graphviz.org

4 Progettazione

Il progetto verrà sviluppato come un'applicazione grafica, per la visualizzazione del grafo, e una libreria statica per la gestione delle strutture dati e degli algoritmi. I linguaggi di programmazione utilizzati saranno rispettivamente il C++11 e il C99.[8]

4.1 Strumenti utilizzati

Lo sviluppo avverrà su piattaforma Windows, utilizzando i seguenti strumenti:

Sistema operativo	Microsoft - Windows XP 5.1[4]
Linguaggi di programmazione	C99, C++11
GUI application framework	Qt Project - Qt 5.5[5]
Disegno grafo	AT&T Labs Research - Graphviz[6]
IDE	Oracle Corporation - NetBeans 8.0.2[7]
GUIs designer	Qt Project - Qt Designer 3.4[8]
Compilatori	MinGW Project - MinGW 0.6[9]

Tabella 4.1: Strumenti utilizzati

4.2 Grafo casuale

Per risolve il primo problema, cioè quello di creare un grafo orientato casuale, è stato progettato un algoritmo, Graph_Random, che ha tempo di esecuzione O(E) e complessità spaziale O(1). L'algoritmo richiede i seguenti parametri: min_vertices, max_vertices, min_edges_for_vertex, max_edges_for_vertex, min_edges, max_edges.

4.3 K cammini differenti

Il secondo problema, può essere risolto utilizzando una versione modificata della DFS: la ricerca può evitare di ripetere i vertici contrassegnandoli come visitati durante la ricorsione, quindi rimuovere il contrassegno poco prima di ritornare dalla chiamata ricorsiva. In questo modo otteniamo, nel peggiore dei casi, una complessità temporale di $\Omega(V+E)$ e una complessità spaziale di $\Omega(V)$ per l'elaborazione, poiché esistono $\Omega(2^n)$ percorsi di dimensione O(V), serve una complessità spaziale di $O(k2^n)$ per la memorizzazione del risultato.

[4] http://www.microsoft.com
[5] http://www.qt.io
[6] http://graphviz.org
[7] http://netbeans.org
[8] http://www.qt.io
[9] http://mingw.org

4.3.1 KPaths

KPaths è un algoritmo che lavora su un grafo, formato da liste di adiacenza, in grado di trovare tutti i cammini esistenti tra un nodo i e un nodo f.

I parametri necessari all'algoritmo sono:

- [IN] G, grafo formato da liste di adiacenza (Adj)
- [OUT] cammini, array di liste
- [IN-OUT] visitati, array di nodi
- [IN] inizio, il nodo di partenza
- [IN] fine, il nodo di destinazione
- [IN] max_k, numero massimo di cammini da cercare
- [IN-OUT] cur_k, numero attuale di cammini trovati

4.3.1.1 L'algoritmo

```
KPaths(G, cammini, visitati, inizio, fine, max_k, cur_k)
01      ultimo = Vector-Back(visitati)
02      foreach n ∈ G.Adj[ultimo]
03              Presente = Vector-Contains(visitati, n)
04              Procedi = (Presente and inizio == fine and n = inizio)
05              if Presente and !Procedi
06                      continue
07              if n == fine
08                      Vector-Add(visitati, n)
09                      for i = 0 to visitati.Length
10                              if visitati[i]
11                                      List-Add(cammini[cur_k], visitati[i])
12                      cur_k++
13                      if cur_k ≥ max_k
14                              return
15                      Vector-Delete(visitati, visitati[0].Index + visitati.Length)
16                      break
17      foreach n ∈ G.Adj[ultimo]
18              if n == fine or Vector-Contains(visitati, n)
19                      continue
20              Vector-Add(visitati, n)
21              KPaths(G, cammini, visitati, inizio, fine, max_k, cur_k)
22              if cur_k ≥ max_k
23                      return
24              Vector-Delete(visitati, visitati[0].Index + visitati.Length)
```

4.3.1.2 Funzionamento

KPaths memorizza in visitati, un array di nodi, i nodi scoperti tra quelli adiacenti al nodo attuale. Nel caso in cui il nodo attuale sia il nodo finale, allora, viene creata una nuova lista formata dai nodi presenti in visitati, e viene memorizzata nell'array cammini. Una volta arrivati ad un nodo senza adiacenze, viene eliminato l'ultimo nodo nell'array dei visitati e si torna indietro di un livello di ricorsione, così facendo non si può rivisitare uno stesso percorso. Nel caso in cui si tornasse al nodo iniziale il processo si comporterebbe come nel caso di un nodo senza adiacenze.

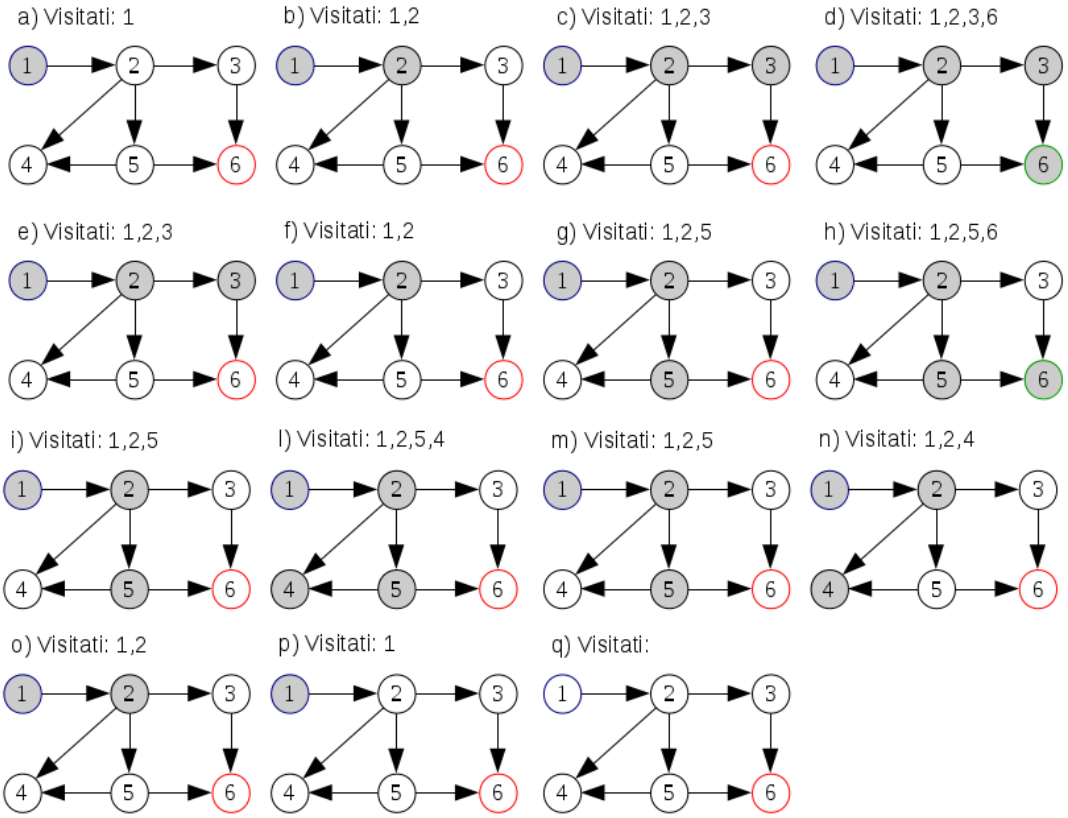

Illustrazione 1: Funzionamento di KPaths

4.4 Rappresentazione grafica del grafo

Per rappresentare graficamente il grafo è sorta l'esigenza di utilizzare le potenzialità dei linguaggi ad oggetti, e si è passati al C++11. È stata utilizzata la libreria Graphviz[10] della AT&T Labs Research, rilasciata sotto licenza Eclipse Public License. È stata creata un interfaccia grafica tramite il GUI application framework Qt 5.5 della Qt Project, sotto licenza GPL-3.0.[9]

[10] http://www.graphviz.org

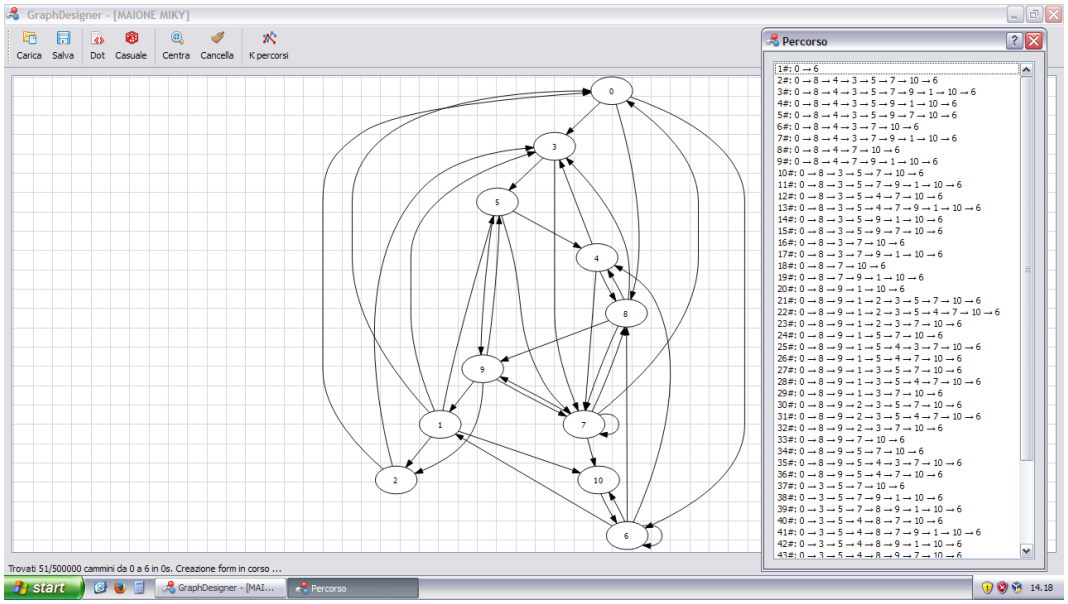

Illustrazione 2: GUI per la progettazione di grafi

4.4.1 Funzionalità della GUI

L'interfaccia permette di generare un grafo casuale, utilizzando la funzione di cui abbiamo parlato nel paragrafo 4.2, e di trovare i k cammini da un nodo ad un altro. Permette anche di caricare e salvare i grafi nel linguaggio DOT e tramite una finestra di editarlo direttamente.

4.4.2 Esempio di linguaggio DOT

```
digraph Grafo1 {
  1->{2,3}
  3->{1}
  4->{5}
  5->{1,2}
}
```

5 Benchmarks

Molto importante per qualsiasi algoritmo è la complessità asintotica e spaziale. Dopo aver scritto gli algoritmi e averli codificati in C99, sono stati compilati e sono stati eseguiti alcuni test per scoprire le effettive risorse utilizzate.

5.1 Elaboratore utilizzato

Per i test è stato utilizzato un computer con le seguenti caratteristiche:

Sistema operativo	Microsoft Windows 10
Architettura	64 bit
Processore	AMD E1-1500 1,48 GHz
Numero di core	2
Memoria RAM	4 GB

Tabella 5.1: Computer utilizzato per i test

5.2 Risultati

I risultati ottenuti sono i seguenti:

Nodi	10	100	1.000
Archi	100	10.000	1.000.000
Min N/E	1	10	100
Max N/E	5	50	500
Tempo (s)	0	1	62
Percorsi possibili: $\Omega(2^n)$	2^{10}	2^{100}	$2^{1.000}$
Percorsi cercati	10.000	10.000	10.000
Percorsi trovati	691	10.000	10.000
Memoria (MB)	1,8	23,6	238,9
CPU	2%	60%	100%

Tabella 5.2: Benchmarks

6 Conclusioni

Dai risultati ottenuti possiamo dedurre che l'algoritmo di generazione di un grafo casuale è velocissimo, infatti ha complessità asintotica di O(E) e per generare un grafo casuale di 1.000 nodi e 1.000.000 archi impiega meno di 1 secondo.

Per quanto riguarda, invece la ricerca dei k cammini, abbiamo una complessità di Ω(V+E), e una complessità spaziale di O(V) se stampassimo direttamente i nodi senza memorizzarli. Per memorizzarli in un vettore di liste, purtroppo utilizziamo una complessità spaziale di $O(k2^n)$ che si traduce, nel caso di un grafo di 1.000 nodi e 1.000.000 archi con k uguale a 10.000, in uno spazio allocato di 244.633KB, che equivale a 24,46-KB per cammino trovato, cammino composto da O(1.000.000) archi.

Per quanto concerne la rappresentazione grafica del grafo, potremmo definire la capacità di disegno della libreria Graphviz stupefacente, peccato sia stata poco utilizzabile per il nostro algoritmo dei k cammini.

7 Ringraziamenti

È d'obbligo ringraziare il Dr. Vadim Malvone per i consigli e la disponibilità e il Prof. Dr. Aniello Murano per avermi assegnato questo progetto che mi ha fatto appassionare ancor di più alla teoria dei grafi.

8 Appendice delle strutture dati

I. Grafo

```
struct nodo {
   void *key;
   struct nodo *p;
};

struct grafo {
   unsigned long n_Nodi, n_Archi;
   struct nodo **V;
   struct lista **Adj;
};
```

II. Lista

```
struct lista {
   void *key;
   struct lista *next;
};
```

III. Vector

```
struct vector
{
   void **A;
   unsigned long lunghezza, indice;
};
```

9 Appendice degli algoritmi

IV. Graph_Random

```
long random_number(long min_num, long max_num)
{
   long low_num = 0;
   long hi_num = 0;

   if (min_num < max_num)
   {
      low_num = min_num;
      hi_num = max_num + 1;
   }
   else
   {
      low_num = max_num + 1;
      hi_num = min_num;
   }

   return ((rand() % (hi_num - low_num)) + low_num);
}
```

```
struct grafo *Grafo_Random(
          long min_vertices,
          long max_vertices,
          long min_edges_for_vertex,
          long max_edges_for_vertex,
          long min_edges,
          long max_edges) //O(E)
{
   long da, a, y, x, n_edges, cur_edges;
   long n_vertices, possible_max_edges, used_edges, tentativi_a;

   used_edges = 0;
   n_vertices = random_number(min_vertices, max_vertices);

   if (max_edges_for_vertex > n_vertices - 1)
      max_edges_for_vertex = n_vertices - 1;

   possible_max_edges = n_vertices * max_edges_for_vertex;

   if (max_edges > possible_max_edges)
      max_edges = possible_max_edges;

   if (min_edges < max_edges)
      n_edges = random_number(min_edges, max_edges);
   else
      n_edges = max_edges;

   struct grafo *foo = Grafo_New2(sizeof (long), n_vertices, "%lu");

   for (y = 0; y < n_vertices; y++)
   {
      struct nodo *n = Nodo_New2(y, y);
      foo->V[y] = n;
      foo->Adj[y] = Lista_New(n);
   }

   bool usati[n_vertices];
   for (da = 0; da < n_vertices; da++) //O(E)
   {
      for (y = 0; y < n_vertices; y++)
         usati[y] = false;

      cur_edges = random_number(min_edges_for_vertex, max_edges_for_vertex);

      for (x = 0; x < cur_edges; x++)
      {
         tentativi_a = 0;

         do
         {
            a = random_number(0, n_vertices - 1);
            tentativi_a += 1;
```

```
            if (tentativi_a > 30)
               for (y = 0; y < n_vertices; y++)
                  if (usati[y] == false)
                  {
                     a = y;
                     break;
                  }
         }
         while (usati[a]);

         used_edges += 1;
         usati[a] = true;
         Grafo_CollegaNodi(foo, da, a);

         if (used_edges > max_edges)
            return foo;
      }
   }

   return foo;
}
```

V. KPath

```
long Grafo_KPath(
         struct grafo *G,
         struct lista **cammini,
         long id_da,
         long id_a,
         long k,
         double *elapsed_secs) //Ω(V+E)
{
   time_t start, end;
   long cur_k = 0;
   long n_iterazioni = 0;

   struct vector *visitati = Vector_New(sizeof (struct nodo), G->n_Nodi);
   Vector_Add(visitati, G->V[id_da]);

   time(&start);
   Grafo_KPath2(G, cammini, visitati, G->V[id_da], G->V[id_a], &cur_k, k, &n_iterazioni);
   time(&end);
   *elapsed_secs = difftime(end, start);

   return cur_k;
}

long Grafo_KPath3(
         struct grafo *G,
         struct lista **cammini,
         long id_da,
         long n_destinazioni,
         long *ids_a,
         long k,
         double *elapsed_secs) //Ω(V+E)
```

```
{
   long j = G->n_Nodi;
   G = Grafo_Add(G, j + 1);

   for(long i = 0; i < n_destinazioni; i++)
      Grafo_CollegaNodi(G, ids_a[i], j);

   return Grafo_KPath(G, cammini, id_da, j, k, elapsed_secs, true);
}

void Grafo_KPath2(
         struct grafo *G,
         struct lista **cammini,
         struct vector *visitati,
         struct nodo *inizio,
         struct nodo *fine,
         long *cur_k,
         long k,
         long *n_iterazioni)
{
   struct nodo *ultimo = Vector_Back(visitati);
   struct lista *n = G->Adj[ultimo->Index];

   while (n = n->next)
   {
      bool Presente = Vector_Contains(visitati, n->key);
      bool SonoInizio = (Presente && inizio == fine && n->key == inizio);

      if (Presente && !SonoInizio)
         continue;

      if (n->key == fine)
      {
         Vector_Add(visitati, n->key);

         for (long i = 0; i <= visitati->indice; i++)
            if (visitati->A[i])
               if (cammini[*cur_k ])
                  Lista_AggiungiSuccessivo(cammini[*cur_k ], ToNodo(visitati->A[i])->key);
               else
                  cammini[*cur_k ] = Lista_New(ToNodo(visitati->A[i])->key);

         *cur_k += 1;
         if (*cur_k >= k)
            return;

         Vector_Delete(visitati, ToNodo(visitati->A[0])->Index + visitati->indice);
         break;
      }
   }

   n = G->Adj[ultimo->Index];
   while (n = n->next)
   {
```

```
        if (Vector_Contains(visitati, n->key) || n->key == fine)
            continue;

        Vector_Add(visitati, n->key);
        Grafo_KPath2(G, cammini, visitati, inizio, fine, cur_k, k, n_iterazioni);
        if (*cur_k >= k)
            return;

        Vector_Delete(visitati, ToNodo(visitati->A[0])->Index + visitati->indice);
    }
}
```

Bibliografia

1: contributori di Wikipedia, Grafo, 2015, https://it.wikipedia.org/wiki/Grafo
2: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduzione agli algoritmi e strutture dati, 2010
3: contributori di Wikipedia, Teoria dei grafi, 2015, https://it.wikipedia.org/wiki/Teoria_dei_grafi
4: Istituto dell'Enciclopedia Italiana, Enciclopedia Italiana di scienze, lettere ed arti, 2015
5: Prof. Dr. Paolo Ceravolo, Dall'ipertesto alla macchina sociale, 2009, http://sesar.dti.unimi.it/TecNuoviMedia/Lezioni
6: Wikipedia contributors, K shortest path routing , 2015, https://en.wikipedia.org/wiki/K_shortest_path_routing
7: Vreda Pieterse and Paul E. Black, Dictionary of Algorithms and Data Structures, 2008, http://www.nist.gov/dads/HTML/allSimplePaths.html
8: Brian W. Kernighan, The C Programming Language: ANSI C Version, 1988
9: Stanley B. Lippman, Josée Lajoie, Barbara E. Moo, C++ Primer, 2012