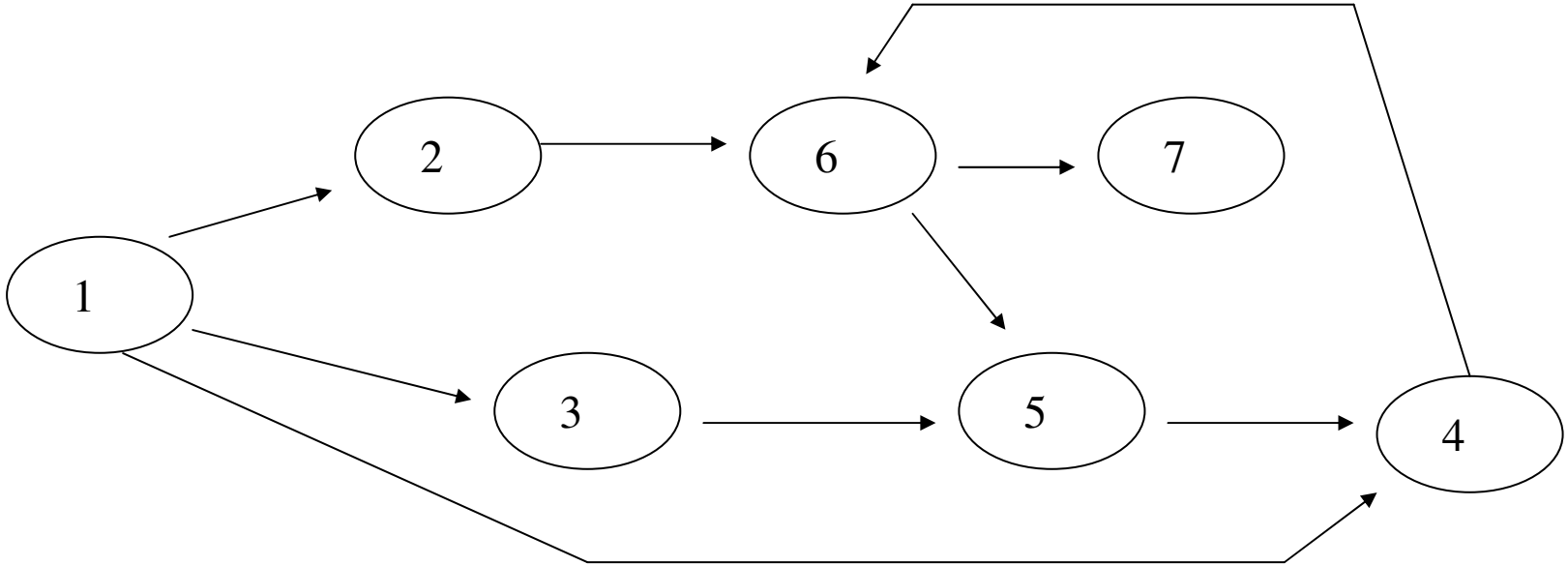
Rappresentazione di grafi mediante lista di archi




```
type 'a graph = Gr of ('a * 'a) list;;
let grafo1 = Gr [(1,2);(1,3);(1,4);(2,6); (3,5);(4,6);(6,5);(6,7);(5,4)];;
```

Il costruttore Gr è usato per distinguere il tipo dei grafi dal tipo ('a * 'a) list

Ovviamente, possiamo anche identificare grafi e liste di archi:

```
type 'a graph = ('a * 'a) list
let grafo1 = [(1,2);(1,3);(1,4);(2,6); (3,5);(4,6);(6,5);(6,7);(5,4)];;
```

Lista dei successori di un nodo

```
(* succ : 'a graph -> 'a -> 'a list *)
let succ (Gr arcs) node =
        let rec aux = function
                [] -> []
                | (x,y)::rest -> if x = node then y::aux rest
                                                else aux rest
in aux arcs;;

let rec succ (Gr arcs) node =
        match arcs with
                [] -> []
                | (x,y)::rest -> if x = node then y::succ (Gr rest) node  (* rest è visto come una lista di coppie
        *)
                                                else succ (Gr rest) node;;
```

OPPURE

```
(* succ : 'a graph -> 'a -> 'a list *)
let succ (Gr arcs) node =
      List.map snd (List.filter (function (x,y) -> x=node) arcs);;
```

arcs = [(1,2);(1,3);(1,4);(2,6); (3,5);(4,6);(6,5);(6,7);(5,4)];; node = 1

[(1,2);(1,3);(1,4)]

[2;3;4]

Successori in un grafo non orientato

```
(* succ : 'a graph -> 'a -> 'a list *)
let rec succ_in_nog (Gr arcs) node =
      let rec aux = function
            [] -> []
            | (x,y)::rest -> if x = node then y::aux rest
                                    else if y = node then x::aux rest
                                                else aux rest
in aux arcs
```

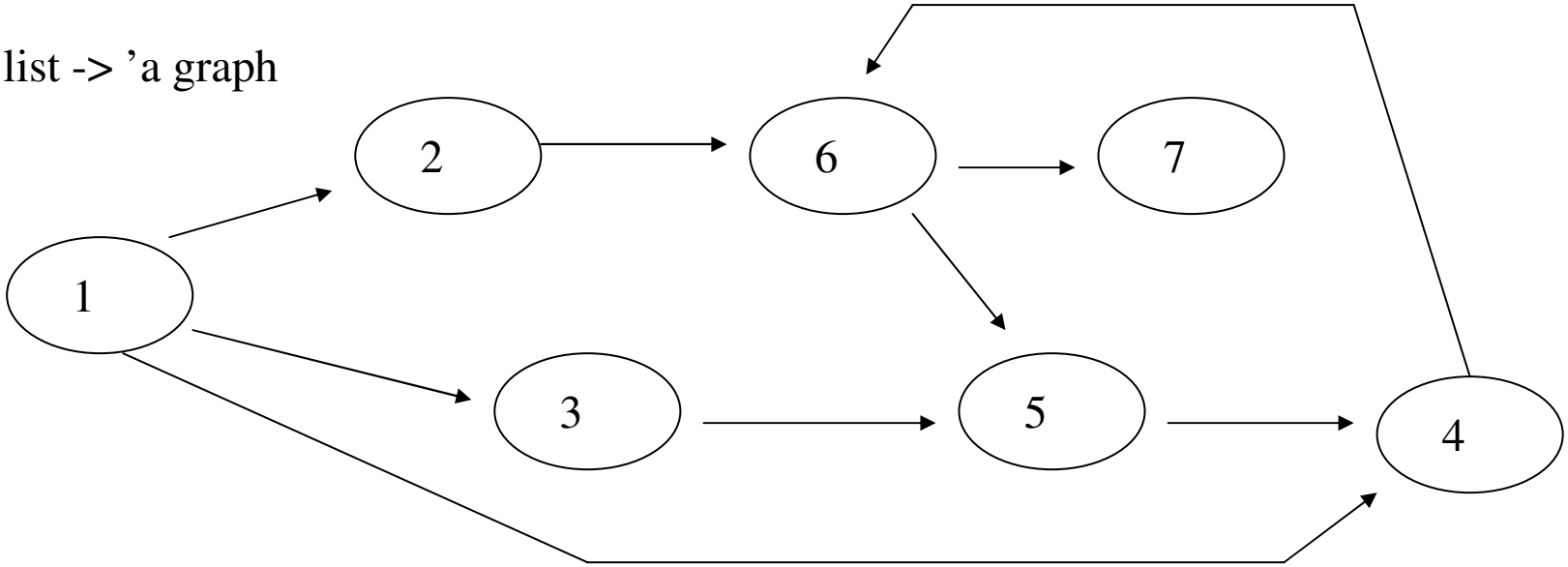
Lista dei nodi del grafo

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

Esercizio

Può essere utile avere una funzione che trasformi nella rappresentazione “interna” una rappresentazione più compatta del grafo:

paths2graph: ’a list list -> ’a graph



paths2graph [[1;2;6;7];[1;3;5;4;6;5];[1;4]]
paths2graph [[1;4;6;5];[1;3;5;4;6;7];[1;2]]

La rappresentazione “compatta” è una lista L i cui elementi sono cammini del grafo (liste di nodi), tali che per ogni arco (x,y) del grafo, L contiene uno ed un unico cammino in cui x e y sono elementi adiacenti.

Implementazione di paths2graph: per esercizio

```
let paths2graph  cammini=
        let rec aux risultato = function
            [] ->  risultato
            | []::rest -> aux risultato rest
            | [x]::rest -> aux risultato rest
            | (x::y :: coda) ::rest -> aux ((x,y)::risultato) ((y :: coda) ::rest)
        in aux [] cammini ;;
```

Rappresentazione di grafi orientati tramite lista di successori

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

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

(* succ : 'a graph -> 'a -> 'a list *)
let succ (Gr succlist) node =
        try List.assoc node succlist with Not_found -> [];;

(* nodes : 'a graph -> 'a list *)
let nodes (Gr succlist) =
        let setadd x set = if List.mem x set then set
                                                else x::set
        in let rec union set = function
                [] -> set
                | x::rest -> setadd x (union set rest)
        in let rec aux = function
                [] -> []
                | (n,lst)::rest -> union (n::lst) (aux rest)
in aux succlist ;;
```

Algoritmi di visita

A partire da un nodo iniziale dato (punto di ingresso), analizzare tutti i nodi raggiungibili dal nodo dato.

Attenzione ai cicli.

Visita in profondità (depth first search): se il nodo di partenza start non è stato già visitato,
si analizza start e,
per ogni successore x di start, si visita, con lo stesso metodo, il grafo a partire da x,
ricordando che start è già stato visitato.
I nodi in attesa di essere visitati vengono gestiti come una **pila**

Visita in ampiezza (breadth first search): se il nodo di partenza start non è già stato visitato
allora si analizza start,
si visitano tutti i suoi successori (ricordando che start è già stato considerato),
poi tutti i successori dei successori di start, e così via.
I nodi in attesa di essere visitati vengono gestiti come una **coda**

Ricerca in profondità per collezionare i nodi del grafo

```
(* depth_first_collect : 'a graph -> 'a -> 'a list *)
let depth_first_collect graph start =
      let rec search visited = function
            [] -> visited
            | n::rest -> if List.mem n visited
                  then search visited rest
                  else search (n::visited) ((succ graph n) @ rest)
                  (* i nuovi nodi sono inseriti in testa *)
in search [] [start];;


# depth_first_collect grafo1 1;;
- : int list = [3; 7; 4; 5; 6; 2; 1]
```

Esercizio: implementare la visita in profondità con una funzione

depth_first : 'a graph -> ('a -> bool) -> 'a -> 'a

che applicata a un grafo, a un predicato di nodi e un nodo di ingresso, riporti (se esiste) un nodo raggiungibile da quello di ingresso che soddisfa il predicato; un errore altrimenti.

```
(* depth_first_collect : 'a graph -> 'a -> 'a list *)
let depth_first_cerca  graph f start  =
      let rec search visited = function
            [] -> false
            | n::rest -> if List.mem n visited
                  then search visited rest
                  else if f n then true
                        else search (n::visited) ((succ graph n) @ rest)
                  (* i nuovi nodi sono inseriti in testa *)
in search [] [start];;
```

Ricerca in ampiezza per collezionare i nodi del grafo

```
(* breadth_first_collect : 'a graph -> 'a -> 'a list *)
let breadth_first_collect graph start =
      let rec search visited = function
            [] -> visited
            | n::rest -> if List.mem n visited
                  then search visited rest
                  else search (n::visited) (rest @ (succ graph n))
                  (* i nuovi nodi sono inseriti in coda *)
in search [] [start];;
```

```
# breadth_first_collect grafo1 1;;
- : int list = [7; 5; 6; 4; 3; 2; 1]
```

Ricerca di un cammino mediante backtracing

Nei grafi, si deve fare attenzione ai cicli.

from node: viene applicata a una lista di nodi già visitati e un singolo nodo e riporta una soluzione path, se esiste, un errore altrimenti; richiama la funzione:

from list: si applica a una lista di nodi già visitati e una lista di nodi (i successori di uno stesso nodo); per ogni nodo di questa seconda lista viene richiamata la funzione from node; al primo successo riportato con risultato path, la ricerca si interrompe e la funzione riporta lo stesso valore path;
se invece tutte le chiamate a from node falliscono, allora from list solleva l'eccezione NotFound.

```
exception NotFound;;
(* search_path : 'a graph -> ('a -> bool) -> 'a -> 'a list *)
let search_path graph p start =
      let rec from_node visited a =
            if List.mem a visited then raise NotFound
                                    else if p a then [a] (* il cammino e' trovato *)
                                                else a::from_list (a::visited) (succ graph a)
      and from_list visited = function
            [] -> raise NotFound
            | a::rest -> try from_node visited a
                         with NotFound -> from_list visited rest
in from_node [] start;;

# search_path grafo1 (function x -> x=7) 1;;
- : int list = [1; 2; 6; 7]
```

and serve a definire funzioni mutuamente ricorsive

Implementazione senza mutua ricorsione

```
exception NotFound;;


(* gpath: 'a graph -> ('a -> bool) -> 'a -> 'a list *)
let gpath g p start =
      let rec aux visited = function
            [] -> raise NotFound
            | x::rest -> if List.mem x visited then aux visited rest
                                                else if p x then [x]
                                                            else try x:: aux (x::visited) (succ g x)
                                                                 with NotFound -> aux (x::visited) rest
in aux [] [start];;
```

Implementazione alternative: ricerca in ampiezza?

Si potrebbe pensare che, per ottenere una ricerca di cammino in ampiezza, sia sufficiente modificare l'ordine delle alternative: prima si cerca un cammino a partire dai "fratelli" del nodo considerato, poi a partire dai successori:

```
(* bf_path: 'a graph -> ('a -> bool) -> 'a -> 'a list *)
let bf_path g p start =
      let rec aux visited = function
            [] -> raise NotFound
            | x::rest -> if List.mem x visited then aux visited rest
                                                else if p x then [x]
                                                      else try aux (x::visited) rest
                                                        with NotFound ->
                                                              x:: aux (x::visited) (succ g x)
in aux [] [start];;
```

Ma non è così:

```
let grafo = Gr [(1,[2;3]);(2,[4]);(3,[5]);(5,[4])] ;;

# bf_path grafo ((=) 4) 1;;
-: int list = [1; 3; 5; 4]
```

Rappresentazione di grafi mediante funzioni

Per risolvere diversi problemi sui grafi, viene utilizzata soltanto la funzione succ

Si può rappresentare un grafo direttamente mediante una funzione “successori”

```
(* f:int->intlist*)
let f = function
        1 -> [2;3;4]
      | 2 -> [1;3;5]
      | 3 -> [5;6]
      | 4 -> [3]
      | 6 -> [4]
      | _ -> [];;

type ’a graph = Graph of (’a -> ’a list);;

# let g = Graph f;;
val g : int graph = Graph <fun>
```

Visita in ampiezza

```
(* bf : 'a graph -> ('a -> bool) -> 'a -> 'a *)
let bf (Graph succ) p start =
        let rec search visited = function
                [] -> raise NotFound
                | a::rest -> if List.mem a visited then search visited rest
                                                        else if p a then a
                                                                else search (a::visited) (rest@(succ a))
in search [] [start];;

# let lessthan x y = y<x;;
val lessthan : 'a -> 'a -> bool = <fun>

# bf g (lessthan 4) 6;;
-: int = 3
```

Ricerca di un cammino mediante backtracing

```
(* search_path : 'a graph -> ('a -> bool) -> 'a -> 'a list *)
let search_path (Graph succ) p start =
      (* from_node cerca un cammino a partire dal nodo a *)
      let rec from_node visited a =
            if List.mem a visited
                  then raise NotFound
                  else if p a then [a] (* il cammino e' trovato *)
                              else a::from_list (a::visited)(succ a)
                                          (* cerca un cammino da uno dei successori di a *)
(*    from_list cerca un cammino che parta da uno dei nodi della lista suo argomento *)
            and from_list visited = function
                  [] -> raise NotFound
                  | a::rest -> try from_node visited a
                               with NotFound -> from_list visited rest
in    from_node [] start;;
```

Esempi

```
# search_path g ((=) 6) 1;;
-: int list = [1; 2; 3; 6]
```

Grafo costituito dai numeri interi compresi tra 1 e 99,
in cui i successori di un nodo n sono 2n,3n e5n – se minori di 100

```
let mult235 n = List.filter (function n -> n< 100) [2*n;3*n;5*n];;
```

```
# search_path (Graph mult235) ((=) 30) 1;;
-: int list = [1; 2; 6; 30]
```

Esempio: il problema dei 2 boccali

Si hanno a disposizione due boccali non graduati, uno da 4 litri e uno da 3 litri, un lavandino e un rubinetto. Si vogliono ottenere esattamente 2 litri d'acqua nel boccale da 4 litri

Operazioni possibili: riempire un boccale, svuotarlo nel lavandino, travasare acqua da un boccale a un altro.

Nodi del grafo: (X,Y) stato in cui ci sono X litri nel boccale da 4 litri e Y nel boccale da 3

Il problema dei due boccali è un esempio di problema di ricerca in uno spazio di stati.
Uno spazio di stati è come un grafo, ma la sua rappresentazione esplicita come insieme di nodi e archi è spesso molto complessa, a volte impossibile (infinito).

Uno spazio di stati viene rappresentato implicitamente, mediante l'insieme degli operatori che, applicati a uno stato, riportano i suoi successori.

Gli operatori

Definiamo gli operatori, che, applicati a uno stato, generano un suo successore:

int * int -> int * int

Alcuni operatori non sono applicabili ad alcuni stati

```
exception Fail;;

let riempiX = function
 (4,y) -> raise Fail
| (x,y) -> (4,y);;
let riempiY = function
(x,3) -> raise Fail
| (x,y) -> (x,3);;
let svuotaX = function
(0,y) -> raise Fail
| (x,y) -> (0,y);;
let svuotaY = function
(x,0) -> raise Fail
| (x,y) -> (x,0);;
```

```
let riempiXdaY = function
      (4,_) | (_,0) -> raise Fail
      | (x,y) -> if x+y < 4 then raise Fail
                            else (4,y-4+x);;
let riempiYdaX = function
      (0,_) | (_,3)-> raise Fail
      | (x,y) -> if x+y < 3 then raise Fail
                            else (y-3+x,3);;
let vuotaXinY = function
      (0,y) -> raise Fail
      | (x,y) -> if x+y > 3 then raise Fail
                             else (0,x+y);;
let vuotaYinX = function
      (x,0) -> raise Fail
      | (x,y) -> if x+y > 4 then raise Fail
                            else (x+y,0);;
```

```
exception Fail;;
let riempiX = function
 (4,y) -> raise Fail
| (x,y) -> (4,y);;
let riempiY = function
(x,3) -> raise Fail
| (x,y) -> (x,3);;
let svuotaX = function
(0,y) -> raise Fail
| (x,y) -> (0,y);;                                                    |
let svuotaY = function
(x,0) -> raise Fail
| (x,y) -> (x,0);;
let riempiXdaY = function
        (4,_) | (_,0) -> raise Fail
        | (x,y) -> if x+y < 4 then raise Fail
                                    else (4,y+x-4);;
let riempiYdaX = function
        (0,_) | (_,3)-> raise Fail
        | (x,y) -> if x+y < 3 then raise Fail
                                    else (y+x-3,3);;
let vuotaXinY = function
        (0,y) -> raise Fail
        | (x,y) -> if x+y > 3 then raise Fail
                                     else (0,x+y);;
let vuotaYinX = function
        (x,0) -> raise Fail
        | (x,y) -> if x+y > 4 then raise Fail
                                    else (x+y,0);;
```

La funzione successori

```
(* boccali : int * int -> (int * int) list *)
let boccali stato =
      let rec aux =
            function
                  [] -> []
                  | f::rest -> try f stato :: aux rest
                                    with Fail -> aux rest
in aux [riempiX; riempiY; svuotaX; svuotaY; riempiXdaY; riempiYdaX; vuotaXinY; vuotaYinX];;

# boccali  (0,0);;
-: (int * int)  list = [(4, 0); (0, 3)]

# boccali (4,0);;
-: (int * int) list = [(4, 3); (0, 0); (1, 3)]
```

Il test obiettivo

```
(* goal : int * int -> bool *)
let goal (x,_) = x=2;;
```

Ricerca del cammino soluzione (in profondità):

```
# search_path (Graph boccali) goal (0,0);;
  -: (int * int) list = [(0, 0); (4, 0); (4, 3); (0, 3); (3, 0); (3, 3); (4, 2); (0, 2); (2, 0)]
```

La ricerca di un cammino in ampiezza si può implementare mediante la visita in ampiezza del grafo dei cammini corrispondente al grafo dato

Il grafo dei cammini

Dato un grafo G, possiamo costruire un grafo G* i cui nodi sono cammini in G (liste di nodi)
Se G è un 'a graph, G* è un 'a list graph

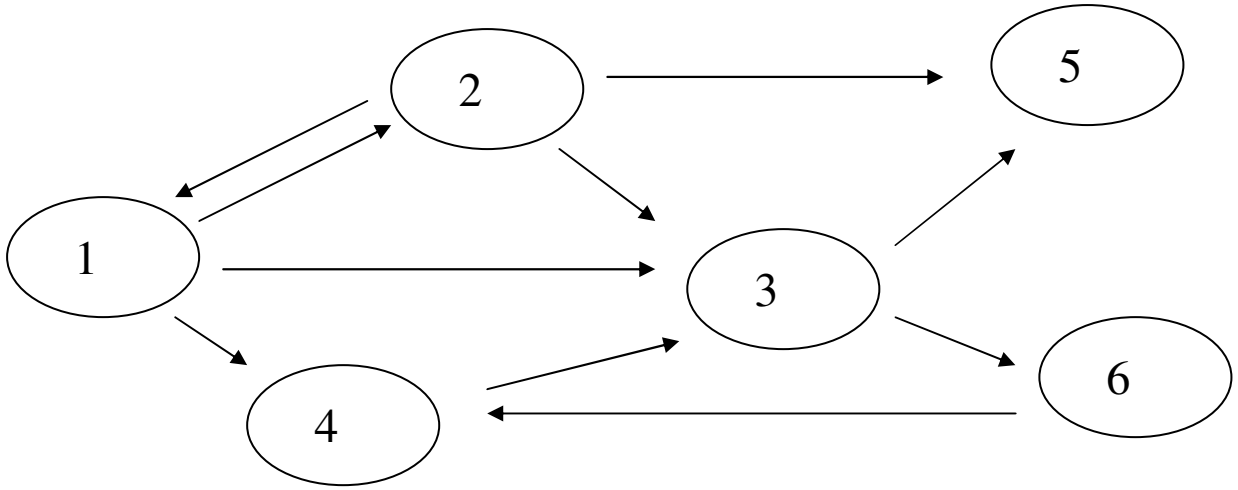
Rappresentiamo un cammino con la lista rovesciata rispetto al cammino;

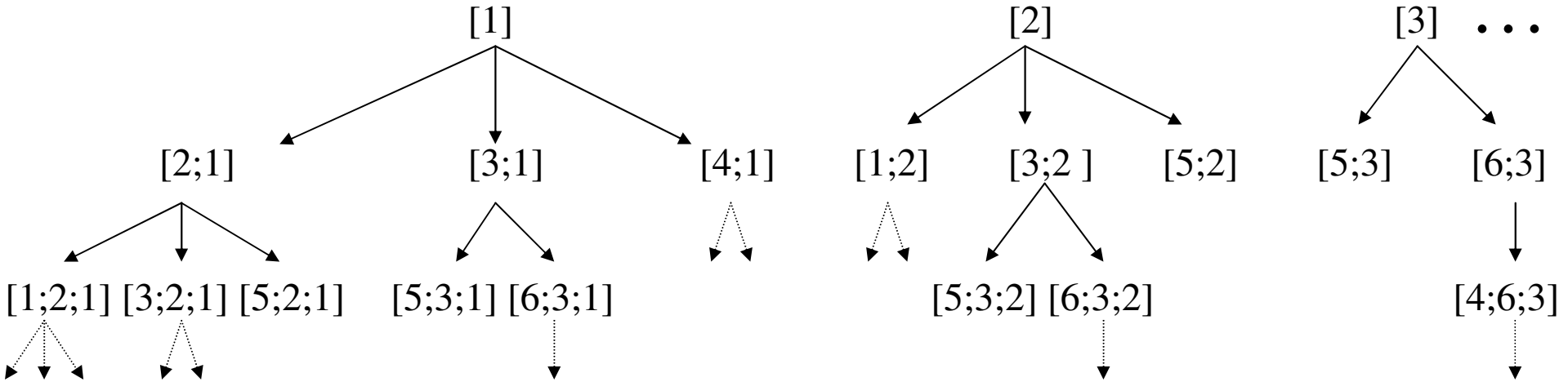
ad esempio, il cammino <1, 2, 6, 7> è rappresentato dalla lista [7;6;2;1].

I successori di un nodo (a::p) di G* sono tutti i cammini (b::a::p) tali che b è un successore di a in G.

Se G ha n nodi, G* è costituito da n componenti non connesse: ciascuna contiene tutti i cammini che iniziano in un nodo dato.

Esempio





Per definire G* è sufficiente definire una funzione che, applicata a una funzione “successori”
(la funzione che rappresenta G), riporta la funzione “cammino successore” (che rappresenta G*)

```
(* succpath : (’a -> ’a list) -> ’a list -> ’a list list *)
let succpath succ path =
        let rec aux = function
                [] -> []
                | b::rest -> (b::path)::(aux rest)
in aux (succ (List.hd path));;
```

Oppure:

```
(* consto : ’a list -> ’a -> ’a list *)
let consto lst a =
        a::lst;;

let succpath succ path =
        List.map (consto path) (succ (List.hd path));;
```

Esempi

Graph (succpath f) è il grafo dei cammini corrispondente a Graph f

```
# let fstar = succpath f;;
val fstar : int list -> int list list = <fun>

# fstar [2];;
-: int list list = [[1; 2]; [3; 2]; [5; 2]]

# fstar [1;2];;
-: int list list = [[2; 1; 2]; [3; 1; 2]; [4; 1; 2]]

# fstar [2;1;2];;
-: int list list = [[1; 2; 1; 2]; [3; 2; 1; 2]; [5; 2; 1; 2]]
```

Graph (succpath mult235) è il grafo dei cammini corrispondente a Graph mult235

```
# let fpath = succpath mult235;;
val fpath : int list -> int list list = <fun>

# fpath [2];;
-: int list list = [[4; 2]; [6; 2]; [10; 2]]

# fpath [4;2];;
-: int list list = [[8; 4; 2]; [12; 4; 2]; [20; 4; 2]]
```

Ricerca di un cammino in G
Come ricerca di un nodo nel grafo dei cammini G*

La ricerca di un cammino nel grafo Graph f da un nodo N a un nodo N' con la proprietà P
si riduce alla ricerca nel grafo Graph (succpath f), a partire dal nodo [N],
di un nodo (N'::rest) tale che N' abbia la proprietà P.

La ricerca può avvenire in ampiezza o in profondità.

```
(* hdeq : 'a -> 'a list -> bool *)
let hdeq y path = List.hd path = y;;

# bf (Graph fpath) (hdeq 30) [1];;
-: int list = [30; 6; 2; 1]
```

MA ATTENZIONE:

```
# bf (Graph fstar) (hdeq 30) [1];;
..... LOOP .....
```

Il grafo dei cammini è aciclico: i successori di un cammino sono sempre più lunghi di esso.
Ma, se il grafo di partenza contiene cicli, il grafo dei cammini ha un numero infinito di nodi:
la ricerca in ampiezza, se non esiste un nodo obiettivo, va avanti all'infinito

Visita in ampiezza nel grafo dei cammini

Il test di ciclicità va effettuato sui nodi del grafo G (sul primo elemento di ciascun cammino);
la lista dei nodi già visitati contiene quindi nodi di G e non nodi di G*.

Il predicato p si applica a nodi del grafo G

```
(* funbf : 'a list graph -> ('a -> bool) -> 'a list -> 'a list *)
let funbf (Graph succ) p start =
        let rec search visited = function
        [] -> raise NotFound
        | path::rest -> let a = List.hd path
                        in if List.mem a visited then search visited rest
                                                 else if p a then path
                                                             else search (a::visited) (rest@(succ path))
in search [] [start];;
```

```
# funbf (Graph fstar) ((=) 30) [1];;
Uncaught exception: NotFound
```

Ricerca di un cammino in ampiezza

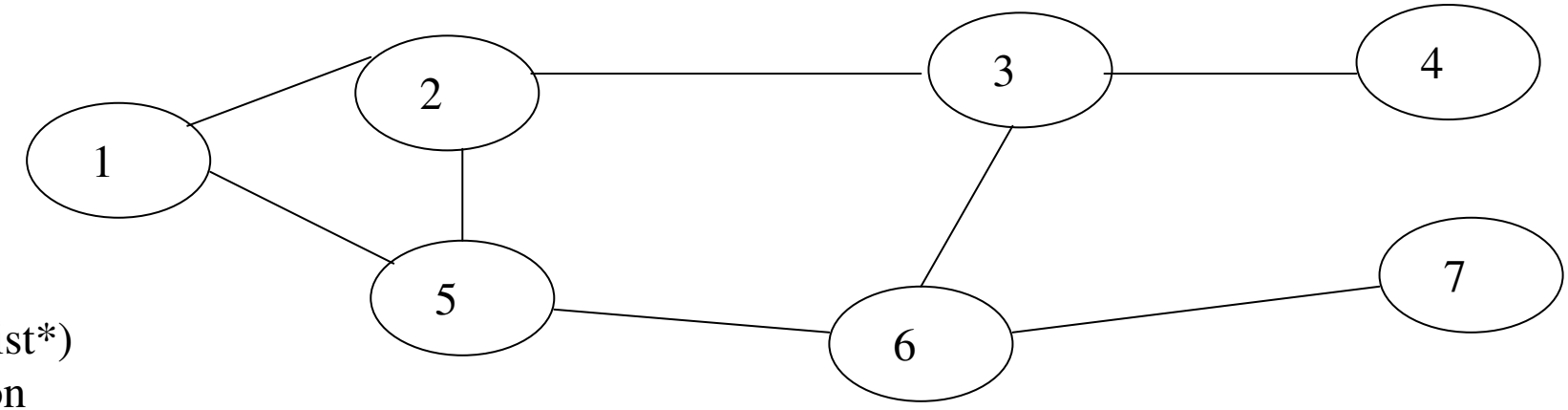
La funzione bf search path cerca un cammino nel grafo G, eseguendo una visita in ampiezza nel grafo dei cammini G*

```
(* bf_search_path : 'a graph -> ('a -> bool) -> 'a -> 'a list *)
let bf_search_path (Graph succ) p start =
        List.rev (funbf (Graph (succpath succ)) p [start]);;

# bf_search_path (Graph mult235) ((<) 20) 1;;
-: int list = [1; 5; 25]

# bf_search_path (Graph f) ((=) 30) 1;;
Uncaught exception: NotFound
```

Soluzione del problema dei 2 boccali
mediante visita in ampiezza del grafo dei cammini

```
(* solve : unit -> (int * int) list *)
let solve () = bf_search_path (Graph boccali) goal (0,0);;

# solve ();;
-: (int * int) list = [(0, 0); (4, 0); (1, 3); (1, 0); (0, 1); (4, 1); (2, 3)]
```

Ricerca con rappresentazione esplicita dei cammini




```
(* f:int->int list*)
let f = function
        1 -> [2;5]
       | 2 -> [1;3;5]
       | 3 -> [2;4;6]
       | 4 -> [3]
       | 5 -> [1;2;6]
       | 6 -> [3;5;7]
       |7 -> [6]
       | _ -> [];;

type ’a graph = Graph of (’a -> ’a list);;

# let g = Graph f;;
val g : int graph = Graph <fun>
```

Schema generale

```
exception NotFound;;
let search inizio fine (Graph succ)=
        let rec search_aux  fine = function
                [] -> raise NotFound
                | cammino::rest -> if fine = List.hd cammino
                        then List.rev cammino
                        else search_aux fine nuova_lista_cammini
in search_aux fine [[inizio]]

 let estendi cammino (Graph succ)=
        let prossimi = (succ (List.hd cammino))
                in let prossimi_buoni = List.filter (function x -> not (List.mem x cammino)) prossimi
                        in List.map (function x -> x::cammino) prossimi_buoni;;


let estendi cammino (Graph succ)=
        List.map (function x -> x::cammino)
                (List.filter (function x -> not (List.mem x cammino)) (succ (List.hd cammino)));;
```

Ricerca in profondità

```
exception NotFound;;
let searchp inizio fine (Graph succ)=
        let estendi cammino =
                List.map (function x -> x::cammino)
                        (List.filter (function x -> not (List.mem x cammino)) (succ (List.hd cammino)))
        in let rec search_aux fine = function
                [] -> raise NotFound
                | cammino::rest -> if fine = List.hd cammino
                        then List.rev cammino
                        else search_aux fine ((estendi cammino) @ rest)
in search_aux fine [[inizio]];;
```

```
# searchp 1 4 g;;
- : int list = [1; 2; 3; 4]
```

```
# searchp 1 7 g;;
- : int list = [1; 2; 3; 6; 7]
```

```
# searchp 2 6 g;;
- : int list = [2; 1; 5; 6]
```

```
# searchp 1 14 g;;
Exception: NotFound.
```

Ricerca in profondità bis

```
exception NotFound;;

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

let searchp inizio fine (Graph 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 search_aux fine = function
                [] -> raise NotFound
                | cammino::rest -> if fine = List.hd cammino
                        then List.rev cammino
                        else search_aux fine ((estendi cammino) @ rest)
in search_aux fine [[inizio]];;


# searchp 1 4 g;;
1;
2; 1;
3; 2; 1;
- : int list = [1; 2; 3; 4]
```

```
# searchp 1 7 g;;
1;
2; 1;
3; 2; 1;
4; 3; 2; 1;
6; 3; 2; 1;
5; 6; 3; 2; 1;
- : int list = [1; 2; 3; 6; 7]

# searchp 2 6 g;;
2;
1; 2;
5; 1; 2;
- : int list = [2; 1; 5; 6]
```

```
# searchp 1 14 g;;
1;                                  [[2;1];[5;1]]
2; 1;                               [[3;2;1]; [5;2;1]; [5;1]]
3; 2; 1;                            [[4;3;2;1]; [6;3;2;1]; [5;2;1]; [5;1]]
4; 3; 2; 1;
6; 3; 2; 1;
5; 6; 3; 2; 1;
7; 6; 3; 2; 1;
5; 2; 1;
6; 5; 2; 1;
3; 6; 5; 2; 1;
4; 3; 6; 5; 2; 1;
7; 6; 5; 2; 1;
5; 1;
2; 5; 1;
3; 2; 5; 1;
4; 3; 2; 5; 1;
6; 3; 2; 5; 1;
7; 6; 3; 2; 5; 1;
6; 5; 1;
3; 6; 5; 1;
2; 3; 6; 5; 1;
4; 3; 6; 5; 1;
7; 6; 5; 1;
                                    Exception: NotFound.
```

Ricerca in ampiezza

```
exception NotFound;;
let searcha inizio fine (Graph succ)=
        let estendi cammino =
                List.map (function x -> x::cammino)
                        (List.filter (function x -> not (List.mem x cammino)) (succ (List.hd cammino)))
        in let rec search_aux fine = function
                [] -> raise NotFound
                | cammino::rest -> if fine = List.hd cammino
                        then List.rev cammino
                        else search_aux fine (rest @ (estendi cammino))
in search_aux fine [[inizio]];;

# searcha 1 4 g;;
- : int list = [1; 2; 3; 4]

# searcha 1 7 g;;
- : int list = [1; 5; 6; 7]

# searcha 2 6 g;;
- : int list = [2; 3; 6]

# searcha 1 14 g;;
Exception: NotFound.
```

Ricerca in ampiezza

```
exception NotFound;;
let searcha inizio fine (Graph 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 search_aux fine = function
                [] -> raise NotFound
                | cammino::rest -> if fine = List.hd cammino
                        then List.rev cammino
                        else search_aux fine (rest @ (estendi cammino))
in search_aux fine [[inizio]];;

# searcha 1 4 g;;
1;
2; 1;
5; 1;
3; 2; 1;
5; 2; 1;
2; 5; 1;
6; 5; 1;
- : int list = [1; 2; 3; 4]
```

```
# searcha 1 7 g;;
1;
2; 1;
5; 1;
3; 2; 1;
5; 2; 1;
2; 5; 1;
6; 5; 1;
4; 3; 2; 1;
6; 3; 2; 1;
6; 5; 2; 1;
3; 2; 5; 1;
3; 6; 5; 1;
- : int list = [1; 5; 6; 7]
# searcha 2 6 g;;
2;
1; 2;
3; 2;
5; 2;
5; 1; 2;
4; 3; 2;
- : int list = [2; 3; 6]
```

```
# searcha 1 14 g;;
1;
2; 1;
5; 1;
3; 2; 1;
5; 2; 1;
2; 5; 1;
6; 5; 1;
4; 3; 2; 1;
6; 3; 2; 1;
6; 5; 2; 1;
3; 2; 5; 1;
3; 6; 5; 1;
7; 6; 5; 1;
5; 6; 3; 2; 1;
7; 6; 3; 2; 1;
3; 6; 5; 2; 1;
7; 6; 5; 2; 1;
4; 3; 2; 5; 1;
6; 3; 2; 5; 1;
2; 3; 6; 5; 1;
4; 3; 6; 5; 1;
4; 3; 6; 5; 2; 1;
7; 6; 3; 2; 5; 1;
Exception: NotFound.#
```

```
[ [1] ]
[[2;1]; [5;1]]
[[5;1]; [3;2;1]; [5;2;1]]
[[3;2;1]; [5;2;1] ;[2;5;1]; [6;5;1]]
```

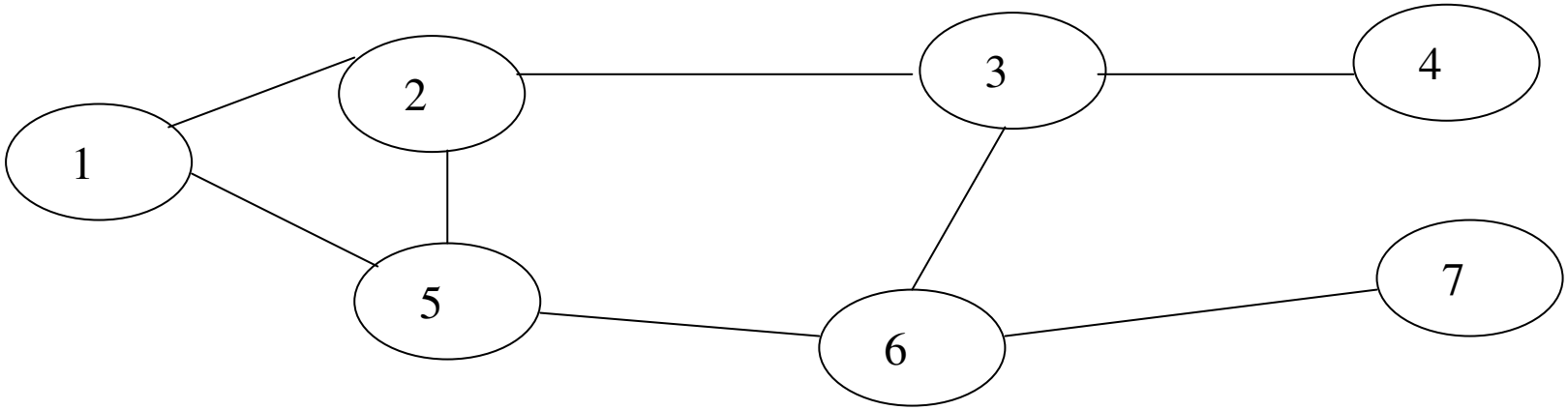
Ricerca euristica della soluzione più promettente

Applicabile quando una funzione di valutazione consente di paragonare due soluzioni parziali allo scopo di selezionare quella che più facilmente potrebbe portare alla soluzione.

In generale la funzione di valutazione non è perfetta: se lo fosse non sarebbe necessaria nessuna ricerca perché ad ogni passo si sceglierebbe l'alternativa giusta. La funzione di valutazione dà una indicazione di massima su quale è la soluzione più promettente.

Nel grafo dell'esempio precedente supponiamo di assegnare ad ogni nodo una coppia di coordinate:

```
# let coordinate = [ (1, (0,3)); (2, (4,6)); (3, (7,6)); (4, (11,6)); (5, (3,0)); (6, (6,0)); (7, (11,3))];;
```



In questo caso la funzione di valutazione calcola quale soluzione parziale porta più vicino al nodo che costituisce la meta

```
let distanza nodo1 nodo2 =
        let x1 = float (fst(List.assoc nodo1 coordinate))
        in let y1 = float (snd(List.assoc nodo1 coordinate))
        in let x2 = float (fst(List.assoc nodo2 coordinate))
        in let y2 = float (snd(List.assoc nodo2 coordinate))
        in sqrt ( (x1 -. x2)**2. +. (y1 -. y2)**2.);;

let piuvicino (cammino1, cammino2, meta) =
        (distanza (List.hd cammino1) meta) < (distanza (List.hd cammino2) meta);;

let confrontacammino cammino1 cammino2 meta=
        if List.hd cammino1= List.hd cammino2
                then 0
                else if piuvicino (cammino1, cammino2, meta)
                                then -1
                                else 1;;
```

Ricerca best-first: ad ogni passo viene espansa la soluzione parziale più promettente

```
exception NotFound;;

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

let searchbf inizio fine (Graph 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 confronta c1 c2 =
            confrontacammino c1 c2 fine
      in let rec search_aux fine = function
            [] -> raise NotFound
            | cammino::rest -> if fine = List.hd cammino
                  then List.rev cammino
                  else search_aux fine (List.sort
                                    confronta
                                    (rest @ (estendi cammino)))
in search_aux fine [[inizio]];;
```

```
# searchbf 1 7 g;;
1;
2; 1;
3; 2; 1;
4; 3; 2; 1;
6; 3; 2; 1;
- : int list = [1; 2; 3; 6; 7]
```

Per migliorare la fase, costosa, di ordinamento dei cammini si può ordinare l'espansione del cammino corrente e poi fare l'unione ordinata di liste ordinate fra l'espansione e la vecchia lista dei cammini

Ricerca hill climbing: ad ogni passo viene espansa la soluzione parziale generata al passo precedente più promettente

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

let searchhc inizio fine (Graph 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 confronta c1 c2 =
                confrontacammino c1 c2 fine
        in let rec search_aux fine = function
                [] -> raise NotFound
                | cammino::rest -> if fine = List.hd cammino
                        then List.rev cammino
                        else search_aux fine ((List.sort confronta (estendi cammino))
                                                @rest)
in search_aux fine [[inizio]];;
```

```
# searchhc 1 7 g;;
1;
2; 1;
3; 2; 1;
4; 3; 2; 1;
6; 3; 2; 1;
- : int list = [1; 2; 3; 6; 7]
```

Il vantaggio rispetto alla ricerca hill-climbing è che ordinare solo l'estensione del cammino attuale è computazionalmente meno costoso che ordinare l'intera lista dei cammini.
Lo svantaggio è il rischio di tendere verso un minimo locale della funzione di valutazione.

Ricerca della soluzione di costo minimo: branch and bound

Quando è definita una funzione costo per le soluzioni parziali, uno schema simile al best first consente di calcolare la soluzione di costo minimo. Ad ogni passo l'insieme delle soluzioni parziali viene ordinata rispetto al costo.

```
let rec costocammino = function
        [] -> 0.
        | [x] -> 0.;
        | x::y::rest -> (distanza x y) +. (costocammino (y::rest));;

exception NotFound;;

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

```
let searchbb inizio fine (Graph 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 confronta c1 c2 =
        let costo1 = costocammino c1
        in let costo2 = costocammino c2
        in if costo1 = costo2 then 0
            else if costo1 < costo2 then -1
                                    else 1
    in let rec search_aux fine = function
        [] -> raise NotFound
        | cammino::rest -> if fine = List.hd cammino
            then List.rev cammino
            else search_aux fine (List.sort
                                confronta
                                (rest @ (estendi cammino)))
in search_aux fine [[inizio]];;
```

```
# searchbb 1 7 g;;
1;
5; 1;
2; 1;
6; 5; 1;
3; 2; 1;
2; 5; 1;
5; 2; 1;
4; 3; 2; 1;
    -   : int list = [1; 5; 6; 7]
```

Ricerca A*: costo del cammino+previsione costo verso la meta

```
let distanza nodo1 nodo2 =
        let x1 = float (fst(List.assoc nodo1 coordinate))
        in let y1 = float (snd(List.assoc nodo1 coordinate))
        in let x2 = float (fst(List.assoc nodo2 coordinate))
        in let y2 = float (snd(List.assoc nodo2 coordinate))
        in sqrt ( (x1 -. x2)**2. +. (y1 -. y2)**2.);;

let rec costocammino = function
        [] -> 0.
        | [x] -> 0.;
        | x::y::rest -> (distanza x y) +. (costocammino (y::rest));;

exception NotFound;;

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


 (* Distanza ammissibile se non sovrastima la distanza dalla meta *)
```

```
let costototale cammino meta = costocammino cammino+. distanza (List.hd cammino) meta;;

let confrontacammino cammino1 cammino2 meta=
        let c1 = costototale cammino1 meta in
                let c2 = costototale cammino2 meta in
                        if c1 = c2 then 0
                                else if c1 < c2 then -1
                                                        else 1;;
```

```
let searchAstar inizio fine (Graph 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 confronta c1 c2 =
                confrontacammino c1 c2 fine
        in let rec search_aux fine = function
                [] -> raise NotFound
                | cammino::rest -> if fine = List.hd cammino
                        then List.rev cammino
                        else search_aux fine (List.sort
                                                confronta
                                                (rest @ (estendi cammino)))
in search_aux fine [[inizio]];;
```

```
# searchAstar 1 7 g;;
1;
2; 1;
5; 1;
3; 2; 1;
6; 5; 1;
- : int list = [1; 5; 6; 7]
```