Grafi

Anno Accademico 2022/2023

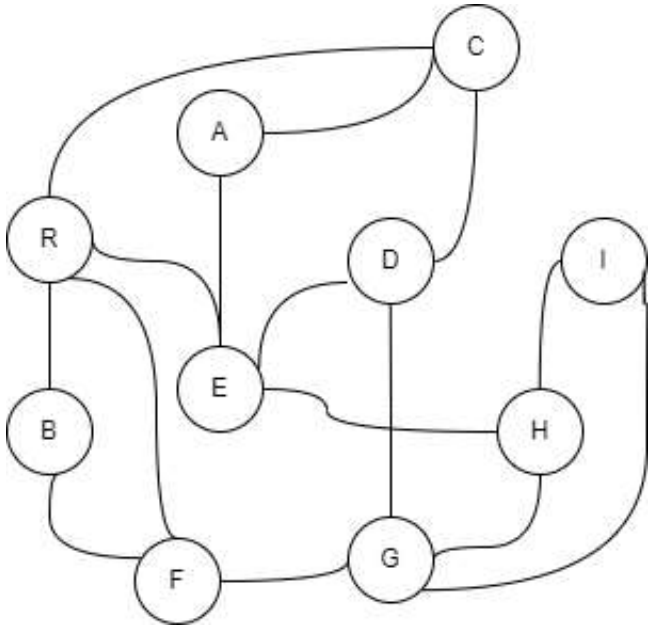
Dott. Staccone Simone

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

BFS (Breadth-First Search)

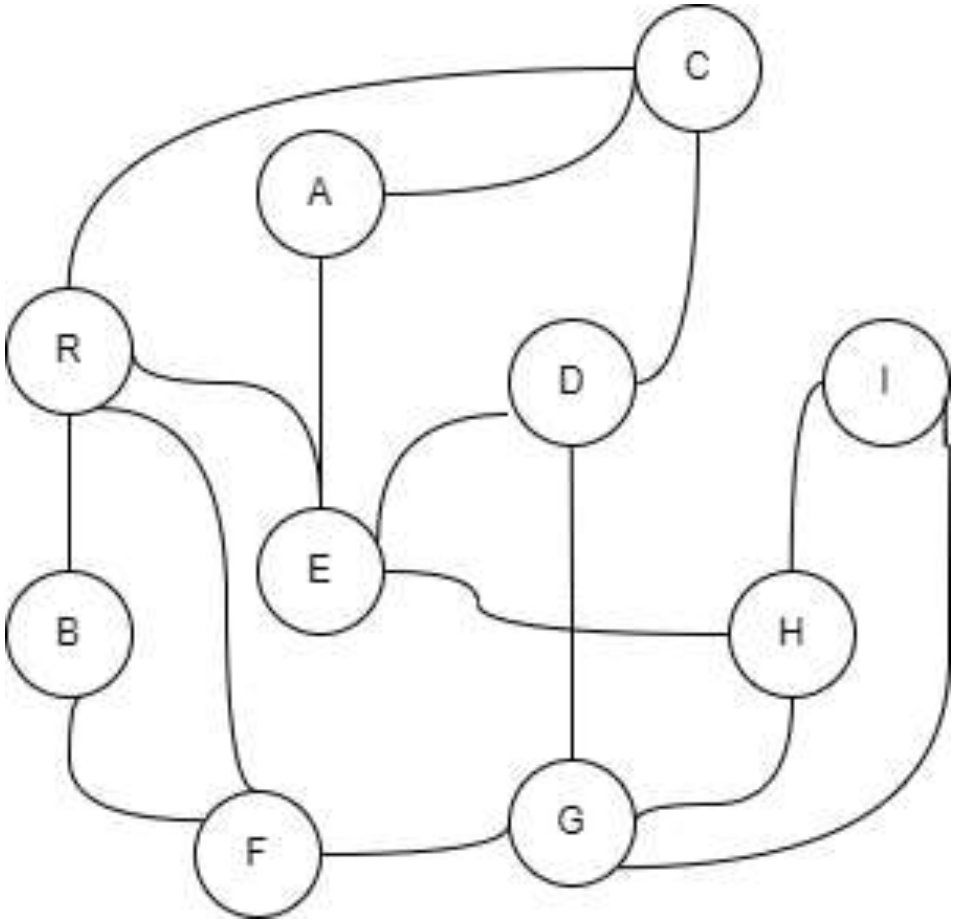
Algorithm 2: BFS(Grafo G, nodo r)

```
QUEUE S ← Queue();
S.enqueue(r);
boolean[] visitato ← new boolean[1 ... G.n];
visitato[r] ← true;
while not S.isEmpty() do
    NODE u ← S.dequeue();
    esamina il nodo u;
    foreach v ∈ G.adj(u) do
        Esamina l'arco (u, v);
        if not visitato[v] then
            visitato[v] ← true;
            S.enqueue(v);
```



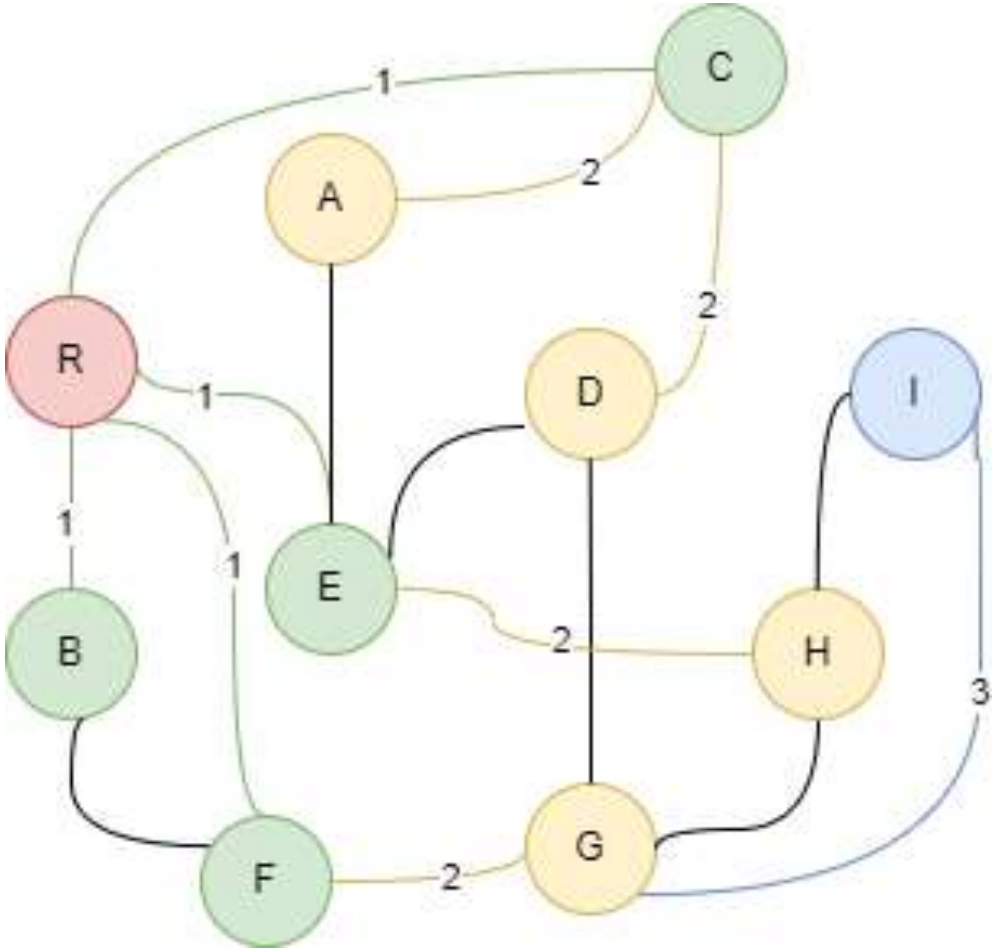
BFS (Breadth-First Search)

Dato il seguente grafo G, trovare la distanza di tutti i nodi dal nodo root R, utilizzando una visita del grafo BFS, mostrando l'albero successivamente generato,



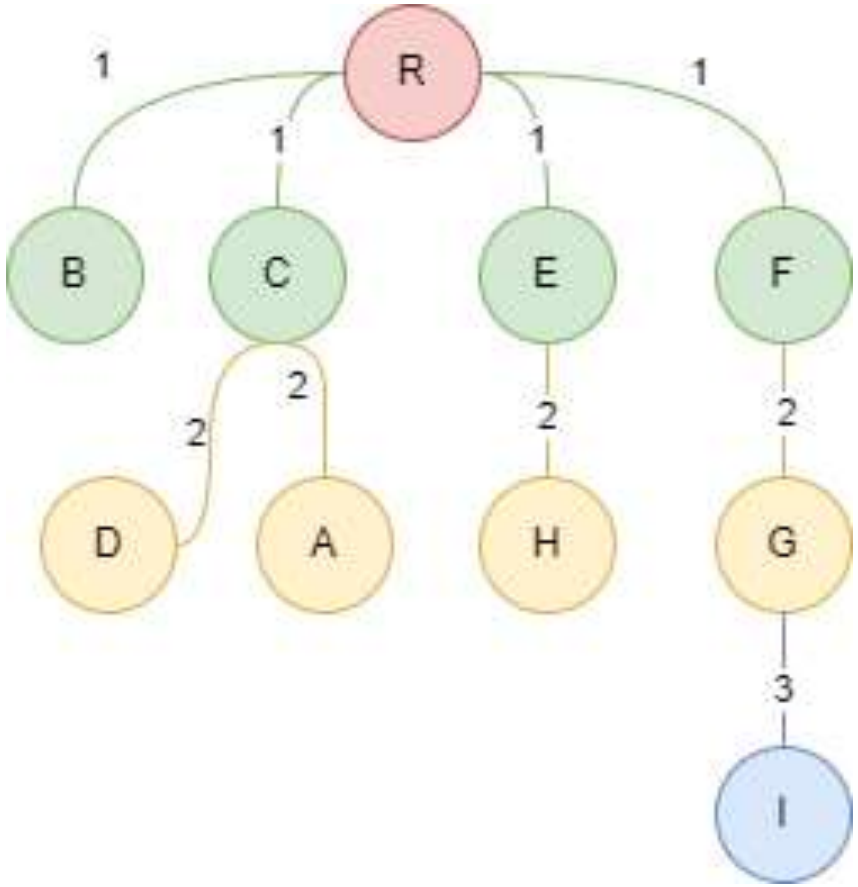
BFS (Breadth-First Search)

Dato il seguente grafo G, trovare la distanza di tutti i nodi dal nodo root R, utilizzando una visita del grafo BFS, mostrando l'albero successivamente generato,



BFS (Breadth-First Search)

Dato il seguente grafo G, trovare la distanza di tutti i nodi dal nodo root R, utilizzando una visita del grafo BFS, mostrando l'albero successivamente generato,



DFS (Dept-First Search) - ricorsiva

Algorithm 1: recursiveCall(vertice v, albero T)

```
v.marked == true;
foreach edge(v,w) do
    if w.marked == false then
        add (v,w) to tree T
        recursiveCall(w,T)
    end
end
```

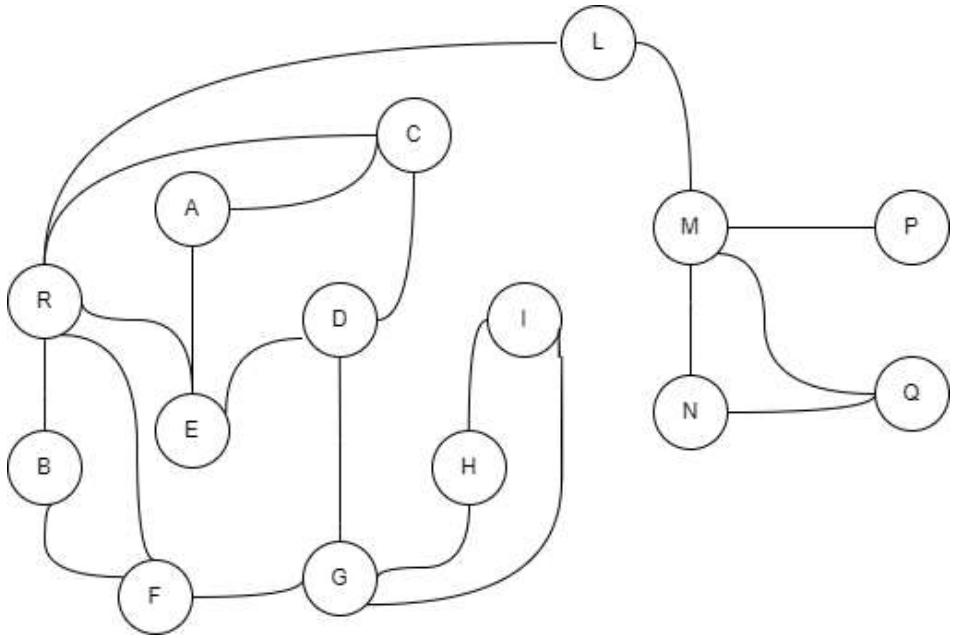
Algorithm 2: DFS(vertice v) : albero T

```
T ← newTree()
recursiveCall(v,T)
return T
```

DFS (Dept-First Search) - iterativa

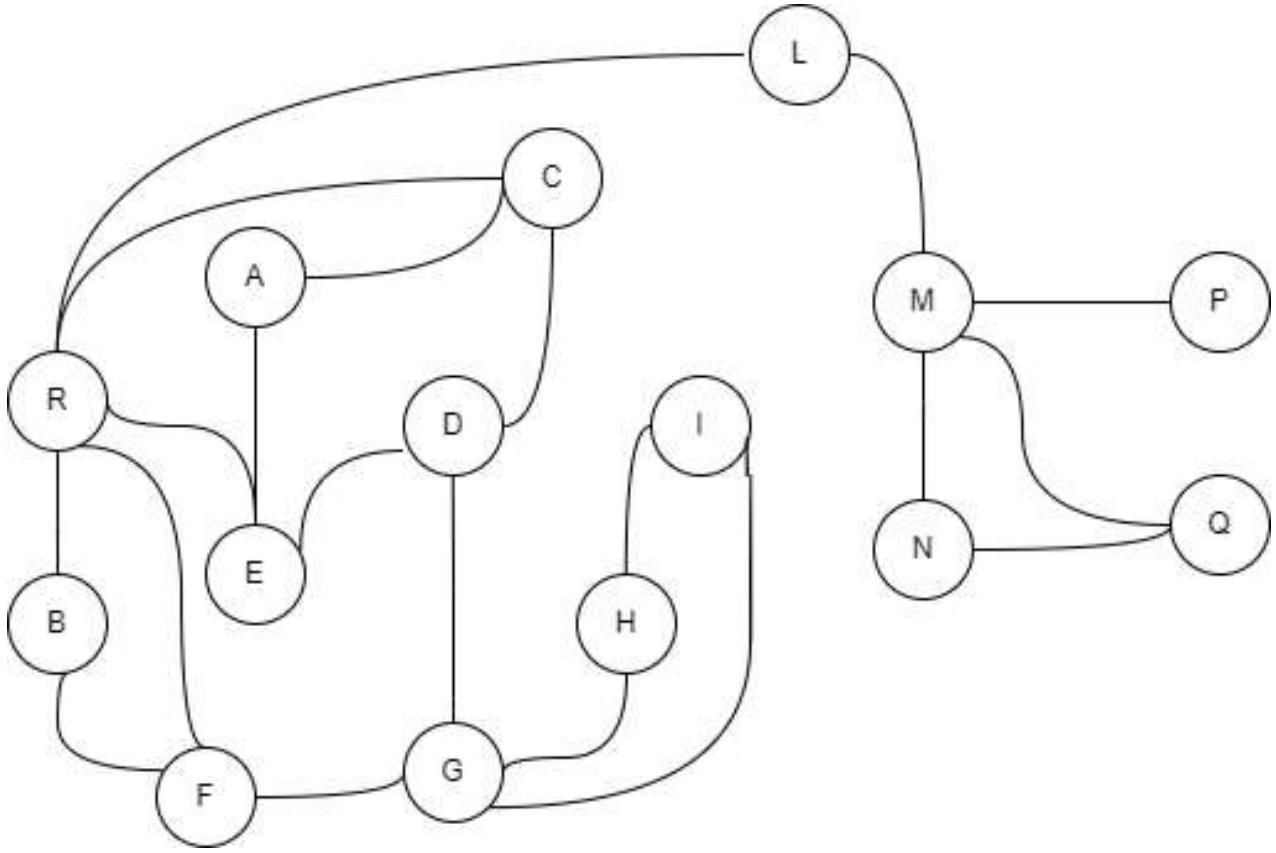
Algorithm 1: dfs(Graph G, Node n)

```
StackS ← Stack()
S.push(r)
foreach u ∈ G.vetrices() do
    u.discovered ← false
end
while !S.isEmpty() do
    Node u ← S.pop()
    if !u.discovered then
        visit node u
        u.discovered ← true
        foreach v in G.adj(u) do
            visit edge (u,v)
            S.push(v)
        end
    end
end
```



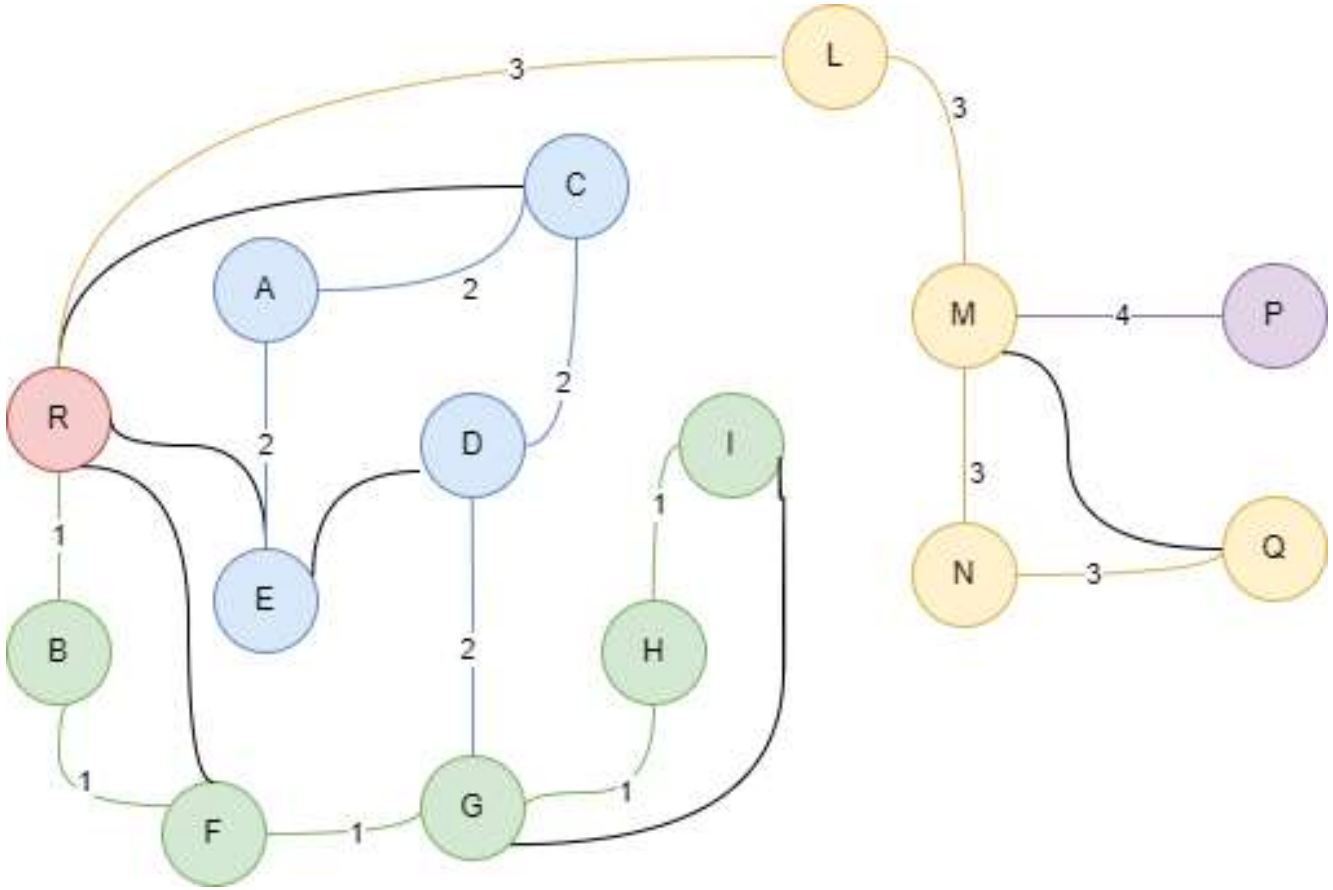
DFS (Dept-First Search)

Dato il seguente grafo G, trovare la distanza di tutti i nodi dal nodo root R, utilizzando una visita del grafo DFS, mostrando l'albero successivamente generato,

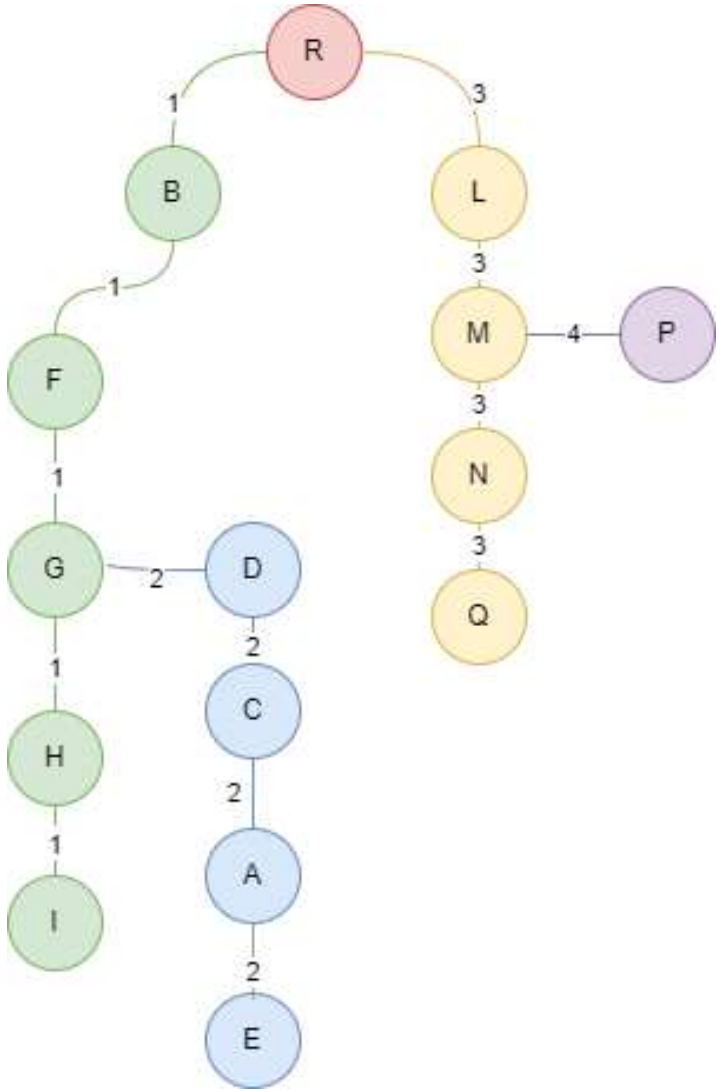


DFS (Dept-First Search)

Dato il seguente grafo G, trovare la distanza di tutti i nodi dal nodo root R, utilizzando una visita del grafo DFS, mostrando l'albero successivamente generato,



DFS (Dept-First Search)

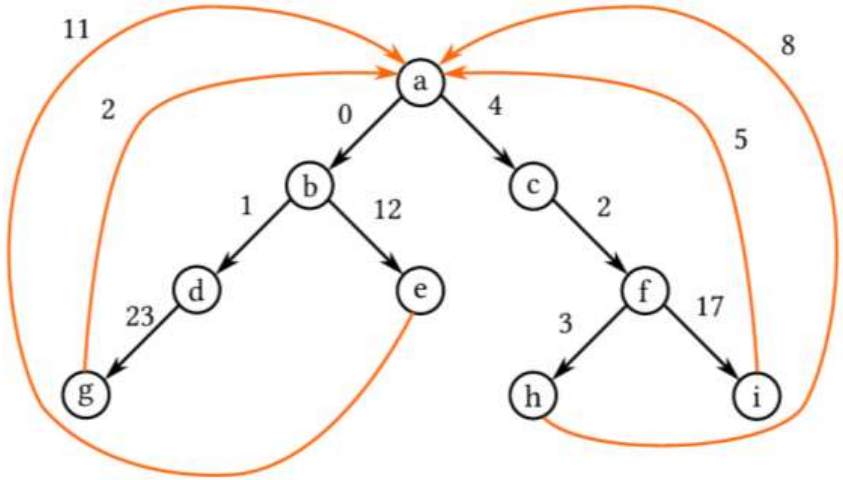


Esercizio

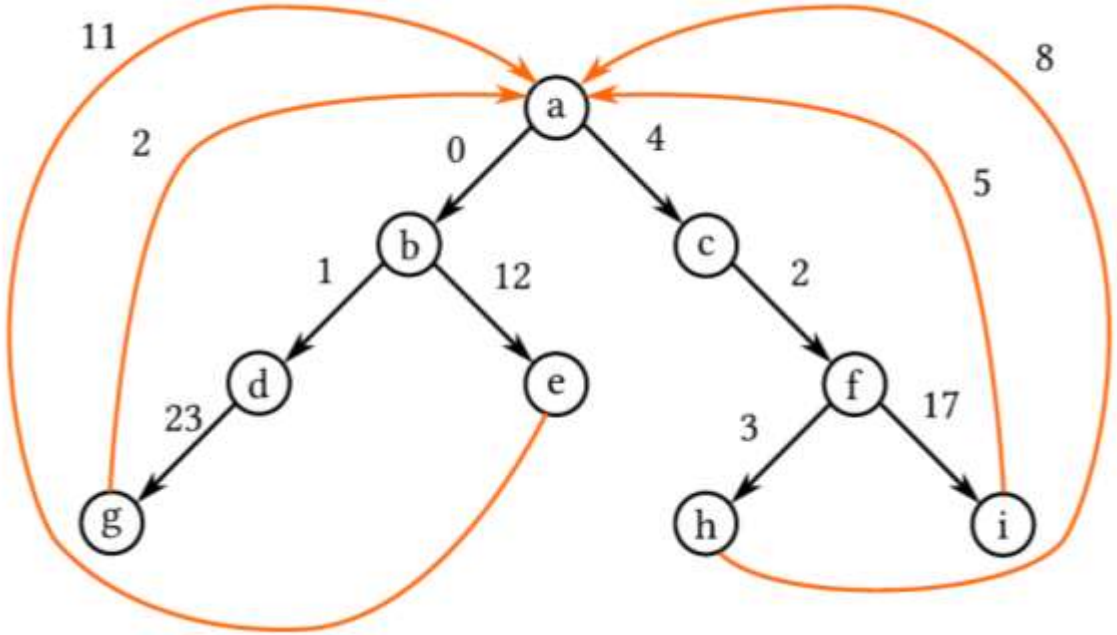
Un grafo è un “albero binario ciclato” se deriva da un albero binario secondo le seguenti regole di costruzione:

• è presente un nodo del grafo per ogni nodo dell’albero;
• è presente un arco per ogni coppia (genitore, figlio), orientato dal padre al figlio;
• è presente un arco “di ritorno” da ogni foglia alla radice dell’albero binario, chiudendo così un ciclo.
• gli archi di questo grafo sono pesati da una funzione w : E ! N.

Un esempio di albero binario ciclato è il seguente (in arancione sono indicati gli archi di ritorno):



Esercizio



Dati in ingresso un albero binario ciclato G = (V;E) con n >= 2 nodi, il nodo r radice, la funzione di pesi w e due nodi u, v, scrivere un algoritmo computeDistance(G, r, u, v) che restituisca il peso del cammino minimo da u a v. La valutazione terrà conto anche del costo computazionale
della soluzione proposta, tenuto conto che è possibile risolvere il problema in O(n).

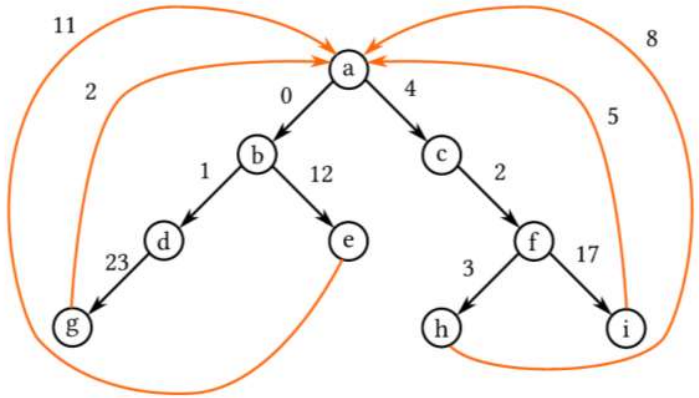
Esercizio - Soluzione

L’esercizio si risolve utilizzando una semplice visita in profondità per calcolare la distanza tra due nodi. Tuttavia, occorre fare attenzione a non seguire più volte il percorso a partire dalla radice (venendo da una foglia), poiché in questo caso il costo che si otterrebbe sarebbe $O(n^2)$. Si può quindi scrivere una variante della visita DFS che, a partire da un nodo x, cerchi un nodo t, senza mai passare per il nodo r—se si incontra il nodo r, la procedura non prosegue la visita. Se questo cammino esiste, si può restituire la distanza minima calcolata, viceversa si restituisce 1. Nel caso in cui il nodo v passato come input all’algoritmo sia in uno dei sottoalberi di u, l’algoritmo descritto può immediatamente calcolare la distanza minima, evitando di passare per r, poiché esiste un solo cammino da u a v. La procedura descritta ha costo $O(n + m)$ (dove m è il numero di archi attraversati), poiché la procedura si arresta sulle foglie. In caso contrario, la versione modificata della visita DFS restituisce 1. Il percorso per raggiungere v da u si comporrà quindi di due parti: il percorso (unico) a costo minimo per andare dalla radice r al nodo v (senza mai passare per r), ed il percorso a costo minimo per andare da u alla radice (senza vincoli di nodi da cui non si può passare). Queste due visite hanno ancora costo $O(m + n)$. Poiché ogni nodo ha al massimo due archi uscenti (per costruzione del grafo a partire da un albero binario di ricerca), si ha che $m < 2n$, dando quindi un costo complessivo dell’algoritmo di $O(n)$.

Esercizio - Soluzione

Algorithm 1: dfsSearch(Graph G, Node x, Node t, Node r)

```
if x == t then
    return 0
end
min ← inf
foreach y ∈ G.adj(x) do
    if y != r then
        d ← dfsSearch(G, y, t, r)
        if d + G.weight(x, y) < min then
            min ← d + G.weight(x, y)
        end
    end
end
return min
```

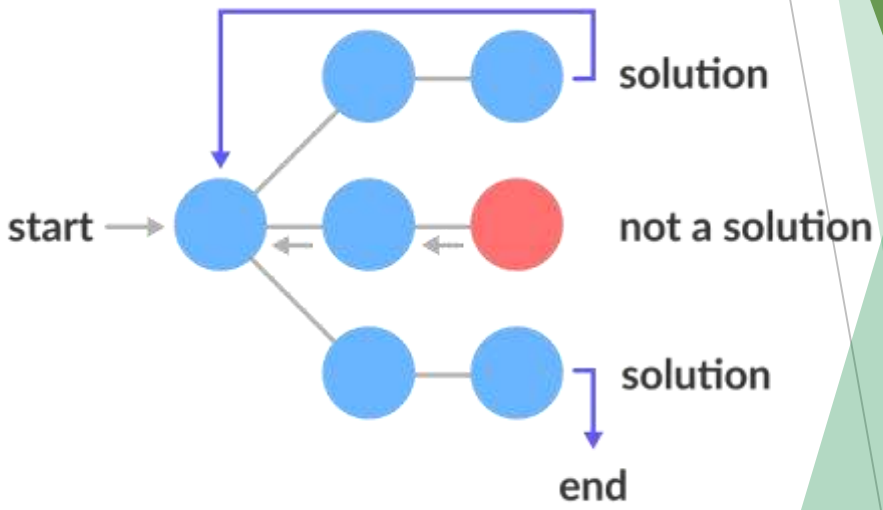


Algorithm 2: computeDistance(Graph G, Node r, Node u, Node v)

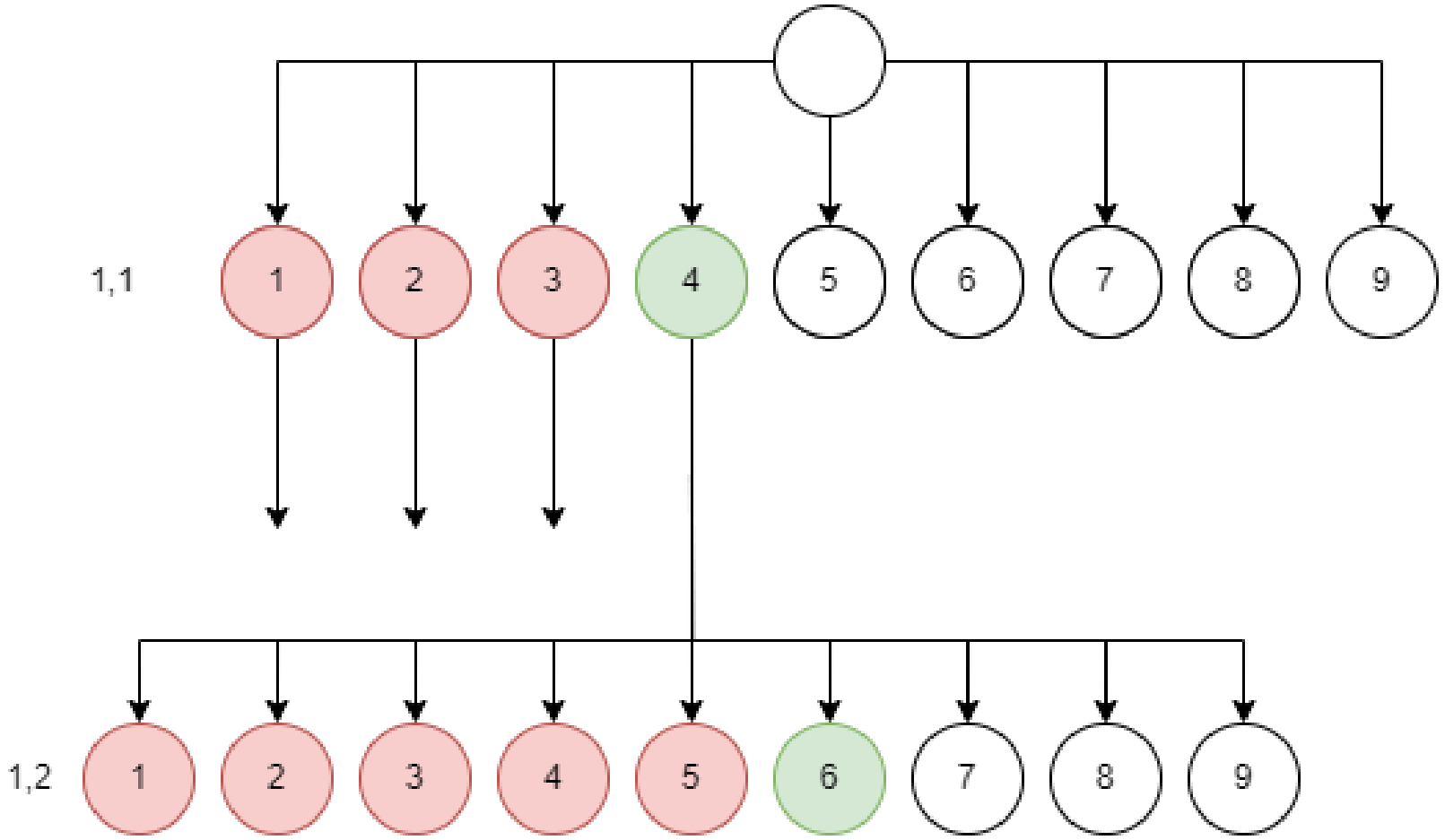
```
d ← dfsSearch(G, u, v, r)
if d < inf then
    return d
end
else
    return dfsSearch(G,u,r,-1) + dfsSearch(G,r,v,r)
end
```

Backtracking

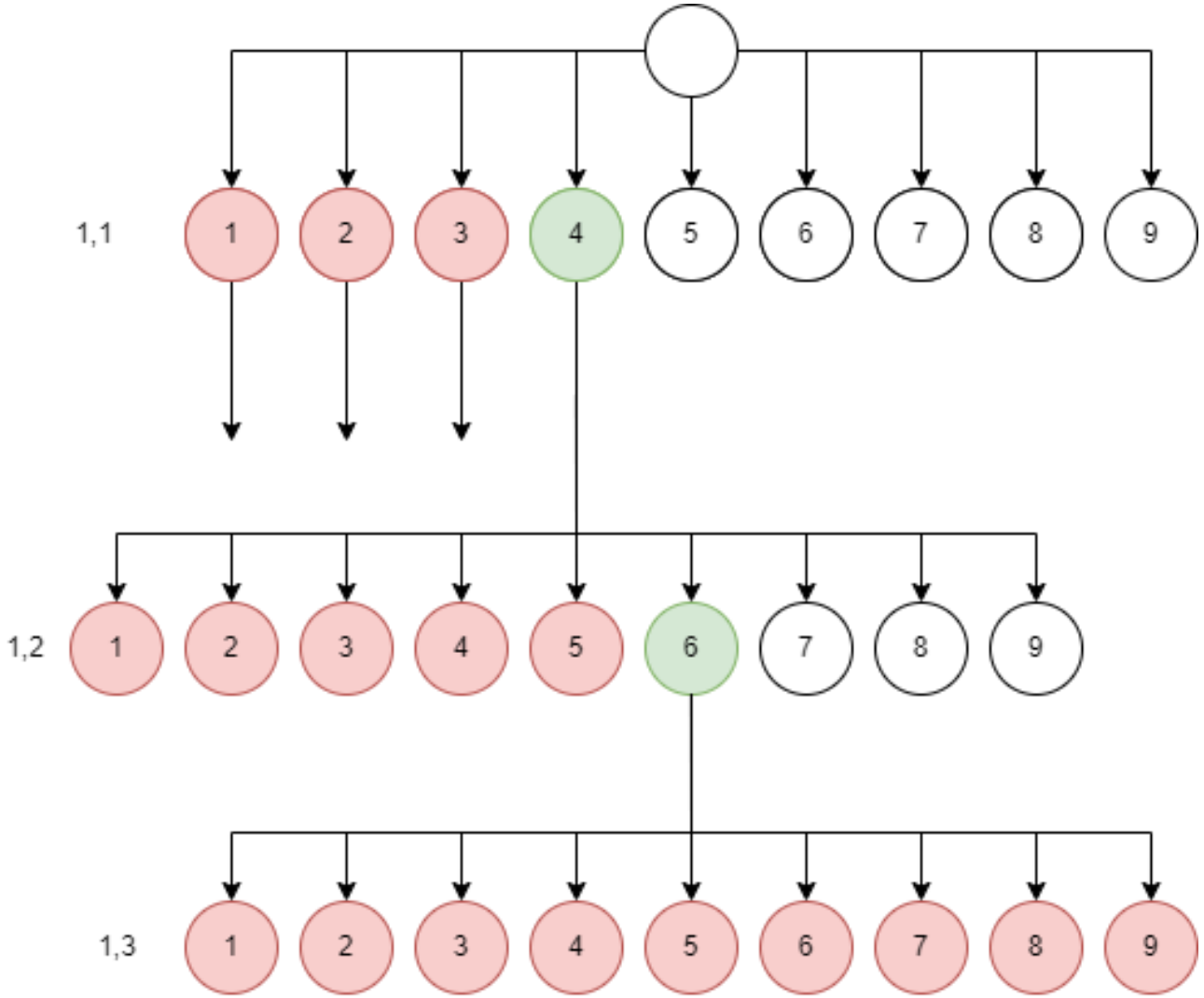
La tecnica di backtracking estende la ricerca esaustiva nella risoluzione di problemi di ricerca attraverso l'introduzione di alcuni controlli per verificare il più presto possibile se una soluzione in via di costruzione soddisfi o meno le condizioni di ammissibilità in modo da ridurre lo spazio di ricerca. Il problema tipico di ricerca risolto con il backtracking consiste nell'assegnare un valore, preso da un dominio V, a ciascuno degli n elementi di un vettore soluzione X in modo che siano soddisfatti alcuni vincoli su tali valori. Mentre la ricerca esaustiva ad ogni passo genera una soluzione possibile effettuando un possibile assegnamento di valori a tutti gli elementi di X e poi verifica i vincoli per essa, il backtracking assegna un valore ad un elemento la volta e verifica i vincoli immediatamente senza aspettare di aver assegnato un valore a tutti gli elementi. In questo modo può essere ridotto enormemente il numero di soluzioni ammissibili generate.



Sudoku - Backtracking



Sudoku - Backtracking



Sudoku - Backtracking