

# Progettazione di Algoritmi - lezione 7

## Discussione dell'esercizio [broadcast]

Rappresentiamo il problema tramite un grafo diretto  $G$  in cui i nodi sono le stazioni e  $c'$  è un arco dal nodo  $u$  a  $v$  se la stazione  $u$  può trasmettere direttamente alla stazione  $v$ . Il problema chiede di trovare un algoritmo che ritorna, se esistono, i *nodi broadcast*, cioè nodi da cui si possono raggiungere tutti i nodi del grafo.

Un algoritmo molto semplice consiste nel controllare nodo per nodo se è un nodo broadcast. Per controllare se  $u$  è un nodo broadcast basta fare una visita a partire da  $u$  e verificare che siano visitati tutti i nodi del grafo. Però è piuttosto inefficiente perchè richiede  $n$  visite con una complessità totale di  $O(n(n + m))$ .

Osserviamo che se  $u$  e  $v$  sono due nodi broadcast, devono appartenere alla stessa componente fortemente connessa perchè ci sono cammini da  $u$  a  $v$  e da  $v$  a  $u$ . Inoltre, un nodo  $w$  che appartiene alla componente in cui  $c'$  è un nodo broadcast  $u$  è anch'esso un nodo broadcast dato che un qualsiasi nodo  $z$  essendo raggiungibile da  $u$  è anche raggiungibile da  $w$  perchè  $c'$  è un cammino da  $w$  a  $u$ . Quindi se nel grafo ci sono nodi broadcast, allora  $c'$  è una componente fortemente connessa che è formata dall'insieme di tutti i nodi broadcast. Inoltre, se ci sono anche altre componenti, ci sono cammini che vanno dai nodi della componente broadcast verso tutte le altre componenti. Questo implica che in un qualsiasi ordinamento topologico del DAG delle componenti, la componente broadcast (se esiste) deve necessariamente essere la prima componente.

Queste osservazioni ci portano al seguente algoritmo. Eseguiamo l'algoritmo per determinare le componenti fortemente connesse. L'algoritmo di Tarjan numera le componenti secondo un ordinamento topologico (in ordine di indice decrescente). Così la componente  $B$  con indice più alto è la prima componente dell'ordinamento. Scegliamo un nodo  $u$  da  $B$  e verifichiamo tramite una visita se tutti i nodi sono raggiungibili da  $u$ . In caso affermativo,  $B$  è la componente broadcast, altrimenti non ci sono nodi broadcast nel grafo.

Ecco quindi lo pseudo-codice dell'algoritmo che ritorna in una lista tutti i nodi broadcast (se non ci sono nodi broadcast, ritorna la lista vuota):

```
DFS_BROAD(G: grafo diretto)
  CC <- SCC(G)          /* Calcola l'array delle componenti con l'algoritmo di Tarjan */
  nc <- 0
  u <- 0
  FOR ogni nodo v DO   /* Cerca la prima componente, cioè quella di indice più alto */
    IF CC[v] > nc THEN
      nc = CC[v]
      u <- v
  vis <- DFS(G, u)     /* Esegue la DFS da u e ritorna il numero di nodi visitati */
  B <- lista vuota
  IF vis = n THEN     /* Se da u si raggiungono tutti i nodi, */
    FOR ogni nodo v DO /* salva in B i nodi della componente */
      IF CC[v] = nc THEN
        B.append(v)
  RETURN B
```

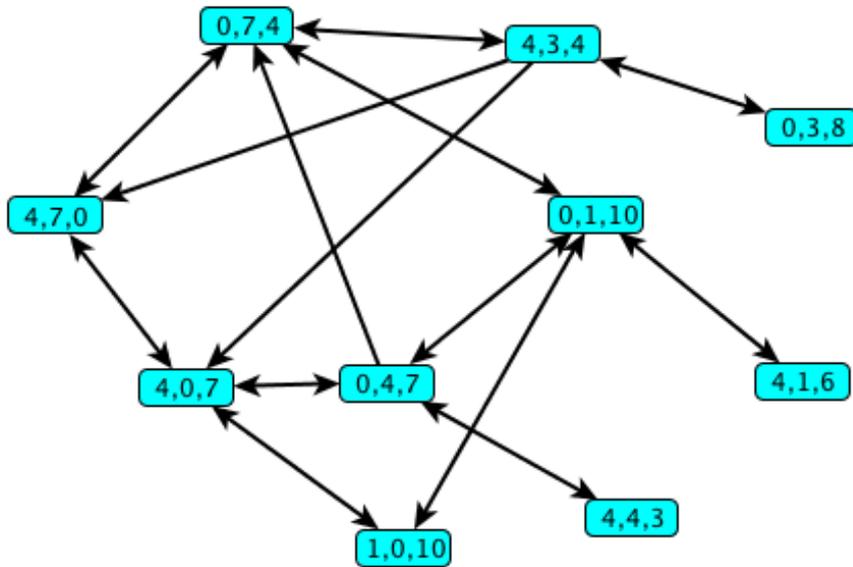
Siccome la complessità dell'algoritmo di Tarjan è lineare come quella della visita DFS, la complessità dell'algoritmo è  $O(n + m)$ .

## Il problema delle distanze

Nelle prime lezioni abbiamo visto alcuni problemi come il 9-puzzle

<table style="border-collapse: collapse; width: 100%;"> <tr><td style="border: 1px solid black; padding: 2px;">  7   3   9  </td></tr> <tr><td style="border: 1px solid black; padding: 2px;"> --- --- --- </td></tr> <tr><td style="border: 1px solid black; padding: 2px;">  2   5   1  </td></tr> <tr><td style="border: 1px solid black; padding: 2px;"> --- --- --- </td></tr> <tr><td style="border: 1px solid black; padding: 2px;">  4   8   6  </td></tr> <tr><td style="border: 1px solid black; padding: 2px;"> --- --- --- </td></tr> </table>	7   3   9	--- --- ---	2   5   1	--- --- ---	4   8   6	--- --- ---	----> . . . . . ---->	<table style="border-collapse: collapse; width: 100%;"> <tr><td style="border: 1px solid black; padding: 2px;">  1   2   3  </td></tr> <tr><td style="border: 1px solid black; padding: 2px;"> --- --- --- </td></tr> <tr><td style="border: 1px solid black; padding: 2px;">  4   5   6  </td></tr> <tr><td style="border: 1px solid black; padding: 2px;"> --- --- --- </td></tr> <tr><td style="border: 1px solid black; padding: 2px;">  7   8   9  </td></tr> <tr><td style="border: 1px solid black; padding: 2px;"> --- --- --- </td></tr> </table>	1   2   3	--- --- ---	4   5   6	--- --- ---	7   8   9	--- --- ---
7   3   9														
--- --- ---														
2   5   1														
--- --- ---														
4   8   6														
--- --- ---														
1   2   3														
--- --- ---														
4   5   6														
--- --- ---														
7   8   9														
--- --- ---														

o il problema dei tre contenitori d'acqua



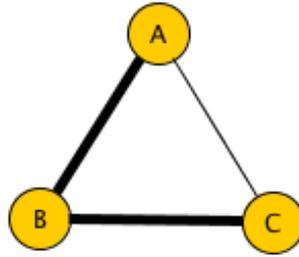
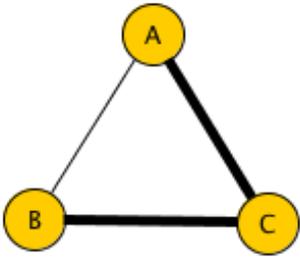
in cui ci si chiedeva se da una data configurazione fosse possibile raggiungere, effettuando mosse ammissibili, delle configurazioni target. Per risolvere tali problemi è stato sufficiente fare una visita a partire dalla data configurazione nel grafo di tutte le configurazioni connesse dalle mosse ammissibili e controllare se si raggiunge una delle configurazioni target. Ma se non ci accontentiamo di conoscere che una configurazione target è raggiungibile e vogliamo trovare il numero *minimo* di mosse per raggiungerla? Lo stesso problema si presenta, ad esempio, se vogliamo trovare il percorso più breve (inteso come minimo numero di bivi che devono essere attraversati) per uscire da un labirinto.

In termini generali, il problema è il seguente: dato un grafo (diretto o non diretto) e dati due nodi  $u$  e  $v$  si vuole trovare un *cammino di lunghezza minima* da  $u$  a  $v$ , se ne esiste almeno uno. Per *lunghezza di un cammino* (orientato o non orientato) si intende il numero di archi del cammino. Spesso la lunghezza minima di un cammino da  $u$  a  $v$  è detta *distanza* da  $u$  a  $v$  ed è denotata  $d(u, v)$ . Se non ci sono cammini da  $u$  a  $v$ , si conviene che  $d(u, v) = \infty$ . Ovviamente, risulta sempre  $d(u, u) = 0$ . Se il grafo non è diretto le distanze sono simmetriche, cioè, per qualsiasi  $u$  e  $v$ ,  $d(u, v) = d(v, u)$ . Invece, per grafi diretti le distanze non sono, in generale, simmetriche. Una proprietà che vale per ogni tipo di grafo è la *disuguaglianza triangolare*:

$$\text{per qualsiasi } u, v \text{ e } w, d(u, v) \leq d(u, w) + d(w, v)$$

La validità di questa proprietà deriva semplicemente dal fatto che possiamo sempre concatenare un cammino da  $u$  a  $w$  con uno da  $w$  a  $v$  e ottenere un cammino da  $u$  a  $v$ .

Per determinare le distanze tra i nodi di un grafo la visita DFS non è adatta perchè i cammini che trova non sono in generale di lunghezza minima. Ad esempio, nel grafo qui sotto sono marcati i due possibili alberi della DFS a partire dal nodo A.



In entrambi i casi, per uno dei due nodi  $B$  o  $C$  il cammino trovato non è di lunghezza minima. Per determinare le distanze occorre un'altra procedura di visita.

## Visita in ampiezza (BFS)

Per visitare, a partire da un nodo  $u$ , i nodi di un grafo, trovando i cammini di lunghezza minima da  $u$ , dovremmo prima di tutto visitare i nodi a distanza 1 da  $u$ , che sono tutti gli adiacenti di  $u$ . Avendo determinato tutti i nodi a distanza 1, i nodi a distanza 2 sono quelli (diversi dai primi) che sono adiacenti ai nodi a distanza 1. In generale, avendo determinati i nodi a distanza  $k$ , quelli a distanza  $k + 1$  sono i nodi (diversi da tutti quelli già classificati) che sono adiacenti ai nodi a distanza  $k$ . Per effettuare questo tipo di visita manteniamo in una coda i nodi visitati i cui adiacenti non sono stati ancora esaminati. Ad ogni passo, preleviamo il primo nodo della coda, esaminiamo i suoi adiacenti e se scopriamo un nuovo nodo lo aggiungiamo alla coda. Grazie all'uso della coda siamo sicuri che considereremo tutti gli adiacenti dei nodi a distanza  $k$  prima di considerare gli adiacenti dei nodi a distanza  $k + 1$ . Questo tipo di visita si chiama **visita in ampiezza** (in inglese *Breadth First Search*, abbreviato *BFS*).

Ecco un'implementazione della BFS che ritorna sia l'array delle distanze che l'albero di visita rappresentato tramite vettore dei padri. Assumiamo per semplicità che tutti i nodi del grafo siano raggiungibili da  $u$ .

```
BFS(G: grafo, u: nodo)
  P: vettore dei padri, inizializzato a 0
  Dist: array delle distanze
  P[u] <- u          /* u è la radice dell'albero della BFS */
  Dist[u] <- 0
  Q <- coda vuota
  Q.enqueue(u)      /* Accoda u alla coda */
  WHILE Q non è vuota DO
    v <- Q.dequeue() /* Preleva il primo nodo della coda */
    FOR ogni adiacente w di v DO
      IF P[w] = 0 THEN
        P[w] <- v
        Dist[w] <- Dist[v] + 1
        Q.enqueue(w) /* Accoda w alla coda */
  RETURN P, Dist
```

Il vettore dei padri rappresenta l'albero di visita della BFS e quindi per ogni nodo  $v$  contiene un cammino di lunghezza minima da  $u$  a  $v$ . Per ricostruire un tale cammino basta partire da  $v$  e percorrerlo di padre in padre tramite  $P$  fino alla radice  $u$ :

```
L <- lista vuota
L.head(v) /* Aggiunge v in testa alla lista */
WHILE v <> u DO
  v <- P[v]
  L.head(v)
RETURN L /* Ritorna in L il cammino da u a v */
```

Per quanto riguarda la complessità della BFS facciamo le seguenti osservazioni. Le inizializzazioni degli array  $P$  e  $Dist$  costano  $O(n)$ . La coda  $Q$  può essere implementata in modo tale che le operazioni su di essa hanno costo costante. Il `WHILE` esegue esattamente  $n$  iterazioni perchè ogni nodo viene aggiunto alla coda  $Q$  una sola volta e ad ogni iterazione

un nodo è estratto da  $Q$ . In ogni iterazione del WHILE il FOR interno fa un numero di iterazioni pari al numero degli adiacenti del nodo estratto in quella iterazione del WHILE. Perciò il numero complessivo di iterazioni del FOR è la somma dei gradi (per un grafo diretto, i gradi uscenti) dei nodi e quindi  $O(m)$ . In totale, la complessità è dunque  $O(n + m)$ .

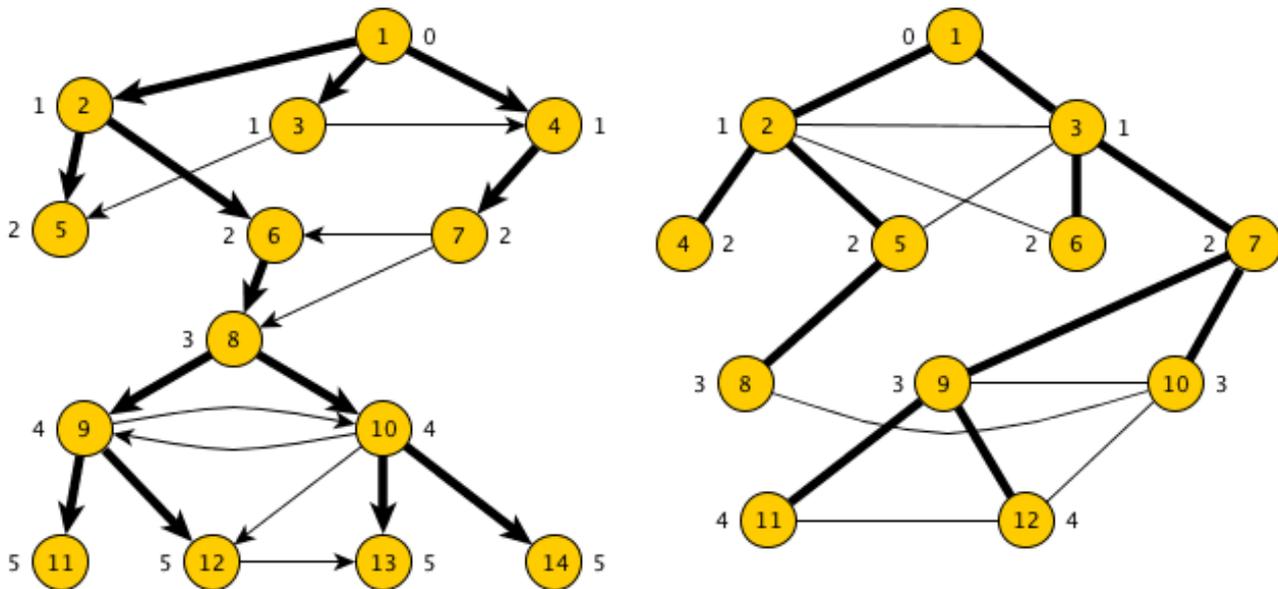
La dimostrazione della correttezza della BFS può essere fatta per induzione dimostrando che l'algoritmo della BFS partendo da un nodo  $u$  soddisfa il seguente enunciato:

Per ogni  $k = 0, 1, 2, \dots$  c'è un passo dell'algoritmo della BFS in cui

- ogni nodo  $v$  con  $d(u, v) \leq k$  è stato visitato,  $\text{Dist}[v] = d(u, v)$  e nessun'altro nodo è stato visitato;
- la coda  $Q$  contiene esattamente i nodi a distanza  $k$  da  $u$ .

La dimostrazione è per induzione su  $k$ . Per  $k = 0$ , prima di iniziare il WHILE le proprietà (a) e (b) sono soddisfatte perchè  $u$ , l'unico nodo a distanza 0, ha  $\text{Dist}[u] = 0$  ed è l'unico nodo nella coda. Supponiamo, per ipotesi induttiva, che ci sia un passo  $t$  per cui le proprietà (a) e (b) valgono per  $k$  e dimostriamole per  $k + 1$ . Se l'insieme dei nodi a distanza  $k$  è vuoto, le proprietà (a) e (b) valgono perchè la coda è vuota e anche l'insieme dei nodi a distanza  $k + 1$  è vuoto. I nodi a distanza  $k + 1$ , se ci sono, sono esattamente quelli adiacenti a nodi a distanza  $k$  che non sono a distanza  $\leq k$ . Per ipotesi induttiva nessuno di questi è stato visitato fino al passo  $t$ . Ogni volta che un nodo  $v$  a distanza  $k$  è estratto dalla coda ogni adiacente  $w$  di  $v$  è esaminato e se non è già stato visitato, cioè  $P[w] = 0$ , e quindi, per l'ipotesi induttiva, la sua distanza non è  $\leq k$ , la distanza  $\text{Dist}[w]$  è impostata a  $k + 1$  (cioè,  $d(u, w)$ ) ed è aggiunto alla coda. Ci sarà un passo in cui tutti i nodi a distanza  $k$  sono stati estratti dalla coda e i loro adiacenti sono stati esaminati. In quel passo, ogni nodo con  $d(u, v) \leq k + 1$  è stato visitato,  $\text{Dist}[v] = d(u, v)$ , nessun'altro nodo è stato visitato e la coda  $Q$  contiene esattamente i nodi a distanza  $k + 1$  da  $u$ . Quindi, le proprietà (a) e (b) valgono per  $k + 1$ .

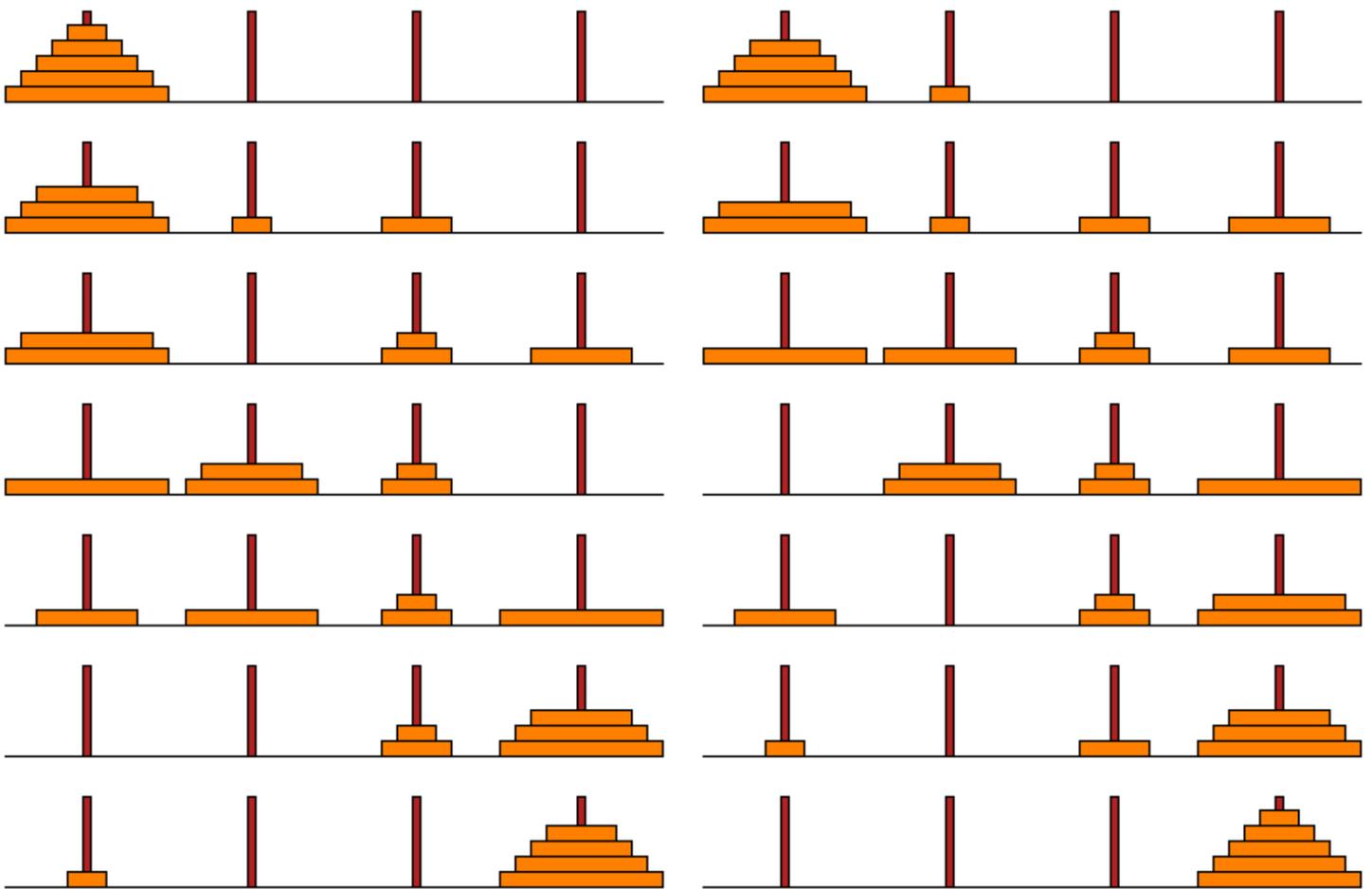
Qui sotto ci sono due esempi di visite BFS, una su un grafo diretto e una su un grafo non diretto. I nodi sono numerati in ordine di visita, gli archi dell'albero (o arborescenza) della BFS sono marcati e accanto ad ogni nodo c'è la distanza dal nodo di partenza (il nodo 1).



## Esempio [reve's puzzle]

Il famoso rompicapo della *torre di hanoi* consiste di 3 paletti e  $n$  dischi di grandezza decrescente, che possono essere infilati in uno qualsiasi dei paletti. Inizialmente gli  $n$  dischi sono incolonnati su un paletto in ordine decrescente. Lo scopo è portare tutti i dischi su un altro paletto, potendo spostare solo un disco alla volta e potendo mettere un disco solo su un disco più grande. Com'è noto c'è un semplice algoritmo ricorsivo che risolve il rompicapo in modo ottimale in  $2^n - 1$  mosse.

Se i paletti sono 4 o più, il problema di trovare un algoritmo risolutivo con il minimo numero di mosse è molto più difficile ed è tuttora un problema aperto. Nella versione con quattro paletti è conosciuto come *reve's puzzle*<sup>1</sup>. Qui sotto sono mostrate 13 mosse che spostano 5 dischi su 4 paletti (con meno di 13 non è possibile):



Per tutti i problemi in cui si vuole trovare il percorso più breve tra due punti o il minor numero di mosse per trasformare una certa configurazione in un'altra, basta formulare il problema con un opportuno grafo e applicare la BFS per trovare una soluzione. Però l'algoritmo che si ottiene non è garantito essere anche efficiente. Tipicamente se il grafo è molto grande, la BFS può non essere efficiente a meno di "potare" il grafo.

Nel caso del reve's puzzle, i nodi del grafo sono le possibili configurazioni di dischi sui vari paletti e due configurazioni sono connesse da un arco se si può passare dall'una all'altra spostando il disco in cima ad un paletto su un altro paletto purché il disco in cima a quest'ultimo sia più grande di quello che vogliamo spostare. Si tratta quindi di adattare la BFS ad un grafo implicito, cioè un grafo che non è dato ma viene "costruito" durante la visita e solamente là dove serve. Infatti, possiamo "costruire" gli adiacenti ad un nodo considerando tutti i possibili movimenti leciti di dischi. Per mantenere l'albero di visita non possiamo usare un vettore dei padri perché non conosciamo tutti i nodi del grafo. Però possiamo usare una mappa o dizionario (`Map` in Java o `dict` in Python) che conterrà le associazioni (`conf2`, `conf`) tali che la configurazione `conf2` è visitata da `conf`.

```

REVEPUZZLE(np: numero paletti, nd: numero dischi)
  start <- configurazione iniziale /* Tutti i dischi sono sul paletto 1 */
  T[start] <- none /* T è una mappa per rappresentare l'albero di visita */
  Q <- coda vuota
  Q.enqueue(start)
  found <- none
  WHILE Q non è vuota DO
    conf <- Q.dequeue()
    IF conf è una configurazione finale THEN
      found <- conf
      BREAK
    FOR ogni possibile mossa (p1, p2) DO
      IF il disco in cima a p1 è più piccolo di quello in cima a p2 THEN
        conf2 <- configurazione ottenuta da conf spostando il disco in cima a p1 su p2
        IF conf2 non è in T THEN
          T[conf2] <- conf
          Q.enqueue(conf2)
  moves <- lista vuota
  IF found <> none THEN
    conf <- found
    WHILE conf <> none DO
      conf2 <- T[conf]
      IF conf2 <> none THEN
        m <- mossa che trasforma conf2 in conf
        moves.head(m)
      conf <- conf2
  RETURN moves

```

La complessità della visita, in funzione del numero di paletti e del numero di dischi, non è di facile determinazione ma sicuramente cresce velocemente al crescere dei due parametri. C'è un algoritmo chiamato algoritmo di [Frame-Stewart](#) che si congettura produca sempre il minor numero di mosse per un qualsiasi numero di paletti e di dischi.

## Esercizio [labirinto]

In un labirinto ci possono essere diversi percorsi minimi, cioè che attraversano un numero minimo di bivi, per andare dall'entrata all'uscita. Trovare un algoritmo efficiente che calcola il numero di percorsi minimi in un labirinto.

1. Il rompicapo fu originariamente proposto da uno dei più prolifici inventori di rompicapi, Henry Dudeney, nel 1907. Il nome *reve's puzzle* deriva dalla formulazione originale in cui un *reeve* (una sorta di sindaco di una regione rurale) lancia ad un pellegrino la sfida di spostare 8 forme di formaggio di dimensioni decrescenti da una piattaforma ad un'altra usando solamente 4 piattaforme e non mettendo mai una forma su una forma più piccola in al più 33 movimenti di forme. [↩](#)