METODI DI RICERCA

Per risolvere un problema nello spazio degli stati si può usare il seguente algoritmo (non deterministico):

STATO := stato iniziale;

while STATO ≠ stato finale do

STATO := risultato dell'applicazione di una qualsiasi operazione applicabile a STATO

Questo algoritmo determina una soluzione qualunque.

Certi problemi richiedono di trovare la sequenza OTTIMA di operatori dallo stato iniziale allo stato finale (ogni operazione ha un costo).

RICERCA DI SOLUZIONI SUL GRAFO DEGLI STATI

Normalmente è impossibile avere in memoria una struttura dati che descriva tutto lo spazio degli stati.

Lo spazio degli stati può essere infinito o finito, ma di dimensioni enormi.

Esempi:

- spazio degli stati degli scacchi: 10^{120} stati;
- spazio degli stati della dama: 10^{40} stati;

Generando tre miliardi di stati al secondo occorrono 10^{21} secoli per costruire tutto lo spazio degli stati.

I metodi di ricerca costruiscono passo passo un grafo di ricerca che è una porzione dello spazio degli stati, partendo dallo stato iniziale e scegliendo ad ogni passo uno stato da espandere, finché si raggiunge uno stato finale.

ESPANDERE: generare tutti gli stati raggiungibili con una operazione.

Metodi di ricerca in ampiezza

Espandono i nodi nell'ordine in cui sono generati.

Algoritmo:

1. porre il nodo di partenza in una lista di nome OPEN;
2. se OPEN è vuota, uscire con fallimento; altrimenti continuare;
3. rimuovere il primo nodo da OPEN e porlo in una lista di nome CLOSED; chiamare *n* questo nodo;

4. espandere il nodo *n* generando tutti i suoi successori. Se non ci sono successori andare subito a 2. Porre i successori alla *fine* di OPEN e creare i puntatori da tali successori a *n*;
5. se uno dei successori è un nodo finale uscire con la soluzione ottenuta percorrendo i puntatori all'indietro; altrimenti andare a 2.

Questo algoritmo assume che il nodo di partenza non sia anche un nodo finale, benché sia facile aggiungere un test che copra questa possibilità.

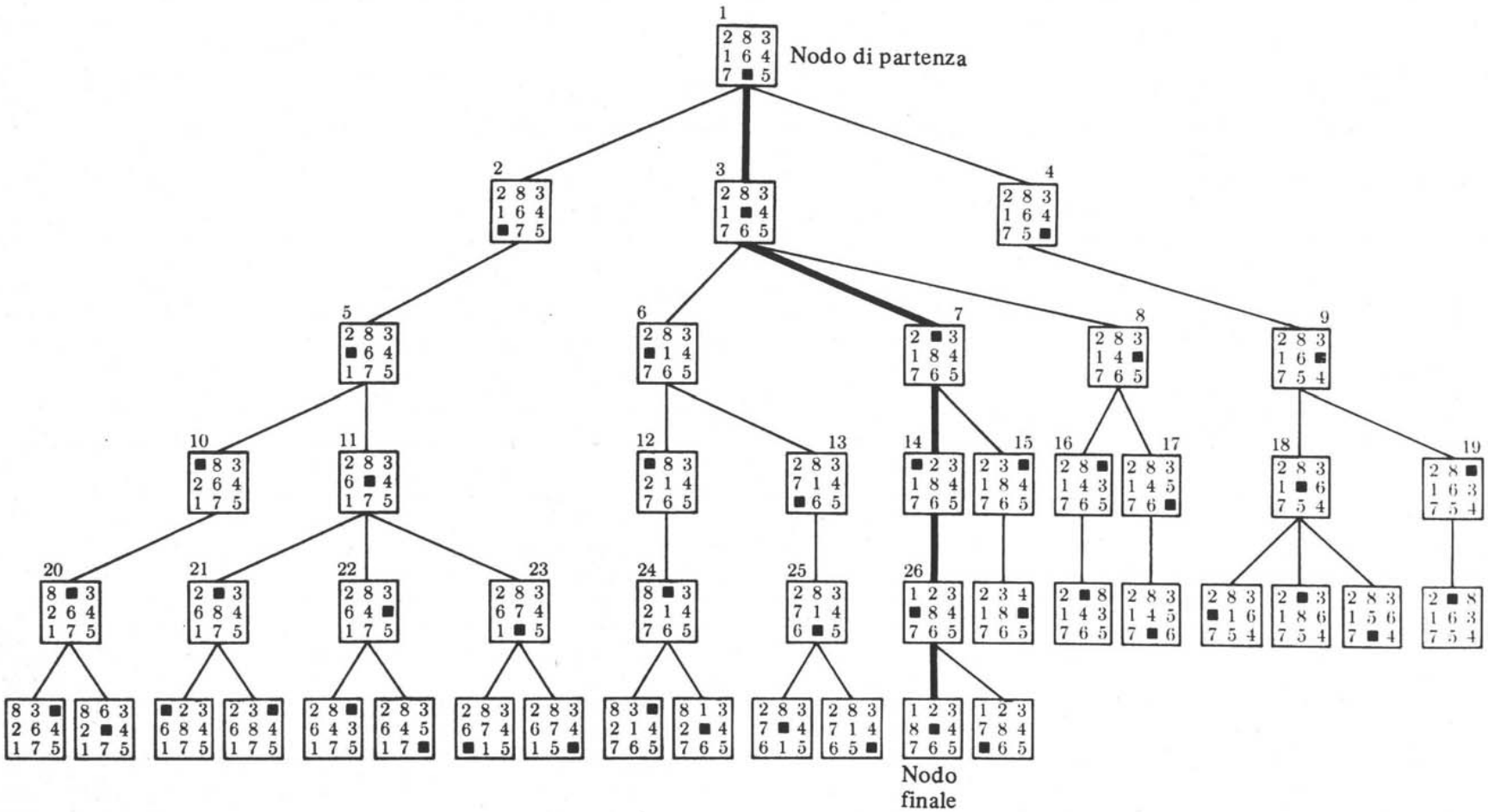
Più formalmente, l'algoritmo usa due insiemi:

- OPEN: insieme dei nodi ancora da espandere
- CLOSED: insieme dei nodi già espansi

OPEN := {nodo iniziale};
CLOSED := Ø;
TROVATO := false;
while nonvuoto(OPEN) *and not* TROVATO *do*
begin
togliere un nodo n da OPEN e metterlo in CLOSED;
if n è un nodo goal
then TROVATO := true
else generare tutti i successori di n, se ci sono, e metterli in OPEN;
end;
if TROVATO *then* successo
else fallimento.

Nella figura seguente l'applicazione della ricerca in ampiezza al gioco dell'otto.

Fig. 3.2 – L'albero prodotto da una ricerca in ampiezza



COSTO MINIMO

Ogni arco n-m ha associato un costo *c(n, m)*.

Il costo di un cammino è la somma dei costi degli archi che lo compongono.

Problema: trovare la soluzione di costo minimo.

Sia g(*n*) il costo del cammino dal nodo iniziale *s* a *n*.

L’algoritmo viene modificato come segue:

- ad ogni passo si toglie da OPEN il nodo *n* con g(*n*) minimo

- quando si genera, un successore *m*, gli si associa

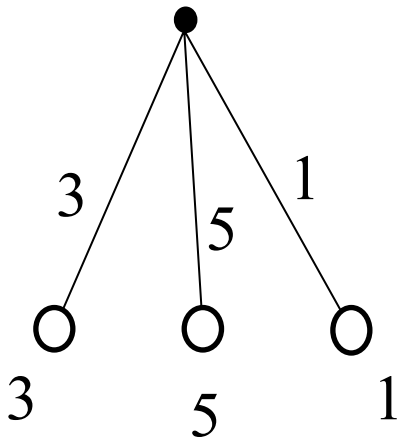
$$g(m)=g(n) + c(n, m)$$

L'algoritmo si ferma con la soluzione di costo minimo.

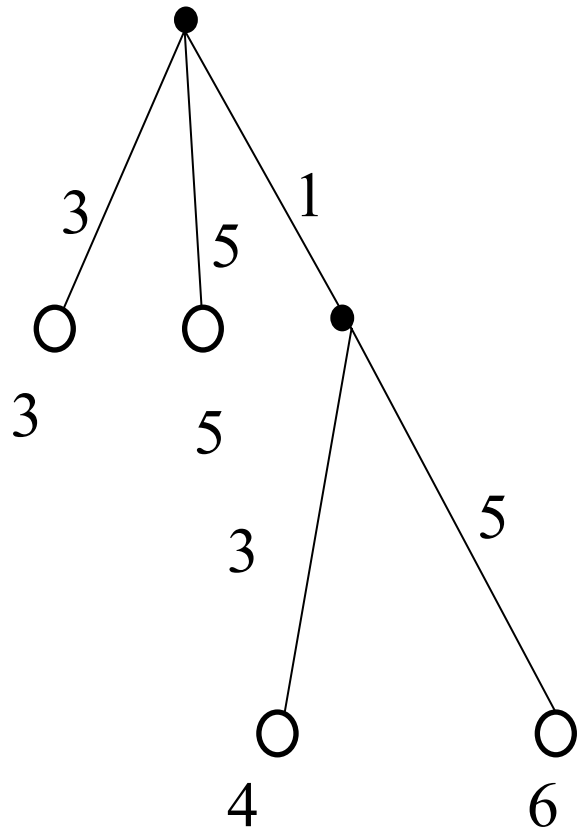
Esempio:

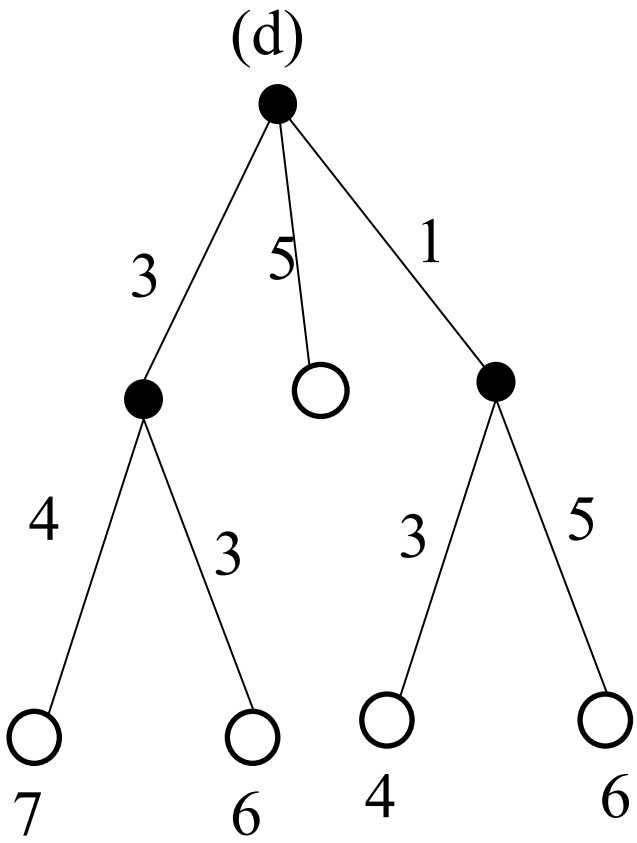
(a)

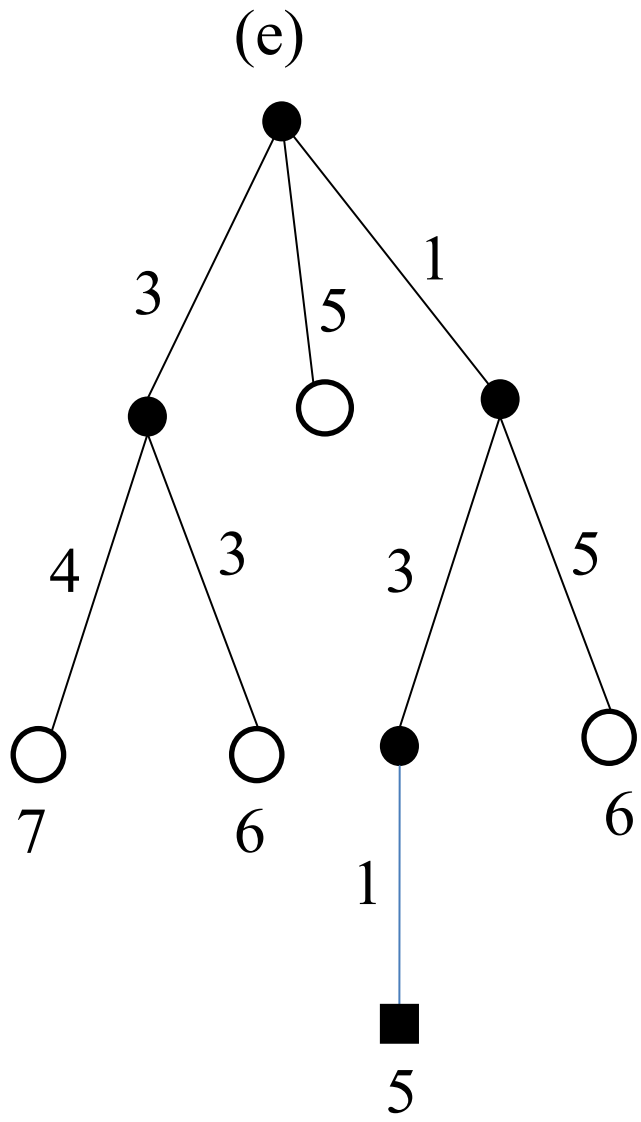
(b)



(c)




(d)
3
5
1
4
3
3
5
7
6
4
6


(e)
3
5
1
4
3
3
5
7
6
6
1
5

È una generalizzazione del metodi di ricerca in ampiezza:

il primo espande i nodi mantenendo *uguale la lunghezza*,

questo espande i cammini mantenendone *uguale il costo*, donde la denominazione di *metodo del costo uniforme*.

Formalizzazione del metodo del costo uniforme

Espande i nodi secondo valori crescenti di $\hat{g}(n)$.

Algoritmo:

1. porre il nodo di partenza *s* in una lista di nome OPEN. Porre .

2. se OPEN è vuota, uscire con fallimento; altrimenti continuare;

3. rimuovere da OPEN il nodo corrispondente al più piccolo valore di $\hat{g}$ e porlo in una lista di nome CLOSED, chiamandolo *n* (eventuali conflitti fra diversi valori minimi di $\hat{g}$ si possono risolvere arbitrariamente, dando comunque la precedenza ad un eventuale nodo finale);
4. se *n* è un nodo finale uscire con il cammino risolutivo ottenuto percorrendo i puntatori all'indietro; altrimenti continuare;

5. espandere il nodo n generando tutti i suoi successori. Se non ci sono successori andare subito a 2. Per ogni successore n_i, calcolare $\hat{g}(n_i) = \hat{g}(n) + c(n, n_i)$. Porre i successori in OPEN, associandoli ai rispettivi valori di $\hat{g}$ appena calcolati, e creare i puntatori a n;

6. andare a 2.

- Se $c(n, n_i) = 1$, si ricade nell'algoritmo per trovare cammini *a lunghezza minima*

- Se vi sono più nodi di partenza, si pongono *tutti* in OPEN al passo 1.

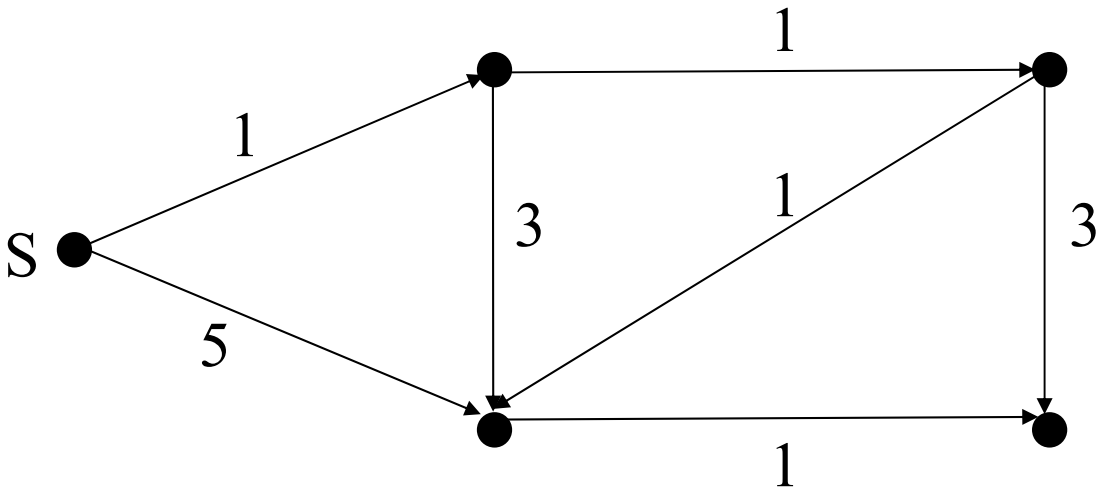
Algoritmo per costo minimo su grafi

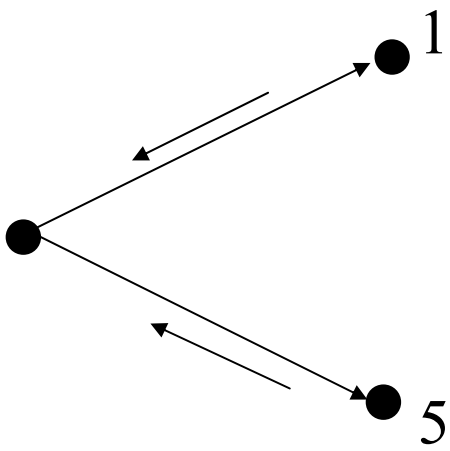
- Si associa ad ogni nodo *n* il costo $\hat{g}(n)$ del cammino minimo da *s* a *n* trovato fino a quel momento.

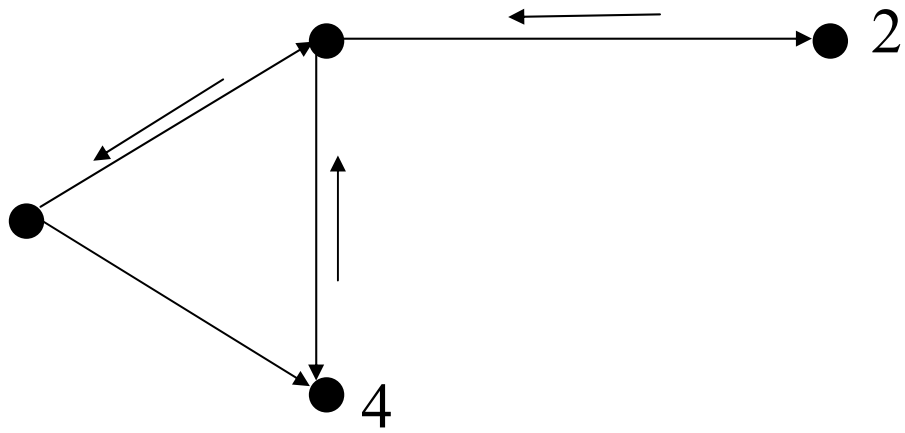
- Si modifica l'algoritmo precedente:

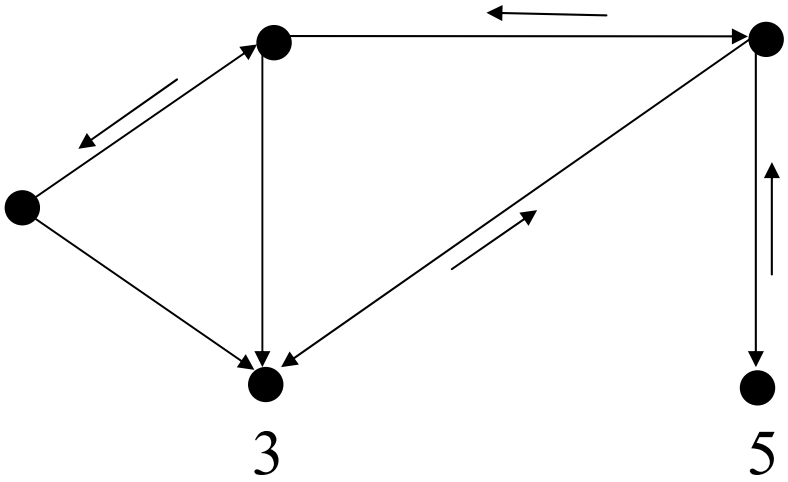
- Ad ogni passo togliere da OPEN il nodo *n* con $\hat{g}(n)$ minimo. Per ogni successore *m*:
  - se *m* è nuovo, metterlo in OPEN con $\hat{g}(m) = \hat{g}(n) + c(n, m)$ e mettere il puntatore all'indietro verso *n*;
  - se *m* è in OPEN e se $\hat{g}(m) > \hat{g}(n) + c(n, m)$ aggiornare il valore $\hat{g}(n)$ e spostare il puntatore verso *n*, altrimenti non fare niente;
  - se *m* è in CLOSED non fare niente. (Quando un nodo viene chiuso il suo costo è quello minimo e quindi non può più essere riaperto).

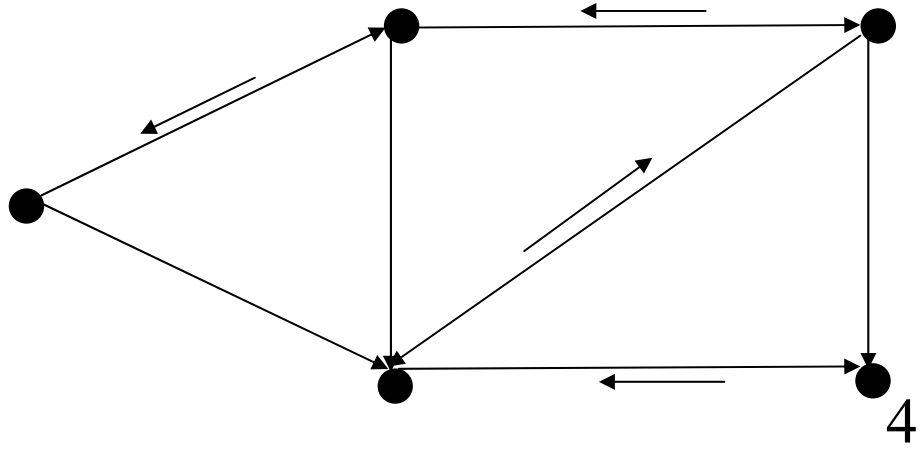
L'algoritmo è noto come Algoritmo di Dijkstra.








3
5


4

Un metodo di ricerca cieco porta ad una esplosione combinatoria.

Occorre usare conoscenza euristica per orientare la ricerca.

EURISTICO: che aiuta a scoprire.

Metodo di programmazione in cui la macchina procede lungo linee empiriche, usando regole pratiche, per trovare soluzioni.

Che tipo di informazione euristica usare?

1. Nella scelta del nodo da espandere (Algoritmi di ricerca euristica)

2. Per generare solo alcuni dei successori invece che tutti insieme

3. Decidere di *potare* alcuni rami dell'albero di ricerca.

Ricerca in ampiezza con potatura

(pruned breadth-first, detta anche *beam search*)

Si usa nei problemi complessi, quando si dispone di una *euristica* per misurare quanto è promettente un nodo.

Ad esempio, nel gioco dell'8, si valuta il numero di tessere fuori posto.

Si generano tutti i nodi di un certo livello di profondità, ma si tengono solo i β nodi più promettenti, cancellando tutti gli altri (β è scelto in base all'intuizione e alle risorse disponibili).

Sono necessarie tre liste: OPEN, CLOSED e RESERVE.

Quando si generano i nodi, si immagazzinano in RESERVE.

Alla fine della generazione si ordinano in base a quanto sono promettenti e si trasferiscono in OPEN i β migliori, cancellando gli altri.

Algoritmo:

1. Se il nodo sorgente *s* coincide con il goal, si esce con *successo* e con il percorso risolutivo vuoto.

2. Si inizializzano le liste OPEN, CLOSED e RESERVE a vuote. Si inserisce *s* in OPEN.

3. Se OPEN è vuota, allora
    1. se RESERVE è vuota, si esce con *fallimento*.
    2. se RESERVE non è vuota, allora si selezionano i β nodi più promettenti e si copiano in OPEN mantenendo l'ordinamento. Si pone RESERVE a vuota.
4. Si estrae il *primo* nodo *x* da OPEN.
5. Si espande *x*.
6. Se x non ha figli (è inerte), si va a (3).

7. Se un figlio di *x* è il nodo destinazione, si esce con *successo* e il percorso risolutivo ottenuto tramite i puntatori ai genitori.

8. Si cancellano tutti i figli di *x* che sono *renegade** o ricorrenti.

9. Si pongono i figli sopravvissuti in RESERVE, in un ordine qualsiasi.

10. Si va a (3).

(*) È *renegade* un nodo che viola i vincoli del problema.

Questa strategia è *incompleta*: potrebbe essere cancellato in quanto non promettente un nodo che sta nel percorso risolutivo.

Ecco un esempio:

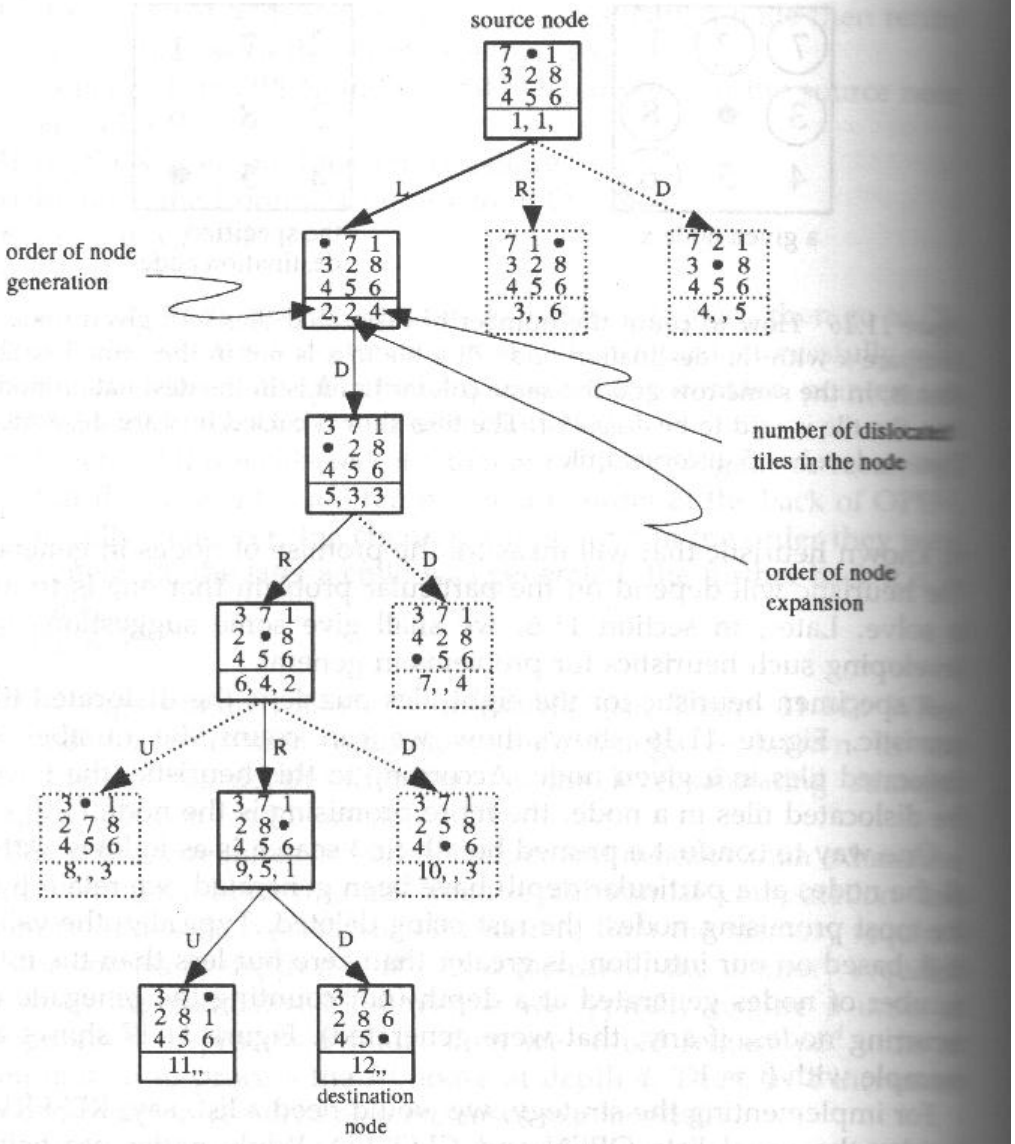


Figure 11.17 A pruned breadth-first search tree for the eight-tiles puzzle. Only the most promising node, as measured by the dislocated-tile heuristic, is retained at every depth. The deleted nodes are shown dotted above. Compare the above search tree with the search tree of Figure 11.15. The three-tuple below a node indicates the order in which the node was generated, the order in which it was expanded, and the number of dislocated tiles in it.

Metodi di ricerca in profondità

Si espande per primo l'ultimo nodo generato.

Definizione di profondità:

- la profondità della radice è zero
- la profondità di un discendente è uguale alla profondità del genitore più 1

Occorre assegnare un limite di profondità e prevedere una procedura di ritorno (backtraking)

Algoritmo:

1. porre il nodo di partenza in una lista di nome OPEN

2. se OPEN è vuota, uscire con fallimento; altrimenti continuare;

3. rimuovere il *primo* nodo da OPEN e porlo in una lista di nome CLOSED; chiamare *n* questo nodo;

4. se la profondità di *n* è uguale al limite di profondità, andare al passo 2

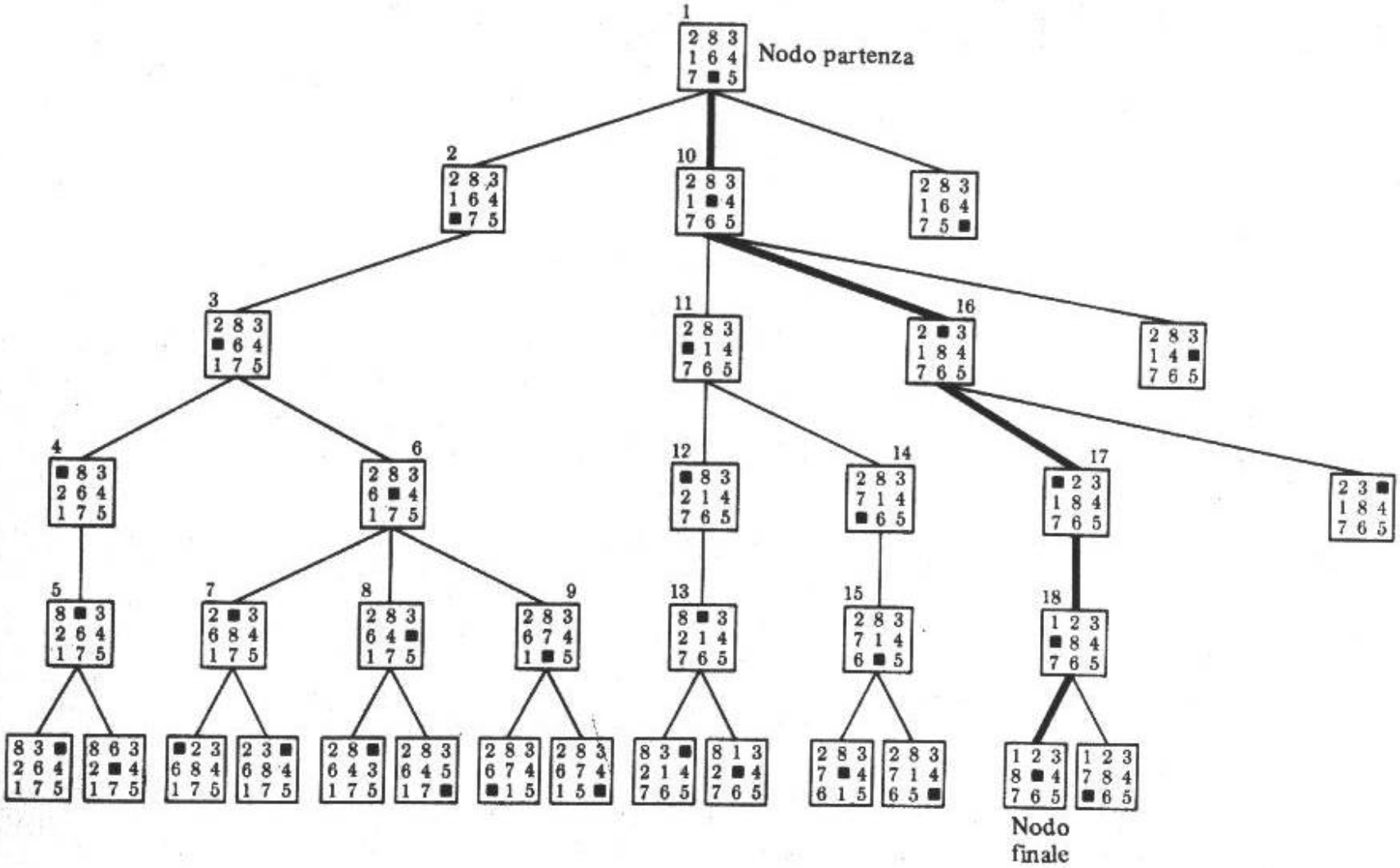
5. espandere il nodo *n* generando tutti i suoi successori. Porre questi (in ordine arbitrario) all'*inizio* di OPEN creando i puntatori a *n*;

6. se uno dei successori è un nodo finale uscire con la soluzione ottenuta percorrendo i puntatori all'indietro; altrimenti andare a 2.

In pratica, la lista OPEN è utilizzata come uno *stack*.

Nella figura successiva la ricerca in profondità è applicata al gioco dell'otto.

– *L'albero prodotto da una ricerca in profondità*



Un'implementazione dell'algoritmo di ricerca in profondità alternativa alla precedente, che utilizza uno stack esplicito, è quella che utilizza una procedura ricorsiva. Per iniziare, si consideri la seguente

Procedura ricorsiva BACKTRACK;

Ha in input lo stato iniziale e restituisce una sequenza di operazioni che portano da esso ad un goal; restituisce FAIL se non c'è nessuna soluzione.

```
procedure backtrack (STATO: state-description);

var    OPER: operator-name;
       OPERLIST:[operator-name];
       PATH:[operator-name] or  FAIL;

begin
   if goal (STATO) then return[];
   if deadend (STATO) then return FAIL;
   OPERLIST := prendi-operatori-applicabili (STATO);
   PATH:= FAIL;

   while nonvuota (OPERLIST) and PATH=FAIL do
       begin
              OPER:= first (OPERLIST);
              OPERLIST:= tail (OPERLIST);
              NUOVOSTATO:= applica (OPER, STATO);
              PATH:= backtrack (NUOVOSTATO)
       end;
   if PATH = FAIL then return FAIL
   else return aggiungi1 (OPER, PATH)
end.
```

Dove:

"goal (X)" : true, se X è uno stato goal.

"deadend (X) " : true se da X non si può procedere.

"prendi-operatori-applicabili (X)" : restituisce la lista di tutti gli operatori applicabili allo stato X.

"nonvuota (L)" : true se la lista L non è vuota

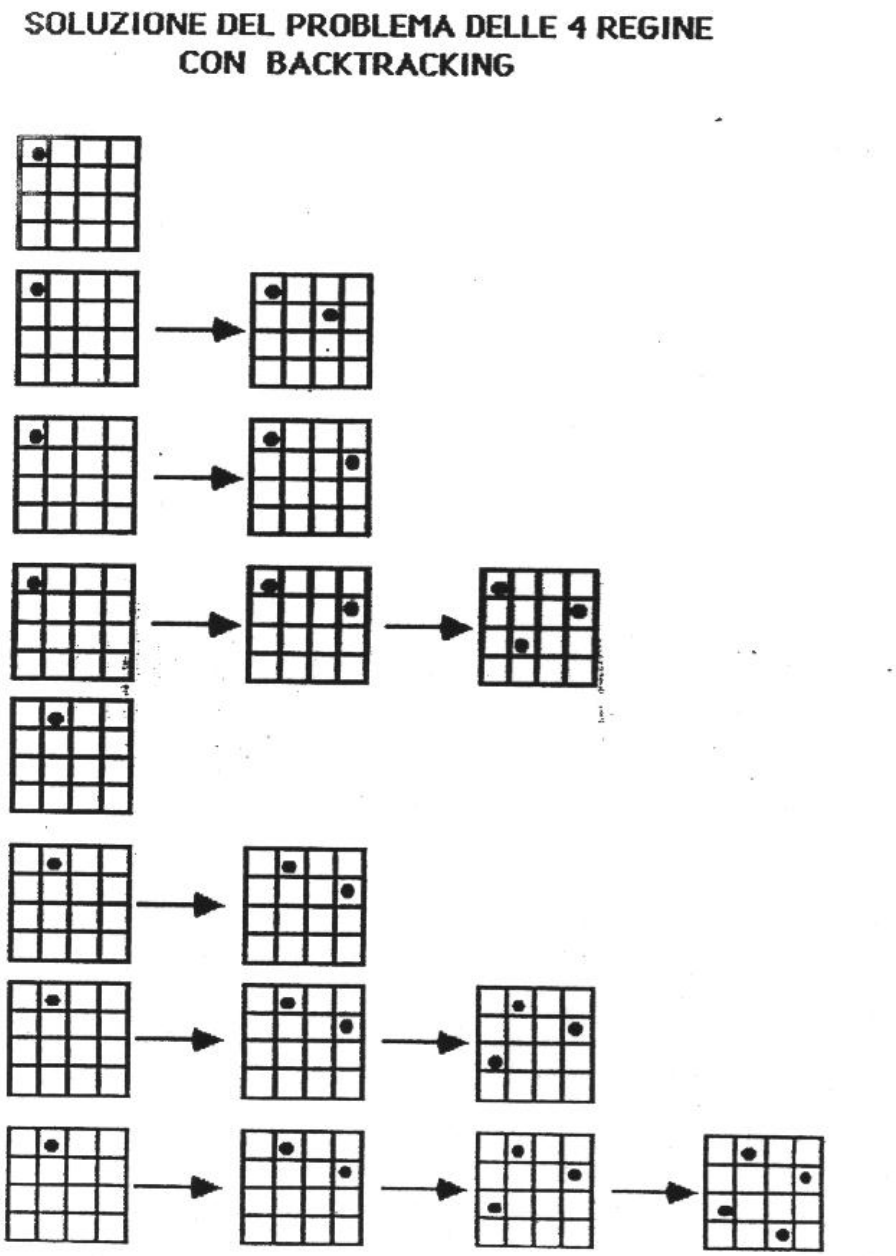
"first (L)": restituisce il primo elemento della lista L.

"tail (L)": restituisce la lista L senza il primo elemento.

"applica (O, X)" : restituisce lo stato che si raggiunge applicando l'operatore O allo stato X.

"aggiungil (E, L)": restituisce la lista che si ottiene inserendo l'elemento E in testa alla lista L.

Nella figura che segue un'applicazione dell'algoritmo.



Si è visto che il BACKTRACKING può essere realizzato iterativamente con uno stack. Ogni elemento dello stack contiene tutta l'informazione relativa ad una chiamata ricorsiva

Stato s	Operazioni applicabili a s
.....	
Stato iniziale	Operazioni applicabili allo stato iniziale

Ad ogni posso la sequenza di elementi sullo stack corrisponde ad un cammino parziale nel grafo di ricerca.

La procedura di backtracking potrebbe non terminare se l'albero di ricerca ha dei cammini infiniti. Si deve imporre una profondità massima. La ricerca in profondità in forma ricorsiva si può attuare mediante la seguente:

Procedura ricorsiva BACKTRACK1

Come la BACKTRACK, ma effettua controlli supplementari per evitare cicli e rami troppo lunghi.

Ha in input una sequenza di stati, inizialmente composta dal solo stato iniziale e restituisce una sequenza di operazioni che portano da tale stato iniziale ad un goal; restituisce FAIL se non c'è nessuna soluzione.

N.B. si assume che sia definita la variabile globale LIVELLO-MAX, che dice qual è il numero massimo di operatori che possono comparire in una soluzione.

```
procedure backtrack1 (SEQUENZA-STATI: [state-description]);

var    STATO: state-description;
       OPER:  operator-name;
       OPERLIST: [operator-name];
       PATH:  [operator-name] or FAIL;
       NUOVA-SEQUENZA: [state-description];

begin
   STATO := first (SEQUENZA-STATI);
   if member (STATO, tail (SEQUENZA-STATI))
        then return FAIL;
   if goal (STATO) then return[];
   if deadend (STATO) then return FAIL;
   if length (SEQUENZA-STATI) > LIVELLO-MAX
        then return FAIL;
   OPERLIST := prendi-operatori-applicabili (STATO);
   PATH:= FAIL;
```

```
while nonvuota (OPERLIST) and PATH=FAIL do
  begin
OPER := first (OPERLIST);
OPERLIST := tail (OPERLIST);
NUOVOSTATO := applica (OPER, STATO);
NUOVA-SEQUENZA := aggiungi1(NUOVOSTATO,
  SEQUENZA-STATI);
PATH := backtrack1 (NUOVA-SEQUENZA)
  end;

  if PATH FAIL then return FAIL
  else return aggiungi1 (OPERPATH)

end.
```

Ricerca in profondità con hindsight backtracking

L'algoritmo precedente è solitamente detto "con backtracking *cronologico*".

Si possono immaginare delle varianti.

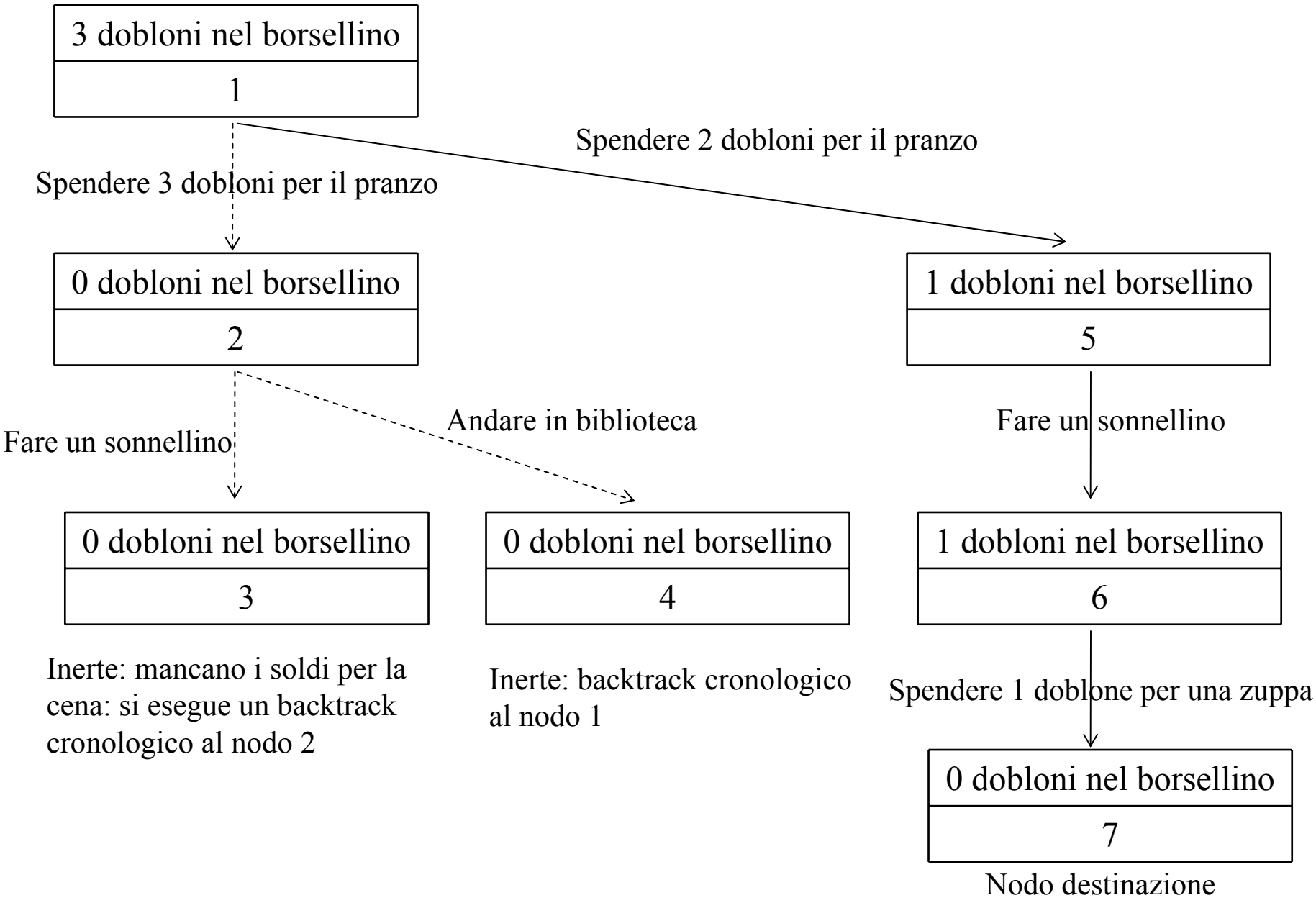
Definizione:

Un nodo *z* è detto *subversive* se e solo se qualunque percorso dal nodo sorgente passante per *z* diventa barricato in qualcuno dei suoi discendenti.

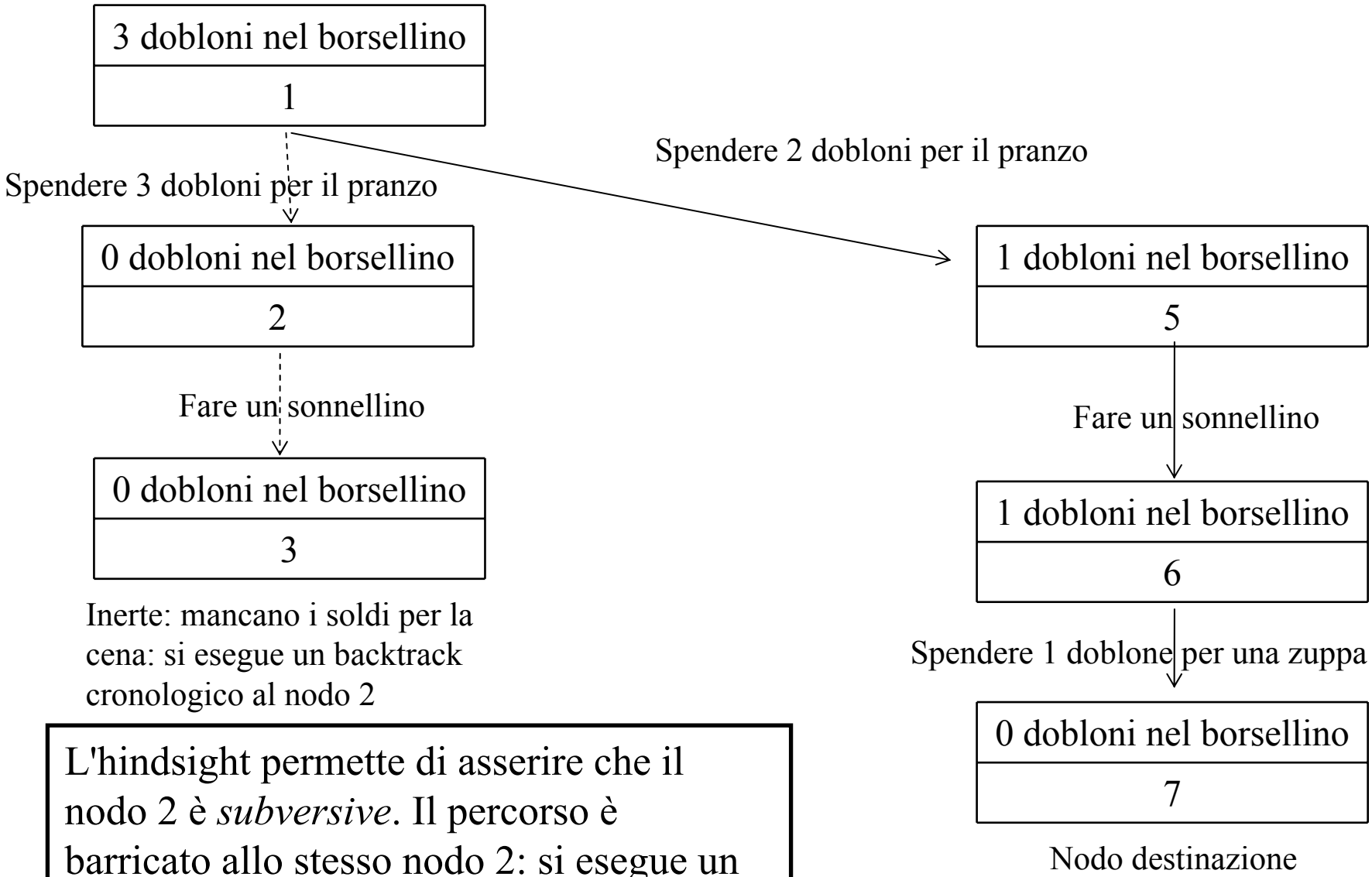
Un nodo subversive non può quindi appartenere al percorso risolutivo.

Si consideri il caso in cui si voglia pianificare il pranzo, l'attività pomeridiana e la cena con un budget di 3 dobloni.

La figura seguente mostra una ricerca in profondità con backtracking cronologico.


3 dobloni nel borsellino
1
Spendere 3 dobloni per il pranzo
Spendere 2 dobloni per il pranzo
0 dobloni nel borsellino
2
1 dobloni nel borsellino
5
Fare un sonnellino
Andare in biblioteca
Fare un sonnellino
0 dobloni nel borsellino
3
0 dobloni nel borsellino
4
1 dobloni nel borsellino
6
Inerte: mancano i soldi per la cena: si esegue un backtrack cronologico al nodo 2
Inerte: backtrack cronologico al nodo 1
Spendere 1 doblone per una zuppa
0 dobloni nel borsellino
7
Nodo destinazione

Invece la figura seguente ne mostra uno con *hindsight backtracking* (si acquista maggiore efficienza).



L'hindsight permette di asserire che il nodo 2 è *subversive*. Il percorso è barricato allo stesso nodo 2: si esegue un bactrack al nodo 1.

Come formalizzare questa intuizione?

Si supponga che si stia attraversando un percorso $x_0, x_1, x_2 \ldots$, dove x_0 è il nodo origine, e che il percorso diventa barricato ad un certo nodo x_n.

Per tentare l'hindsight backtracking occorre:

1. Utilizzando le conoscenze acquisite su come il percorso diventa barricato, si scandiscono i nodi $x_0, x_1, x_2 \ldots$, cercando un nodo x_i che si asserisce essere *subversive* (comunque quello a profondità minore).

2. Si effettua il bactracking cronologico a partire da x_i, cancellando tutti i nodi che stanno sotto questo.

Per la realizzazione dell'algoritmo servono due liste:

la lista OPEN per i nodi inespansi o parzialmente espansi;

la lista PROGENY per assicurarsi che un nodo appena generato sia nuovo.

Algoritmo:

1. Se il nodo di partenza è un nodo goal, uscire con *successo* e il percorso di soluzione vuoto.

2. Si inizializzano OPEN e PROGENY a vuote. Si pone il nodo sorgente in OPEN.

3. Se OPEN è vuota, si esce con *fallimento*.

4. Si preleva da OPEN il nodo *x* più recente e si genera un *nuovo* figlio: si effettua cioè una verifica in PROGENY.

5. Se non si può generare nessun figlio di *x*, cioè è inerte o completamente già espanso, allora
   1. si elimina *x* da OPEN;
   2. se si è acquisita conoscenza specifica (*hindsight*), si cerca un nodo *subversive*, partendo dal fondo di OPEN; se lo si scopre, lo si elimina da OPEN insieme a tutti i nodi successori (quelli che stanno nella parte anteriore di OPEN);
   3. si va a (III).

6. Se *x* ha un nuovo figlio, si pone questo in PROGENY.

7. Se il figlio di *x* è un nodo destinazione, si esce con *successo* e con il percorso risolutivo che si trova in OPEN.

8. Se il figlio di *x* è *renegade*, ricorrente o *insipid**, si va a (IV).

9. Si pone il figlio di *x* in cima alla lista OPEN.

10. Si va a (III).

(*) Si ha un nodo *insipid* quando si supera la profondità o il costo previsto.

Questo algoritmo è noto nella letteratura anche con i nomi *dependency-based*, *dependency-directed* o *non-chronological* backtracking.

Ricerca in profondità con leap-frogging

Invece di generare di un nodo un figlio alla volta, si generano tutti.

Se ne sceglie uno, con un qualche criterio, e lo si espande.

Si procede così:

Sia $x_0, x_1, x_2 \ldots$ il percorso di attraversamento. Questo percorso sia barricato in x_n.

Possiamo *saltare* (**leap-frog**) ad uno dei fratelli dei nodi $x_0, x_1, \ldots, x_n$.

Due modi per farlo: cronologico oppure con *hindsight*.

Cronologico: si torna indietro scandendo i nodi $x_n, x_{n-1}, \ldots$ fino a scoprire un nodo x_j che ha un fratello x'_j che non è stato ancora attraversato.

Si prosegue da quel nodo, sempre con le stesse modalità.

Esempio:

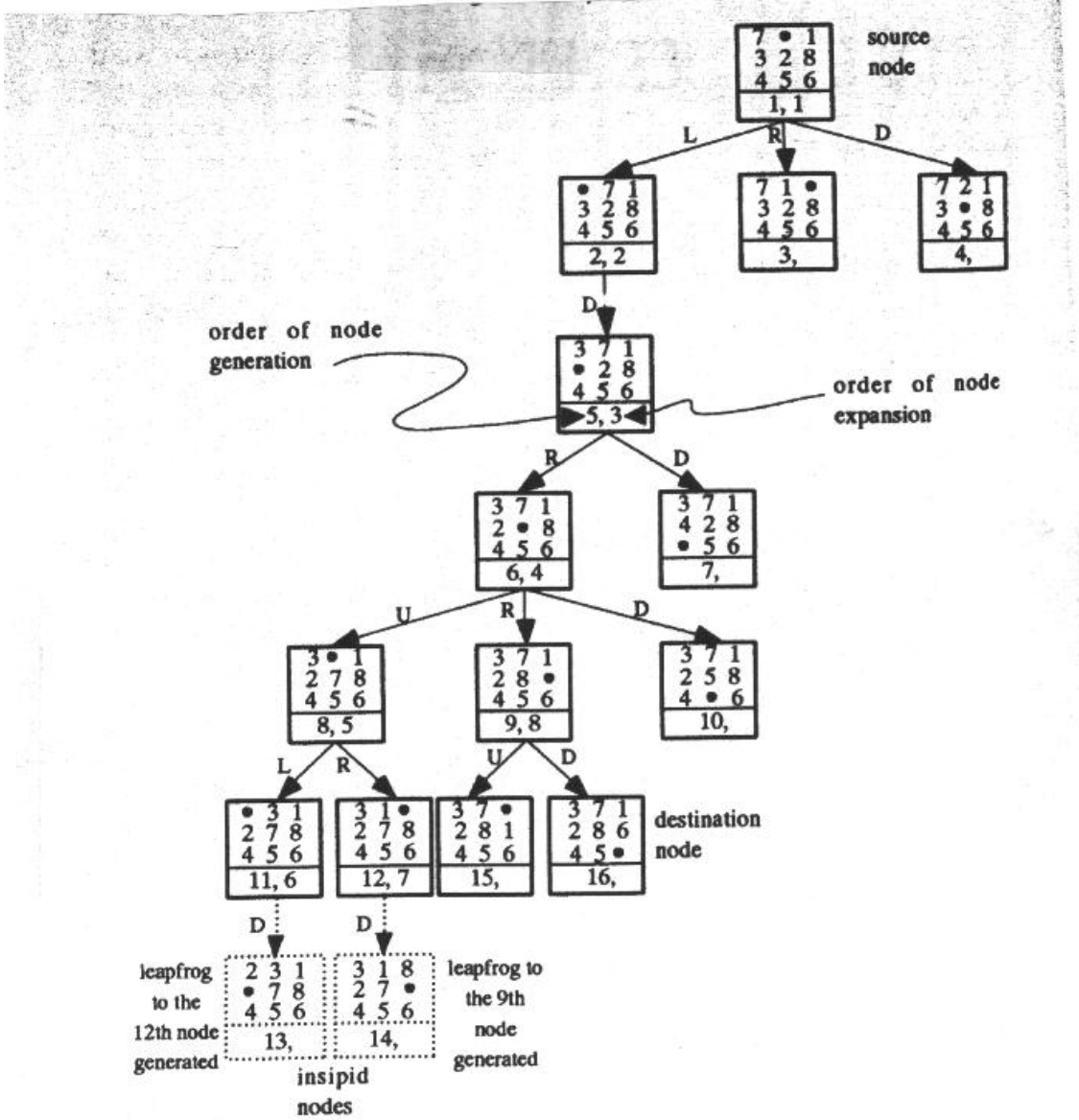


Figure 11.23 An example of depth-first search, with leap-frogging and a depth-bound of six, for the eight-tiles puzzle. One child is arbitrarily selected [shown above in left to right order] for expansion from the children of a node. To prevent cluttering the figure, we have not shown any recurring nodes. Neither any renegade nor any inert nodes were generated. The source and destination nodes are as specified in Figure 11.12.

Con hindsight: quando si scopre che il percorso è barricato in x_n, si procede secondo i seguenti passi:

1. si cerca un nodo *subversive* x_i, esattamente come visto prima;

2. si effettua un "salto della rana" (leap-frog) cronologico a partire da x_i.

Scelta del nodo da espandere: o arbitrario, o basato su qualche *euristica* (quanto un nodo è promettente).

Una ricerca in profondità che sceglie i figli da espandere in base alla promessa del nodo, è noto in letteratura col nome **hill-climbing** (il nodo destinazione è visto come la cima di una collina, e ci si arrampica per il sentiero più ripido).

Nella realizzazione della ricerca in profondità con leap-frogging, si utilizzano due liste, OPEN e CLOSED.

Algoritmo:

1. Se il nodo di partenza è un nodo goal, uscire con *successo* e il percorso di soluzione vuoto.

2. Si inizializzano OPEN e CLOSED a vuote. Si pone il nodo sorgente in OPEN.

3. Se OPEN è vuota, si esce con *fallimento*.

4. Si preleva da OPEN il nodo *x* più recente.

5. Si pone *x* in CLOSED.

6. Si espande *x*.

7. Se non si può generare nessun figlio di *x*, cioè è inerte, allora
   1. se si è acquisita conoscenza specifica (*hindsight*), si cerca un antenato subversive di *x*; se lo si scopre, lo si rimuove da OPEN insieme a tutti i discendenti in OPEN;
   2. si va a 3.

8. Se il figlio di *x* è un nodo destinazione, si esce con *successo* e con il percorso risolutivo ottenuto inseguendo all'indietro i puntatori al padre.

9. Cancellare tutti i figli di *x* che siano *renegade*, ricorrenti o *insipid*.

10. Si pongono i figli sopravvissuti di *x* in ordine decrescente in cima alla lista OPEN.

11. Si va a 3.

Esempio:

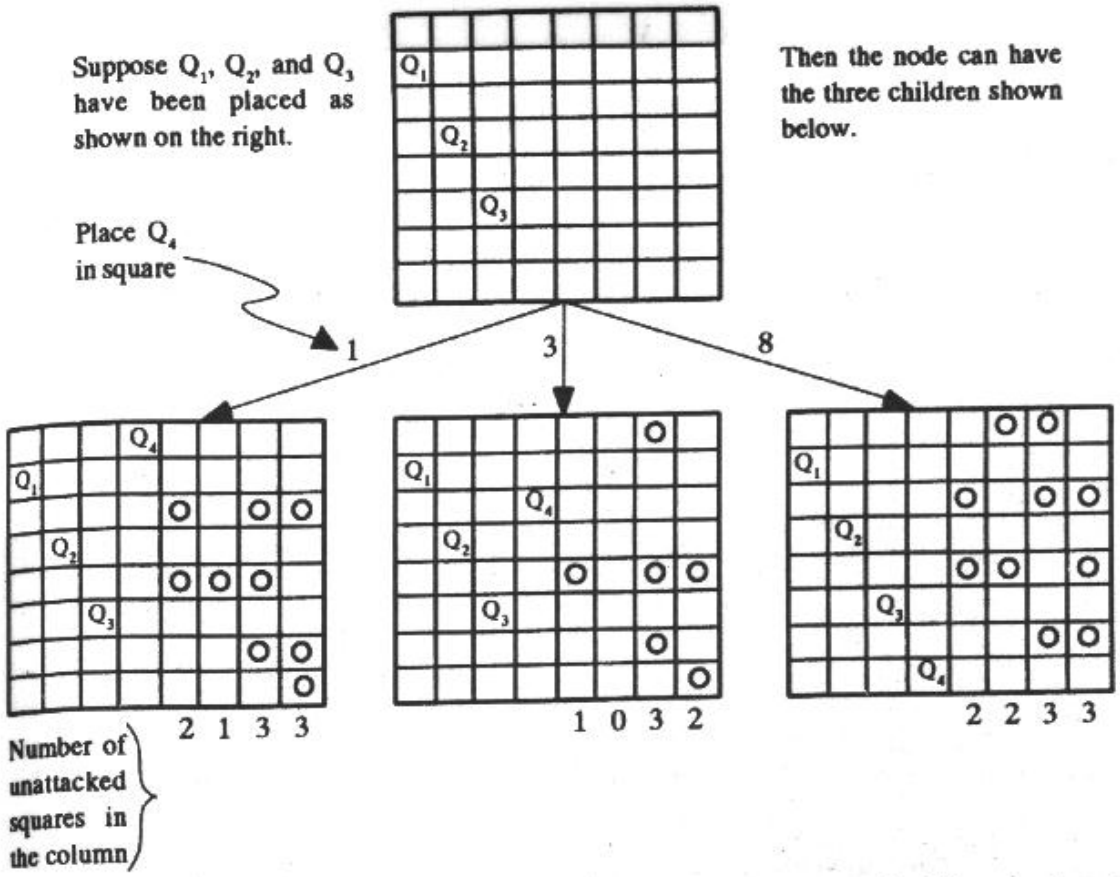


Figure 11.25 A specimen heuristic to compare the promise of sibling nodes for the eight-queens puzzle. Let the promise of a node be equal to the minimum of the number of unattacked squares in each of the columns in which the queens are yet to be placed. Unattacked squares are denoted by the symbol 'O' in the three siblings above. Then the promise of the siblings from left to right are min(2,1,3,3), min(1,0,3,2), and min(2,2,3,3); that is, 1, 0, and 2. Thus the rightmost sibling is the most promising. The middle sibling is in fact a subversive node.

Classificazione delle strategie di ricerca.

Una *strategia* di ricerca è COMPLETA se e solo se garantisce di trovare un percorso risolutivo, se esiste.

È INCOMPLETA se non è completa.

Una stategia dovrebbe essere completa.

Occasionalmente si può adottare una strategia incompleta se, per particolari problemi, questa strategia dà fiducia di trovare la soluzione con notevole *incremento di efficienza*.

Una strategia è AMMISSIBILE se garantisce di trovare il percorso risolutivo di *minimo costo*.

Una strategia è DIFENDIBILE (*defensible,* giustificabile) se garantisce di trovare il percorso risolutivo *più corto*.

Una strategia è CONCESSIBILE (*concessible*) se è completa ma non è né ammissibile né defendibile.

Number of solution paths in the problem	The cost of all moves is	The class or subclass of the search strategy, and the solution path found			
		Complete			Incomplete
		Admissible	Defensible	Concessible	
more than one	not the same	cheapest, which may not be the shortest	shortest, which may not be the cheapest	any solution path, which may be neither the cheapest nor the shortest	no solution path may be found even though it exists
	the same	cheapest, which will also be the shortest	shortest, which will also be the cheapest		
one	of no relevance	the single solution path			
zero		none, since no solution path exists			

Figure 11.14 The solution path, if any, found by a search strategy for a given problem depends on [1] the number of solution paths that exist in the problem, [2] the cost of the moves as compared to one another, and [3] the class or subclass of the search strategy. Since exhaustive search finds all solution paths that exist, it is both admissible and defensible. An admissible strategy can be adopted to find the shortest solution path in a problem where all moves are considered to cost the same.

Algoritmi di ricerca euristica

Nei precedenti algoritmi di ricerca euristica si è lasciato aperto il problema di come selezionare il nodo da espandere (procedura *previsione*).

Utilizzando diverse versioni di *previsione*, si possono modellare diverse strategie di ricerca:

- Nella ricerca in ampiezza (bradth-first) si ha che

  previsione(X) = livello(X)

  (dove il livello di un nodo è, come al solito, la sua distanza dalla radice)

- Nella ricerca in profondità (depth-first) si ha che

  previsione(X) = -livello(X)

- Nell'algoritmo di Dijkstra (costo minimo) si ha che

  previsione(X) = costo del cammino fatto per raggiungere X

N.B. 1: L'algoritmo di Dijkstra è una versione più generale della ricerca in ampiezza (archi non necessariamente di costo unitario) che opera sui grafi.

N.B. 2: In tutti i casi visti, la scelta è fatta sulla base del cammino già percorso. In altre parole, si guarda solo "indietro".

Utilizzando la conoscenza del dominio si può ottenere una indicazione più precisa.

Si deve cercare di valutare quanto promettente ("vicino" al goal) è lo stato in cui ci si trova:

previsione(X) = stima del costo minimo da START a X + stima del costo minimo da X a GOAL

Uso delle funzioni di valutazione

Si suppone di conoscere una funzione $\hat{f}$ (generalmente arbitraria) che fornisce *una stima* del costo di un cammino a costo minimo dal nodo di partenza al nodo finale vincolato a passare per un certo nodo *n*. $\hat{f}(n)$ è il valore della funzione in *n*.

$\hat{f}$ viene usata per ordinare i nodi da espandere, cioè ad ogni passo si ordinano i nodi secondo valori crescenti di $\hat{f}$.

Si ha allora un *algoritmo di ricerca ordinata* (algoritmo A).

Algoritmo A

Algoritmo di ricerca ordinata (per alberi e grafi):

1. porre il nodo di partenza s in una lista di nome OPEN e calcolare $\hat{f}(s)$;

2. se OPEN è vuota, uscire con fallimento; altrimenti continuare;

3. rimuovere da OPEN il nodo con il più piccolo valore di $\hat{f}$ e porlo in una lista di nome CLOSED, chiamandolo *n* (eventuali conflitti fra diversi valori minimi di $\hat{f}$ si possono risolvere arbitrariamente, dando comunque la precedenza a un eventuale nodo finale);

4. se *n* è un nodo finale uscire con il cammino risolutivo ottenuto percorrendo i puntatori all'indietro; altrimenti continuare;

5. espandere il nodo *n* generando tutti i suoi successori. Se non ci sono successori andare subito a 2. Per ogni successore n_i calcolare $\hat{f}(n_i)$;

6. associare ai successori che ancora non si trovano né in OPEN né in CLOSED i rispettivi valori di $\hat{f}$ appena calcolati, creare i puntatori a *n* e inserirli in OPEN;

7. associare ai successori che già si trovassero in OPEN o in CLOSED il più piccolo fra il valore di $\hat{f}$ appena calcolato e quello precedente. Porre in OPEN i successori che si trovavano in CLOSED per cui è diminuito il valore di $\hat{f}$, e ridirigere a *n* i puntatori di tutti i nodi il cui valore di $\hat{f}$ è diminuito;

8. andare al passo 2.

Esempio

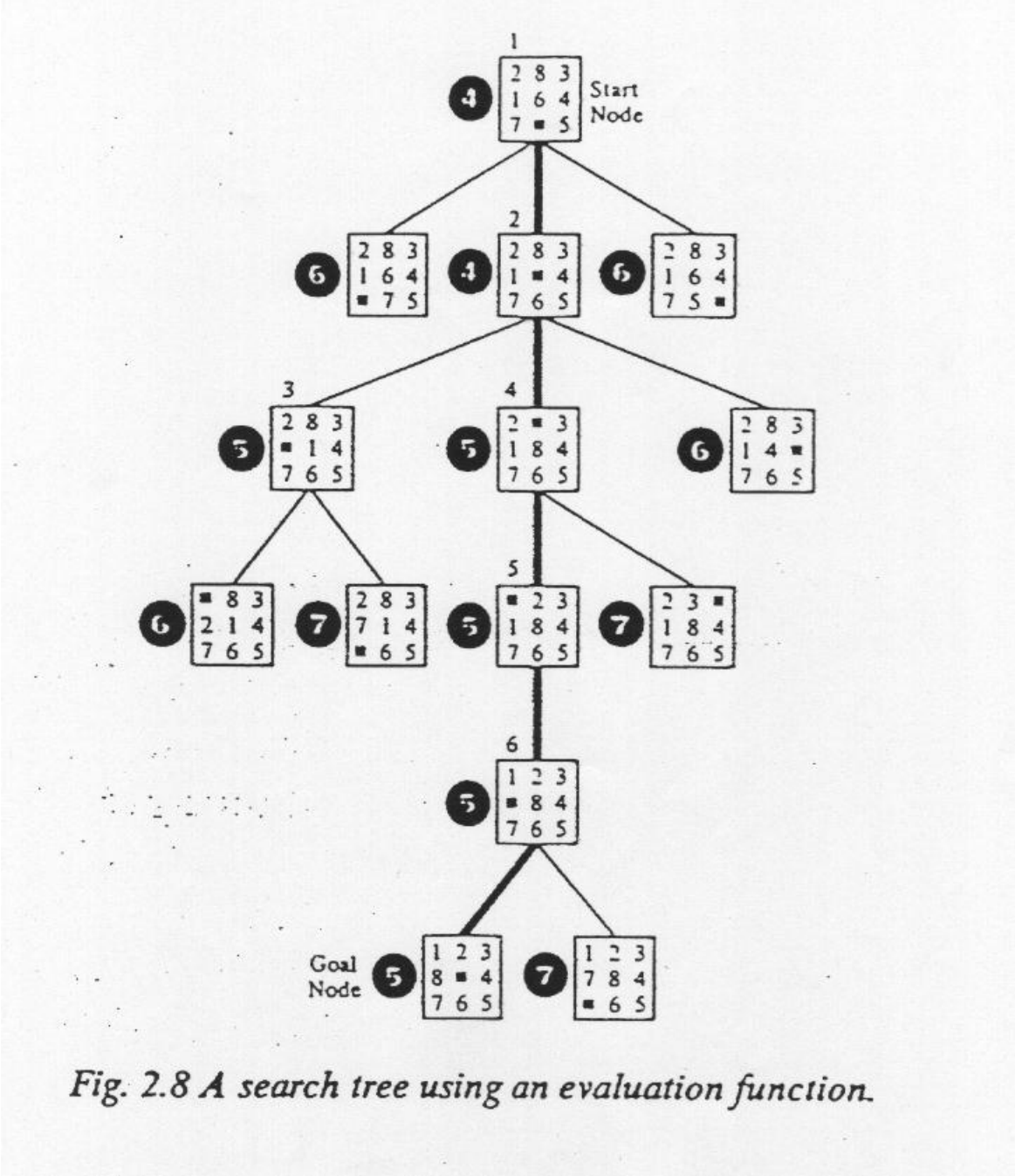


Fig. 2.8 A search tree using an evaluation function.

Anche il precedente algoritmo può essere realizzato in forma ricursiva.

Procedura GRAPHSEARCH

Rispetto a BACKTRACK e BACKTRACK1 essa "ricorda" tutti i nodi già trattati e non solo quelli sul cammino attuale.

Ha in input lo stato iniziale e restituisce una sequenza di stati che portano da quello iniziale ad un goal; restituisce FAIL se non c'è nessuna soluzione.

```
procedure graphsearch (START: state-description);

type    node == <state-description, real-number>;
        arc == <node, node, boolean>
var
        N-STATE:        state-description;
        N-NODE: node;
        START-NODE: node;
        ATTUALE: node;
        OPEN: [node];
        CLOSED: [node];
        GRAFO:  <[node], [arc]>;
        SUCCESSORI: [state-description];
/*
Le strutture dati sono formate da nodi che includono, oltre
allo stato, anche un numero che rappresenta il costo del
cammino migliore (presunto) dallo stato iniziale ad un goal che
passi per quello stato.
Lo spazio degli stati e' rappresentato da GRAFO, che è
costituito da un insieme di nodi e un insieme di archi
Ciascun arco include un booleano, che specifica se l'arco fa
parte del cammino migliore che conduce dallo start node al nodo
di arrivo.
OPEN e CLOSED sono liste di nodi.
*/
```

```
begin
START-NODE:= <START, previsione (START>;
OPEN:= [START-NODE];
CLOSED:= [ ];

/*
commento: questo e' il ciclo principale; si prende un nodo da OPEN e
lo si espande; a seconda della "bontà" degli stati raggiunti si
possono o no compiere complesse ristrutturazioni del grafo
*/

    while nonvuota (OPEN) do
        begin
        ATTUALE:= prendi-nodo (OPEN);
        CLOSED:= aggiungi (CLOSED, ATTUALE);
        if goal (ATTUALE) then
            return estrai-cammino (ATTUALE,GRAFO)
        else
            SUCCESSORI:= espandi (ATTUALE);
```

```
/*
in questo ciclo interno vengono trattati, uno ad uno gli stati
ottenuti dall'espansione di ATTUALE
*/
    for-each N-STATE in SUCCESSORI do
        begin
        COSTO-P:= previsione (N-STATE);
/*
se lo stato era già presente in OPEN, ma il "costo previsto" ottenuto
col nuovo cammino è minore della precedente, aggiorna sia OPEN che il
GRAFO
*/
        if già-presente (N-STATE, OPEN) then
            begin
            if COSTO-P < prendi-previsione (N-STATE,OPEN) then
                begin
                sostituisci (N-STATE, COSTO-P, OPEN);
                OLD-PARENT:= predecessore (N-STATE, GRAFO);
                falsifica (N-STATE, OLD-PARENT, GRAFO);
                N-NODE:= <N-STATE, COSTO-P>;
                espandi-grafo (N-NODE, ATTUALE, GRAFO)
                end
            end
```

```
/*
se lo stato era già presente in CLOSED, ma il "costo previsto"
ottenuto col nuovo cammino è minore della precedente, aggiorna sia
CLOSED che il GRAFO.  In questo caso, però, l'aggiornamento del GRAFO
è molto più complesso, in quanto è necessario propagare in avanti
l'aggiornamento
*/

        else if già-presente (N-STATE, CLOSED) then
            begin
            if COSTO-P < prendi-previsione (N-STATE, CLOSED) then
                begin
                sostituisci (N-STATE, COSTO-P, CLOSED);
                OLD-PARENT:= predecessore (N-STATE, GRAFO);
                falsifica (N-STATE, OLD-PARENT, GRAFO);
                N-NODE:= <N-STATE, COSTO-P>;
                espandi-grafo (N-NODE, ATTUALE, GRAFO)
                propaga (N-NODE, BONTÀ, GRAFO)
                end
            end
```

```
/*
se il nodo non era né in CLOSED, né in OPEN, è sufficiente metterlo
in OPEN e aggiornare il GRAFO
*/
        else
            begin
            inserisci (N-NODE, BONTÀ, OPEN);
            espandi-grafo (N-NODE,ATTUALE,GRAFO)
            end
        end
/*
fine ciclo interno : sono stati trattati tutti i successori
*/
    end;
/*
fine del ciclo principale; se si arriva qui significa che si è
svuotata OPEN senza aver mai incontrato un nodo goal. Si ha quindi un
fallimento.
*/

return FAIL
end.
```

Dove:

"previsione (X)": calcola quanto si presume possa costare il cammino migliore da START a un GOAL che passi attraverso X.

"prendi-nodo (L)": sceglie, secondo qualche criterio, un nodo di L e lo toglie da L.

"aggiungi (L, N)": inserisce N nella lista L.

"espandi (N)": trova tutti gli stati successori di N.

"già-presente (S, L)": true se in L e' presente un nodo avente S come stato.

"prendi-previsione (N, L)": prendi il valore che, nella lista L, è associato al nodo N.

"sostituisci (S, V, L)": il nodo che nella lista L ha S come stato viene rimpiazzato da un nodo avente lo stesso S come descrittore di stato, ma il nuovo valore V.

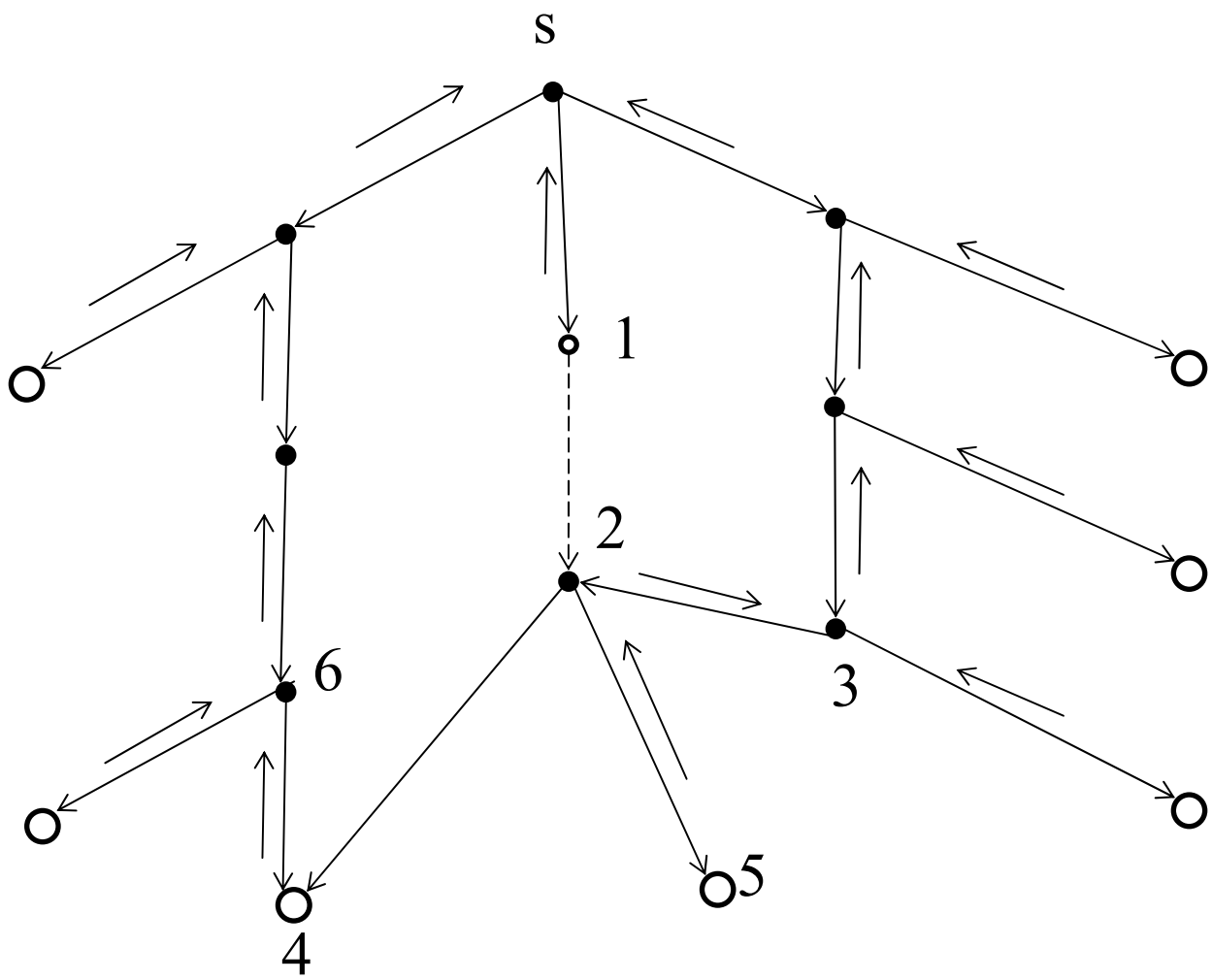
"predecessore (S, G)": preleva dal grafo G il nodo che precede quello associato a S sul cammino migliore (e cioè quell'X tale che <X, N, true> sia nel grafo, dove N è il nodo che ha S come stato).

"falsifica (S, N2, G)": nel grafo G, l'arco <N1, N2, true> viene sostituito da <N1, N2, false>, dove N1 è il nodo che ha S come stato.

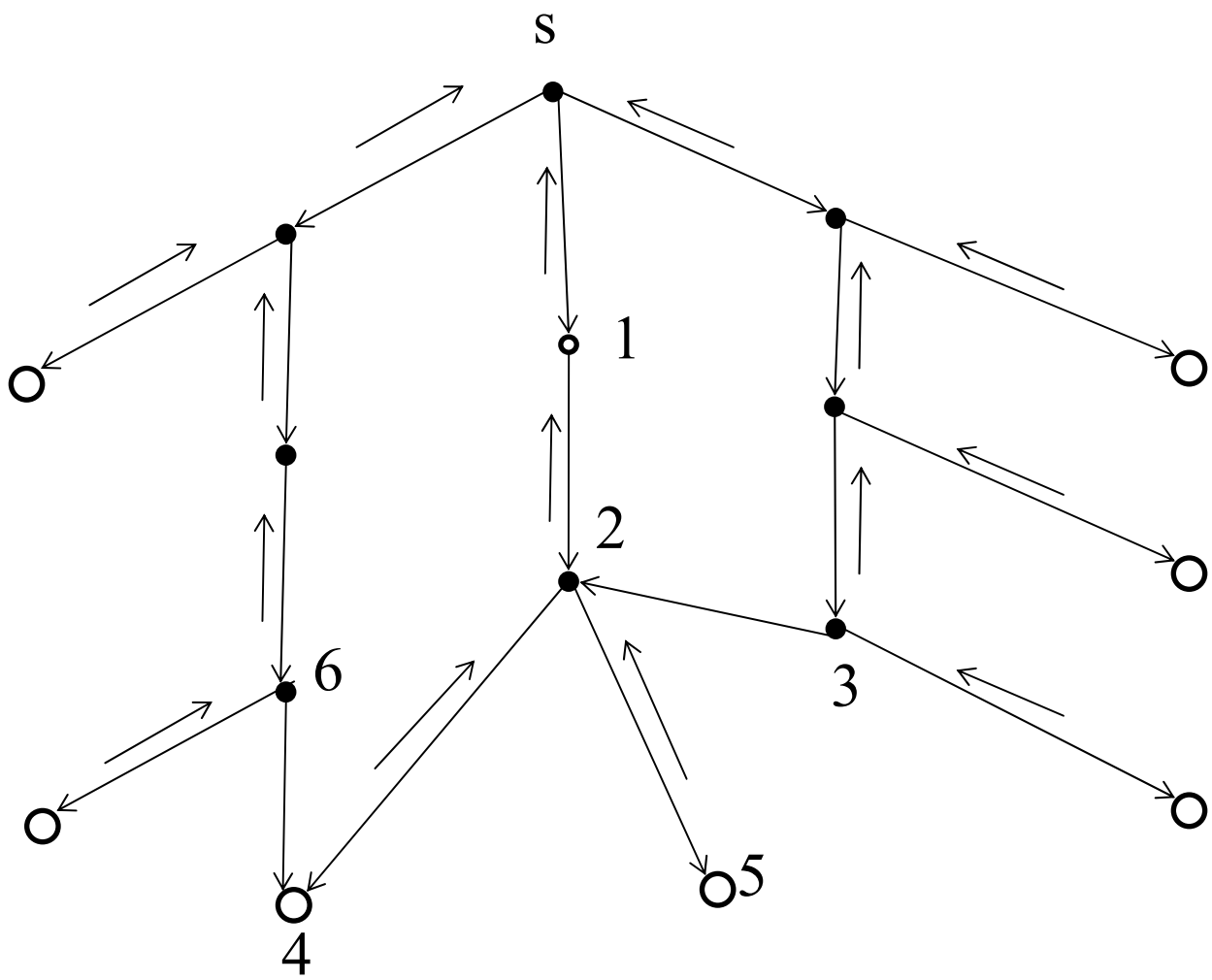
"espandi-grafo (N1, N2, G)": l'arco <N1, N2, true> viene aggiunto a G.

"propaga (N, V, G)": percorre il grafo a partire da N, verificando se per i discendenti è necessario (sulla base del nuovo valore di N, e cioè V) modificare gli archi (v. figura che segue).

Esempio: l'albero di ricerca *prima* di espandere il nodo 1:



Esempio: l'albero di ricerca *dopo* aver espanso il nodo 1:



per il nodo 4 si è trovato un percorso migliore che passa per il nodo 2.

Algoritmo di ricerca ottimo

Obiettivo: definire una funzione di valutazione che massimizza una misura di efficienza e insieme garantisce un cammino di costo minimo fino alla meta.

Si vuole definire una funzione $\hat{f}$ tale che il suo valore nel nodo n, $\hat{f}(n)$ sia una stima del costo di un cammino a costo minimo *vincolato a passare per il nodo n* (così il nodo di OPEN con il valore di $\hat{f}$ più piccolo è il nodo che, trovandosi su un cammino a costo minimo, dovrà essere espanso per primo).

- Sia $k(n_i, n_j)$ il costo *effettivo* di un cammino a costo minimo tra n_i e n_j.

- Se T è un insieme di stati finali, chiameremo $h(n_i)$ il costo di un cammino a costo minimo da n_i ad uno di questi stati finali, definito come:

$$h_i = \min_{n_j \in T} k\left(n_i, n_j\right)$$

- Un cammino che passa per n_i e ha costo $h(n_i)$ è detto *ottimo*.

- Sia ancora $g(n) = k(s, n)$ il costo di un cammino ottimo dal nodo di partenza s e il generico nodo n.

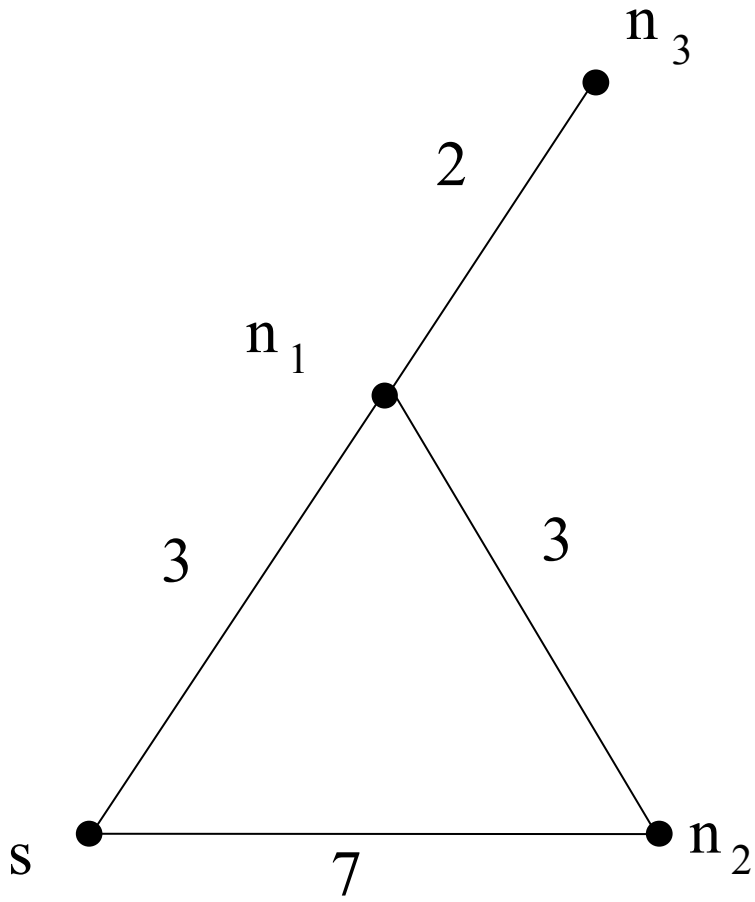
Allora:

$f(n) = g(n) + h(n)$ è il costo di un cammino ottimo vincolato a passare per il nodo *n*. (NB $f(s) = h(s)$ è il costo effettivo di un cammino ottimo (non vincolato) da *s* alla meta).

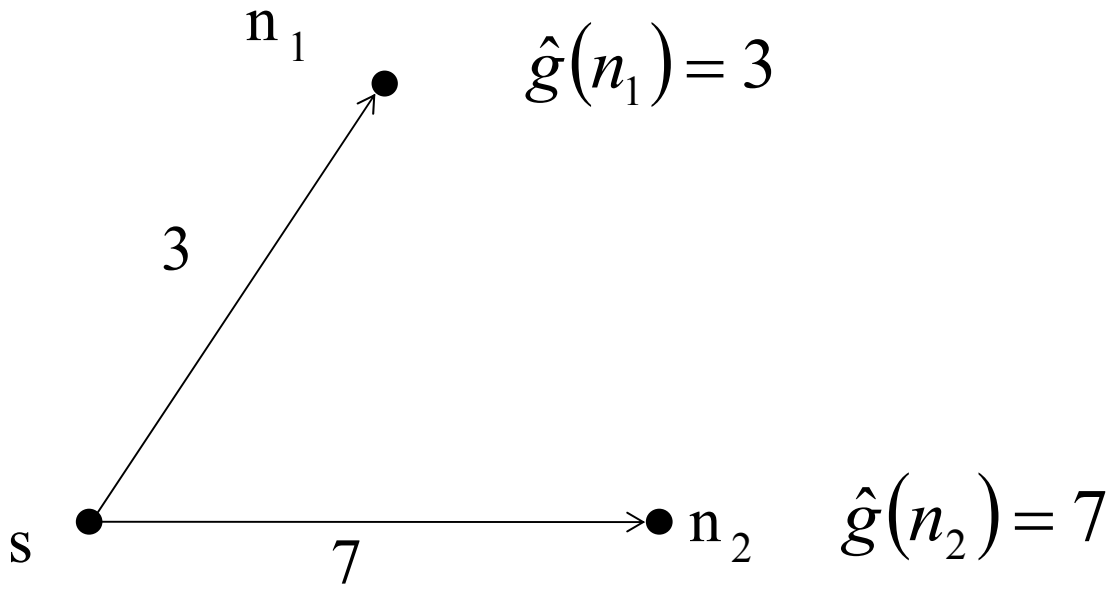
Vogliamo che $\hat{f}$ sia una valutazione (stima) di *f*, e supporremo che $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$.

Ora per $\hat{g}(n)$ si può usare il costo del cammino finora percorso (sommando i costi di tutti gli archi incontrati). Quindi $\hat{g}(n) \geq g(n)$

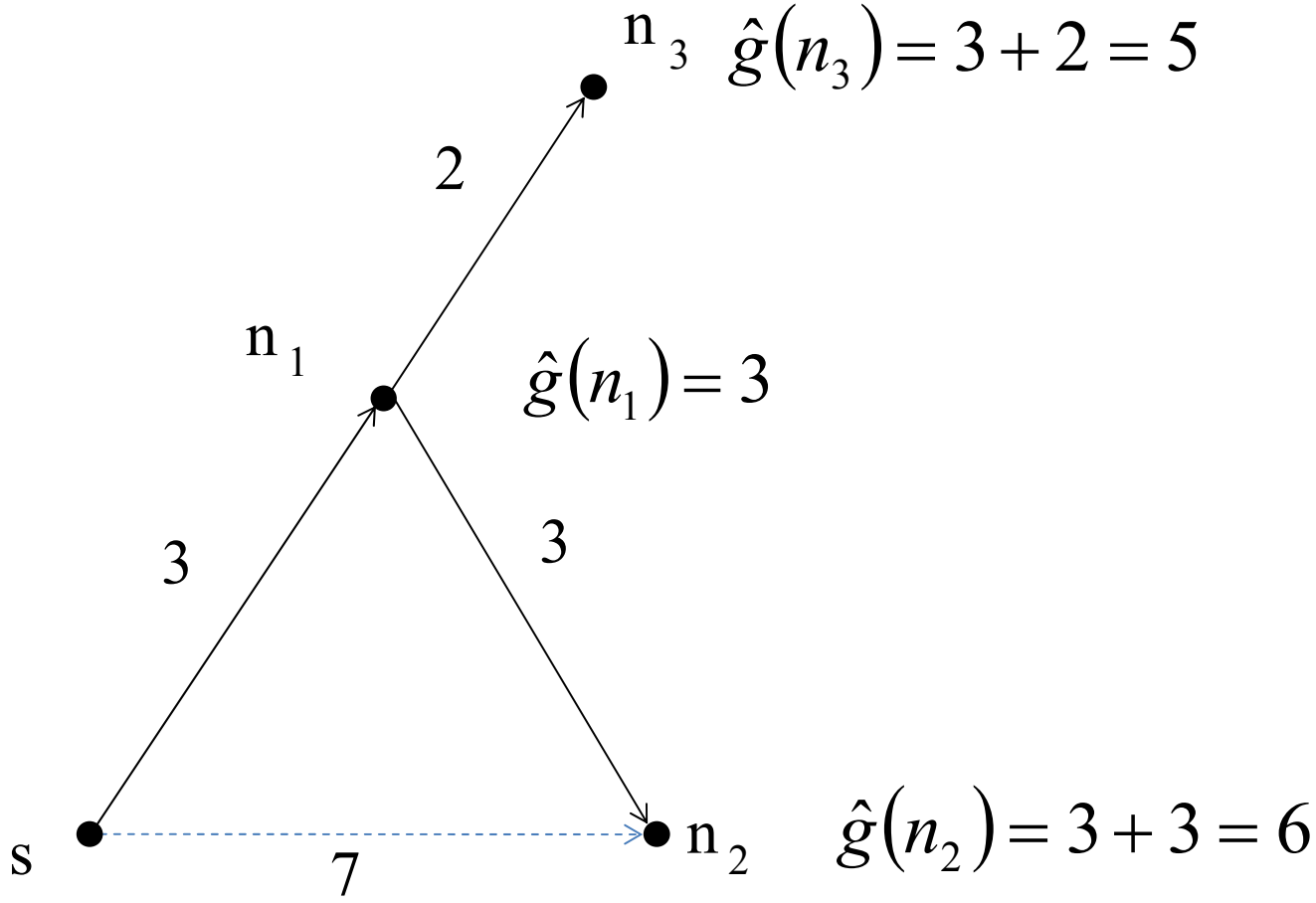
Esempio:



Al passo 1)



Al passo 2) (l'algoritmo espande n_1)



Si è trovato un cammino meno costoso per n_2 , e quindi $\hat{g}(n_2)$ viene diminuito (passa da 7 a 6).

Per la stima $\hat{h}(n)$ di $h(n)$ occorre utilizzare tutta la conoscenza euristica disponibile nel dominio del problema (per esempio può essere la funzione $w(n)$ utilizzata in precedenza).

Supponiamo di usare $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$ come funzione di valutazione e chiameremo A^* l'algoritmo di ricerca ordinata che ne fa uso. Si vuol dimostrare che se $\hat{h}(n)$ è una stima per difetto di h, A^* trova un cammino ottimo.

N.B. Se $\hat{h} \equiv 0$, l'algoritmo è quello a costo uniforme. Poiché $\hat{h} \equiv 0$ è sicuramente una stima per difetto di *h*, dimostrando che A^* trova un cammino ottimo si dimostra che lo trova anche l'algoritmo a costo uniforme.

Ammissibilità di A^*

Un algoritmo di ricerca è ***ammissibile*** se per ogni grafo termina sempre dando un cammino ottimo, quando questo esiste.

Si vuole dimostrare che se $\hat{h}$ è una stima per difetto di h, A^* è ammissibile.

Strategia della dimostrazione:

- provare che prima della terminazione di A^* c'è sempre un nodo di OPEN che sta su un cammino ottimo ed ha un valore di $\hat{f} \leq f(s)$, costo di un cammino ottimo (*necessità*);

- provare l'ammissibilità di A^*, che discende dal fatto che il nodo di OPEN con il più piccolo valore di $\hat{f}$ non può mai essere un nodo finale (e quindi far terminare l'algoritmo) finché non venga trovato un nodo finale con un valore di $\hat{f}$ pari a $f(s)$(*sufficienza*).

Lemma: se $\hat{h}$ è una stima per difetto di *h* allora prima della terminazione di A^* ogni cammino ottimo possiede un nodo aperto (cioè in OPEN) con $\hat{f}$ non superiore al costo $f(s)$ di un cammino ottimo.

Formalmente:

se $\hat{h}(n) \leq h(n)$ per ogni *n*, ad ogni istante prima della terminazione di A^* e per ogni cammino ottimo P dal nodo *s* ad un nodo finale esiste un nodo aperto n' su P con $\hat{f}(n') \leq f(s)$ (cioè è *condizione necessaria*).

Dimostrazione:

Il cammino ottimo P sia rappresentato da $n_0 (= s), n_1, n_2, ..., n_k$. n_k è il nodo finale.

Sia n' il primo nodo aperto della successione (ce n'è almeno uno, poiché se n_k è chiuso, A^* è terminato).

Dalla definizione: $\hat{f}(n') = \hat{g}(n') + \hat{h}(n')$

Ora A^* ha trovato un cammino ottimo fino a n', poiché questo è in P e tutti i suoi antecedenti su P sono chiusi.

Pertanto:

$$\hat{g}(n') = g(n')$$

e

$$\hat{f}(n') = \hat{g}(n') + \hat{h}(n') = g(n') + \hat{h}(n')$$

Avendo ipotizzato $\hat{h}(n) \leq h(n)$:

$$\hat{f}(n') \leq g(n') + h(n') = f(n') = f(s)$$

Infatti f in corrispondenza di ogni nodo di un cammino ottimo è $f(s)$, il costo minimo, e quindi $\hat{f}(n') \leq f(s)$ come affermato dal lemma.

Dimostreremo ora che A^* è ammissibile.

Teorema: se $\hat{h}(n) \leq h(n)$ per tutti i nodi di n e se tutti i costi degli archi sono maggiori di un numero δ piccolo a piacere, A^* è ammissibile (cioè $\hat{h}(n) \leq h(n)$ è *condizione sufficiente*).

Si dimostra per assurdo, assumendo cioè che A^* non termini sempre trovando un cammino ottimo. Si considerano tre casi:

Promemoria: l'algoritmo A (che diventa A*)

Algoritmo di ricerca ordinata (per alberi e grafi):

1. porre il nodo di partenza s in una lista di nome OPEN e calcolare $\hat{f}(s)$;

2. se OPEN è vuota, uscire con fallimento; altrimenti continuare;

3. rimuovere da OPEN il nodo con il più piccolo valore di $\hat{f}$ e porlo in una lista di nome CLOSED, chiamandolo *n* (eventuali conflitti fra diversi valori minimi di $\hat{f}$ si possono risolvere arbitrariamente, dando comunque la precedenza a un eventuale nodo finale);

4. se *n* è un nodo finale uscire con il cammino risolutivo ottenuto percorrendo i puntatori all'indietro; altrimenti continuare;

5. espandere il nodo *n* generando tutti i suoi successori. Se non ci sono successori andare subito a 2. Per ogni successore n_i calcolare $\hat{f}(n_i)$;

6. associare ai successori che ancora non si trovano né in OPEN né in CLOSED i rispettivi valori di $\hat{f}$ appena calcolati, creare i puntatori a *n* e inserirli in OPEN;

7. associare ai successori che già si trovassero in OPEN o in CLOSED il più piccolo fra il valore di appena calcolato e quello precedente. Porre in OPEN i successori che si trovavano in CLOSED per cui è diminuito il valore di $\hat{f}$, e ridirigere a *n* i puntatori di tutti i nodi il cui valore di $\hat{f}$ è diminuito;

8. andare al passo 2.

CASO 1: *l'algoritmo termina senza trovare uno stato finale .*

L'algoritmo *A* termina al passo 4 con un successo se trova uno stato finale.

L'unica altra possibilità di terminazione con fallimento (passo 2) si può avere solo se OPEN è vuota.

Ma per il lemma precedente, OPEN non si può svuotare prima della terminazione se esiste un cammino da *s* all'obiettivo.

CASO 2: *l'algoritmo non termina.*

Sia t un nodo finale e sia $f(s)$ il costo minimo associato. Ogni arco ha un costo pari almeno a $\delta(>0)$. Per un nodo *n* che dista da s di più di $M = f(s)/\delta$ vale che $\hat{f}(n) \geq \hat{g}(n) \geq g(n) \geq M\delta (= f(s))$ e, per il lemma visto, non potrà mai essere espanso.

Infatti per il lemma ci sarà un nodo aperto n' su un cammino ottimo tale che $\hat{f}(n') \leq f(s) < \hat{f}(n)$ e pertanto A^* sceglierà n' invece di *n* al passo 3.

C’è ancora da considerare la continua riapertura dei nodi che si trovano a meno di *M* passi da *s*, come nel passo 7 (presenza di cicli).

Si dimostra (ma è intuitivo) che non si può superare un numero massimo di riaperture di un nodo, perché vi è un numero finito di cammini da *s* a *n* che tocchi solo nodi a meno di *M* passi da *s*.

CASO 3: *L'algoritmo termina in corrispondenza di un nodo finale senza trovare un cammino ottimo.*

Supponiamo che A^* termini in modo finale t con $\hat{f}(t) = \hat{g}(t) > f(s)$. Per il lemma visto esisteva subito prima della terminazione un nodo aperto n' su un cammino ottimo con $\hat{f}(n') \leq f(s) < \hat{f}(t)$. Quindi n' sarebbe stato scelto per l'espansione il nodo invece del nodo t, contraddicendo l'ipotesi che A^* terminasse.

Definizione:

Si dice che l'algoritmo A è *più informato* dell'algoritmo di B se la conoscenza euristica utilizzata da A permette di calcolare una stima per difetto di $h(n)$ che è sempre strettamente maggiore (per tutti i nodi n non finali) di quella permessa della conoscenza euristica utilizzata da B.

Teoremi (si tralascia la dimostrazione):

- Se A è "*più informato*" di B, allora B espanderà almeno tanti nodi quanti ne espande A.

- Se la "*restrizione di monoticità*" (se n_k è un successore di n_i, allora $h(n_i) - h(n_k) \leq c(n_i, n_k)$) è soddisfatta, allora quando A^* seleziona un nodo per l'espansione ha già trovato un cammino ottimo fino ad esso.

L'importanza di $\hat{g}$

Se interessa una soluzione qualunque, indipendentemente dal costo (ma non indipendentemente dallo sforzo di ricerca !), vi sono ragioni per includere $\hat{g}$ e anche per non includerla.

Ragioni per includerla:

Se $\hat{g}$ è inclusa in $\hat{f}$, si è sicuri che un cammino venga trovato, prima o poi. $\hat{g}$ in pratica aggiunge alla ricerca una componente in ampiezza, mentre $\hat{h}$ tende a contribuire per la profondità.

Ragioni per non includerla:

tendenzialmente non interessa il costo dei cammini costruiti fino ad ora, proprio perché lo si è già fatto. Interessa piuttosto lo sforzo necessario per arrivare alla fine.

Anche se ambedue le cose sembrano ragionevoli, la prima è più corretta, anche se non sempre dimostrabile analiticamente.

Conclusioni

L'algoritmo A^* fa parte del metodo di ricerca detto **best-first**.

Si rammenta che in questo metodo il nodo da espandere è selezionato tra *tutti* i nodi inespansi esistenti, senza riguardo a dove si trovano nel grafo di ricerca.

Invece:

- nella ricerca in ampiezza, il nodo da espandere è scelto *solo* tra i nodi inespansi allo stesso livello di profondità;

- nella ricerca in profondità, il nodo da espandere è scelto *solo* tra i nodi inespansi che sono fratelli.

Nel metodo best-first, se si pone

$$h(x) = 0,\ f(x) = g(x).$$

La ricerca è detta a *costo uniforme* (o *incurred cost search*).

Se si pone

$$g(x) = 0,\ f(x) = h(x).$$

La ricerca è detta *greedy* (o *predicted cost search*).