



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Strategie di ricerca non informate

Federico Chesani

DISI

Department of Informatics – Science and Engineering

Cercare soluzioni

- L'idea è quella di mantenere ed estendere un insieme di sequenze/soluzioni parziali.
 - Un agente con diverse opzioni immediate di esito sconosciuto può decidere cosa fare esaminando prima le differenti sequenze possibili di azioni che conducono a stati di esito conosciuto scegliendo, poi, quella migliore.
 - Il processo di cercare tale sequenza è chiamato **RICERCA**.
 - È utile pensare al processo di ricerca come la costruzione di un albero di ricerca i cui nodi sono stati e i cui rami sono operatori.
- Un algoritmo di ricerca prende come input un problema e restituisce una soluzione nella forma di una sequenza di azioni.
- Una volta che viene trovata la soluzione, le azioni suggerite possono essere realizzate.
 - Questa fase è chiamata **ESECUZIONE**.



CERCARE SOLUZIONI

Generare sequenze di azioni.

- *Espansione*: si parte da uno stato e applicando gli operatori (o la funzione successore) si generano nuovi stati.
- *Strategia di ricerca*: ad ogni passo scegliere quale stato espandere.
- *Albero di ricerca*: rappresenta l'espansione degli stati a partire dallo stato iniziale (la radice dell'albero).
- Le foglie dell'albero rappresentano gli stati da espandere.



Alberi di ricerca

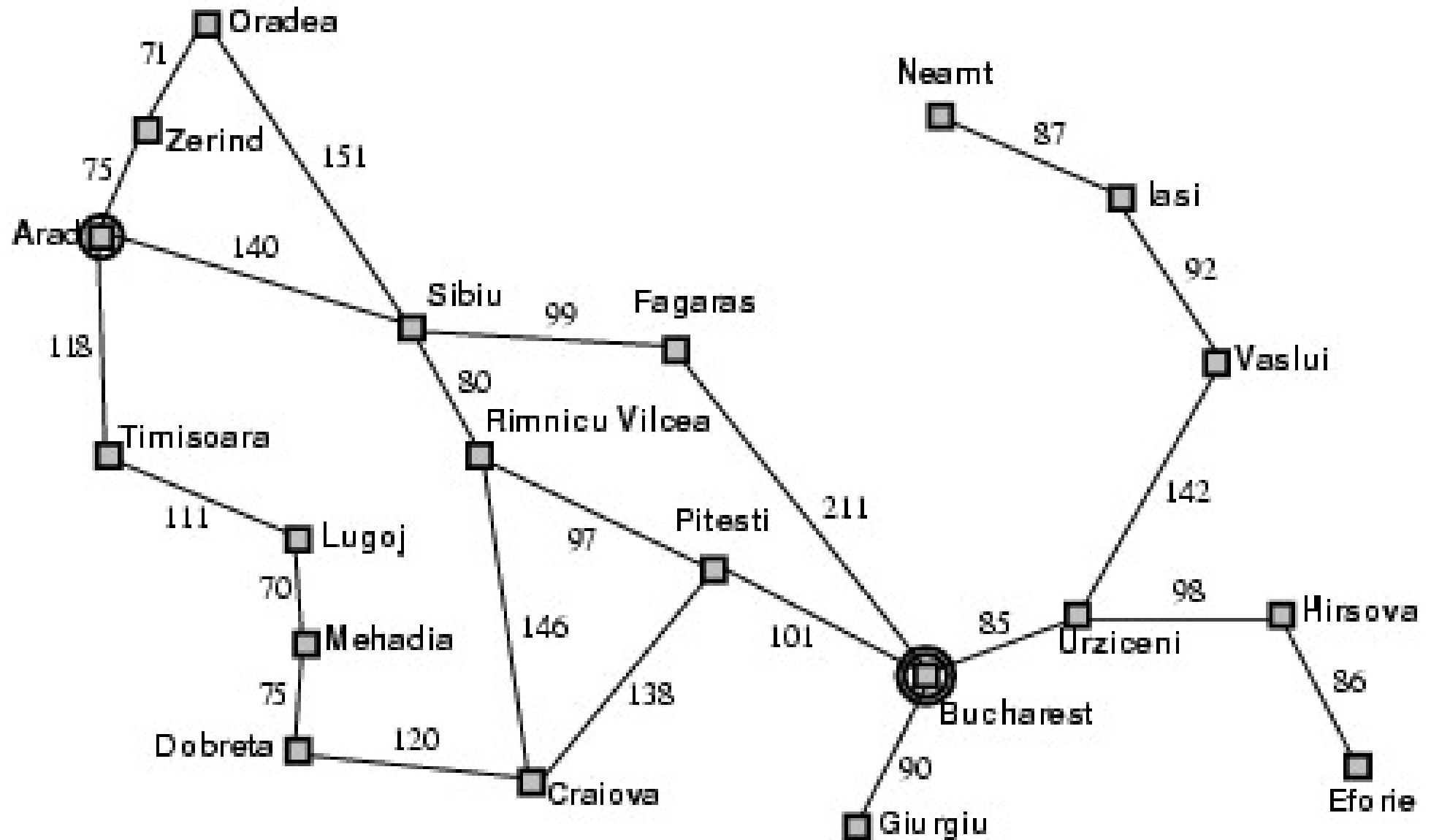
- Idea base:
 - esplorazione, fuori linea, simulata, dello spazio degli stati generando successori di stati già esplorati.

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

Esempio: Romania

- Una vacanza in Romania; attualmente in Arad.
- I voli partono da Bucharest domani
- goal:
 - Essere in Bucharest
- problema:
 - stati: varie città
 - azioni: guida fra le città
- soluzione:
 - Sequenza di città, ad es., Arad, Sibiu, Fagaras, Bucharest.

Esempio: Romania



Da: S. Russell & P. Norvig: "Intelligenza Artificiale: un approccio moderno", Pearson ed.

Formulazione del problema

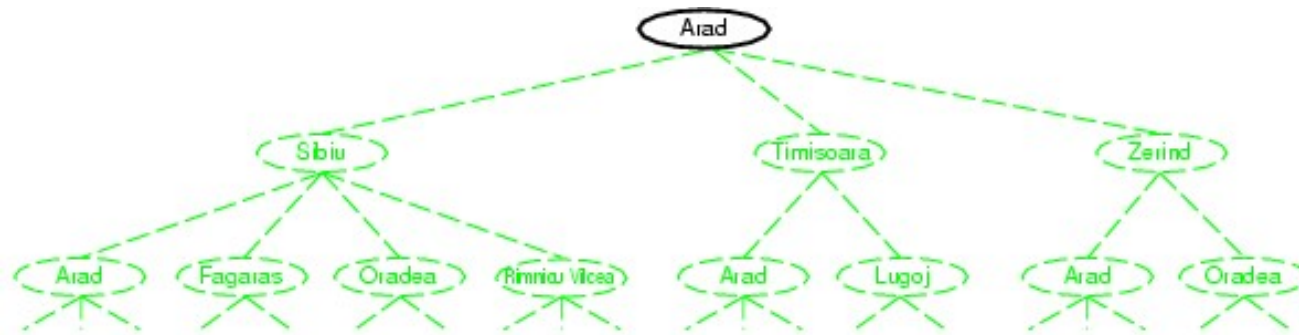
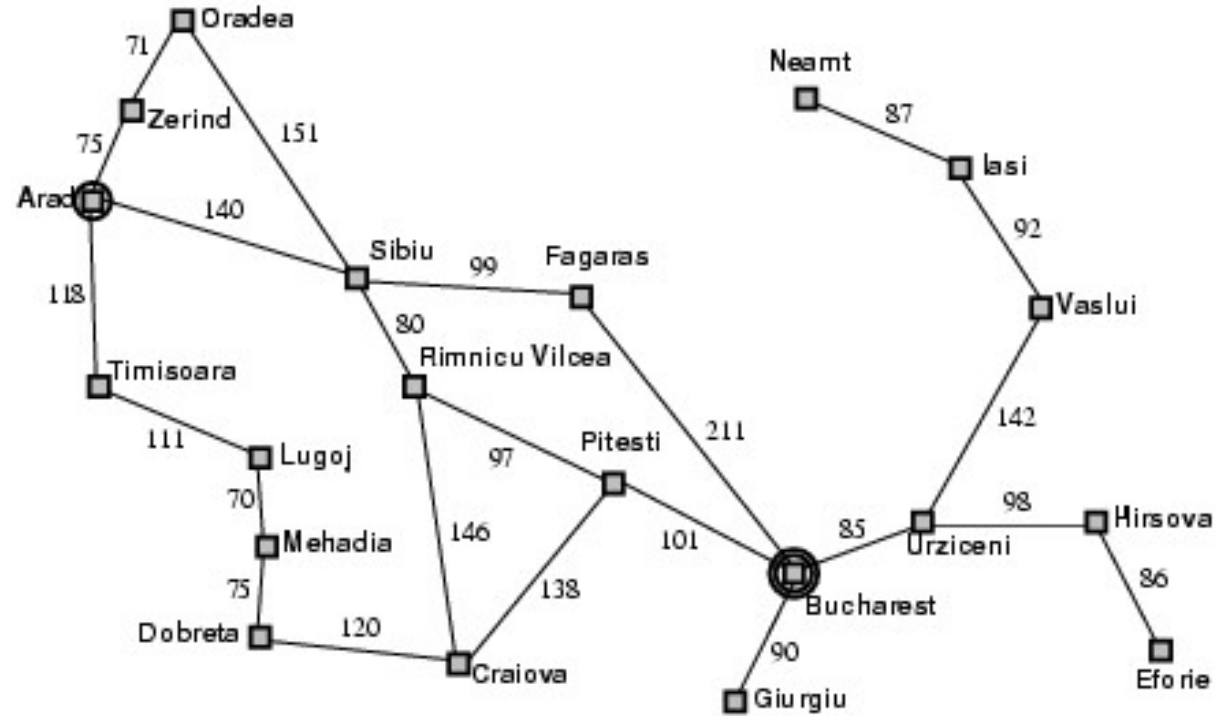
Il **problema** è definito da quattro punti:

1. **Stato iniziale** es., "at Arad"
2. **Azioni o funzioni successore** $S(x)$ = insieme di coppie azione-stato
 - es., $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots \}$
3. **goal test**, può essere
 - **esplicito**, es., $x = \text{"at Bucharest"}$
 - **implicito**, es., $\text{controllamappa}(x)$
4. **Costo della strada** es., somma delle distanze, numero di azioni eseguite ecc. $c(x, a, y) \geq 0$

Una **soluzione** è una sequenza di azioni che portano dallo stato iniziale al goal.



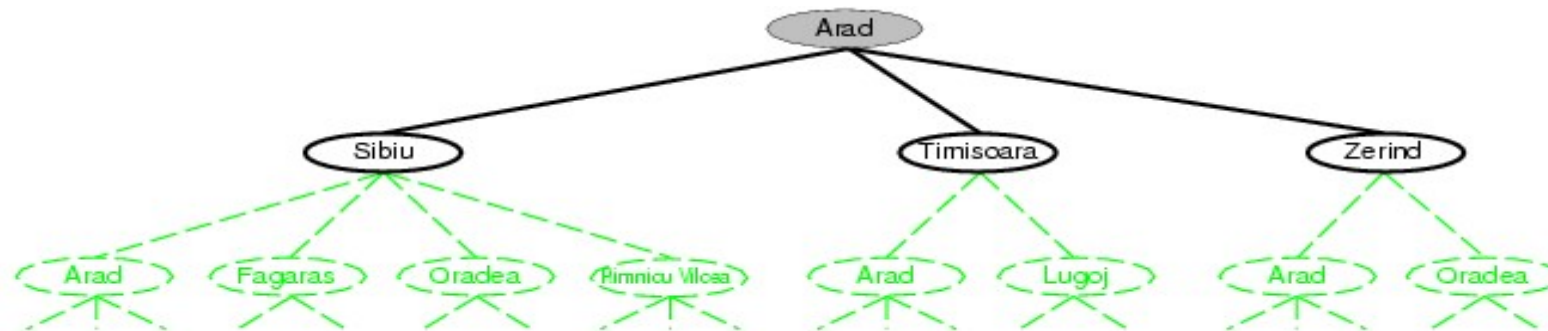
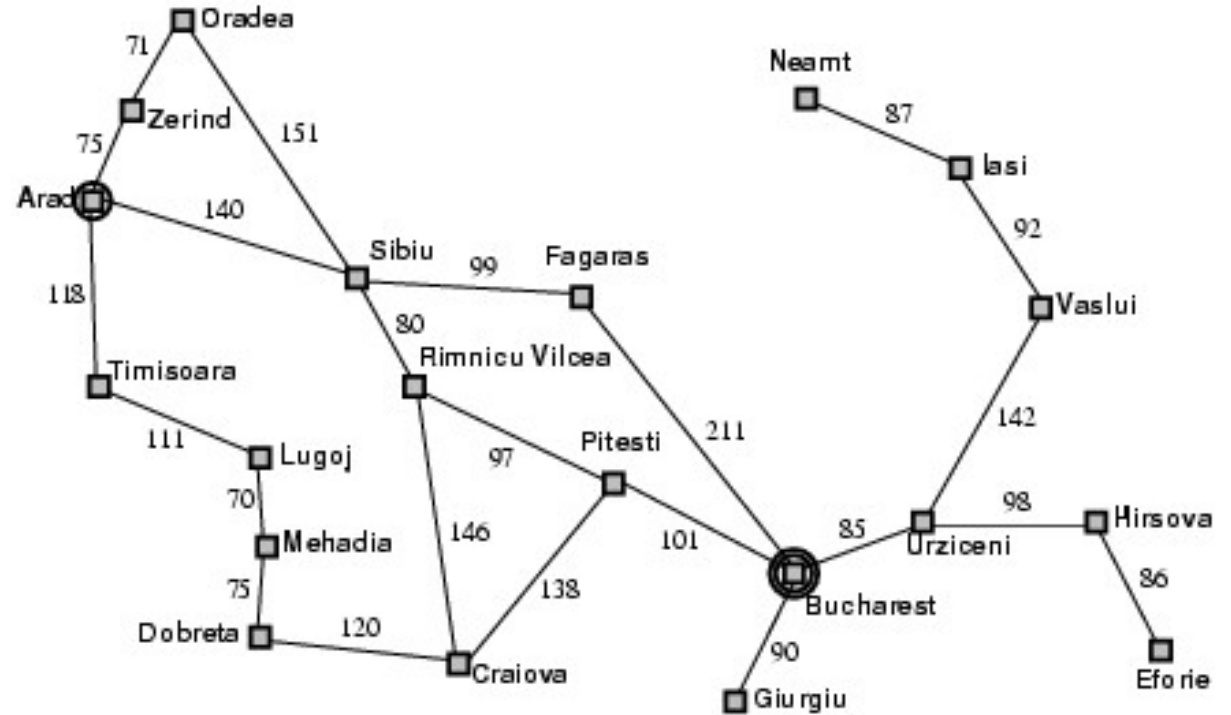
Albero di ricerca



Da: S. Russell & P. Norvig: "Intelligenza Artificiale: un approccio moderno", Pearson ed.



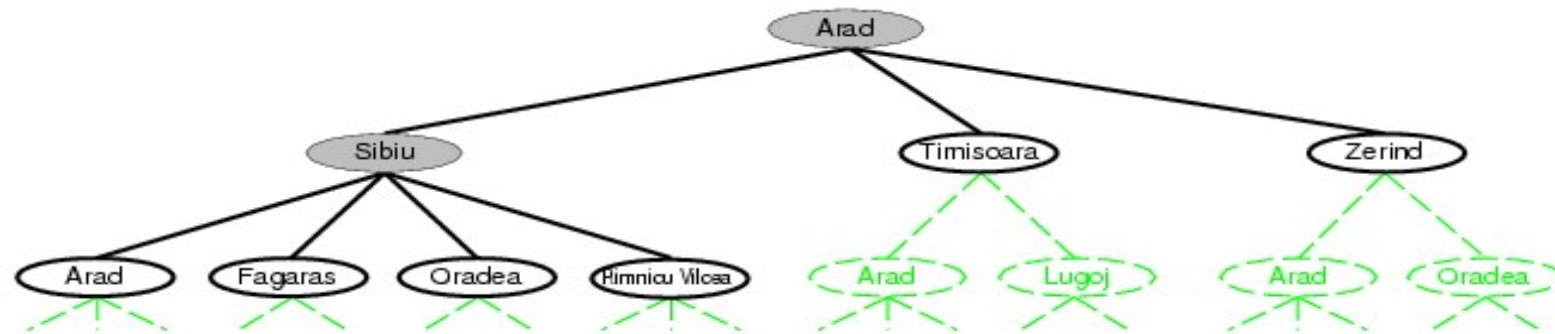
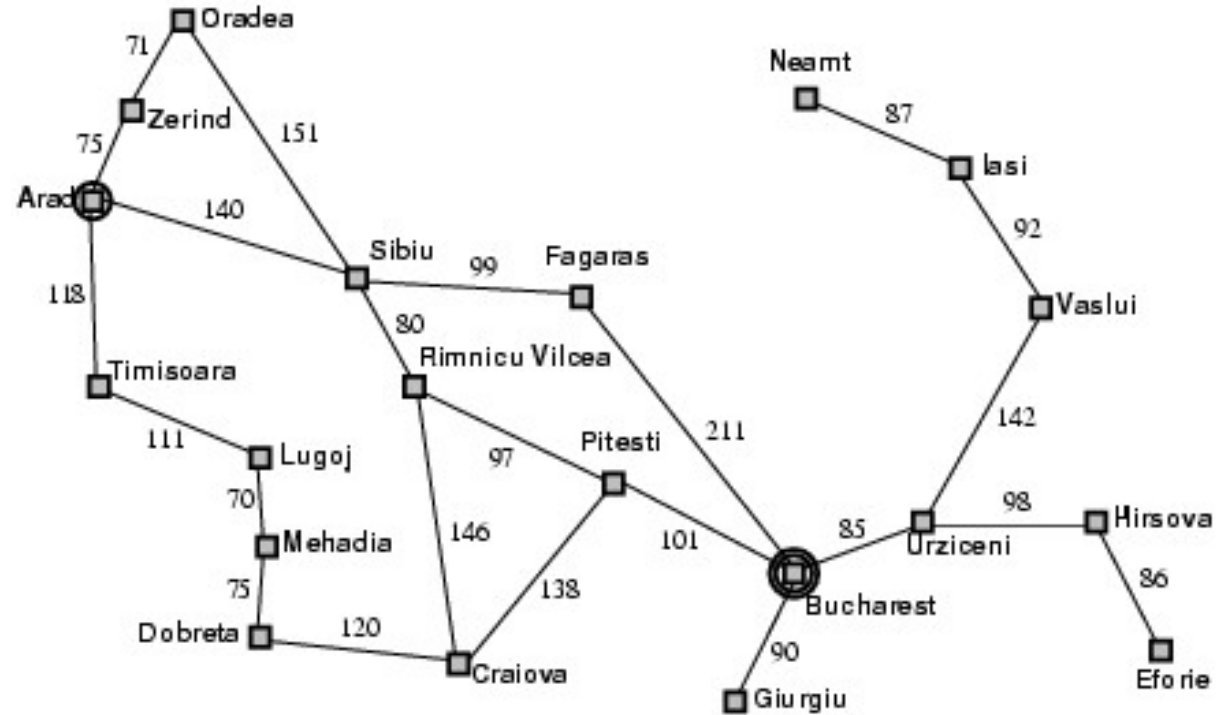
Albero di ricerca



Da: S. Russell & P. Norvig: "Intelligenza Artificiale: un approccio moderno", Pearson ed.



Albero di ricerca

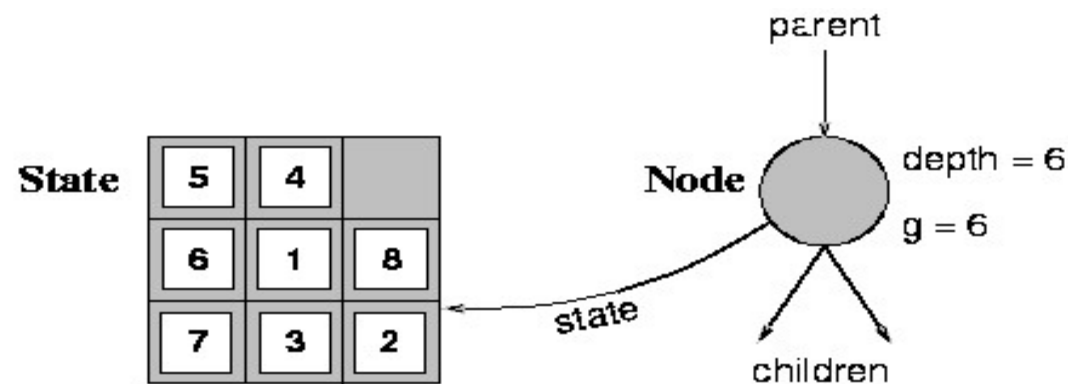


Da: S. Russell & P. Norvig: "Intelligenza Artificiale: un approccio moderno", Pearson ed.



Strutture dati per l'albero di ricerca

- Il nodo contiene:
 - Lo **stato** nello spazio degli stati a cui il nodo corrisponde.
 - Il **nodo genitore**.
 - L'**operatore** che è stato applicato per ottenere il nodo.
 - La **profondità** del nodo.
 - Il **costo del cammino** dallo stato iniziale al nodo



Implementazione

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
  successors  $\leftarrow$  the empty set
  for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
    s  $\leftarrow$  a new NODE
    PARENT-NODE[s]  $\leftarrow$  node; ACTION[s]  $\leftarrow$  action; STATE[s]  $\leftarrow$  result
    PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s]  $\leftarrow$  DEPTH[node] + 1
    add s to successors
  return successors
```

L' EFFICACIA DELLA RICERCA

- Si riesce a trovare una soluzione?
- È una buona soluzione? (con basso costo di cammino – costo in linea)
- Qual è il costo della ricerca? (tempo per trovare una soluzione – costo fuori linea)
- Costo totale di ricerca = costo di cammino + costo di ricerca.
- Scegliere stati e azioni → L' importanza dell'astrazione



ESEMPIO: IL GIOCO DEL 8

- Stati: posizione di ciascuna delle tessere;
- Operatori: lo spazio vuoto si sposta a destra, a sinistra, in alto e in basso;
- Test obiettivo: descrizione dello stato finale;
- Costo di cammino: ciascun passo costa 1.



ESEMPIO: IL GIOCO DEL 8

Stato iniziale

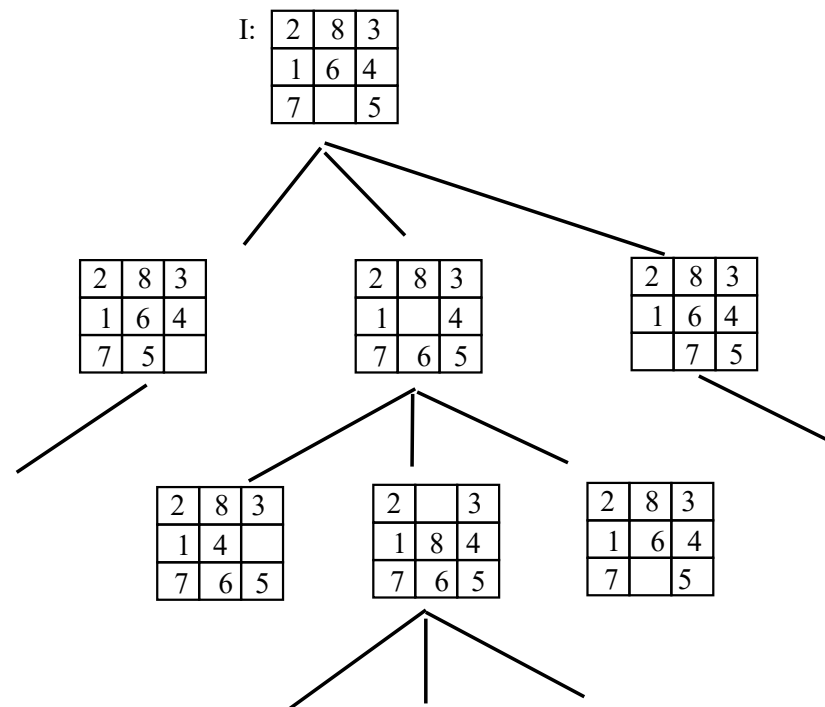
Goal

I:

2	8	3
1	6	4
7		5

G:

1	2	3
8		4
7	6	5



STRATEGIE DI RICERCA

- I problemi che i sistemi basati sulla conoscenza devono risolvere sono non-deterministici (don't know)
- In un certo istante più azioni possono essere svolte (azioni: applicazioni di operatori)
- **STRATEGIA**: è un'informazione sulla conoscenza che sarà applicata potendone invocare molteplici. Due possibilità
 - Non utilizzare alcuna conoscenza sul dominio: applicare regole in modo arbitrario (strategie non-informate) e fare una ricerca ESAUSTIVA.
 - Impraticabile per problemi di una certa complessità.



STRATEGIE DI RICERCA

- La strategia di controllo deve allora utilizzare **CONOSCENZA EURISTICA** sul problema per la selezione degli operatori applicabili
- Le strategie che usano tale conoscenza si dicono **STRATEGIE INFORMATE**
- ESTREMO:
 - La conoscenza sulla strategia è così completa da selezionare ogni volta la regola CORRETTA
 - REGIME IRREVOCABILE (ALTRIMENTI PER TENTATIVI)



STRATEGIE DI RICERCA

- La scelta di quale stato espandere nell'albero di ricerca prende il nome di **strategia**.
- Abbiamo **strategie informate** (o euristiche) e **non informate** (blind).
- Le strategie si valutano in base a quattro criteri:
 - **Completezza**: la strategia garantisce di trovare una soluzione quando ne esiste una?
 - **Complessità temporale**: quanto tempo occorre per trovare una soluzione?
 - **Complessità spaziale**: Quanta memoria occorre per effettuare la ricerca?
 - **Ottimalità**: la strategia trova la soluzione di "qualità massima" quando ci sono più soluzioni?



STRATEGIE DI RICERCA

- STRATEGIE DI RICERCA NON-INFORMATE:
 - **breadth-first** (a costo uniforme);
 - **depth-first**;
 - **depth-first a profondità limitata**;
 - **ad approfondimento iterativo**.



L'algoritmo generale di ricerca

```
function GENERAL-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

L'algoritmo generale di ricerca

function GENERAL-SEARCH(*problem*, QUEUING-FN) **returns** a solution, or failure

nodes \leftarrow MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))

loop do

if *nodes* is empty **then return** failure

node \leftarrow REMOVE-FRONT(*nodes*)

if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*

nodes \leftarrow QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))

end

Tramite l'argomento **Queuing-Fn** viene passata una funzione per inserire i nodi ottenuti dall'espansione

BREADTH-FIRST

- Definizione di profondità:
 - La **PROFONDITÀ** del nodo da cui si parte è uguale a 0; la profondità di un qualunque altro nodo è la profondità del genitore più 1.
- **ESPANDE sempre i nodi MENO PROFONDI dell'albero.**
- Nel caso peggiore, se abbiamo profondità **d** e fattore di ramificazione **b** il numero massimo di nodi espansi nel caso peggiore sarà **b^d** . (complessità temporale).
 - $1 + b + b^2 + b^3 + \dots + (b^d - 1) \rightarrow b^d$



BREADTH-FIRST

- All'ultimo livello sottraiamo 1 perché il goal non viene ulteriormente espanso.
- Questo valore coincide anche con la complessità spaziale (numero di nodi che manteniamo contemporaneamente in memoria).
- L'esplorazione dell'albero avviene tenendo CONTEMPORANEAMENTE aperte più strade.
- Tale strategia garantisce la COMPLETEZZA, ma NON permette una EFFICIENTE IMPLEMENTAZIONE su sistemi mono-processore (architetture multi-processore).



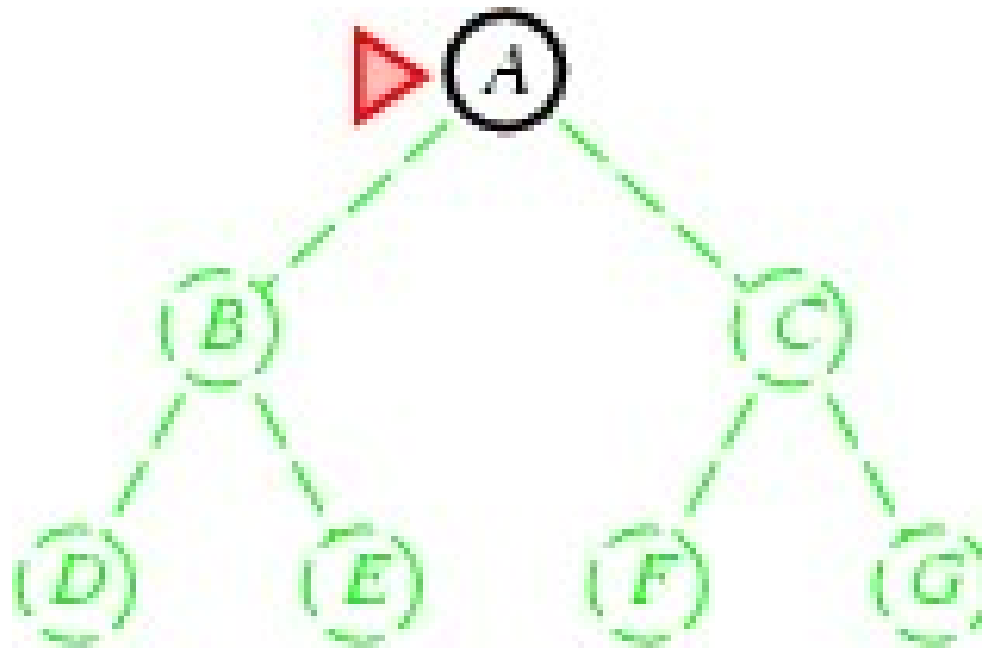
BREADTH-FIRST

- In particolare, con profondità 10 e fattore di ramificazione 10 dovremmo espandere 10^{10} nodi, (tempo 128 giorni e 1 terabyte di memoria immaginando che un nodo richieda 100 byte di memoria e vengano espansi 1000 nodi al secondo).
- Il problema della memoria sembra essere il più grave.
- Trova sempre il cammino a costo minimo se il costo coincide con la profondità (altrimenti dovremmo utilizzare un'altra strategia che espande sempre il nodo a costo minimo → [strategia a costo uniforme](#)).
- La strategia a costo uniforme è [completa](#) e, a differenza della ricerca in ampiezza, ottimale anche quando gli operatori non hanno costi uniformi. (complessità temporale e spaziale uguale a quella in ampiezza).



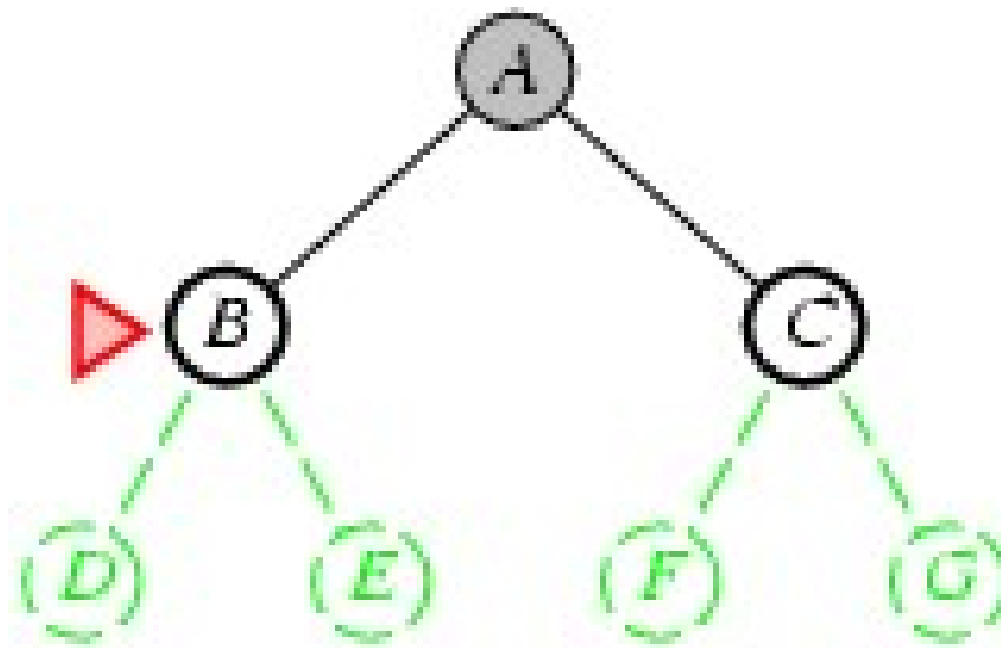
Ricerca Breadth-first

- Espande I nodi a profondita` minore
- Implementazione:
 - *fringe* è una coda FIFO, i.e. successori in fondo.



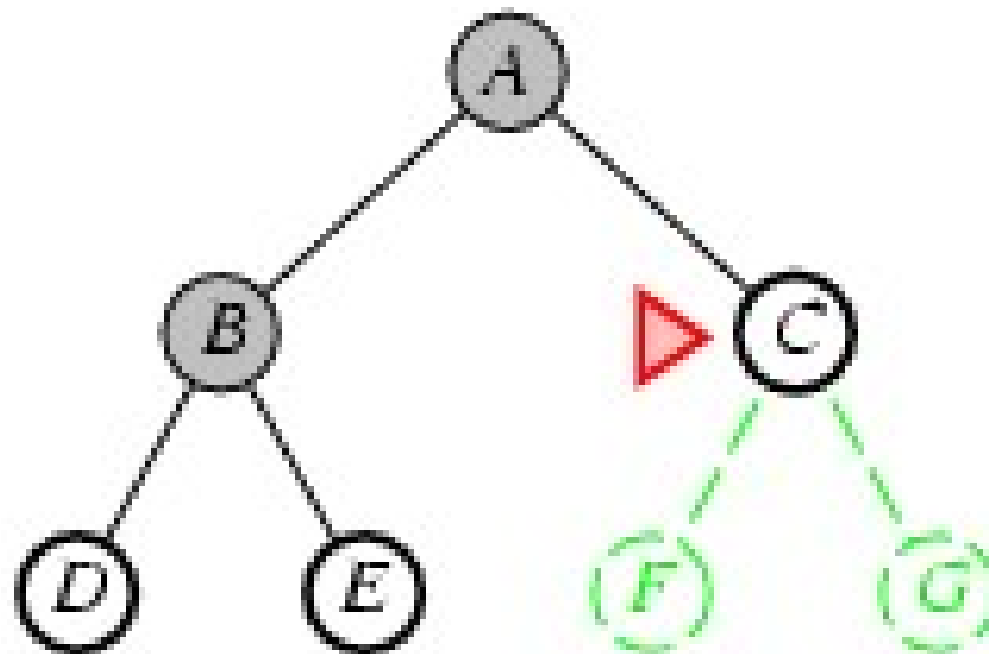
Ricerca Breadth-first

- Espande I nodi a profondita` minore
- Implementazione:
 - *fringe* e' una coda FIFO, i.e. successori in fondo.



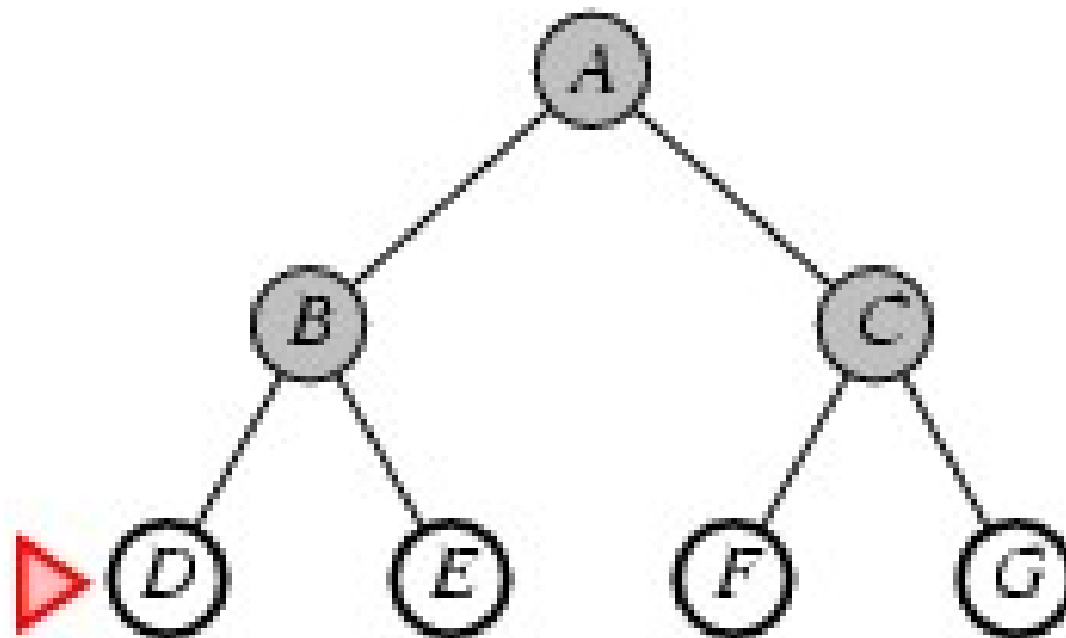
Ricerca Breadth-first

- Espande I nodi a profondita` minore
- Implementazione:
 - *fringe* e' una coda FIFO, i.e. successori in fondo.

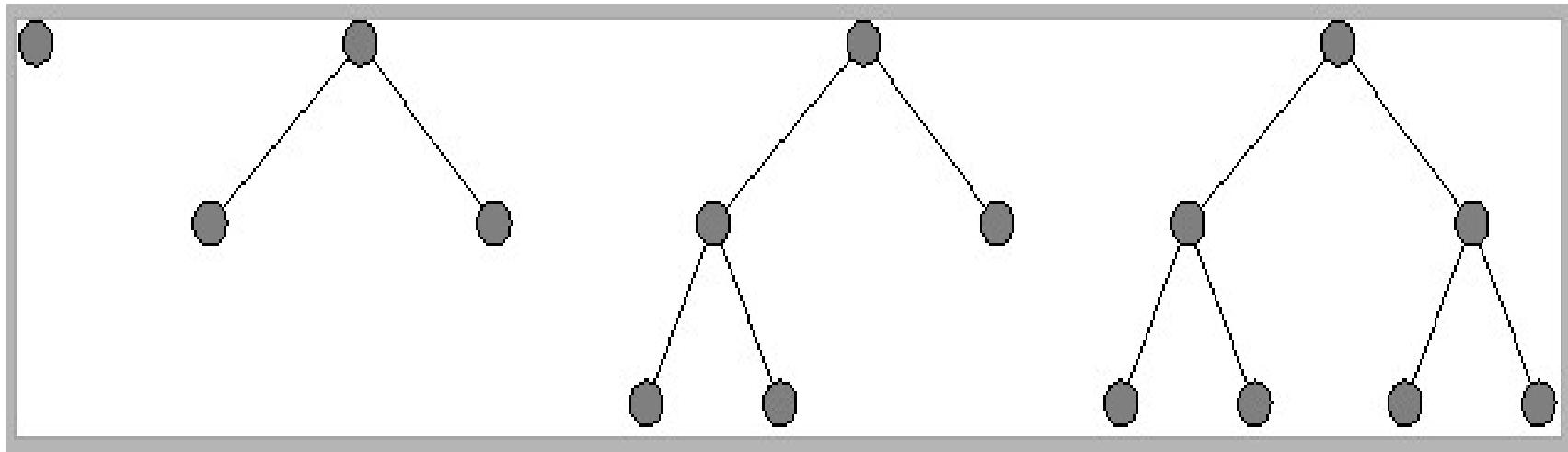


Ricerca Breadth-first

- Espande I nodi a profondita` minore
- Implementazione:
 - *fringe* e' una coda FIFO, i.e. successori in fondo.



Ricerca Breadth-first



QueueingFn mette i successori alla fine della coda



Breadth-first

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$, i.e., exponential in d

Space?? $O(b^d)$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 1MB/sec
so 24hrs = 86GB.

b - massimo fattore di diramazione dell'albero di ricerca

d - profondità della soluzione a costo minimo

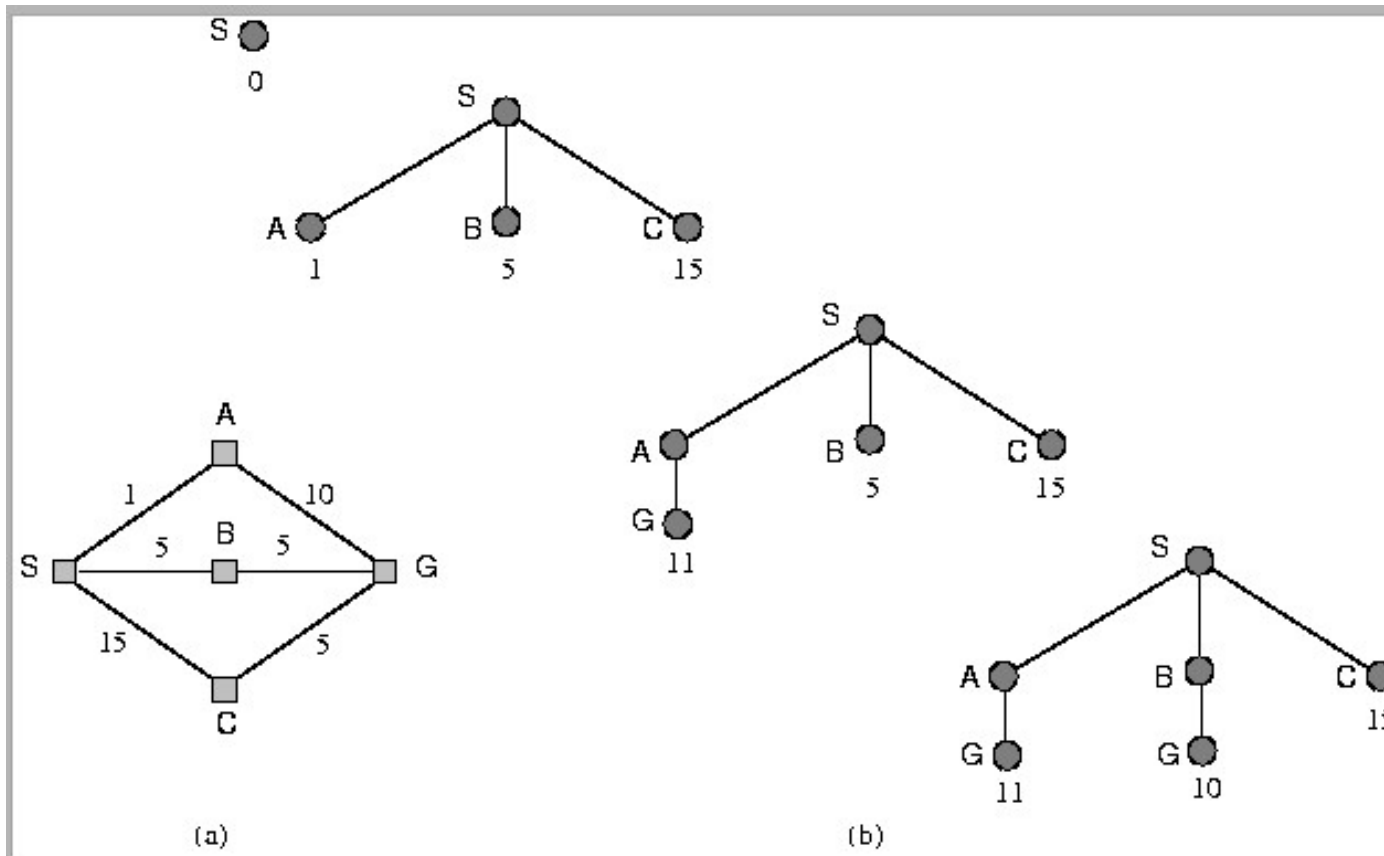
m - massima profondità dello spazio degli stati (può essere infinita)

Ricerca in ampiezza

D e p t h	N o d e s	T i m e	M e m o r y
0	1	1 m illisecond	100 bytes
2	111	1 seconds	11 kilobytes
4	11,111	11 seconds	1 m egabyte
6	10^6	18 m inutes	111 m egabytes
8	10^8	31 hours	11 g igabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Lo svantaggio principale è l'eccessiva occupazione di memoria. Nell'esempio si suppone che il fattore di ramificazione sia $b=10$. Si espandono 1000 nodi/secondo. Ogni nodo occupa 100 byte di memoria.

Ricerca a costo uniforme



Ciascun nodo è etichettato con il **costo $g(n)$**

QueueingFn inserisce i successori in ordine di costo di cammino crescente

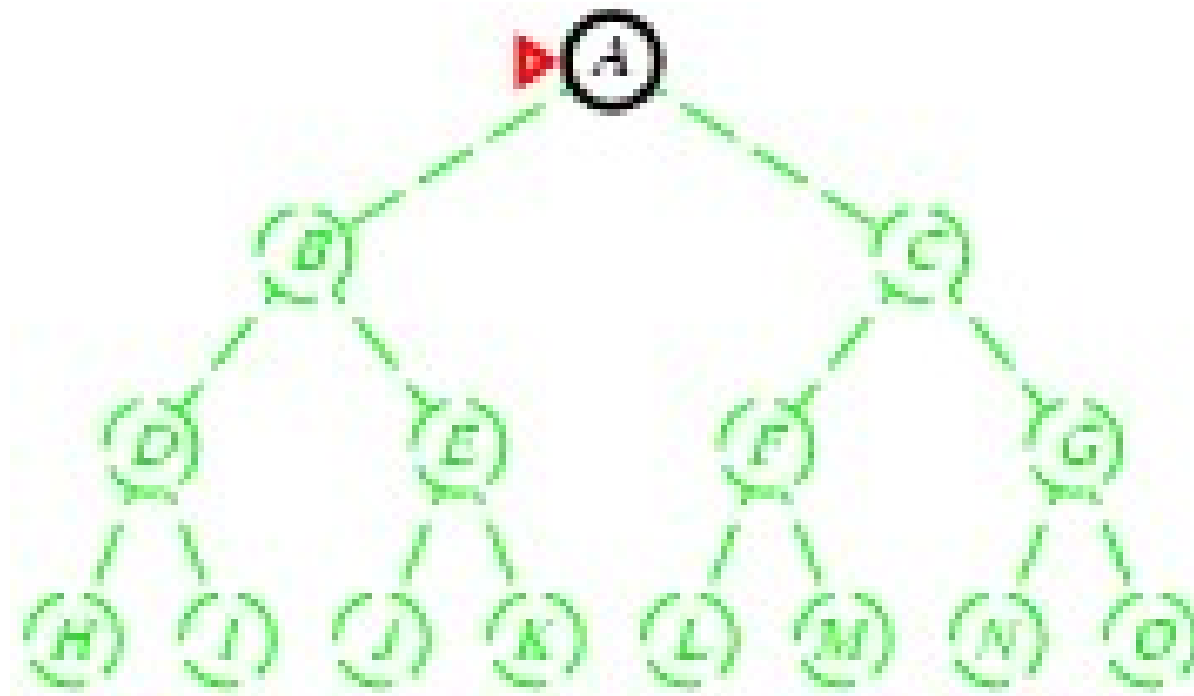
Ricerca Depth-first

- **ESPANDE per primi nodi PIÙ PROFONDI**;
- I nodi di UGUALE PROFONDITÀ vengono selezionati **ARBITRARIAMENTE** (quelli più a sinistra).
- La ricerca in profondità richiede un'occupazione di memoria molto modesta.
- Per uno spazio degli stati con fattore di ramificazione b e profondità massima d la ricerca richiede la memorizzazione di $b*d$ nodi.
- La complessità temporale è invece analoga a quella in ampiezza.
- Nel caso peggiore, se abbiamo profondità d e fattore di ramificazione b il numero massimo di nodi espansi nel caso peggiore sarà b^d . (complessità temporale).
- **EFFICIENTE** dal punto di vista realizzativo: può essere memorizzata una sola strada alla volta (un unico stack)
- Può essere **NON-COMPLETA** con possibili loop in presenza di rami infiniti.(INTERPRETE PROLOG).



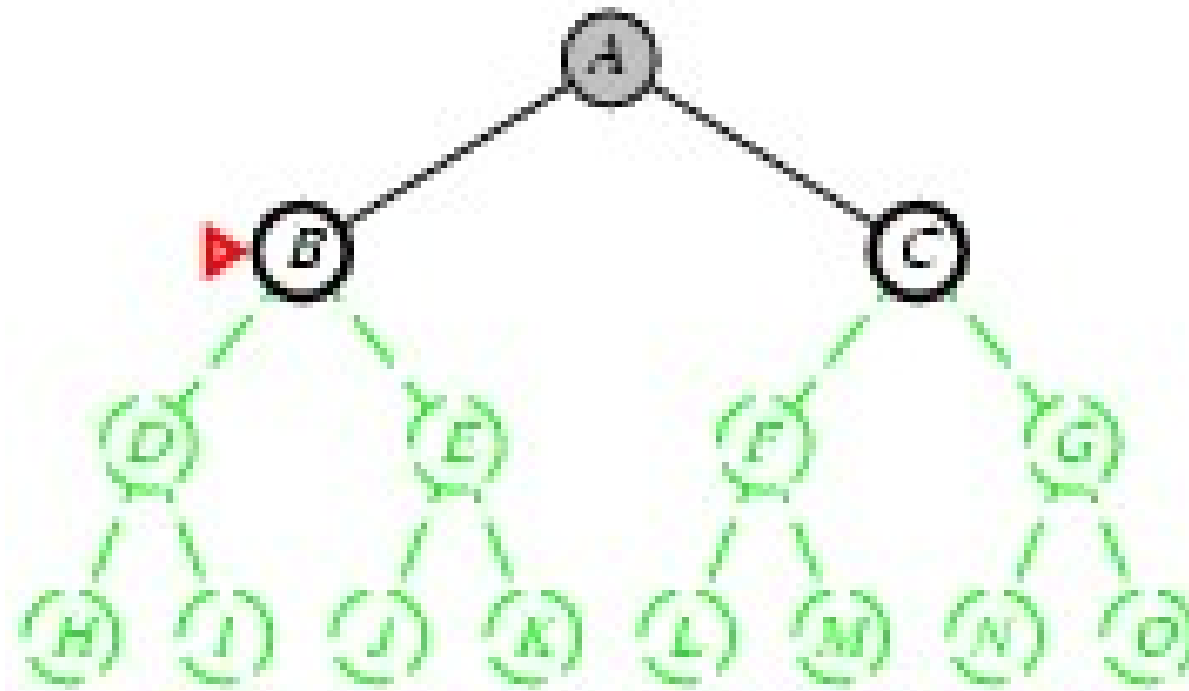
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



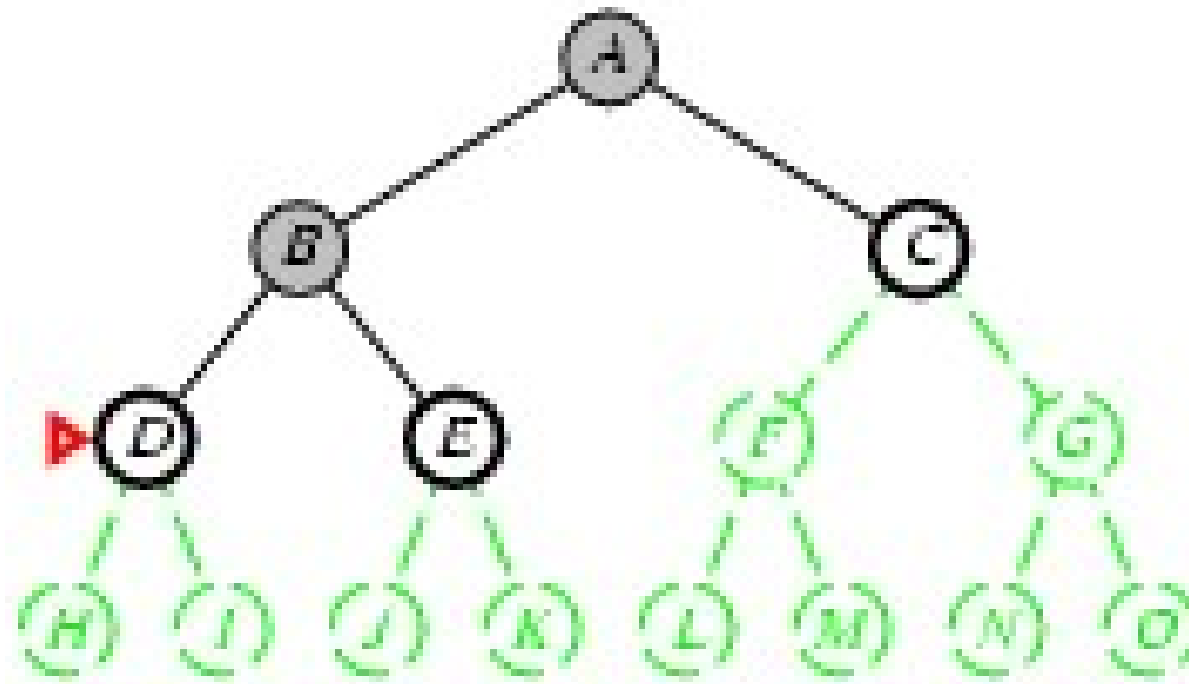
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



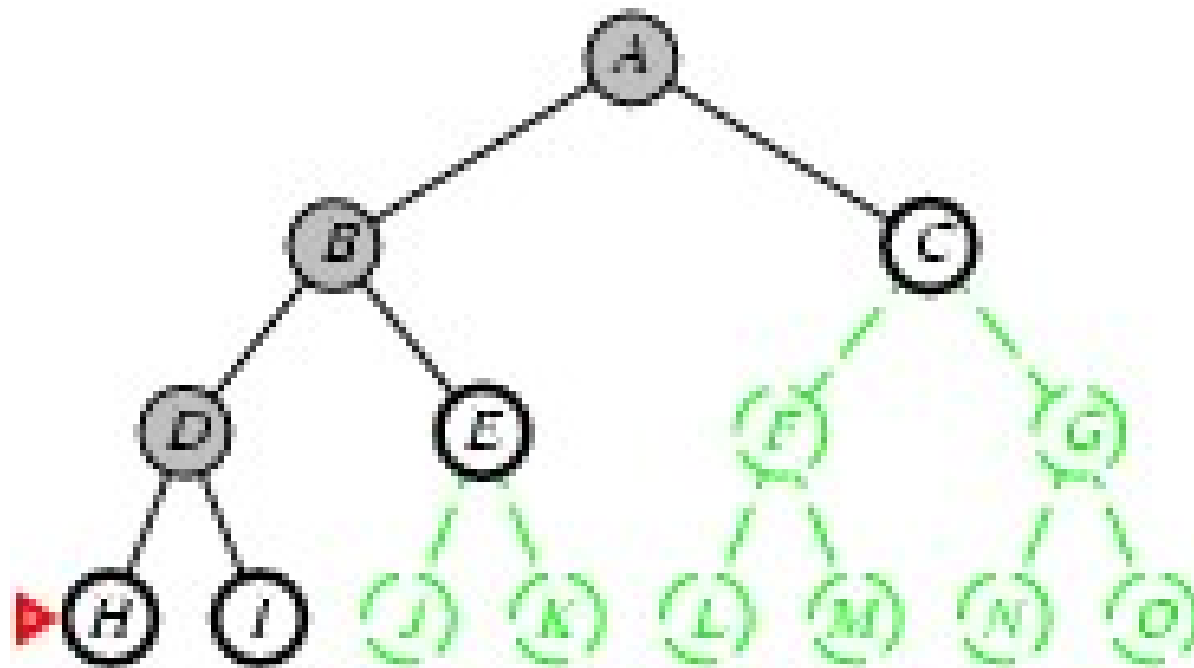
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



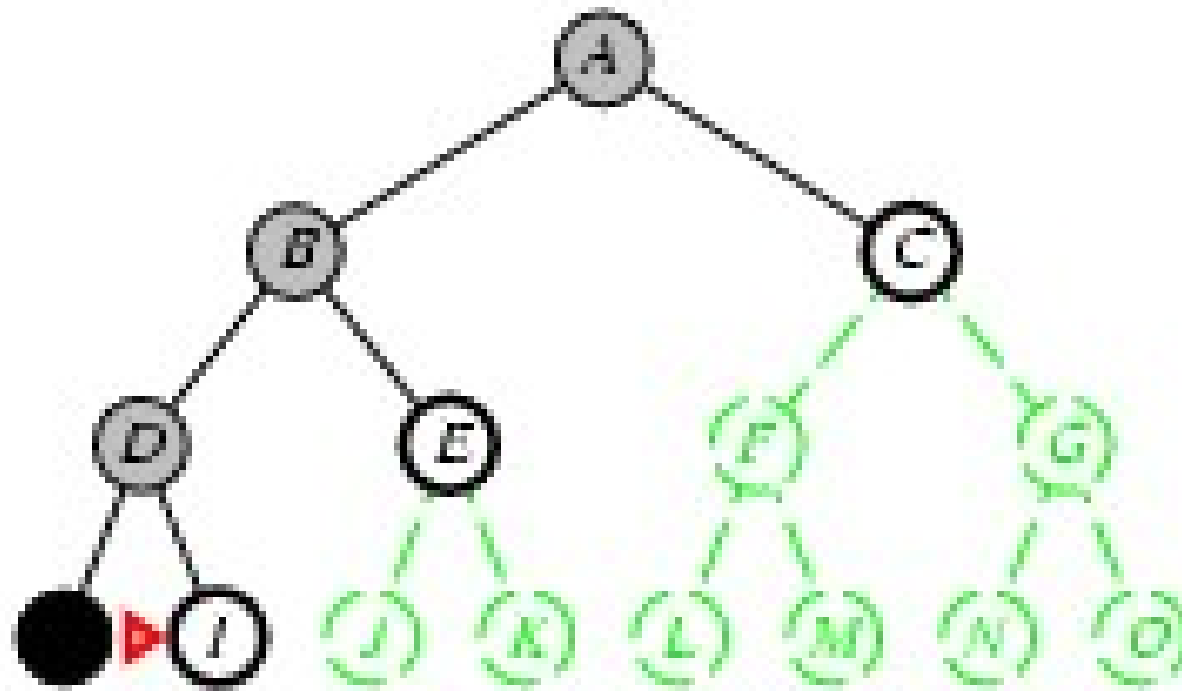
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



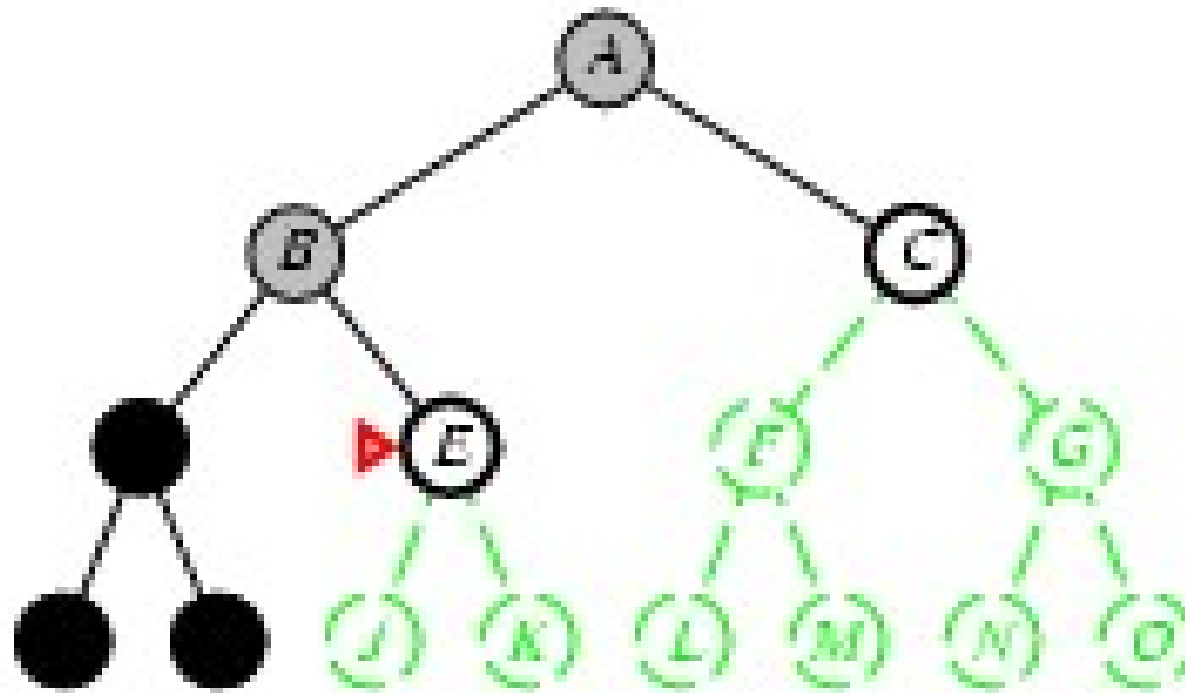
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



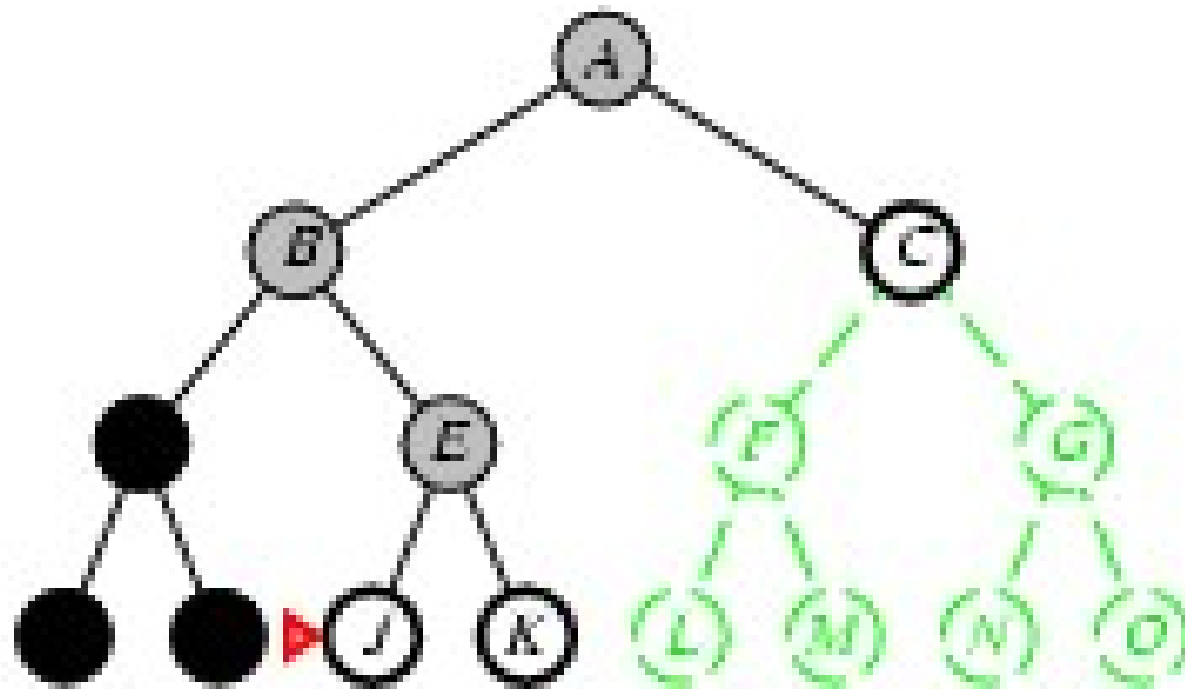
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



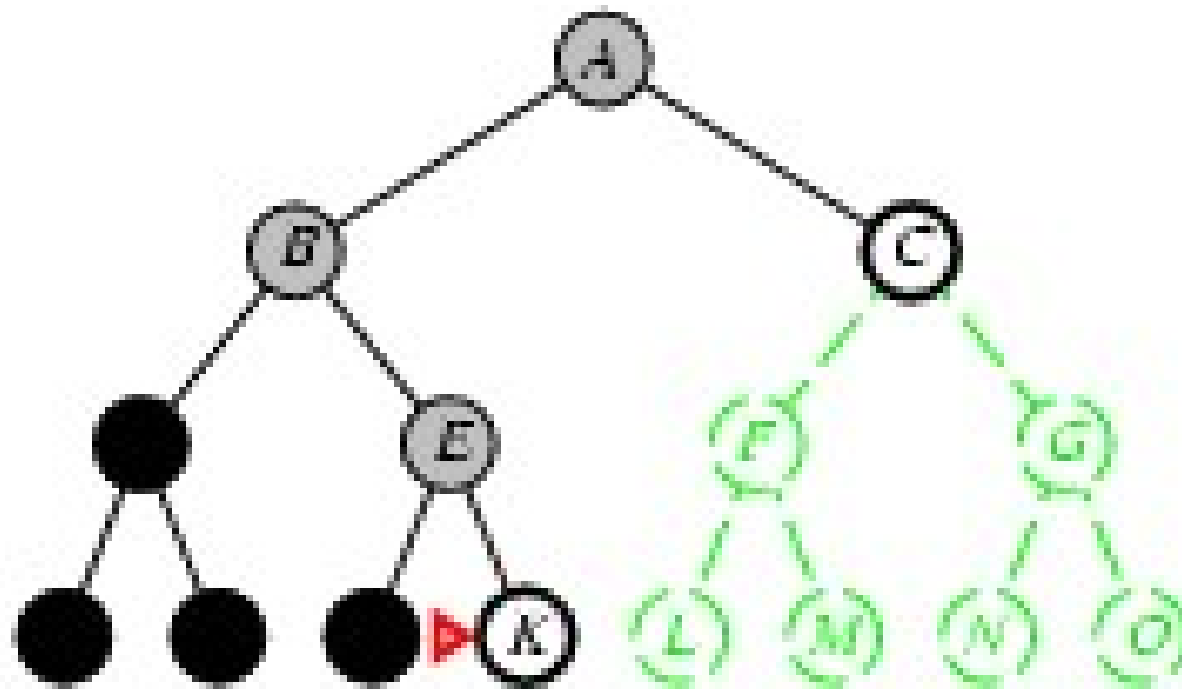
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



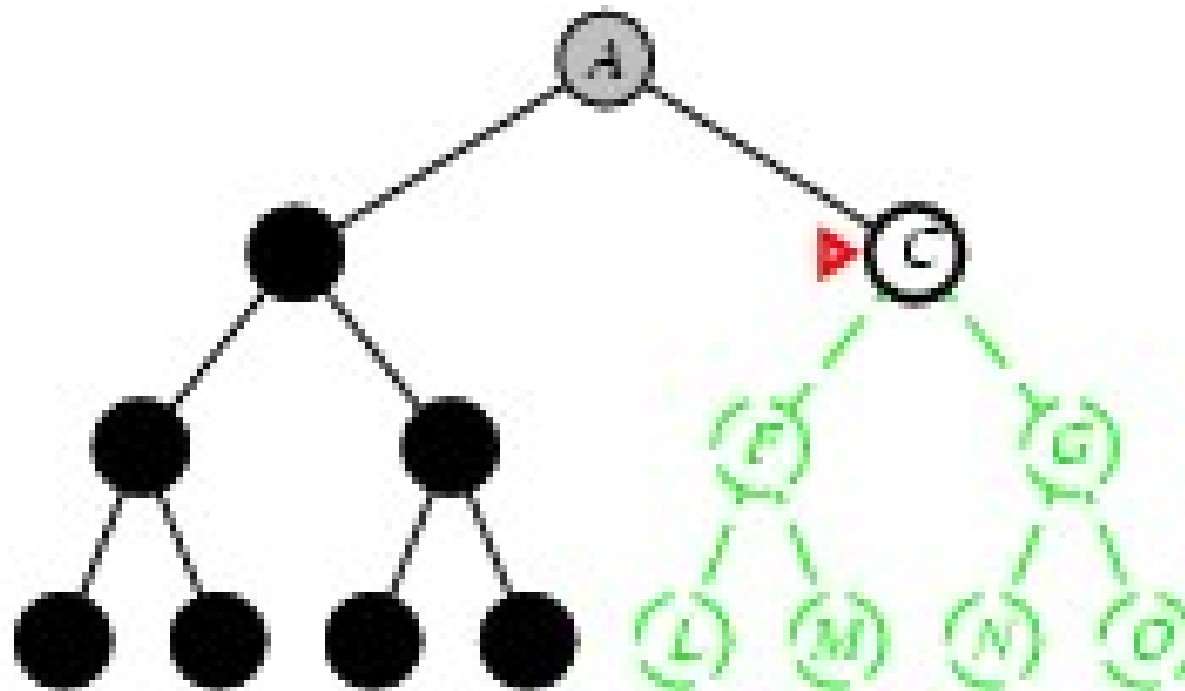
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



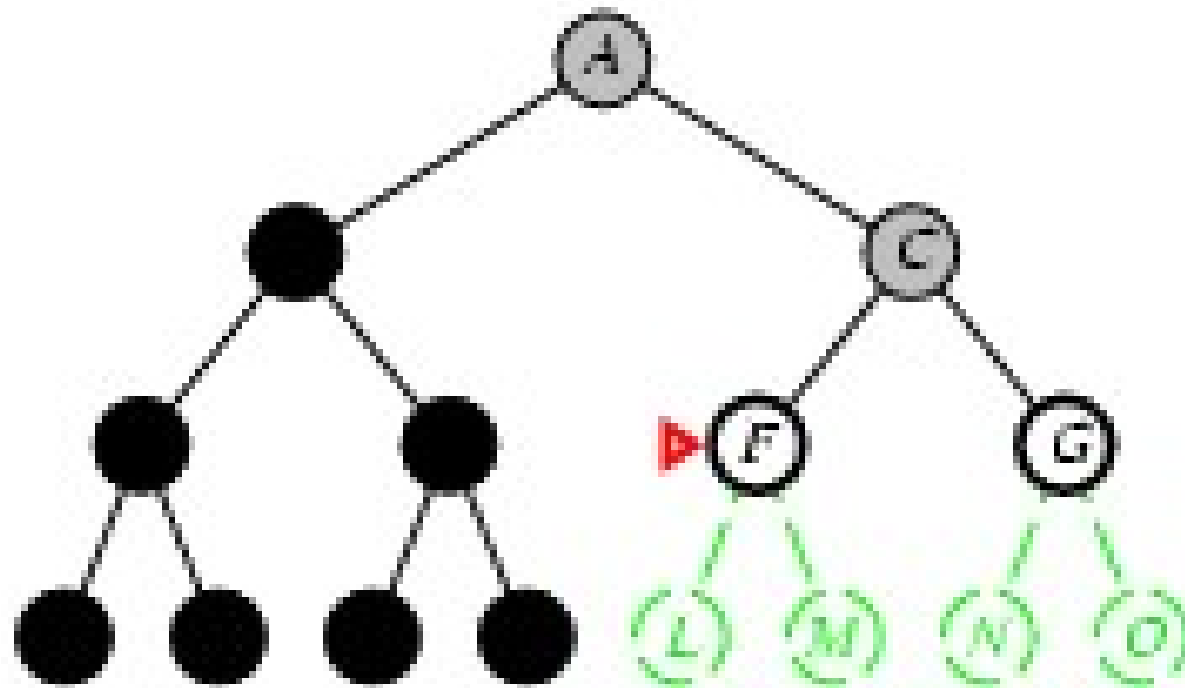
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



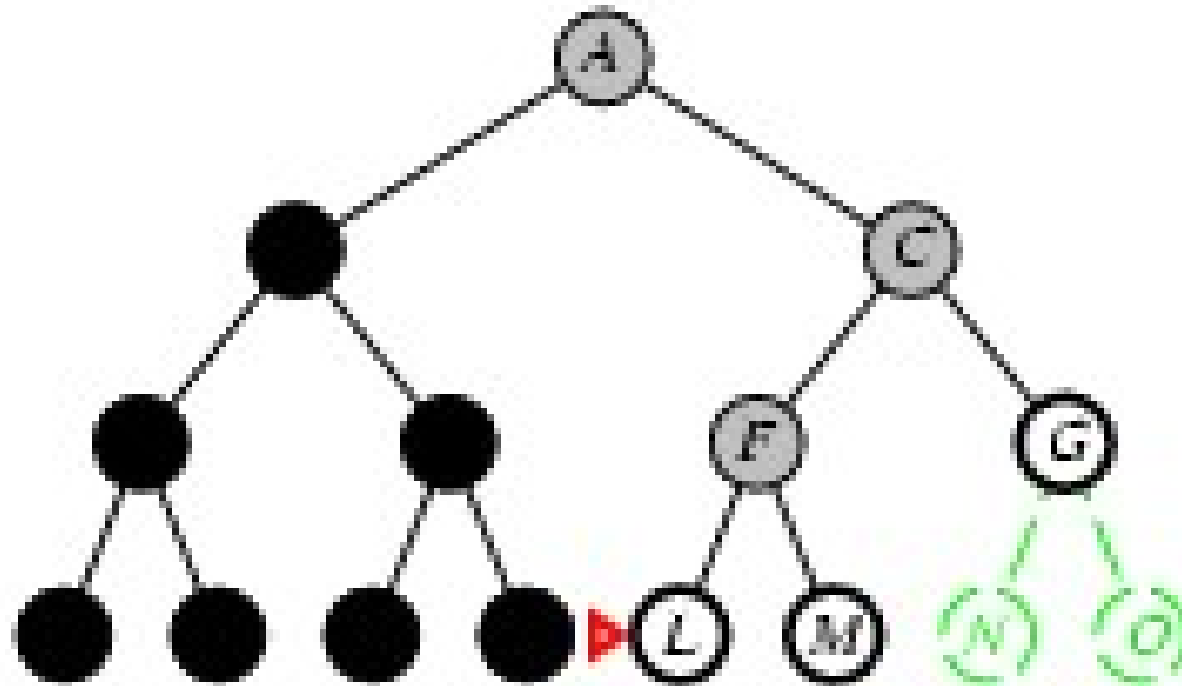
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



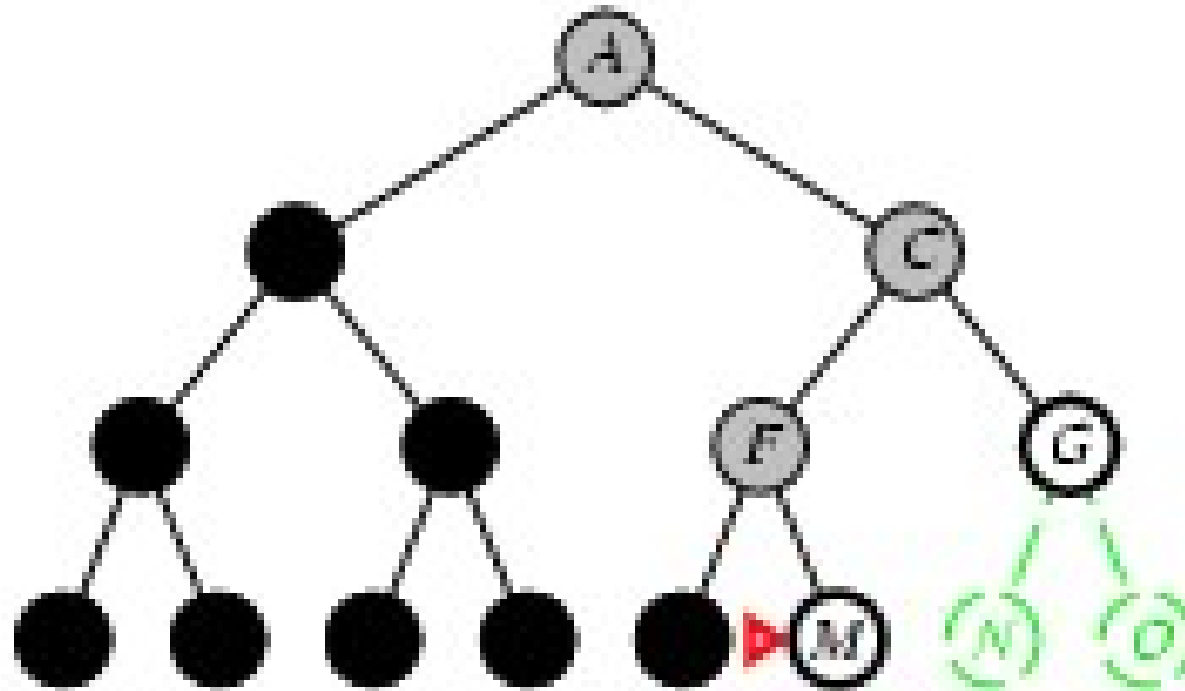
Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.

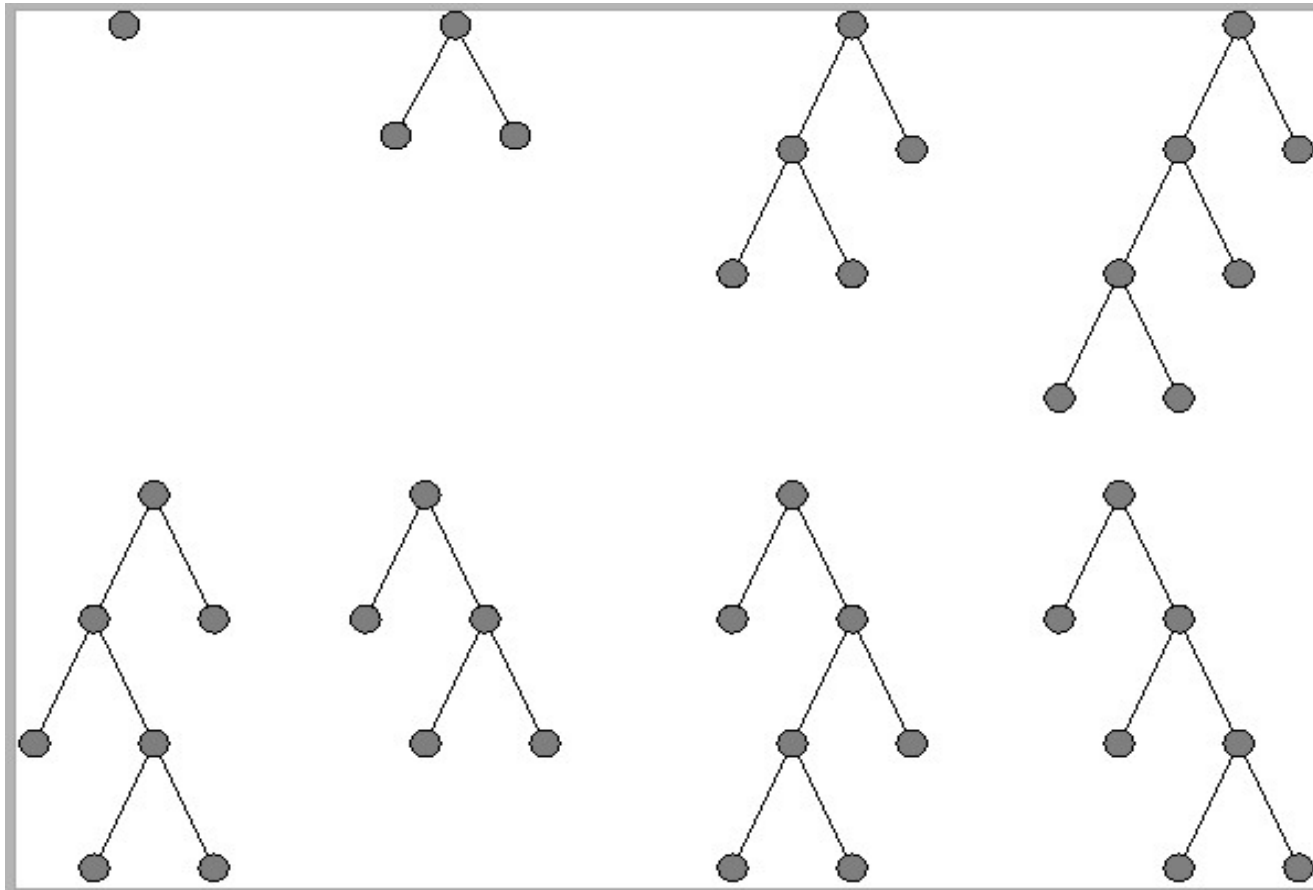


Ricerca Depth-first

- Espande I nodi piu' profondi
- Implementazione:
 - *fringe* = coda LIFO, successori in testa.



Ricerca Depth-first



QueueingFn inserisce i successori all'inizio della coda.
si assume che i nodi di profondità 3 non abbiano
successori



Depth-first – proprietà

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

b - massimo fattore di diramazione dell'albero di ricerca

d - profondità della soluzione a costo minimo

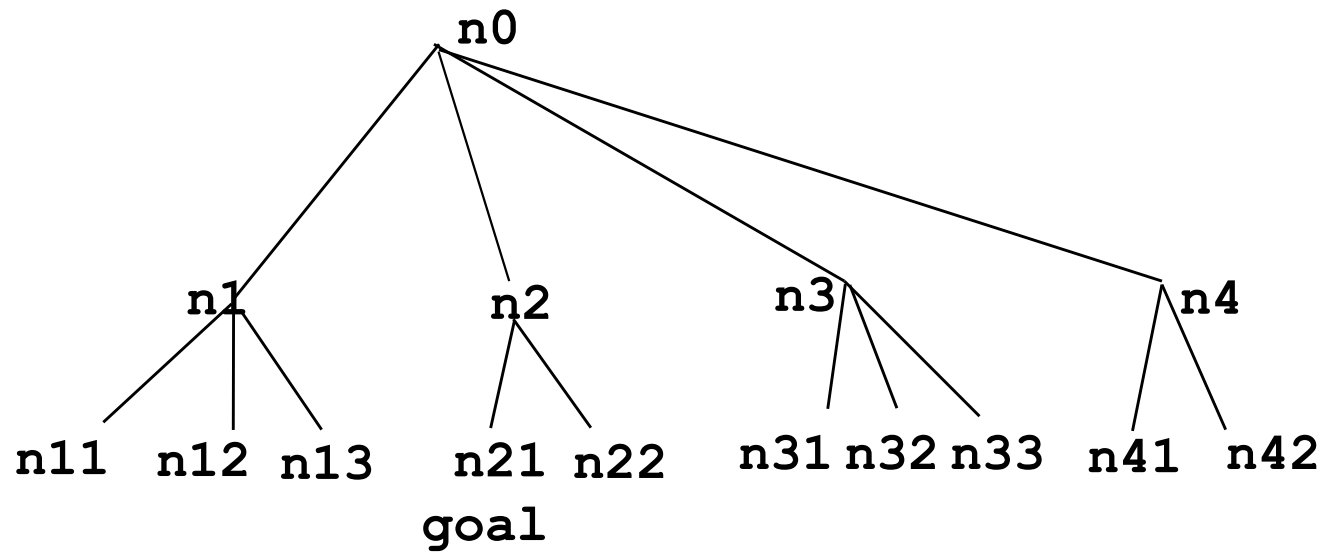
m - massima profondità dello spazio degli stati (può essere infinita)

Implementazione delle differenti Strategie di Ricerca

- Depth-first:
 - Estrai il primo elemento di Nodes
 - Aggiungi i nodi ottenuti dall'espansione in testa alla coda di Nodes (**Queuing-Fn**)
- Breadth-first
 - Estrai il primo elemento di Nodes;
 - Aggiungi i nodi ottenuti dall'espansione in fondo alla coda di Nodes (**Queuing-Fn**)



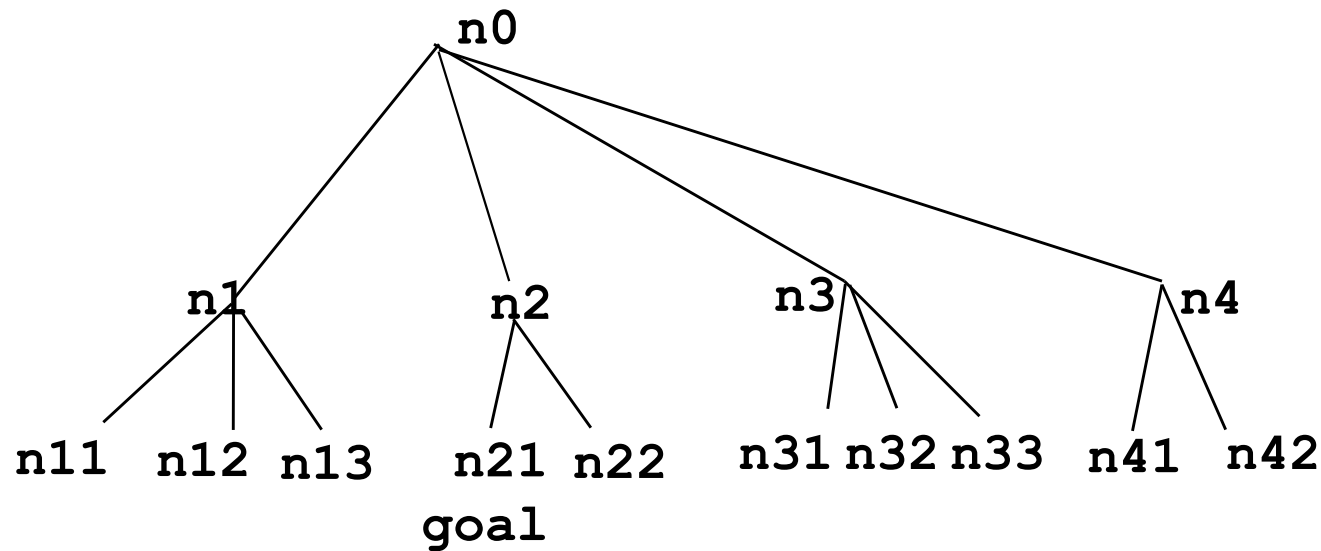
ESEMPIO: DEPTH-FIRST



- Depth-first: figli espansi aggiunti in testa:
 - n0
 - n1,n2,n3,n4
 - n11,n12,n13,n2,n3,n4
 - n11,n12,n13,n2,n3,n4
 - n12,n13,n2,n3,n4
 - n13,n2,n3,n4
 - n2,n3,n4
 - **n21**,n22,n3,n4 Successo



ESEMPIO: BREADTH-FIRST



- Breadth-first: figli espansi aggiunti in coda.
 - n0
 - n1,n2,n3,n4
 - n2,n3,n4,n11,n12,n13
 - n3,n4,n11,n12,n13,n21,n22
 - n4,n11,n12,n13,n21,n22,n31,n32,n33
 - n11,n12,n13,n21,n22,n31,n32,n33,n41,n42
 - n12,n13,n21,n22,n31,n32,n33,n41,n42
 - n13,n21,n22,n31,n32,n33,n41,n42
 - **n21**,n22,n31,n32,n33,n41,n42 Successo



RICERCA A PROFONDITÀ LIMITATA

- E' una variante della depth-first
- Si prevede una PROFONDITÀ MASSIMA di ricerca.
- Quando si raggiunge il MASSIMO di profondità o un FALLIMENTO si considerano STRADE ALTERNATIVE della stessa profondità (se esistono), poi minori di una unità e così via (BACKTRACKING).
- Si possono stabilire limiti massimi di profondità (non necessariamente risolvono il problema della completezza).
- Evita di scendere lungo rami infiniti



RICERCA A PROFONDITÀ LIMITATA

Implementazione Ricorsiva:

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred?  $\leftarrow$  false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred?  $\leftarrow$  true
    else if result  $\neq$  failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Da: S. Russell & P. Norvig: "Intelligenza Artificiale: un approccio moderno", Pearson ed.



Ricerca ad approfondimento iterativo – Iterative deepening search (IDS)

- La ricerca ad approfondimento iterativo evita il problema di scegliere il limite di profondità massimo provando tutti i possibili limiti di profondità.
 - Prima 0, poi 1, poi 2 ecc...
- Combina i vantaggi delle due strategie. È completa e sviluppa un solo ramo alla volta.
- In realtà tanti stati vengono espansi più volte, ma questo non peggiora sensibilmente i tempi di esecuzione.
- In particolare, il numero totale di espansioni è: $(d+1)1 + (d)b + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + b^d$.
- In generale è il metodo preferito quando lo spazio di ricerca è molto ampio.



Ricerca ad approfondimento iterativo – Iterative deepening search (IDS)

Può emulare la breadth first mediante ripetute applicazioni della depth first con una profondità limite crescente.

1. Depth=0
2. Applica depth-first con limite Depth, se trovi successo o fallimento termina
3. Altrimenti incrementa Depth e vai al passo 2

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-  
ure  
  inputs: problem, a problem  
  for depth  $\leftarrow$  0 to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH( problem, depth)  
    if result  $\neq$  cutoff then return result
```

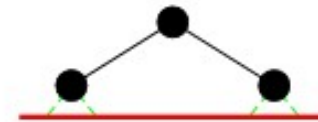
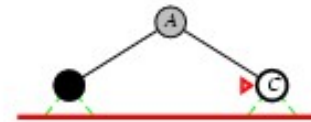
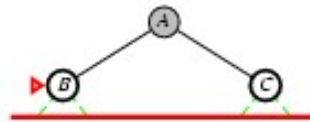
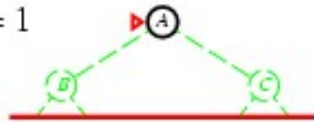
Iterative deepening search $l = 0$

Limit = 0



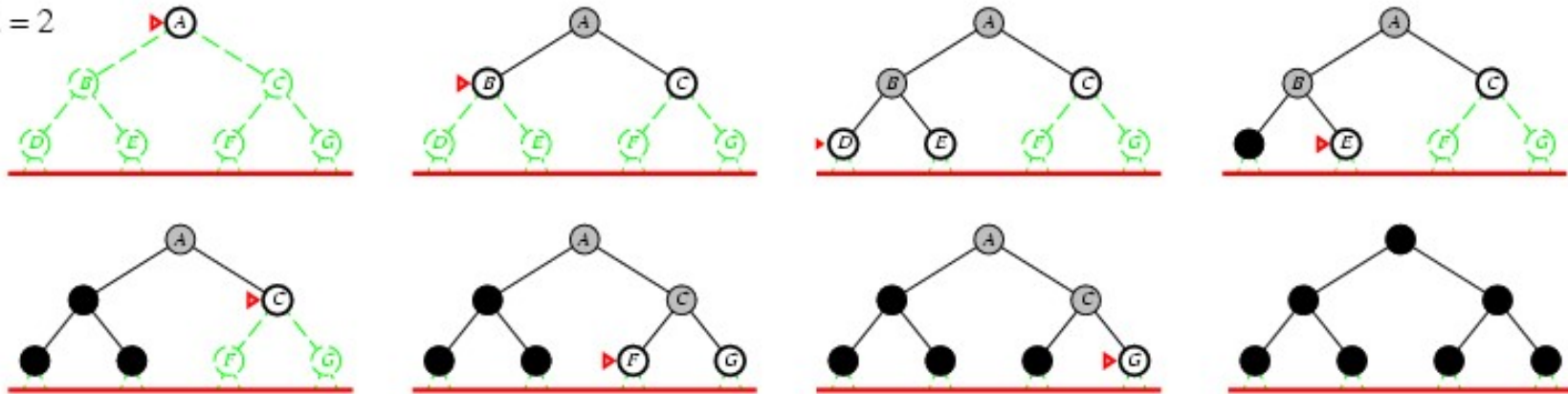
Iterative deepening search $l = 1$

Limit = 1



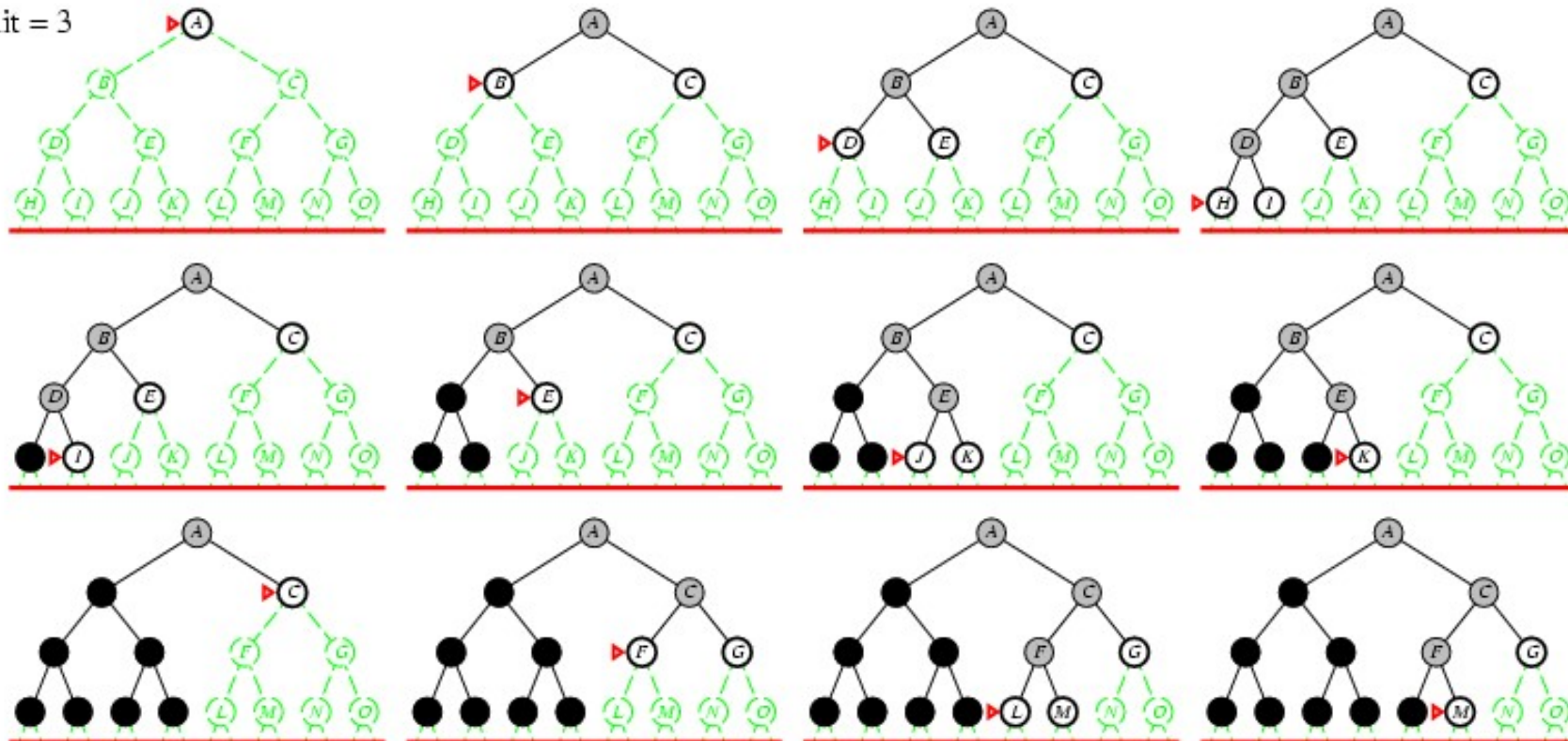
Iterative deepening search l = 2

Limit = 2

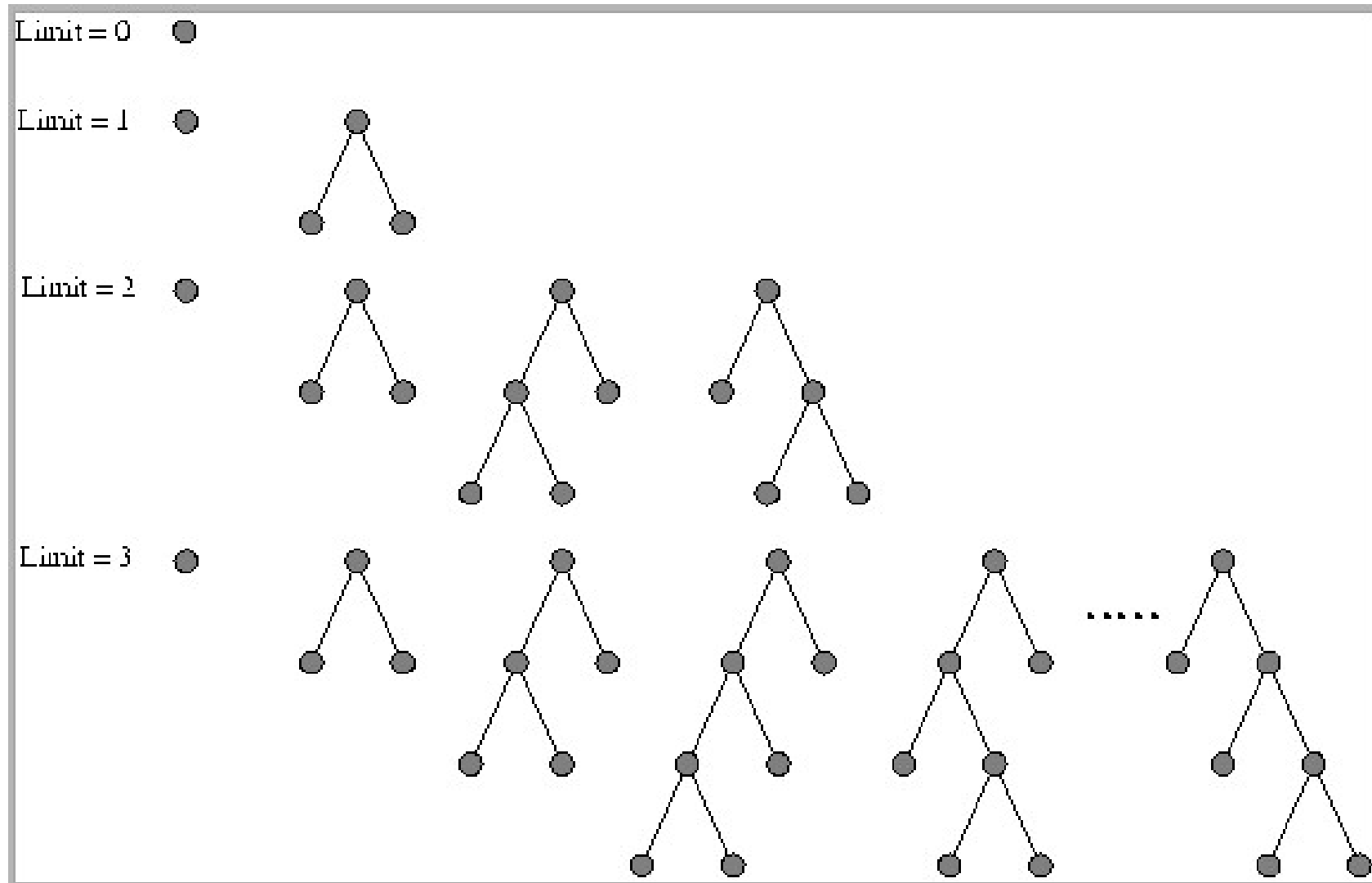


Iterative deepening search $l = 3$

Limit = 3



Ricerca con approfondimento iterativo



Ricerca con approfondimento iterative – proprietà

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost = 1

Can be modified to explore uniform-cost tree

b - massimo fattore di diramazione dell'albero di ricerca

d - profondità della soluzione a costo minimo

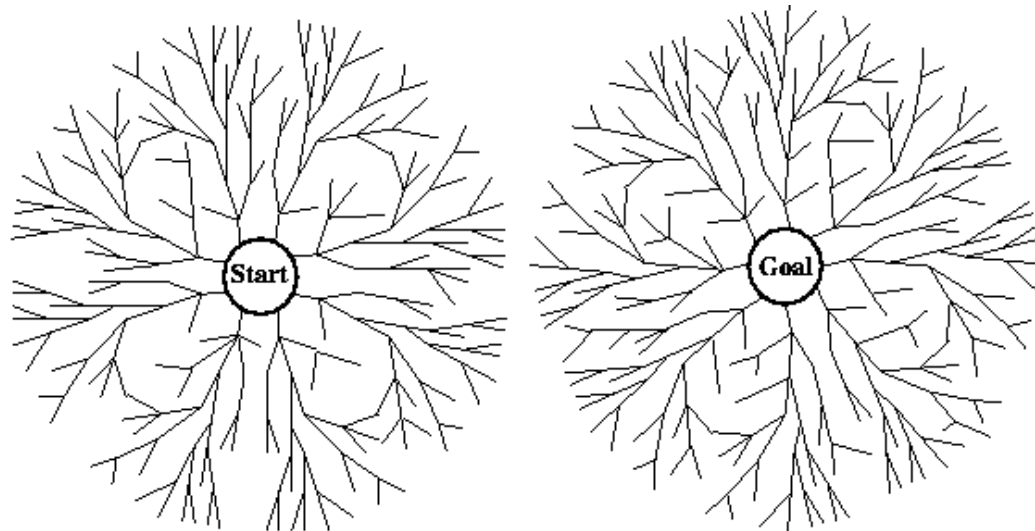
m - massima profondità dello spazio degli stati (può essere infinita)



Ricerca bidirezionale

BIDIREZIONALE o MISTO:

- È la combinazione dei metodi backward e forward
- La memoria di lavoro viene suddivisa in due parti l'una contenente i fatti e l'altra i goals o subgoals;
- Si applicano simultaneamente F-rules e B-rules alle due parti di memoria di lavoro e si termina il procedimento con successo quando la parte di memoria di lavoro ricavata mediante backward chaining è uguale o un sottoinsieme di quella ricavata mediante forward chaining (CONDIZIONE DI TERMINAZIONE).



Confronto fra le strategie di ricerca

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Da: S. Russell & P. Norvig: "Intelligenza Artificiale: un approccio moderno", Pearson ed.

b = fattore di ramificazione;

d = profondità della soluzione;

m =profondità massima dell'albero di ricerca;

l =limite di profondità.

