

# Problem-solving agents

---

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

## Formulazione del problema :

---

- lo **stato iniziale**;
- Descrizione delle **azioni** possibili che sono disponibili da parte dell'agente (operatori che trasformano uno stato in un altro o più formalmente una funzione successore  $S(X)$  che riceve in ingresso uno stato e restituisce *l'insieme degli stati raggiungibili*);
- **test obiettivo** (descrizione di uno stato goal)
- funzione **costo di cammino**

# Cercare soluzioni (albero di ricerca esplicito)

---

- L'idea è quella di **mantenere ed estendere un insieme di sequenze che sono 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 e scegliendo, poi, quella migliore.
  - Il processo di cercare tale sequenza è chiamato **RICERCA (SEARCH)**.
  - 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.

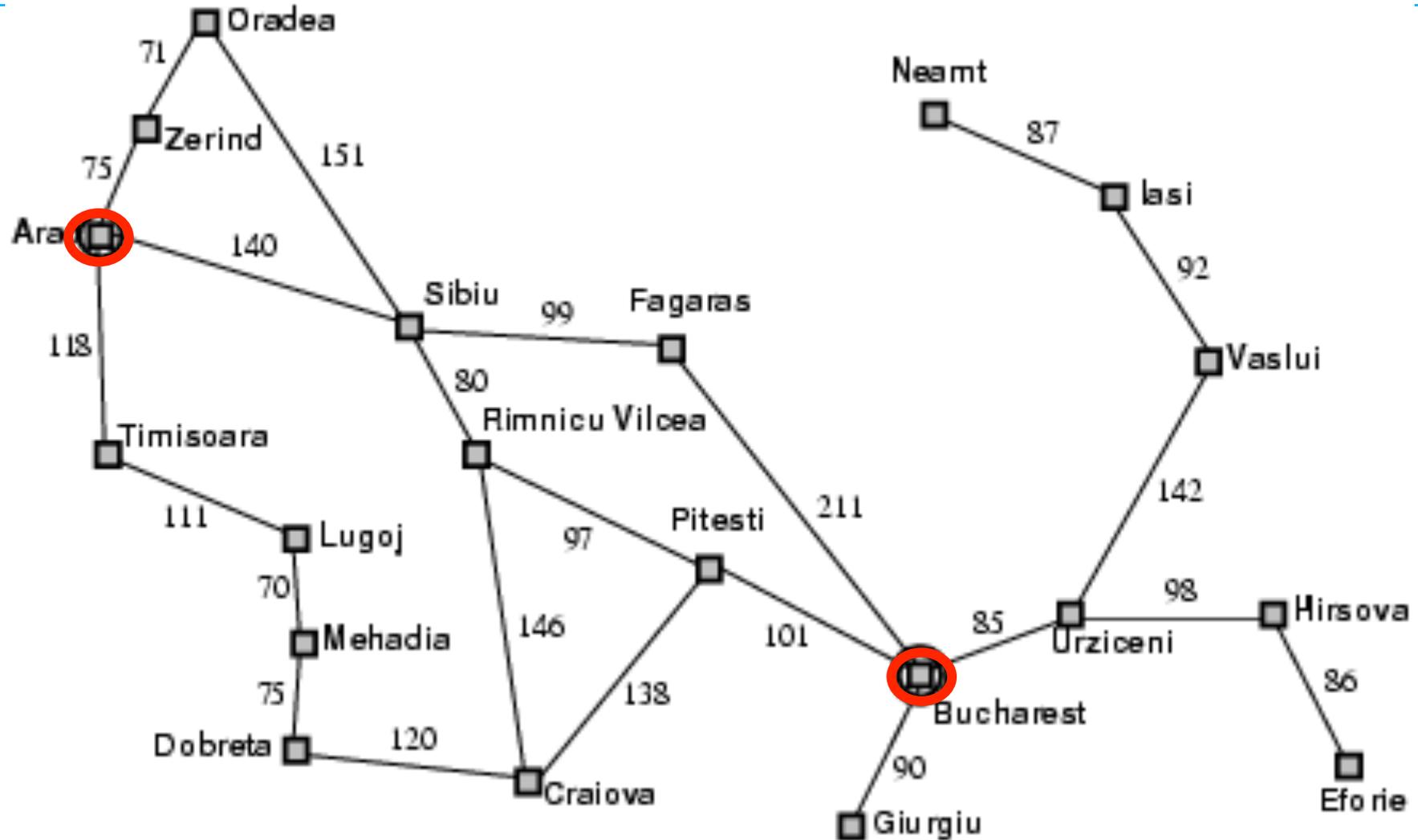
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a 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 ad Arad.
- I voli partono da Bucharest domani
- **goal:**
  - Essere a Bucharest
- **problema:**
  - **stati:** varie città
  - **azioni:** guida fra le città
- **soluzione:**
  - Sequenza di città, ad es., Arad, Sibiu, Fagaras, Bucharest.

# Esempio: Romania



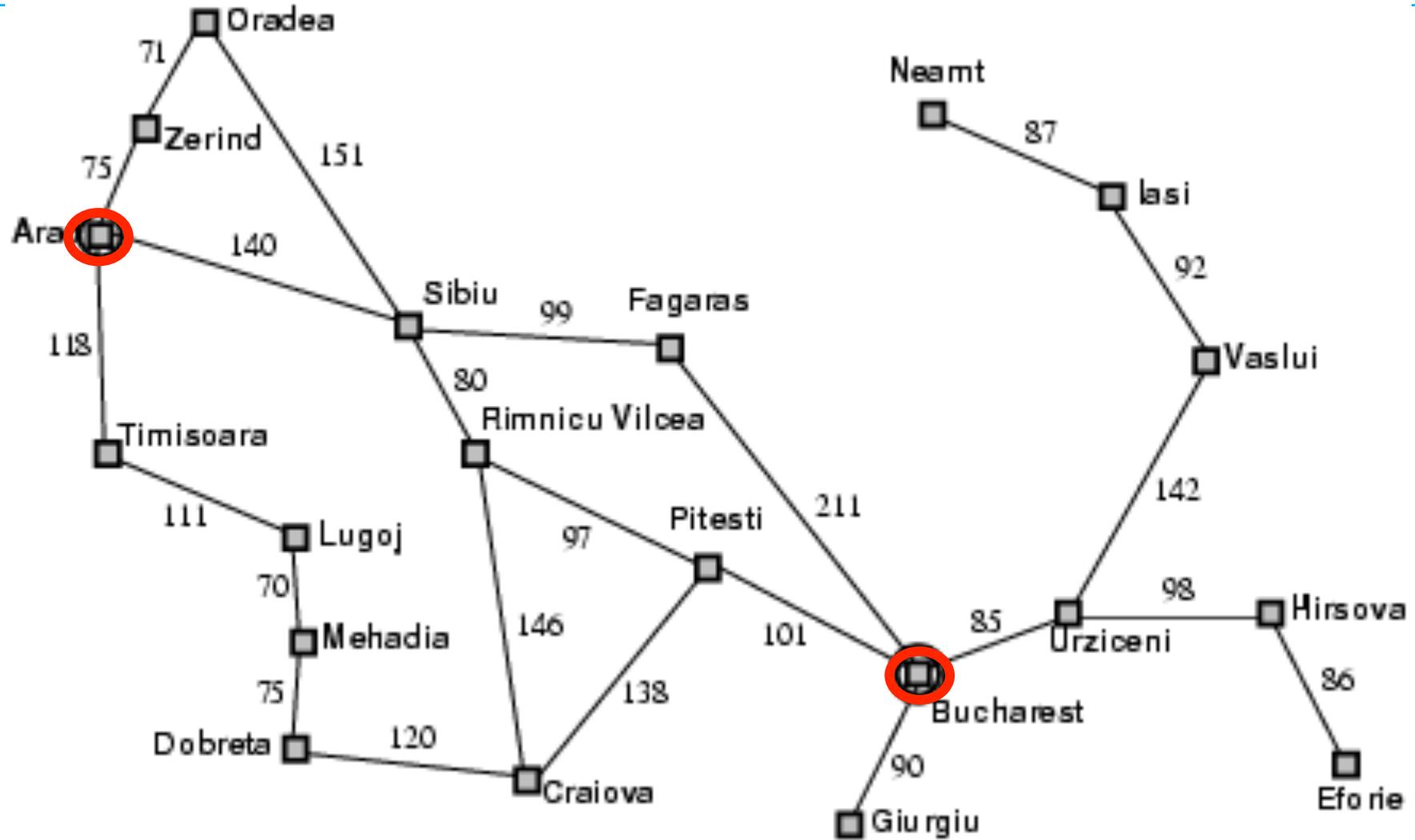
# Formulazione del problema

---

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

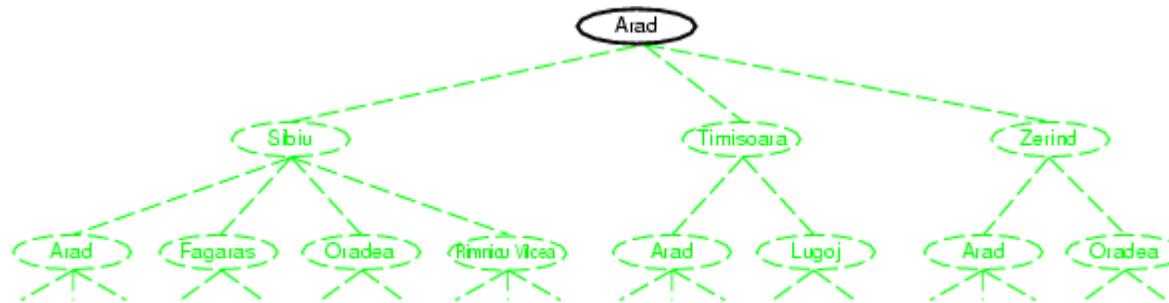
1. **Stato iniziale** es., "at Arad"
  2. **Azioni o funzioni successore**  $S(x)$  = insieme di coppie azione-stato
    - es.,  $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots \}$
  3. **goal test**, può essere
    - **esplicito**, es.,  $x = \text{"at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

# Esempio: Romania



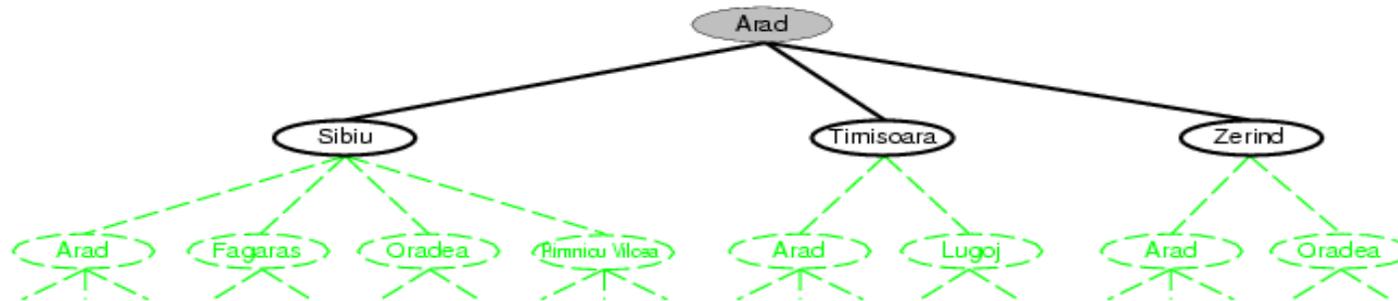
# Tree search example

---



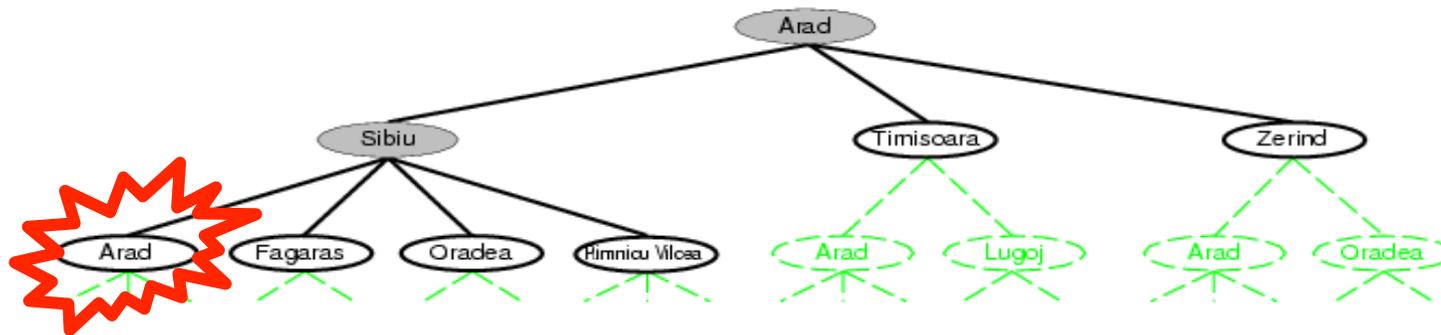
# Albero di ricerca, esempio

---



# Albero di ricerca, esempio

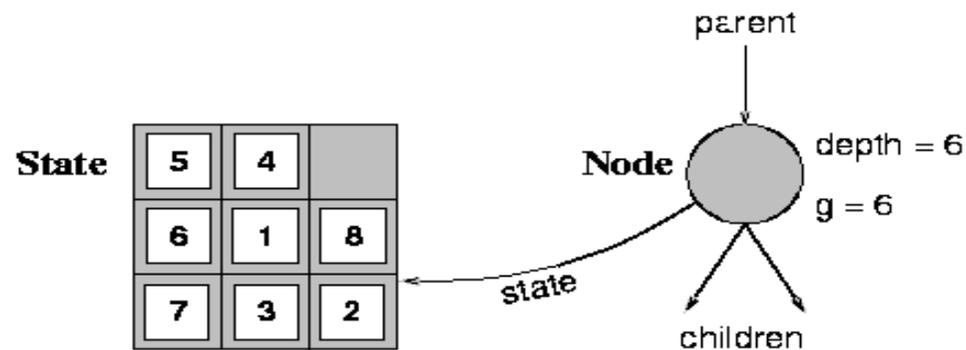
---



# Strutture dati per l'albero di ricerca (struttura di un nodo)

---

- 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



## L'algoritmo generale di ricerca su albero

---

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

---

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

# Albero di ricerca: un esempio

---

Stato iniziale

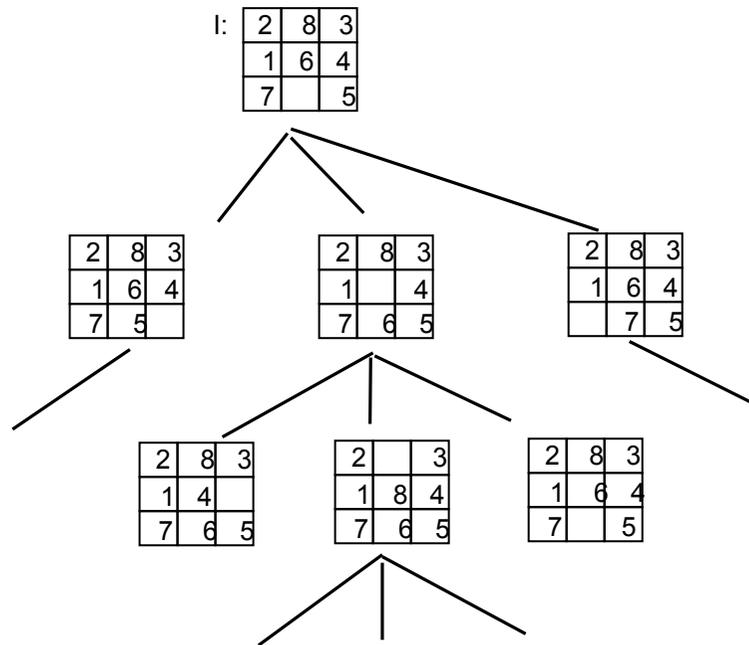
Goal

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 possono essere svolte più azioni (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 o cieche**) e fare una ricerca ESAUSTIVA.
    - Impraticabile per problemi di una certa complessità
  - Applicare conoscenza sul problema (**strategie informate o euristiche**)

# 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 è fatta in base ad una strategia.
- Abbiamo **strategie informate** (o *euristiche*) e **non informate** (cieche o *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*, QUEUING-FN) **returns** a solution, or failure

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

**loop do**

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

*node* ← REMOVE-FRONT(*nodes*)

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

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

**end**

Tramite l'argomento **QUEUING-FN** viene passata una funzione per inserire (accodare nel caso della Breadth First) 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.
- **QUEUING-FN** accoda i nodi ottenuti dall'espansione di un *node* e li inserisce in fondo alla coda *nodes* (frontiera)
- 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

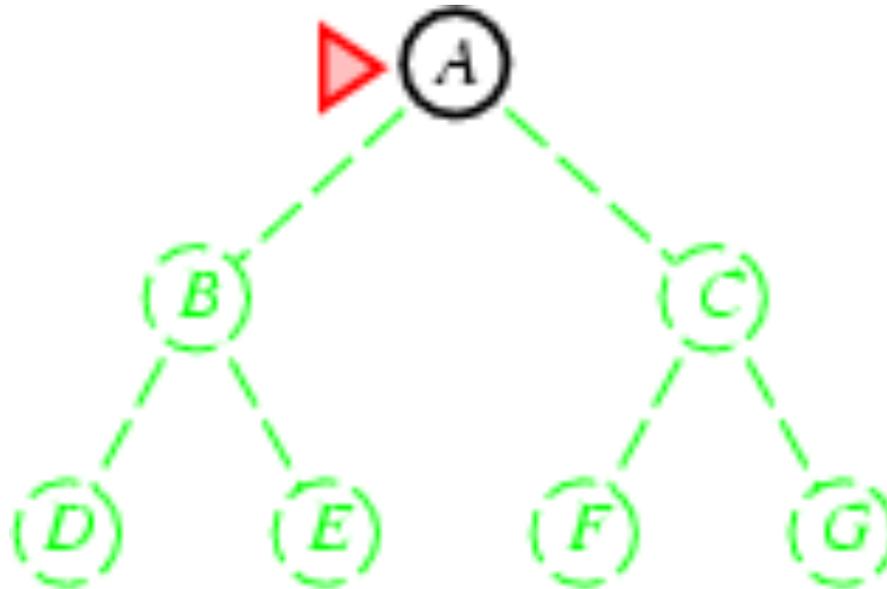
---

- 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 aperte più strade **CONTEMPORANEAMENTE**.
- Tale strategia garantisce la **COMPLETEZZA**, ma **NON** permette una **IMPLEMENTAZIONE EFFICIENTE** su sistemi mono-processore (architetture multi-processore).

# Ricerca Breadth-first

---

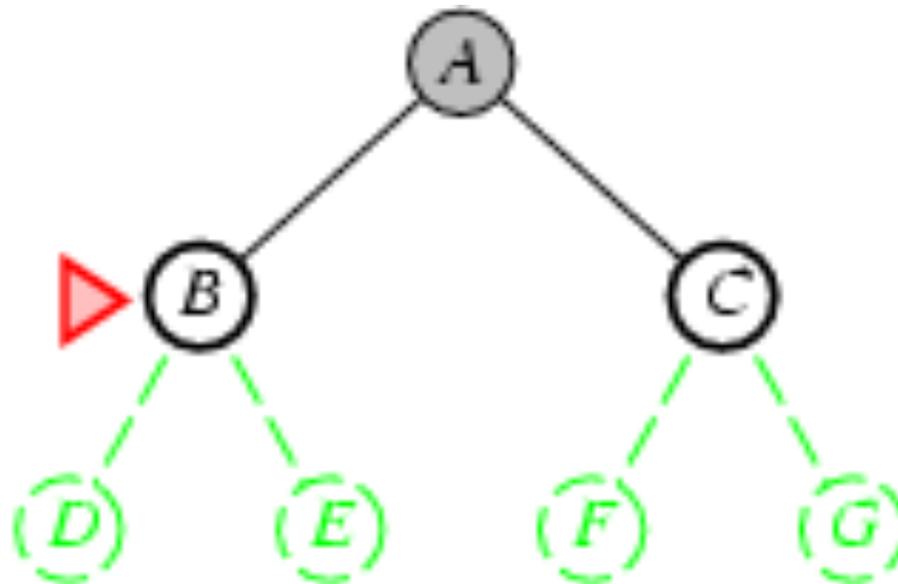
- Espande i nodi a profondità minore
- Implementazione:
  - *nodes* e' una coda FIFO, i.e. successori in fondo.



# Ricerca Breadth-first

---

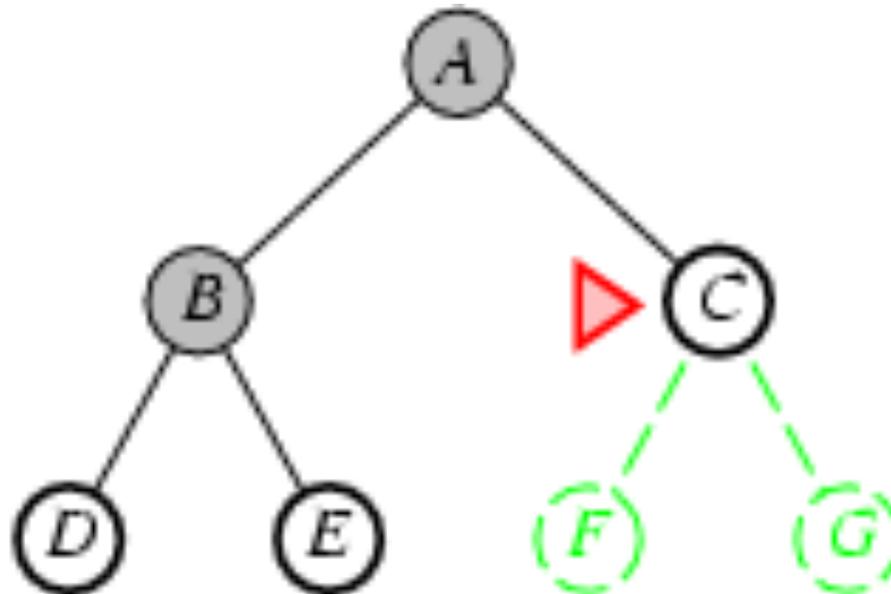
- Espande i nodi a profondità minore
- Implementazione:
  - *nodes* e' una coda FIFO, i.e. successori in fondo.



# Ricerca Breadth-first

---

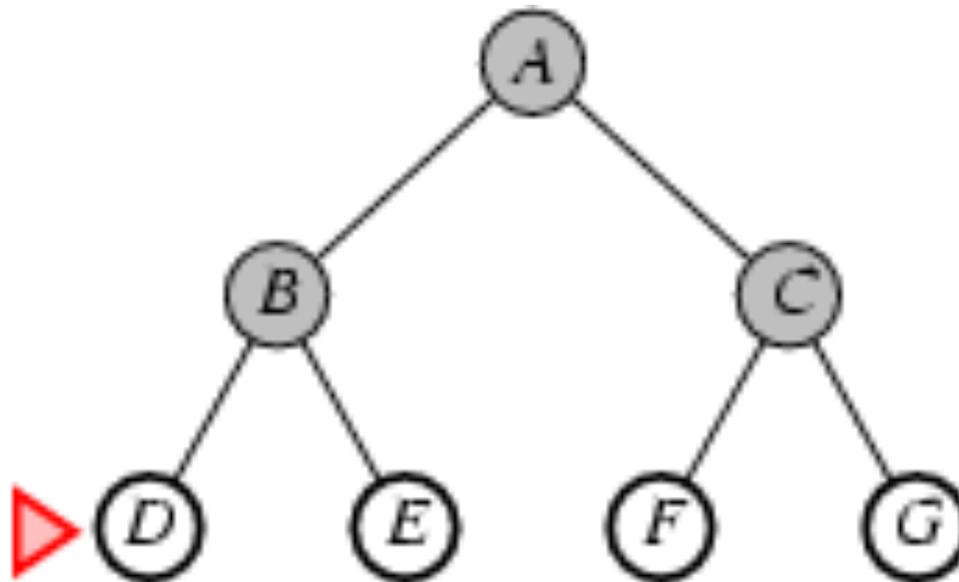
- Espande i nodi a profondita` minore
- Implementazione:
  - *nodes* e' una coda FIFO, i.e. successori in fondo.



# Ricerca Breadth-first

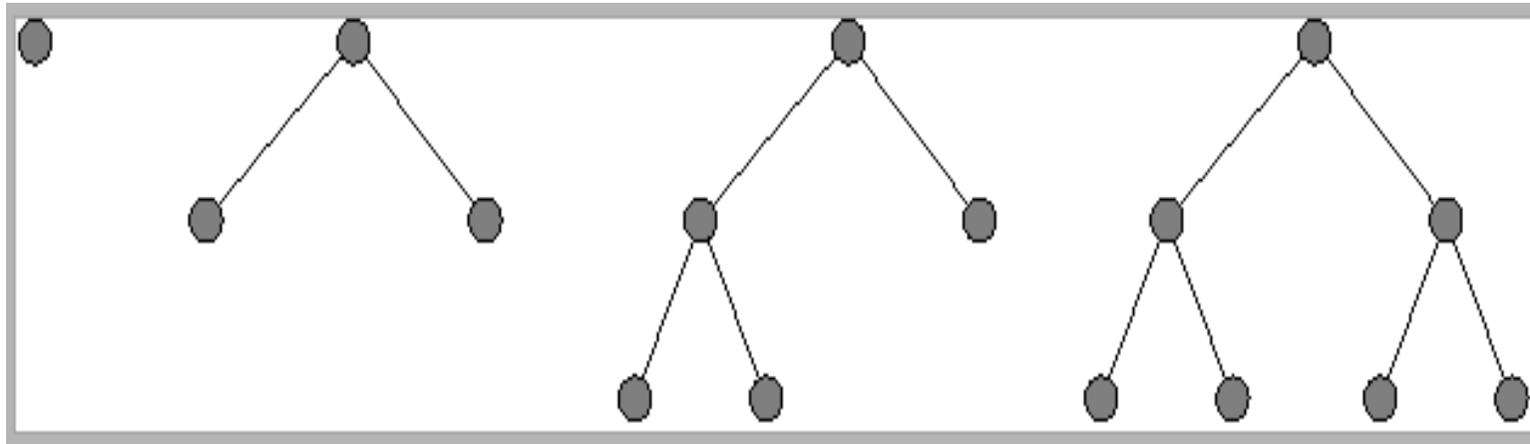
---

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



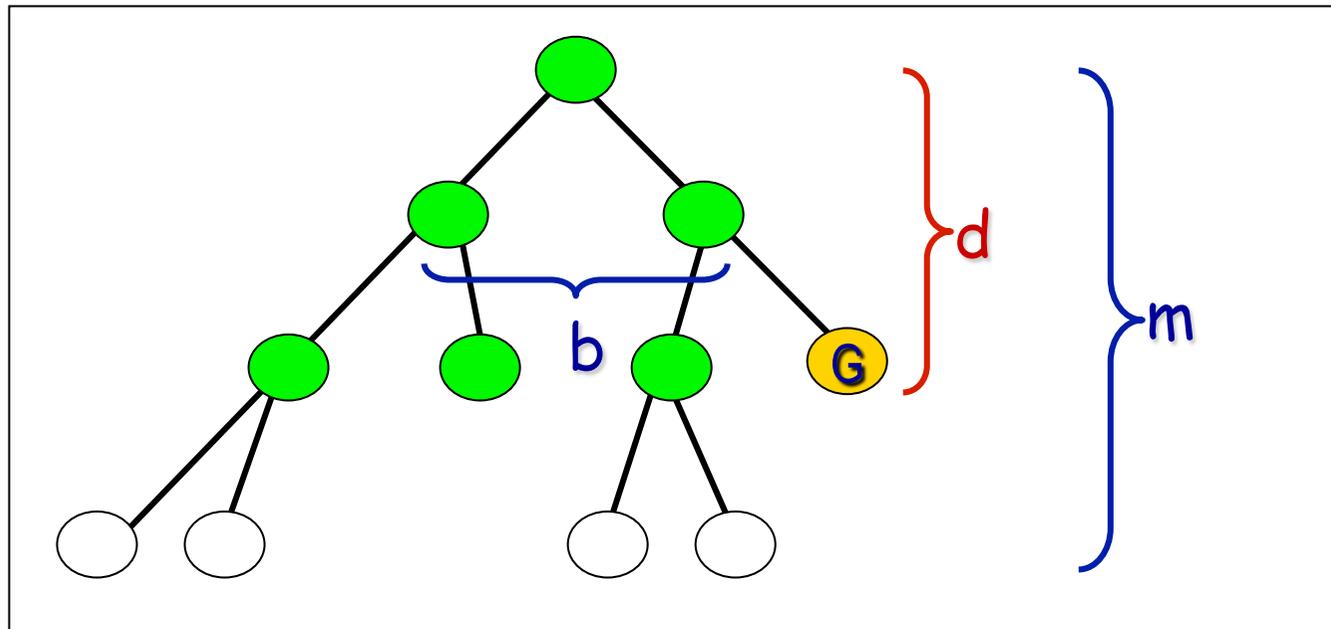
# Ricerca in ampiezza

---



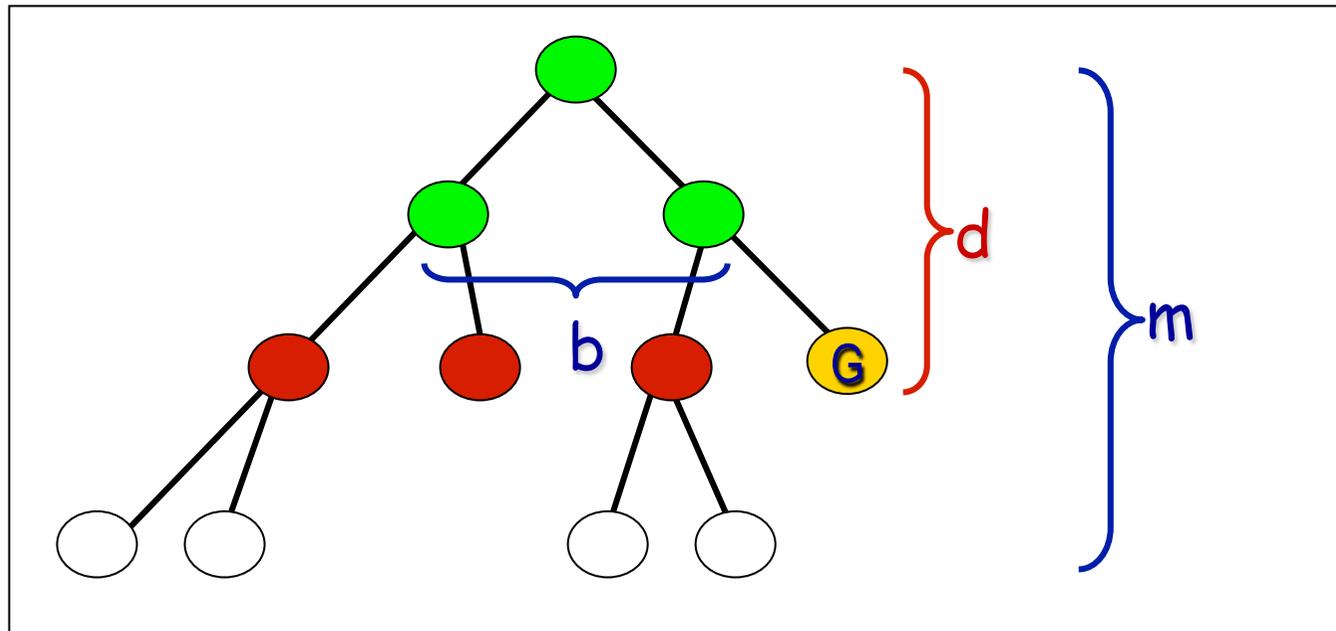
**QUEUING-FN** = metti i successori alla fine della coda (si ottiene così una lista ordinata per profondità crescente dei nodi)

# Tempo (breadth-first)



- $O(b^d)$  se la soluzione esiste a profondità  $d$
  - $O(b^m)$  se la soluzione non esiste e l'albero ha profondità  $m$
- $b$  fattore medio di ramificazione (branching) dell'albero,  $d$  profondità della soluzione (se esiste),  $m$  profondità massima dell'albero

# Memoria (breadth-first)



- Nel caso la soluzione esista a profondità  $d$ :  $b^d$

## 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

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	$10^6$	18 minutes	111 megabytes
8	$10^8$	31 hours	11 gigabytes
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.

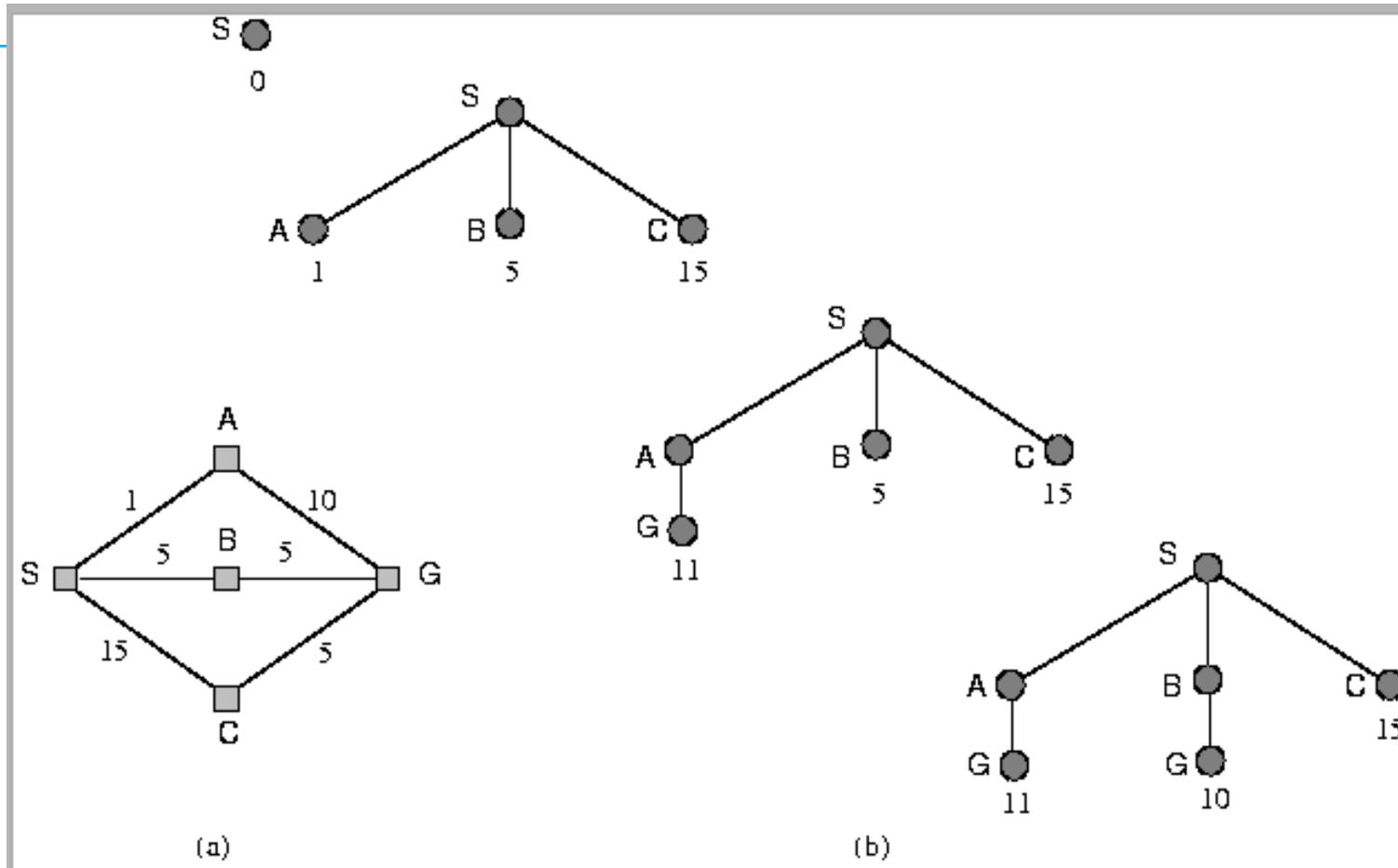
# BREADTH-FIRST

---

- 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 a costo uniforme

ciascun nodo è etichettato con il costo  $g(n)$



**QUEUING-FN** = inserisci i successori di un nodo nella frontiera in ordine di costo di cammino crescente

## Proprietà:

---

### Properties of uniform-cost search

Complete?? Yes, if step cost  $\geq \epsilon$

Time?? # of nodes with  $g \leq$  cost of optimal solution

Space?? # of nodes with  $g \leq$  cost of optimal solution

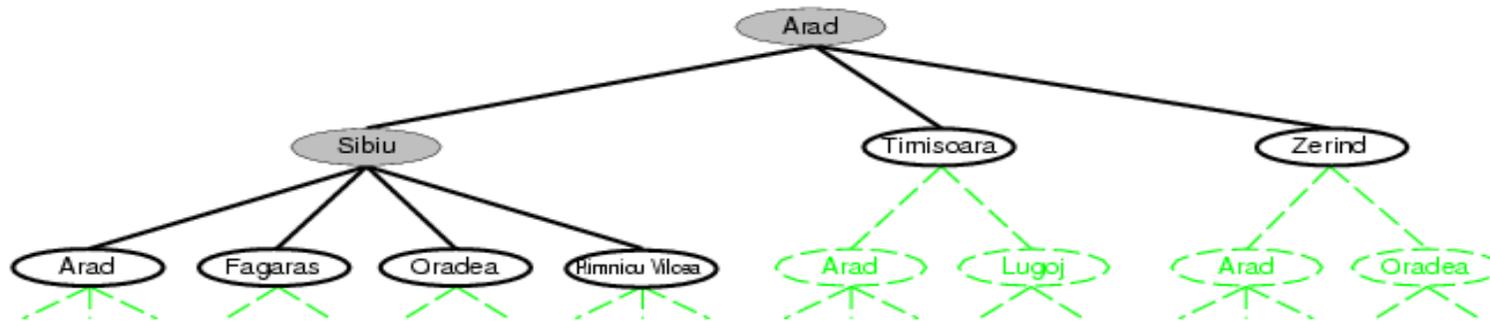
Optimal?? Yes

Complessità dipende dal costo dei cammini e non dalla loro profondità

Può essere anche maggiore di  $b^d$

# Complessità spaziale: come ridurla?

---



- Esplorare l'albero lungo cammini, in profondità, sino a raggiungere un nodo non espandibile, o un nodo soluzione

# STRATEGIE DI RICERCA: DEPTH FIRST

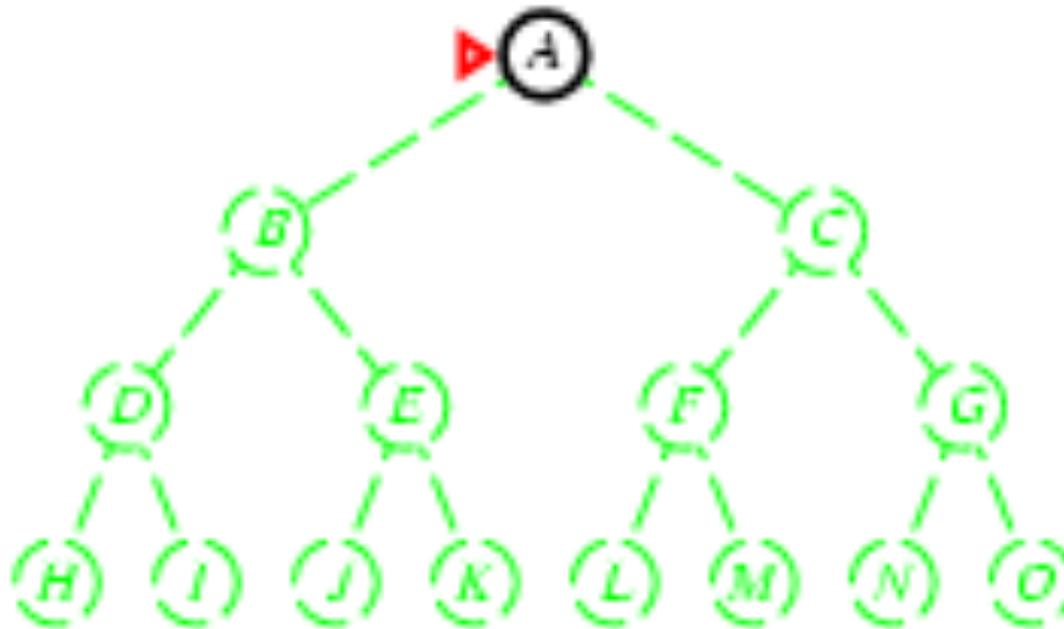
---

- ESPANDE per primi nodi PIÙ PROFONDI;
- I nodi di UGUALE PROFONDITÀ vengono selezionati ARBITRARIAMENTE (quelli più a sinistra).

# Ricerca Depth-first

---

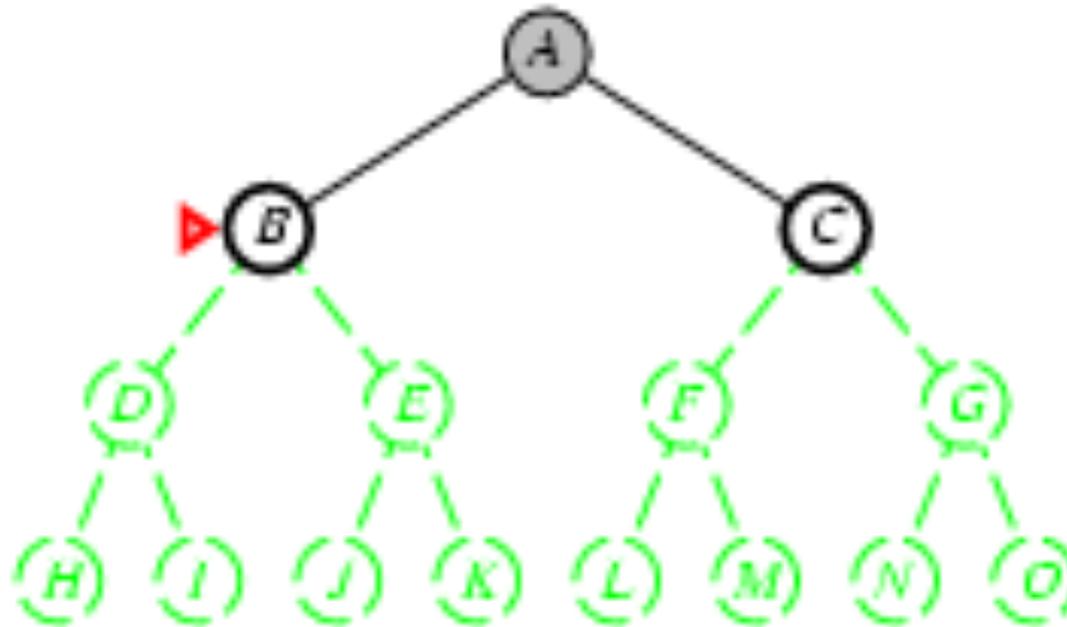
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



# Ricerca Depth-first

---

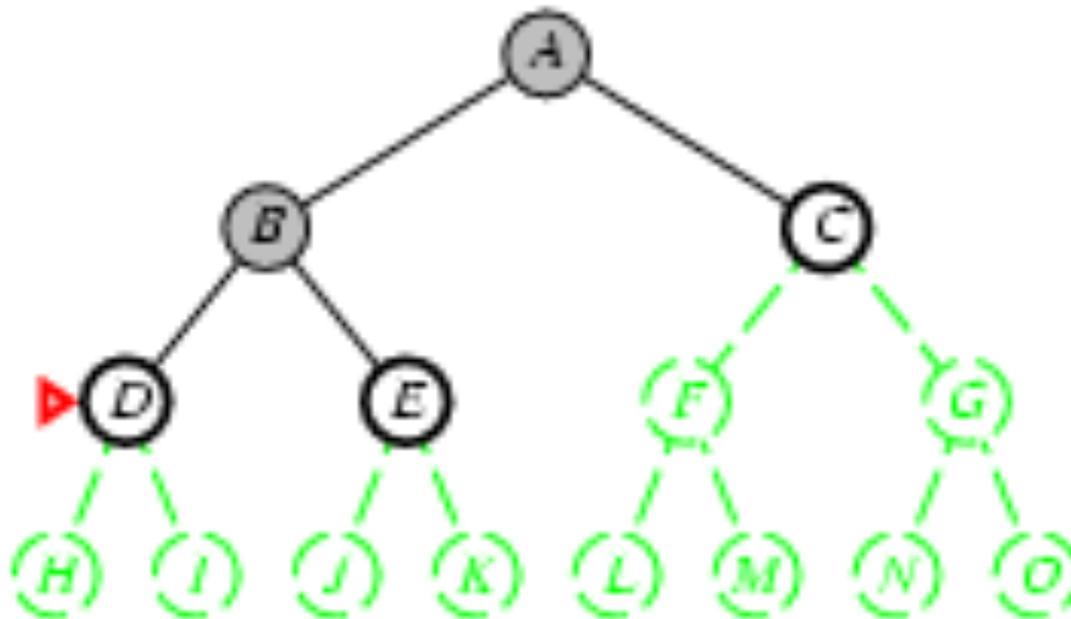
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



# Ricerca Depth-first

---

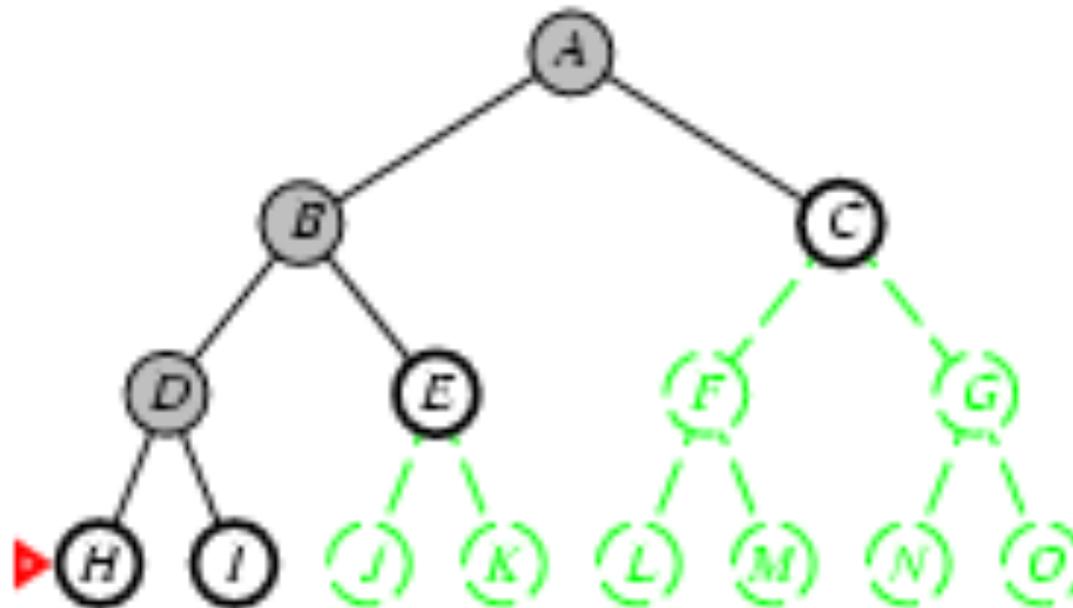
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



# Ricerca Depth-first

---

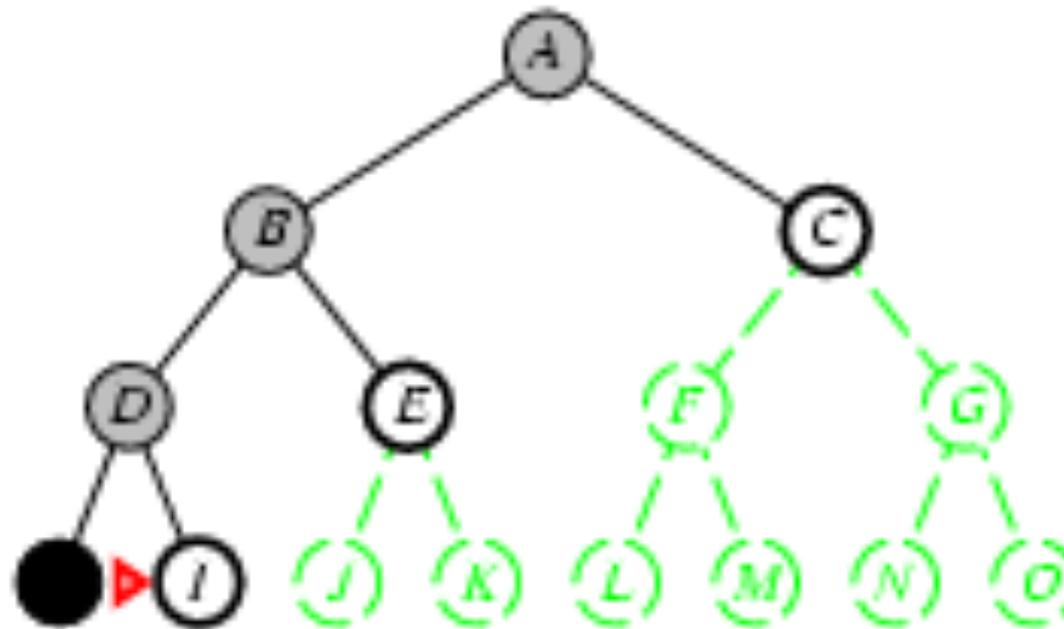
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



# Ricerca Depth-first

---

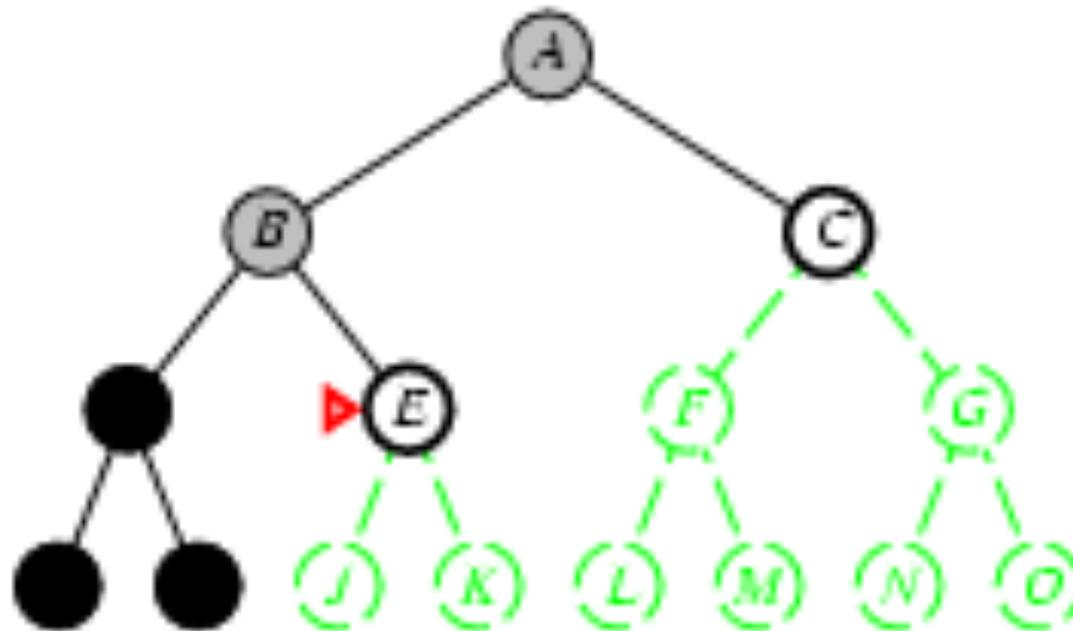
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



# Ricerca Depth-first

---

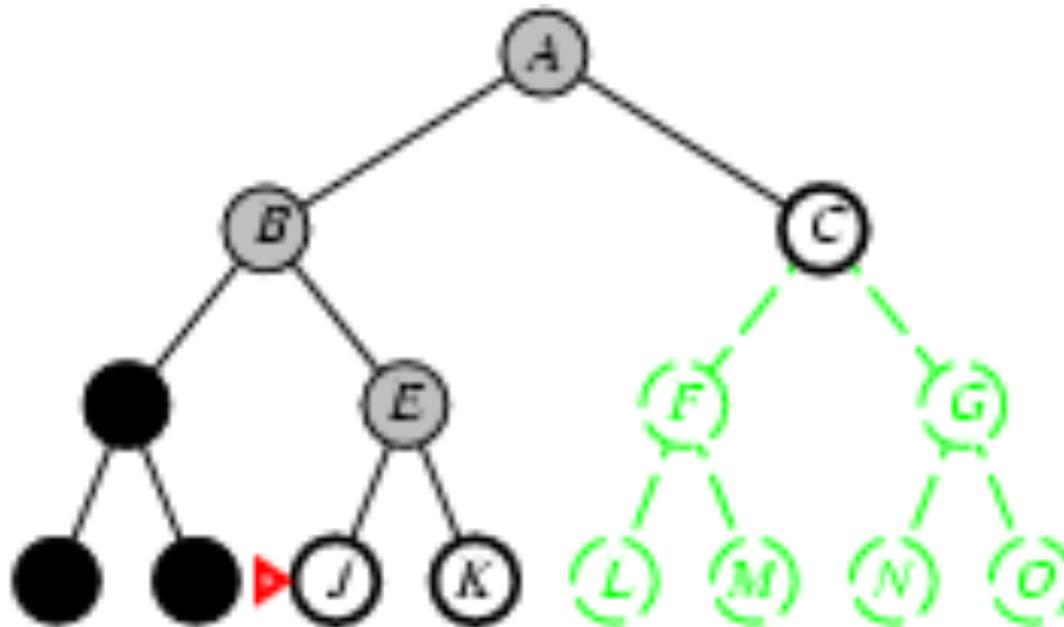
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



# Ricerca Depth-first

---

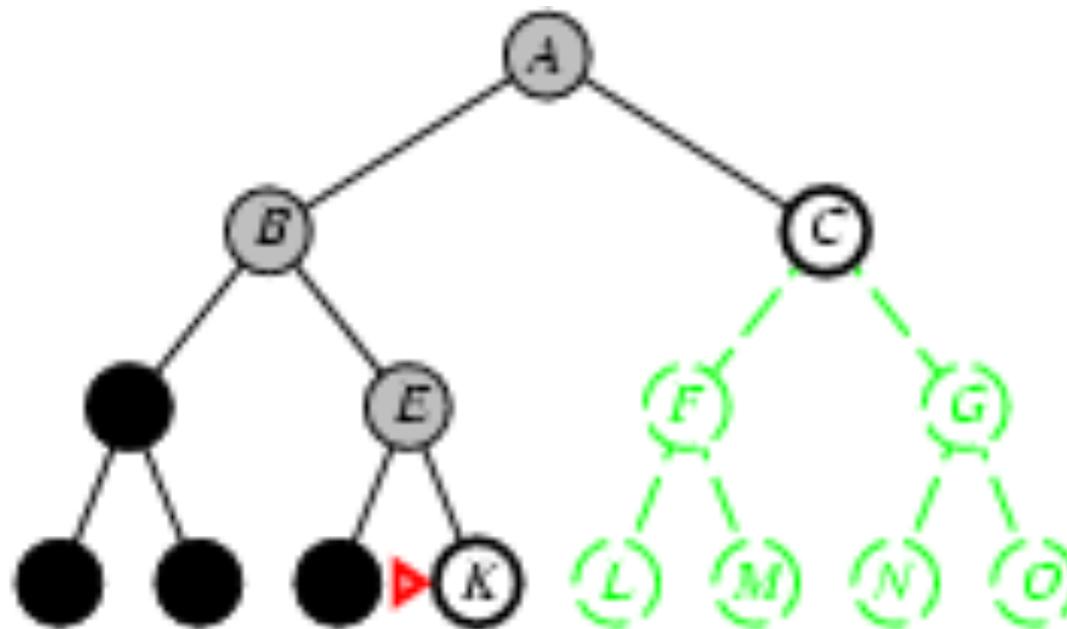
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



# Ricerca Depth-first

---

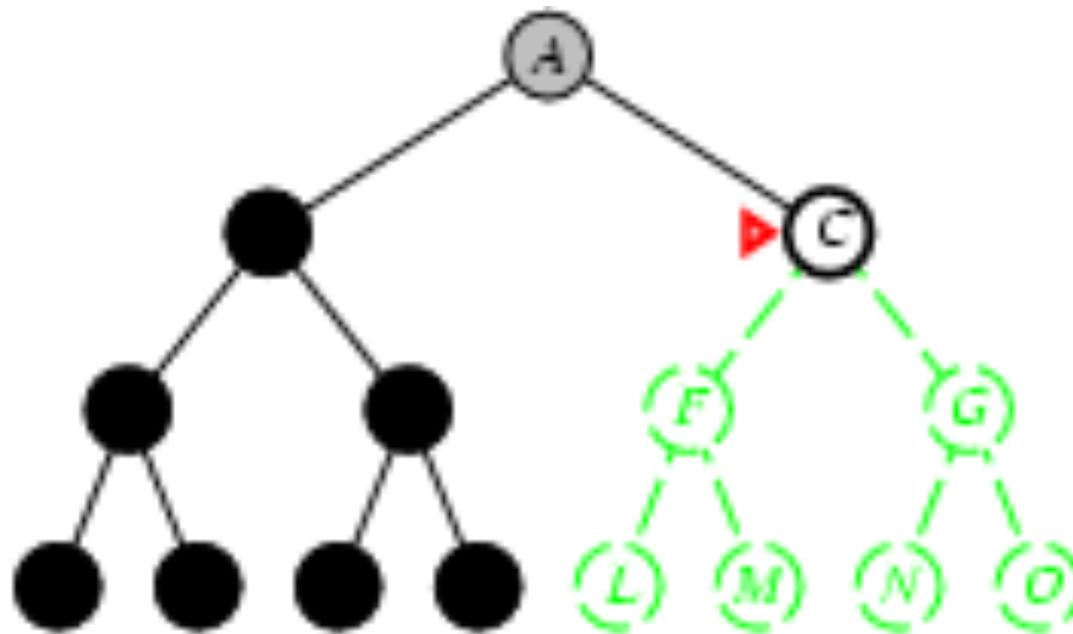
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



# Ricerca Depth-first

---

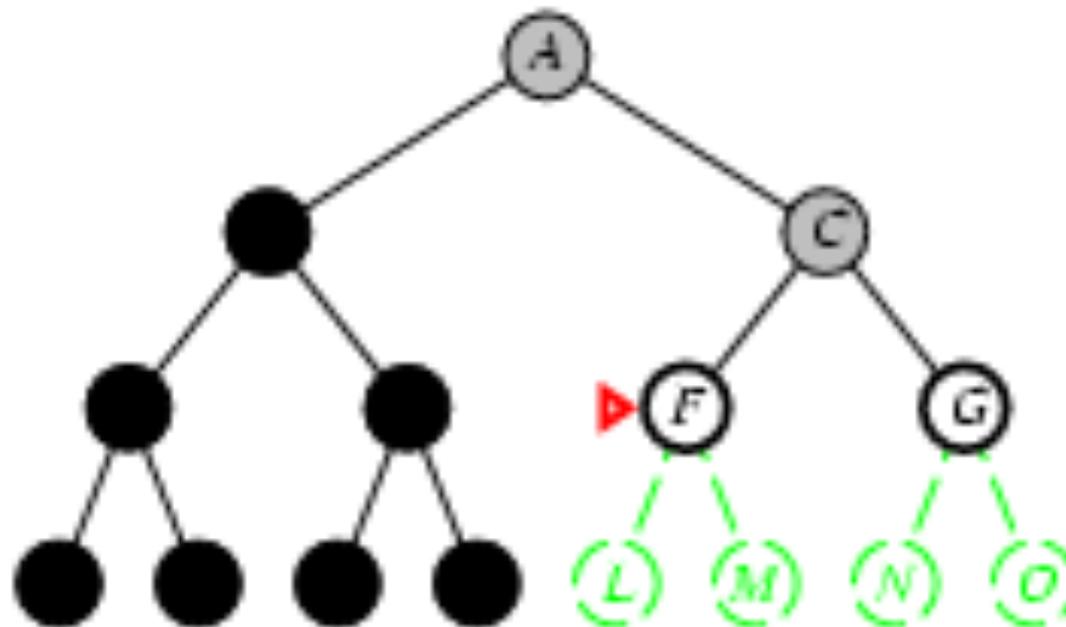
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



# Ricerca Depth-first

---

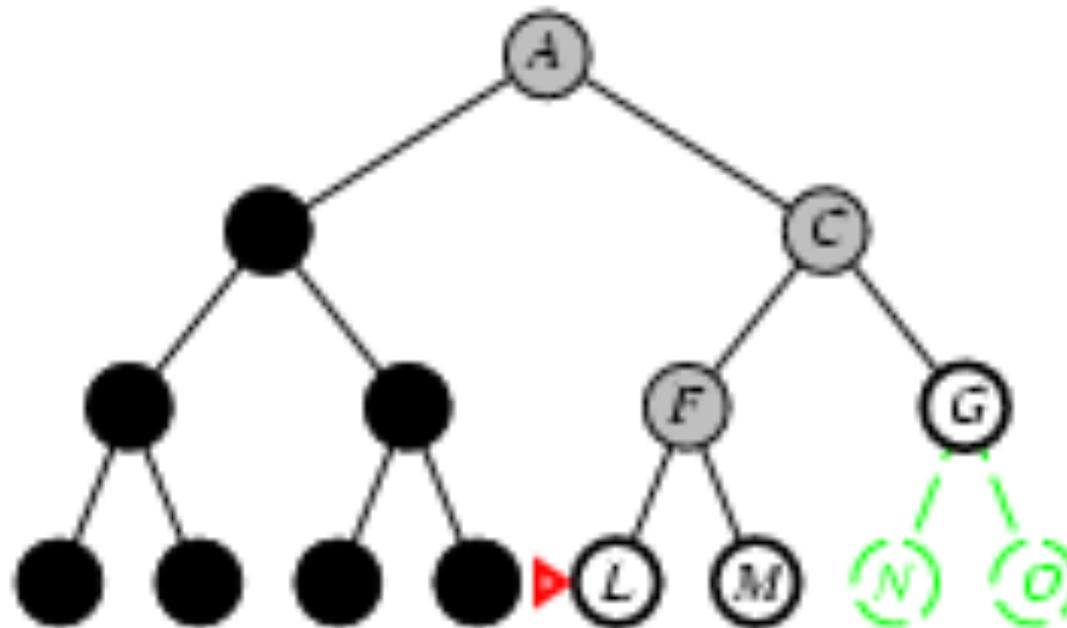
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



# Ricerca Depth-first

---

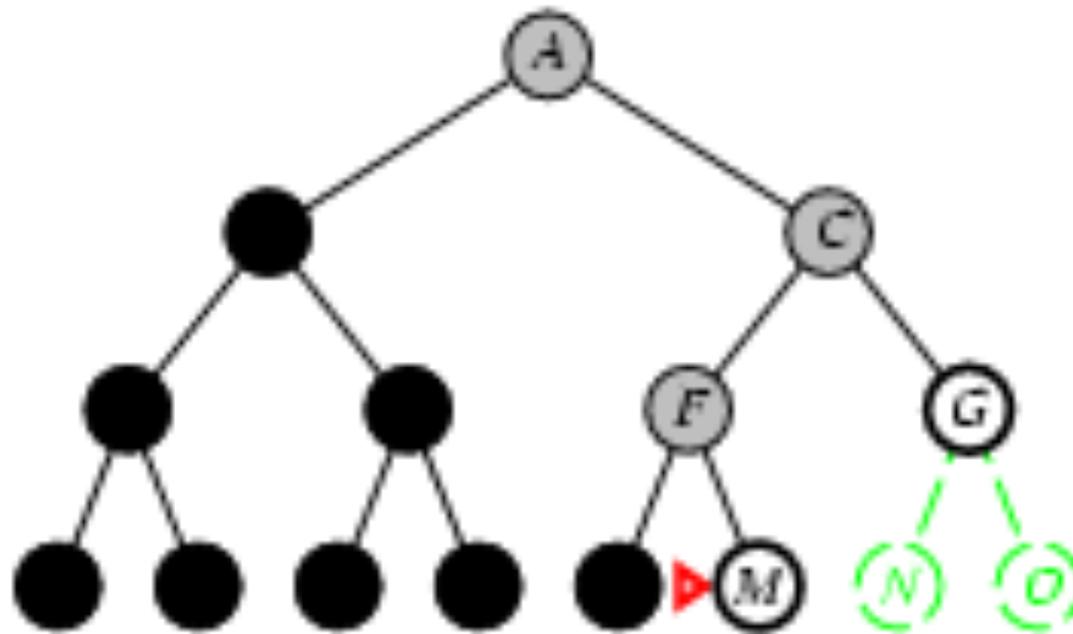
- Espande i nodi piu' profondi
- Implementazione:
  - *nodes* = coda LIFO, successori in testa.



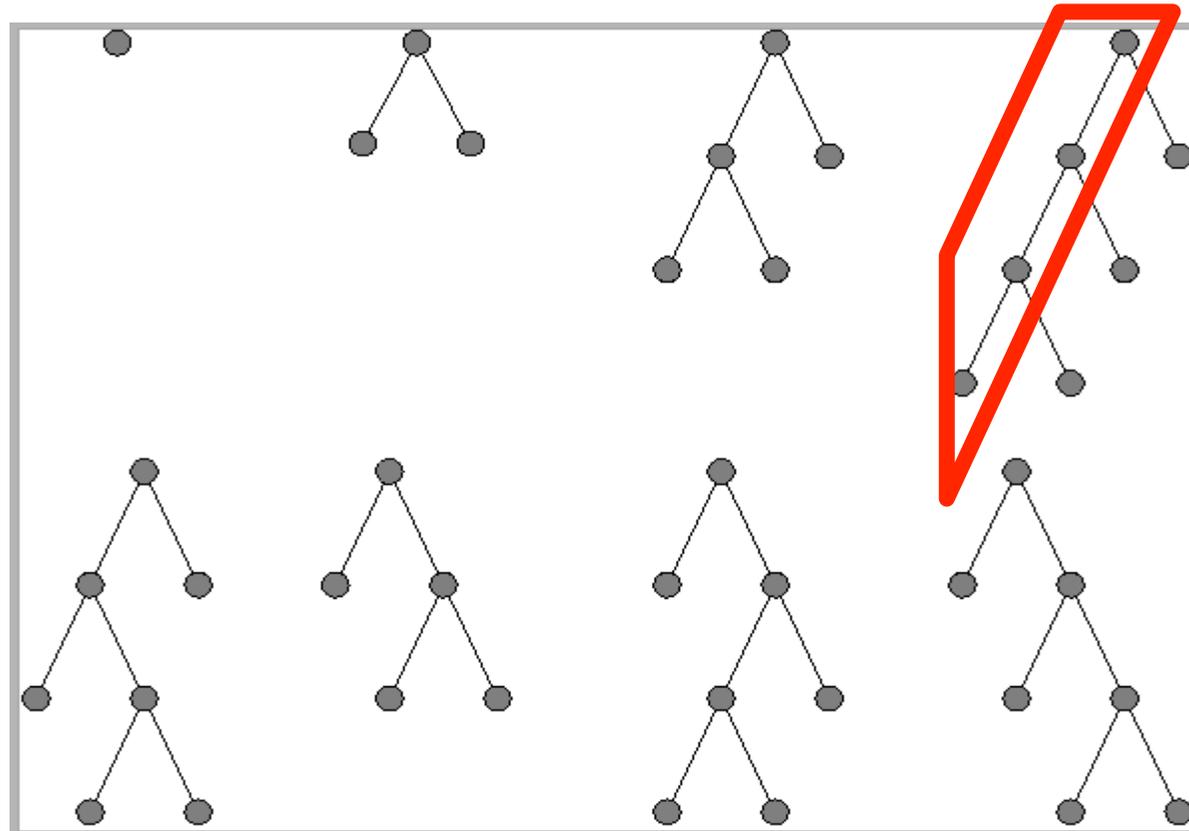
# Ricerca Depth-first

---

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



# Ricerca in profondità



**QUEUING-FN** = inserisci i successori all'inizio della coda (in testa)  
si assume che i nodi di profondità 3 non abbiano successori

## Depth-first: complessità (1)

---

- La ricerca in profondità richiede un'occupazione di memoria più bassa della breadth-first
- ***La complessità spaziale è lineare nella profondità raggiunta***
- Per uno spazio degli stati con fattore di ramificazione  $b$  e profondità massima  $m$ , la ricerca richiede memorizzare al massimo  **$b \cdot m$  nodi**.



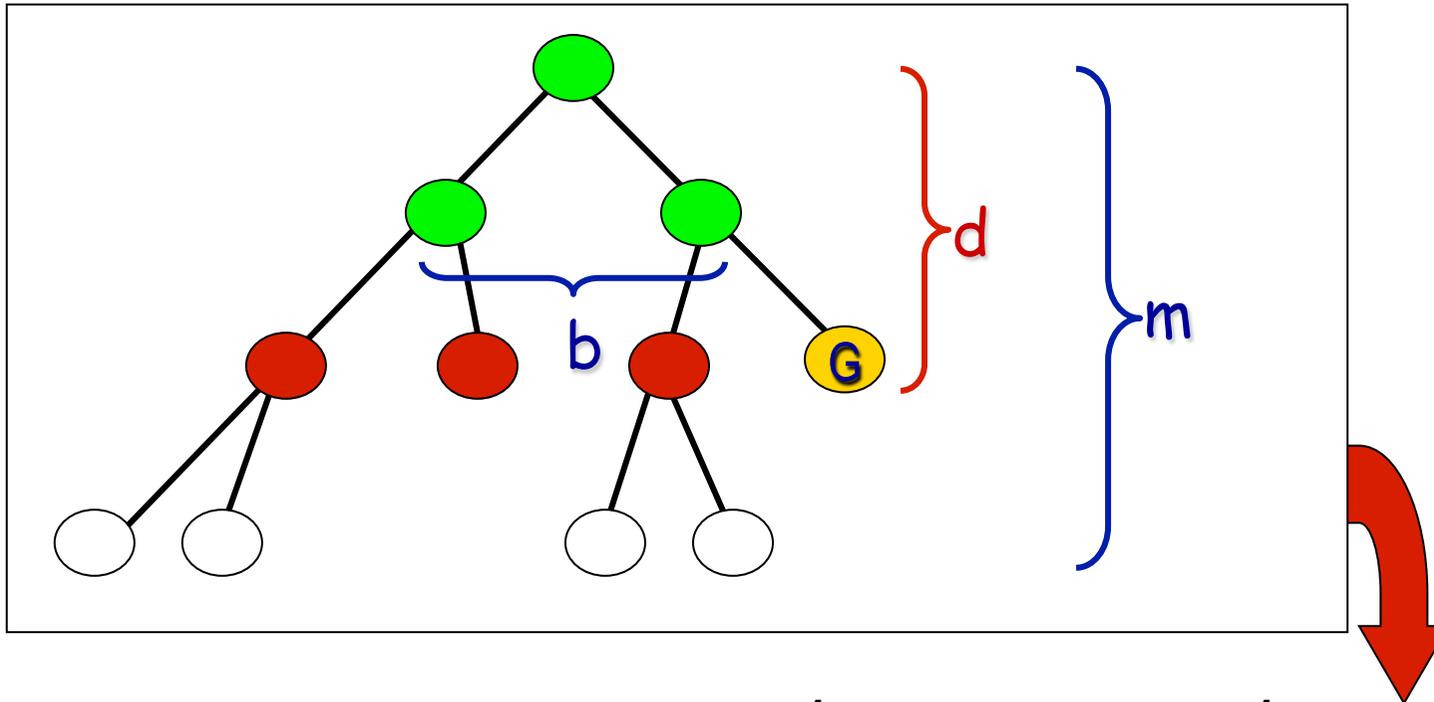
## Depth-first: complessità (2)

---

- ***La complessità temporale è invece analoga a quella in ampiezza*** (esponenziale nella profondità)
- Nel caso peggiore, se abbiamo profondità  $m$  e fattore di ramificazione  $b$  il numero massimo di nodi espansi nel ***caso peggiore sarà  $b^m$***  (complessità temporale).

## Complessità in tempo (depth-first)

- Caso peggiore:
  - goal nel ramo più profondo a destra,

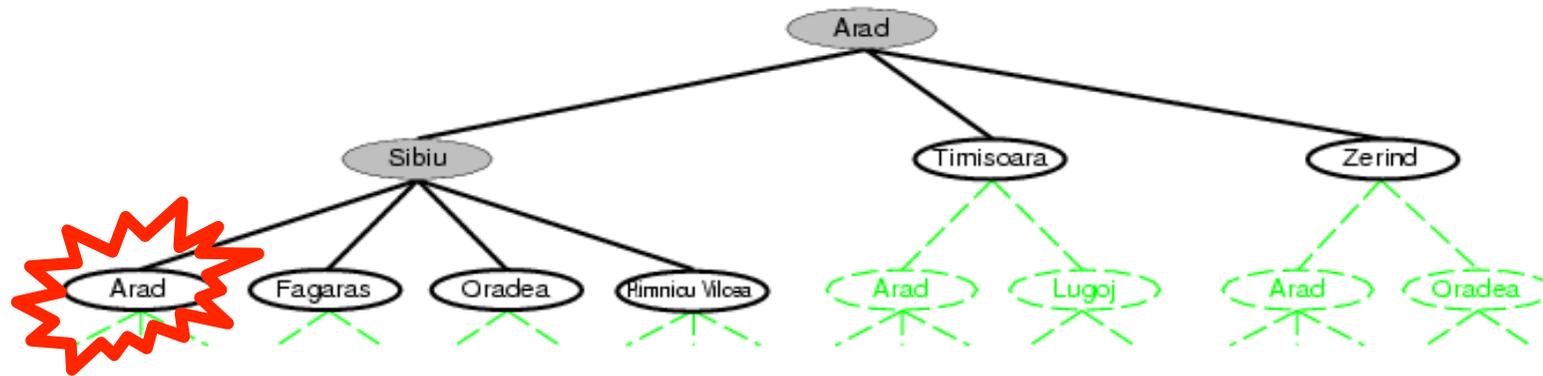


- Complessità in tempo =  $b^m + b^{m-1} + \dots + b + 1 = b^{m+1} - 1$
- Quindi:  $O(b^m)$  anche se la soluzione esiste a profondità  $d$
- $m$  può essere  $\gg d$



# Depth-first: loop

---



## 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, nel caso di cicli - grafo)

# Algoritmo di ricerca (evitando di ripetere stati)

---

- Un nodo di (un albero di) ricerca è una strada da uno stato X allo stato iniziale (ad esempio [X,B,A,S])
  - Lo stato di un nodo di ricerca è lo stato più recente della strada
  - Sia L una lista di nodi (ad esempio [[X,B,A,S], [C,B,A,S]) da espandere e *Visited* lista dei nodi già espansi
  - Sia S lo stato iniziale.
1. Inizializza L con S (e *Visited* = [])
  2. **Estrai un nodo n da L.** Se L è vuota fallisci;
  3. Se lo stato di n è il goal fermati e restituiscilo insieme alla strada percorsa per raggiungerlo (n).
  4. Altrimenti rimuovi n da L e **aggiungi a L** tutti i nodi figli di n non in *Visited*, con la strada percorsa partendo dal nodo iniziale.
  5. Aggiungi n a *Visited*
  6. Ritorna al passo 2

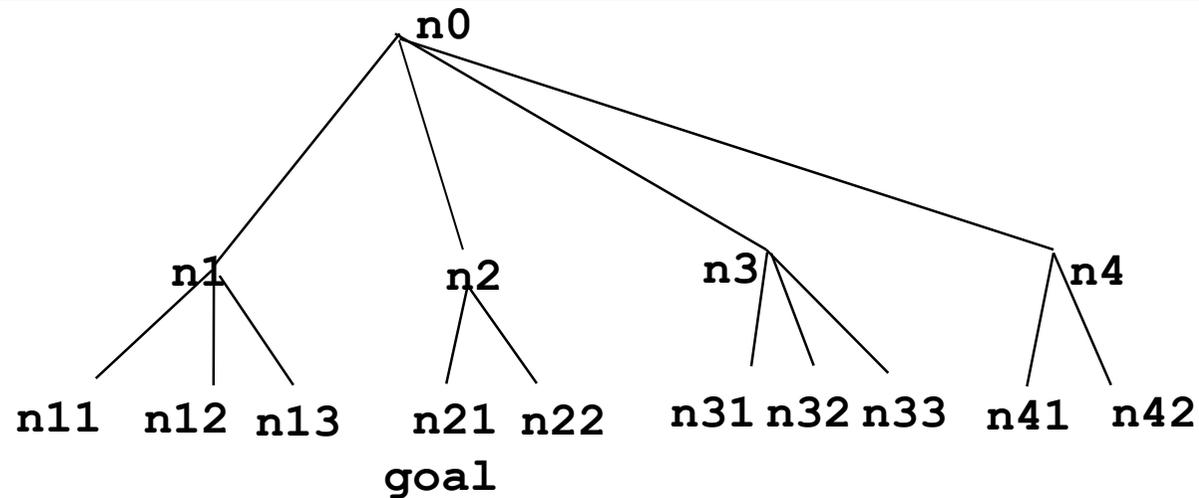
# Implementazione delle differenti Strategie di Ricerca

---

- Breadth-first
  - Estrai il primo elemento di L;
  - Aggiungi i nodi figli in coda a L
    - L è gestita come una coda (queue)
- Depth-first:
  - Estrai il primo elemento di L;
  - Aggiungi i nodi figli in testa a L
    - L è gestita come una pila (stack)

# ESEMPIO: BREADTH-FIRST

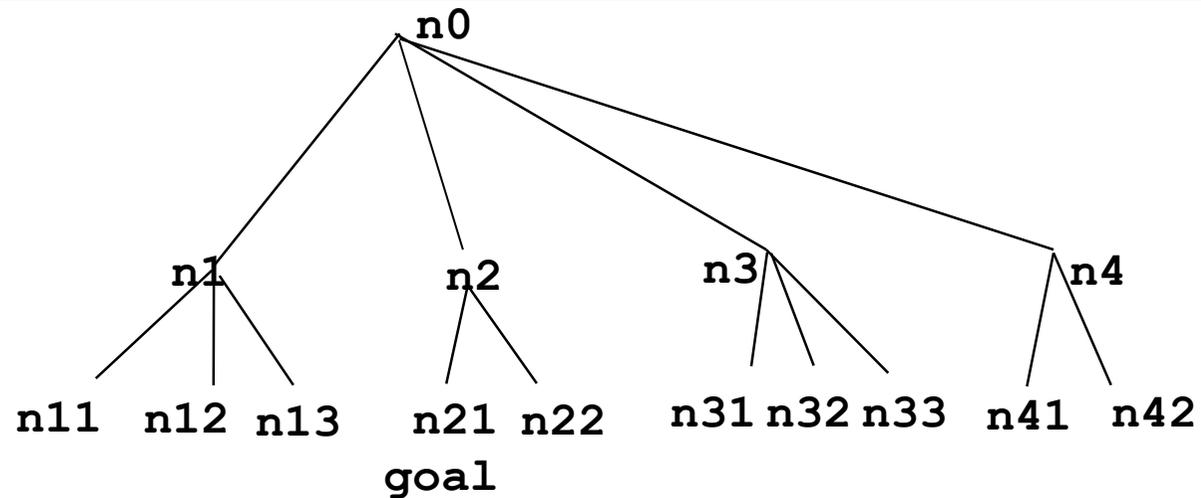
---



- Breadth-first: figli espansi aggiunti in coda a L.
  - 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

# ESEMPIO: DEPTH-FIRST

---



- Depth-first: figli espansi aggiunti in testa a L.:
  - 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

# STRATEGIE DI RICERCA

---

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

# RICERCA A PROFONDITÀ LIMITATA

---

- E' una variante della depth-first
- Per non incorrere in loop, 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 1

---

I nodi a profondità *limit* non hanno successori.

- 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? ← 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 ← RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

# STRATEGIE DI RICERCA

---

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

# RICERCA AD APPROFONDIMENTO ITERATIVO

---

- 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 (in ampiezza e in profondità). 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 preferito quando lo spazio di ricerca è molto ampio.

# Ricerca ad approfondimento iterativo – Iterative deepening search (IDS)

---

- Può emulare la breadth first mediante applicazioni ripetute della depth first con una profondità limite crescente.
  1.  $C=1$
  2. Applica **limited depth first** con limite  $C$ , se trovi una soluzione termina
  3. Altrimenti incrementa  $C$  e vai al passo 2

# Ricerca ad approfondimento Iterativo

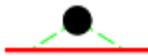
---

```
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 $l = 1$

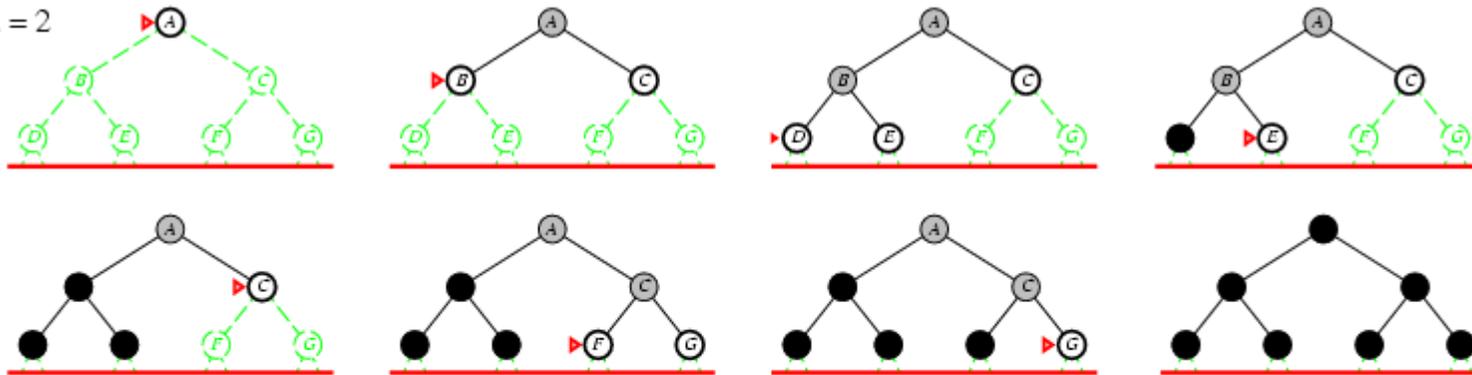
---

Limit = 1



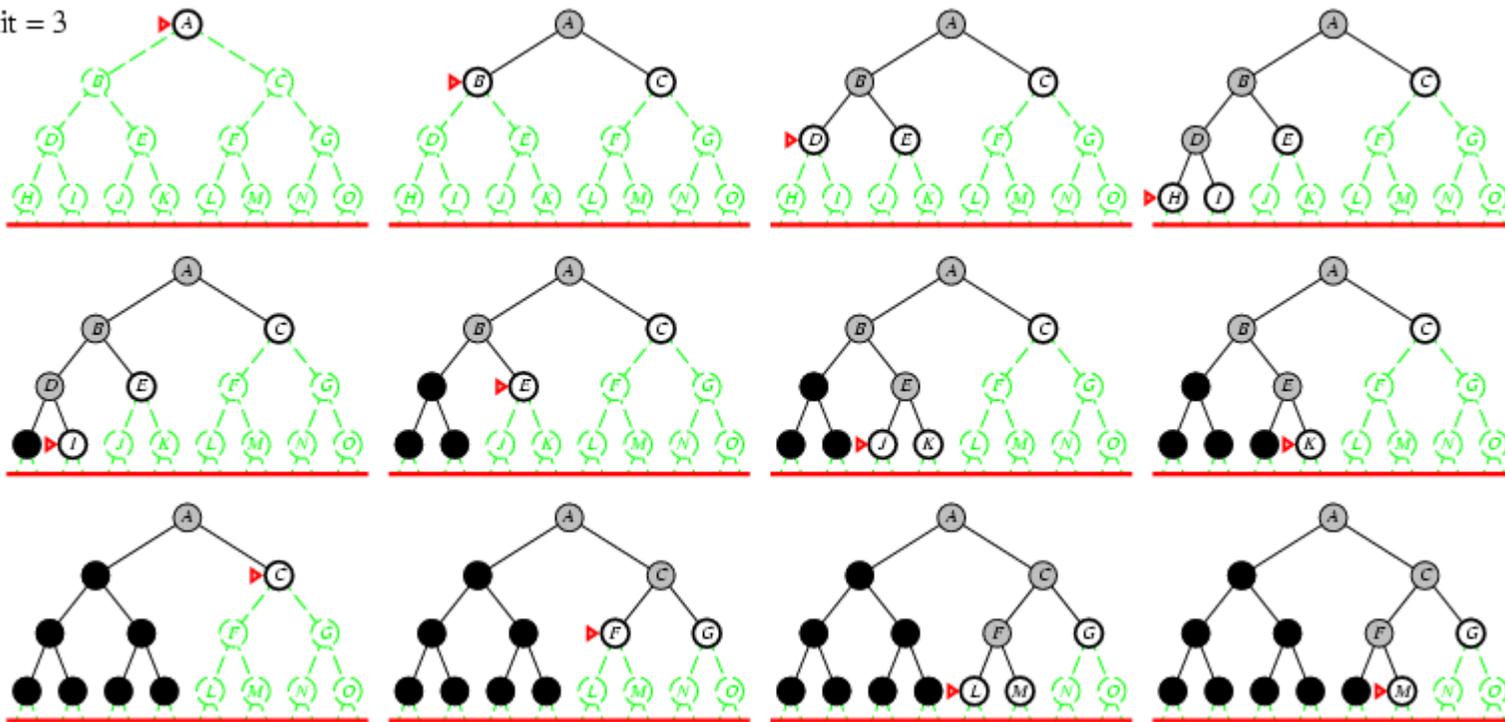
# Iterative deepening search $l=2$

Limit = 2

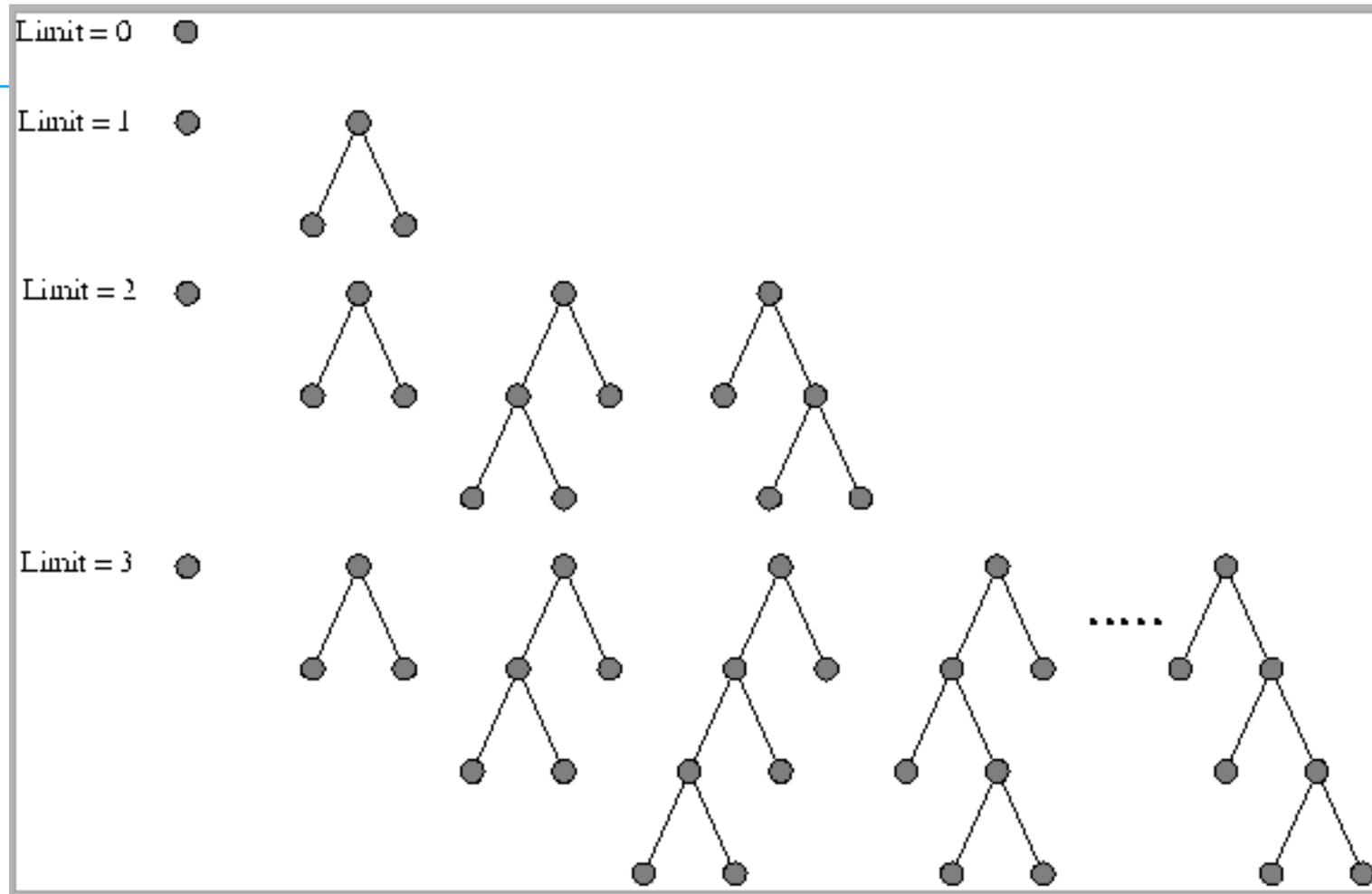


# Iterative deepening search $l=3$

Limit = 3



# Ricerca con approfondimento iterativo



Genera gli stessi stati più volte: il numero totale di espansioni è:  $(d+1)1 + (d)b + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + \mathbf{b^d}$ .

## 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

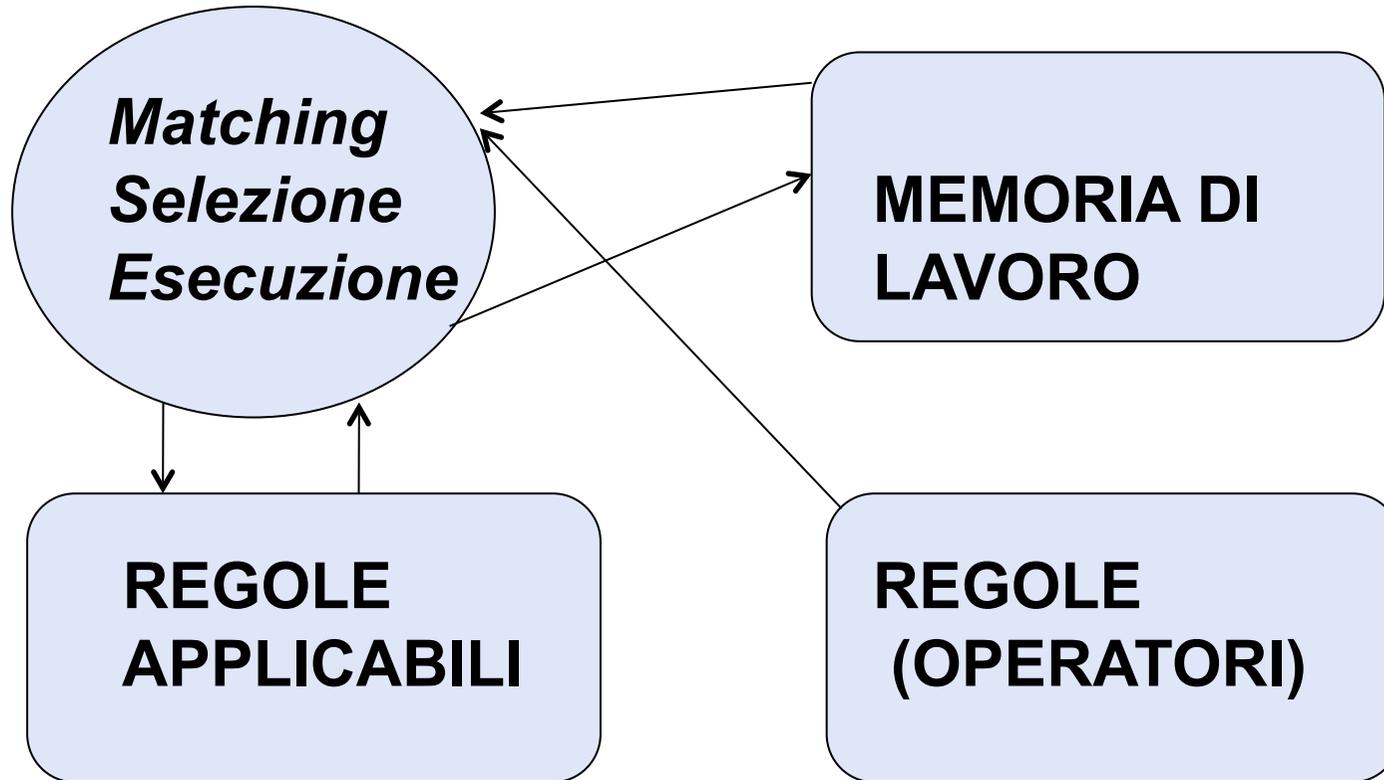
$d$  - profondità della soluzione a costo minimo

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

# ARCHITETTURA GENERALE:

---

**INTERPRETE O  
CONTROLLO**



# SISTEMI DI PRODUZIONE

---

- Insieme di Operatori (regole);
- Uno o più database (memorie di lavoro);
- Strategia di Controllo.

- Consentono modularità e flessibilità
- Operatori:

**IF <pattern> THEN <body>**

- non si chiamano per nome, ma si attivano in base al *pattern-matching*

# ESEMPIO BASE DI CONOSCENZA

---

- FATTI:

`gram(neg) .`  
`not(allergic(antb)) .`

- REGOLE:

– R1: `gram(neg) → id(ecoli) .`

Se il risultato dell'esame è *gram-negativo* allora l'identità è *enterium-coli*

– R2: `gram(pos) → id(strep) .`

Se il risultato dell'esame è *gram-positivo* allora l'identità è *streptococco*

– R3: `id(strep) OR id(bact) → ind(pen) .`

Se l'identità è streptococco o bactero allora è bene indicare penicillina

– R4: `ind(X) AND not(allergic(X)) → prescribe(X)`

Se è bene indicare una certa medicina e il paziente non è allergico a tale medicina, allora si può prescrivere tale medicina al paziente

– R5: `id(ecoli) → ind(antb) .`

Se l'identità è *enterium-coli* allora è bene indicare antibiotici

# Production Rule Systems (PRS):

---

- Definizione:

Programmi che realizzano metodi di ricerca per problemi rappresentati come spazio degli stati

- Consistono di:

- Un insieme di regole,
- una 'working memory' , che contiene gli stati correnti raggiunti
- una strategia di controllo per selezionare le regole da applicare agli stati della 'working memory' (matching, verifica di precondizioni e test sullo stato goal se raggiunto).

# DUE MODALITA' DI "RAGIONAMENTO"

---

- FORWARD O DATA-DRIVEN:
  - La memoria di lavoro nella sua configurazione iniziale contiene la conoscenza iniziale sul problema, cioè i fatti noti.
  - Le regole di produzione applicabili sono quelle il cui antecedente può fare *match con la memoria di lavoro (F-rules)*.
  - Ogni volta che una regola è selezionata ed eseguita, i nuovi fatti (che compaiono nel conseguente) sono aggiunti alla memoria di lavoro.
  - Il procedimento termina con successo quando nella memoria di lavoro è inserito (o verificato) il goal da dimostrare (condizione di terminazione).

# DUE MODALITA' DI "RAGIONAMENTO"

---

- BACKWARD O GOAL-DRIVEN:
  - La memoria di lavoro iniziale contiene il goal (o i goal) del problema.
  - Le regole di produzione applicabili sono quelle il cui conseguente può fare *match con la memoria di lavoro (B-rules)*.
  - Ogni volta che una regola viene selezionata ed eseguita, si inseriscono nuovi *subgoal* da dimostrare nella memoria di lavoro.
  - Il procedimento termina con successo quando nella memoria di lavoro vengono inseriti fatti noti (condizione di terminazione).

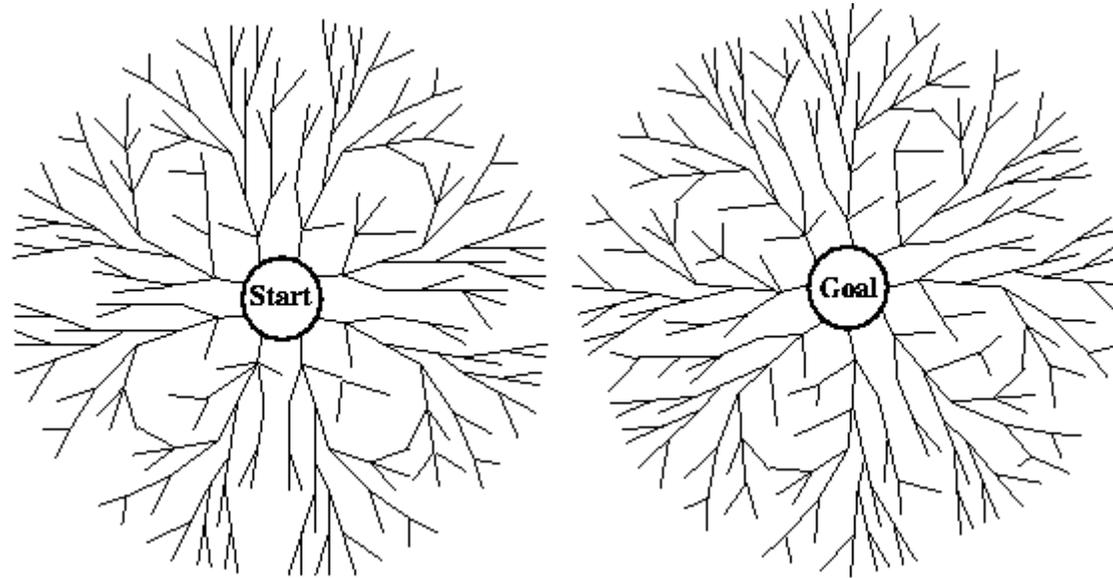
# QUANDO APPLICARE BACKWARD E QUANDO FORWARD?

---

- Esistono più fatti iniziali o più goal?
- Quale è il numero medio di rami generati da un singolo nodo?
- Quale è la modalità di ragionamento più naturale? (spiegazione all'utente)
- **BIDIREZIONALE O MISTO:**
  - È la combinazione dei metodi descritti precedentemente;
  - 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)

# Ricerca bidirezionale

---



L'idea è eseguire due ricerche in parallelo, una in avanti dallo stato iniziale e l'altra all'indietro partendo dall'obiettivo, fermandosi quando si incontrano a metà strada.

Il suo uso è motivato dal fatto che  $b^{d/2} + b^{d/2}$  è molto più piccolo di  $b^d$

(ciascuna ricerca, prima di espandere un nodo, deve controllare che non appartenga alla frontiera dell'altra)

## Confronto fra le strategie di ricerca

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

b = fattore di ramificazione; d = profondità della soluzione; m=profondità massima dell'albero di ricerca; l=limite di profondità.

# Esercizi

---

## Strategie ricerca non informate

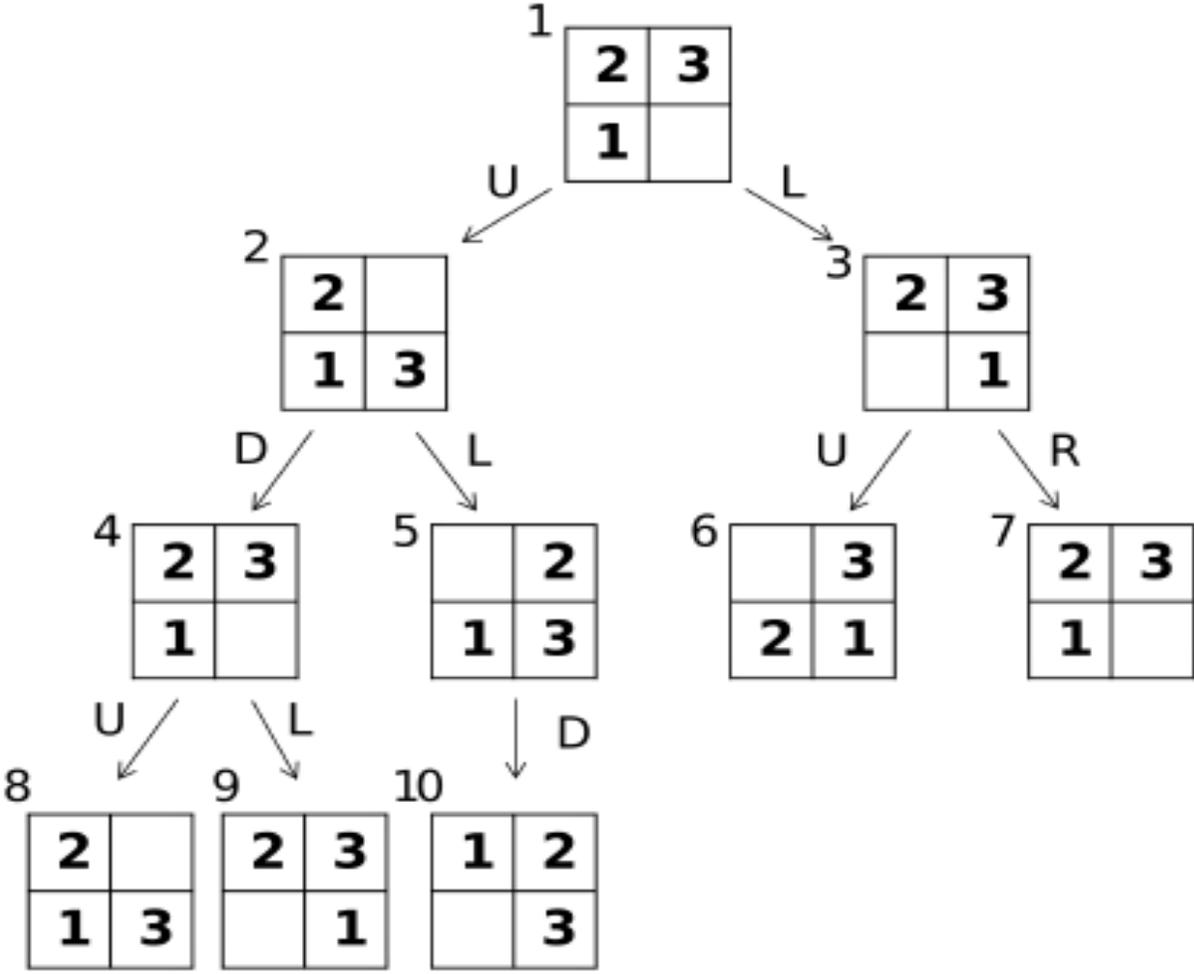
## Esercizio (compito del 28 Gennaio 2016)

- Si consideri il problema 3-puzzle (una versione semplificata dell' 8-puzzle, o gioco del filetto) in cui la griglia è 2 x 2 e ci sono tre piastrelle, numerate 1 , 2 , e 3 , e uno spazio vuoto. Si hanno **quattro operatori, che muovono il vuoto su, giù, a sinistra, o a destra.**
- Lo stato iniziale (Start) e obiettivo (Goal) sono indicati di seguito:

Start	Goal								
<table border="1"><tr><td>2</td><td>3</td></tr><tr><td>1</td><td></td></tr></table>	2	3	1		<table border="1"><tr><td>1</td><td>2</td></tr><tr><td></td><td>3</td></tr></table>	1	2		3
2	3								
1									
1	2								
	3								

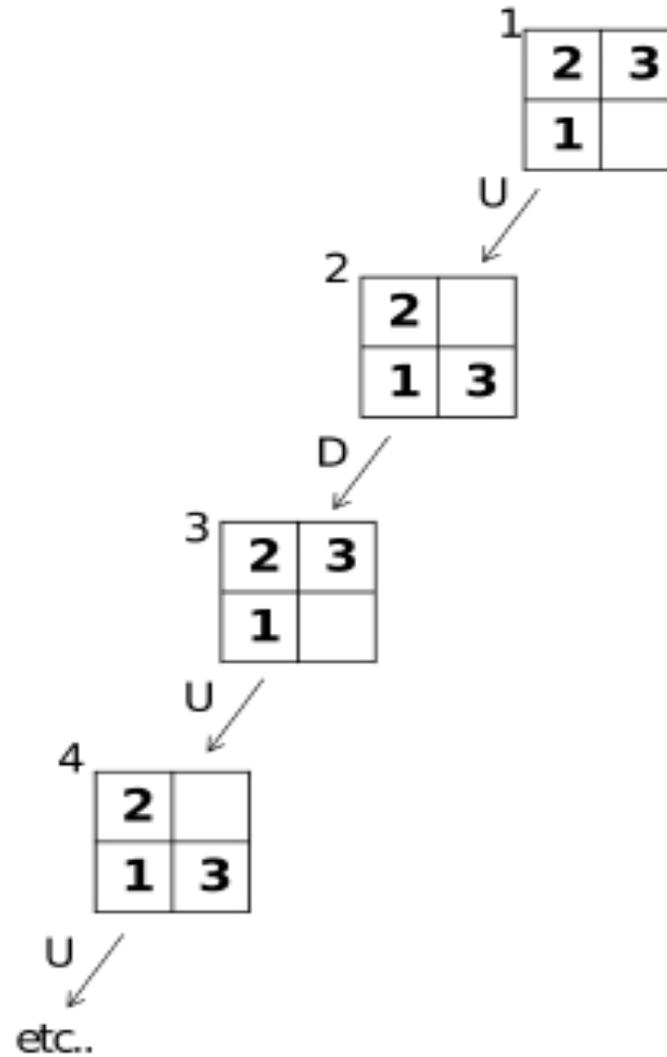
- Mostrare come si individua la **sequenza di azioni che porta da Start a Goal**, sviluppando lo spazio di ricerca utilizzando le seguenti strategie:
- breadth-first
- depth-first
- Si assuma che le strategie di ricerca non “ricordino” i nodi visitati in precedenza. Inoltre, si utilizzino gli operatori (su, giù, sinistra, destra) nell'ordine indicato, a meno che il metodo di ricerca non stabilisca in diverso modo. Nello sviluppo dell'albero, etichettare ogni nodo visitato con un numero che indica l'ordine di visita. Si indichino inoltre quale/quali strategia/e porta ad una situazione di loop.

# Breadth-first



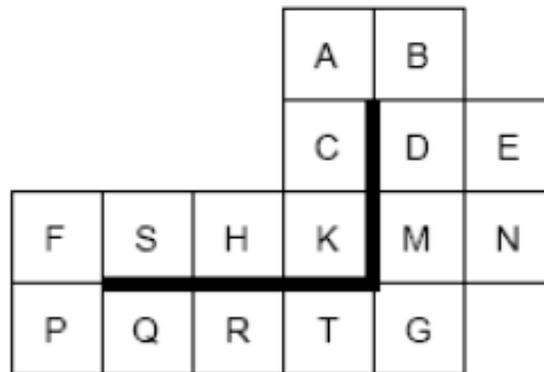
# Depth-first (loop)

---



## Esercizio (compito del 15 Giugno 2010)

- Si consideri la seguente scacchiera in cui i successori di ciascuna cella sono tutte le celle adiacenti in direzione Nord, Sud, Ovest e Est (in questo ordine), eccettuato per le celle sui confini della scacchiera o in corrispondenza del confine rappresentato dalla linea in grassetto. Per esempio,  $\text{successori}(M) = \{D, G, N\}$ . Si assuma che ogni mossa abbia costo 1.



- Si vuole trovare un cammino dalla cella S alla cella G. Nel caso di scelte multiple data una strategia, si utilizzi l'ordine alfabetico delle celle. Si descriva l'ordine di espansione dei nodi per:
  - *Breadth-First Search* (senza visite ripetute dello stesso nodo)
  - *Depth-First Search*. Si assuma che la strategia identifichi i cicli e li elimini, evitando di espandere un nodo già visitato.

# Soluzione

Breadth-first, frontiera:

[S]

[F,H]

[H,P]

[P,K]

[K,Q]

[Q,C]

[C,R]

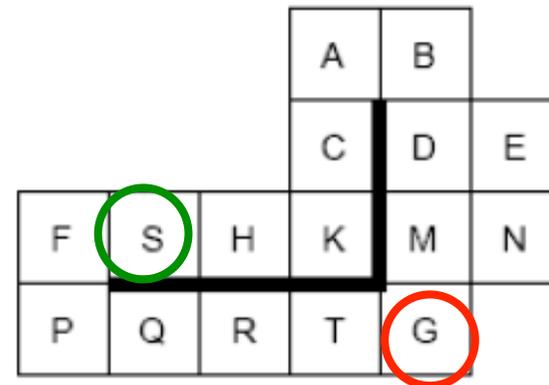
[R,A]

[A,T]

[T,B]

[B,G]

[G,D] → goal G, soluzione: SFHPKQCRATBG



# Soluzione

Depth-first, frontiera:

[S]

[F,H]

[P,H]

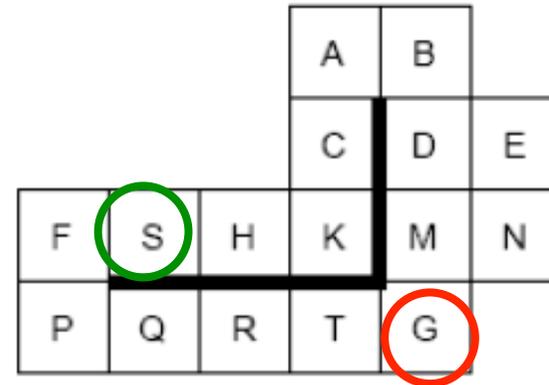
[Q,H]

[R,H]

[T,H]

[G,H] → goal G

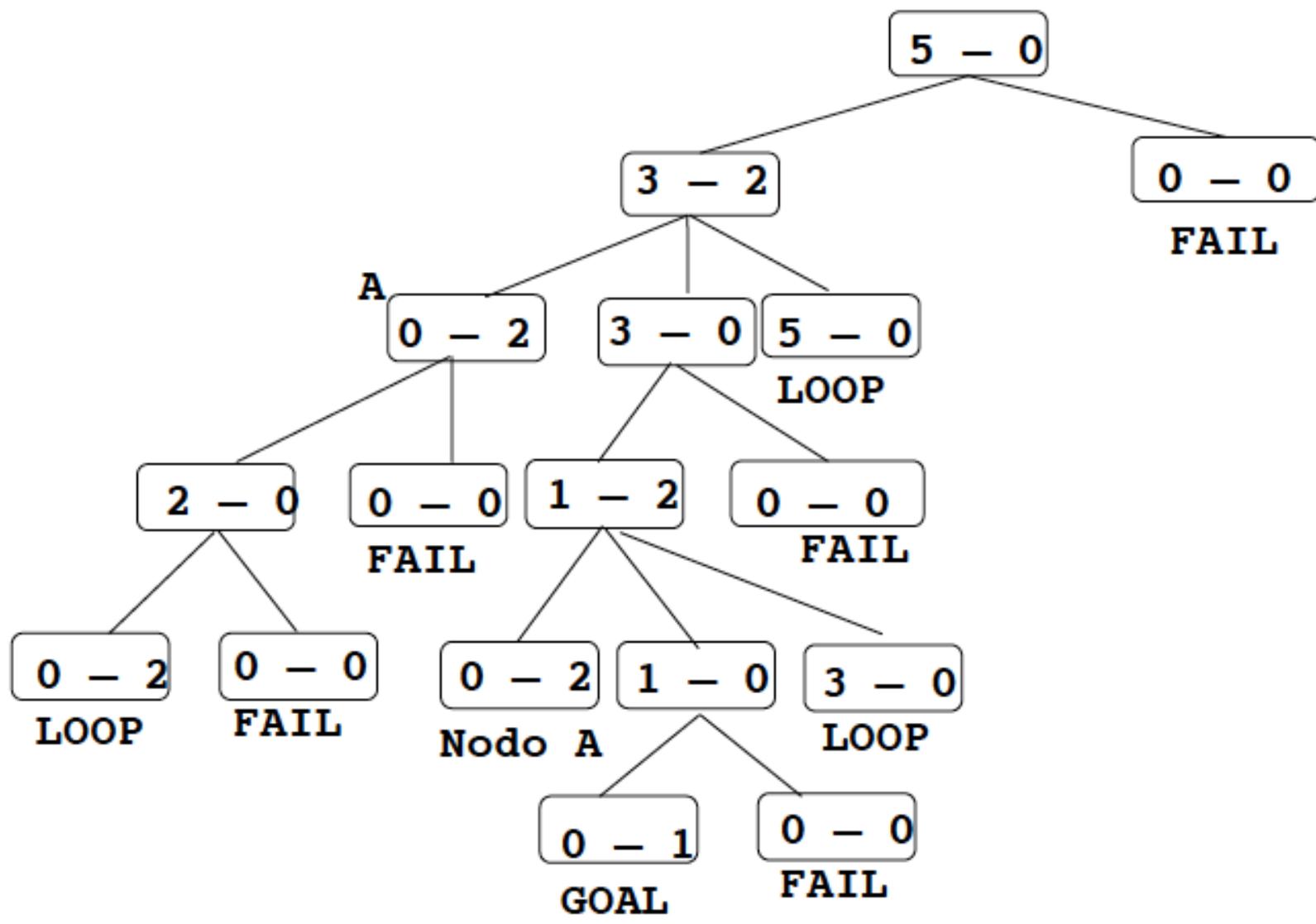
Soluzione: SFPQRTG



## Esercizio (compito del 27 Gennaio 1995)

---

- Si consideri il seguente problema. Si hanno due contenitori per liquidi, uno da 5 litri completamente pieno d'acqua ed uno da 2 litri vuoto.
- Vogliamo ottenere precisamente un litro d'acqua nel contenitore da due litri.
- Le azioni possibili sono:
  - trasferire acqua da un contenitore all'altro,
  - buttarla,
  - ma non ottenerne di nuova.
- Si descriva il problema come sistema di produzioni, si descrivano gli operatori e si mostri tutto lo spazio degli stati per risolvere il problema.
- Si indichi, inoltre dopo quanti passi si giungerebbe alla soluzione se si utilizzasse una strategia bread-first e depth-first.



# Esercizio

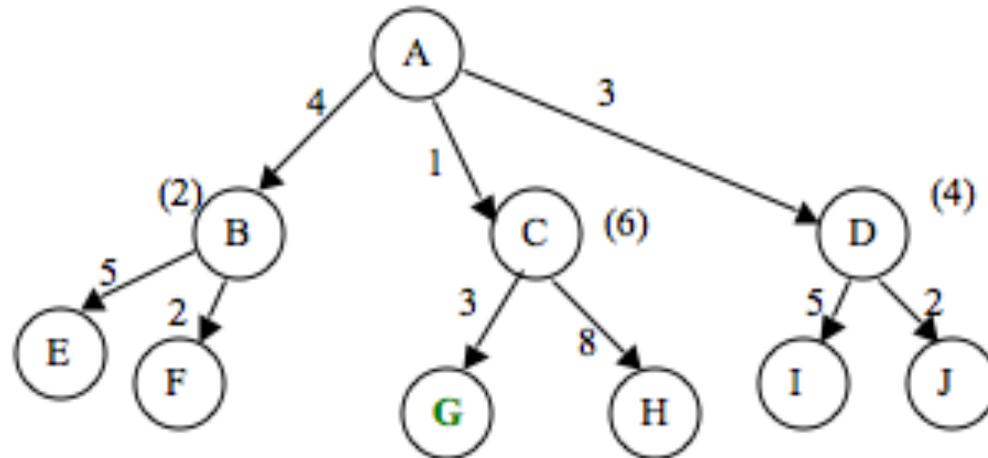
---

- Modellare un problema di ricerca di cammino in un grafo con 5 nodi e un fattore di ramificazione pari a 2 (inventarlo)
- Scegliere un nodo di partenza S e un nodo di arrivo F ed espandere l'albero che si genera nella ricerca del cammino da S a F applicando le strategie:
  - In ampiezza
  - In profondità (costi unitari per archi)
  - A Costo uniforme (costi  $>0$  epr archi, non necessariamente unitari)
  - A profondità limitata

Verificare cosa cambia inserendo il test sui nodi già visitati.

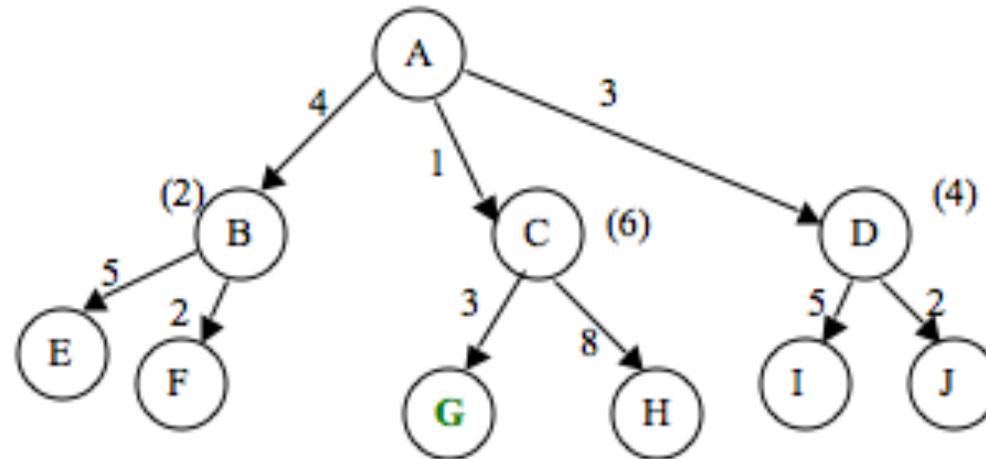
## Esercizio (9 Dicembre 2010)

- Si consideri il seguente albero di ricerca in cui i numeri sugli archi sono i costi (quelli tra parentesi vicino agli stati le stime euristiche  $h$ ). Ove  $h$  non è specificata, vale 0  $\rightarrow$  li useremo poi nelle strategie euristiche).



- Si assuma che i nodi siano espansi in ordine alfabetico, nel caso in cui la strategia di ricerca applicata li consideri equivalenti. Il goal sia lo stato G e la ricerca termini al suo raggiungimento.
- Si indichi la lista di stati espansi da:
  - Breadth First
  - Depth First
  - Iterative Deepening Search

# Soluzione(9 Dicembre 2010)



Breadth First

Depth First

Iterative Deepening Search

A B C D E F **G**

A B E F C **G**

A A B C D A B E F C **G**