Intelligenza Artificiale

Anno Accademico 2022 - 2023

Ricerca Cieca

Norme di utilizzo dei materiali didattici

La visione e l'utilizzo del presente materiale didattico è riservato agli utenti iscritti al corso al solo fine di studio e approfondimento didattico.

L'utente che accede alla sezione e-learning e che ne consulta i contenuti è tenuto a rispettare le disposizioni legislative che tutelano il diritto d'autore e pertanto è fatto espresso divieto di riprodurre, pubblicare o distribuire i materiali tratti dal presente sito, anche in forma parziale, fatta salva la possibilità di realizzare un'unica copia del materiale, in formato cartaceo e/o digitale, all'esclusivo fine di studio.

L'utente è responsabile per l'uso improprio o non autorizzato del materiale pubblicato effettuato in violazione delle suddette disposizioni.

Sommario

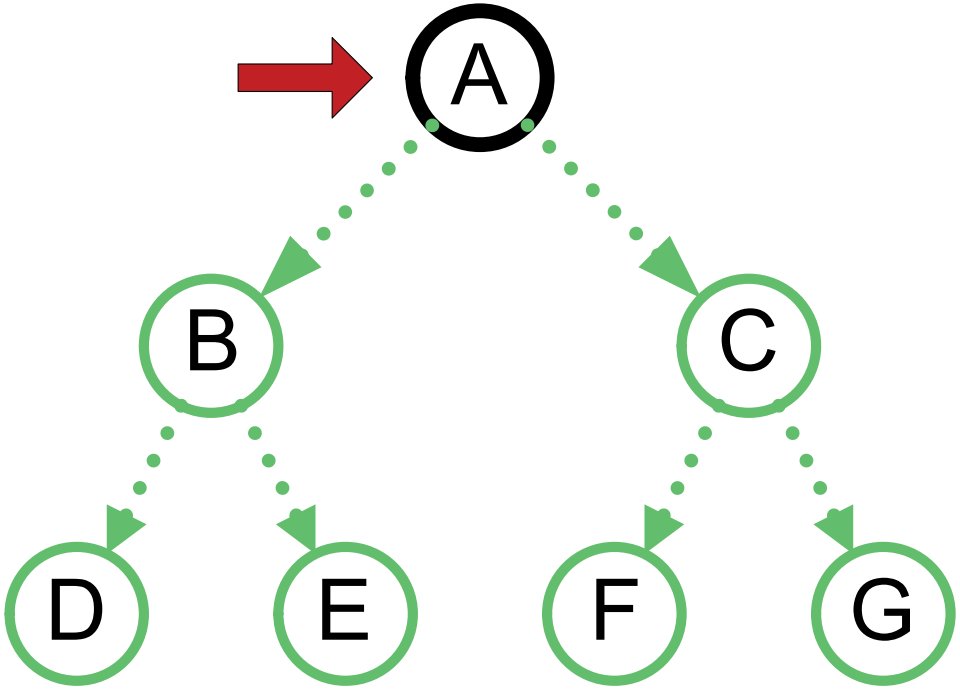
- Ricerca in Ampiezza
- Ricerca Guidata dal Costo
- Ricerca in Profondità
- Ricerca in Profondità Limitata
- Ricerca per Approfondimenti Successivi

Ricerca in ampiezza (1)

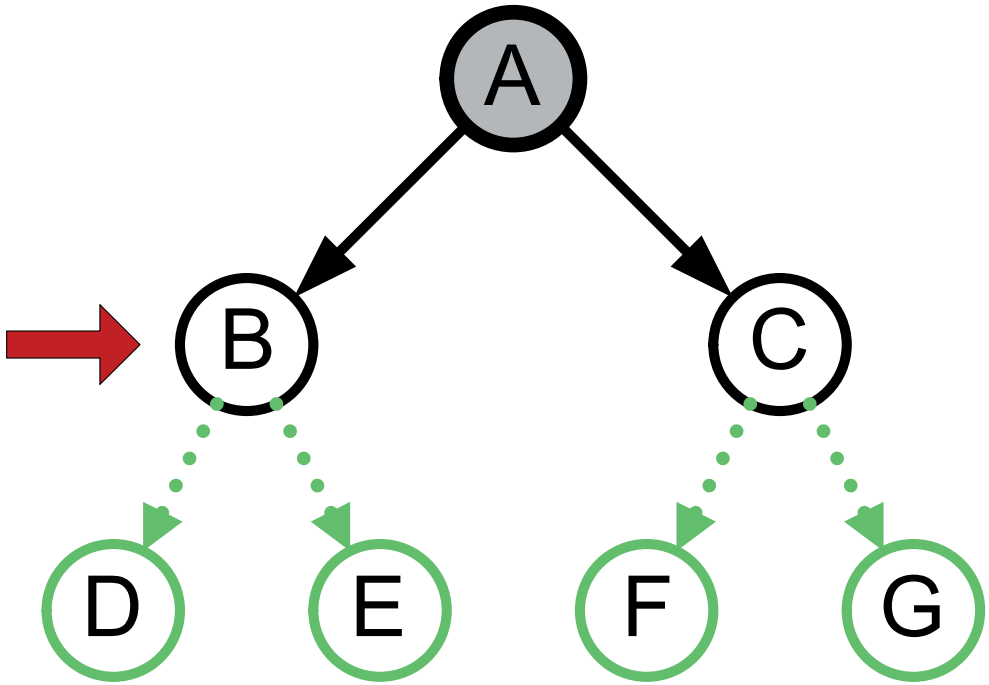
Nella **Ricerca in Ampiezza** si procede come segue:

- si espande il nodo radice
- si espandono i nodi generati dalla radice
- si espandono i loro successori e così via

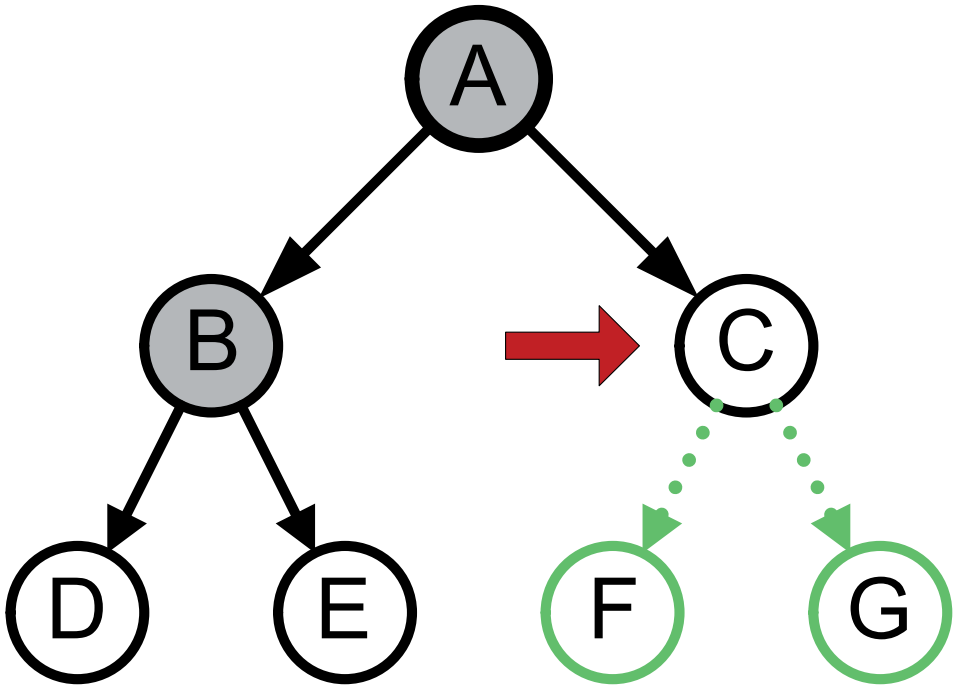
Ricerca in ampiezza (2)



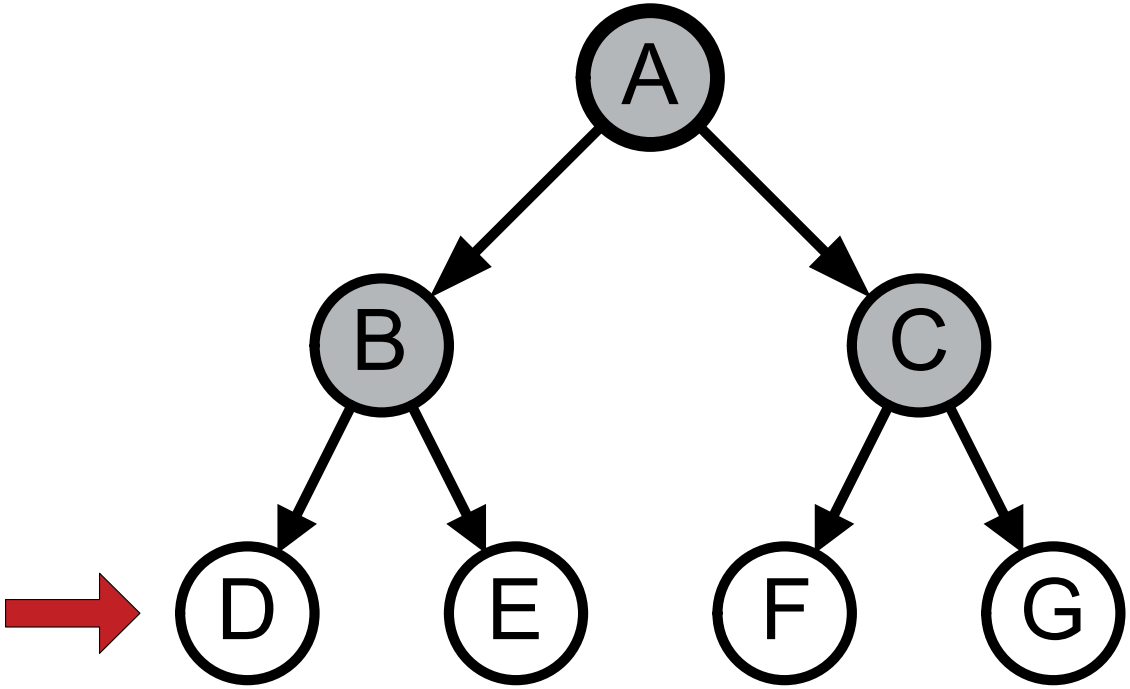
Ricerca in ampiezza (2)



Ricerca in ampiezza (3)



Ricerca in ampiezza (4)



Ricerca in ampiezza (5)

Funzione di inserimento nella coda:
ENQUEUE-AT-END

```
function BREADTH-FIRST-SEARCH(problem) returns a solution or
                                                    failure
fringe ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem])
loop do
     If EMPTY?(fringe) then return failure
     node ← REMOVE-FRONT(fringe)
     if GOAL-TEST(problem, STATE[node]) then return SOLUTION(node)
     fringe ← ENQUEUE-AT-END(fringe,EXPAND(node,OPERATORS(problem)))
end
```

Ricerca in ampiezza (6)

- Tutti i nodi di profondità d sono espansi prima di quelli di profondità d+1.
- Strategia sistematica. Trova sicuramente (se c'è) la soluzione realizzabile con il cammino più breve, cioè trova gli stati obiettivo più superficiali. Ma il cammino più breve non è necessariamente quello a costo minimo.

Proprietà della ricerca in ampiezza (1)

Completezza: sì

Ottimalità: sì, **se:**

il costo di cammino è una funzione monotòna non decrescente della profondità del nodo, ossia se:

$$depth(n) < depth(m) \Rightarrow path\_cost(n) \leq path\_cost(m)$$

$$depth(n) = depth(m) \Rightarrow path\_cost(n) = path\_cost(m)$$

Proprietà della ricerca in ampiezza (2)

Complessità in tempo: $O(b^d)$, con

b = fattore di ramificazione dell'albero (ogni nodo ha al massimo b figli)

d = lunghezza minima di un cammino dal nodo iniziale alla soluzione

Il massimo numero di nodi da generare prima di trovare una soluzione è:

(nodi del livello 1) + (nodi del livello 2) + ...
... + (nodi del livello d) + (nodi del livello $(d+1)$ - b)

$$\leq b + b^2 + \ldots + b^d + (b^{d+1} - b)$$

Complessità in spazio: $O(b^d)$

Tutte le foglie dell'albero devono essere conservate in memoria.

Tempo e memoria richiesti per la ricerca in ampiezza

Assumendo un branching factor di b=10, generazione di 1000 nodi al secondo, e 100 bytes per conservare il nodo:

Profondità	Nodi	Tempo		Spazio	
0	1	1	*millisecondo*	100	*bytes*
2	111	.1	*secondo*	11	*kilobytes*
4	11.111	11	*secondi*	1	*megabyte*
6	10^6	18	*minuti*	111	*megabytes*
8	10^8	31	*ore*	11	*gigabytes*
10	10^{10}	128	*giorni*	1	*terabyte*
12	10^{12}	35	*anni*	111	*terabytes*
14	10^{14}	3500	*anni*	11.111	*terabytes*

Ricerca guidata dal costo
(uniform cost search)

Modifica la ricerca in ampiezza espandendo il nodo nella frontiera con costo più basso (l'algoritmo di Dijkstra è all'origine di questa ricerca).

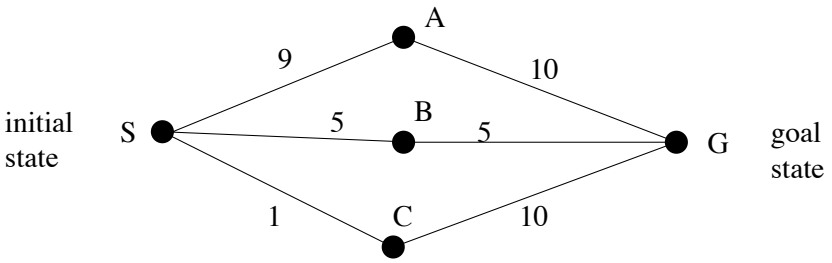
g(n) è il costo del cammino dalla radice a **n**.

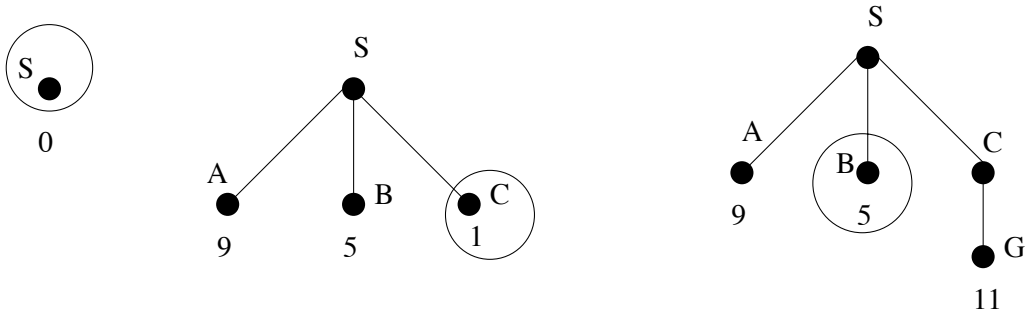
Viene scelto per l'espansione il nodo n il cui costo g(n) è minore.

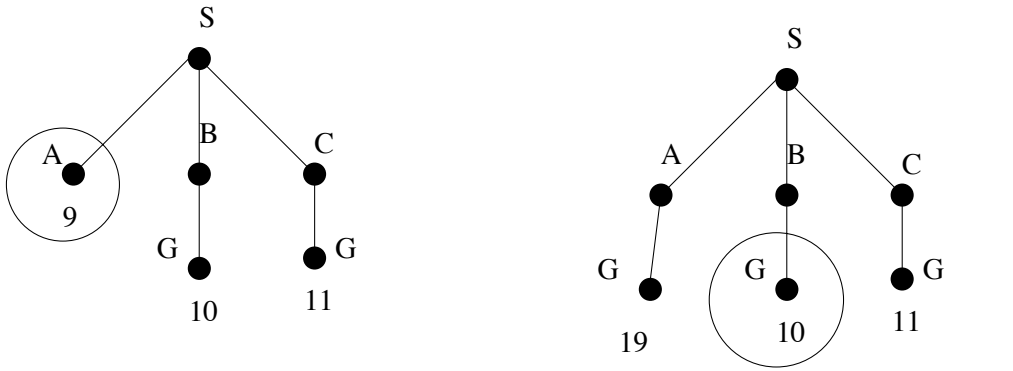
Se $g(n) = depth(n)$ si ha la ricerca in ampiezza.

E' completo e ottimale se il costo di ogni step $(g(successor(n)) - g(n))$ è sempre maggiore o uguale a una costante positiva **ε**.

Esempio


A
9
10
initial
state
S
5
B
5
G
goal
state
1
C
10


S
0
S
A
B
C
9
5
1
S
A
B
C
9
5
G
11

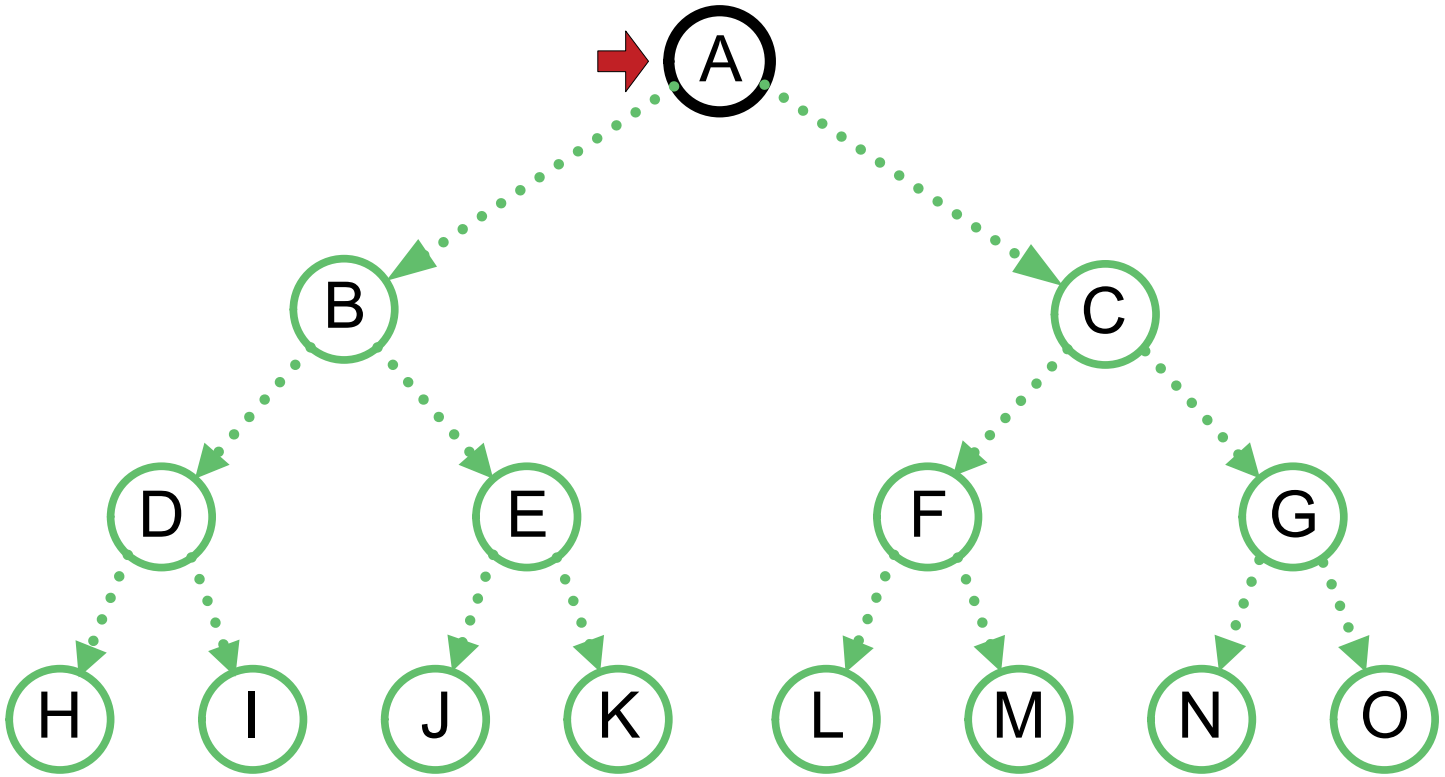

S
A
B
C
9
G
G
10
11
S
A
B
C
G
G
G
19
10
11

Ricerca in profondità (1)

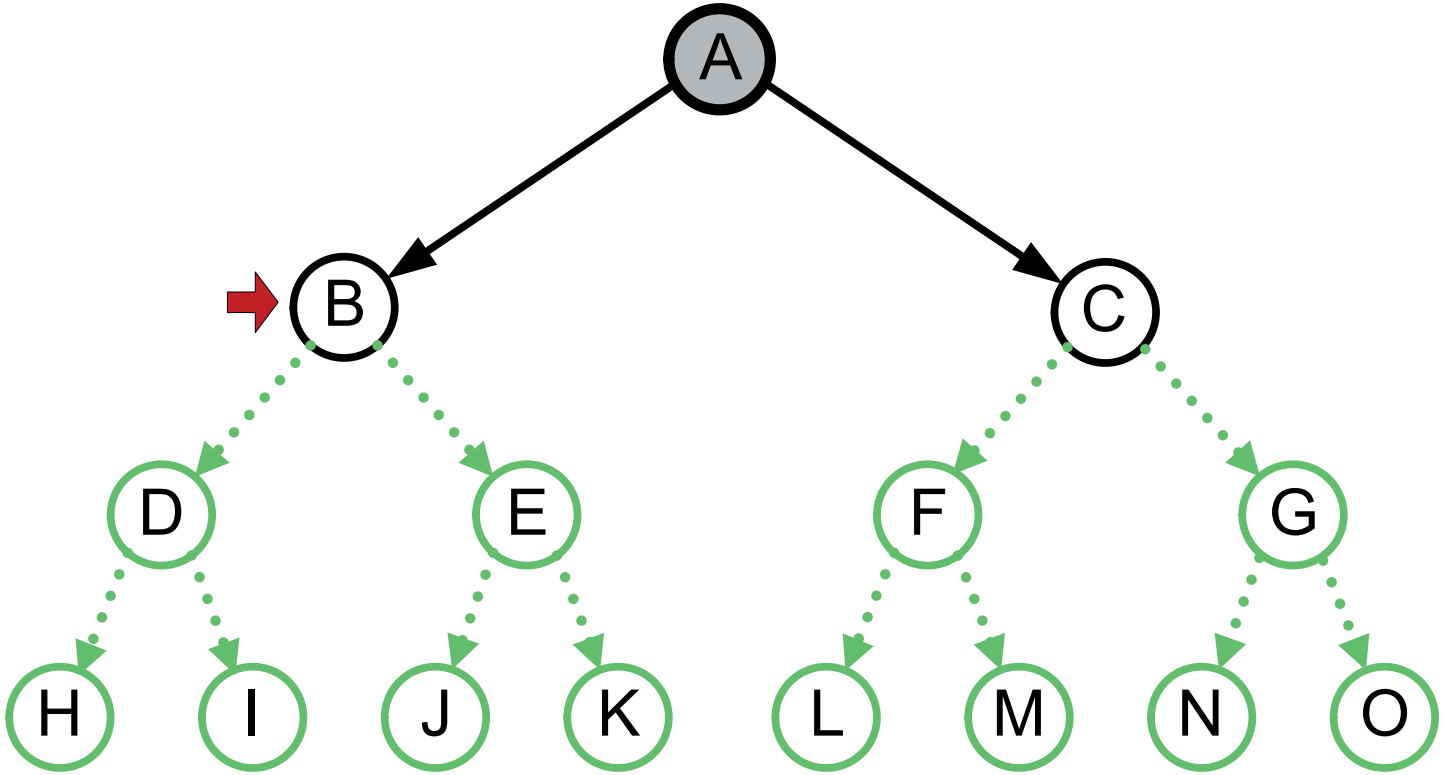
Nella **Ricerca in Profondità** si procede come segue:

- si espande il nodo radice.
- Si procede espandendo sempre per primo il nodo più **profondo** nella frontiera corrente dell'albero di ricerca.

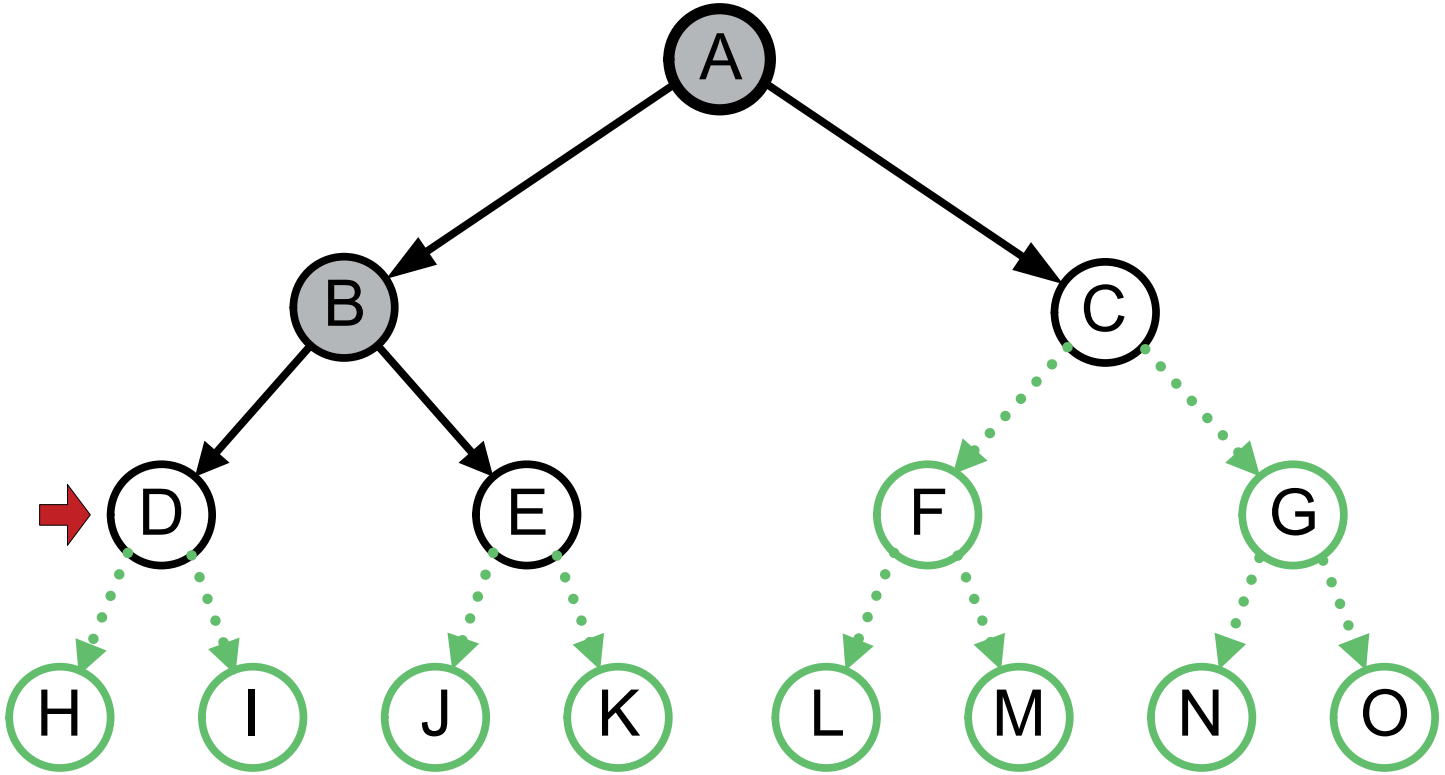
Ricerca in profondità (2)



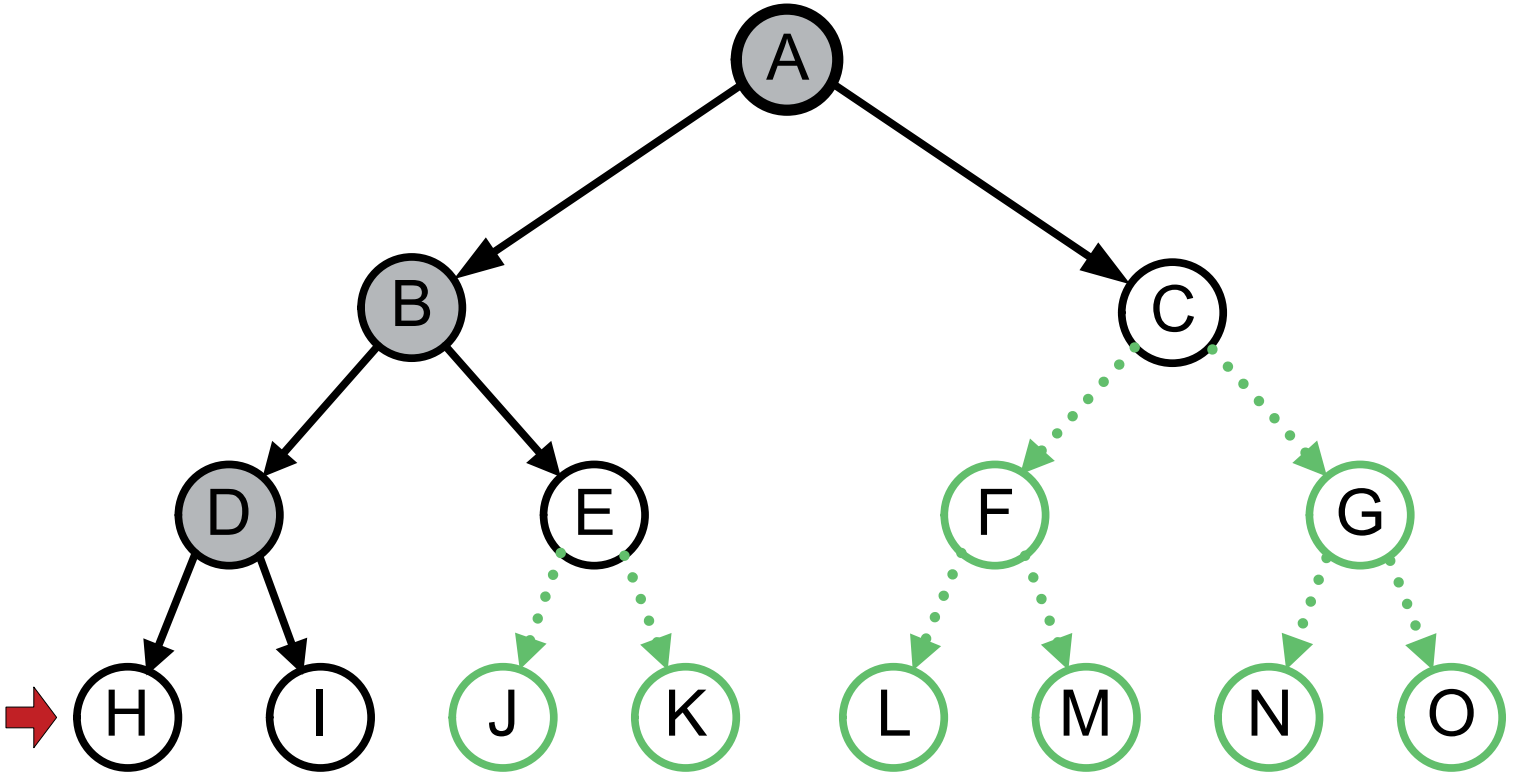
Ricerca in profondità (3)



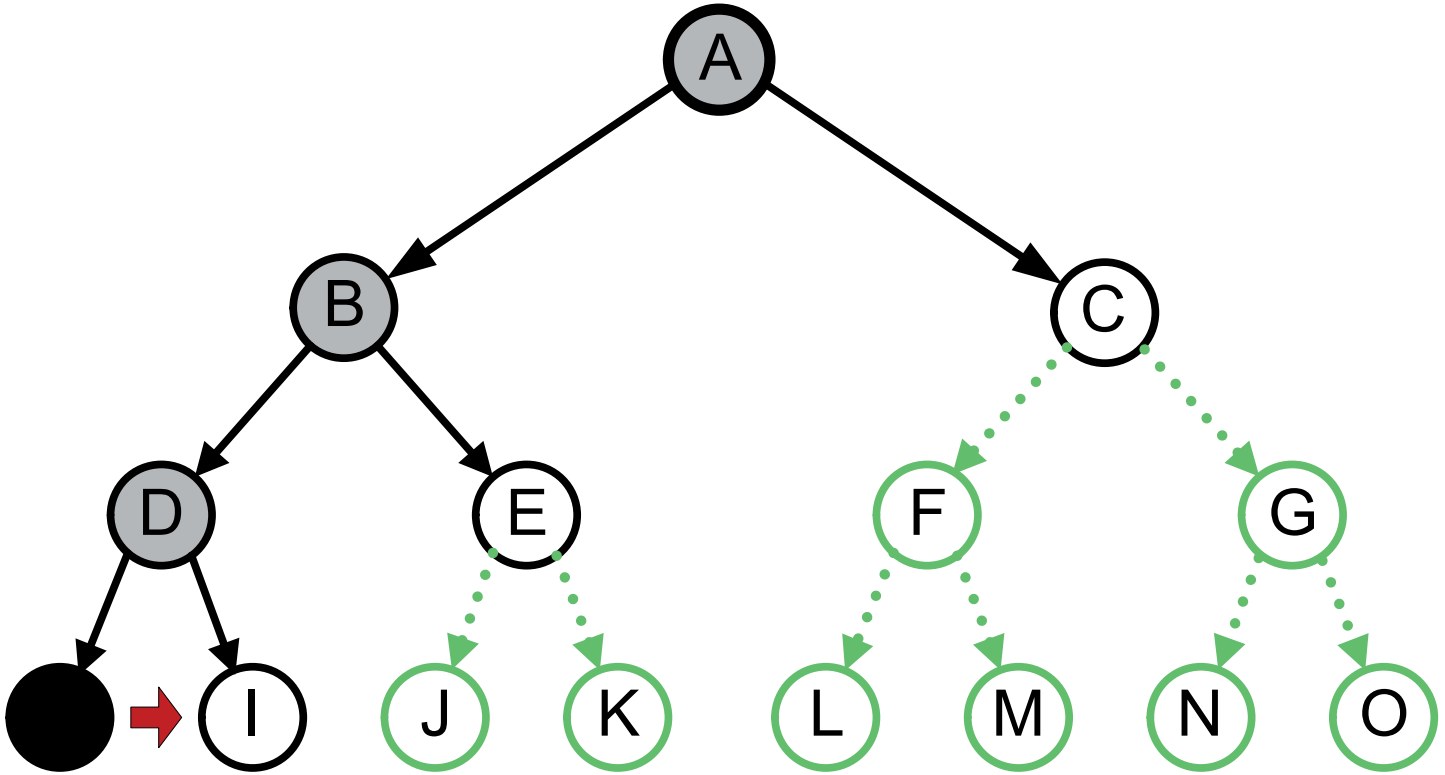
Ricerca in profondità (4)



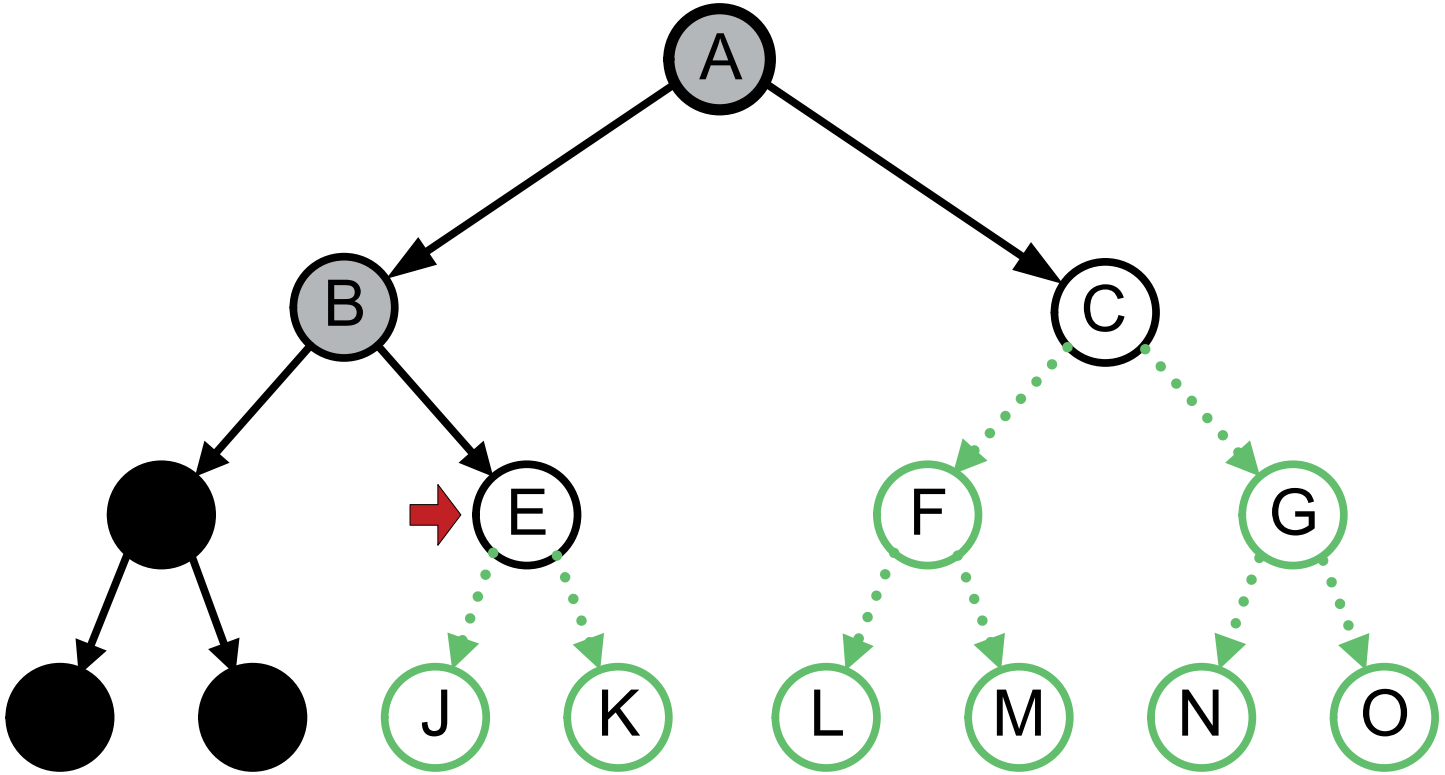
Ricerca in profondità (5)



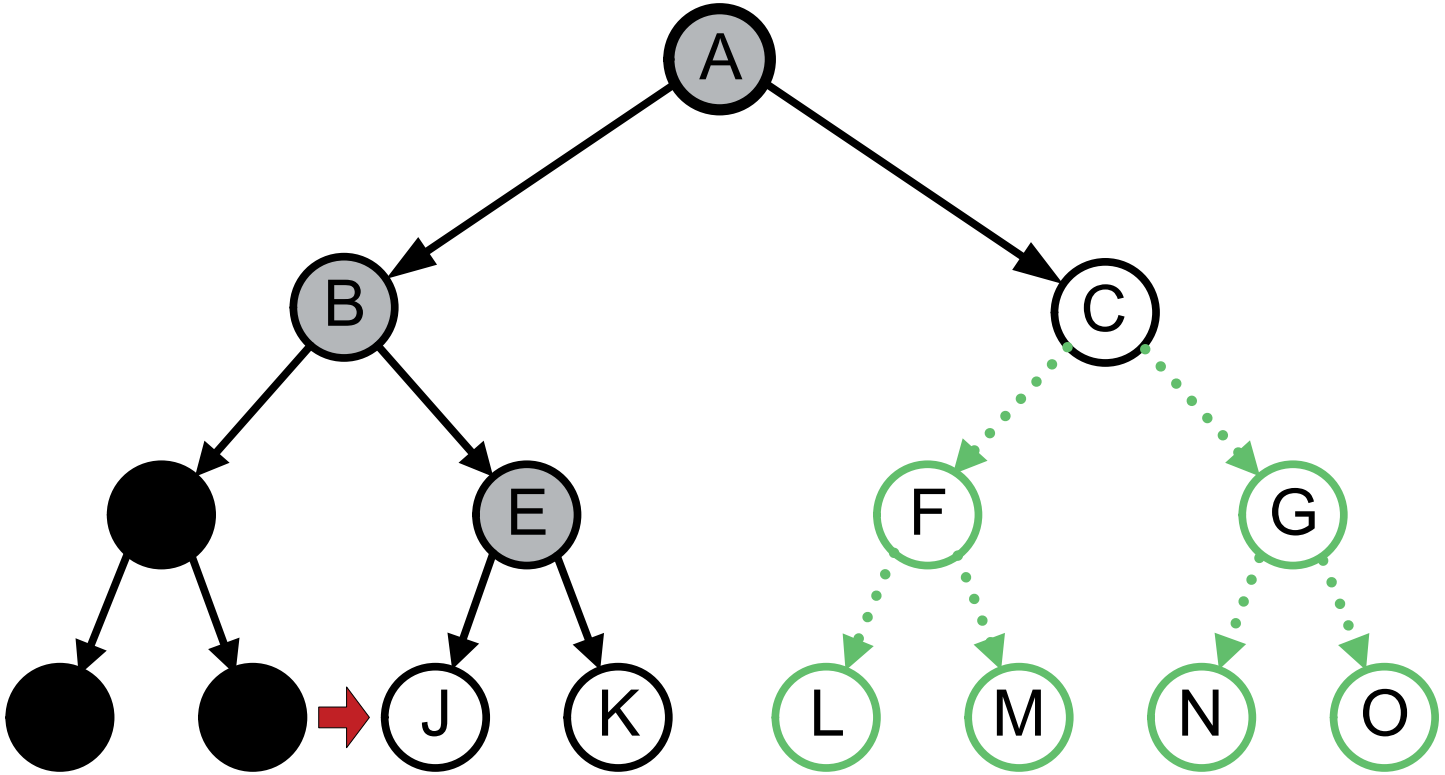
Ricerca in profondità (6)



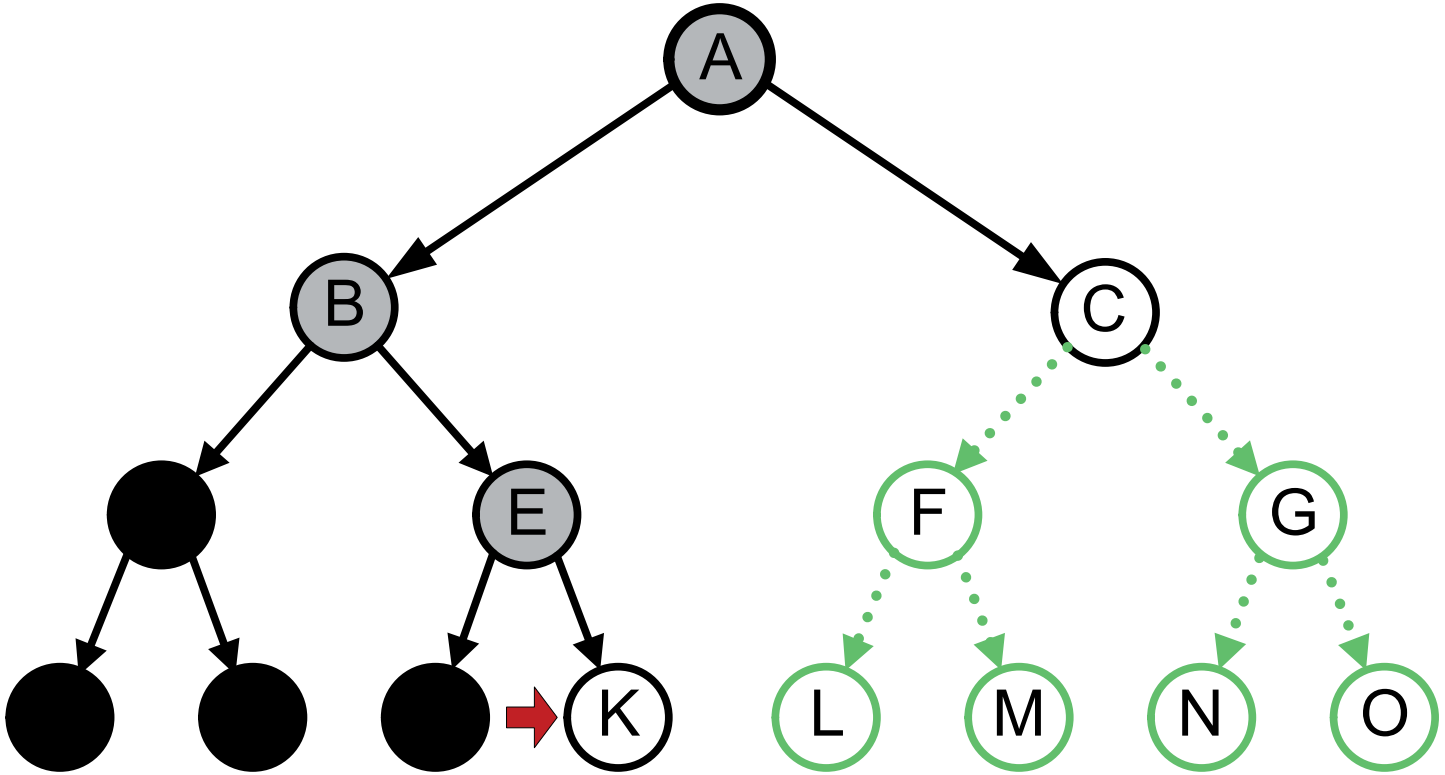
Ricerca in profondità (7)



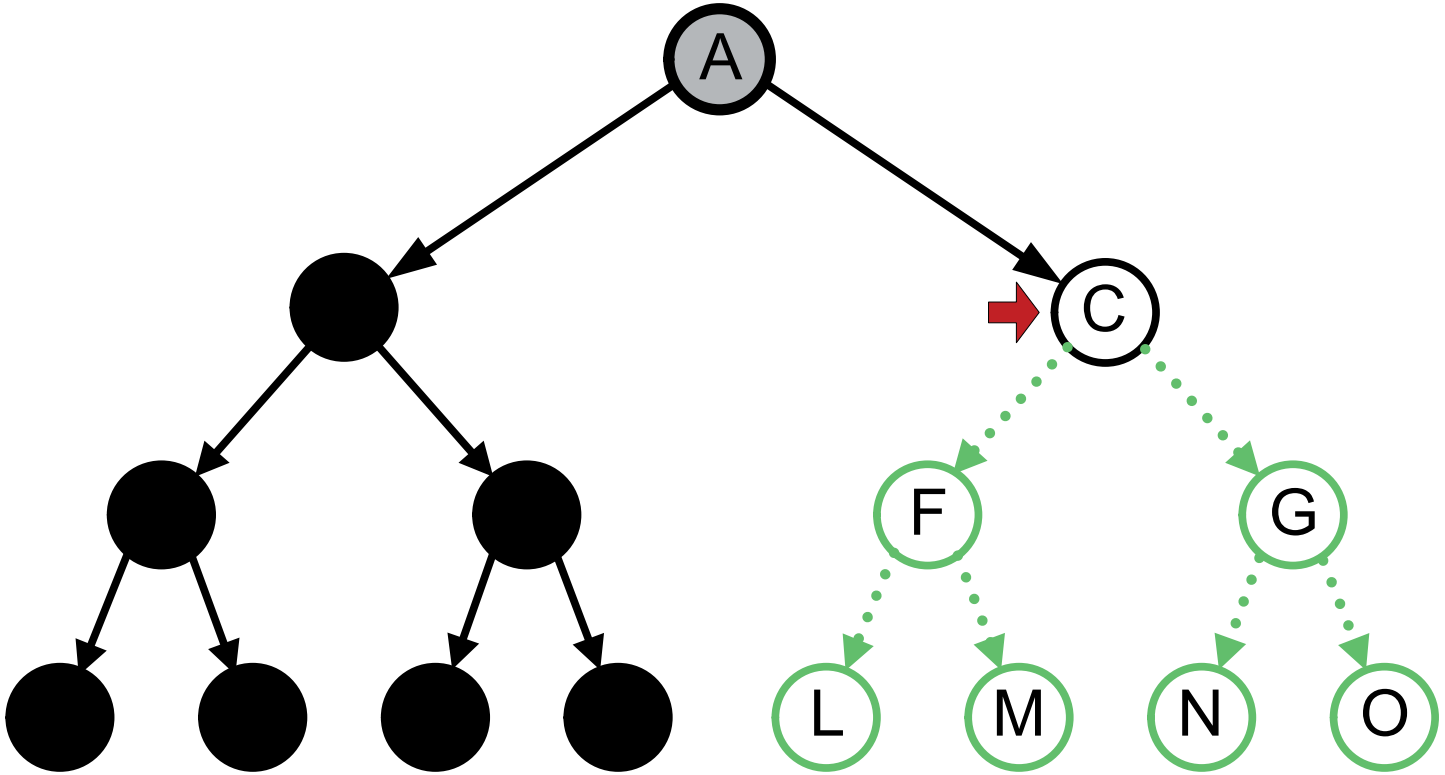
Ricerca in profondità (8)



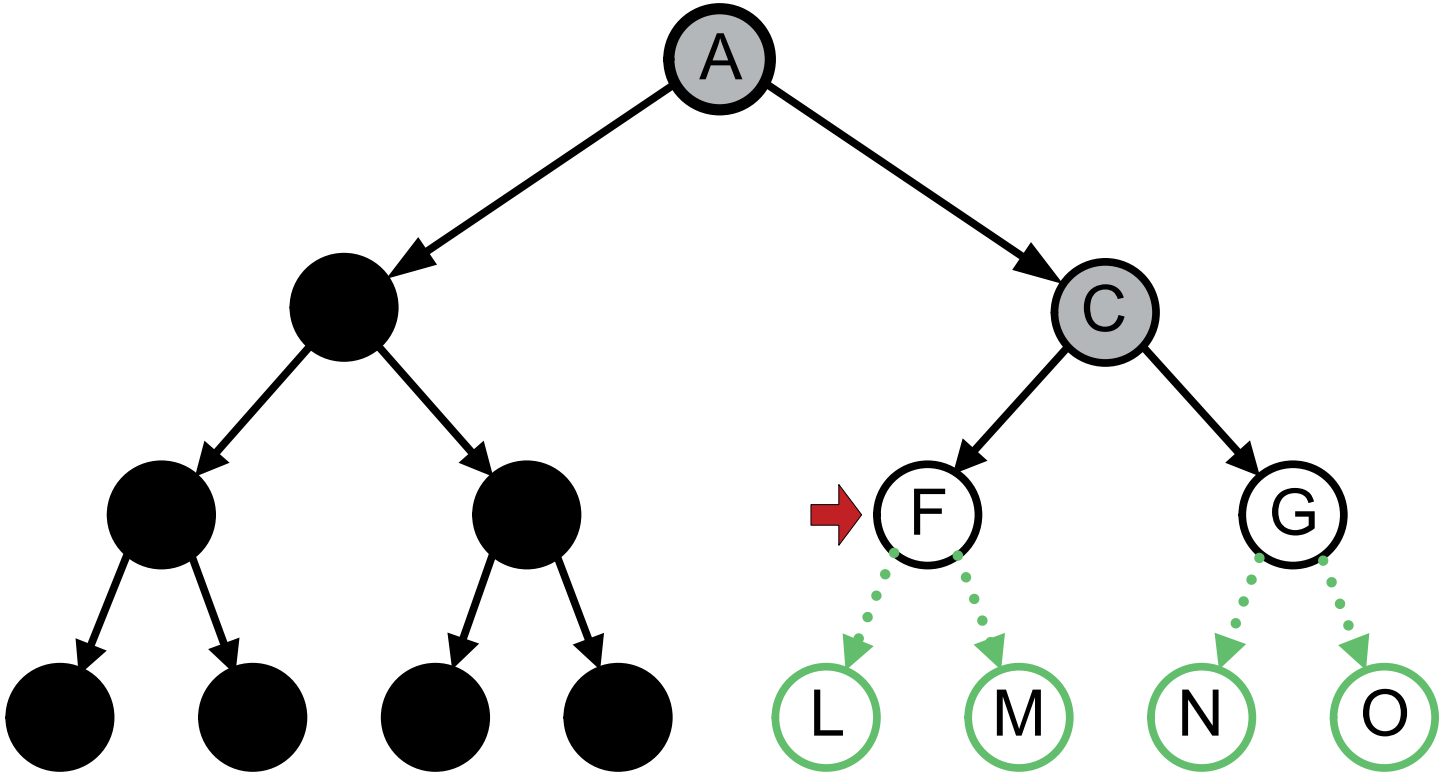
Ricerca in profondità (9)



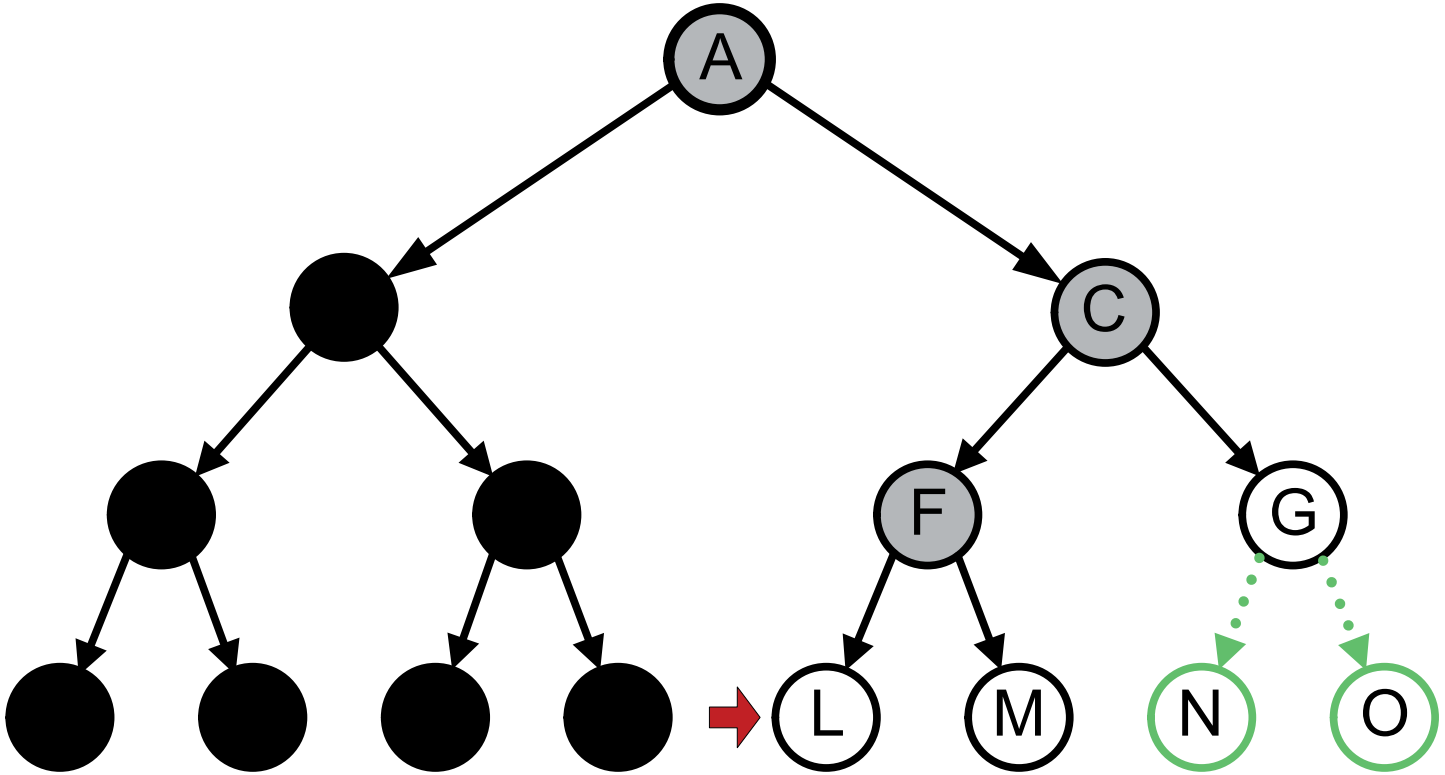
Ricerca in profondità (10)



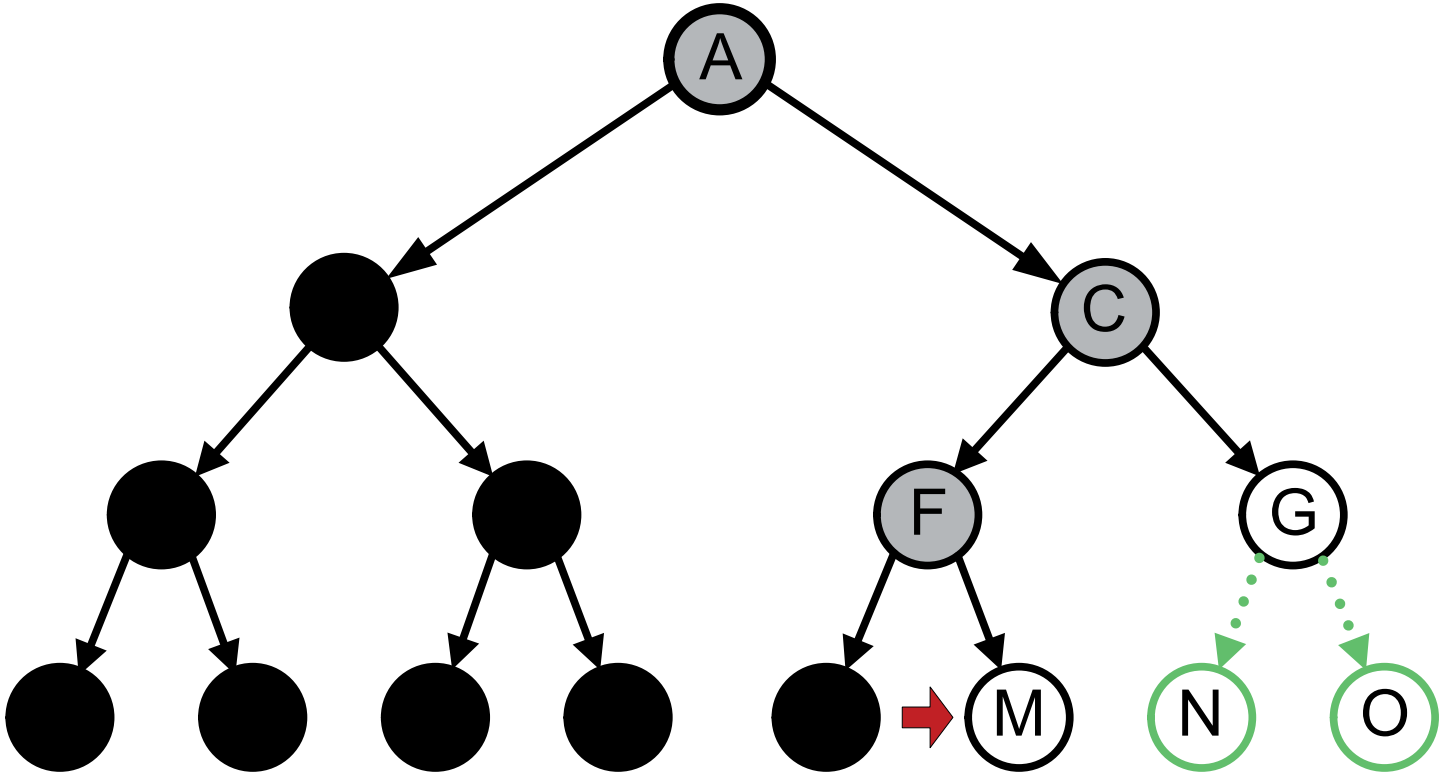
Ricerca in profondità (11)



Ricerca in profondità (12)



Ricerca in profondità (13)



Ricerca in profondità (13)

Funzione di inserimento nella coda: ENQUEUE-AT-FRONT

```
function DEPTH-FIRST-SEARCH(problem) returns a solution or
                                                    failure
fringe ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem])
loop do
    If EMPTY?(fringe) then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    fringe ← ENQUEUE-AT-FRONT(fringe, EXPAND(node,OPERATORS(problem)))
end
```

Ricerca in profondità (14)

- Si può implementare mediante una funzione ricorsiva.
- In tal caso la lista di nodi da visitare (pila) è conservata implicitamente nello stack dei record di attivazione.
- Quando si espande un nodo non obiettivo e senza figli, sia fa **backtracking**, cioè si torna indietro fino all'ultimo nodo in cui è possibile effettuare una scelta.

Ricerca in profondità (15)

- Non è né completa né ottimale
- Complessità in tempo: $O(b^m)$

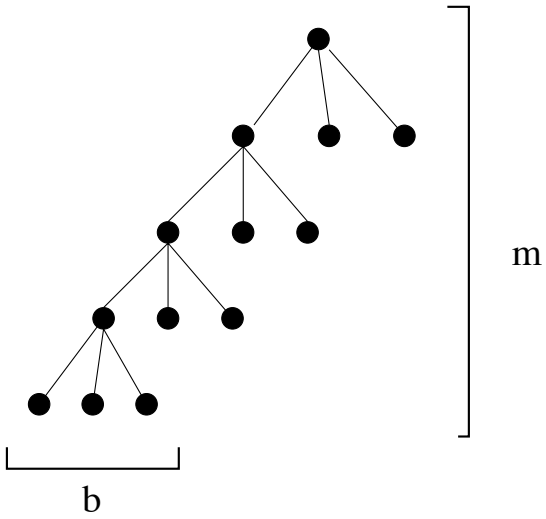
  b = fattore di ramificazione dell'albero
  m= profondità MASSIMA dell'albero di ricerca

Ricerca in profondità (16)

- Complessità in spazio: $O(b \cdot m)$

  Si deve memorizzare solo un cammino radice-foglia e i fratelli non espansi di ciascun nodo del cammino.

  Il massimo numero di nodi da conservare in memoria è b · m:



  Se b = 10, 100 bytes/nodo, m = 12: 12 Kb di memoria

N.B.: Se l'albero ha rami infiniti, la ricerca può non terminare.

Ricerca in profondità limitata (1)

- Opera come la ricerca in profondità imponendo, però, un limite alla profondità massima dei cammini: un nodo viene espanso solo se la lunghezza del cammino corrispondente è minore del massimo stabilito.

- Se non viene trovata alcuna soluzione l'algoritmo restituisce il valore speciale taglio se alcuni nodi non sono stati espansi per il limite della profondità, altrimenti fallimento.

N.B.: Si possono utilizzare conoscenze specifiche del problema per fissare il limite di profondità.

Ricerca in profondità limitata (2)

È completa se il problema è tale che, se ha soluzione, allora esiste una soluzione di lunghezza minore o uguale al limite massimo.

Non è ottimale

Complessità in tempo : $O(b^l)$, con

b = fattore di ramificazione dell'albero

l = limite di profondità fissato

Complessità in spazio : $O(b \cdot l)$

La ricerca in profondità limitata può risolvere il problema della completezza, ma resta non ottimale.

Ricerca per approfondimenti successivi (1) (Iterative-deepening search)

Non sempre si conosce un limite adeguato per la ricerca in profondità limitata.
Questa strategia evita il problema della scelta di un limite adeguato provando iterativamente tutti i limiti possibili.

```
function ITERATIVE-DEEPENING—SEARCH(problem) returns solution
                                                    or failure
for depth = 0 to ∞ do
     If DEPTH-LIMITED-SEARCH(problem, depth) succeeds
     then return its result
end
```

Combina i benefici della ricerca in profondità con quelli della ricerca in ampiezza

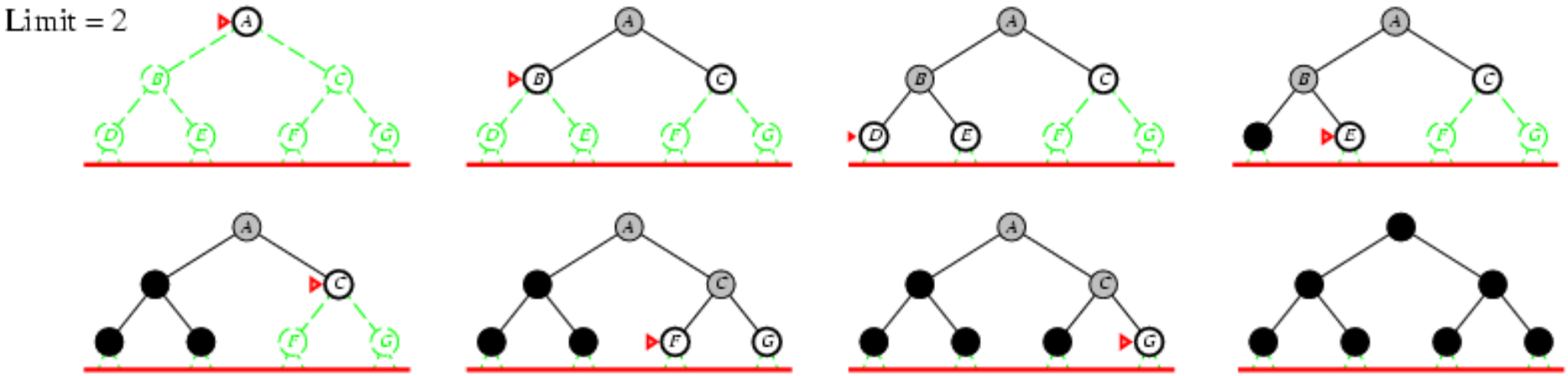
Iterative-deepening search (2)

Limit = 0

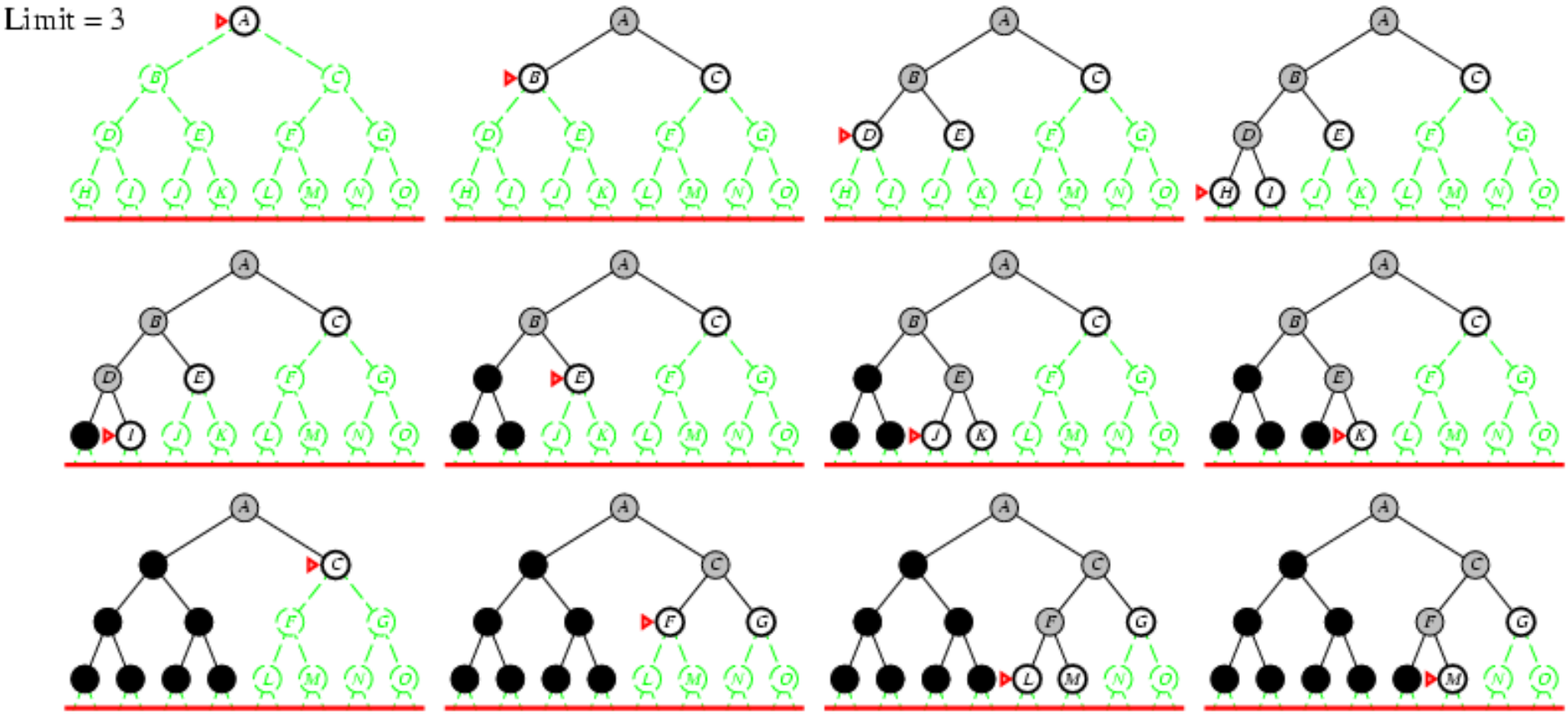
Iterative-deepening search (3)

Limit = 1

Iterative-deepening search (4)



Iterative-deepening search (5)



Iterative-deepening search (6)

È completa (sotto le stesse ipotesi della ricerca in ampiezza)

È ottimale (sotto le stesse ipotesi della ricerca in ampiezza)

Complessità in spazio : $O(b \cdot d)$, dove

b = fattore di ramificazione dell'albero

d = profondità minima di una soluzione

Iterative-deepening search (7)

Complessità in tempo : $O(b^d)$

- i nodi del livello 1 (b) sono generati d volte
- i nodi del livello 2 (b^2) sono generati $d-1$ volte

  .....
- i nodi del livello d (b^d) sono generati 1 volta

Numero di nodi generati:

$$b \cdot d + b^2 \cdot (d-1) + \ldots + b^{d-1} \cdot 2 + b^d \cdot 1$$

Se $b = 10$ e $d = 5$: ≈ 123.000, con $b^d \approx 111.000$

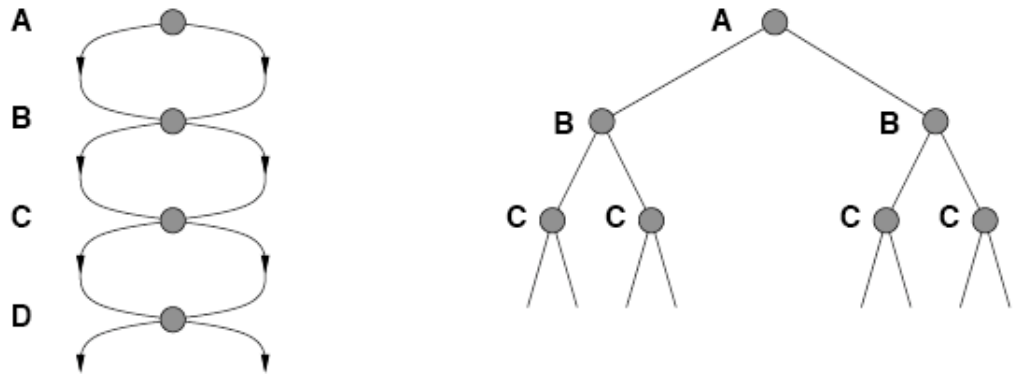
11% in più rispetto alla ricerca in ampiezza

Evitare Ripetizioni negli Stati

- Una delle complicazioni più gravi del processo di ricerca è quella di perdere tempo di elaborazione espandendo stati che sono già stati incontrati.
- Ci sono casi in cui la ripetizione degli stati è inevitabile; ad esempio tutti i problemi in cui le azioni sono reversibili, come le ricerche di itinerari o i rompicapi a tasselli mobili: in questi casi gli alberi di ricerca sono infiniti.

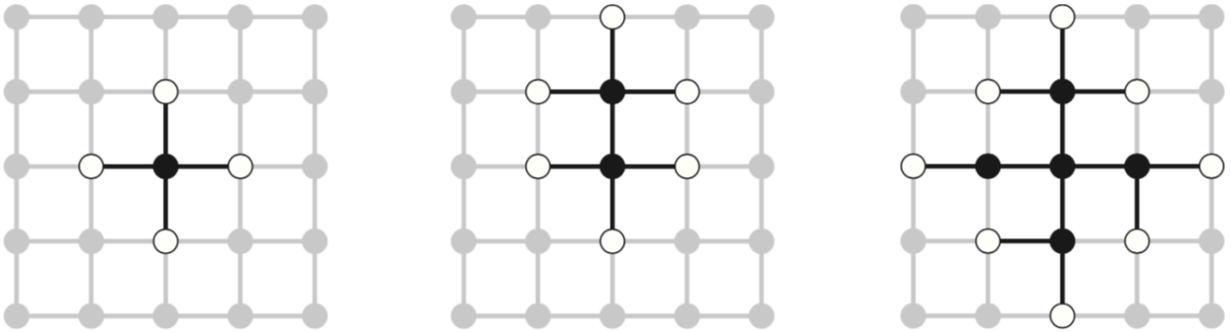
Evitare Ripetizioni negli Stati

- La ricerca con stati ripetuti può trasformare un problema lineare in un problema esponenziale!
- Uno spazio degli stati di dimensione $d + 1$ diventa un albero con 2^d foglie (vedi figura).



Evitare Ripetizioni negli Stati

- Un esempio più realistico è la **griglia rettangolare** illustrata in figura.
- Ogni stato ha quattro successori.
- L'albero di ricerca ha 4^d foglie se includiamo gli stati ripetuti, ma ci sono solo circa $2d^2$ stati distinti entro d passi da ogni stato.
- Per $d = 20$: circa mille miliardi di nodi, ma solo 800 stati distinti.



(a) (b) (c)

Evitare Ripetizioni negli Stati

- Gli stati ripetuti non rilevati dall'algoritmo possono in buona sostanza far diventare irrisolvibile un problema che non lo è.
- Può essere dunque conveniente controllare se uno stato è replicato più volte nell'albero di ricerca.
- Rilevare le ripetizioni in genere significa confrontare i nodi che si stanno generando con quelli già espansi o generati.
- Se si ha una corrispondenza, l'algoritmo ha scoperto due diversi cammini che portano allo stesso stato e può scartarne uno.

Evitare Ripetizioni negli Stati

Gli algoritmi che dimenticano la loro storia

sono condannati a ripeterla

- L'unico modo per evitare il rischio di esplorare gli stati già visitati è quello di ricordare dove si è passati, tenendo quindi più nodi in memoria (compromesso tra spazio e tempo).
- Possiamo modificare l'algoritmo **TREE-SEARCH** aggiungendo una struttura dati **insieme esplorato** o **lista chiusa** (*closed*), che memorizza ogni nodo espanso.
- I nuovi nodi che corrispondono a nodi generati precedentemente (quelli nell'insieme esplorato) possono essere scartati.

Algoritmo Generale di Ricerca
Graph-Search

```
function TREE-SEARCH(problem) returns a solution or failure

fringe ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem])
loop do
     If EMPTY?(fringe) then return failure
     node ← REMOVE-FRONT(fringe)
     if GOAL-TEST(problem, STATE[node]) then return SOLUTION(node)
     fringe ← QUEUING-FN(fringe, EXPAND(node, OPERATORS(problem)))
end
```

Dobbiamo apportare all'algoritmo TREE-SEARCH le seguenti modifiche:

- Aggiunta di una struttura dati ***listachiusa*** (close)
- Modifica della parte evidenziata in figura, in modo da inserire in fringe solo i nodi non incontrati in precedenza.

Algoritmo Generale di Ricerca
Graph-Search

```
function GRAPH-SEARCH(problem) returns solution or failure

close  ← un insieme vuoto
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]))
loop do
     If EMPTY?(fringe) then return failure
     node ← REMOVE-FRONT(fringe)
     if GOAL-TEST?(problem, STATE[node]) then return SOLUTION(node)
     aggiungi STATE[node] in close
     espandi node e aggiungi i nodi generati in fringe
       solo se i loro stati non sono già in close
end
```

Questo algoritmo presume che il primo cammino che raggiunge uno stato **s** sia sempre il più conveniente.

Algoritmo Generale di Ricerca
Graph-Search

```
function GRAPH-SEARCH(problem) returns solution or failure

close  ← un insieme vuoto
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]))
loop do
     If EMPTY?(fringe) then return failure
     node ← REMOVE-FRONT(fringe)
     if GOAL-TEST?(problem, STATE[node]) then return SOLUTION(node)
     aggiungi STATE[node] in close
     child_list ← EXPAND(node, OPERATORS(problem))
     for child_node in child_list do
          if STATE[child_node] NOT in close then
          fringe ← QUEUING-FN(fringe, child_node)
end
```

Questo algoritmo presume che il primo cammino che raggiunge uno stato **s** sia sempre il più conveniente.

Algoritmo Generale di Ricerca
Graph-Search

- L'albero di ricerca costruito dall'algoritmo **GRAPH-SEARCH** contiene al più una sola copia di ciascuno stato.
- Possiamo dunque pensare che faccia crescere un albero direttamente sul grafo dello spazio degli stati:

Algoritmo Generale di Ricerca
Graph-Search

- Si noti che la frontiera separa il grafo dello spazio degli stati nella regione esplorata e in quella inesplorata, in modo che ogni cammino che vada dallo stato iniziale a uno inesplorato debba passare attraverso uno stato che sta nella frontiera.
- L'algoritmo esamina sistematicamente gli stati dello spazio degli stati, uno a uno, finché trova una soluzione.

Algoritmo Generale di Ricerca
Graph-Search

- L'ottimalità è un aspetto spinoso della ricerca su grafo. Quando si rileva uno stato ripetuto significa che l'algoritmo ha scoperto un altro cammino che porta allo stesso stato.
- L'algoritmo scarta sempre il cammino *appena scoperto*. Se quest'ultimo è più breve di quello originale, l'algoritmo potrebbe perdere una soluzione ottima.
- L'uso della lista chiusa significa che la ricerca in profondità e quella ad approfondimento iterativo non hanno più i requisiti spaziali lineari.

Sintesi degli Argomenti Trattati nella Lezione

- La **ricerca in ampiezza** (**BREADTH-FIRST-SEARCH**) espande prima il nodo più superficiale nell'albero di ricerca. E' completa, ottimale per operatori con lo stesso costo e ha una complessità temporale e spaziale $O(b^d)$. Ciò la rende impraticabile nella maggior parte dei casi.

- La **ricerca a costo uniforme** (**UNIFORM-COST-SEARCH**) espande prima il nodo foglia con costo minore. E' completa e, a differenza della ricerca in ampiezza, è ottimale anche quando gli operatori hanno costi differenti. Il costo di ogni passo deve però essere maggiore o uguale a una costante positiva ε.

- La **ricerca in profondità** (**DEPTH-FIRST-SEARCH**) espande prima il nodo più profondo dell'albero di ricerca. Non è né completa né ottimale e ha una complessità temporale $O(b^m)$ e una complessità spaziale $O(bm)$, dove m è la profondità massima dei cammini nello spazio di stato. Negli alberi di ricerca con profondità elevata o infinita la complessità temporale la rende impraticabile.

Sintesi degli Argomenti Trattati nella Lezione

- La **ricerca a profondità limitata** (**DEPTH-LIMITED-SEARCH**) espande prima il nodo più profondo dell'albero di ricerca. Non è né completa né ottimale. Ha una complessità temporale $O(b^m)$ e una complessità spaziale $O(bm)$, dove m è la profondità massima.

- La **ricerca ad approfondimento iterativo** (**ITERATIVE-DEEPENING-SEARCH**) richiama la ricerca a profondità limitata con limiti crescenti fino a trovare un obiettivo. E' completa e ottimale e ha complessità temporale $O(b^d)$ e complessità spaziale $O(bd)$.

- Quando lo spazio degli stati è un grafo invece di un albero, può essere conveniente controllare se uno stato è replicato più volte nell'albero di ricerca. L'algoritmo **GRAPH-SEARCH** elimina tutti gli stati ripetuti.

Riferimenti

Russell, S., Norvig, P. *Artificial Intelligence - a Modern Approach*, fourth Edition, Pearson Education, 2021.

Nilsson, N.J. *Artificial Intelligence - a New Synthesis*, Morgan Kaufman, 1998.