Ricerca in uno spazio di soluzioni

Outline

- ♢ Formulazione del problema
- ♢ Esempio di problema
- ♢ Alcuni algoritmi di ricerca

Esempio: Romania

In vacanza in Romania; ora ad Arad.
Il volo parte domani da Bucharest

Formulare il goal:

essere a Bucharest

Formulare il problema:

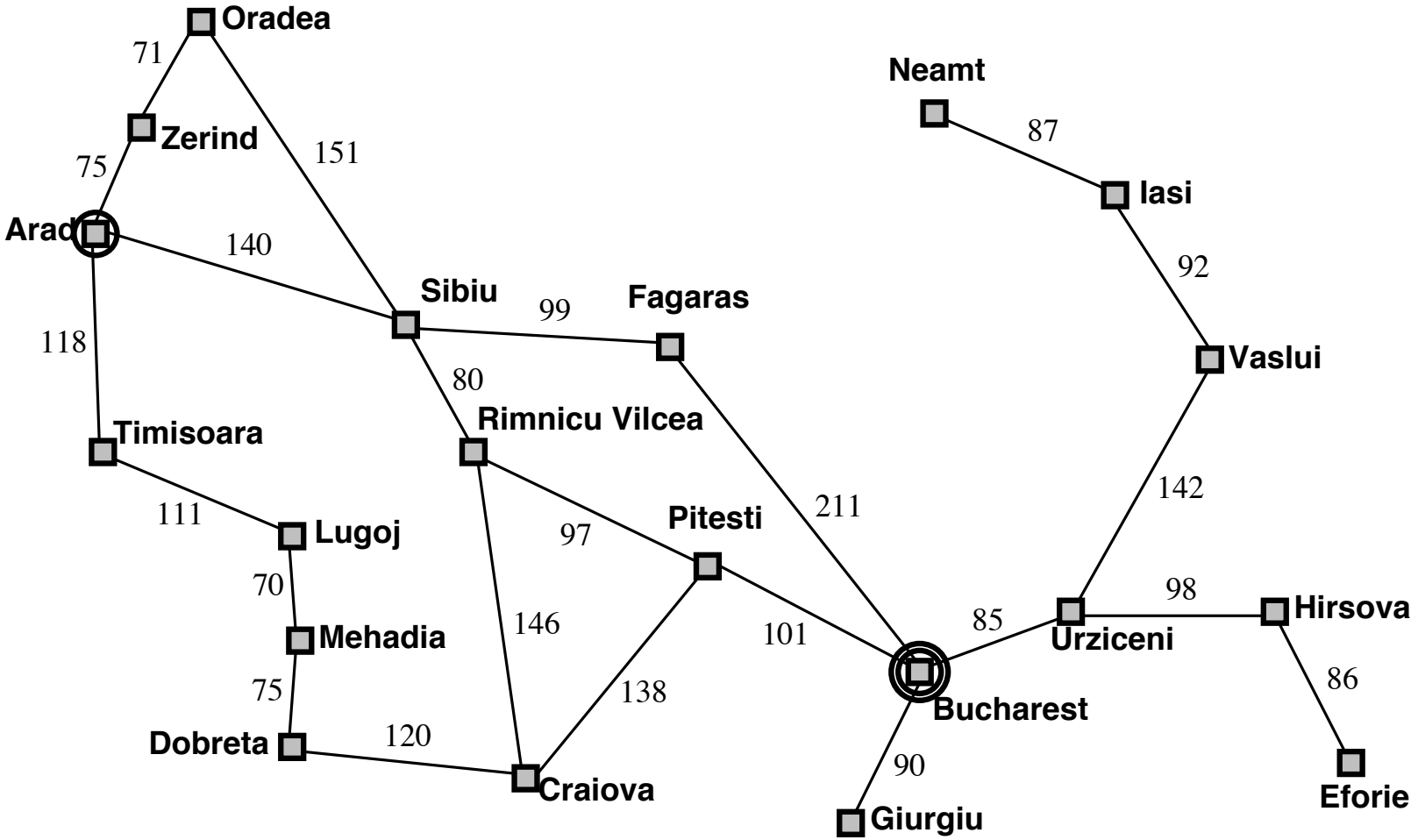
stati: varie città

azioni: viaggi fra città

Trovare una soluzione:

sequenza di città, esempio: Arad, Sibiu, Fagaras, Bucharest

Esempio: Romania



Formulazione del problema ("Single-state")

Un *problema* è definito da quattro elementi:

stato iniziale es., "ad Arad"

funzione successore $S(x) =$ insieme di coppie azione–stato
es., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \ldots\}$

test per il goal, può essere
esplicito, es., $x =$ "a Bucharest"
implicito, es., $Bucharest(x)$

costo di un cammino (additivo)
es., somma di distanze, numero di azioni eseguite, etc.
$c(x, a, y)$ è il costo del singolo passo, assunto essere ≥ 0

Una *soluzione* è una sequenza di azioni
che conduce dallo stato iniziale ad uno stato di goal

Selezionare uno spazio degli stati

Il mondo reale è molto complesso

$\Rightarrow$ lo spazio degli stati deve essere *astratto* per risolvere il problema

Stato (astratto) = insieme di stati reali

Azione (astratta) = combinazione complessa di azioni reali

es., "Arad $\rightarrow$ Zerind" rappresenta un insieme complesso

di possibili strade, detour, fermate, etc.

Per garantire la realizzabilità, qualsiasi stato reale "in Arad"

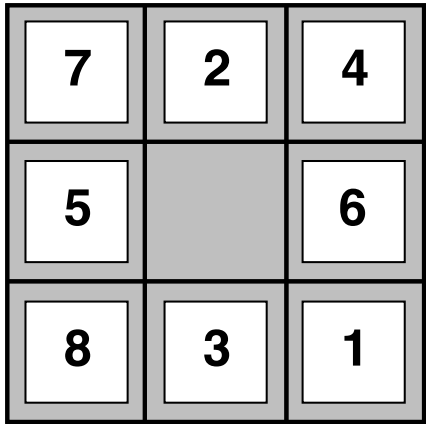
deve condurre a *qualche* stato reale "in Zerind"

Soluzione (astratta) =

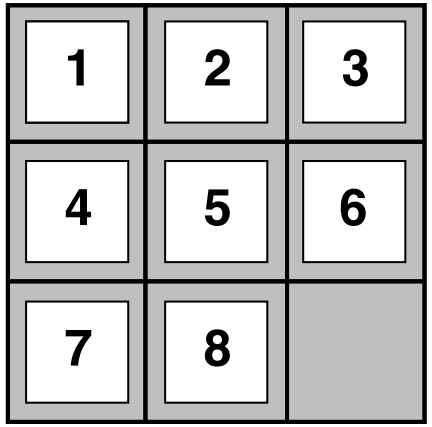
insieme di cammini reali che sono soluzioni nel mondo reale

Ogni azione astratta dovrebbe essere "più semplice" del problema originale!

Esempio: il puzzle



Start State

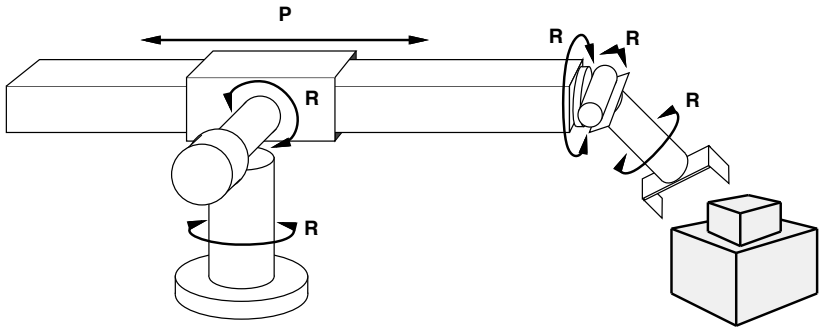


Goal State

stati??: locazioni intere dei tasselli (ignora posizioni intermedie)
azioni??: mossa dello spazio a sinistra, destra, alto, basso
test per il goal??: $=$ stato di goal (dato)
costo di un cammino??: 1 per mossa

[Nota: la soluzione ottima di n-Puzzle è NP-hard]

Esempio: assemblaggio robotico



stati??: coordinate reali
degli snodi del robot e delle parti dell'oggetto da assemblare

azioni??: mosse continue degli snodi del robot

test per il goal??: assemblaggio completo

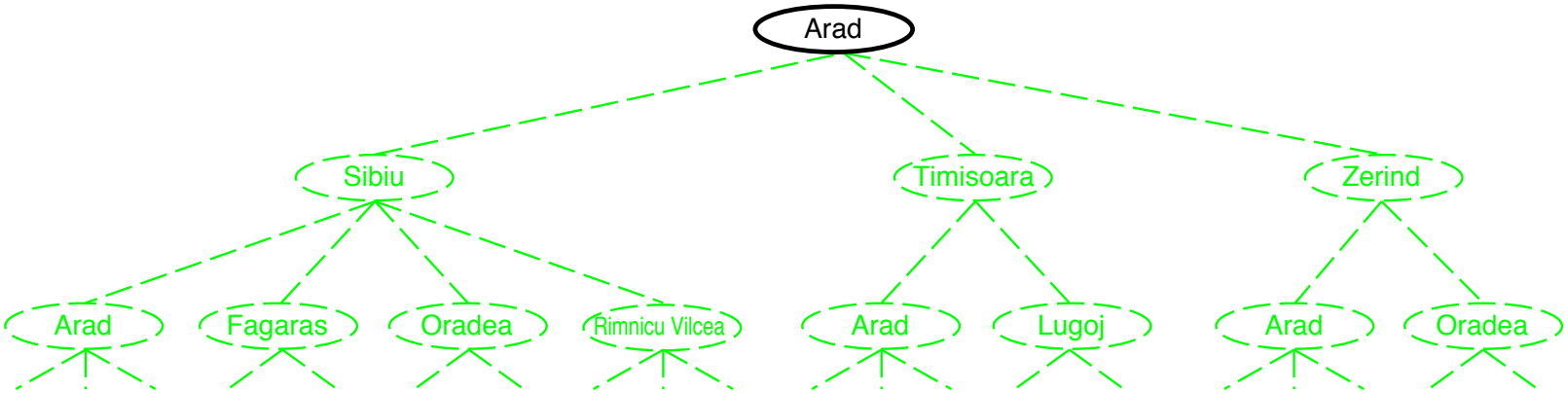
costo di un cammino??: tempo per eseguire il tutto

Algoritmi di ricerca (ad albero)

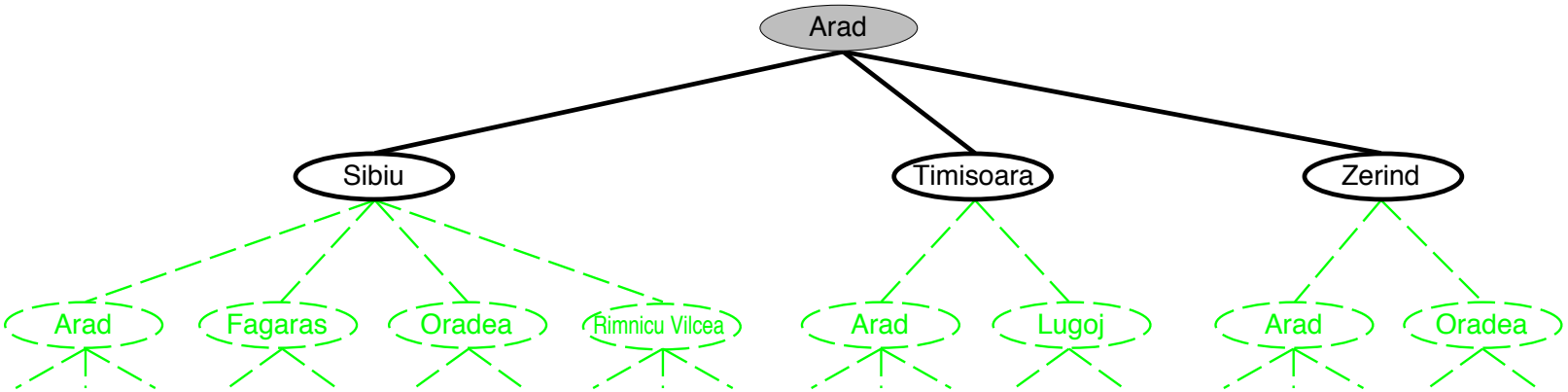
Idea di base: esplorazione simulata dello spazio degli stati generando successori di stati già esplorati (cioè espandendo stati)

```
function RICERCA-ALBERO(problema, strategia) returns una soluzione, o il fallimento
    inizializza l'albero di ricerca usando lo stato iniziale di problema
    loop do
        if non ci sono più candidati per l'espansione then return fallimento
        scegli un nodo foglia per l'espansione in base alla strategia
        if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
        else espandi il nodo e aggiungi i nodi risultanti all'albero di ricerca
```

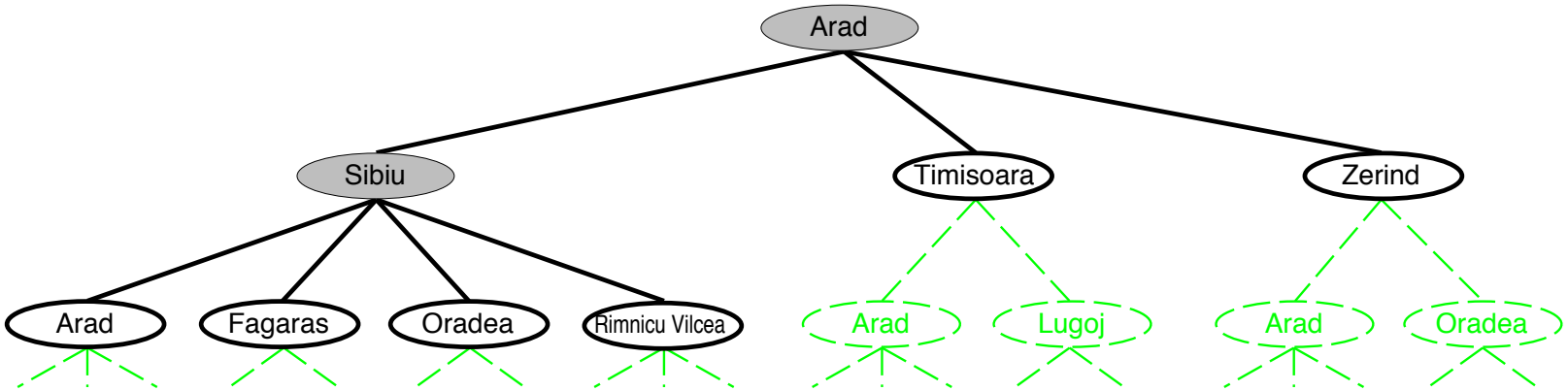
Esempio di ricerca



Esempio di ricerca

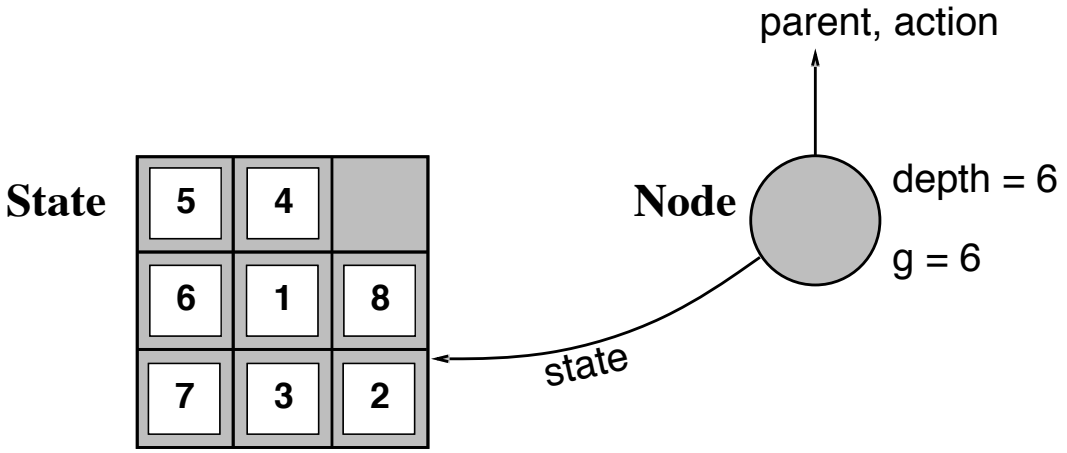


Esempio di ricerca



Implementazione: stati vs. nodi

Uno *stato* è una rappresentazione di una configurazione fisica
Un *nodo* è una struttura dati che fa parte di un albero di ricerca
include *genitori*, *figli*, *profondità*, *costo del cammino* $g(x)$
Stati non hanno genitori, figli,...



La funzione EXPAND crea nuovi nodi, riempiendo i vari campi ed usando la funzione SUCCESSORFN del problema per creare gli stati corrispondenti.

Implementazione degli algoritmi di ricerca

```
function RICERCA-ALBERO(problema, frontiera) returns una soluzione, o il fallimento

    frontiera ← INSERISCI(CREA-NODO(STATO-INIZIALE[problema]), frontiera)
    loop do
        if VUOTA?(frontiera) then return fallimento
        nodo ← RIMUOVI-PRIMO(frontiera)
        if TEST-OBIETTIVO[problema] applicato a STATO[nodo] ha successo
            then return SOLUZIONE(nodo)
        frontiera ← INSERISCI-TUTTI(ESPANDI(nodo, problema), frontiera)


function ESPANDI(nodo, problema) returns un insieme di nodi

    successori ← l'insieme vuoto
    for each ⟨azione, risultato⟩ in FUNZIONE-SUCCESSORE[problema](STATO[nodo]) do
        s ← un nuovo NODO
        STATO[s] ← risultato
        NODO-PADRE[s] ← nodo
        AZIONE[s] ← azione
        COSTO-DI-CAMMINO[s] ← COSTO-DI-CAMMINO[nodo] +
                                  COSTO-DI-PASSO(nodo, azione, s)
        PROFONDITÀ[s] ← PROFONDITÀ[nodo] + 1
        aggiungi s a successori
    return successori
```

Strategie di ricerca

Una strategia è definita scegliendo un *ordine di espansione dei nodi*

Le strategie sono valutate secondo le seguenti dimensioni:

- completezza—trova sempre una soluzione se ne esiste una?
- complessità in tempo—numero di nodi generati/espansi
- complessità in spazio—numero massimo di nodi in memoria
- ottimalità—trova sempre una soluzione di costo minimo ?

A complessità in spazio e tempo sono misurate in termini di

- b—massimo fattore di branching dell'albero di ricerca
- d—profondità della soluzione di costo minimo
- m—massima profondità dello spazio degli stati (può essere ∞)

Strategie di ricerca non informate

Le strategie *non informate* usano solo l'informazione disponibile nella definizione del problema

Ricerca Breadth-first

Ricerca a costo uniforme

Ricerca Depth-first

Ricerca Depth-limited

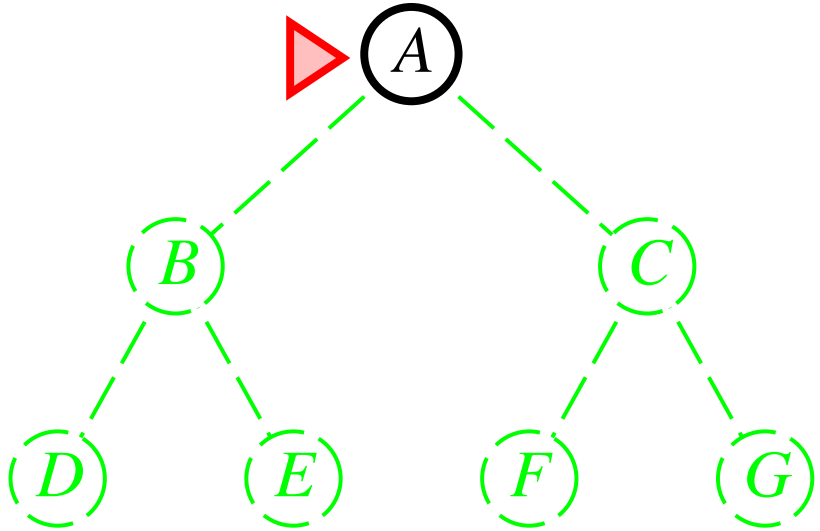
Ricerca Iterative deepening

Ricerca Breadth-first

Espande il nodo a profondità più bassa

Implementazione:

frontiera è una coda FIFO, cioè i successori sono inseriti in fondo

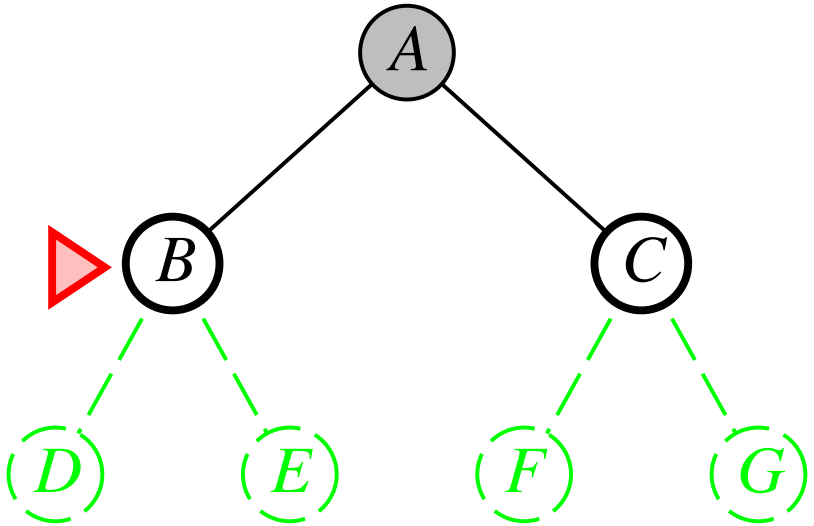


Ricerca Breadth-first

Espande il nodo a profondità più bassa

Implementazione:

frontiera è una coda FIFO, cioè i successori sono inseriti in fondo

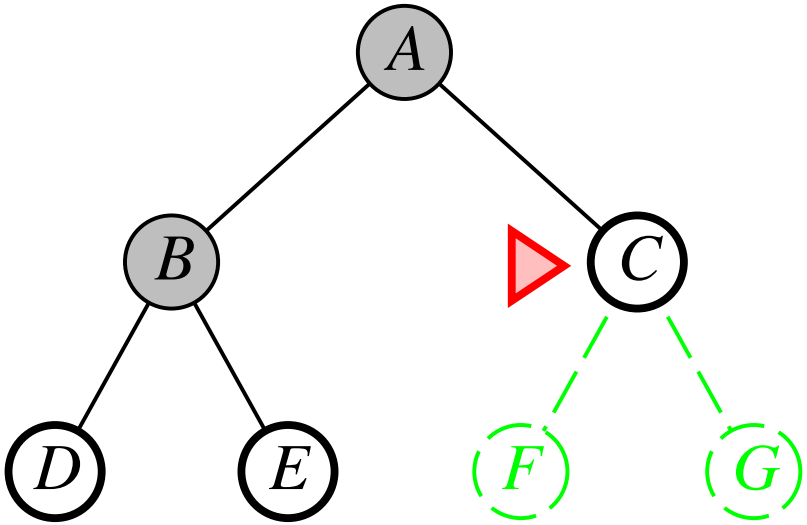


Ricerca Breadth-first

Espande il nodo a profondità più bassa

Implementazione:

frontiera è una coda FIFO, cioè i successori sono inseriti in fondo

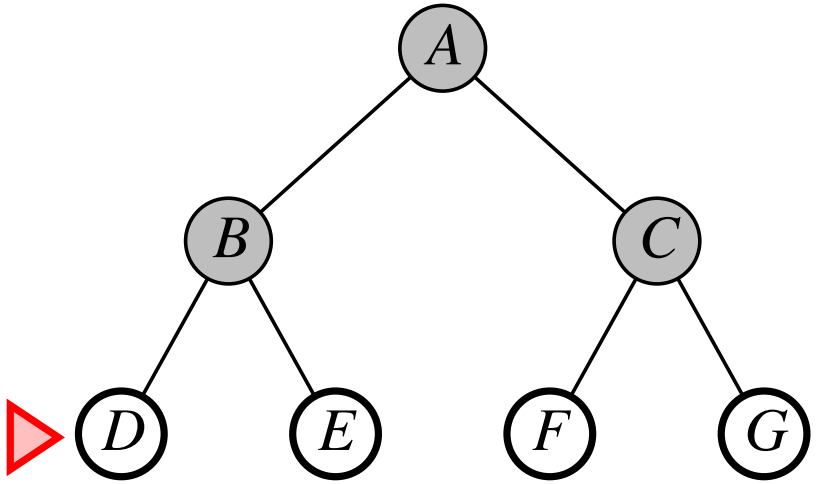


Ricerca Breadth-first

Espande il nodo a profondità più bassa

Implementazione:

frontiera è una coda FIFO, cioè i successori sono inseriti in fondo



Proprietà della ricerca breadth-first

Completa?? Si (se b è finito)

Tempo?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, cioè esponenziale in d

Spazio?? $O(b^{d+1})$ (mantiene ogni nodo in memoria)

Ottima?? Si (se costo $= 1$ per passo); non ottima in generale

Lo spazio è il vero problema; può facilmente generare nodi per una occupazione di 10MB/sec.

Ricerca a costo uniforme

Espande il nodo a costo (di cammino) inferiore

Implementazione:

$frontiera =$ coda (a priorità) ordinata per costo di cammino

Equivalente alla ricerca breadth-first se i costi dei singoli passi è identico

Completa?? Si, se il singolo passo costa $\geq \epsilon$

Tempo?? $\#$ di nodi con $g \leq$ costo della soluzione ottima, $O(b^{\lceil C^*/\epsilon \rceil})$
dove C^* è il costo della soluzione ottima

Spazio?? $\#$ di nodi con $g \leq$ costo della soluzione ottima, $O(b^{\lceil C^*/\epsilon \rceil})$

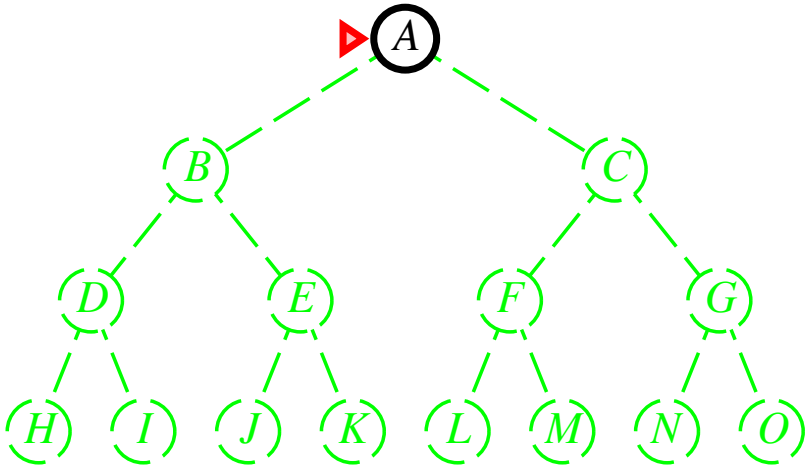
Ottima?? Si—nodi espansi in ordine crescente di $g(n)$

Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

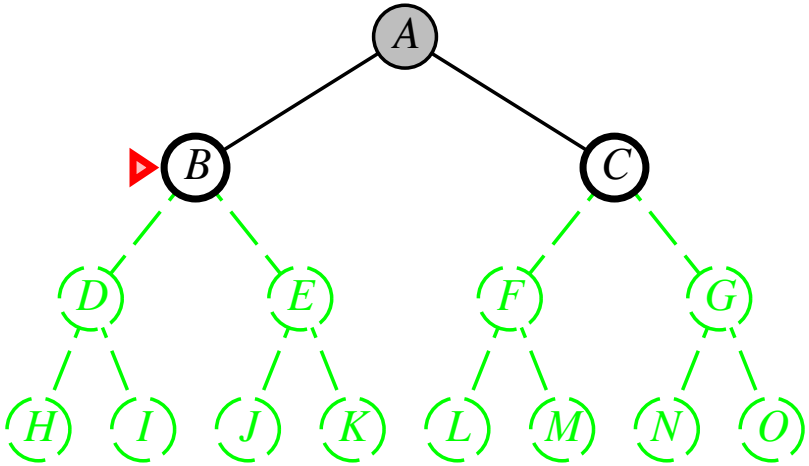


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

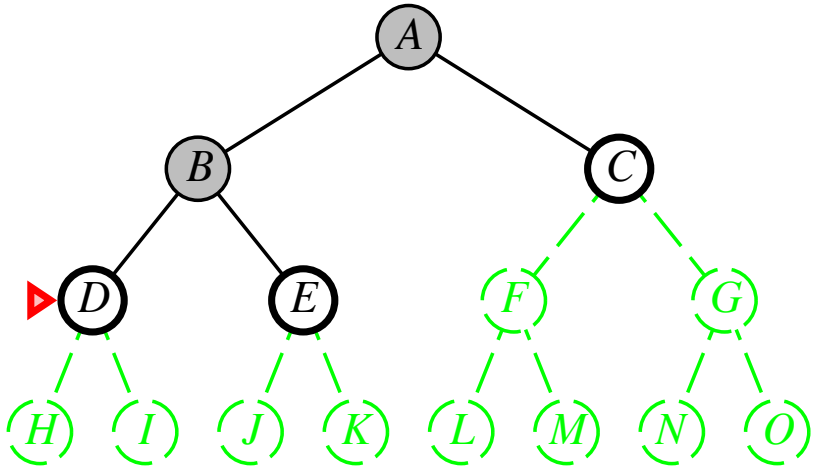


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

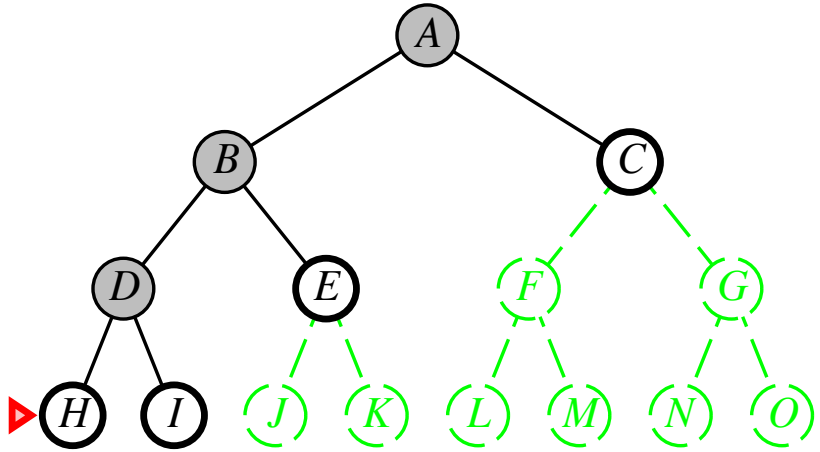


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

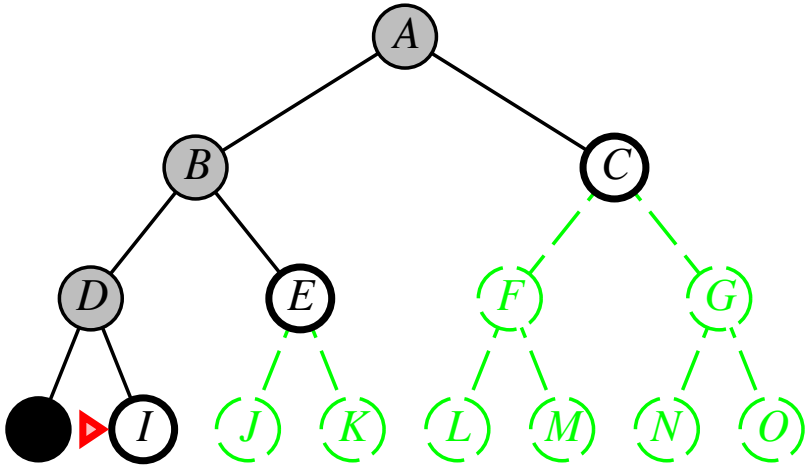


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

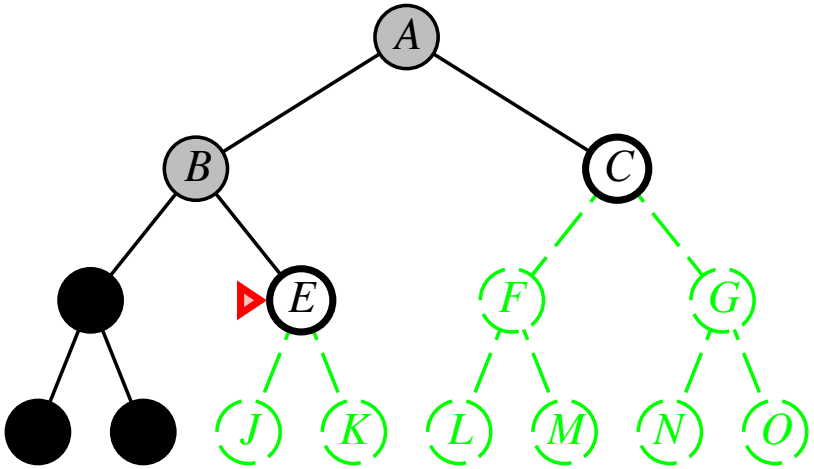


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

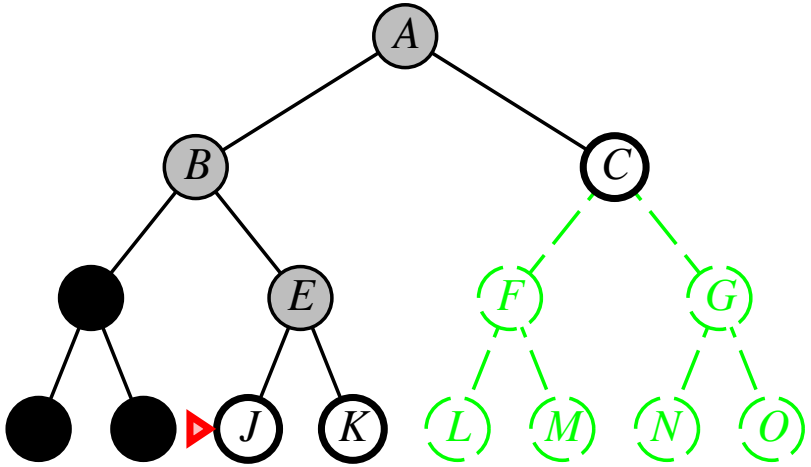


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

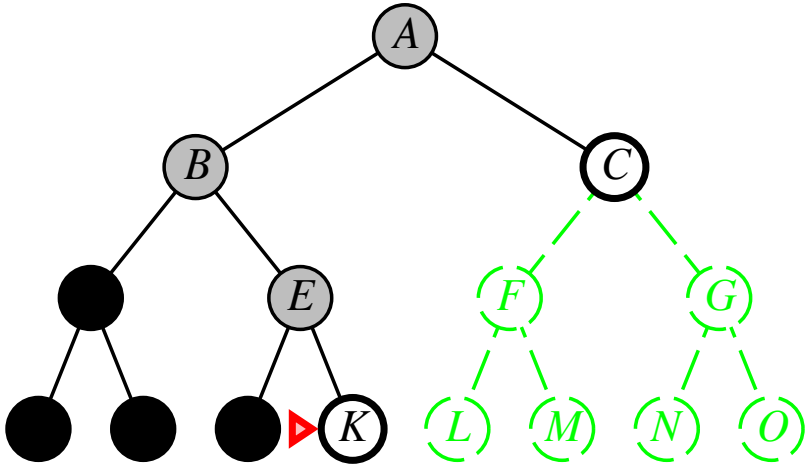


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

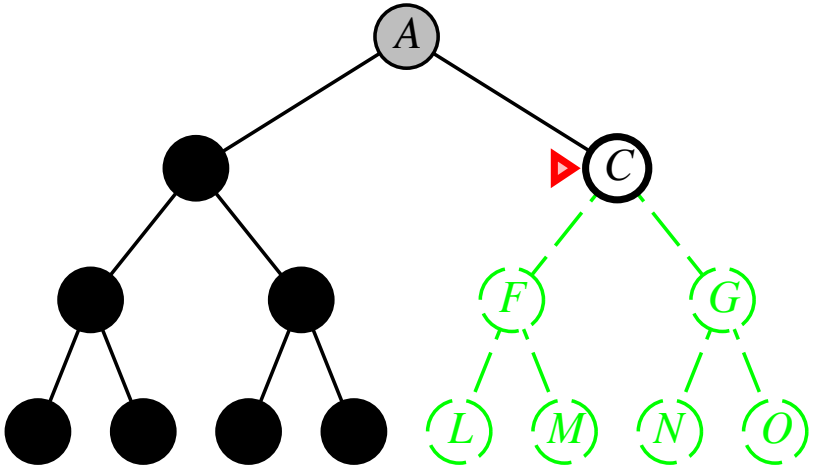


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

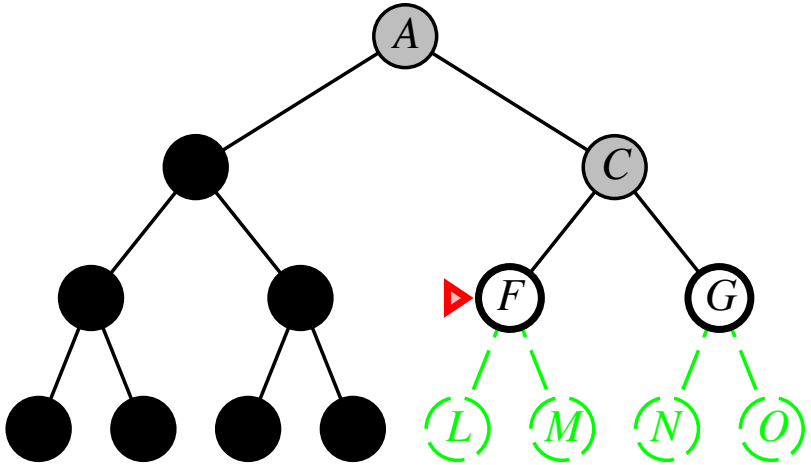


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

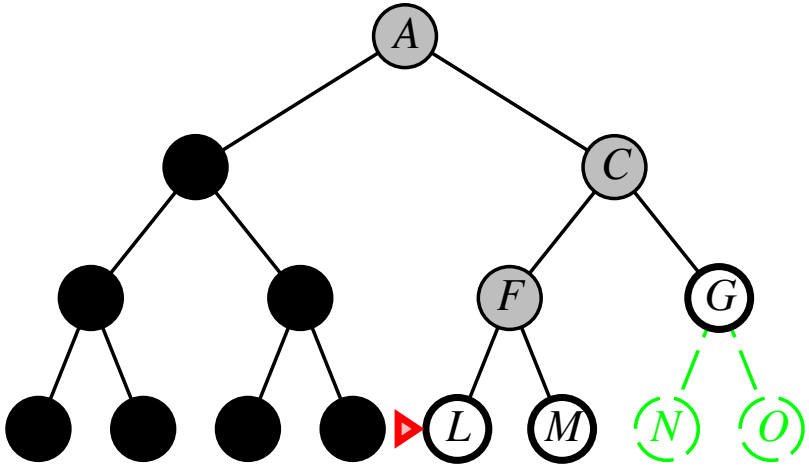


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa

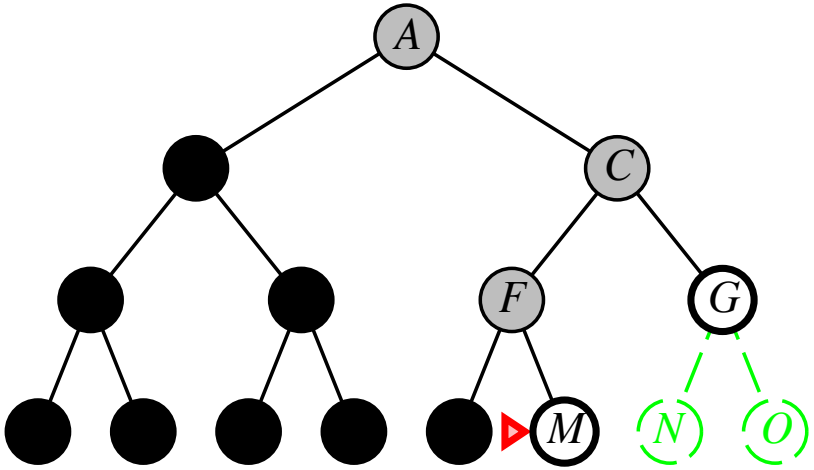


Ricerca Depth-first

Espande il nodo a profondità massima

Implementazione:

$frontiera =$ coda LIFO (pila), cioè inserisce i successori in testa



Proprietà della ricerca depth-first

Completa?? No: fallisce se spazio stati a progfondità infinita o con cicli

Modificato per evitare stati ripetuti lungo un cammino

$\Rightarrow$ completa in spazio degli stati finito

Tempo?? $O(b^m)$: terribile se m è più grande di d

ma se le soluzioni sono dense, può essere più veloce della ricerca breadth-first

Spazio?? $O(bm)$, cioè spazio lineare!

Ottima?? No

Ricerca Depth-limited: algoritmo

= ricerca depth-first con limite di profondità l,
cioè per i nodi a profondità l non si generano i successori

```
function RICERCA-PROFONDITÀ-LIMITATA(problema, limite) returns una soluzione, o il
                                                              fallimento/taglio
    return RPL-RICORSIVA(CREA-NODO(STATO-INIZIALE[problema]), problema, limite)

function RPL-RICORSIVA(nodo, problema, limite) returns una soluzione, o il fallimento/taglio
    avvenuto_taglio? ← false
    if TEST-OBIETTIVO[problema](STATO[nodo]) then return SOLUZIONE(nodo)
    else if PROFONDITÀ[nodo] = limite then return taglio
    else for each successore in ESPANDI(nodo, problema) do
        risultato ← RPL-RICORSIVA(successore, problema, limite)
        if result = taglio then avvenuto_taglio? ← true
        else if risultato ≠ fallimento then return risultato
    if avvenuto_taglio? then return taglio else return fallimento
```

Ricerca Iterative Deepening

Idea di base: Prova tutti i possibili limiti di profondità

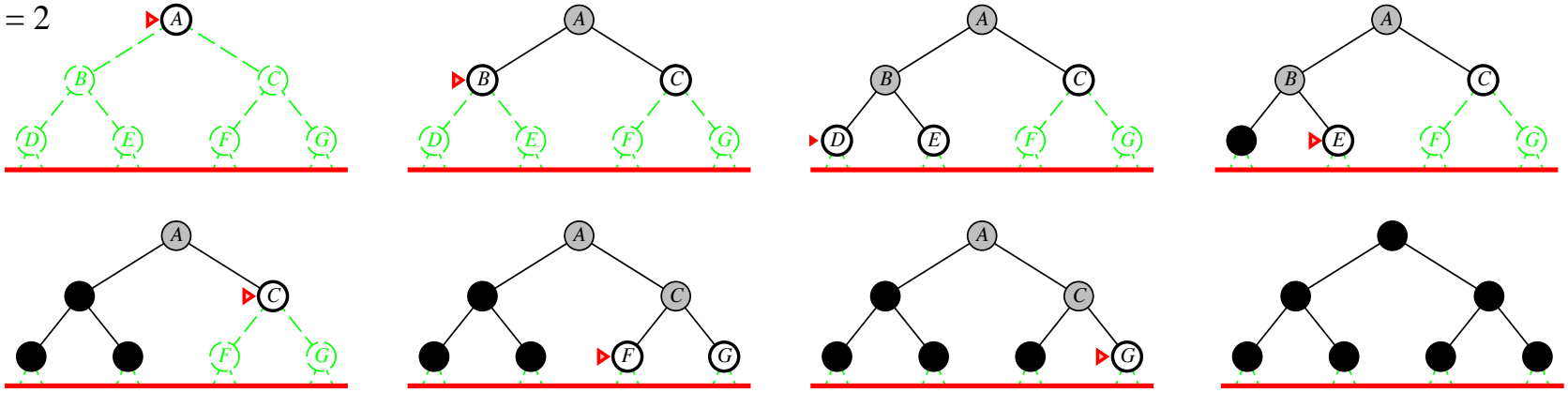
```
function RICERCA-APPROFONDIMENTO-ITERATIVO(problema) returns una soluzione, o il fallimento
    inputs: problema, un problema

    for profondità ← 0 to ∞ do
        risultato ← RICERCA-PROFONDITÀ-LIMITATA(problema, profondità)
        if risultato ≠ taglio then return risultato
```

Ricerca Iterative Deepening $l = 1$

Limit = 1

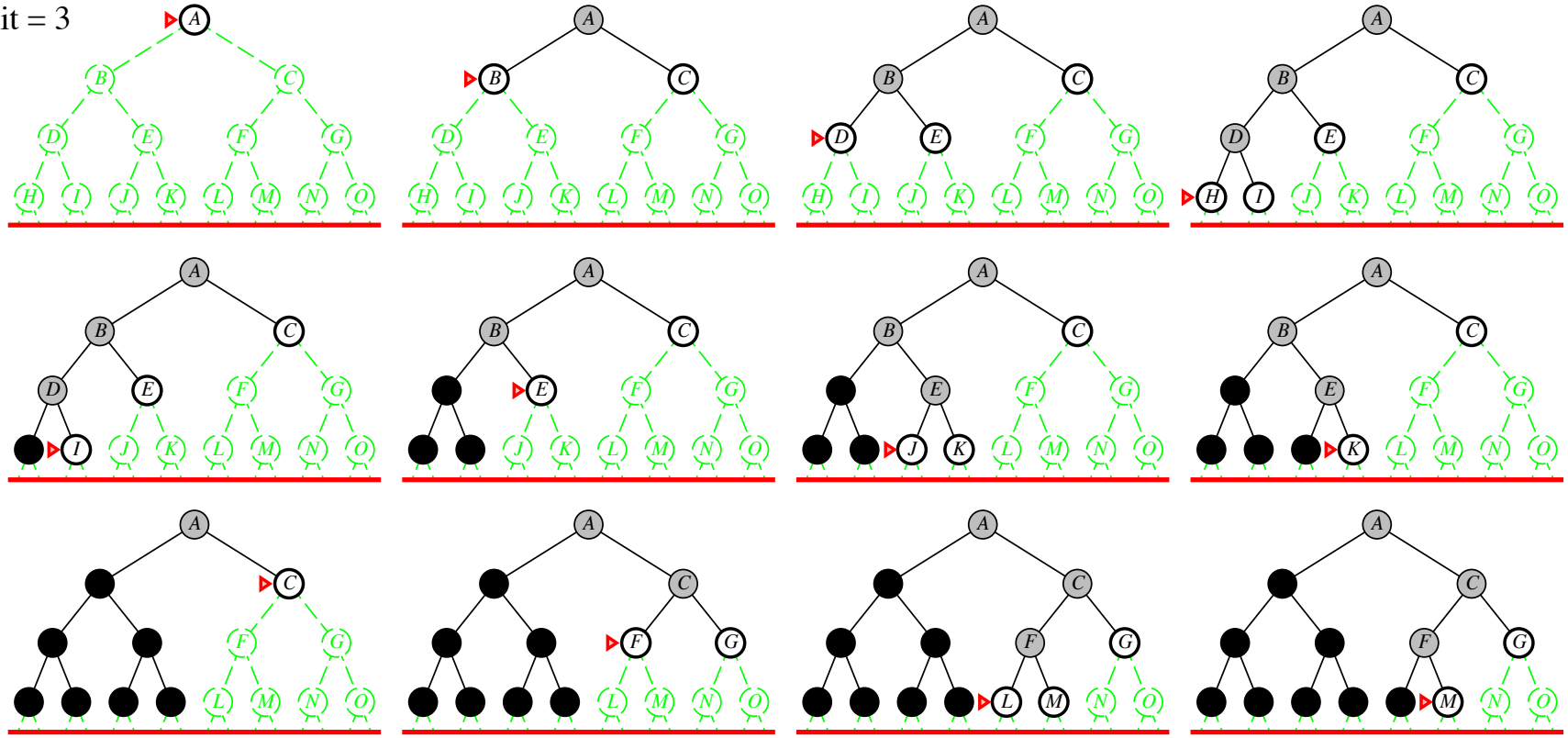
Ricerca Iterative Deepening $l = 2$

Limit = 2



Ricerca Iterative Deepening $l = 3$

Limit = 3



Proprietà della ricerca iterative deepening

Completa?? Si

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

Spazio?? $O(bd)$

Ottima?? Si, se costo del singolo passo $= 1$

Paragone numerico per $b = 10$ e $d = 5$, soluzione in fondo a destra:

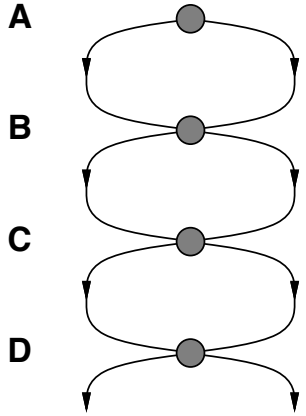
$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

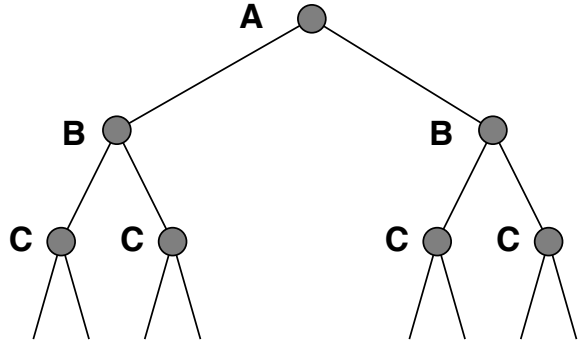
Riassunto degli algoritmi

Criterio	Breadth-First	Costo-Uniforme	Depth-First	Depth-Limited	Iterative Deepening
Completa?	Si*	Si*	No	Si, se $l \geq d$	Si
Tempo	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Spazio	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Ottima?	Si*	Si*	No	No	Si

Stati ripetuti

Se non si evitano stati ripetuti, un problema con numero di stati lineari può generare un numero esponenziale di nodi!





Stati ripetuti

Il problema si affronta di solito in 3 possibili modi:

- evitare di generare il nuovo nodo se è uguale al nodo corrente (spazio: costante)
- evitare di generare il nuovo nodo se è uno degli avi (nel cammino dalla radice al nodo corrente) (spazio: $O(d)$)
- evitare di generare il nuovo nodo se è stato già generato (spazio: $O(b^d)$, in realtà $O(s)$)

Ricerca su Grafo

```
function RICERCA-GRAFO( problema, frontiera) returns una soluzione, o fallimento
    chiuso ← un insieme vuoto
    frontiera ← INSERISCI(CREA-NODO(STATO-INIZIALE[problema]), frontiera)
    loop do
        if VUOTO?(frontiera) then return fallimento
        nodo ← RIMUOVI-PRIMO(frontiera)
        if TEST-OBIETTIVO[problema](STATO[nodo]) then return SOLUZIONE(nodo)
        if STATO[nodo] non è in chiuso then
            aggiungi STATO[nodo] a chiuso
            frontiera ← INSERISCI-TUTTI(ESPANDI(nodo, problema), frontiera)
    end
```