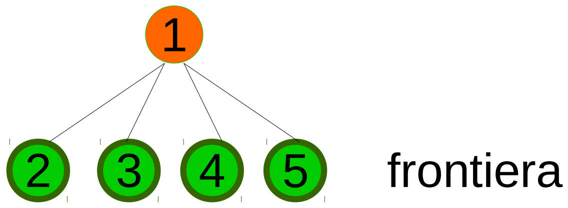
Schema di una strategia di ricerca

```
function RICERCA-ALBERO(problema) returns una soluzione, o il fallimento
    inizializza la frontiera usando lo stato iniziale di problema
    loop do
        if la frontiera è vuota then return fallimento
        scegli un nodo foglia e rimuovilo dalla frontiera
        if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
        espandi il nodo scelto, aggiungendo i nodi risultanti alla frontiera
```

1 frontiera

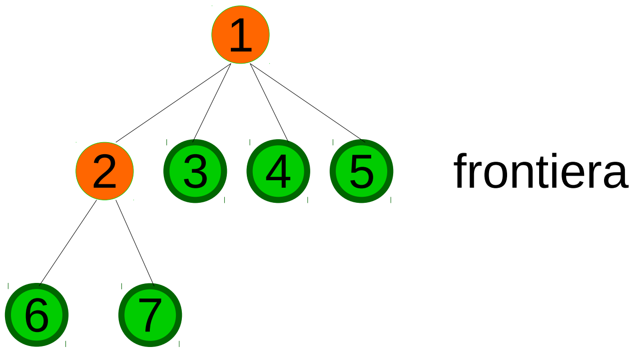
Schema di una strategia di ricerca

```
function RICERCA-ALBERO(problema) returns una soluzione, o il fallimento
    inizializza la frontiera usando lo stato iniziale di problema
    loop do
        if la frontiera è vuota then return fallimento
        scegli un nodo foglia e rimuovilo dalla frontiera
        if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
        espandi il nodo scelto, aggiungendo i nodi risultanti alla frontiera
```



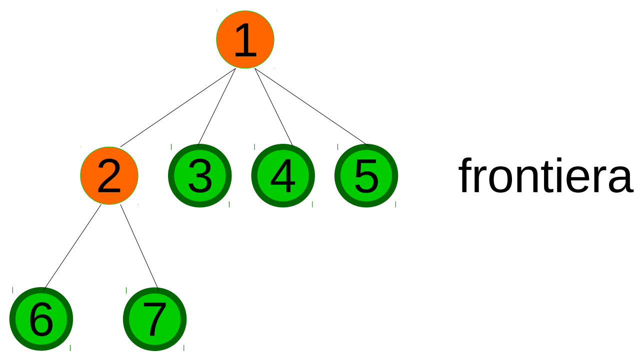
Schema di una strategia di ricerca

function RICERCA-ALBERO(*problema*) **returns** una soluzione, o il fallimento
  inizializza la frontiera usando lo stato iniziale di *problema*
  loop do
    if la frontiera è vuota **then return** fallimento
    scegli un nodo foglia e rimuovilo dalla frontiera
    if il nodo contiene uno stato obiettivo **then return** la soluzione corrispondente
    espandi il nodo scelto, aggiungendo i nodi risultanti alla frontiera



Schema di una strategia di ricerca

```
function RICERCA-ALBERO(problema) returns una soluzione, o il fallimento
    inizializza la frontiera usando lo stato iniziale di problema
    loop do
        if la frontiera è vuota then return fallimento
        scegli un nodo foglia e rimuovilo dalla frontiera
        if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
        espandi il nodo scelto, aggiungendo i nodi risultanti alla frontiera
```



Perché scegliere proprio il nodo 2?
A seconda del modo in cui è gestita la frontiera è in realtà possibile realizzare **diverse strategie** di ricerca ...

Confronto delle strategie

- Se le strategie sono tante ve ne è una migliore? Come le confronto?
- Criteri di valutazione:
  1. Completezza
  2. Ottimalità
  3. Complessità temporale
  4. Complessità spaziale

Confronto delle strategie

- Criteri di valutazione:
  1. Completezza:
     garanzia di trovare una soluzione, se esiste
  2. Ottimalità:
     garanzia di trovare una soluzione ottima (a costo minimo)
  3. Complessità temporale:
     quanto tempo occorre per trovare una soluzione
  4. Complessità spaziale:
     quanta memoria occorre per effettuare la ricerca

Come si valuta la complessità?

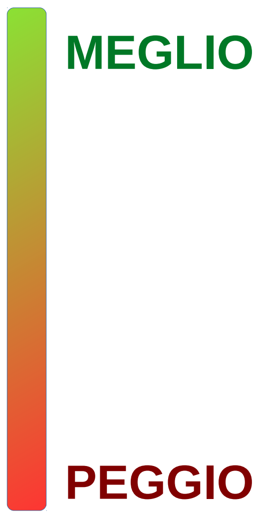
- Complessità computazionale: dato un problema, esistono infiniti algoritmi che lo risolvono, si può dire che uno sia migliore di un altro?
- Termine di paragone?
  - **Tempo**: quanto tempo richiede l'esecuzione nel caso (migliore, medio) peggiore?
  - **Spazio**: quanta memoria richiede l'esecuzione nel caso (migliore, medio) peggiore?
- Criterio di preferenza: **economicità**

Come si valuta la complessità?

- Gli algoritmi non sono programmi: **sono astratti**
- Se anche fossero programmi il tempo di esecuzione dipenderebbe dal calcolatore su cui sono eseguiti. Vogliamo una valutazione che prescinda da questi aspetti.
- **Tempo e spazio non metrici** ma parametrici:
  - e.g. calcolati in termini di **numero di nodi** (di un albero, di un grafo) **creati** o **visitati**
  - Tempo di esecuzione vero: costante ignota per il valore parametrico calcolato (es. numero passi, numero di nodi visitati)
- È interessante l' **andamento del costo** man mano che l' algoritmo viene applicato a strutture sempre più grandi o complesse

Notazione O-grande

- La notazione O-grande viene usata per specificare la complessità di un algoritmo, cioè quanto tempo/spazio l' esecuzione richiede in funzione di uno o più parametri, ad es:
  - **O(1)**: costante
  - **O(n)**: lineare
  - **O(n^2)**: quadratica
  - **O(n^3)**: cubica
  - **O(2^n)**: esponenziale



- **O(f(n))**: f è una funzione definita su numeri interi non negativi.

  Si legge O-grande di f(n) o dell' ordine di f(n)

Notazione O-grande

- Chiamiamo **T(n)** il tempo di esecuzione di un programma, espresso in termini della **taglia n dei dati**
- **T(n)** è **O(f(n))** se esiste un numero naturale n_0 e una costante $c > 0$ tale che $\forall\ n > n_0$ si ha $T(n) \leq c \cdot f(n)$

Approcci: quale e quanta conoscenza?

ALGORITMI

1) **Approcci blind:**
   usano esclusivamente la struttura del problema per cercare (e trovare) una soluzione

2) **Approcci informati:**
   usano la struttura del problema + ulteriore conoscenza per guidare la ricerca

   a) Monoagente

   b) Multiagente (giochi)

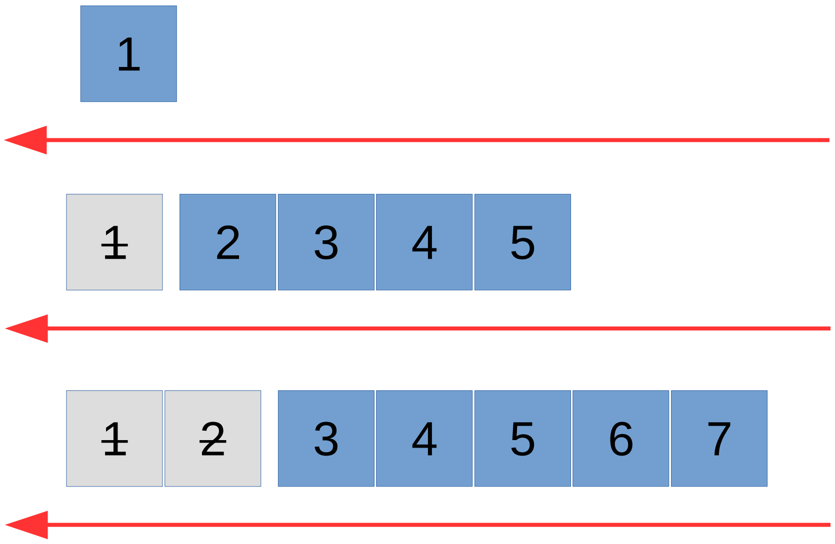
   c) Problemi di assegnamento (CSP)

Lista delle strategie

- Ricerca in ampiezza
- Ricerca a costo uniforme
- Ricerca in profondità (senza backtracking)
  - e variante a profondità limitata
- Iterative deepening
- Ricerca bidirezionale

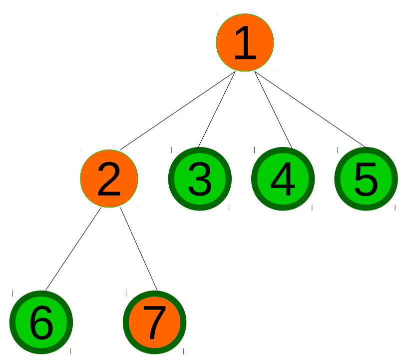
1. Ricerca in ampiezza

- La ricerca espande il nodo radice, poi tutti i suoi successori, poi tutti i discendenti di secondo livello, ecc.
- Si realizza gestendo la frontiera come una coda FIFO:



1. Ricerca in ampiezza

- Supponiamo che 7 sia il nodo obiettivo, allora la soluzione è catturata dal percorso 1 → 2 → 7
- Tutti i nodi sulla frontiera e tutti i loro antenati vanno tenuti in memoria per poter ricostruire la soluzione quando si trova il nodo obiettivo

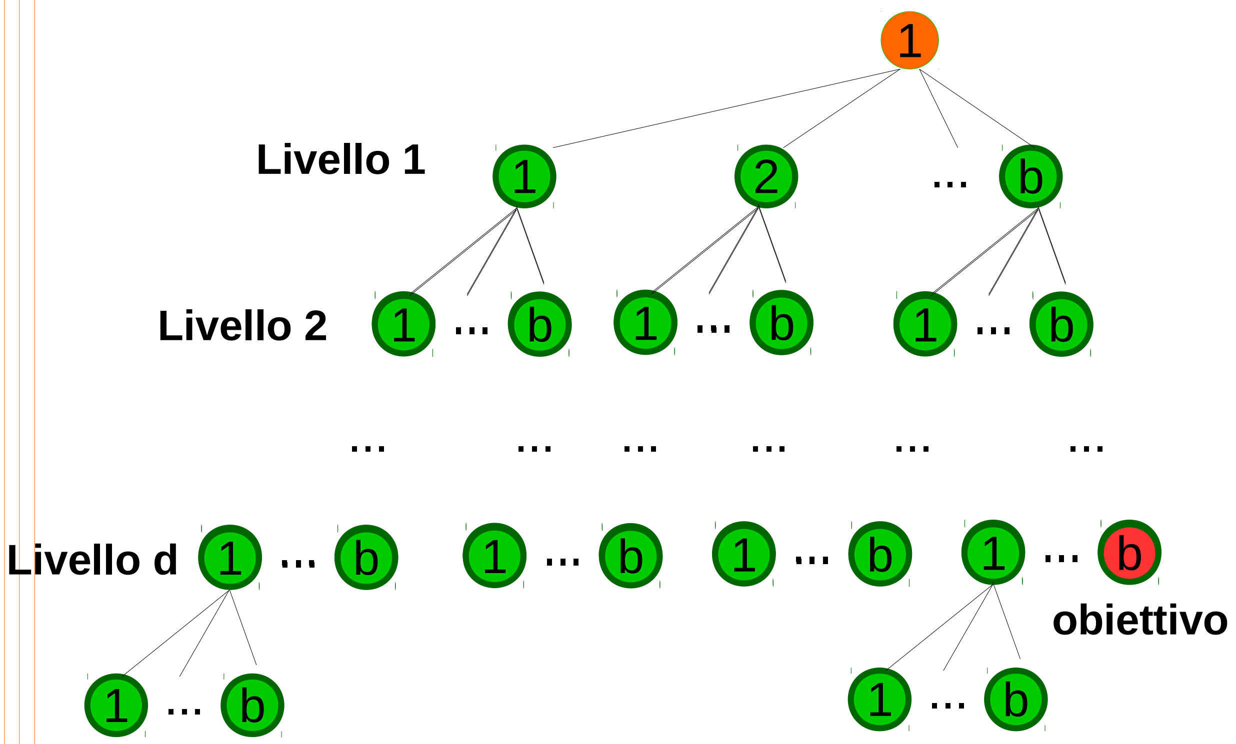


1. Ricerca in ampiezza: valutazione

- Completezza: se esiste un nodo obiettivo a una profondità finita *d*, la ricerca in ampiezza lo troverà a patto che il fattore di ramificazione *b* (cioè il numero di figli che un nodo può avere) sia finito
- Ottimalità: la soluzione trovata è ottima solo se il costo del cammino è una *funzione monotona crescente della profondità* (es. tutte le azioni hanno lo stesso costo)
- Complessità temporale: quanto tempo occorre per trovare una soluzione col crescere dello spazio di ricerca
- Complessità spaziale: quanta memoria occorre per effettuare la ricerca col creacere dello spazio di ricerca

1. Ricerca in ampiezza: valutazione

- Complessità temporale: è di tipo esponenziale $O(b^{d+1})$

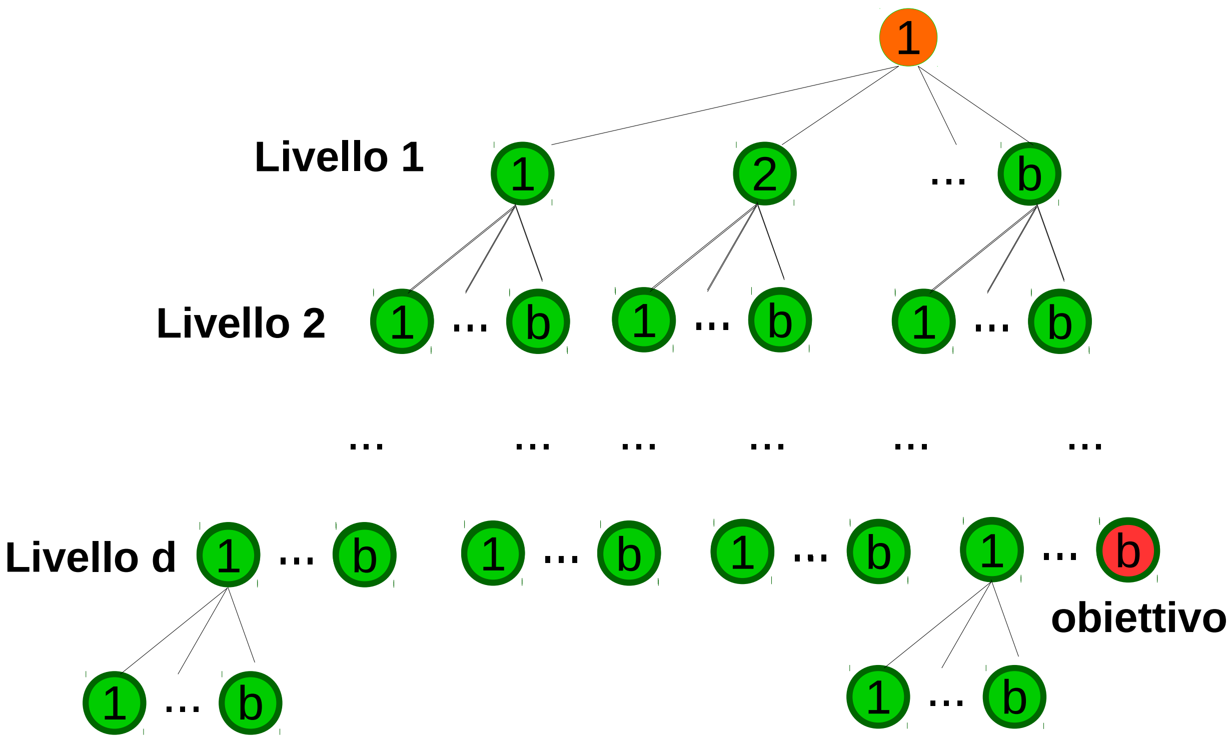


- Il tempo è misurato in termini di numero di nodi generati
- Nel caso peggiore ogni nodo avrà b figli

- Livello 1: b nodi (figli della radice)
- Livello 2: b^2 nodi (b nodi per ciascuno di quelli del Livello 1)
- Livello 3: b^3 nodi
- …
- Livello d (livello del nodo obiettivo più vicino): nel caso peggiore il nodo obiettivo sarà l'ultimo quindi si espandono tutti i nodi del livello d meno 1 producendo (b^{d+1} -b) nodi.

1. Ricerca in ampiezza: valutazione

- Complessità spaziale: $O(b^{d+1})$ perché tutti i nodi della frontiera e tutti i loro antenati vanno mantenuti in memoria



1. Ricerca in ampiezza: valutazione

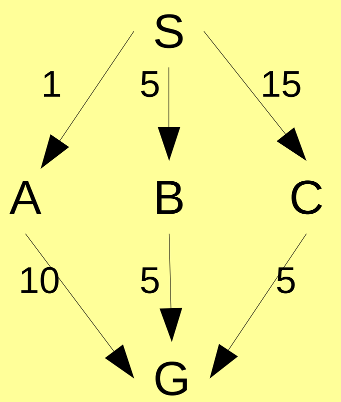
profondità	2	4	6	8	10	12	14
nodi	1100	111.100	10^7	10^9	10^{11}	10^{13}	10^{15}
tempo	0,11 sec	11 sec	19 min	31 h	129 gg	35 anni	3523 anni
memoria	1MB	106MB	10GB	1TB	101TB	10 peta B	1 exa B

- $O(b^{d+1})$ è bene o male?
- Supponiamo che il branching factor b=10, che vengano generati 10.000 nodi/sec e che 1 nodo occupi 1000 byte. La precedente tabella riporta le misure.

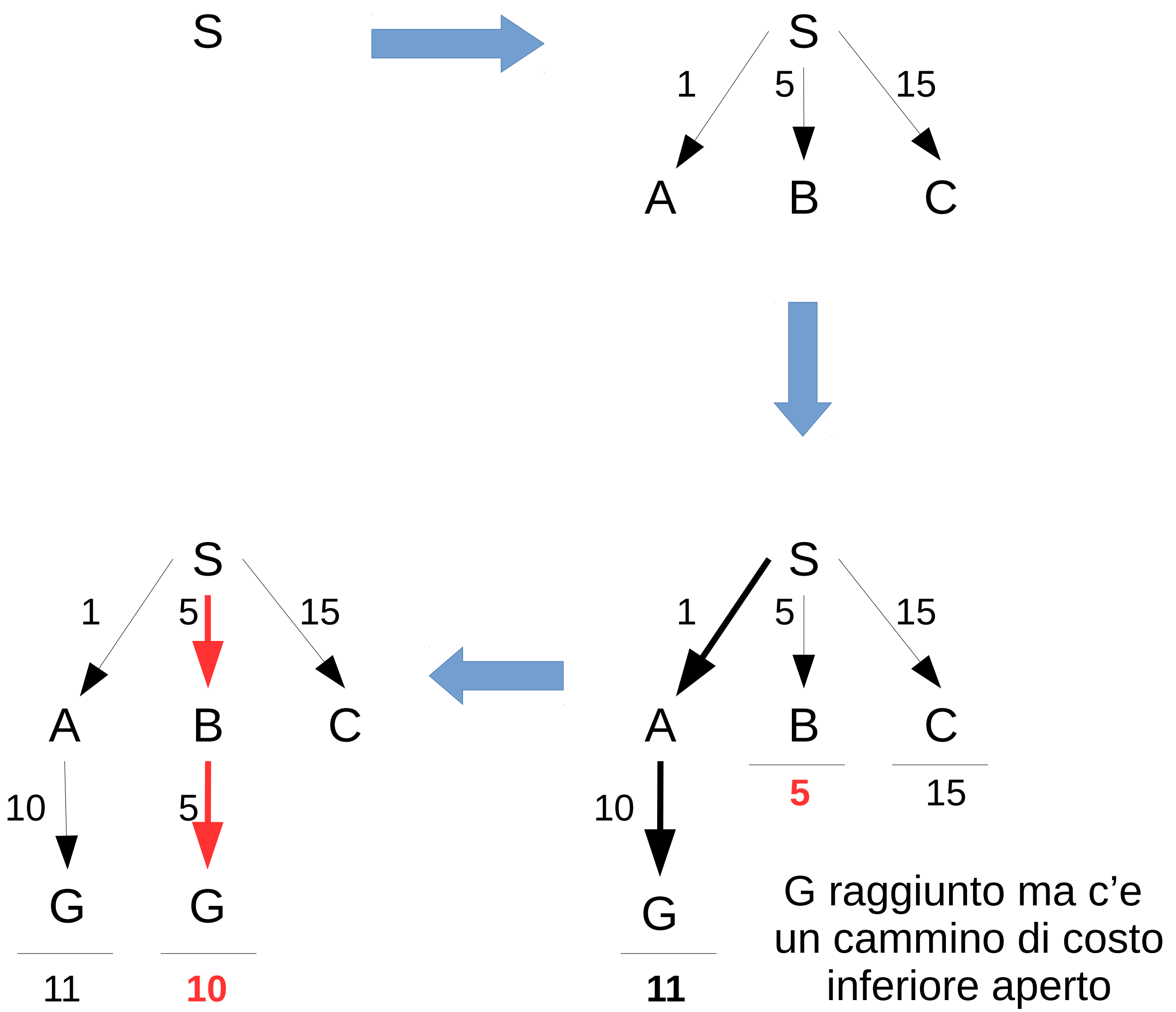
2. Ricerca a costo uniforme

- Quando i costi dei passi **non sono** tutti identici, può essere utile utilizzare la strategie di ricerca a costo uniforme:
  - ogni nodo ha associato il costo del cammino con cui è stato raggiunto
  - la frontiera è mantenuta ordinata
  - a ogni iterazione espande il nodo appartenente a un cammino di costo minimo
- Quando i costi sono tutti uguali diventa la ricerca in ampiezza
- Attenzione:
  - **COSTO ≠ NUMERO DEI PASSI**
    il numero dei passi effettuati non conta, conta solo il costo dei cammini
  - quando trova il nodo obiettivo non si ferma subito, prima controlla se vi sono cammini aperti di costo inferiore e nel caso prova a espanderli

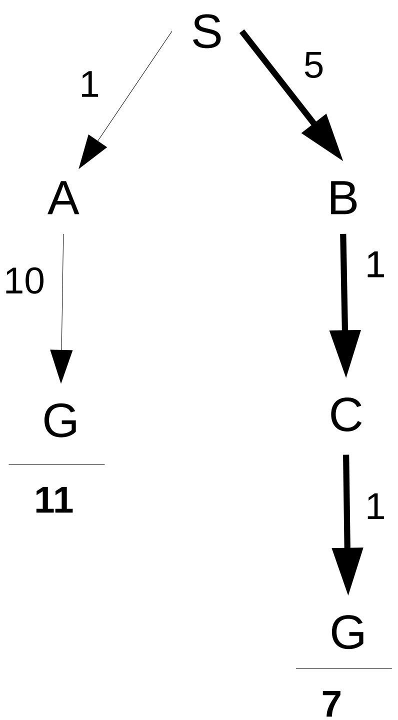
2. Ricerca a costo uniforme, esempio



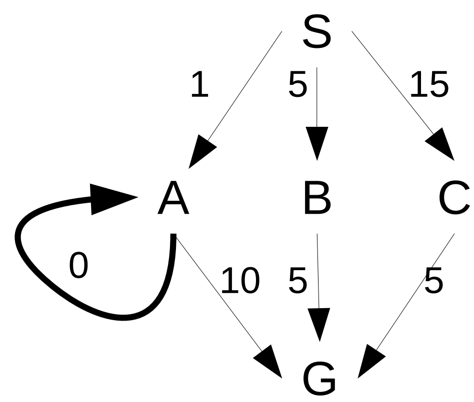
Grafo di transizione completo
(ignoto al risolutore di problemi)



2. Ricerca a costo uniforme, altri esempi



Il percorso a costo minimo ha più passi



Grafo di transizione completo (ignoto al risolutore di problemi)
Supponiamo contenga un passo a costo 0 (esempio noOp)

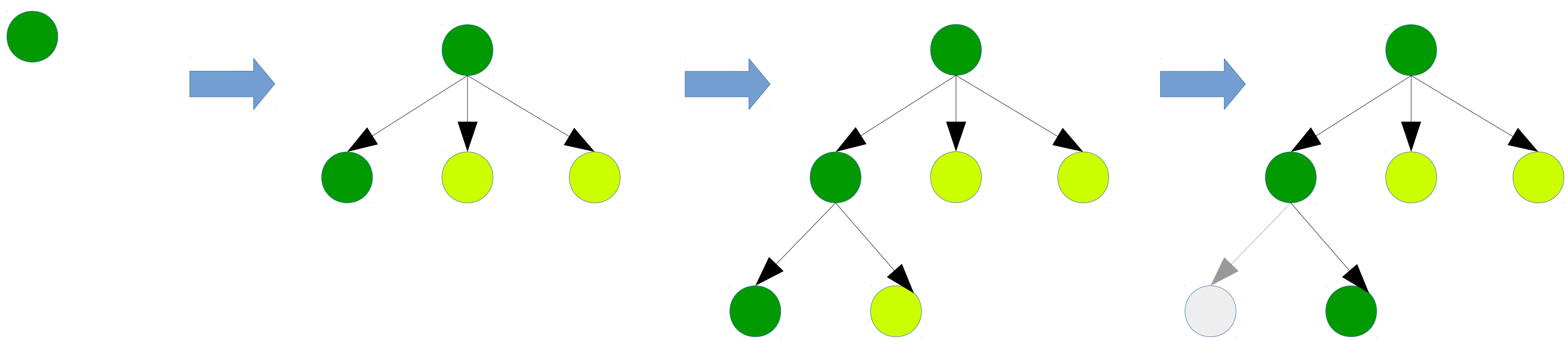
Loop infinito

2. Ricerca a costo uniforme: valutazione

- Completezza:
  garantibile solo se tutti i passi hanno **costo $\geq \varepsilon > 0$.**
  È il costo minimo delle operazioni
- Ottimalità:
  - garantibile solo se tutti i passi hanno **costo $\geq \varepsilon > 0$**
    MOTIVO: i costi dei cammini aumentano sempre con l'aumentare dei passi e l'algoritmo espande sempre quello attualmente di costo minimo
  - In altri termini: non sa gestire la nozione di guadagno unita a quella di costo
- Complessità temporale e spaziale:
  - **indipendenti da branching factor e profondità**; nel caso peggiore, detto C* il costo della soluzione ottima ed ε costo minimo delle azioni avremo $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
  - In parole è dell'ordine del branching factor elevato al numero di passi del percorso ottimale se i costi dei passi fossero uniformi
  - Quando i costi sono tutti uguali diventa come per la *visita in ampiezza*

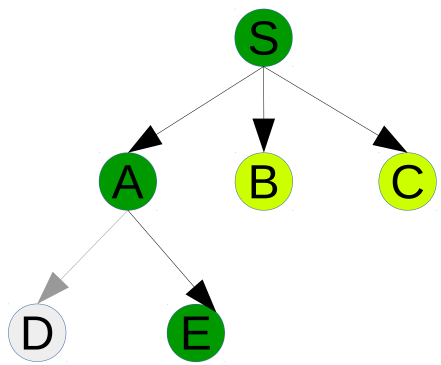
3. Ricerca in profondità senza backtracking

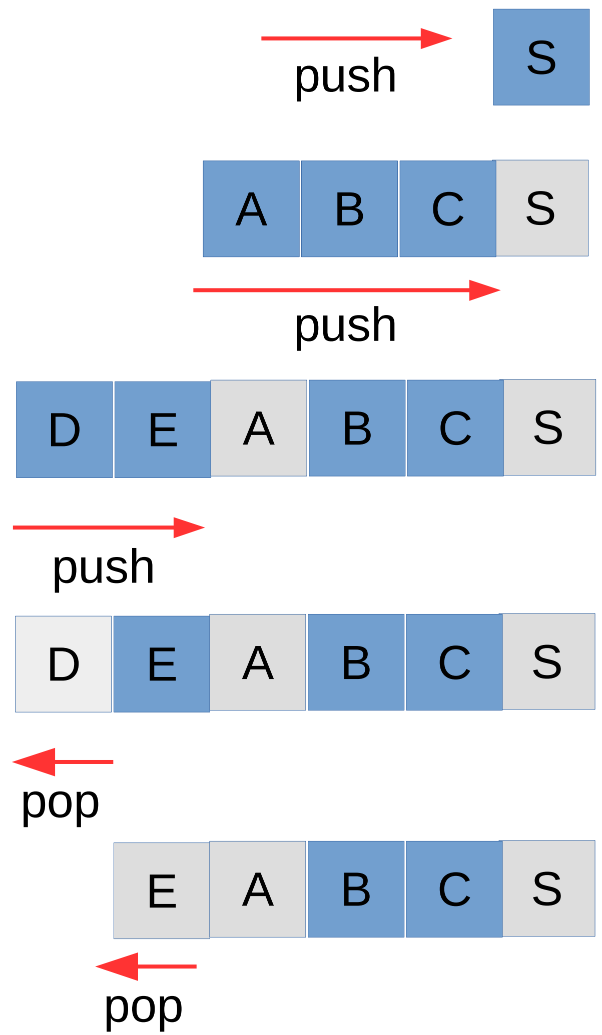
- La ricerca espande sempre il nodo più profondo della frontiera, cioè il più lontano dalla radice
- L'espansione produce tutti i successori di un nodo
- Quando la ricerca tenta di espandere un nodo che non ha successori, l'effetto è che il nodo viene rimosso e si "torna indietro" nell'albero per esplorare eventuali alternative



3. Ricerca in profondità

La frontiera è gestita come un coda LIFO, cioè come uno stack:





eccetera ...

Nodo espanso

Nodo non espanso

3. Ricerca in profondità, valutazione

- Complessità spaziale: O(b*m), dove:
  - b = branching factor
  - m = profondità massima
  - Infatti occorre mantenere tutti i nodi del cammino esplorato più tutti i loro fratelli
- Nel caso peggiore occorrerà mantenere l'informazione relativa a un numero di nodi pari a: b * m +1
- Confronto con ricerca in ampiezza, esempio di slide 33:
  di gran lunga migliore, se d = 12, basterebbero 118 KB invece di 10 peta B

3. Ricerca in profondità, valutazione

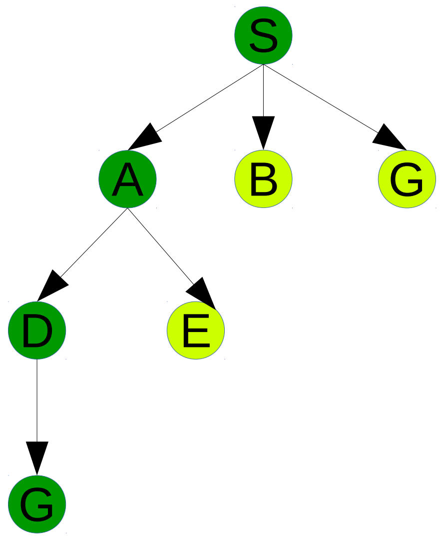
- Complessità spaziale:
- nella variante con backtracking, un nodo non viene espanso totalmente ma viene generato solo uno dei suoi possibili successori
- Se il cammino generato non porta alla soluzione si tornerà indietro e arrivati al nodo si genererà un altro dei suoi possibili successori
- Quindi la complessità spaziale diventa $O(m)$

3. Ricerca in profondità, valutazione

- Complessità temporale:
- Nel caso peggiore vengono percorsi tutti i nodi dell' albero
- Supponendo che b sia il branching factor medio e m la profondità massima la complessità temporale nel caso peggiore è $O(b^m)$

3. Ricerca in profondità, valutazione

- Completezza: garantibile <u>solo se</u> tutti i cammini sono finiti
- Ottimalità: in generale <u>non</u> è garantita, esempio:



Supponiamo che i passi abbiano lo stesso costo

La ricerca restituisce la soluzione:

S → A → D → G

Invece della soluzione ottimale

S → G

3. Variante: Ricerca a profondità limitata

- **Per evitare che la ricerca entri in percorsi infiniti** che non conducono alla soluzione, viene introdotto un limite artificiale, l:
  - Un nodo viene espanso solo se la sua **profondità p ≤ l**
  - Altrimenti viene trattato come un nodo privo di successori
- Tutti i cammini saranno lunghi al più l
- Problemi:
  - Si *riduce la completezza* perché la profondità dell'obiettivo è ignota a priori, quindi potrebbe essere superiore a l
  - Se invece l ≫ d, profondità minima dell'obiettivo, si *perde efficienza*

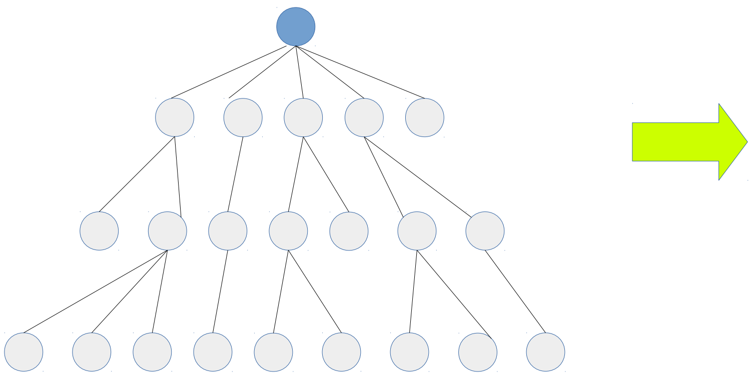
4. Iterative deepening depth first search

- Esegue una ricerca in profondità limitata, con iterazioni successive in cui la profondità massima viene aumentata via via
- Si noti che a ogni iterazione l'albero di ricerca viene ricostruito da zero
- L'indicazione "taglio" è la convenzione per dire che la ricerca è fallita

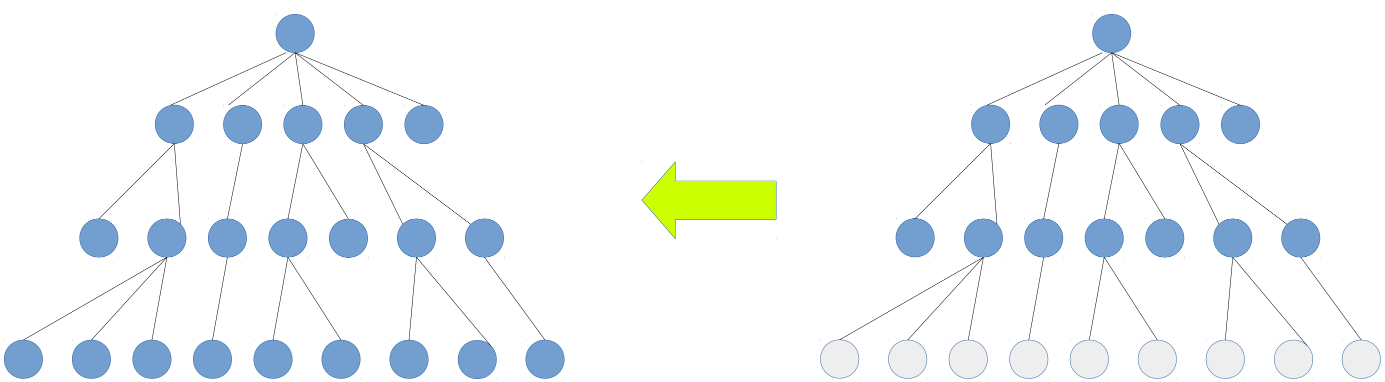
```
function RICERCA-APPROFONDIMENTO-ITERATIVO(problema) returns una soluzione, o il fallimento
    inputs: problema, un problema

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

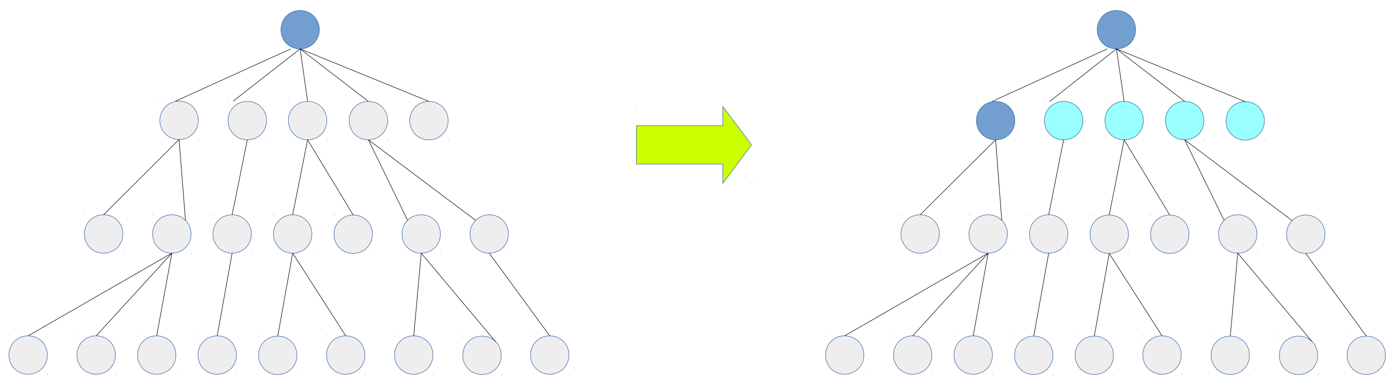
4. Ricerca per iterative deepening



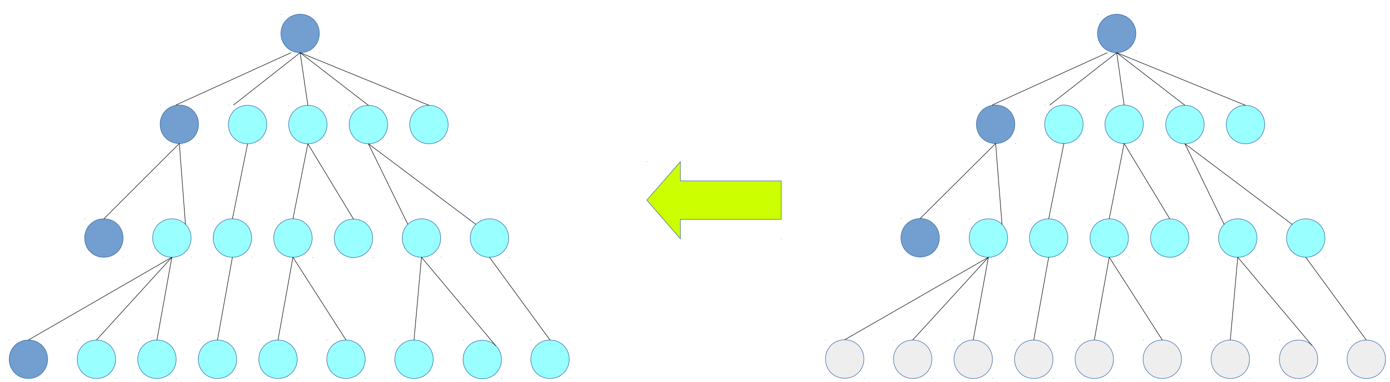
In questo esempio è come se venissero esplorati 4 alberi diversi, in successione



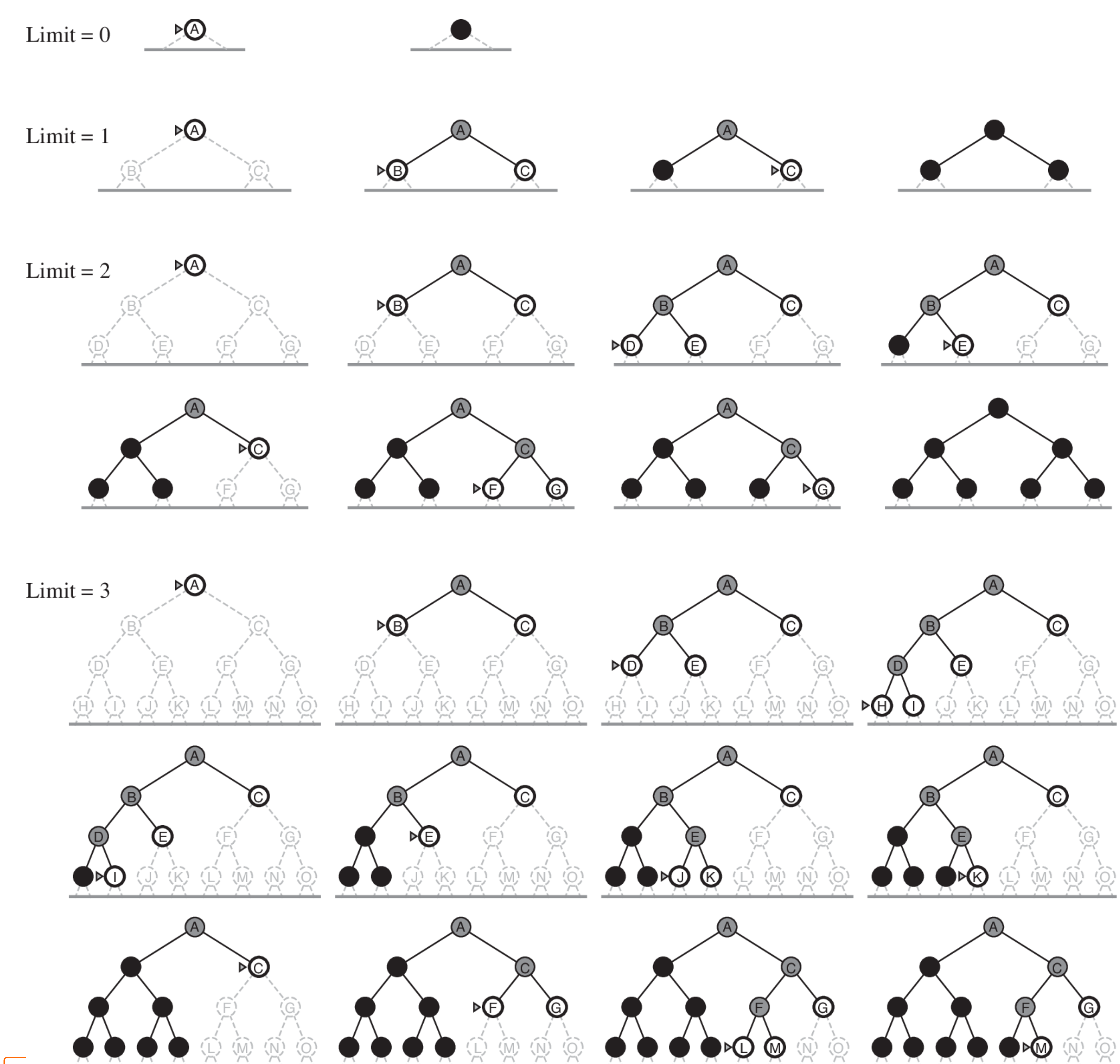
4. Ricerca per iterative deepening



I 4 alberi sono esplorati applicando la ricerca in profondità

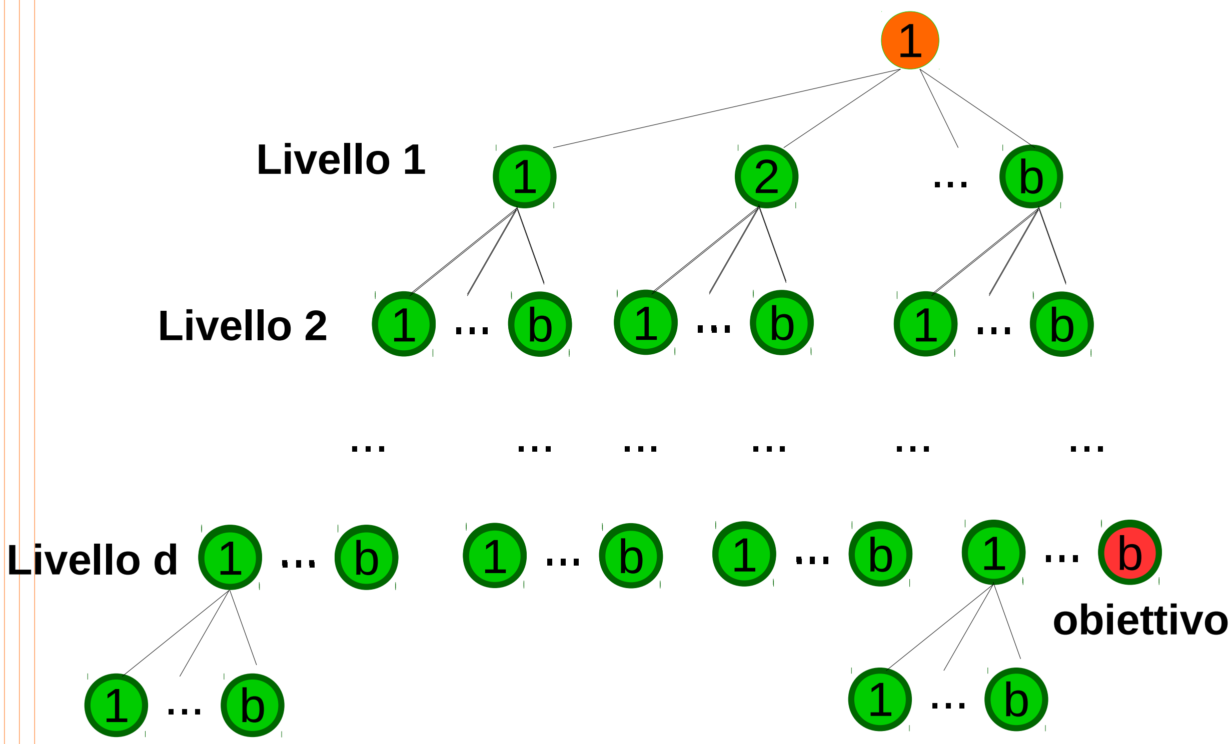


4. Ricerca per iterative deepening



Ampiezza: nodi espansi

- $O(b^{d+1})$: ricordiamo che vengono generati anche i nodi di livello d+1

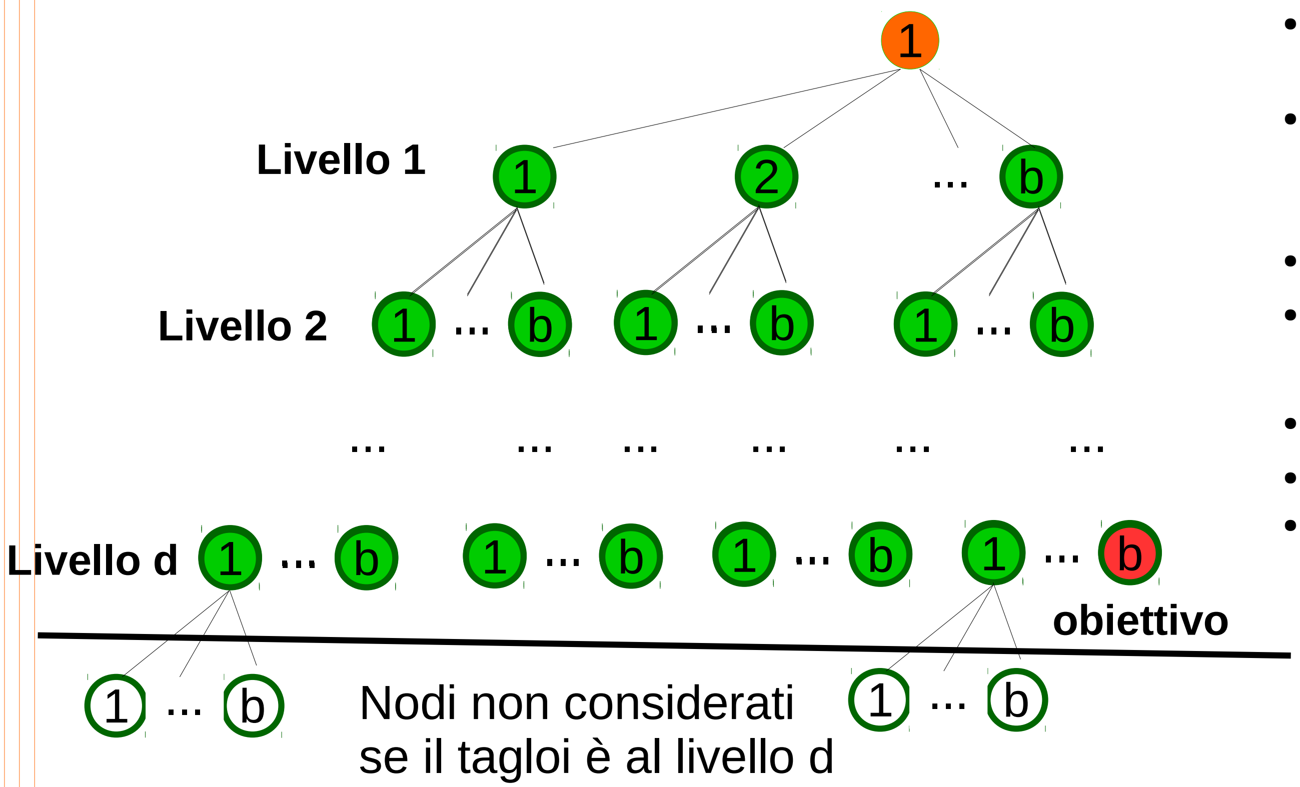


- Il tempo è misurato in termini di numero di nodi generati
- Nel caso peggiore ogni nodo avrà b figli

- Livello 1: b nodi (figli della radice)
- Livello 2: b^2 nodi (b nodi per ciascu-no di quelli del Livello 1)
- Livello 3: b^3 nodi
- …
- Livello d (livello del nodo obiettivo più vicino): nel caso peggiore il nodo obiettivo sarà l'ultimo quindi si espandono tutti i nodi del livello d meno 1 producendo $(b^{d+1}-b)$ nodi.

Iterative deepening: nodi espansi

- Ricordiamo che vengono generati i nodi solo fino al livello d, dove viene posto il taglio



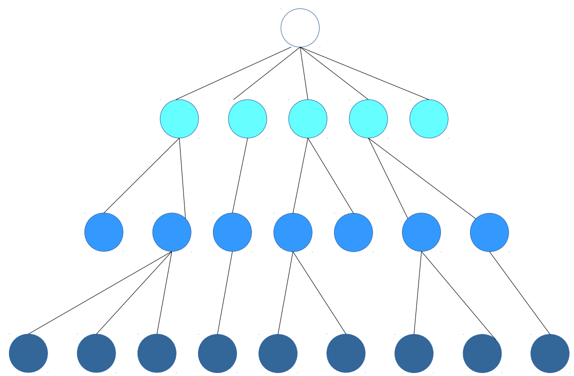
- Il tempo è misurato in termini di numero di nodi espansi
- Nel caso peggiore ogni nodo avrà b figli

- Livello 1: b nodi (figli della radice)
- Livello 2: b^2 nodi (b nodi per ciascuno di quelli del Livello 1)
- Livello 3: b^3 nodi
- …
- Livello d (livello del nodo obiettivo più vicino): nel caso peggiore il nodo obiettivo sarà l'ultimo quindi si considerano MA NON SI espandono tutti i nodi del livello d

Attenzione alla versione del libro: questa spiegazione non è corretta in tutte

4. Ripartire da zero costa molto?

Esempio, sia d = 3



0 — Nodi costruiti quattro volte (trascurabile)

1 — Nodi costruiti 3 volte

2 — Nodi costruiti 2 volte

3 — Nodi costruiti 1 volta

La maggior parte dei nodi si trova vicina alla frontiera, per cui cui ricostruire l'albero non risulta significativamente più costoso di espandere la sola frontiera. Il numero di nodi N_{id} complessivamente generato dall'iterative deepening è:

$$N_{id} = d{*}b + (d\text{-}1)b^2 + \ldots + (d\text{-}i)b^{i+1} + \ldots + 1{*}b^d = \Sigma_{\, i \in [1,d]} \, (d\text{-}i)b^i$$

dove d è la profondità minima dell'obiettivo e b il branching factor. **Complessità temporale: O(b^d)**. N_{id} risulta MINORE del numero di nodi generati dalla ricerca in ampiezza N_{amp} nel caso peggiore:

$$N_{amp} = b + b^2 + \ldots + b^i + \ldots + b^d + (b^{d+1} - b)$$

Il motivo è che in questo caso N_{amp} produce anche nodi di livello d+1

4. Perché ripartire da zero?

Poiché è <u>controintuitivo</u> che iterative deepening possa essere una strategia più efficiente della ricerca in ampiezza facciamo un esempio:

Siano $b=10$ e $d=5$

$N_{id} = (d+1)*b^0 + d*b + (d-1)*b^2 + \ldots + (d-i)*b^{i+1} + \ldots + 1*b^d =$

$= 6 + 50 + 400 + 3000 + 20.000 + 100.000 = 123.456$

$N_{amp} = b + b^2 + \ldots + b^i + \ldots + b^d + (b^{d+1} - b) =$

$= 10 + 100 + 1000 + 10.000 + 100.000 + 999.990 = 1.111.100$

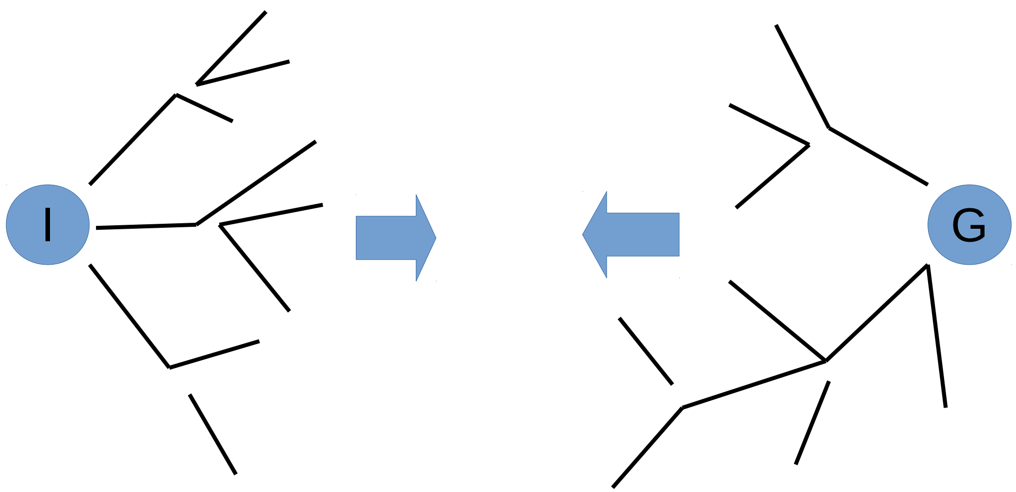
La ricerca in ampiezza crea meno nodi quando **esplora in minima parte il (d+1)-mo livello**, Con b=10 produce circa l'11% dei nodi prodotti dalla ricerca in ampiezza

4. Iterative deepening, valutazione

- Combina gli aspetti migliori delle ricerche in ampiezza e in profondità
- È preferito quando lo spazio di ricerca è ampio e la profondità della soluzione non prevedibile
- Come la ricerca in profondità ha una complessità spaziale modesta: $O(b*d)$, b = branching factor, d = profondità minima del nodo obiettivo
- La complessità temporale è alta, simile a ricerca in ampiezza: $O(b^d)$
- Come quella in ampiezza:
  - è completa quando b è finito
  - è ottima quando il costo è funzione non decrescente della profondità

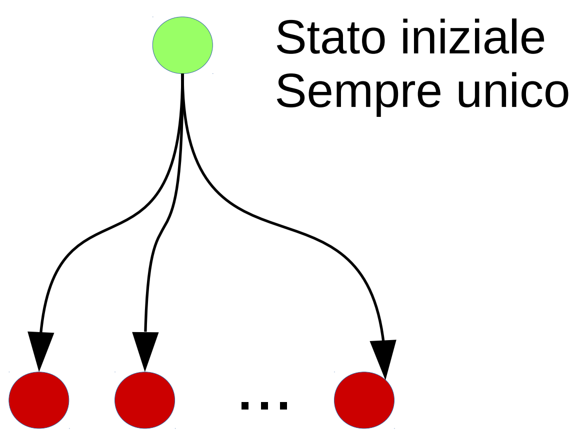
4. Ricerca bidirezionale

- È composta da due ricerche:
  - Una <u>forward</u> dallo stato iniziale
  - Una <u>backward</u> dallo stato obiettivo
- Termina quando le due ricerche si incontrano, cioè quando le frontiere hanno intersezione non vuota
- Il test si fa alla selezione (o espansione) di un nodo; se si usano hash table il tempo sarà costante

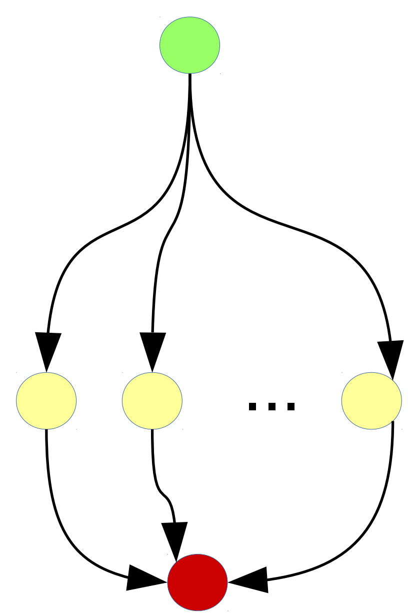


4. Ricerca bidirezionale

- E se lo stato obiettivo non è unico? Da dove iniziare la ricerca backward?
- Un modo è introdurre un nuovo stato di comodo, e renderlo raggiungibile in un paso da tutti gli stati obiettivo reali

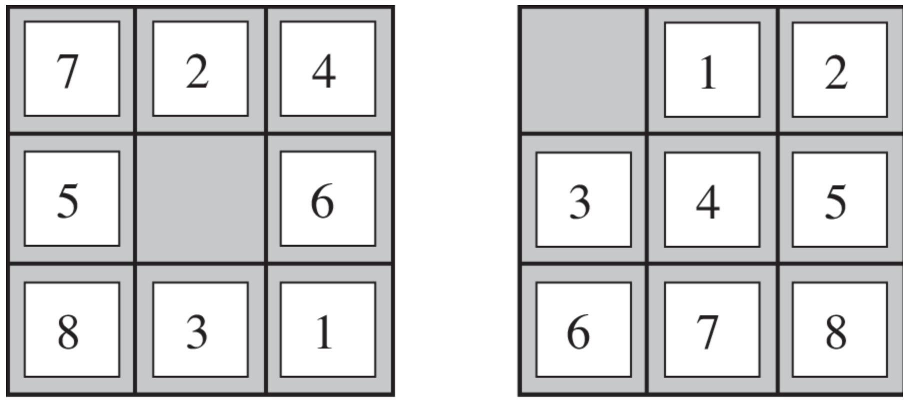


Molti possibili stati obiettivo



Nuovo stato obiettivo “di comodo”

4. ricerca bidirezionale

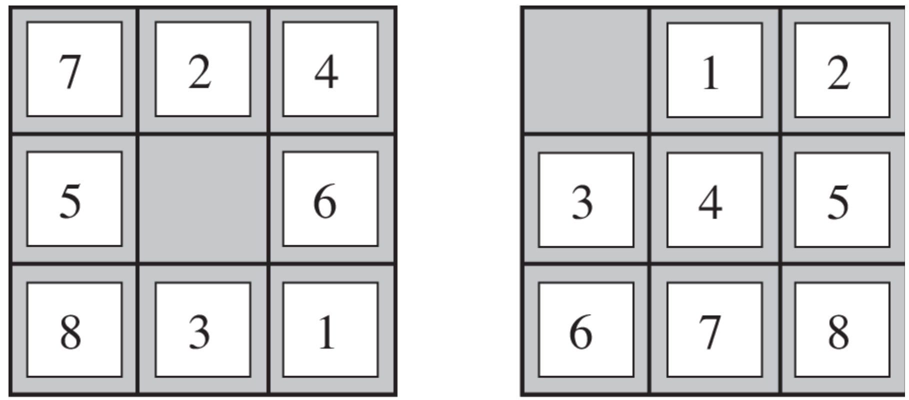


Start State　　Goal State

PROBLEMA: come determinare I possibili stati predecessori di un nodo?

Nel gioco dell'8 ricerca in avanti e all'indietro sono molto simili ma non è così per qualsiasi problema. Quando lo spazio degli stati è molto grande può essere difficile generare predecessori in modo efficiente.

4. ricerca bidirezionale



Start State Goal State

Immaginiamo gli operatori come aventi la forma

IF (antecedente) THEN (conseguente)

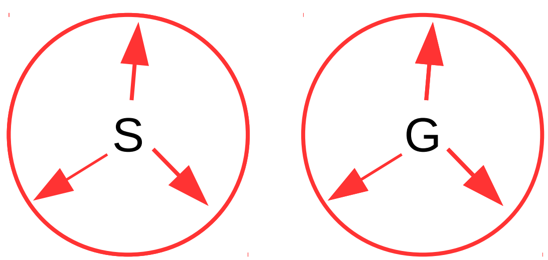
Ricerca forward: controllo quali operatori hanno l'antecedente verificato dallo stato corrente e produco I possibili stati successori applicando il conseguente

Ricerca backward: controllo quali operatori hanno il conseguente verificato dallo stato corrente e produco I possibili stati successori utilizzando la conoscenza degli antecedenti

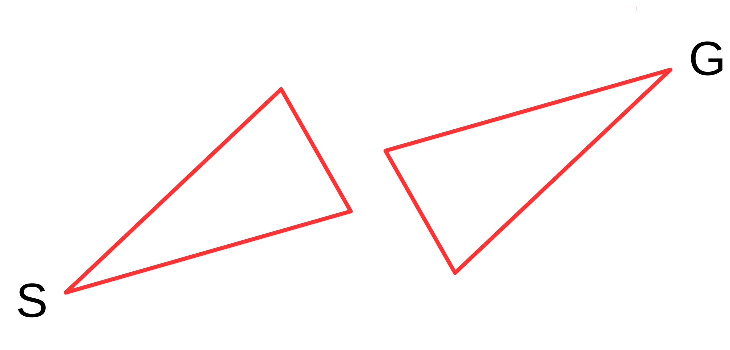
Terminazione: quando gli stati ottenuti ricercando backward hanno intersezione non vuota con quelli ottenuti ricercando forward

4. Ricerca bidirezionale

- Quando le due ricerche procedono si possono avere due andamenti



A fronte d'onda



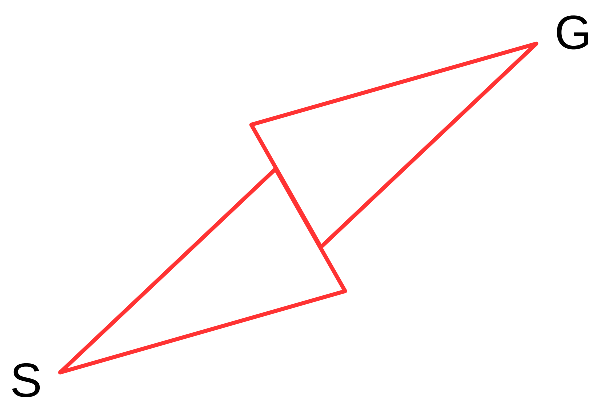
A cono

Quando non si ha informazione aggiuntiva si esplorano tutte le possibili operazioni

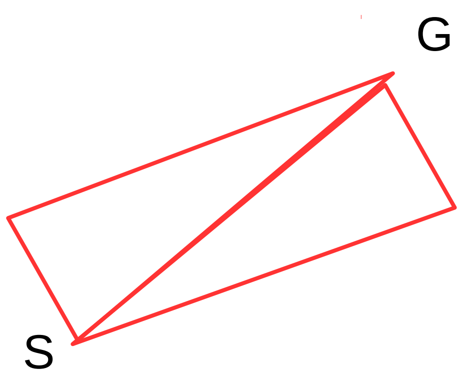
Quando si ha informazione aggiuntiva si esplora solouna parte delle operazioni

4. Ricerca bidirezionale: coni

- Quando le due ricerche esplorano una parte delle possibili operazioni l'ideale è che si incontrino a metà strada o si rischia di fare il doppio del lavoro



ideale



Da evitare

4. Ricerca bidirezionale: motivazione

- Interessante perché dai costi molto contenuti
- Complessità temporale e spaziale:
  $O(b^{d/2})$più precisamente $2*O(b^{d/2})$. È vantaggioso rispetto a $O(b^d)$?
  Molto! Esempio: se $b=2$ e $d=6$, $b^d = 64$, $b^{d/2} = 8$
- Completezza:
  se il branching factor è finito e se si usa una ricerca in ampiezza in entrambe le direzioni
- Ottimalità:
  se i costi dei passi sono identici e se si usa una ricerca in ampiezza in entrambe le direzioni

Grafi di ricerca e nodi già esplorati

function RICERCA-ALBERO(*problema*) **returns** una soluzione, o il fallimento
inizializza la frontiera usando lo stato iniziale di *problema*
loop do
if la frontiera è vuota **then return** fallimento
scegli un nodo foglia e rimuovilo dalla frontiera
if il nodo contiene uno stato obiettivo **then return** la soluzione corrispondente
espandi il nodo scelto, aggiungendo i nodi risultanti alla frontiera

function RICERCA-GRAFO(*problema*) **returns** una soluzione, o il fallimento
inizializza la frontiera usando lo stato iniziale di *problema*
inizializza al vuoto l'insieme esplorato
loop do
if la frontiera è vuota **then return** fallimento
scegli un nodo foglia e rimuovilo dalla frontiera
if il nodo contiene uno stato obiettivo **then return** la soluzione corrispondente
aggiungi il nodo all'insieme esplorato
espandi il nodo scelto, aggiungendo i nodi risultanti alla frontiera
solo se non è nella frontiera o nell'insieme esplorato

Figura 3.7 Una descrizione informale degli algoritmi di ricerca su albero e di ricerca su grafo. Le parti di RICERCA-ALBERO evidenziate in grassetto corsivo sono le aggiunte necessarie per gestire gli stati ripetuti.

Grafi di ricerca

- In molti problemi di ricerca lo stesso stato può essere raggiunto tramite percorsi differenti, causando una ripetizione di parte dell' esplorazione
- Marcare i nodi già visitati e controllare sempre se i nodi da esplorare sono per caso già stati visitati rende (molto) più efficiente la ricerca