Università di Parma
Dipartimento di Ingegneria e Architettura

Introduzione all'Intelligenza Artificiale

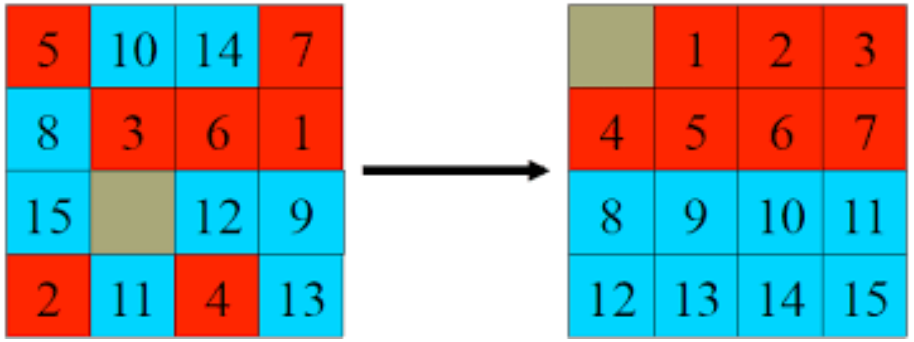
Big Data & Business Intelligence

A.A. 2022/2023

Corso di «Introduzione all'Intelligenza Artificiale»
Corso di «Big Data & Business Intelligence»

Agenti Intelligenti

Monica Mordonini (monica.mordonini@unipr.it)



PROBLEM SOLVING AGENTS

RICERCARE LE SOLUZIONI

Cercare soluzioni

Dopo aver formulato i problemi, si devono risolverli.
Una soluzione è una sequenza di azioni, quindi gli algoritmi di ricerca funzionano considerando varie possibili sequenze di azioni.

I problemi che i sistemi basati sulla conoscenza devono risolvere sono ***non-deterministici***
(don't know)

In un certo istante più azioni possono essere svolte
(azioni: applicazioni di operatori)

Cercare soluzioni

Passi da seguire:

1. Determinazione obiettivo

2. Formulazione del problema

3. Determinazione della soluzione mediante ricerca

4. Esecuzione del piano

Cercare soluzioni

- Stato iniziale

- Operatori: stato -> stato
  (funzione successore)

Spazio degli stati
insieme di tutti gli stati raggiungibili dallo stato iniziale attraverso qualsiasi sequenza di azioni

- Test obiettivo: stato -> {*true*, *false*}

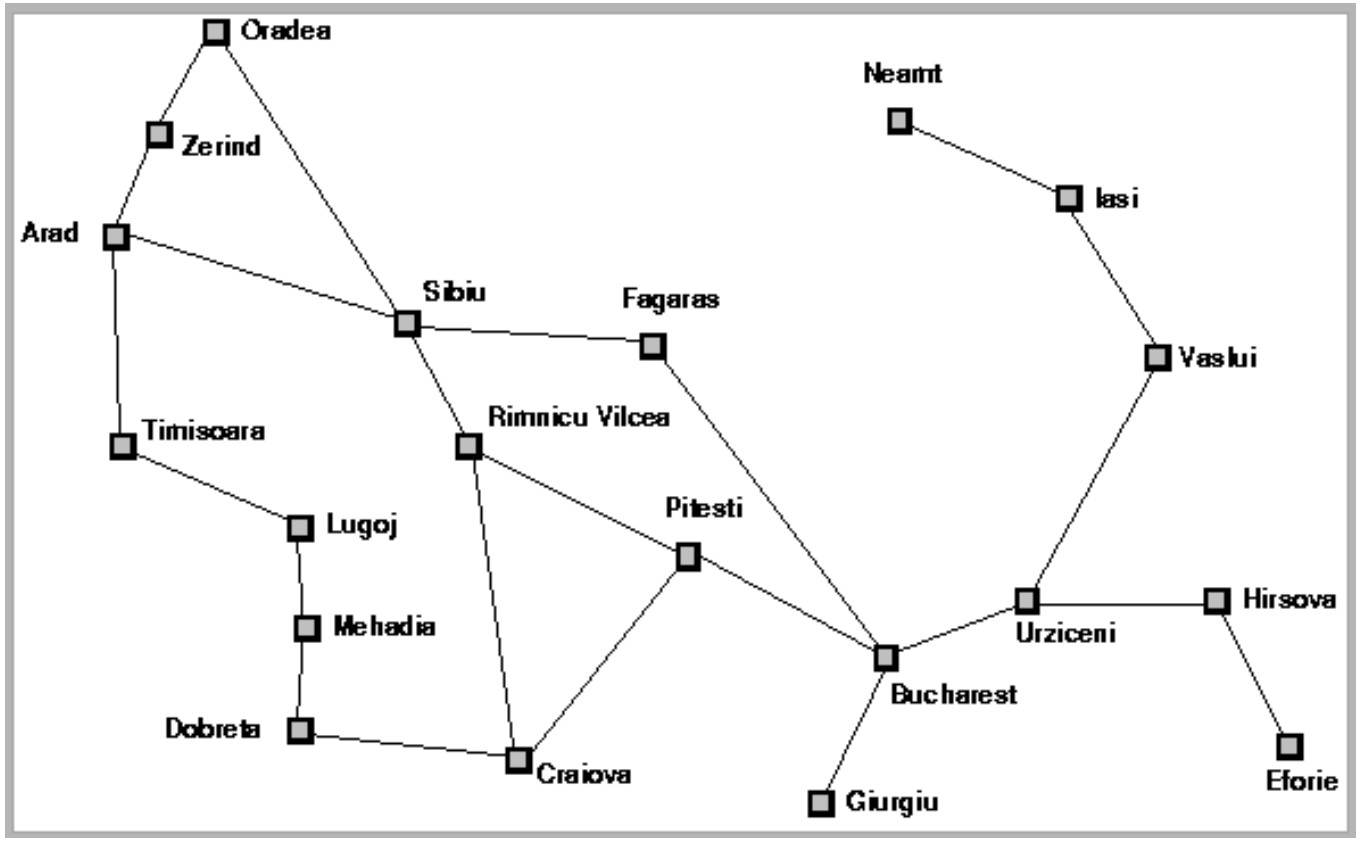
- Costo del cammino: assegna un costo ad ogni cammino

(somma dei costi delle azioni), spesso indicata con "g "

- *Un cammino nello spazio degli stati e semplicemente una qualsiasi sequenza di azioni che conduce da uno stato all'altro*

Cercare soluzioni

Vacanza in Romania



Cercare soluzioni

stato iniziale: essere “in Arad”

operatori: Arad -> Zerind, Arad -> Sibiu etc.

La funzione successore S fa passare dallo stato x agli stati S(x).

L’insieme degli stati raggiungibili definisce lo ***spazio degli stati***.

test obiettivo:

esplicito: “in Bucarest”

costo del cammino: es. Somma delle distanze, numero di operatori applicati, etc.

soluzione: una sequenza di operatori che porta da uno stato iniziale

a uno stato obiettivo.

Cercare soluzioni

Generare sequenze di azioni.

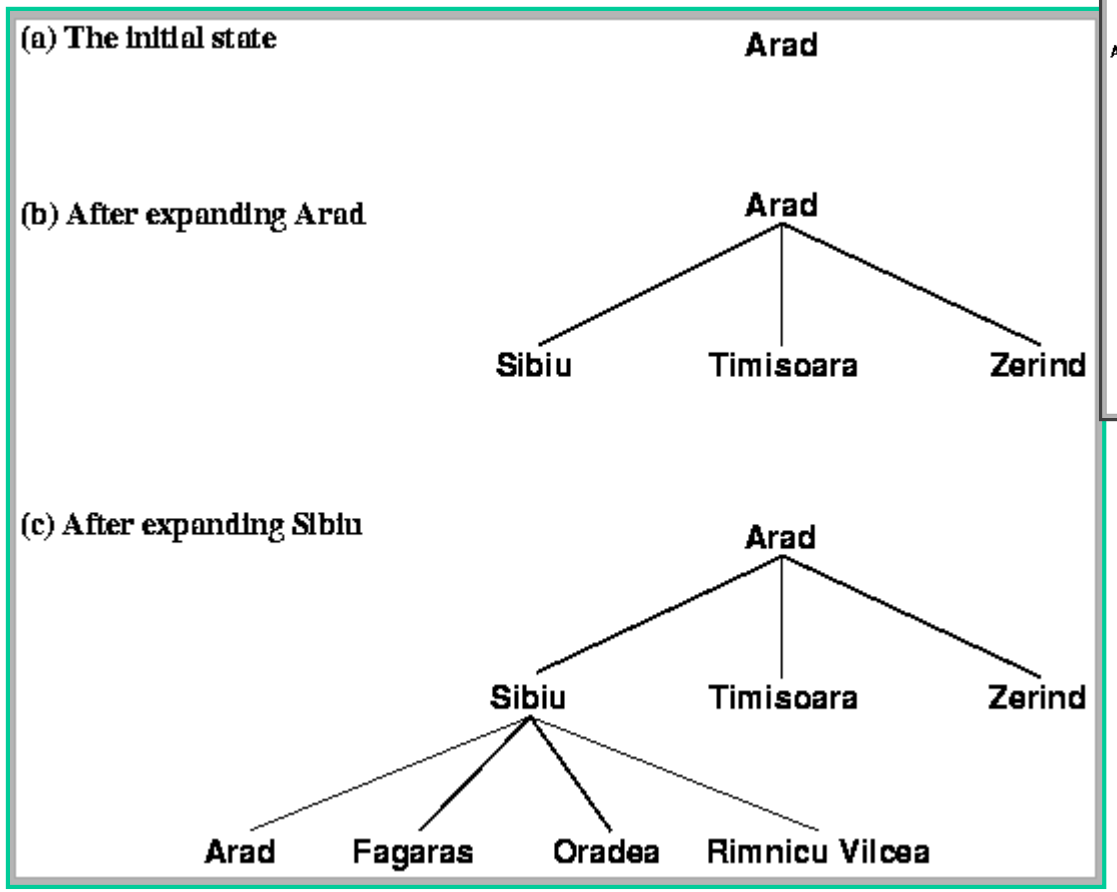
Espansione: si parte da uno stato e applicando gli operatori (o la funzione successore) si generano nuovi stati.

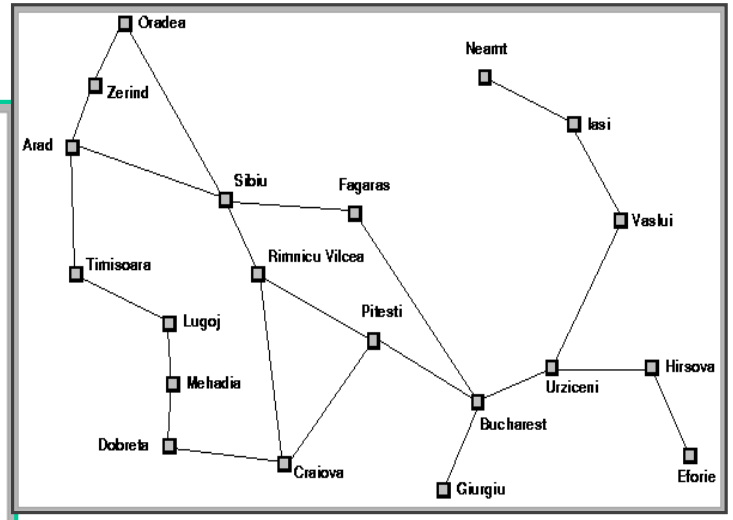
Strategia di ricerca: ad ogni passo scegliere quale stato espandere.

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.

Cercare soluzioni





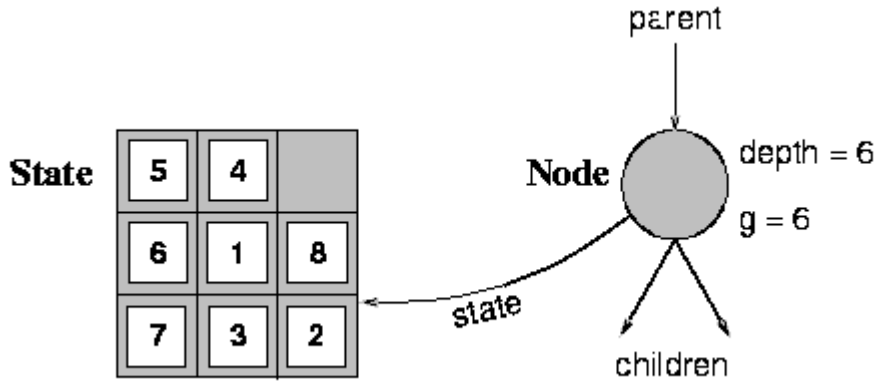
Albero di ricerca parziale per trovare un itinerario da Arad a Bucarest.

Cercare soluzioni

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



Cercare soluzioni

L'algoritmo generale di ricerca

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

N.B. Spazio degli stati ≠ albero di ricerca

Esempio Bucarest → 20 → Numero ∞ di nodi

Cercare soluzioni

L'algoritmo generale di ricerca su un grafo dove ci possono essere cicli

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Algoritmo Generale di Ricerca

- Cerchiamo di definire un algoritmo generale di ricerca per il caso di problemi a stato singolo.
- Per semplificare la sua definizione, introduciamo alcune funzioni ausiliarie.
- Occorre definire il tipo di dato ***Problem***

  Datatype `Problem`
  Components: `Initial-State, Operators, Goal-Test, Path-Cost-Function`
- Occorre una funzione per ricavare lo stato iniziale dalla descrizione del problema:
  - Init-State(Problem)
- Occorre una funzione per trasformare una descrizione di stato nel nodo di un albero:
  - Make-Node(State)

Algoritmo Generale di ricerca

- Occorre definire il tipo di dato ***Node***

  Datatype `Node`
  Components: `State, Parent-Node, Operator, Depth, Path-Cost`
- Occorre una funzione per espandere un nodo:
  - Expand(Node, Problem, [Depth ∞]), genera i successori del nodo se non è stata raggiunta la profondità Depth.
- Servono 4 funzioni per gestire delle code di nodi:
  - Make-Queue(Elements), genera una coda contenente Elements;
  - Empty(Queue), controlla se la coda è vuota;
  - Remove-Front(Queue), estrae il primo elemento;
  - Queueing-Fn(Queue, Elements), inserisce Elements nella coda.
- Infine serve una funzione per controllare se il nodo coincide con un goal del problema:
  - Goal(Node, Problem)

Algoritmo Generale di ricerca

- Le code sono caratterizzate dall'ordine in cui memorizzano i nodi inseriti. Tre varianti comuni sono
- first-in, first-out FIFO che prende l'lemento più vecchio ;
- last-in, first-out or LIFO (stack), che prende l'elemento più nuovo
- priority queue, che prende l'elemento della coda con priorità maggiore secondo una certa funzione o criterio

Algoritmo Generale di ricerca

- Possiamo definire un algoritmo generale per la ricerca su alberi come segue:

```
function General-Search(problem, Queueing-Fn[ Depth ∞]) {
  nodes = Make-Queue(Make-Node(Initial-State(problem)));
  while true {
    if Empty(nodes) return failure;
    node =Remove-Front(nodes);
    if Goal(State(node), problem) return node;
    nodes = Queueing-Fn(nodes, Expand(node, problem, Depth));
}
```

- L'insieme di nodi contenuti in nodes è detta ***frontiera*** dell'albero di ricerca.

Algoritmo Generale di ricerca

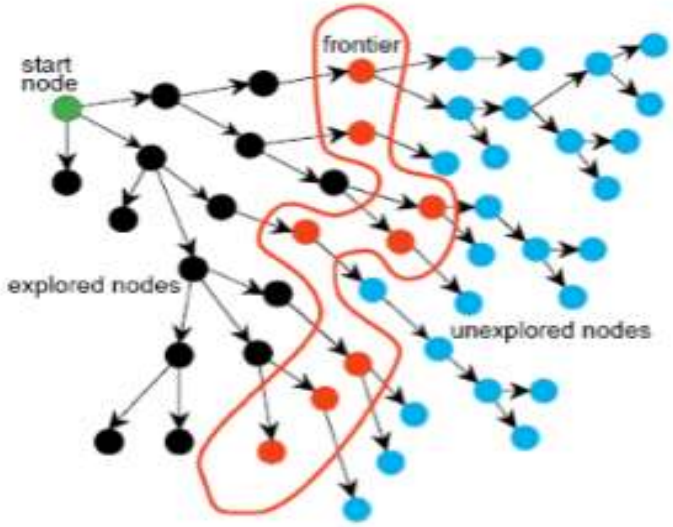
Important definitions

Search strategy

Decides, which node (among those determined by available actions) is expanded next.

Fringe (frontier)

Queue of nodes to be expanded.



Valutazione delle *strategie di ricerca*

- **Completezza** : l'algoritmo garantisce di trovare una soluzione, se questa esiste ?
- **Complessità temporale** : quanto tempo è necessario per raggiungerla ?
- **Complessità spaziale** : quanta memoria è richiesta per raggiungerla ?
- **Ottimalità** : la soluzione trovata è la migliore (a costo minimo) quando ci sono varie soluzioni differenti?

Valutazione delle strategie di ricerca

- La **complessità temporale e spaziale** è misurata in termini di:

  - ***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)

Valutazione delle strategie di ricerca

- La scelta di quale stato espandere nell'albero di ricerca prende il nome di ***strategia.***
- All search strategies are distinguished by the *order* in which nodes are expanded
- Distinguiamo:

- Strategie di *ricerca non informata* (***blind search*** o ricerca cieca)
  - Non si conosce informazione aggiuntiva sugli stati oltre a quelle fornite nella definizione del problema
- Strategie di *ricerca informata* (***ricerca euristica***)
  - Sanno distinguere se un obiettivo è più promettente di un altro

Strategie non-informate: principali

Non richiedono nessuna conoscenza specifica relativa al problema (approccio *brute-force*)
Il termine significa che le strategie non hanno informazioni aggiuntive sugli stati oltre a quelle fornite nella definizione del problema.
Tutto ciò che possono fare è generare successori e distinguere uno stato obiettivo da uno stato non obiettivo.
Vantaggio: *generalità*

- 1. breadth-first;
- 2. depth-first;
- 3. depth-first a profondità limitata;
- 4. ad approfondimento iterativo.

Ricerca in Ampiezza (Breadth-First Search)

- Un algoritmo di ricerca in ampiezza garantisce che ogni nodo di profondità *d* sia espanso prima di qualsiasi altro nodo di profondità maggiore.
- Possiamo definire un algoritmo di ricerca in ampiezza come segue:

```
function Breadth-First-Search(problem) {
  return General-Search(problem, Enqueue-At-
   End);
}
```

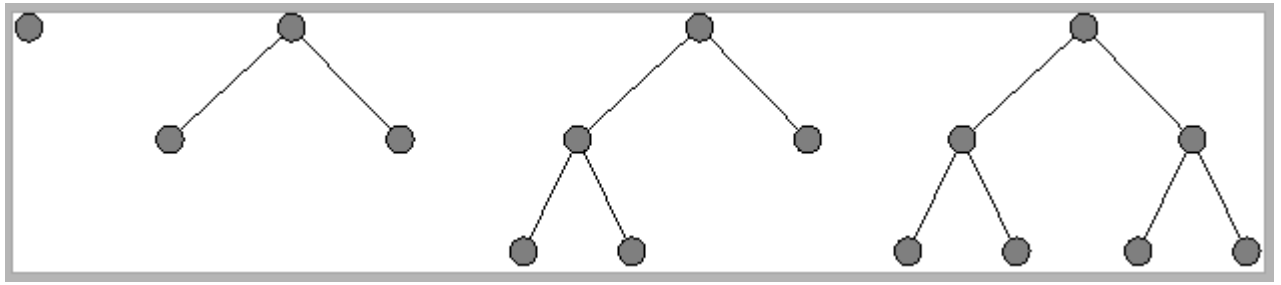
Ricerca in Ampiezza (Breadth-First Search)


```
function Breadth-First-Search(problem) {
  return General-Search(problem, Enqueue-At-End);
}
```

- La ricerca in ampiezza è un'istanza dell'algoritmo generale di ricerca su grafo in cui viene scelto per l'espansione il nodo non espanso più vicino alla radice
- .Ciò si ottiene molto semplicemente utilizzando una coda di tipo FIFO per analizzare i nodi della frontiera.
- Pertanto, i nuovi nodi (che sono sempre più profondi dei loro genitori) vanno in fondo alla coda e i vecchi nodi, che sono meno profondi dei nuovi nodi, vengono espansi per primi.

Ricerca in Ampiezza

Esempio



QueueingFn = metti i successori alla fine della coda

L'esplorazione dell'albero avviene tenendo contemporaneamente aperte più strade.

Ricerca in Ampiezza *su grafo*

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Ricerca in Ampiezza

- Ottimalità: è ottimo quando
  - il costo è una funzione non decrescente della profondità del nodo;
  - il costo è lo stesso per ciascun operatore.
- Completezza: è completo visto che utilizza una strategia sistematica che considera prima le soluzioni di lunghezza 1, poi quelle di lunghezza 2 e così via.
- Complessità: Se *b* è il fattore di ramificazione e *d* la profondità della soluzione, allora il numero massimo di nodi espansi è:
  $1 + b + b^2 + b^3 + \ldots + b^d$
- Complessità temporale: $O(b^d)$
- Complessità spaziale: $O(b^d)$

Ricerca in Ampiezza

- La figura mostra per vari valori della profondità di soluzione d, il tempo e la memoria necessari per una ricerca in ampiezza con fattore di ramificazione b = 10.

La tabella presuppone che sia possibile generare 1 milione di nodi al secondo e che un nodo richieda 1000 byte di memoria. Molti problemi di ricerca si adattano all'incirca a questi presupposti (dare o prendere un fattore 100) quando eseguiti su un moderno personal computer

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Ricerca in Ampiezza

- Per le ricerche in ampiezza i requisiti in memoria sono un problema maggiore del tempo di esecuzione.
  - Si potrebbero aspettare 13 giorni per la soluzione di un problema importante con profondità di ricerca 12, ma nessun personal computer ha il petabyte di memoria necessario. Fortunatamente, altre strategie richiedono meno memoria
- Ma anche quelli temporali non sono pochi
  - Se il problema ha una soluzione alla profondità 16, allora ci vorranno circa 350 anni prima che la ricerca in ampiezza (o addirittura qualsiasi ricerca non informata) la trovi
- In generale *i problemi di ricerca di complessità esponenziale non possono essere risolti con metodi non informati tranne che nelle istanze più piccole*

Ricerca a Costo Uniforme (Uniform Cost Search)

- L'algoritmo di ricerca in ampiezza trova la soluzione col valore di profondità più basso.
- Se consideriamo il costo della soluzione, questa soluzione può non essere la soluzione ottima.
- L'algoritmo di ricerca a costo uniforme permette di trovare *la soluzione a minor costo espandendo tra i nodi della frontiera il nodo caratterizzato dal minor costo* (già valutato).
- Se il costo è uguale alla profondità del nodo il funzionamento di questo algoritmo è uguale all'algoritmo di ricerca in ampiezza.

Ricerca a Costo Uniforme

- Possiamo definire un algoritmo di ricerca a costo uniforme come segue:

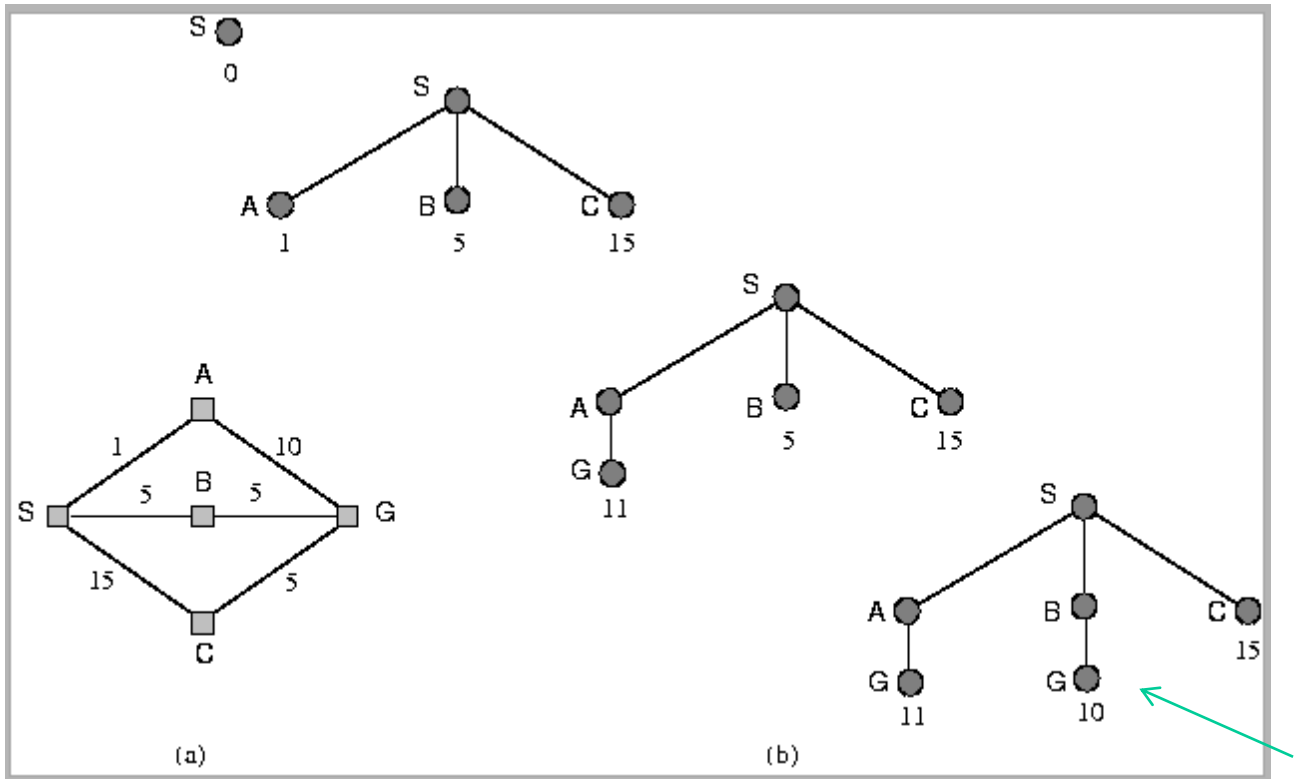
```
function Uniform-Cost-Search(problem) {
  return General-Search(problem, Enqueue-by-
   Cost);
}
```

Ricerca a Costo Uniforme

- Questo algoritmo è ottimo quando il costo è una funzione non decrescente della profondità del nodo.
- È completo poiché utilizza una strategia sistematica simile all'algoritmo di ricerca in ampiezza.
- Ha la stessa complessità temporale e spaziale dell'algoritmo di ricerca in ampiezza.

Ricerca a Costo Uniforme

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



Al passo successivo sarà scelto il nodo goal con g =10

QueueingFn = inserisci i successori in ordine di costo di cammino crescente

Ricerca a Costo Uniforme

- Questo algoritmo è in grado di trovare la soluzione più economica, purché sia verificato il requisito:
  - *il costo del cammino non deve mai decrescere quando percorriamo il cammino stesso.*

- Tale restrizione ha senso se il costo di cammino di un nodo viene considerato come somma dei costi degli operatori che determinano il cammino.
  - Se ogni operatore ha un costo non negativo, allora il costo di un cammino non può mai decrescere quando percorriamo il cammino e la ricerca a costo uniforme può così trovare il cammino più economico, senza dover esplorare l'intero albero di ricerca.

Ricerca in Profondità (Depth-First Search)

- L'algoritmo di ricerca in profondità, ad ogni livello di profondità, espande il primo nodo fino a raggiungere un goal o un punto in cui il nodo non può essere più espanso.
- La scelta del nodo da cui partire è arbitraria
- Possiamo definire un algoritmo di ricerca in profondità come segue:

```
function Depth-First-Search(problem) {
  return General-Search(problem, Enqueue-At-Front);
}
```

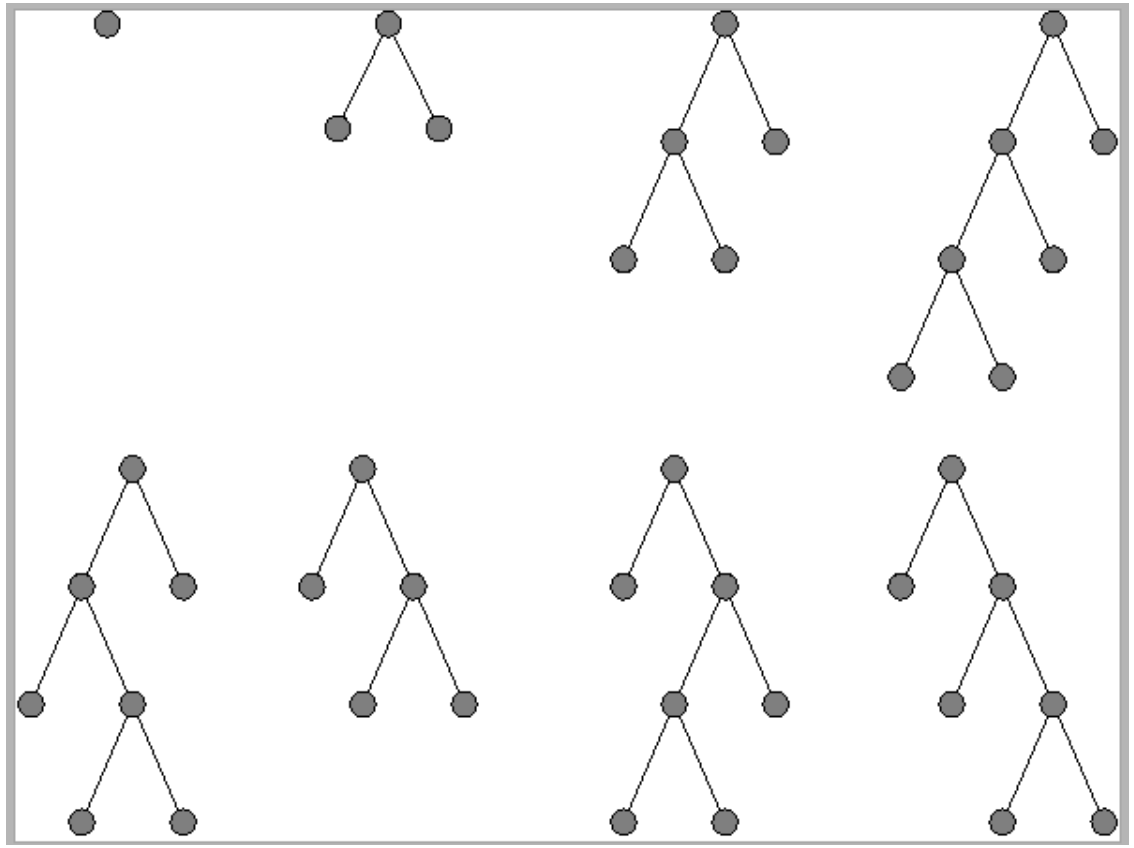
Ricerca in Profondità (Depth-First Search)


```
function Depth-First-Search(problem) {
  return General-Search(problem, Enqueue-At-
   Front);
}
```

- mentre la ricerca in ampiezza utilizza una coda FIFO, la ricerca in profondità utilizza una coda LIFO.
- Una coda LIFO significa che il nodo generato più di recente viene scelto per l'espansione. Questo deve essere il nodo non espanso più profondo perché è uno più profondo del suo genitore, che, a sua volta, era il nodo non espanso più profondo quando è stato selezionato.
- In alternativa all'implementazione in stile GRAPH-SEARCH, è comune implementare la ricerca in profondità con una funzione ricorsiva che richiama a sua volta ciascuno dei suoi figli.

Ricerca in Profondità

si assume che i nodi di profondità 3 non abbiano successori



QueueingFn = inserisci i successori all'inizio della coda.

Ricerca in Profondità

- Ottimalità: non è ottimo perché non usa nessun criterio per favorire le soluzioni a costo minore e quindi la prima soluzione trovata può non essere la soluzione ottima.
- Completezza: non è completo perché può imbattersi in un percorso di profondità infinita.
- Se b è il fattore di ramificazione e m è la massima profondità dell'albero, allora il numero massimo di nodi espansi è:
  $1 + b + b^2 + b^3 + \ldots + b^m$
- Complessità temporale: b^m *(paragonabile alla ricerca in ampiezza)*
- Complessità spaziale:: bm (*molto modesta*: i.e. riduz. a 109 rispetto a ricerca in amp. nel caso visto per d=12)

Ricerca in Profondità con backtracking search

- Una variante della ricerca in profondità chiamata **backtracking** utilizza ancora meno memoria.
- Nel backtracking, viene generato un solo successore alla volta anziché tutti i successori; ogni nodo parzialmente espanso ricorda quale successore generare successivamente. In questo modo è necessaria solo O(m) memoria anziché O(bm).La ricerca all'indietro facilita ancora un altro trucco per risparmiare memoria (e risparmio di tempo): l'idea di generare un successore modificando direttamente la descrizione dello stato corrente anziché copiarla prima. Ciò riduce i requisiti di memoria a una sola descrizione dello stato e O(m) azioni. Affinché funzioni, dobbiamo essere in grado di annullare ogni modifica quando torniamo a generare il successivo successore.
- Per problemi con descrizioni di stato di grandi dimensioni, come l'assemblaggio robotico, queste tecniche sono fondamentali per il successo.

Ricerca Limitata in Profondità (Depth-Limited Search)

- Per superare il problema dell'algoritmo di ricerca in profondità nell'esplorare alberi con rami con un numero infinito di nodi si può imporre un limite alla profondità dei nodi da espandere.
- La scelta del limite dovrebbe essere condizionata dalla previsione sulla profondità a cui si trova la soluzione.
- Possiamo definire un algoritmo di ricerca in profondità limitata come segue:

```
function Depth-Limited-Search(problem, Depth) {
  return General-Search(problem, Enqueue-At-Front, Depth);
}
```

Ricerca Limitata in Profondità

- Ottimalità: non è ottimo per gli stessi motivi della ricerca in profondità.
- Completezza: è completo nel caso in cui il limite sia superiore alla profondità della soluzione.
- Se b è il fattore di ramificazione e se fissiamo il limite di profondità dell'albero a L, allora il numero massimo di nodi espansi è:
  $1 + b + b^2 + b^3 + \ldots + b^L$
- Complessità temporale: b^L
- Complessità spaziale: bL

Ricerca Iterativa in Profondità (Iterative Deepening Search)

- Il problema principale dell'algoritmo di ricerca limitato in profondità è la scelta del limite a cui fermare la ricerca.
- Il modo per evitare del tutto questo problema è quello di applicare iterativamente l'algoritmo di ricerca limitato in profondità con limiti crescenti, cioè 0, 1, 2, ...
- In questo caso l'ordine di espansione dei nodi è simile a quello della ricerca in ampiezza con la differenza che alcuni nodi sono espansi più volte.

Ricerca Iterativa in Profondità

- Possiamo definire un algoritmo di ricerca in profondità iterativa come segue:

```
function Iterative-Deepening-Search(problem) {
  for Depth = 0 to ∞ do {
    result = General-Search(problem, Enqueue-At-
   Front, Depth);
    if result != failure return result;
  }
}
```

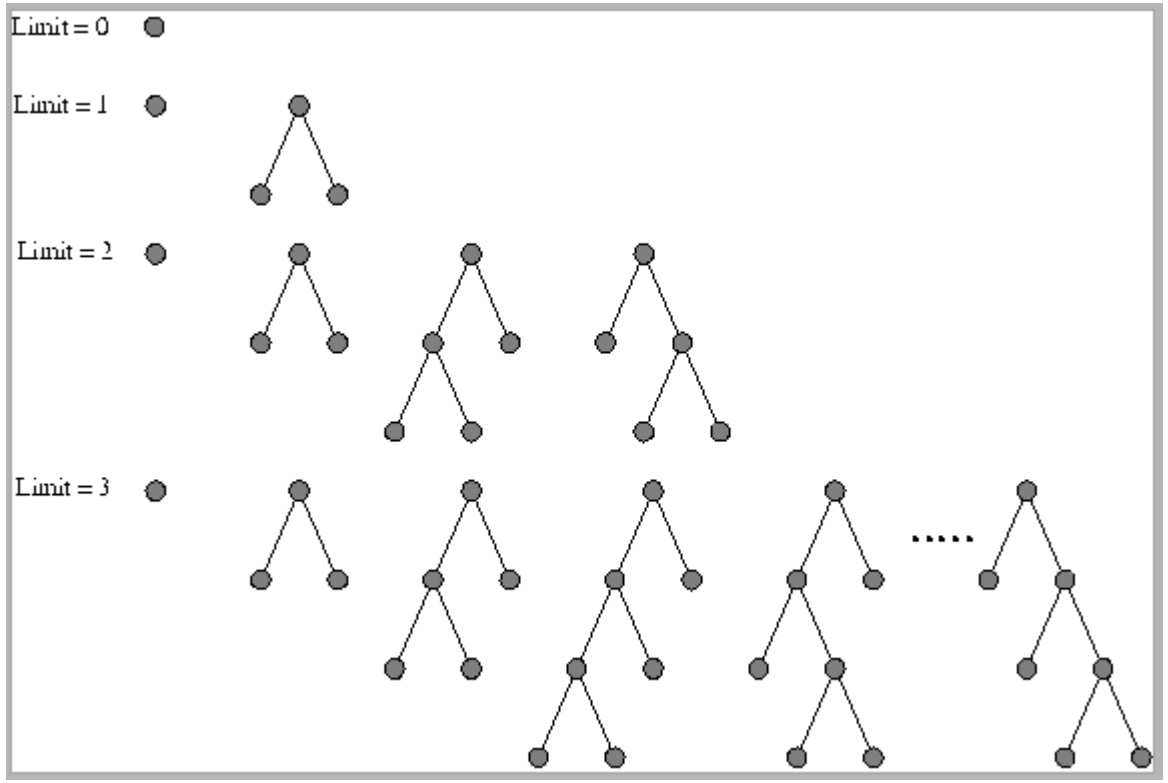
- Questo algoritmo è completo ed è ottimo quando l'algoritmo di ricerca in ampiezza è ottimo.

Ricerca Iterativa in Profondità

- Se b è il fattore di ramificazione e se la profondità della soluzione è d, allora il numero massimo di nodi espansi dall'algoritmo di ricerca iterativa in profondità è:
  $(d+1)1 + (d)\ b + (d-1)b^2 + (d-2)\ b^3 + \ldots + 1b^d$
- Complessità temporale: b^d
- Per $b = 10$ e $d = 5$ rendendo iterativo l'algoritmo passiamo da 111111 a 123456 nodi espansi.
- Complessità spaziale: bd
- ***Questo metodo è preferibile quando lo spazio di ricerca è grande e la profondità della soluzione non è conosciuta.***

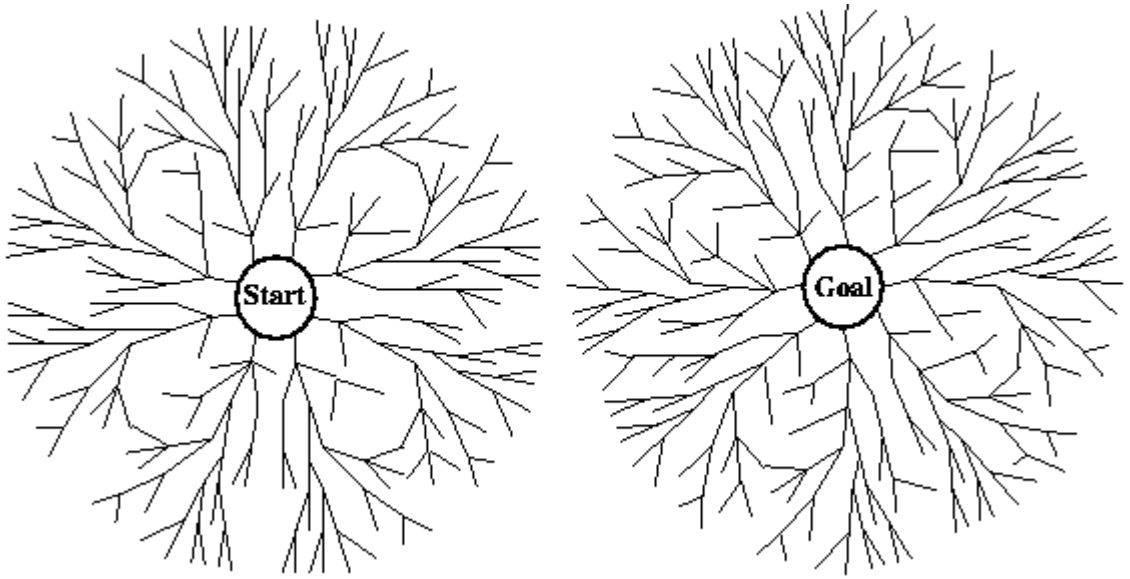
Ricerca Limitata in Profondità

Ricerca con approfondimento iterativo



Ricerca Bidirezionale

(Bidirectional Search)



Non appena un ramo partito dal nodo iniziale incontrerà un ramo partito dal nodo obiettivo la ricerca si concluderà con successo

Ricerca Bidirezionale (Bidirectional Search)

- L'idea è quella di cercare contemporaneamente in avanti dallo stato iniziale e indietro dal goal.
- Se b è il fattore di ramificazione e d è la profondità della soluzione, allora la complessità temporale dall'algoritmo di ricerca bidirezionale è: $2b^{d/2}$ che può essere approssimato con: $b^{d/2}$.
- Per utilizzarlo bisogna superare alcuni problemi:
  - La ricerca all'indietro richiede di generare i predecessori di un nodo. Per fare ciò, gli operatori devono essere reversibili.
  - Come si tratta il caso in cui ci sono diversi goal possibili?
  - Bisogna controllare in modo efficiente se un nodo è già stato trovato dall'altro metodo di ricerca.
  - Che tipo di ricerca è preferibile utilizzare nei due sensi?

Confronto fra le strategie di ricerca

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

b = fattore di ramificazione;

d = profondià della soluzione;

m=profondità massima dell'albero di ricerca;

l=limite di profondità.

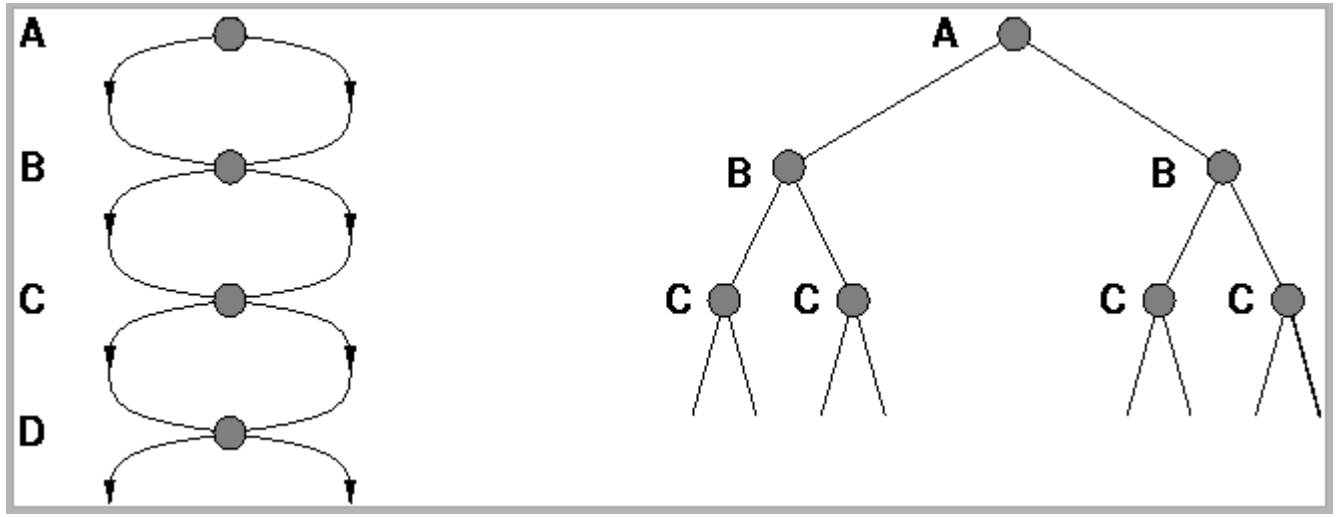
Eliminazione della Ripetizione di Percorsi

- In certi problemi si può risparmiare molto tempo evitando di espandere dei nodi già visitati. Infatti, se non evitiamo di ripetere la ricerca sugli stessi nodi lo spazio di ricerca può essere infinito (es se un operatore, applicato allo stato A, genera lo stato B e, applicato allo stato B, genera lo stato A), mentre lo rendiamo finito tagliando i nodi ripetuti.

Eliminazione della Ripetizione di Percorsi

- Per fare ciò esistono tre tipi di controllo, che hanno diversa complessità ed efficienza:
  - Viene proibito di ritornare nel nodo che ha generato il nodo corrente.
  - Viene proibito di ritornare in un nodo antenato del nodo corrente.
  - Viene proibita la generazione di un nodo già esistente.

Evitare ripetizioni di stati



Uno spazio degli stati che genera un albero di ricerca esponenziale .

Il lato sinistro mostra lo spazio degli stati, nel quale ci sono due azioni possibili che conducono da A a B, due da B a C e così via.

Il lato destro mostra l'albero di ricerca corrispondente.