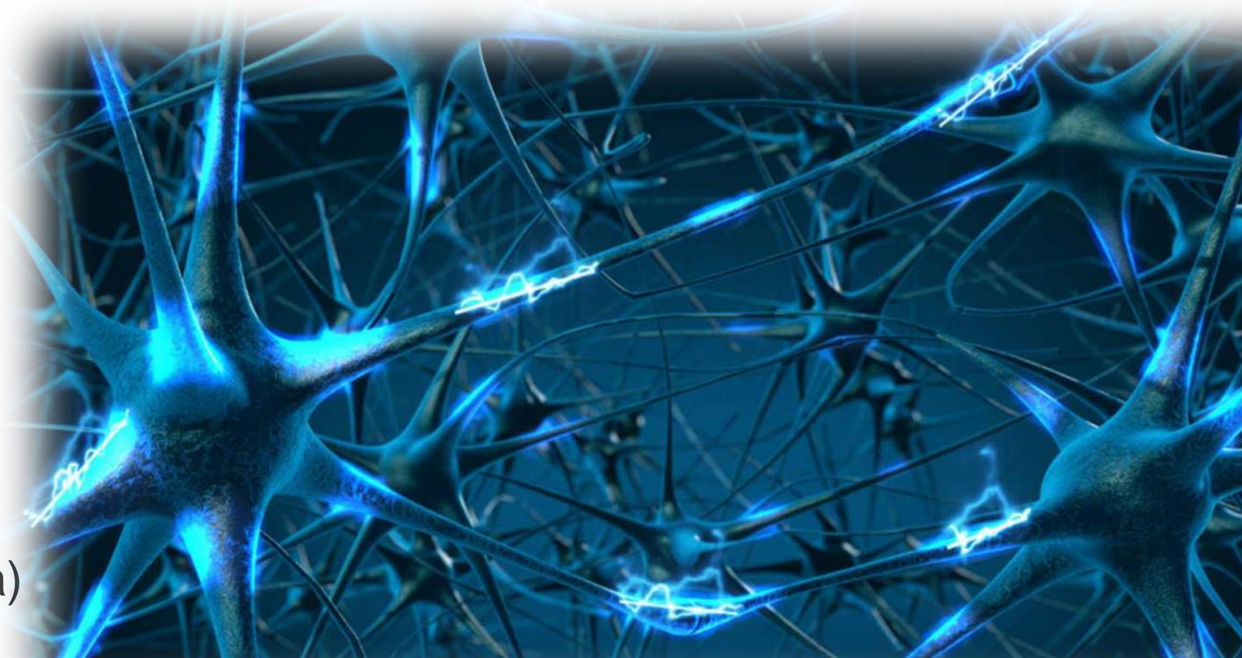
INTELLIGENZA ARTIFICIALE

PROBLEM-SOLVING & RICERCA DELLE SOLUZIONI (*)

Corsi di Laurea in Informatica, Ing. Gestionale, Ing. Informatica, Ing. di Internet
(a.a. 2022-2023)

Roberto Basili

(*) alcune *slides* sono di
Maria Simi (Univ. Pisa)



Overview

- Risoluzione di problemi e ricerca:
- Esemi di problemi e loro mappatura in problemi di ricerca
- Formalizzazione di un problema di ricerca
  - Obbiettivi, Stati, Azioni, Transizioni e Costo
- Algoritmi di Ricerca (AdR)
  - Strutture Dati coinvolte
- Valutazione Algoritmi di Ricerca
- Algoritmi non informati
  - Breadth-first, Depth-first, Uniform Cost
  - Profondità Limitata, Approfondimento Iterativo

Agenti *risolutori di problemi*

- Adottano il paradigma della risoluzione di problemi come ricerca in uno spazio di stati (*problem solving*).
- Sono agenti con modello che adottano una rappresentazione atomica dello stato
- Sono particolari agenti con obiettivo, che pianificano l'intera sequenza di mosse prima di agire

Il processo di risoluzione

- Passi che l'agente segue:
  1. Determinazione obiettivo (un insieme di stati)
  2. Formulazione del problema
     - rappresentazione degli stati
     - rappresentazione delle azioni
  3. Determinazione della soluzione mediante ricerca (definizione/costruzione di un piano)
  4. Esecuzione del piano

Che tipo di assunzioni?

- In genere, l’ambiente è **statico**
- … e **osservabile**
- …. **discreto**
  - L’insieme di azioni possibili è un numero finito
- e **deterministico**
  - Si assume che l’agente possa eseguire il piano “ad occhi chiusi”. Niente può andare storto.

Formulazione del problema

Un problema può essere definito formalmente mediante cinque componenti:

1. Stato iniziale
2. Azioni possibili in *s*: AZIONI(*s*)
3. Modello di transizione:
   Risultato: *stato* × *azione* → *stato*
   RISULTATO*(s, a)* = *s'*, uno stato successore

1, 2 e 3 definiscono implicitamente lo *spazio degli stati*

Formulazione del problema (cnt.)

4. Test obiettivo:
   - Un insieme di stati obiettivo
   - GOAL-TEST: *stato* $\rightarrow$ {*true*, *false*}

5. Costo del cammino
   - somma dei costi delle azioni (costo dei passi)
   - costo di passo: $c(s, a, s')$

   - Il costo di un'azione/passo non è mai negativo

$$c(s, a, s') > 0$$

Algoritmi di ricerca

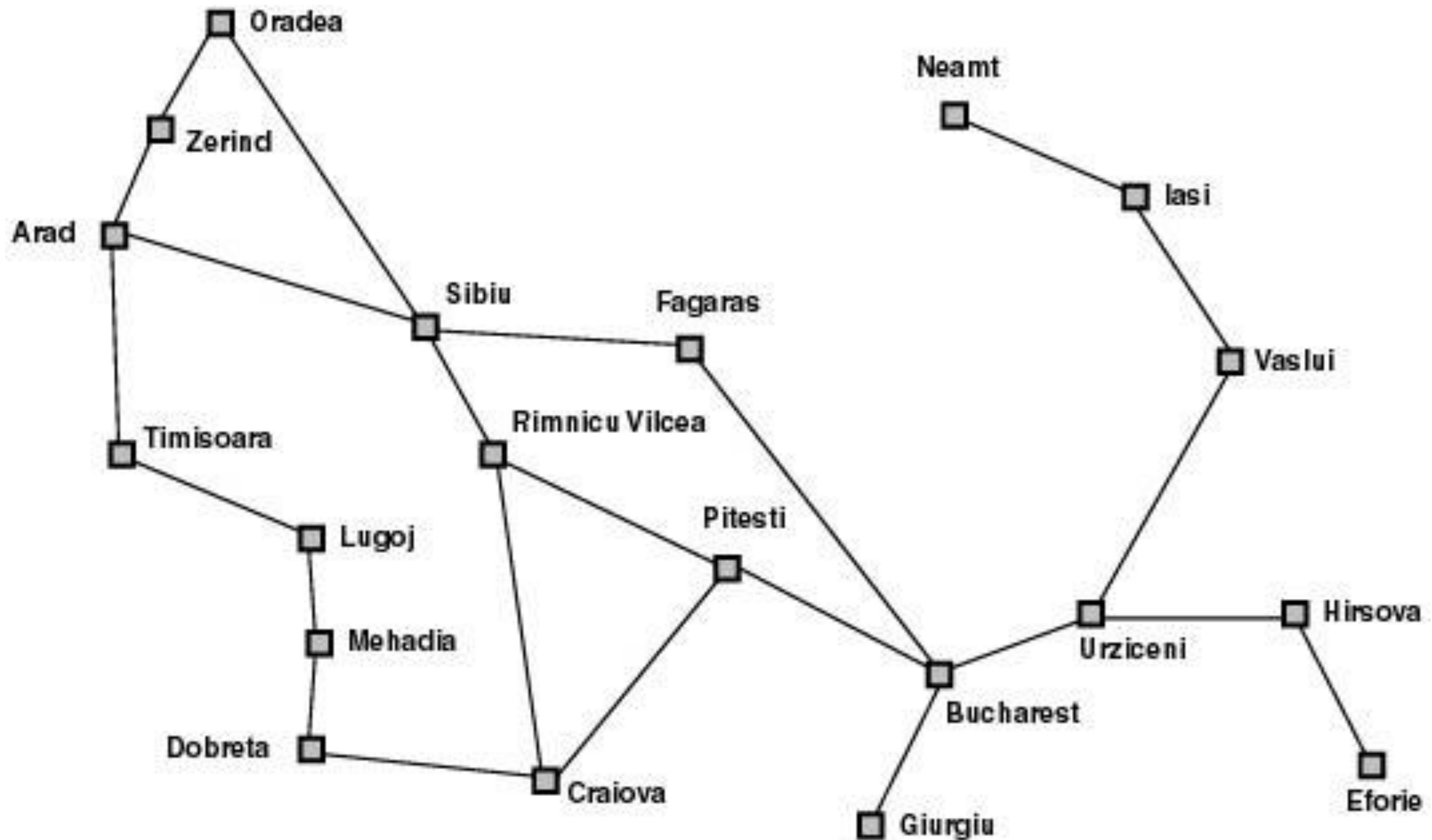
Gli algoritmi di ricerca prendono in input un problema e restituiscono un cammino soluzione, i.e. un cammino che porta dallo stato iniziale a uno stato goal

- *Misura delle prestazioni*

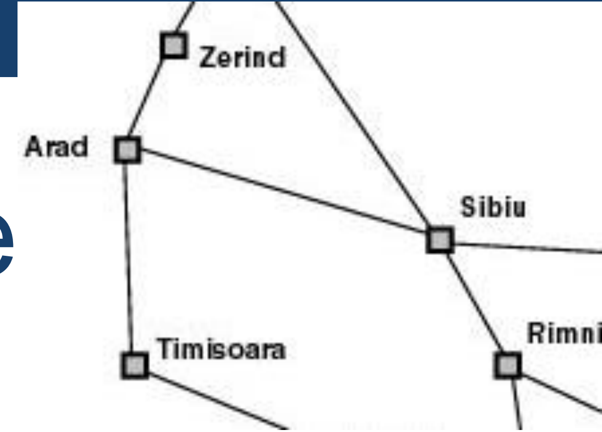
  Trova una soluzione? Quanto costa trovarla? Quanto efficiente è la soluzione?

  Costo totale = costo della ricerca + costo del cammino soluzione

Itinerario: il problema



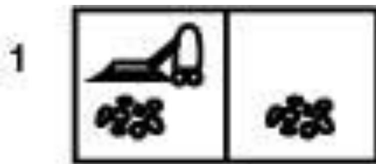
Itinerario: la formulazione

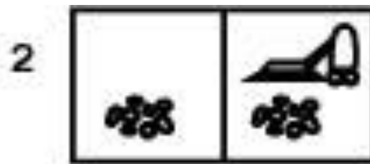


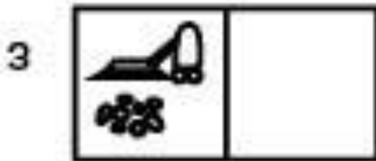
- *Stati*: le città. Es. *In*(Pitesti)
- *Stato iniziale*: la città da cui si parte. *In*(Arad)
- *Azioni*: spostarsi su una città vicina collegata
  - Azioni(*In*(Arad)) ={*Go*(Sibiu), *Go*(Zerind), *Go*(Timisoara)}
- Modello di transizione
  - Risultato(*In*(Arad), *Go*(Sibiu)) = *In*(Sibiu)
- *Costo del cammino*: somma delle lunghezze delle strade
- Lo spazio degli stati coincide con la rete di collegamenti tra città

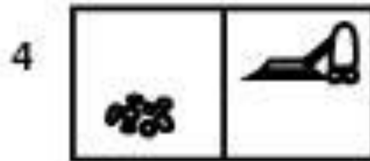
Aspirapolvere: il problema

Versione semplice: solo due locazioni, sporche o pulite, l'agente può essere in una delle due

1 

2 

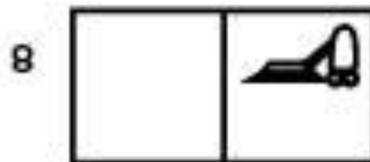
3 

4 

5

6

7

8 

Percezioni:

Sporco

NonSporco

Azioni:

Sinistra (L)

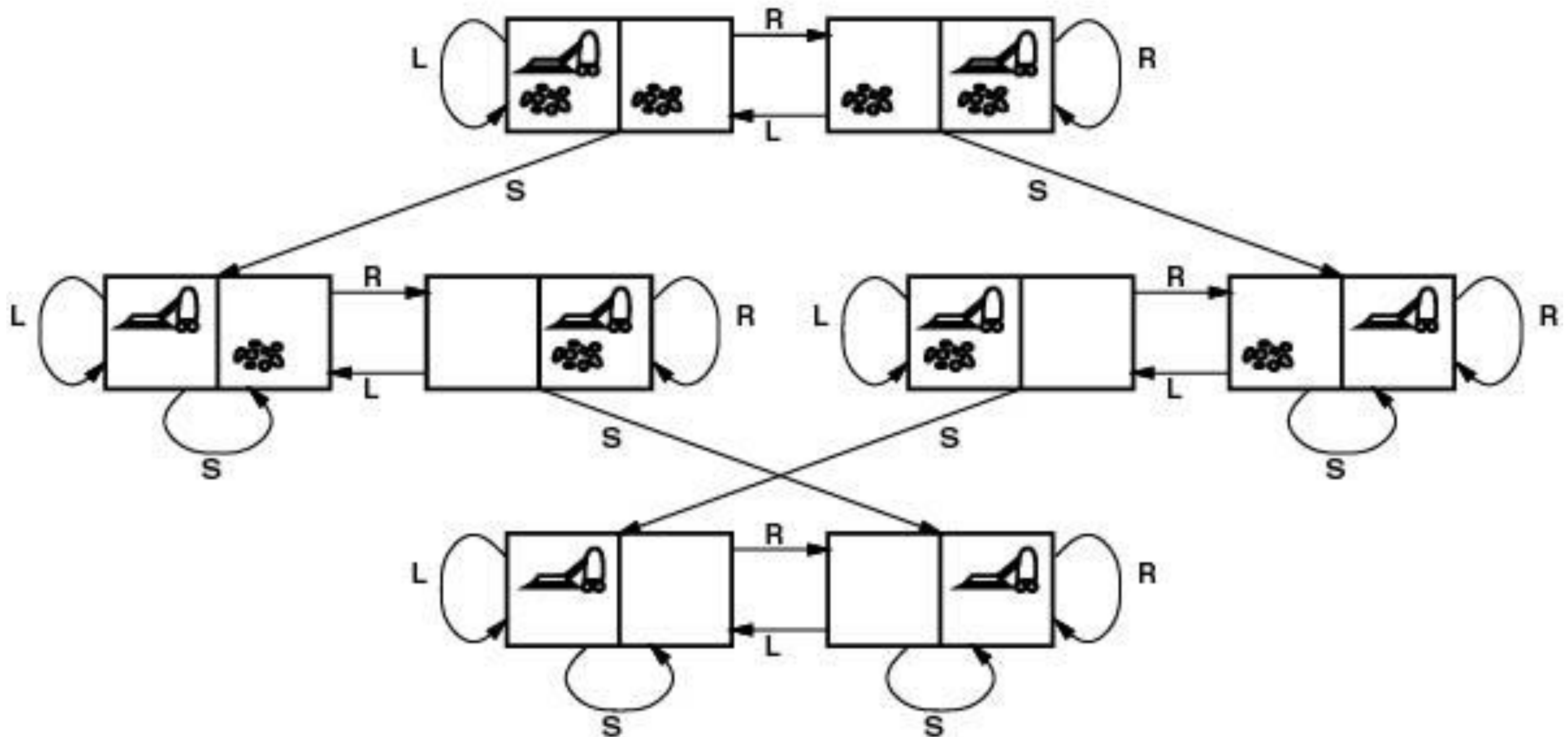
Destra (R)

Aspira (S)

Aspirapolvere: formulazione

- *Obiettivo*: rimuovere tutto lo sporco { 7, 8 }
- Ogni azione ha costo 1

Spazio degli stati :



Il puzzle dell'otto

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

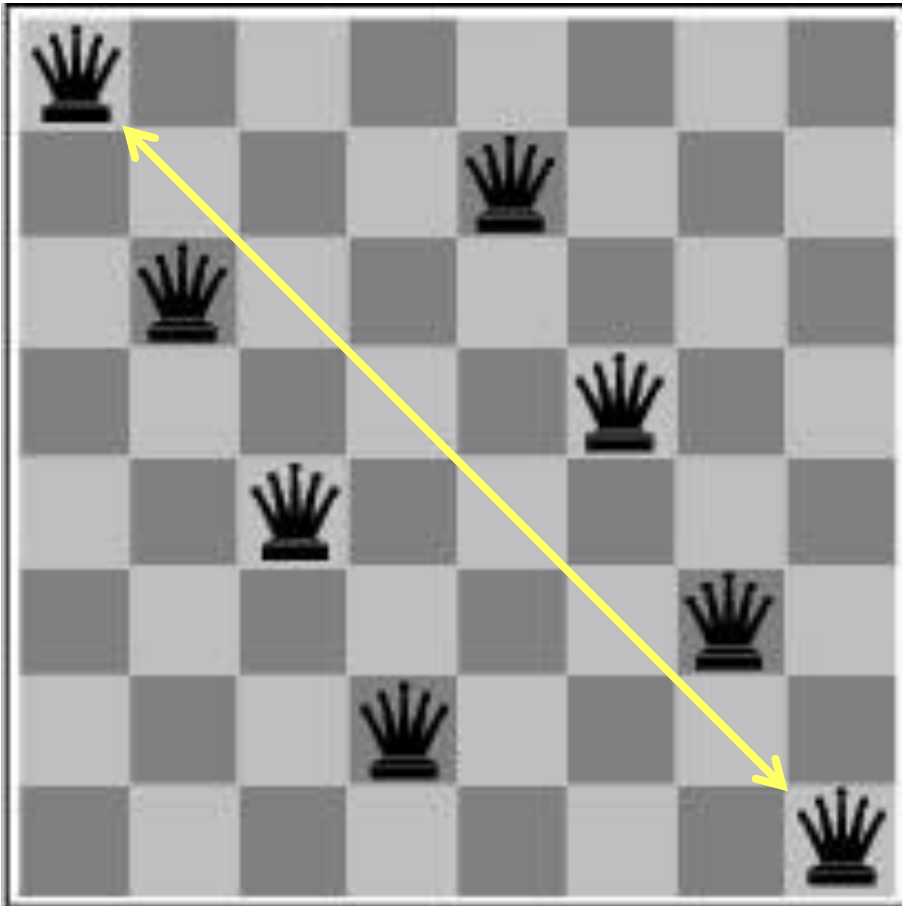
Puzzle dell'otto: formulazione

- *Stati*: possibili configurazioni della scacchiera
- *Stato iniziale*: una configurazione
- *Obiettivo*: una configurazione
  Goal-Test: Stato obiettivo?
- *Azioni*: mosse della casella bianca
  in sù: ↑ in giù: ↓
  a destra: → a sinistra: ←
- *Costo cammino*: ogni passo costa 1

1	2	3
8		4
7	6	5

- Lo spazio degli stati è un grafo con possibili cicli.

Le otto regine: il problema



Collocare 8 regine sulla scacchiera in modo tale che nessuna regina sia attaccata da altre

Le otto regine:
Formulazione incrementale 1

- *Stati*: scacchiere con 0-8 regine
- *Goal-Test*: 8 regine sulla scacchiera, nessuna sotto scacco
- *Costo cammino*: zero *(perché?)*
- *Azioni*: aggiungi una regina

$$64 \times 63 \times \ldots \times 57 \sim 1{,}8 \times 10^{14}$$

sequenze da considerare!

Le otto regine:
Formulazione incrementale 2

- *Stati*: scacchiere con 0-8 regine, nessuna minacciata
- *Goal-Test*: 8 regine sulla scacchiera, nessuna minacciata
- *Costo cammino*: zero
- *Azioni*: aggiungi una regina nella colonna vuota più a destra ancora libera in modo che non sia sotto scacco

solo 2057 sequenze da considerare

Le 8 regine:
Formulazione a stato completo

- *Goal-Test*: 8 regine sulla scacchiera, nessuna minacciata
- *Costo cammino*: zero
- *Stati*: scacchiere con 8 regine, una per colonna
- *Azioni*: sposta una regina nella colonna, se minacciata

Dimostrazione di teoremi: il problema

- Dato un insieme di formule logiche (premesse)

$$\{s,\ t,\ q \Rightarrow p,\ r \Rightarrow p,\ v \Rightarrow q,\ t \Rightarrow r,\ s \Rightarrow v\}$$

dimostrare una proposizione p

- Nel calcolo proposizionale basta un'unica regola di inferenza, il *Modus Ponens (MP):*

Se p e $p \Rightarrow q$ allora q

Dim. teoremi: formulazione

- *Stati*: insiemi di formule proposizionali
- *Stato iniziale*: un certo insieme di formule proposizionali (le premesse).
- *Stato obiettivo*: l'insieme di formule proposizionali che contiene il teorema da dimostrare. *Es p.*
- *Operatori:* l'applicazione del MP, che aggiunge teoremi

continua

Dimostrazione dei teoremi: spazio degli stati

$\{s, t, q \Rightarrow p, r \Rightarrow p, v \Rightarrow q, t \Rightarrow r, s \Rightarrow v\}$

$\{s, t, q \Rightarrow p, r \Rightarrow p, v \Rightarrow q, t \Rightarrow r, s \Rightarrow v, v\}$

$\{s, t, q \Rightarrow p, r \Rightarrow p, v \Rightarrow q, t \Rightarrow r, s \Rightarrow v, r\}$

$\{s, t, q \Rightarrow p, r \Rightarrow p, v \Rightarrow q, t \Rightarrow r, s \Rightarrow v, v, q\}$

$\{s, t, q \Rightarrow p, r \Rightarrow p, v \Rightarrow q, t \Rightarrow r, s \Rightarrow v, r, p\}$

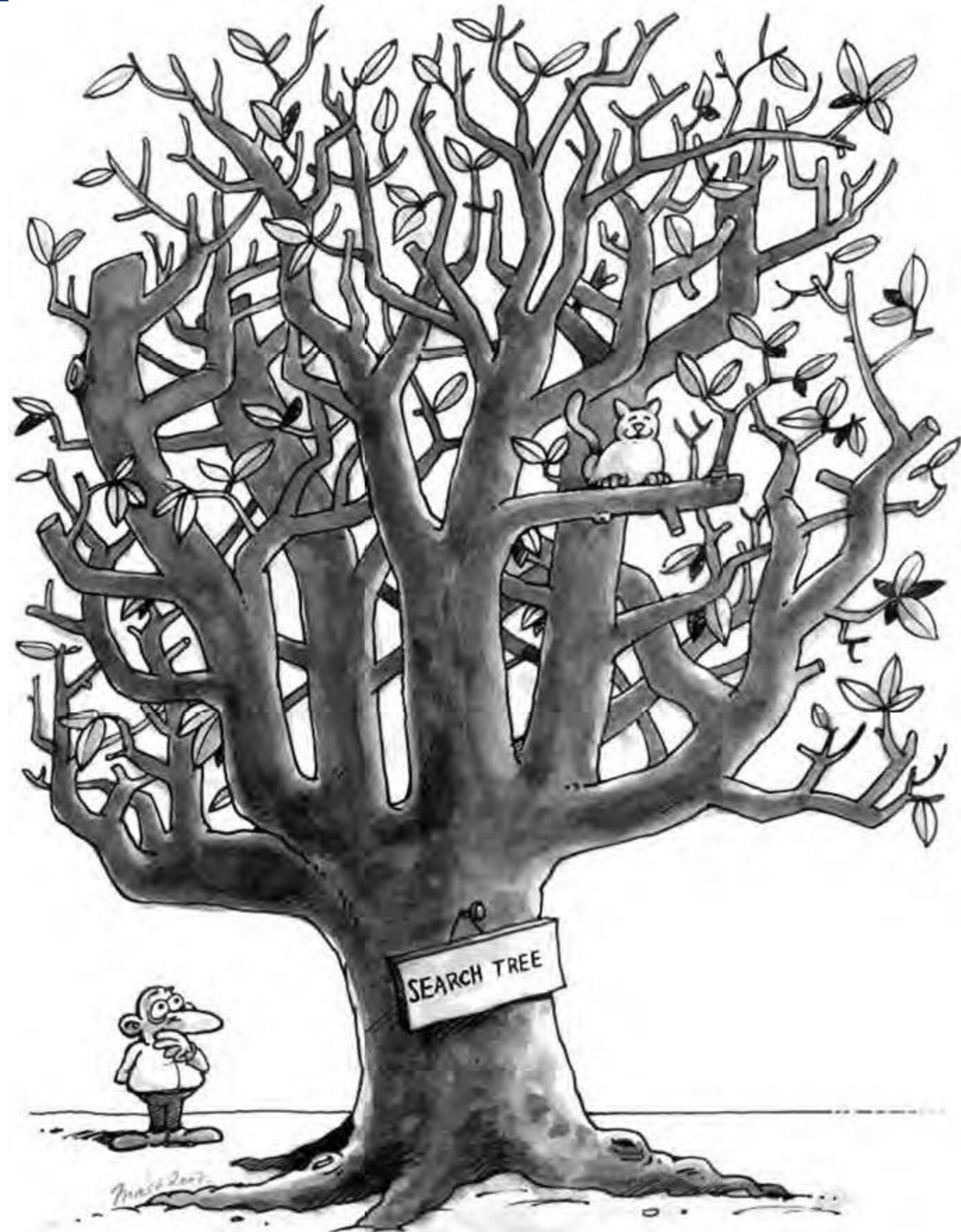
$\{s, t, q \Rightarrow p, r \Rightarrow p, v \Rightarrow q, t \Rightarrow r, s \Rightarrow v, v, q, p\}$

Problemi applicativi: esempi

- Pianificazione di viaggi aerei
- Problema del commesso viaggiatore
- Configurazione VLSI
- Navigazione di robot
- Progettazione di proteine
- Ranking delle risposte di un search engine
- Il riconoscimento delle categorie grammaticali delle parole in un testo
- La classificazione tematica di tweet o documenti

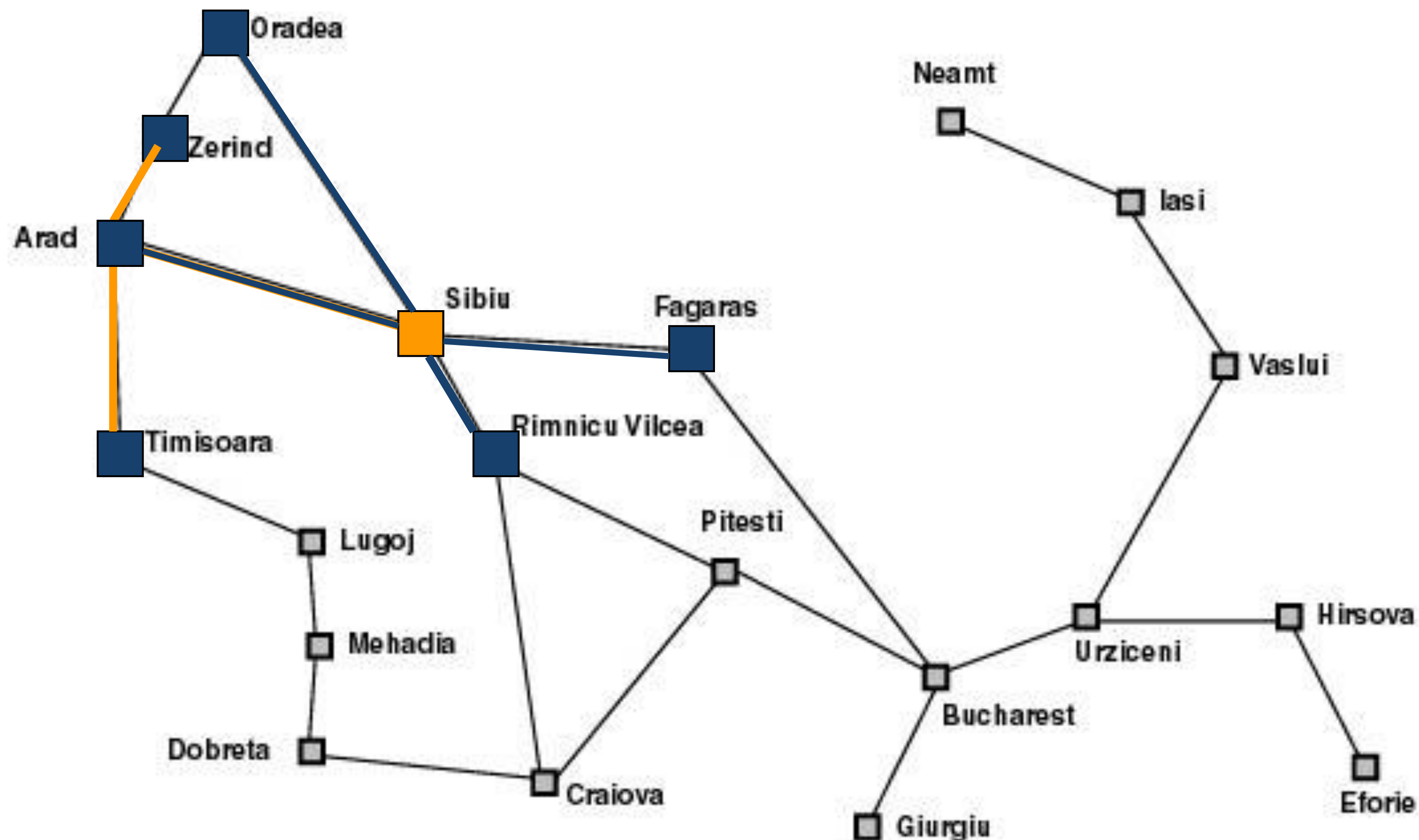
L’albero di ricerca

- *Dov’e il mio gatto?*



Ricerca della soluzione

Generazione di un albero di ricerca sovrapposto allo spazio degli stati



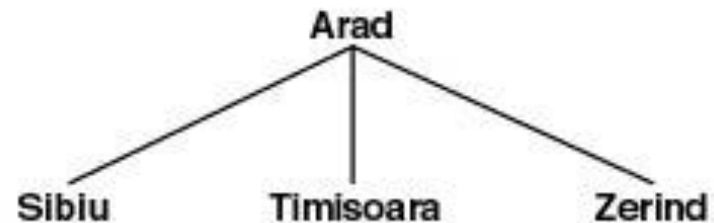
Ricerca della soluzione

Generazione di un albero di ricerca sovrapposto allo spazio degli stati

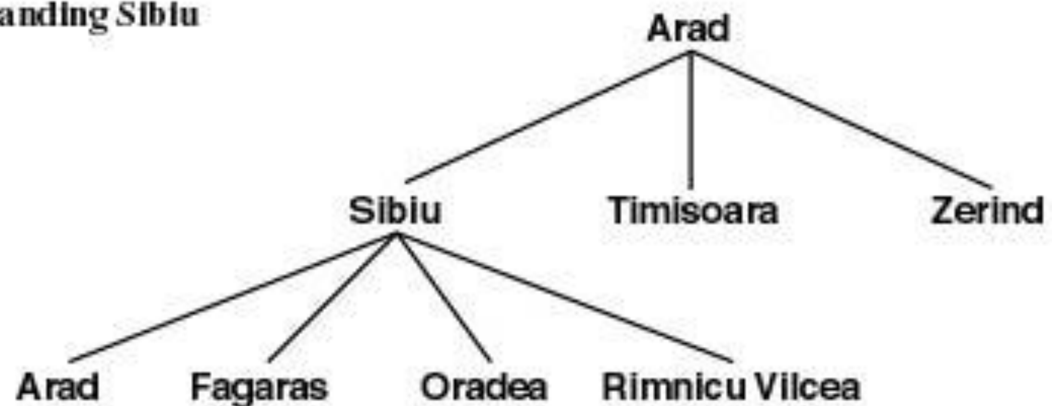
(a) The initial state

Arad

(b) After expanding Arad



(c) After expanding Sibiu



Ricerca ad albero

function Ricerca-Albero (*problema)*
  returns soluzione oppure **fallimento**
 Inizializza la frontiera con stato iniziale del problema
 loop do
  if *la frontiera è vuota* **then return** **fallimento**
  Scegli un nodo foglia da espandere e rimuovilo dalla frontiera
  if *il nodo contiene uno stato obiettivo*
   then return *la soluzione corrispondente*
  E*spandi il nodo e aggiungi i successori alla frontiera*

I nodi dell'albero di ricerca

Un nodo *n* è una struttura dati con quattro componenti:

- Uno stato: *n*.stato
- Il nodo padre: *n*.padre
- L'azione effettuata per generarlo: *n*.azione

- Il costo del cammino dal nodo iniziale al nodo: *n*.costo-cammino indicata come g(*n*)

Struttura dati

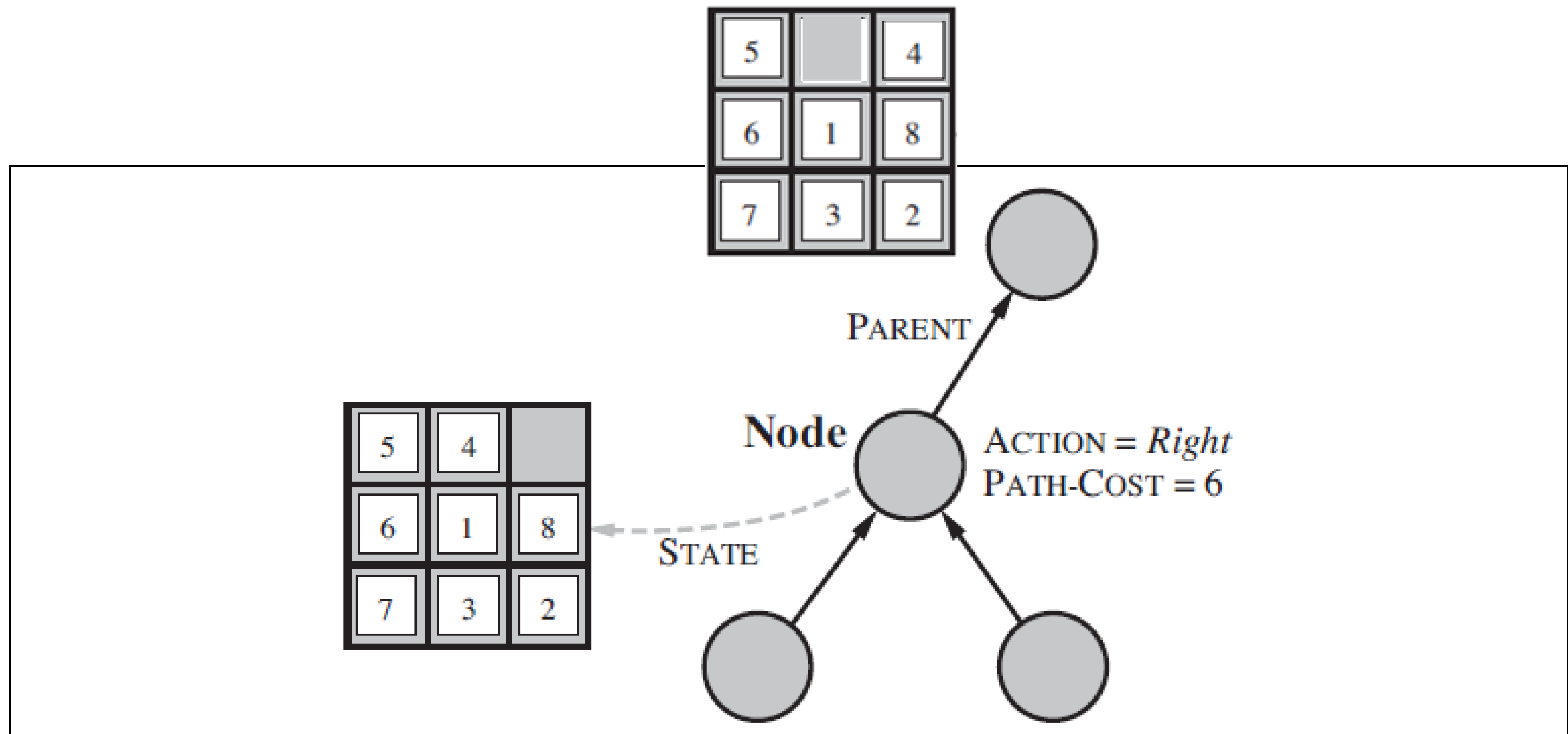


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Struttura dati per la frontiera

- *Frontiera*: lista dei nodi in attesa di essere espansi (le foglie dell'albero di ricerca).
- La frontiera è implementata come una *coda* (cioè una lista) con operazioni:
  - Vuota?(*coda*)
  - POP(coda) estrae il primo elemento
  - Inserisci(elemento, coda)
- Diversi tipi di coda hanno diverse funzioni di inserimento e implementano strategie diverse

Diversi tipi di strategie

- FIFO- First In First Out
  - Viene estratto l'elemento più vecchio (in attesa da più tempo); in nuovi nodi sono aggiunti alla fine.
- LIFO-Last In First Out
  - Viene estratto il più recentemente inserito; i nuovi nodi sono inseriti all'inizio. In genere questi tipi di lista prendono il nome di *pile*.
- Coda con priorità
  - Viene estratto quello con priorità più alta in base a una funzione di ordinamento; dopo l'inserimento dei nuovi nodi si riordina.

Strategie non informate

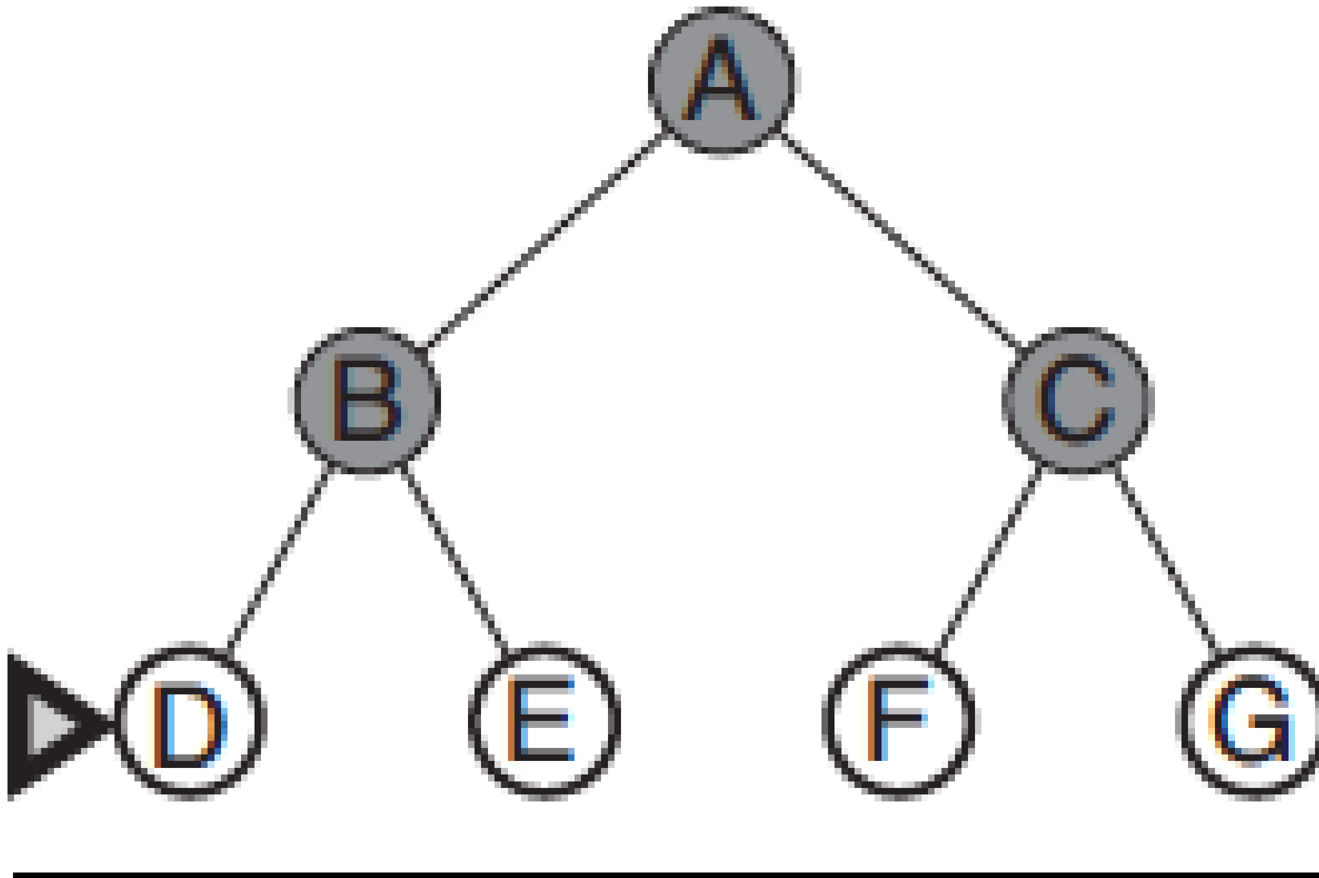
- Ricerca in ampiezza
- Ricerca di costo uniforme
- Ricerca in profondità
- Ricerca in profondità limitata
- Ricerca con approfondimento iterativo

vs strategie di *ricerca euristica* (o informata): fanno uso di informazioni riguardo alla distanza stimata dalla soluzione

Valutazione di una strategia

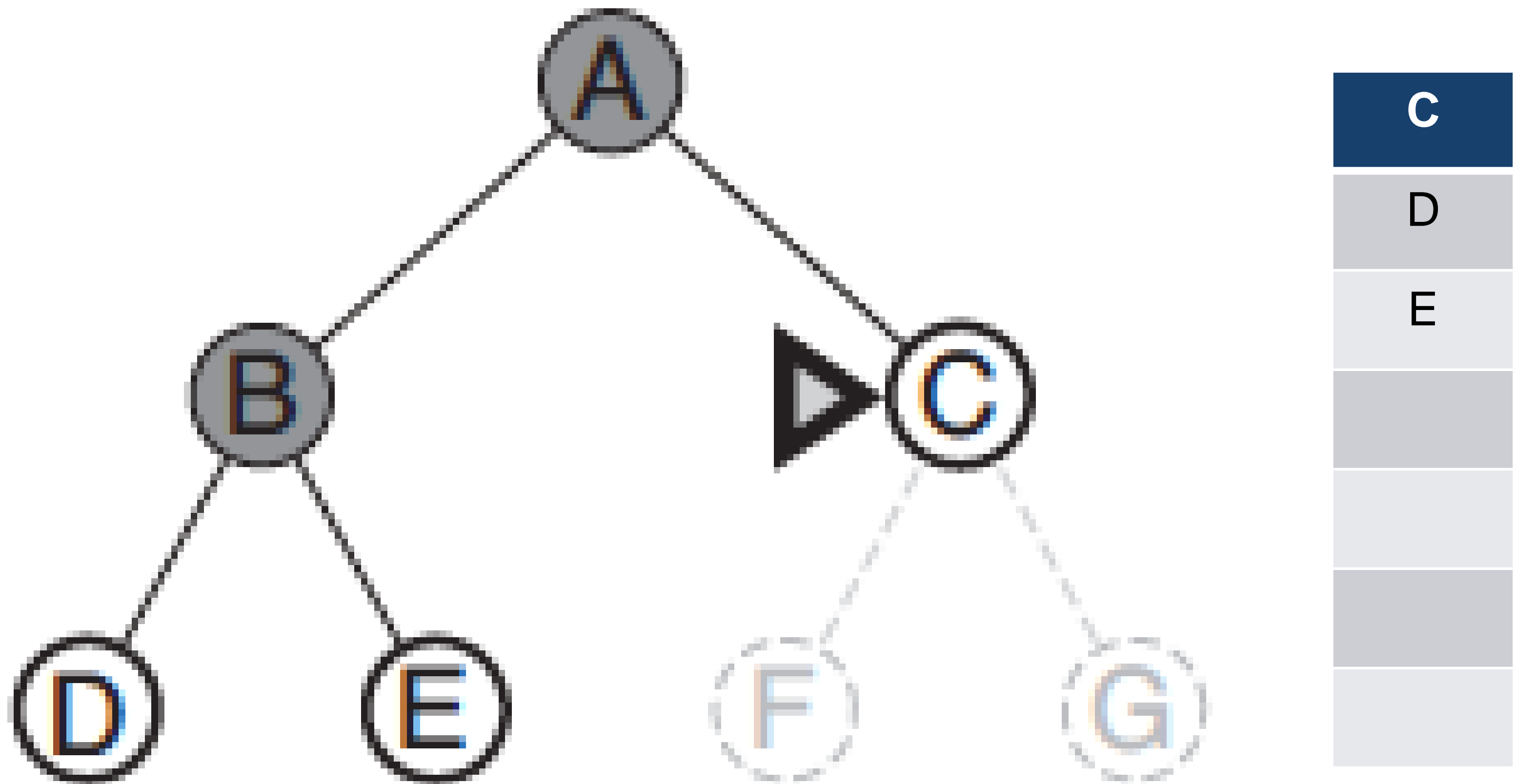
- *Completezza*: se la soluzione esiste l'Algoritmo di Ricerca riesce a trovarla
- *Ottimalità* (ammissibilità): l'AdR trova la soluzione migliore, quella cioè con costo minore
- *Complessità nel tempo*: tempo richiesto affinche l'AdR trovi la soluzione
- *Complessità nello spazio*: a quanto ammonta la memoria richiesta dall'AdR

Ricerca in ampiezza (BF)



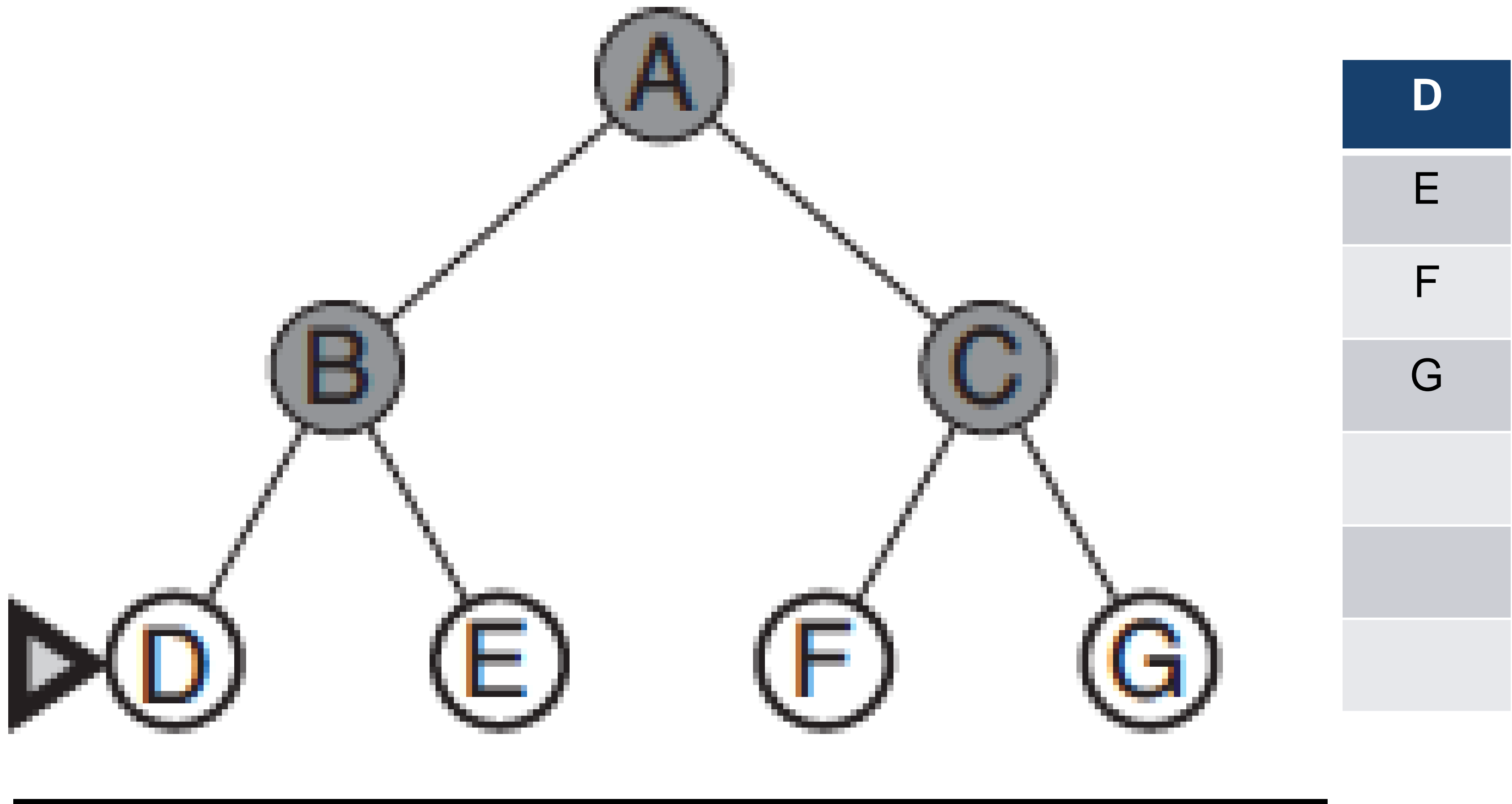
Implementata con una coda che inserisce alla fine (FIFO)

Ricerca in ampiezza (BF)



Implementata con una coda che inserisce alla fine (FIFO)

Ricerca in ampiezza (BF)



Implementata con una coda che inserisce alla fine (FIFO)

Ricerca in ampiezza (su albero)

function Ricerca-Ampiezza-A (*problema)*
returns soluzione oppure **fallimento**
nodo = un nodo con *stato il problema.stato-iniziale e costo-di-cammino=0*
if *problema.Test-Obiettivo(nodo.*Stato*)* **then return** Soluzione(*nodo*)
frontiera = una coda FIFO con *nodo* come unico elemento
loop do
if IsEmpty(*frontiera*) **then return fallimento**
nodo = POP(*frontiera*)
for each azione **in** problema.Azioni(nodo.Stato) **do**
figlio = Nodo-Figlio(*problema*, *nodo, azione*)
if Problema.TestObiettivo(figlio.Stato) **then return** Soluzione(*figlio*)
frontiera = InCoda(*figlio*, *frontiera*) /* frontiera come coda FIFO
end

Analisi complessità spazio-temporale

- Assumiamo
  b = fattore di *ramificazione* (*branching*)
  (numero max di successori)
  d = profondità del nodo obiettivo più superficiale
  m = lunghezza massima dei cammini nello spazio di ricerca

Ricerca in ampiezza: analisi

- Strategia *completa*
- Strategia *ottimale* se gli operatori hanno tutti lo stesso costo *k*, cioè $g(n) = k \cdot \text{depth}(n)$, dove $g(n)$ è il costo del cammino per arrivare a n

- Complessità nel tempo (nodi generati)

$$T(b, d) = b + b^2 + \ldots + b^d \rightarrow O(b^d)$$

- Complessità spazio (nodi in memoria): $O(b^d)$

Nota: O*(.)* notazione per la complessità asintotica

Ricerca in ampiezza: esempio

- *Esempio*: *b*=10; 1 milione nodi al sec generati; 1 nodo occupa 1000 byte

Profondità	Nodi	Tempo	Memoria
2	110	0,11 ms	107 kilobyte
4	11.100	11 ms	10,6 megabyte
6	10^6	1.1 sec	1 gigabyte
8	10^8	2 min	103 gigabyte
10	10^{10}	3 ore	10 terabyte
12	10^{12}	13 giorni	1 petabyte
14	10^{14}	3,5 anni	10 esabyte

Come fare meglio

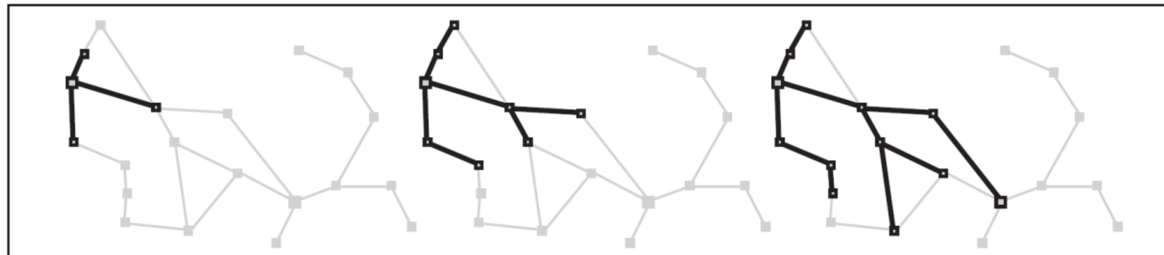


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

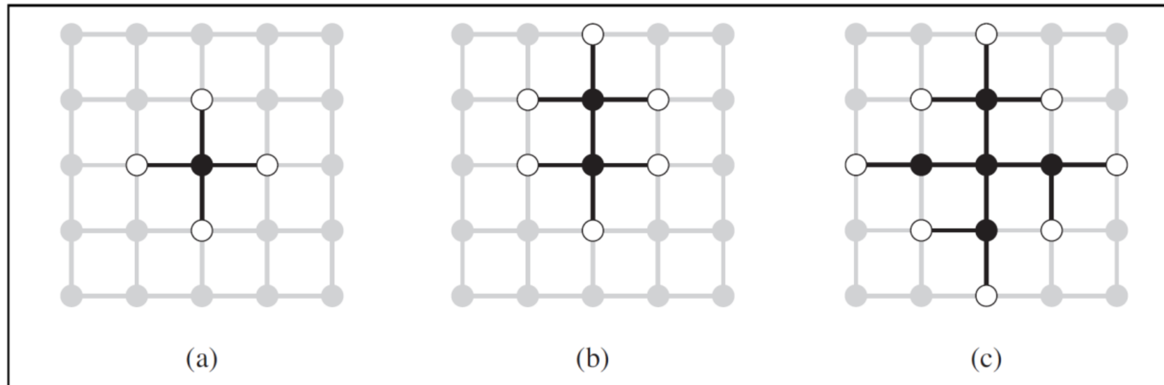
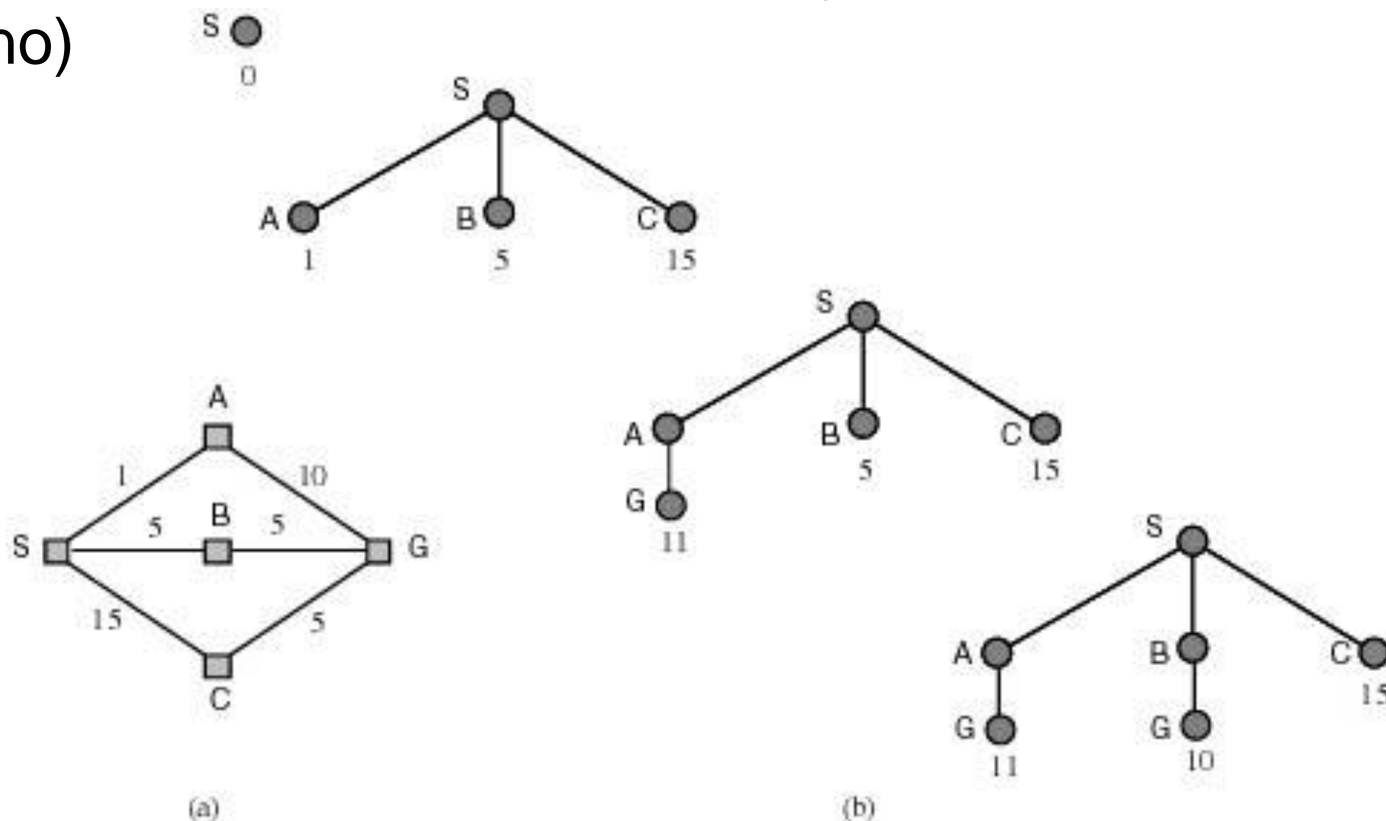


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

Ricerca di costo uniforme (UC)

Generalizzazione della ricerca in ampiezza: *si sceglie il nodo di costo minore sulla frontiera* (si intende il costo del cammino)



Implementata da una coda ordinata per costo crescente (in cima i nodi di costo minore)

Ricerca UC (su albero)

```
function Ricerca-UC-A (problema)
   returns soluzione oppure fallimento
    nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
   frontiera = una coda con priorità con nodo come unico elemento
 loop do
   if IsEmpty(frontiera) then return fallimento
   nodo = Pop(frontiera)
   if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
   for each azione in problema.Azioni(nodo.Stato) do
          figlio = Nodo-Figlio(problema, nodo, azione)
          frontiera = Inserisci(figlio, frontiera)   /* in coda con priorità
end
```

Ricerca UC (modified)

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      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
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

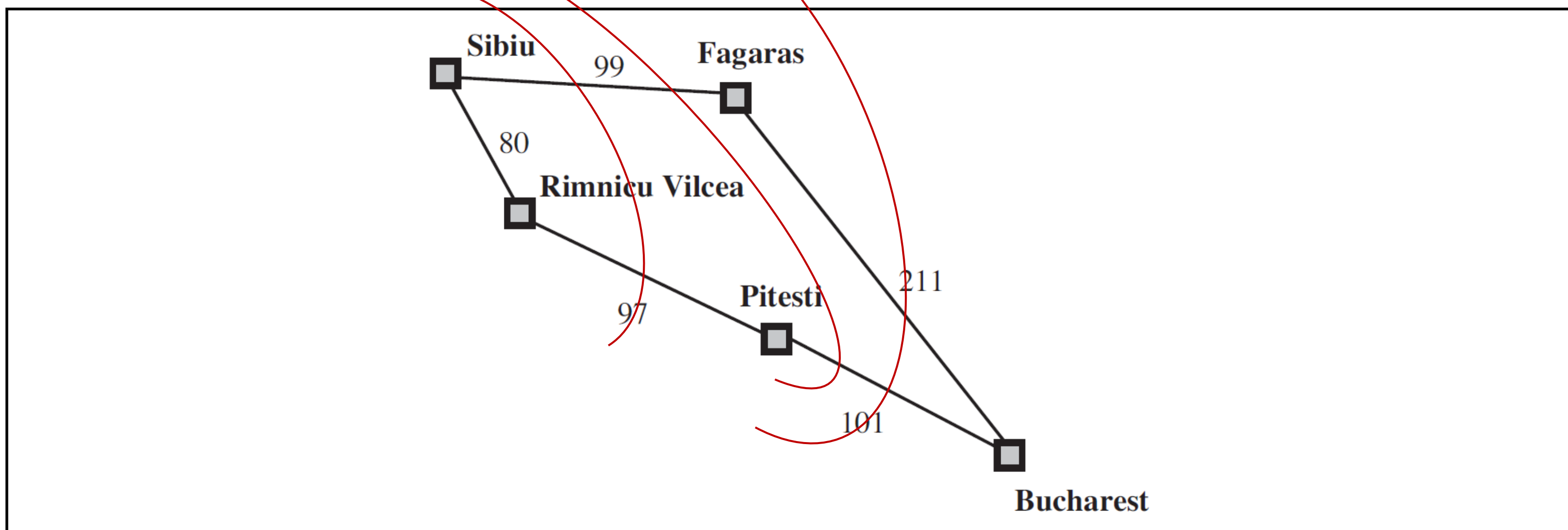
Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Costo uniforme: analisi

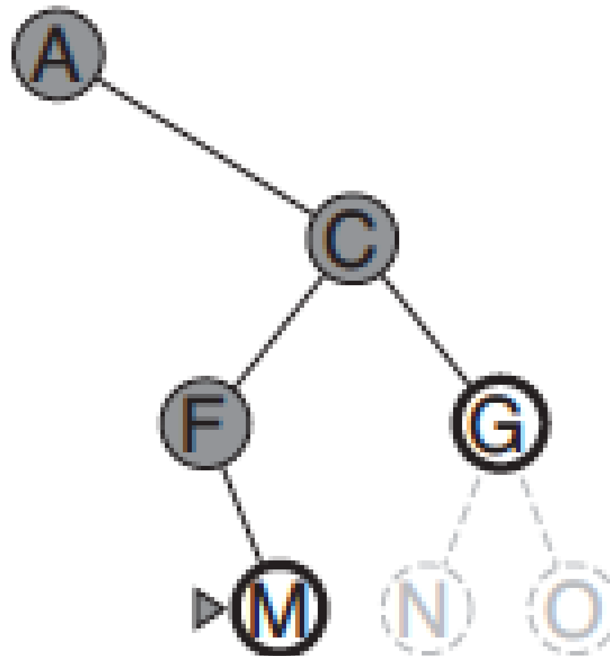
- *Ottimalità* e *completezza* garantite purché il costo degli archi sia maggiore di $\varepsilon>0$.
- C^* è il costo della soluzione ottima
- $\lfloor C^*/\varepsilon \rfloor$ è il numero di mosse nel caso peggiore, arrotondato per difetto
- Complessità: $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
- *Nota*: quando ogni azione ha lo stesso costo UC somiglia a BF ma complessità $O(b^{1+d})$

Ricerca UC: ottimalità

- Goal: Sibiu to Bucarest
- A patto che le transizioni singole abbiano costo non nullo, la scelta UC del cammino con l lunghezza minima implica che il primo cammino trovato dall'algoritmo non può non essere ottimo

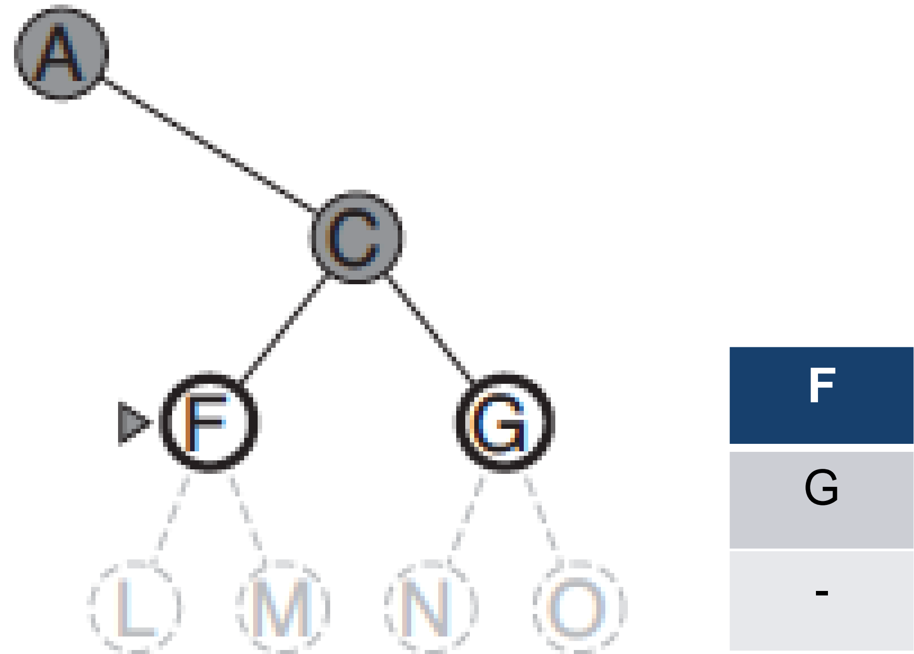


Ricerca in profondità



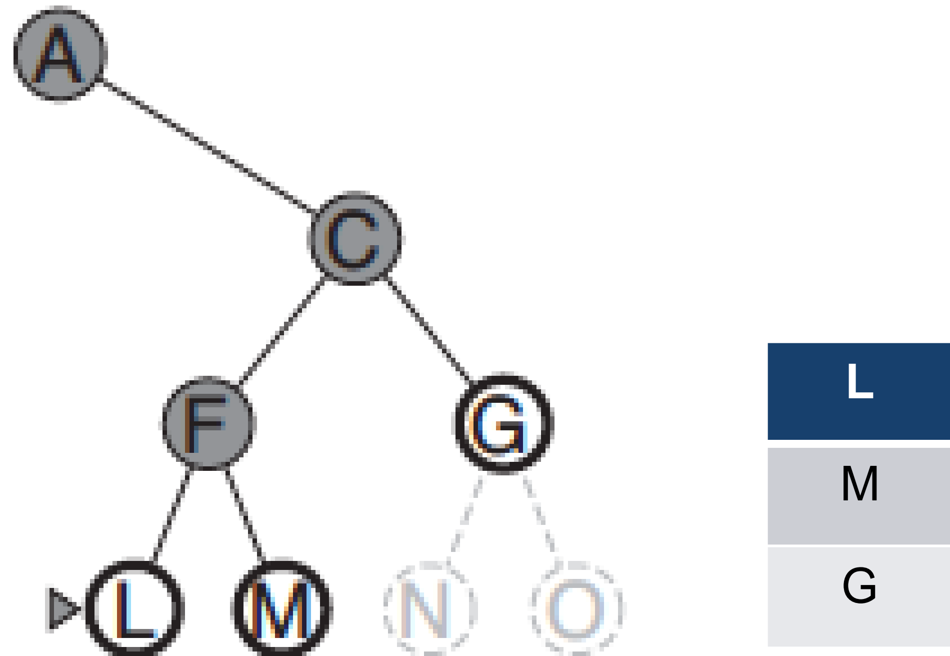
Implementata da una coda che mette i successori in testa alla lista (LIFO, pila o stack).

Ricerca in profondità



Implementata da una coda che mette i successori in testa alla lista (LIFO, pila o stack).

Ricerca in profondità



Implementata da una coda che mette i successori in testa alla lista (LIFO, pila o stack).

Ricerca in profondità: analisi

- Se *m* distanza massima della soluzione nello spazio di ricerca
- *b* fattore di diramazione

- Allora la complessità temporale è: $O(b^{m+1})$
- e la Occupazione di memoria è proporzionale a: $bm+1$

- Strategia non completa e non ottimale.

- Drastico risparmio in memoria:
  - (ampiezza) BF d=16 10 esabyte (10^{14} byte)
  - (profondità) DF d=16 156 Kbyte (10^{2} byte)

Ricerca in profondità ricorsiva

- Ancora più efficiente in occupazione di memoria perché mantiene solo il cammino corrente (solo m nodi nel caso pessimo)
- Realizzata da un algoritmo ricorsivo “con backtracking” che non necessita di tenere in memoria b nodi per ogni livello, ma salva lo stato su uno stack a cui torna in caso di fallimento per fare altri tentativi.

Ricerca in profondità (su albero)

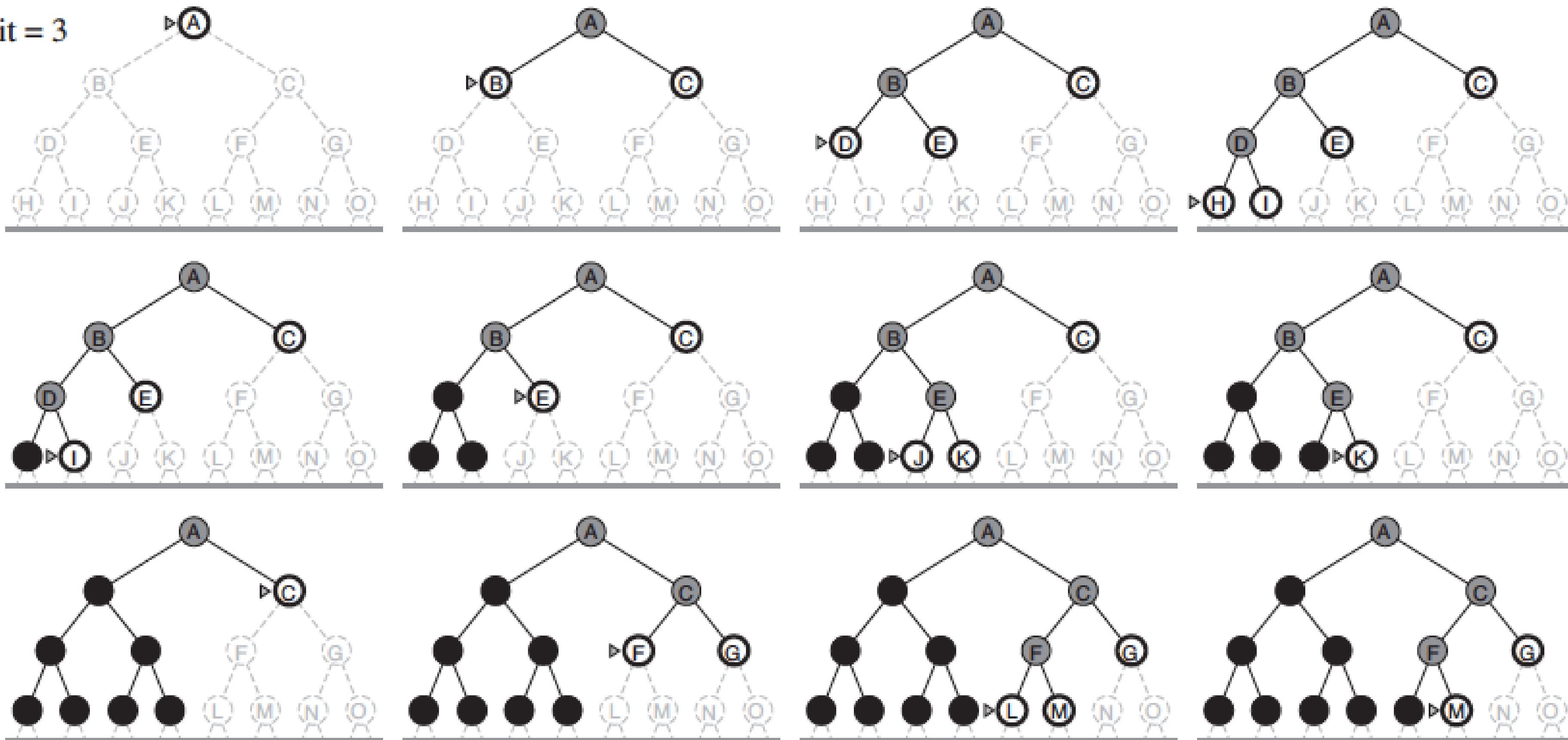
```
function Ricerca-DF-A (problema)
    returns soluzione oppure fallimento
    return Ricerca-DF-ricorsiva( CREANODO(problema.Stato-iniziale), problema)

function Ricerca-DF-ricorsiva(nodo, problema)
    returns soluzione oppure fallimento
    if problema.TESTOBIETTIVO(nodo.Stato) then return Soluzione(nodo)
    else
    for each azione in problema.Azioni(nodo.Stato) do
            figlio = NODO-FIGLIO(problema, nodo, azione)
            risultato = Ricerca-DF-ricorsiva(figlio, problema)
            if  risultato ≠ fallimento  then return risultato
     return fallimento
```

Ricerca in profondità limitata (DL)

- Si va in profondità fino ad un certo livello predefinito da un valore limite l
- *Completa* per problemi in cui si conosce un limite superiore per la profondità della soluzione.

  Es. *Route-finding* limitata da: numero di città – 1
- Completo: se $d<l$
- Non è ottimale
- Complessità tempo: $O(b^l)$
- Complessità spazio: $O(bl)$

Approfondimento iterativo (ID)



ID: analisi

- Miglior compromesso tra BF e DF

  BF: $b+b^2+ \dots +b^{d-1}+b^d$ con b=10 e d=5

  10+100+1000+10.000+100.000=111.110

- ID: I nodi dell'ultimo livello generati una volta, quelli del penultimo 2, quelli del terzultimo 3 … quelli del primo d volte

  ID: $(d)b+(d-1)\, b^2+ \dots +3b^{d-2}+2b^{d-1}+1b^d$

  = 50+400+3000+20.000+100.000=123450

- Complessità tempo: $O(b^d)$ Spazio: $O(b.d)$

Direzione della ricerca

Un problema ortogonale alla strategia è la *direzione della ricerca*:

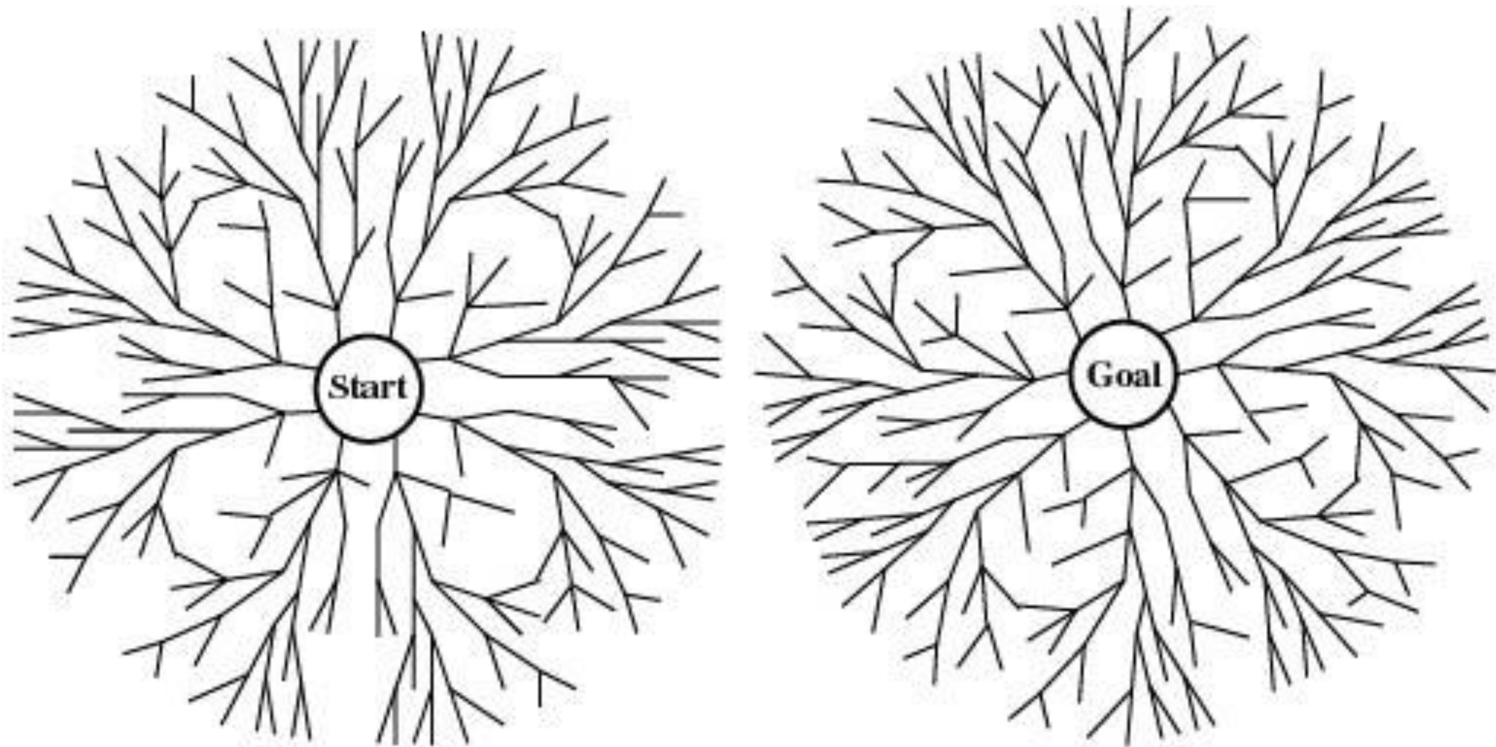
- ricerca *in avanti* o *guidata dai dati*: si esplora lo spazio di ricerca dallo stato iniziale allo stato obiettivo;
- ricerca *all'indietro* o *guidata dall'obiettivo*: si esplora lo spazio di ricerca a partire da uno stato goal e riconducendosi a sotto-goal fino a trovare uno stato iniziale.

Quale direzione?

- Conviene procedere nella direzione in cui il fattore di diramazione è minore
- Si preferisce ricerca all'indietro quando:
  - l'obiettivo e chiaramente definito (ad es. nel *theorem proving*) o si possono formulare una serie limitata di ipotesi;
  - i dati del problema non sono noti e la loro acquisizione può essere guidata dall'obiettivo
- Si preferisce ricerca in avanti quando:
  - gli obiettivi possibili sono molti (*design*)
  - abbiamo una serie di dati da cui partire

Ricerca bidirezionale

Si procede nelle due direzioni fino ad incontrarsi



Ricerca bidirezionale: analisi

- Complessità tempo: O($b^{d/2}$)
  (test intersezione in tempo costante, es. hash table)
- Complessità spazio: O($b^{d/2}$)
  (almeno tutti i nodi in una direzione in memoria, es usando BF)

NOTA: non sempre applicabile, es. predecessori non definiti, troppi stati obiettivo …

Confronto delle strategie (albero)

Criterio	BF Breadth-first	UC Uniform Cost	DF Depth-first	DL Limited Depth	ID Iterative Depth	Bidir Bidirectional
Completa?	si	si(^)	*no*	si (+)	*si*	si
Tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Spazio	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Ottimale?	si(*)	si(^)	no	no	si(*)	si

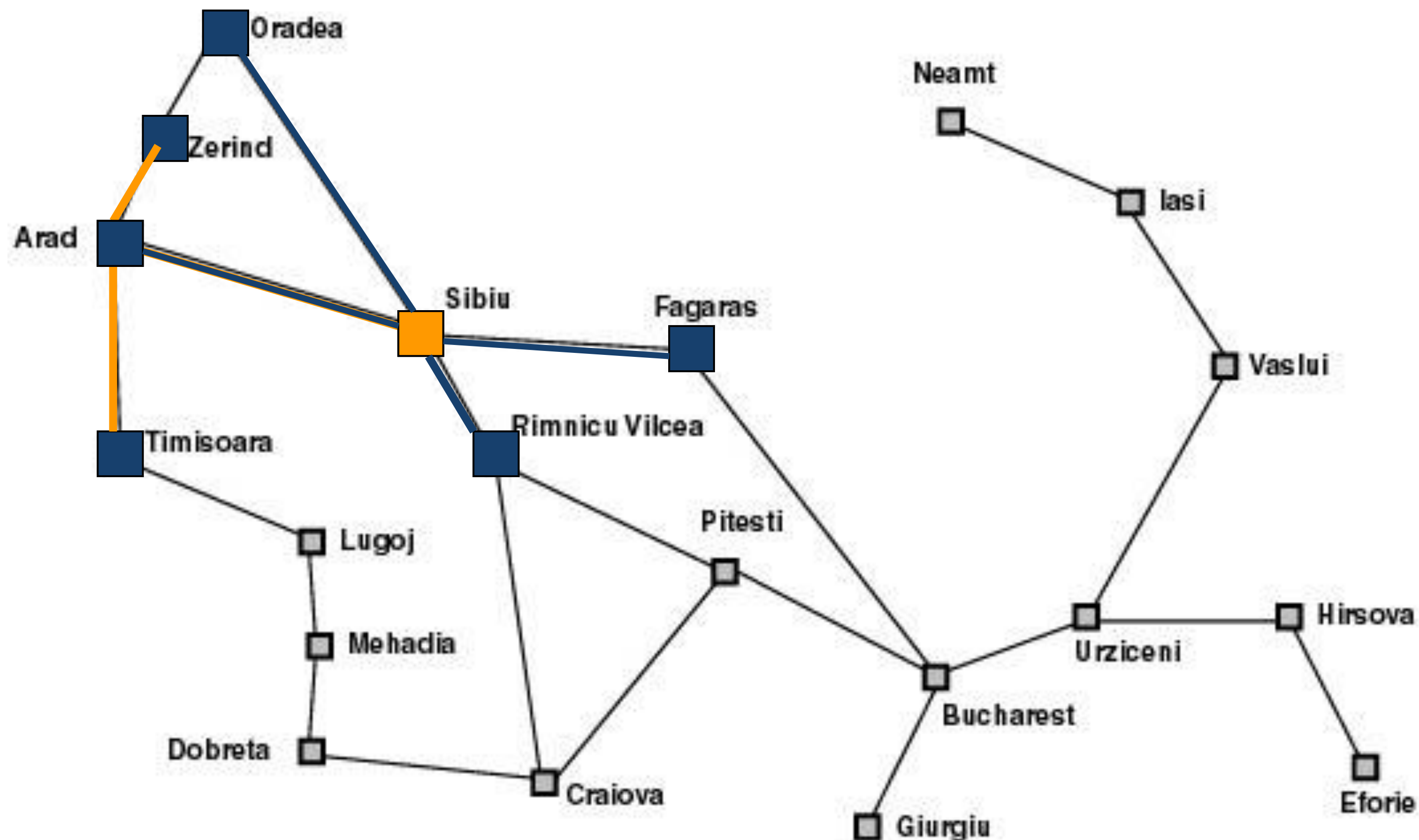
(*) se gli operatori hanno tutti lo stesso costo

(^) per costi degli archi $\geq \varepsilon > 0$

(+) per problemi per cui si conosce un limite alla profondità della soluzione (se l >d)

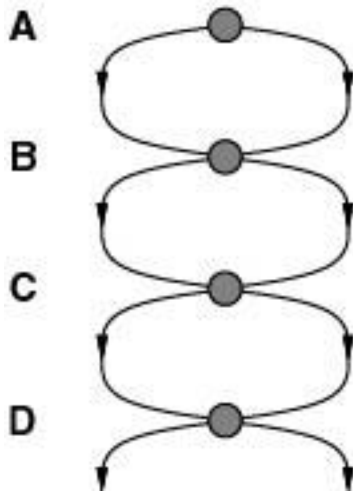
Ricerca su grafi: cammini ciclici

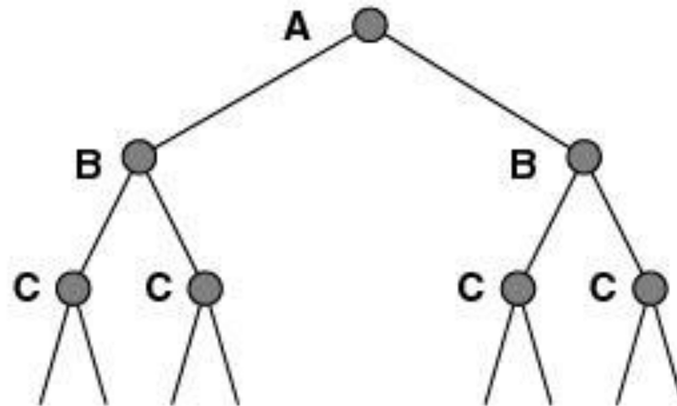
I cammini ciclici rendono gli alberi di ricerca infiniti



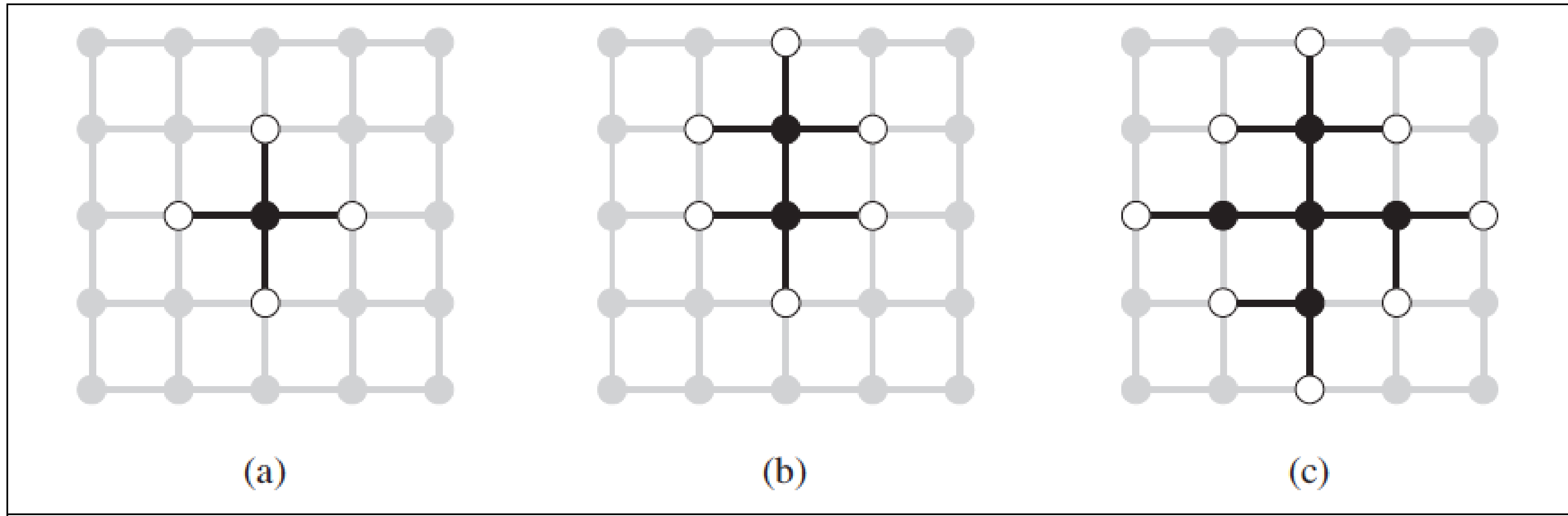
Ricerca su grafi: ridondanze

Su spazi di stati a grafo si generano più volte gli stessi nodi nella ricerca, anche in assenza di cicli.





Ridondanza nelle griglie



Visitare stati già visitati fa compiere lavoro inutile. Come evitarlo?

Come fare meglio

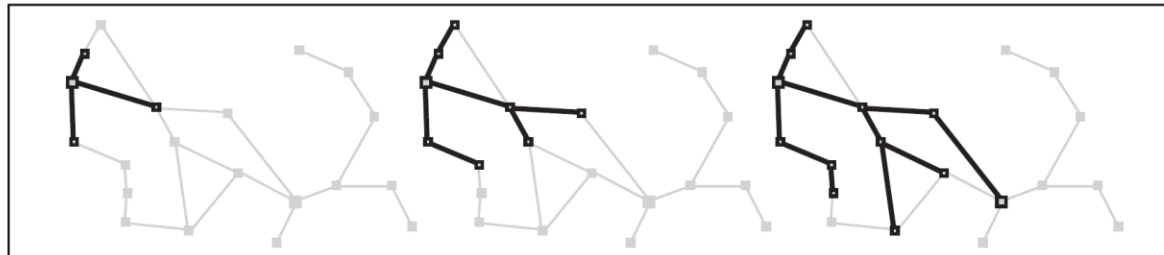


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

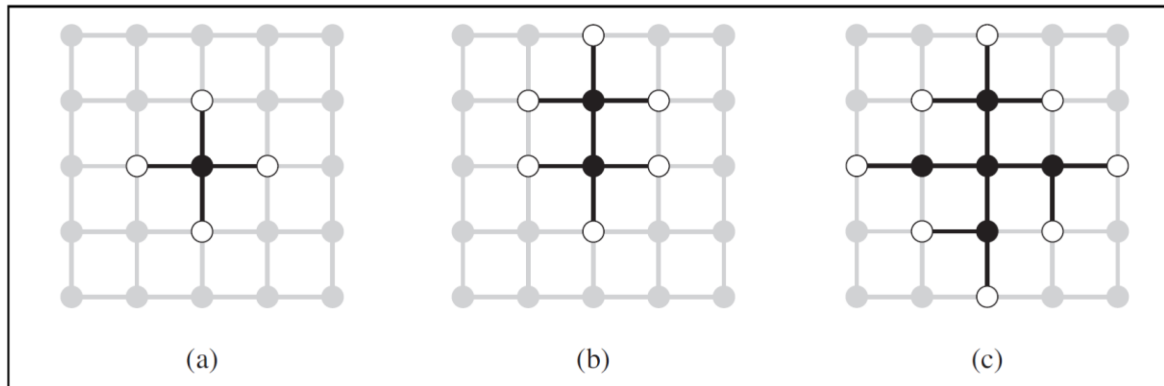


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

Compromesso tra spazio e tempo

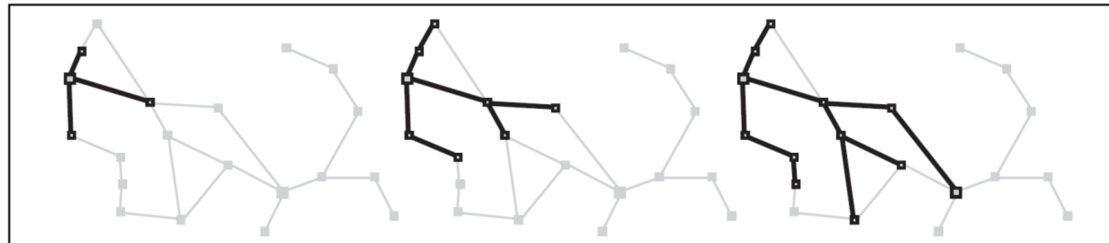
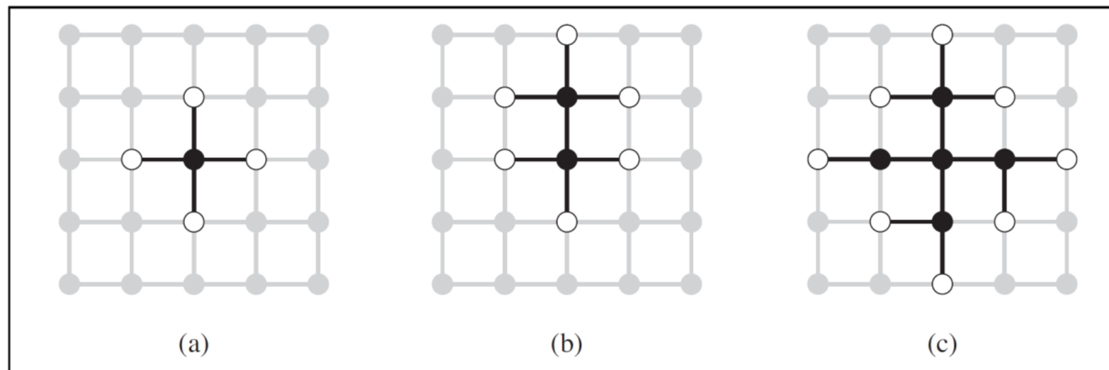


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.



- Ricordare gli stati già visitati occupa spazio ma ci consente di evitare di visitarli di nuovo
- *Gli algoritmi che dimenticano la propria storia sono destinati a ripeterla!*

Tre soluzioni

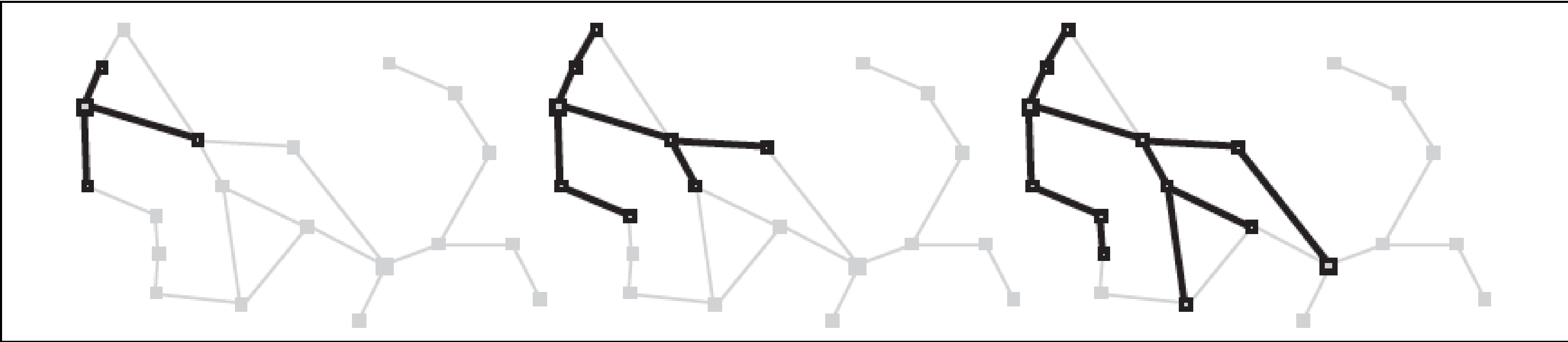
In ordine crescente di costo e di efficacia:

- Non tornare nello stato da cui si proviene: si elimina il genitore dai nodi successori
- Non creare cammini con cicli: si controlla che i successori non siano antenati del nodo corrente
- Non generare nodi con stati già visitati: ogni nodo visitato deve essere tenuto in memoria per una complessità O(s) dove s è il numero di stati possibili (*hash table*).

Ricerca su grafi

- Mantiene una lista dei nodi visitati (*lista chiusa*)
- Prima di espandere un nodo si controlla se lo stato era stato già incontrato prima
- Se questo succede, il nodo appena trovato non viene espanso
- Ottimale solo se abbiamo la garanzia che il costo del nuovo cammino non è maggiore

Ricerca sul grafo della Romania



- La ricerca su grafo esplora uno stato al più una volta
- La frontiera separa i nodi esplorati da quelli non esplorati

Ricerca-grafo in ampiezza

```
function Ricerca-Ampiezza-g (problema)
    returns soluzione oppure fallimento
     nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
    if problema.Test-Obiettivo(nodo.Stato) then return Soluzione(nodo)
    frontiera = una coda FIFO con nodo come unico elemento
    esplorati = insieme vuoto
  loop do
    if Vuota?(frontiera) then return fallimento
    nodo = POP(frontiera); aggiungi nodo.Stato  a esplorati
    for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione)
            if figlio.Stato non è in esplorati e non è in frontiera then
               if Problema.TestObiettivo(figlio.Stato) then return Soluzione(figlio)
               frontiera = Inserisci(figlio, frontiera)   /* in coda
```

Ricerca-grafo UC

function Ricerca-UC-G (*problema)*

  returns soluzione oppure **fallimento**

  nodo = un nodo con *stato il problema.stato-iniziale e costo-di-cammino=0*

  frontiera = una coda con priorità con *nodo* come unico elemento

  esplorati = insieme vuoto

 loop do

  if *Vuota?*(*frontiera*) **then return fallimento**

  nodo = POP(*frontiera*);

  if problema.TestObiettivo(*nodo*.Stato) **then return** Soluzione(*nodo*)

  aggiungi *nodo*.Stato a *esplorati*

  for each azione **in** problema.Azioni(nodo.Stato) **do**

    figlio = Nodo-Figlio(*problema*, *nodo, azione*)

    if *figlio*.Stato non è in *esplorati* e non è in *frontiera* **then**

      frontiera = Inserisci(*figlio*, *frontiera*) /* in coda con priorità

    else if *figlio*.Stato è in *frontiera* con Costo-cammino più alto **then**

      sostituisci quel nodo frontiera con figlio

Conclusioni

- Un agente per "problem solving" adotta un paradigma generale di risoluzione dei problemi:
  - Formula il problema
  - Ricerca la soluzione nello spazio degli stati
- Strategie "non informate" per la ricerca della soluzione
- Prossima volta: come si può ricercare meglio

SummarAIzing

Gli agenti sono orientati in genere alla soluzione di problemi, poichè hanno un certo *goal* come obbiettivo ed una funzione *utilità*

In un caso specifico di ambiente deterministico, osservabile, statico, e completamente conosciuto) l'agente può porre in essere il suo comportamento come *un processo di ricerca*

- **Un problema deve essere dunque formulato come** una quintupla:
  - **Stato iniziale**,
  - l'insieme delle **azioni**,
  - un **modello delle transizioni**,
  - una verifica di successo (**goal test**)
  - una **funzione costo**
- L'ambiente è lo spazio degli stati: un cammino nello spazio degli stati che conduce ad uno stato obbiettivo costituisce una **soluzione**.
- In generale un algoritmo di ricerca ad albero considera tutti i cammini verso la soluzione
- Un algoritmo di GRAPH-SEARCH evita cammini ridondanti.

SummarAIzing (2)

- La qualità di un algoritmo di ricerca è misurata in base alla sua completezza, ottimalità, complessità spaziale e temporale (*b*, il branching factor nello spazio degli stati, e *d*, la profondità della prima soluzione trovata)
- Metodi di **Uninformed search**:
  - **Breadth-first search (**espande i nodi meno profondi prima); è completo, ottimo per costi unitary di passo, ma ha una complessità spaziale esponenziale.
  - **Uniform-cost search (**ottimo per il costo del passo generico)
  - **Depth-first search** espande i nodi più profondi prima (nè completo nè ottimo, ma ha una complessità spaz. lineare. (**Depth-limited search** aggiunge un bound alla profondità**)**.
  - **Iterative deepening search** è un Depth-limited search con profondità crescenti fino al goal: completo, ottimo per costi di step unitari, complessità simile a breadth-first search, complessità spaziale lineare.
  - **Bidirectional search** può ridurre enormemente la time complexity: non sempre applicabile poichè richiede molto spazio di memoria.