Ricerca in uno spazio di stati (SEARCH)

- DEFINIZIONE E CLASSIFICAZIONE DEI PROBLEMI
- ALGORITMI DI RICERCA
  Algoritmi ciechi
  Algoritmi euristici
  Algoritmi A*, IDA*

Riferimenti :
Stuart J. Russell, Peter Norvig
"Intelligenza Artificiale, un approccio moderno", Volume 1
Capitoli 3, 4

PROBLEM SOLVING AGENT

Agente che ha un obiettivo da raggiungere e che deve determinare una sequenza di azioni che permetta di raggiungerlo.

FORMULAZIONE DEL PROBLEMA

- formulazione dello stato iniziale
- formulazione dell'obiettivo (insieme di stati)
- definizione delle azioni (transizioni tra stati)

$\|$

problema

$\Downarrow$

SEARCH

RICERCA DI UNA SOLUZIONE

esame di diverse sequenze di azioni e scelta di una sequenza che permette di raggiungere uno stato obiettivo

$\|$

soluzione = sequenza di azioni

$\Downarrow$

ESECUZIONE

Realizzazione della sequenza di azioni trovata

TIPI DI PROBLEMI

I problemi affrontabili dipendono dalla conoscenza che l'agente ha sullo stato in cui si trova e sulle azioni (effetti delle azioni)

SEARCH/ PLANNING

Problemi a stato singolo

- conoscenza esatta dello stato in cui si trova l'agente (totale accessibilità, presenza di sensori)
- conoscenza esatta degli effetti di ogni azione

SOLUZIONE: sequenza di stati

Problemi a stato multiplo

- conoscenza incompleta sullo stato, azioni note (ambiente deterministico)
- conoscenza esatta dello stato, ambiente indeterministico

SOLUZIONE: sequenza di insiemi di stati

PLANNING

Problemi di contingenza

- azioni non deterministiche, conoscenza incompleta sullo stato

Richiesta di input sensoriale durante l'esecuzione
SOLUZIONE: albero di stati
Alternanza di ricerca ed esecuzione

LEARNING

Problemi di esplorazione

Nessuna informazione sullo stato e sulle azioni

Problemi di search (a stato singolo o multiplo)

Componenti che definiscono un problema:

- Stato iniziale
- Azioni o operatori
- Test obiettivo (*goal test*)
- Funzione costo: associa un costo a ogni operatore

	A stato singolo	A stato multiplo
stato iniziale	$\{s_0\}$	$\{s_0, \ldots s_n\}$
azioni o operatori	$op(s) = s'$	$op^*(\{s_1, \ldots s_k\}) = \{op(s_1)\} \cup \ldots \cup \{op(s_k)\}$ (operatori su insiemi)
goal test	$goal\_state(s)$	$goal\_state^*(\{s_1, \ldots s_k\}) = goal\_state(s_1) \wedge \ldots \wedge goal\_state(s_k)$ (goal test per insiemi di stati)

Un **problema** così definito è **l'input** dell'algoritmo di ricerca.

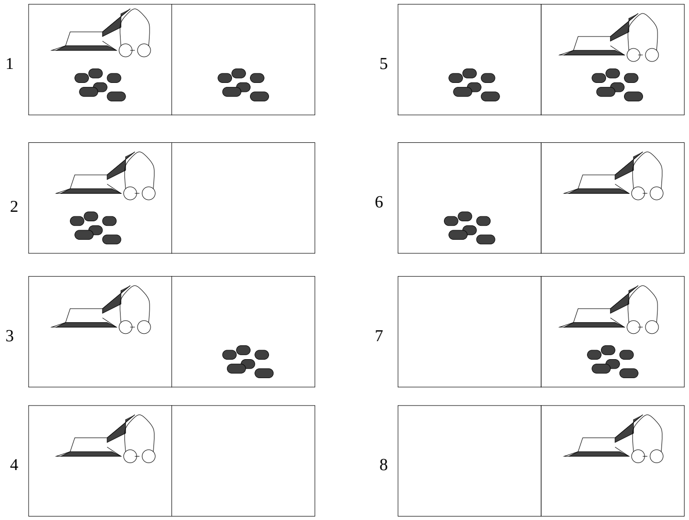
Lo stato (o gli stati) iniziali e gli operatori identificano lo **SPAZIO DEGLI STATI**

Un **CAMMINO** è una **sequenza di operatori**

Il **COSTO DI UN CAMMINO** è la somma dei costi degli operatori che lo compongono

Una **SOLUZIONE** è un cammino dallo stato iniziale (da uno qualsiasi degli stati iniziali) a uno stato che soddisfa il goal-test: è una **sequenza di operatori**

Esempio: il robot aspirapolvere



Gli otto stati possibili del mondo del robot aspirapolvere

Azioni: `Left, Right, Suck`

Problema a stato singolo

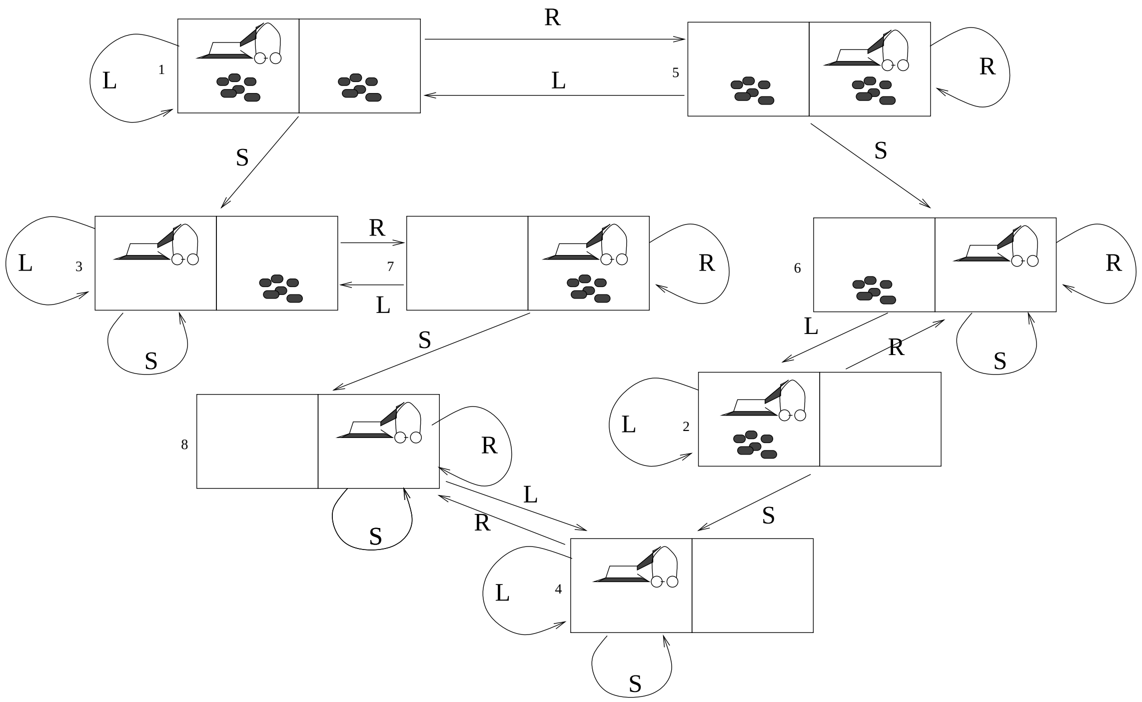
stato iniziale : uno degli stati 1,...,8

operatori : `Left, Right, Suck`

goal test : { 4,8 }

costo : ogni azione ha costo 1

Spazio degli stati



Problema a stato multiplo

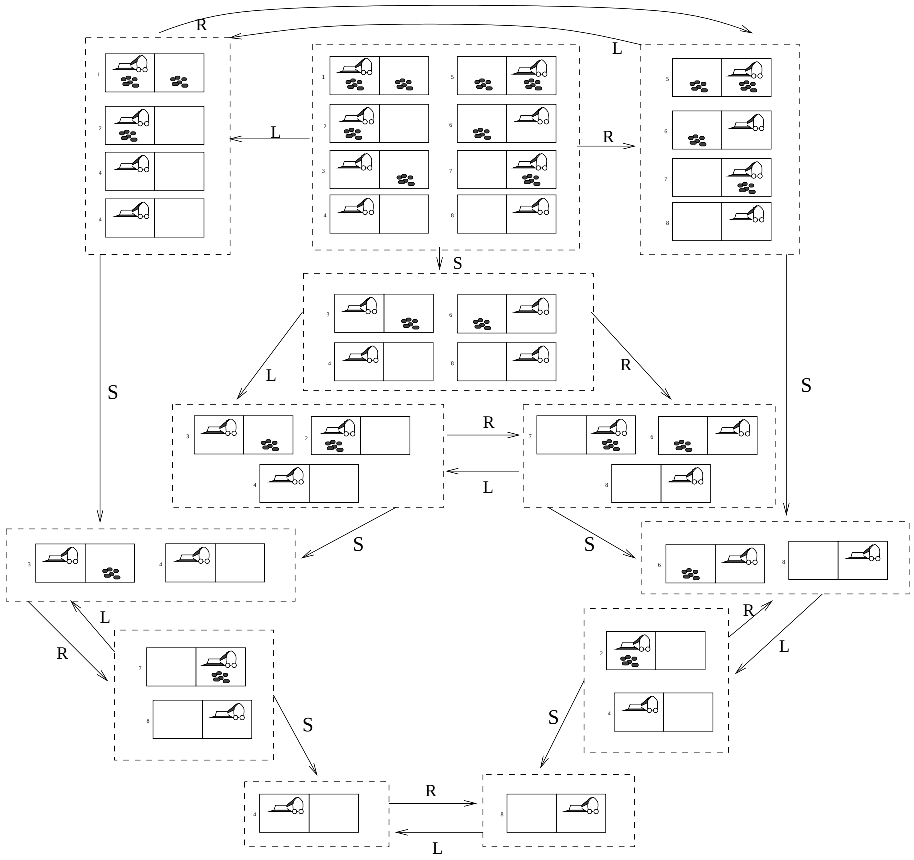
stati iniziali : un sottoinsieme di $\{1, ..., 8\}$

operatori : `Left`, `Right`, `Suck`

goal test : $\{\{4\}, \{8\}, \{4, 8\}\}$

costo : ogni azione ha costo 1

Spazio degli stati



Ricerca di soluzioni

Input: Problema

- stato/i iniziale/i
- goal-test
- operatori (azioni) con costo associato

Output: Sequenza di azioni

Albero di ricerca

La ricerca avviene mediante la costruzione di un **albero di ricerca**, che si sovrappone allo spazio degli stati.

Ogni nodo dell'albero contiene uno stato, insieme ad altre informazioni

```
NODO = <stato, genitore, operatore, profondità,...>
```

Nodo:

- stato
- nodo genitore
- operatore che, applicato allo stato del nodo genitore, genera lo stato del nodo
- profondità del nodo
- costo del cammino dallo stato iniziale al nodo
- ...

La radice dell'albero contiene lo stato iniziale

N.B. nodi diversi possono contenere lo stesso stato

Spazio degli stati $\neq$ Albero di ricerca

Processo di ricerca

- scegliere (tra le foglie dell'albero) un nodo da "espandere", secondo una data *strategia*
- controllare se il nodo scelto è un obiettivo
- se non lo è, "espandere" il nodo: generare i suoi figli, ciascuno dei quali contiene uno stato risultante dall'applicazione di un operatore allo stato del nodo espanso.

  Quando si espande un nodo si calcolano tutte le componenti dei nodi generati.

La collezione di nodi in attesa di essere espansi (le foglie dell'albero) è chiamata **confine**, **frontiera** o **frangia**.

Algoritmo generale di ricerca

```
function GENERAL-SEARCH (problem,strategy)
         riporta una soluzione o fallimento

inizializzare l'albero di ricerca usando lo stato
           iniziale del problema
loop do
     se non ci sono candidati per l'espansione (la frontiera e` vuota)
         riportare fallimento
     scegliere una foglia per l'espansione, in accordo con la strategia
     se il nodo scelto contiene uno stato obiettivo,
         allora riportare la soluzione corrispondente
         (costruita seguendo i "puntatori al padre")
     altrimenti espandere il nodo e aggiungere
         i nodi risultanti all'albero di ricerca
end
```

Attenzione: la frontiera può contenere un nodo contenente uno stato obiettivo, ma la ricerca non termina finché tale nodo non viene scelto per l'espansione: soltanto allora infatti si riconosce che contiene uno stato obiettivo.

La strategia di ricerca

Come rappresentare la **strategia di ricerca** (criterio per la scelta del prossimo nodo da espandere)?

Strategia di ricerca $\equiv$ funzione per la scelta di un elemento da un insieme di nodi (frontiera)
$\equiv$ funzione di inserimento di un elemento (o un insieme di elementi) in una sequenza.
Scelta di un elemento = scelta del primo elemento dalla sequenza

Quindi: la frontiera è implementata da una struttura che chiamiamo **"coda" (queue)** (anche se non è necessariamente una struttura FIFO), sulla quale sono definite le seguenti operazioni:

`MAKE-QUEUE` : `node -> queue` `MAKE-QUEUE(n)`: coda contenente solo il nodo `n`

`EMPTY?` : `queue -> bool` test coda vuota

`REMOVE-FRONT` : `queue -> node`
`REMOVE-FRONT(q)`: elimina il primo elemento da `q` e lo riporta come valore

`QUEUING-FN` : `queue * node list -> queue`
`QUEUING-FN(q,lista)`: aggiunge a `q` tutti i nodi in `lista`

La `QuUEUING-FN` non inserisce necessariamente in coda:
diverse `QUEUING-FN` producono diverse strategie di ricerca e, corrispondentemente, diverse versioni dell'algoritmo di ricerca.

Algoritmo generale di ricerca (2)

Assumiamo che siano definite:

- Operazioni sul tipo di dato PROBLEMA:

  `INITIAL-STATE` : `problem -> state`. Riporta lo stato iniziale del problema

  `GOAL-TEST` : `problem * state -> bool`.
  `GOAL-TEST(p,s)`: vero se lo stato `s` soddisfa il test obiettivo del problema `p`.

  `OPERATORS` : `problem -> operators`. Riporta una lista con tutti gli operatori del problema.
  Ciascun operatore si applica a uno stato e riporta una lista di stati

  ```
  op:  state -> state list
  ```

- Operazioni sul tipo di dato NODO:

  `MAKE-NODE` : `state -> node`. Costruisce il nodo radice contenente lo stato dato

  `STATE` : `node -> state`. Riporta lo stato contenuto nel nodo

  `EXPAND` : `node * operators -> node list`.
  `EXPAND(n,ops)`: lista di tutti i nodi che si ottengono applicando un operatore in `ops` al nodo `n`

Algoritmo generale di ricerca (3)

```
function GENERAL-SEARCH (problem,QUEUING-FN)
            returns a solution or failure
nodes <- MAKE-QUEUE (MAKE-NODE (INITIAL-STATE(problem)));
loop do
     if EMPTY?(nodes) then return failure;
     node <- REMOVE-FRONT(nodes);
     if GOAL-TEST(problem,STATE(node)) then return node;
     nodes <- QUEUING-FN (nodes,
                          EXPAND(node,OPERATORS(problem)))
end
```

Osservazione: In memoria non viene conservato l'intero albero di ricerca, ma soltanto la "coda" con i nodi della frontiera.

Criteri per valutare una strategia di ricerca

Completezza : la strategia garantisce di trovare una soluzione quando ne esiste una

Complessità in tempo : quanto tempo ci vuole per trovare una soluzione (nel caso peggiore)

Complessità in spazio : quanta memoria occorre per effettuare la ricerca

Ottimalità : quando il problema ha diverse soluzioni, la strategia trova una delle migliori (a costo minimo)

Ricerca cieca

Non ci sono informazioni per stimare quanto uno stato sia “vicino” a un obiettivo.
Non abbiamo nessuna informazione per scegliere in base ad una “convenienza” (in lunghezza o costo del cammino)

Ricerca euristica

La scelta del nodo da espandere è guidata da informazioni che consentono di valutare quanto esso sia “promettente”
(in base a informazioni specifiche sul dominio)

Ricerca in uno spazio di stati e ricerca di un cammino in un grafo

In un grafo

- Il grafo è rappresentato esplicitamente in memoria: è un insieme di nodi e di archi (coppie di nodi).
- Un cammino è una sequenza di stati.
- Nella ricerca si controllano i nodi già visitati.

In uno spazio di stati

- Lo spazio degli stati è rappresentato implicitamente dallo stato iniziale e dagli operatori (funzioni da stati a insiemi di stati). Gli stati vengono generati soltanto durante la ricerca.
- Un cammino è una sequenza di operatori.
- Nella ricerca non si controllano necessariamente i nodi già visitati.

Algoritmi di ricerca cieca

(Simili agli algoritmi di visita di grafi)

- Ricerca in ampiezza
- Ricerca guidata dal costo
- Ricerca in profondità
- Ricerca in profondità limitata
- Ricerca per approfondimenti successivi
- Ricerca bidirezionale

RICERCA IN AMPIEZZA

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

Funzione di inserimento nella coda: `ENQUEUE-AT-END`

```
function BREADTH-FIRST-SEARCH (problem)
   returns a solution or failure
return GENERAL-SEARCH (problem, ENQUEUE-AT-END)
```

Tutti i nodi di profondità d sono espansi prima di quelli di profondità $d+1$.

Strategia sistematica. Trova sicuramente (se c'è) la soluzione realizzabile con il cammino più breve, cioè trova gli stati obiettivo più superficiali. Ma il cammino più breve non è necessariamente quello a costo minimo.

Proprietà della ricerca in ampiezza

Completezza : sì

Ottimalità : sì, **se**:

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

cioè se il costo di cammino è una funzione non decrescente della profondità del nodo.

Complessità in tempo : $O(b^d)$, con

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

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

Il massimo numero di nodi da considerare prima di trovare una soluzione è =

(nodi del livello 0) + (nodi del livello 1) + ...
... + (nodi dell livello d)

$$\leq 1 + b + b^2 + \ldots + b^d$$

Complessità in spazio : $O(b^d)$
Tutte le foglie dell'albero devono essere conservate in memoria.
Nel caso peggiore, al livello d ci sono b^d foglie

Tempo e memoria richiesti per la ricerca in ampiezza

Assumendo branching factor $b = 10$, generazione di 1000 nodi al secondo, e 100 bytes per conservare ogni nodo

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

N.B. Requisiti di memoria inaccettabili!

RICERCA GUIDATA DAL COSTO
(uniform cost search)

Modifica la ricerca in ampiezza espandendo il nodo nella frontiera con costo più basso.

Se:

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

viene scelto per l'espansione il nodo n il cui costo $g(n)$ è minore

Coda a priorità: $priority(n) > priority(m)$ sse $g(n) < g(m)$

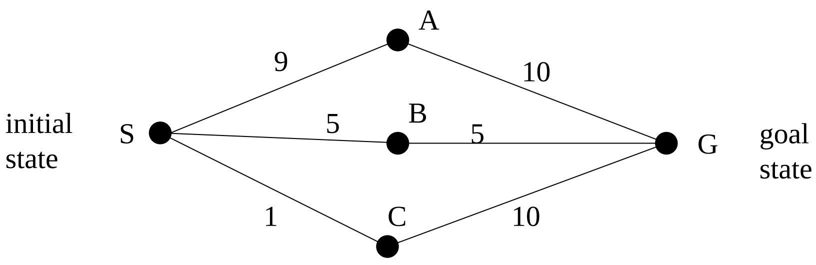
Se $g(n) = depth(n)$ si ha la ricerca in ampiezza

È ottimale **se** si ha sempre

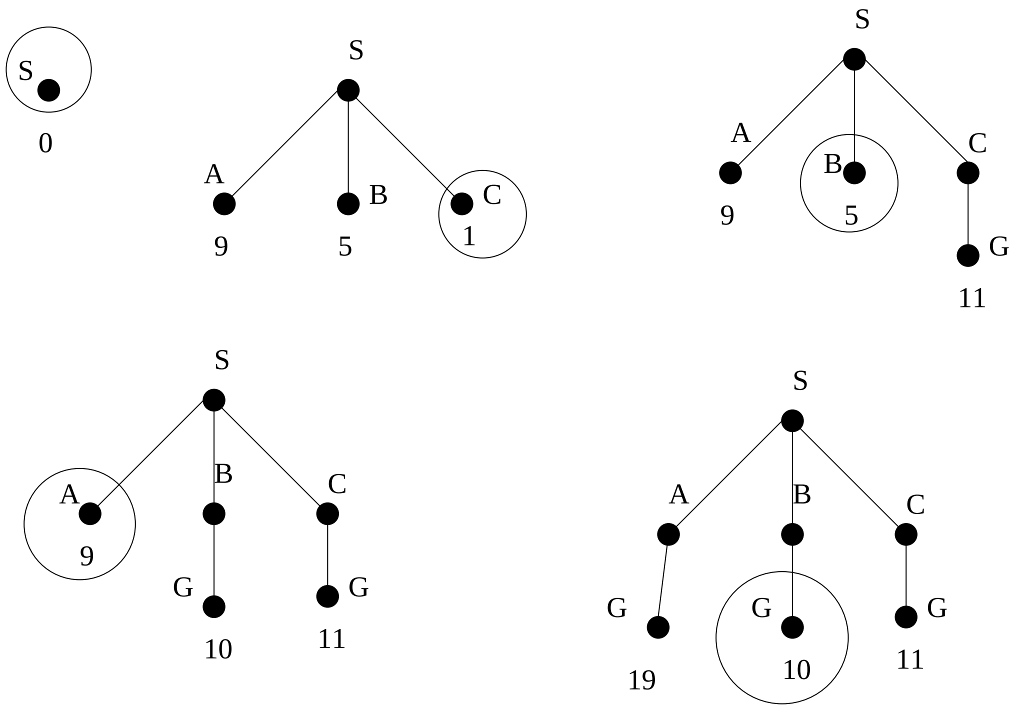
$$g(successor(n)) \geq g(n)$$

cioè se $g(n)$ è la somma dei costi degli operatori nel cammino e il costo di ciascun operatore è non negativo.

Esempio



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



S
0
S
A
B
C
9
5
1
S
A
B
C
G
9
5
11
S
A
B
C
G
G
9
10
11
S
A
B
C
G
G
G
19
10
11

RICERCA IN PROFONDITÀ

Funzione di inserimento nella coda: `ENQUEUE-AT-FRONT`

```
function DEPTH-FIRST-SEARCH (problem)
   returns a solution or failure
return GENERAL-SEARCH (problem, ENQUEUE-AT-FRONT)
```

Si può implementare mediante una funzione ricorsiva.

In tal caso la lista dei nodi da visitare (pila) è conservata implicitamente nello stack dei record di attivazione.

Quando si espande un nodo non obiettivo e senza figli, si fa **backtracking**, cioè si torna indietro fino all'ultimo nodo in cui è possibile effettuare una scelta.

Non è né completa né ottimale

Complessità in tempo : $O(b^m)$, con

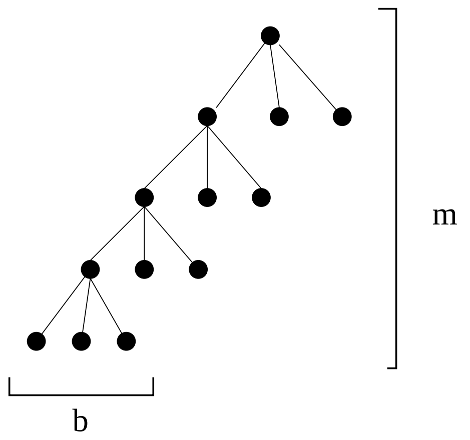
b = fattore di ramificazione dell'albero

m = profondità MASSIMA dell'albero di ricerca

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

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

Il massimo numero di nodi da conservare in memoria è $b \cdot m$



Se $b = 10$, 100 bites/nodo, $m = 12$: 12 K di memoria

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

RICERCA IN PROFONDITÀ LIMITATA
(depth-limited search)

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

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

N.B. Si possono utilizzare conoscenze specifiche del problema per fissare il limite di profondità.
Attenzione alla scelta di un buon limite. ES. Mappa delle città della Romania

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

Non è ottimale

Complessità in tempo : $O(b^l)$, con

$b =$ fattore di ramificazione dell'albero

$l =$ limite di profondità fissato

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

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

RICERCA PER APPROFONDIMENTI SUCCESSIVI
(iterative-deepening search)

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

```
function ITERATIVE-DEEPENING-SEARCH (problem)
         returns a solution sequence
for depth ← 0 to ∞ do
   if DEPTH-LIMITED-SEARCH (problem,depth) succeeds
         then return its result
end
```

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

È completa

È ottimale

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

$b =$ fattore di ramificazione dell'albero

$d =$ profondità minima di una soluzione

Complessità in tempo : $O(b^d)$

- il nodo del livello 0 viene espanso $d + 1$ volte
- i nodi del livello 1 (b) sono espansi d volte
- i nodi del livello 2 (b^2) sono espansi $d - 1$ volte

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

Numero di espansioni:

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

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

11% in più rispetto alla ricerca in ampiezza

Ricerca bidirezionale

Si alterna la ricerca in avanti dallo stato iniziale con la ricerca all'indietro dall'Obiettivo e si controlla l'intersezione tra le due frontiere

Numero di nodi espansi: $O(2 \cdot b^{d/2}) = O(b^{d/2})$

Se $b = 10$ e $d = 6$:

in ampiezza:	$\approx$ 1.000.000
bidirezionale:	$\approx$ 2.000

Ma:

- Non sempre è facile calcolare i predecessori di un nodo
- Si assume di poter generare l'insieme degli stati che soddisfano il goal test
- la compelssità in tempo è $O(b^{d/2})$ **assumendo costo costante per il test di intersezione** (tabella hash)

Complessità in spazio : $O(b^{d/2})$
I nodi di almeno una delle due ricerche devono essere conservati in memoria

ESERCIZIO

Problema: sono a Milano e voglio andare a Napoli.
Ho una mappa d'Italia che rappresenta gli operatori ed i costi loro associati: la mappa indica quali città sono collegate direttamente, insieme al relativo costo di percorrenza, in decine di chilometri.

	AQ	AN	BA	BO	FI	GE	MI	NA	PG	PI	RM	TO
AQ		19							17		11	
AN	19		46	21					16			
BA		46									45	
BO		21			10		21					
FI				10		22			15	9	28	
GE					22		14			16		
MI				21		14						14
NA											22	
PG	17	16			15						17	
PI					9	16					37	
RM	11		45		28			22	17	37		
TO							14					

Rappresentazione del problema

- Quale tipo di struttura si utilizza per rappresentare gli stati?

  Ciascuno stato della ricerca è identificato dalla città in cui mi trovo.

  In questo caso uno stato è una struttura semplice, che possiamo rappresentare mediante un "atomo" (di un tipo di dati semplice: stringa, intero, ...) che identifica il nome della città in cui ci si trova.

  Ad esempio: `Aq, An, Ba, Bo, Fi, ...`

- Qual è lo stato iniziale?

  Stato Iniziale: `Mi`

- Qual è il goal test (funzione da stati a booleani)?

  Goal(s): `s=Na`

- Quali sono gli operatori?

  Operatori: `GO(a,b)`, dove `(a,b)` è una delle coppie di città per le quali è noto il costo del collegamento diretto.

  Condizioni di applicabilità: `GO(a,b)` è applicabile solo allo stato `a`.

  Descrizione dell'operatore: `GO(a,b)`, applicato allo stato `a`, riporta lo stato `b`:

  ```
  GO(a,b)(a) = b
  ```

  Costo dell'operatore: è quello indicato in tabella.

Esercizio:

- Disegnare lo spazio degli stati
- Costruire l'albero di ricerca seguendo le strategie di ricerca in ampiezza, in profondità, guidata dal costo. Indicare in ciascun caso la soluzione trovata (se l'algoritmo termina con una soluzione).

RICERCA EURISTICA

Uso di conoscenza specifica sul problema

EURISTICA: che aiuta nella scoperta

Nell'algoritmo GENERAL-SEARCH si può usare conoscenza euristica per definire la QUEUING-FUNCTION – che è ciò che determina la scelta del nodo da espandere

La conoscenza sul problema è rappresentata da una

funzione di valutazione

che si applica ai nodi dell'albero di ricerca.

$$\mathbf{f:\ nodo} \mapsto \begin{cases} \text{stima della desiderabilità di espandere il nodo;} \\ \text{misura di quanto il nodo è “promettente”} \end{cases}$$

N.B. Normalmente f è una stima del costo della soluzione, quindi viene considerato migliore il nodo n con $f(n)$ minore

Schema di ricerca euristica: BEST-FIRST SEARCH (prima il migliore)

La QUEUING-FN ordina i nodi dal migliore al peggiore secondo la funzione di valutazione

Il nodo scelto per l'espansione è quello la cui valutazione è "migliore" (normalmente quello con valore di f minore), cioè il nodo apparentemente più promettente

Diverse funzioni di valutazione determinano diverse versioni della ricerca BEST-FIRST

Funzione di valutazione:

f : nodo $\mapsto$ stima del costo di una soluzione che passa per il nodo

Negli algoritmi di ricerca euristica, la misura incorpora una stima del costo del miglior cammino dal nodo a una soluzione: conoscenza euristica.

La **ricerca guidata dal costo** si può considerare un caso particolare di ricerca best-first, in cui

$$\boxed{f = g}$$

La funzione di valutazione è $g(n)$: il costo del cammino dallo stato iniziale a n.
Informazione euristica nulla

BEST-FIRST SEARCH

Implementazione della ricerca best first usando l'algoritmo di search generale:

```
function BEST-FIRST-SEARCH (problem,EVAL-FN)
       returns a solution sequence
inputs: problem, a problem
        EVAL-FN, an evaluation function

Queueing-Fn <- a function that orders nodes by EVAL-FN
return GENERAL-SEARCH(problem, Queueing-Fn)
```

Notazioni

Notazioni che riguardano lo spazio degli stati e costi effettivi

$\boxed{s, s_0, s_1, s_2, ...}$: stati del problema

$\boxed{s_0}$: stato iniziale

$\boxed{k^*(s_i, s_j)}$: costo del cammino a costo minimo da s_i a s_j (se esiste)

$\boxed{g^*(s_i)} = k^*(s_0, s_i)$: costo del cammino a costo minimo dallo stato inziale a s_i

$\boxed{h^*(s_i)}$: costo effettivo di un cammino a costo minimo da s_i a uno stato obiettivo

$\boxed{f^*(s_i)} = g^*(s_i) + h^*(s_i)$: costo minimo di una soluzione vincolata a passare per s_i

Notazioni che riguardano l'albero di ricerca

$\boxed{n, n_1, n_2}$: nodi dell'albero di ricerca

Notazioni che riguardano l'albero di ricerca e costi effettivi

$\boxed{g^*(n)} = g^*(STATE(n))$

$\boxed{h^*(n)} = h^*(STATE(n))$

Notazioni che riguardano l'albero di ricerca e la stima di costi

$\boxed{g(n)}$: **stima di** $g^*(n)$ = costo del cammino dalla radice dell'albero a n

$$g(n) \geq g^*(n)$$

$\boxed{h(n)}$: **stima di** $h^*(n)$ = stima del costo del cammino a costo minimo dallo stato di n a uno stato obiettivo

h : FUNZIONE EURISTICA

GREEDY SEARCH (ricerca golosa)

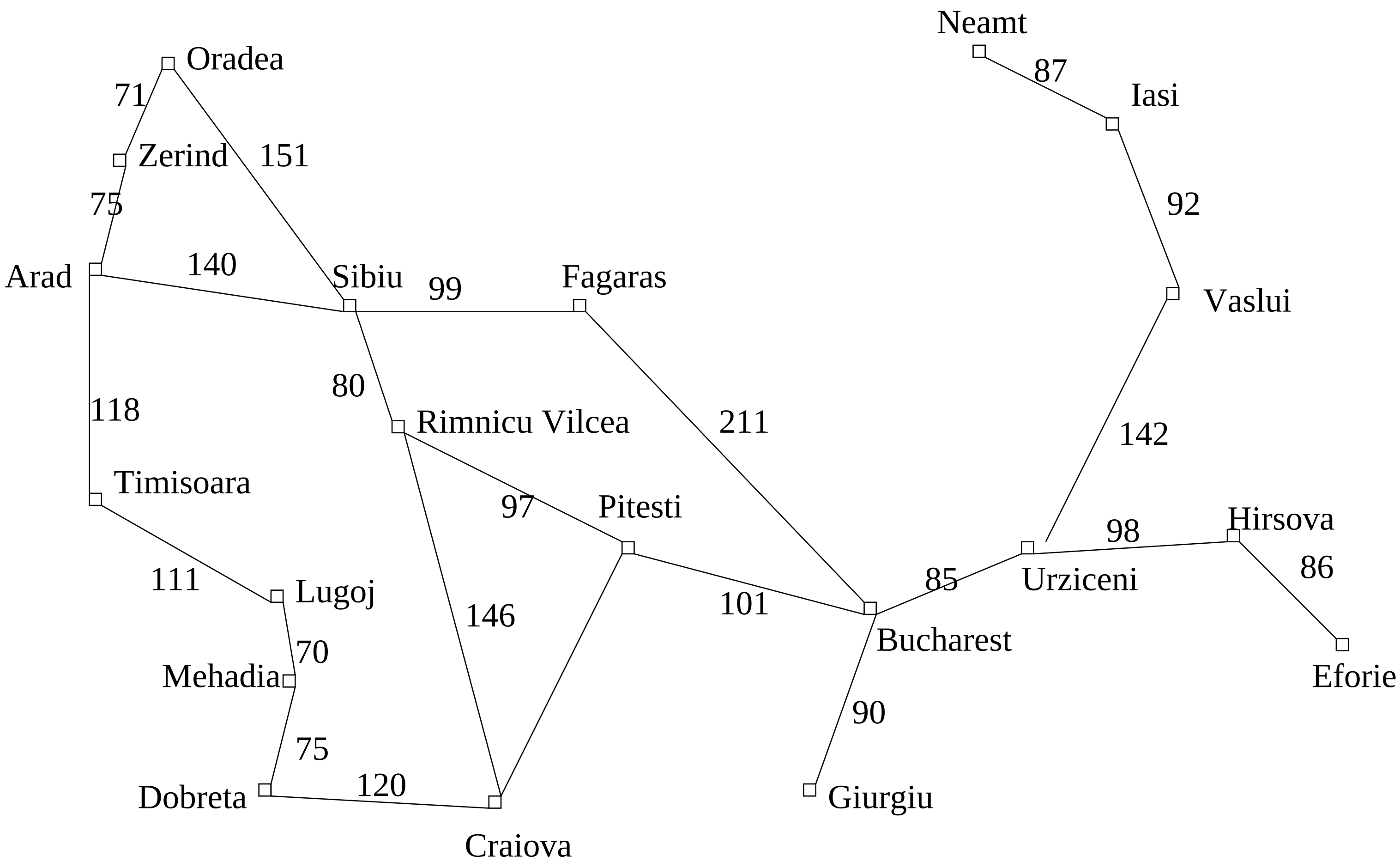
Funzione di valutazione : $\boxed{\mathrm{f = h}}$

È espanso per primo il nodo ritenuto più vicino a un obiettivo

Si deve avere $h(n) = 0$ se lo stato in n è un obiettivo

La ricerca golosa minimizza il costo stimato per raggiungere l'obiettivo.

Esempio: ricerca di un percorso da Arad a Bucharest



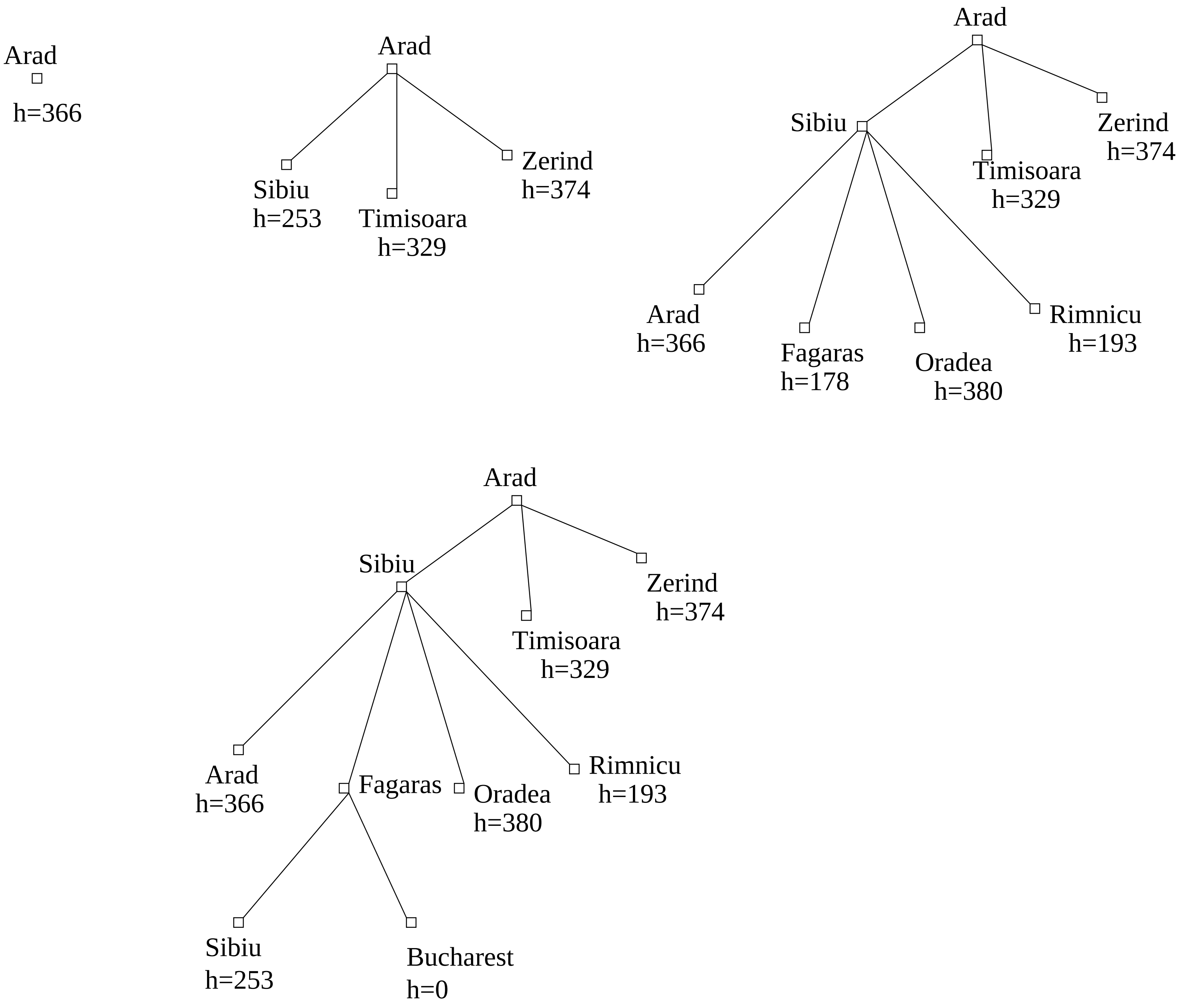
Esempio: funzione euristica

$h_{SLD}(n)$ = distanza in linea d'aria (Straight Line Distance) dallo stato in n all'obiettivo

SLD da Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	98
Eforie	161	Rimnicu Vilcea	193
Fagaras	178	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

La ricerca golosa con $h = h_{SLD}$



Soluzione

Soluzione trovata: Arad - Sibiu - Fagaras - Bucharest

Lunghezza del percorso: 450 Km

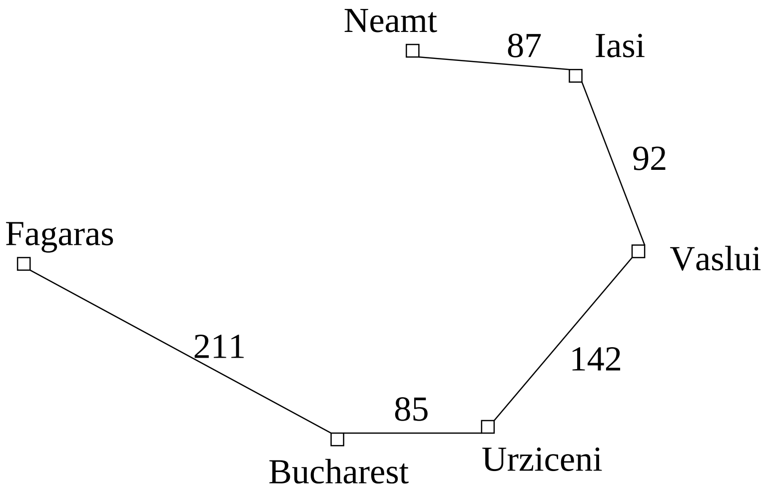
Non è una soluzione ottima: Arad - Sibiu - Rimnicu - Pitesti - Bucharest: 418 Km

La ricerca golosa non è ottimale

Problema delle false partenze

La ricerca golosa non è completa

Esempio: ricerca di un percorso da Iasi a Fagaras: per raggiungere l'obiettivo ci si deve prima allontanare (in linea d'aria)



Necessità di evitare stati già esaminati

Completezza, Ottimalità e Complessità della ricerca golosa

Non è completa

Potrebbe seguire un cammino infinito senza trovare mai la soluzione (come la ricerca in profondità).

Non è ottimale

(come la ricerca in profondità).

Complessità in tempo $O(b^m)$ (caso peggiore)

$m =$ profondità massima dell'albero di ricerca
$b =$ fattore di ramificazione dell'albero di ricerca

Complessità in spazio $O(b^m)$

perchè deve tenere in memoria tutti i nodi.

Nonostante le apparenze con una buona euristica possiamo avere ottimi risultati

L'algoritmo A*

Ricerca Greedy: minimizza il costo stimato ma non è né ottimale né completa.

Ricerca guidata dal costo: ottimale e completa ma a volte inefficiente.

Algoritmo A*:

Funzione di valutazione : $f(n) = g(n) + h(n)$

stima del costo del cammino a costo minimo dallo stato iniziale a un obiettivo, vincolato a passare per lo stato di n,

Ricordando che $h^*(n)$ è il costo effettivo del cammino a costo minimo dallo stato di n a uno stato obiettivo:

Se h è un'euristica ammissibile, cioè se

$$\forall n \quad h(n) \leq h^*(n)$$

(h "ottimista")

allora A* è completo e ottimale

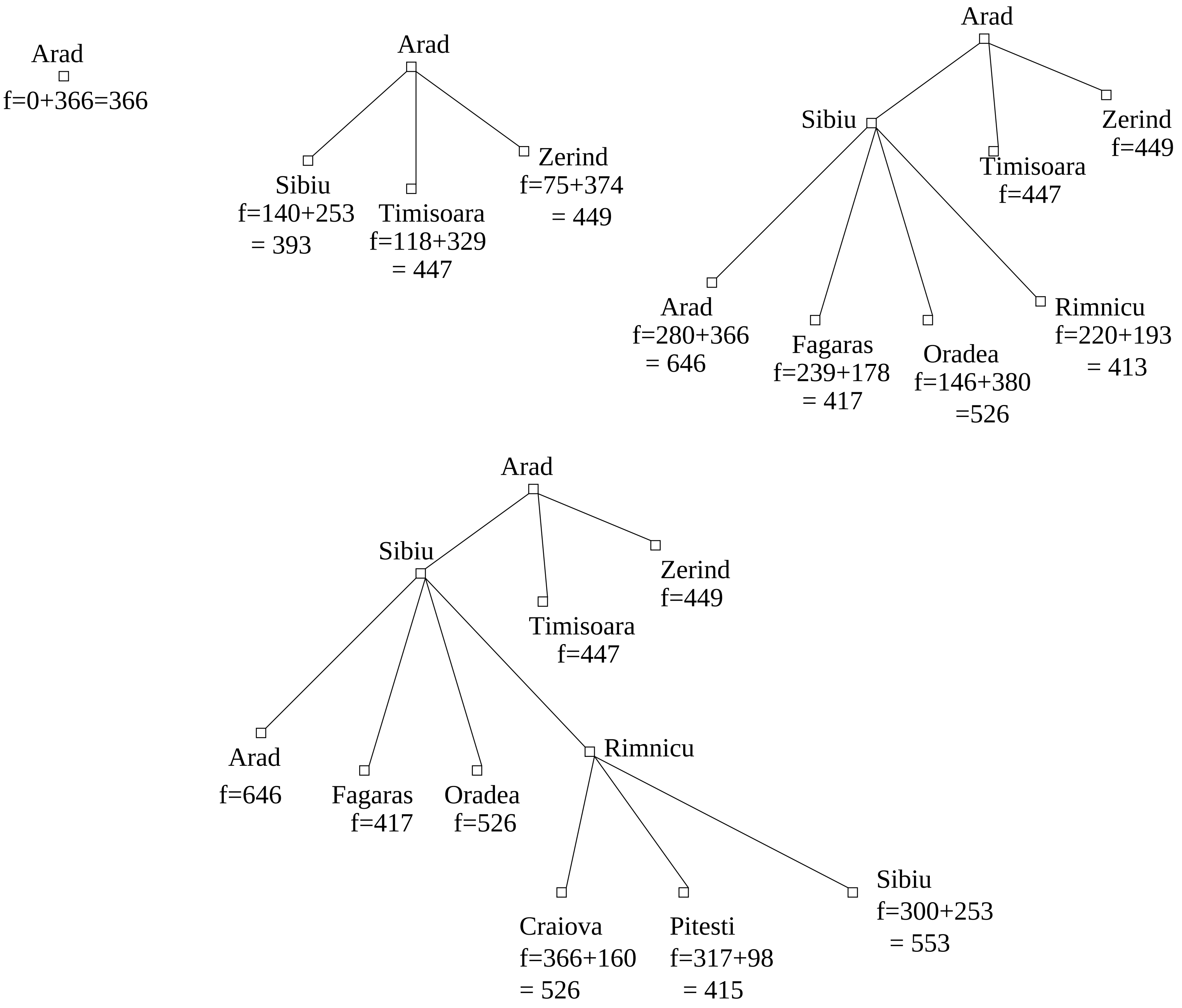
Nella ricerca di percorsi; h_{SLD} è ammissibile.

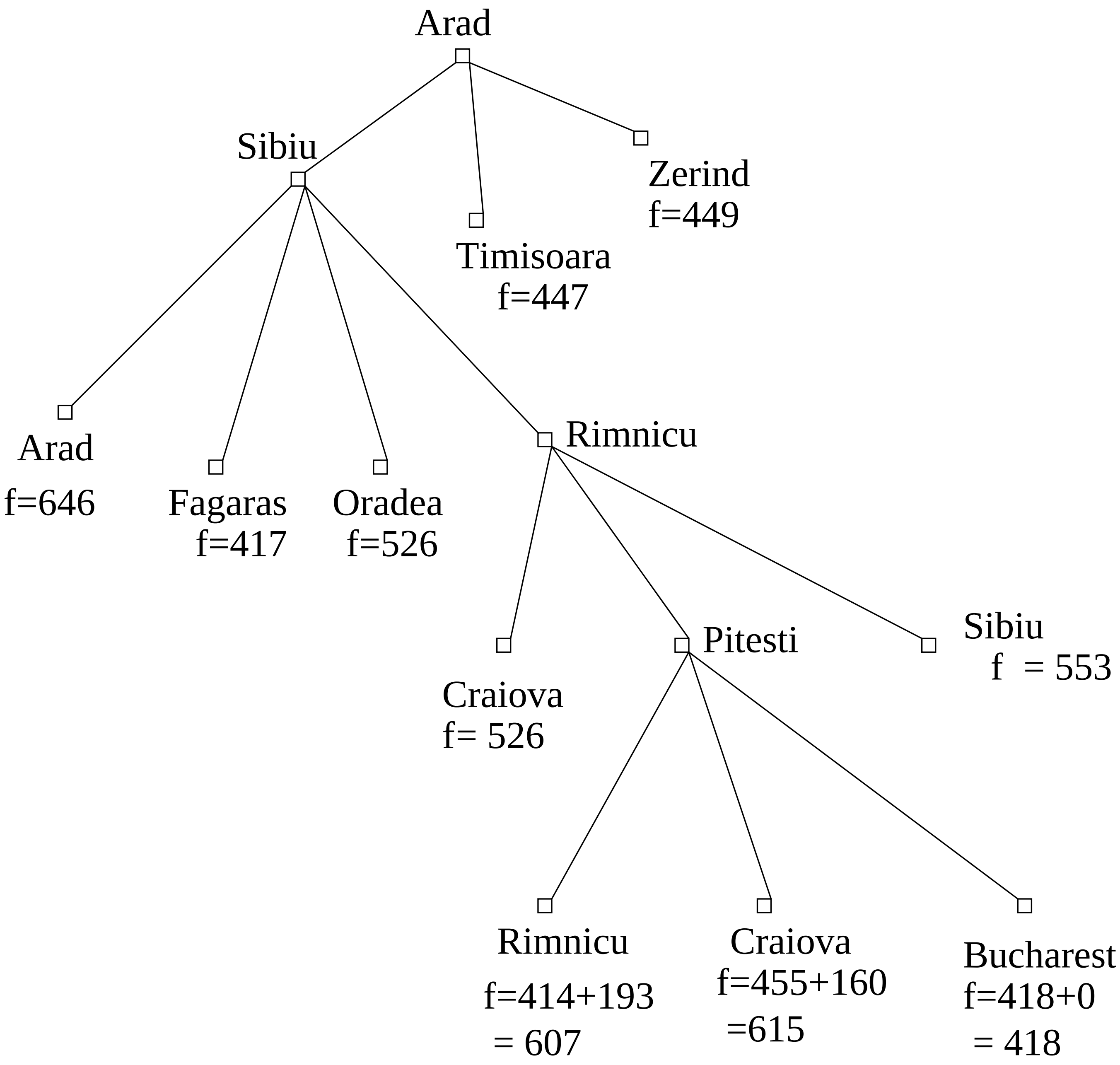
La ricerca guidata dal costo ($f(n) = g(n)$) è un caso particolare di **A*** con h tale che

$$\forall n \;\; h(n) = 0$$

Qui h è ammissibile: l'algoritmo di ricerca guidata dal costo è completo e ottimale

L'algoritmo A*: esempio




Arad
Sibiu
Zerind
f=449
Timisoara
f=447
Arad
f=646
Fagaras
f=417
Oradea
f=526
Rimnicu
Craiova
f= 526
Pitesti
Sibiu
f = 553
Rimnicu
f=414+193
= 607
Craiova
f=455+160
=615
Bucharest
f=418+0
= 418

Teoremi

Teorema: ottimalità di A*. Se h è ammissibile, allora **A*** è ottimale: se termina con soluzione P, allora P è un cammino ottimo

Teorema: completezza di A*. Se:

1. h è un'euristica ammissibile
2. il fattore di ramificazione dell'albero di ricerca del problema è finito, e
3. esiste δ positivo tale che il costo di ogni operatore è maggiore di δ

allora **A*** è completo

ESERCIZIO

Stati $= \{S_0, ..., S_6\}$

Stato iniziale $= S_0$

Goal-Test(s) $= s \in \{S_1, S_2\}$

Euristica tale che $h(S_0)$=20 $h(S_1)$=0 $h(S_2)$=0 $h(S_3)$=10
$h(S_4)$=9 $h(S_5)$=6 $h(S_6)$=12

Operatori = F avente costo 2 tali che
G avente costo 5
K avente costo 10

$$
\begin{array}{lllll}
F(S_0) = S_6 & F(S_1) = S_2 & F(S_3) = S_5 & F(S_4) = S_1 & \\
G(S_0) = S_3 & G(S_5) = S_4 & G(S_6) = S_0 & & \\
K(S_0) = S_4 & K(S_1) = S_0 & K(S_3) = S_1 & K(S_5) = S_2 & K(S_6) = S_3
\end{array}
$$

1. Disegnare lo spazio degli stati
2. Disegnare l'albero di ricerca per A* fino alla terminazione, indicando l'ordine di espansione dei nodi
3. Riportare la soluzione trovata da A*
4. Considerando l'albero costruito, è possibile determinare se h è ammissibile?

ESERCIZIO

Stati $= \{S_0, ..., S_6\}$

Stato iniziale $= \{S_0\}$

Goal-Test(s) $= s \in \{S_1, S_5\}$

Euristica tale che $h(S_0)$=20 $h(S_1)$=0 $h(S_2)$=4 $h(S_3)$=5
$h(S_4)$=6 $h(S_5)$=0 $h(S_6)$=3

Operatori = F avente costo 20, G avente costo 8, K avente costo 10 tali che

$$
\begin{array}{lll}
F(S_0) = S_1 & F(S_2) = S_5 & F(S_4) = S_5 \\
G(S_0) = S_2 & G(S_2) = S_4 & G(S_3) = S_1 \\
K(S_0) = S_3 & K(S_3) = S_6 & K(S_6) = S_2
\end{array}
$$

1. Disegnare lo spazio degli stati
2. Disegnare l'albero di ricerca per A* fino alla terminazione, indicando l'ordine di espansione dei nodi
3. Riportare la soluzione trovata da A*
4. Considerando l'albero di ricerca costruito, è possibile determinare se h è ammissibile?

FUNZIONI EURISTICHE

È importante determinare euristiche "informate".

Nella ricerca guidata dal costo, $h(n) = 0$ per ogni n: euristica a informazione nulla.

Esempio: il gioco dell'otto

- Rappresentazione degli stati: matrice 3×3 contenente i numeri da 1 a 8 e una "casella vuota".
- Stato iniziale: ad esempio

5	4	
6	1	8
7	3	2

- Goal test: vero soltanto per lo stato

1	2	3
4		5
6	7	8

- Operatori (tutti di costo unitario):
  - Spostare la casella vuota in alto
  - Spostare la casella vuota in basso
  - Spostare la casella vuota a destra
  - Spostare la casella vuota a sinistra

  (Esercizio: descriverne le condizioni di applicabilità e darne una "specifica")

Due euristiche per il gioco dell'otto

$h_1(n)$: numero di caselle fuori posto

$h_2(n)$: somma delle distanze di ogni casella dalla propria posizione finale

Per ogni n:

$$h_1(n) \leq h_2(n) \leq h^*(n)$$

h_2 è più informata di h_1

A* espande ogni nodo n con $f(n) < f^*(s_0)$, cioè ogni nodo n con

$$h(n) < f^*(s_0) - g(n)$$

Quindi ogni nodo espanso da A^* con h_2 è espanso anche con h_1

Per avere un'idea dell'importanza di una buona euristica:
confronto tra il costo della ricerca con iterative deepening search (IDS) e **A*** con le due euristiche:

d	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	364404	227	73

d	IDS	$A^*(h_1)$	$A^*(h_2)$
14	3473941	539	113
16	$--$	1301	211
18	$--$	3056	363
20	$--$	7276	676
22	$--$	18094	1219
24	$--$	39135	1641

Numero medio di nodi espansi, su 100 problemi

d = profondità della soluzione

Come inventare funzioni euristiche

h_1 è la h^* per il gioco dell'otto in cui ogni casella può saltare da una posizione a un'altra qualsiasi

h_2 è la h^* per il gioco dell'otto in cui le caselle possono spostarsi anche su caselle occupate

Regola del gioco: una casella si può spostare da A a B se:
A è adiacente a B, e
B è vuota
Relaxed problem : con minori restrizioni sugli operatori

Spesso, il costo effettivo di una soluzione in un relaxed problem è una buona euristica per il problema originario.

ABSOLVER (1993) genera euristiche automaticamente dalla definizione di un problema scritto in un linguaggio formale

- informazione statistica
- identificazione di caratteristiche di uno stato

Attenzione al costo computazionale della funzione euristica

Esempio: il mondo dei blocchi

Esempio:

stato iniziale: $\begin{array}{cc} \boxed{A} & \\ \boxed{B} & \boxed{C} \end{array}$ obiettivo: $\begin{array}{c} \boxed{A} \\ \boxed{B} \\ \boxed{C} \end{array}$

Regole:

- si può spostare un solo blocco alla volta: un blocco X si può mettere sopra un blocco Y o sul tavolo solo se sopra X non c'è alcun blocco
- non si può mettere un blocco sopra se stesso
- non si può mettere un blocco sopra un blocco già occupato
- un blocco libero si può sempre mettere sul tavolo (sul tavolo c'è sempre spazio a sufficienza)

Rappresentazione di problemi nel mondo dei blocchi

Rappresentazione degli stati: insiemi di sequenze (liste), dove ciascuna sequenza rappresenta la torre che ha in cima il primo elemento della sequenza, sopra al secondo ...
Esempi: $\{(A, B), (C)\}$, $\{(A, B, C)\}$, $\{(A), (B), (C, D)\}$

Stato iniziale (dell'esempio): $\{(A, B), (C)\}$

Goal test (dell'esempio): GOAL-TEST(S) $= (A, B, C) \in S$

Il mondo dei blocchi (2)

Operatori:

- $put\_on(b_1, b_2)$

  Condizioni di applicabilità:
  $put\_on(b_1, b_2)$ è applicabile a uno stato S se $b_1 \neq b_2$ e S contiene due sequenze che hanno, rispettivamente, b_1 e b_2 come primo elemento.
  Cioè S deve avere la forma:
  $\{(b_1, x_1, ..., x_n), (b_2, y_1, ..., y_m), t_1, ..., t_k\}$, con $n, m, k \geq 0$.

  Applicazione dell'operatore:
  $put\_on(b_1, b_2)$, applicato allo stato $S = \{(b_1, x_1, ..., x_n), (b_2, y_1, ..., y_m), t_1, ..., t_k\}$, riporta lo stato S' con

$$S' = \begin{cases} \{(x_1, ..., x_n), (b_1, b_2, y_1, ..., y_m), t_1, ..., t_k\} & \text{se } n > 0 \\ \{(b_1, b_2, y_1, ..., y_m), t_1, ..., t_k\} & \text{se } n = 0 \end{cases}$$

- $put\_on\_table(b)$

  Condizioni di applicabilità:
  $put\_on\_table(b)$ è applicabile a uno stato S se S contiene una sequenza della forma $(b, x_1, ..., x_n)$ con $n > 0$.

  Applicazione dell'operatore:
  $put\_on\_table(b)$, applicato allo stato $S = \{(b, x_1, ..., x_n), t_1, ..., t_k\}$, riporta lo stato $S' = \{(b), (x_1, ..., x_n), t_1, ..., t_k\}$

Costo degli operatori: ogni operatore ha costo 1.

Il mondo dei blocchi (3)

Dato un problema del mondo dei blocchi e uno stato S, diciamo che un blocco X è "fuori posto" in S se X si trova sopra un blocco Y e l'obiettivo richiede che X si trovi su un blocco diverso o sul tavolo oppure se in S X sta sul tavolo e l'obiettivo richiede che X non sia sul tavolo.

I blocchi "superficiali" in uno stato S sono i primi elementi delle sequenze in S.

Esercizio: Si considerino le seguenti

Euristiche:

$h_1(S) =$ numero di blocchi superficiali fuori posto
$h_2(S) =$ numero di blocchi fuori posto

1. h_1 e h_2 sono euristiche ammissibili? Qual è quella più informata?
2. Risolvere il problema dell'esempio applicando A*, utilizzando prima h_1 e poi h_2.
3. Può essere una buona scelta un'euristica definita da
   $h_3(S) =$ numero di blocchi nella posizione giusta?

Iterative Deepening A* (IDA*)

La ricerca per approfondimenti successivi riduce lo spazio di memoria utilizzato.

IDA* applica A* per approfondimenti successivi, ponendo come limite, anzichè la profondità, un valore di f.

Inizialmente il limite è $f\_limit = f(s_0)$

Ogni iterazione è una ricerca in profondità che espande solo i nodi n con $f(n) \leq f\_limit$

Terminata un'iterazione, si calcola il nuovo f_limit: per la successiva iterazione f_limit sarà il minimo $f(n)$ tra i nodi n generati ma non espansi

Completo e ottimale sotto le stesse ipotesi di **A***

Spazio: proporzionale al ramo più lungo costruito $O(b \cdot d)$
d: profondità minima di una soluzione

Tempo: Dipende fortemente dal numero di valori differenti che può assumere h: se h assume un numero ristretto di valori (come ad esempio nel gioco dell'otto), normalmente f aumenta poche volte lungo un cammino, dunque IDA* esegue poche iterazioni.

Se f cambia spesso, può succedere che ogni iterazione esamini soltanto un nuovo nodo: se A* espande N nodi, IDA* espande $1 + 2 + ... + N = O(N^2)$ nodi!

Per problemi di questo tipo: si aumenta f_limit di una quantità fissa ϵ. La soluzione è sub-ottima: il suo costo può essere al massimo maggiore di ϵ rispetto al valore ottimo (algoritmo **ϵ-ammissibile**).

Algoritmo IDA*

```
function IDA* (problem) returns a solution sequence
  local variables; f-limit, il limite attuale di f-COST
                   root, un nodo
  root <- MAKE-NODE(INITIAL_STATE(problem))
  f-limit <- f-COST(root)
  loop do
    solution, f-limit <- DFS-CONTOUR(root,f-limit)
    if solution is non-null, then return solution
    if f-limit = infinity then return failure
  end

function DFS-CONTOUR(node,f-limit)
  returns a solution sequence and a new f-COST limit
  local variables: next-f, the f-COST limit of the next
                           contour, initially infinity
  if f-COST(node) > f-limit then return null, f-COST(node)
  if GOAL-TEST(problem)(STATE(node)) then return node, f-limit
  for each node s in SUCCESSORS(node) do
     solution,new-f <- DFS-CONTOUR(s,f-limit)
     if solution is non-null then return solution,f-limit
     next-f <- MIN(next-f, new-f)  end
  return null, next-f
```