Agenti basati su Ricerca: Introduzione

Intelligenza Artificiale

Prof. Alfonso E. Gerevini
Dipartimento Ingegneria dell'Informazione
Università degli Studi di Brescia

Un Agente “Risolutore di Problemi”

Segue un processo ciclico composto da 4 fasi:

- **Formulazione dell’obiettivo** (goal). Ad es: essere in una certa città
- **Formulazione del problema (o dominio)**
  - Stati e azioni (NB: stato ricerca = insieme di stati “reali”)
  - È importante una appropriata *astrazione*
- **Ricerca**: trovare una sequenza di azioni che porta dallo stato iniziale a quello goal (algoritmo di ricerca)
- **Esecuzione** della sequenza di azioni (o della prima azione)

Un Agente “Risolutore di Problemi”

Definizione di un Problema di Ricerca

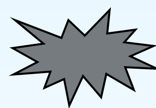
Ambienti deterministici e accessibili (completamente osservabili):

- Insieme di **stati** del mondo (ambiente) semplificato
- Insieme di **operatori** (o azioni) possibili: **una funzione successore (**oppure **azioni + modello transizione)**
- Uno **stato iniziale** in cui l'agente sa di essere
- Uno o più **stati obiettivo: un goal test**
- Un costo per ogni sequenza di azioni (sequenza di stati attraversati): **una funzione costo**

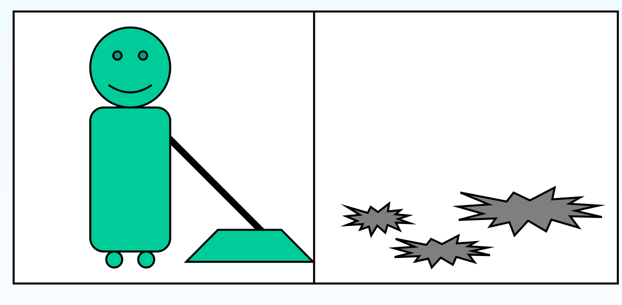
$\Rightarrow$ *Trovare una sequenza di azioni (cammino tra stati) che trasforma lo stato iniziale in uno stato obiettivo (algoritmo o motore di ricerca)*

Problema a Stato Unico

Un semplice esempio (Roby = robot aspirapolvere):

Ambiente: 2 stanze, Roby, polvere. 

Esempi di info irrilevante?

Stato iniziale:  **Num. stati possibili** = 8

stanza1 stanza2

Azioni: VaiDestra, VaiSinistra, Aspira

Stati Obiettivo = tutti gli stati in cui non c'è polvere

Soluzione = VaiDestra, Aspira

Diagramma degli stati

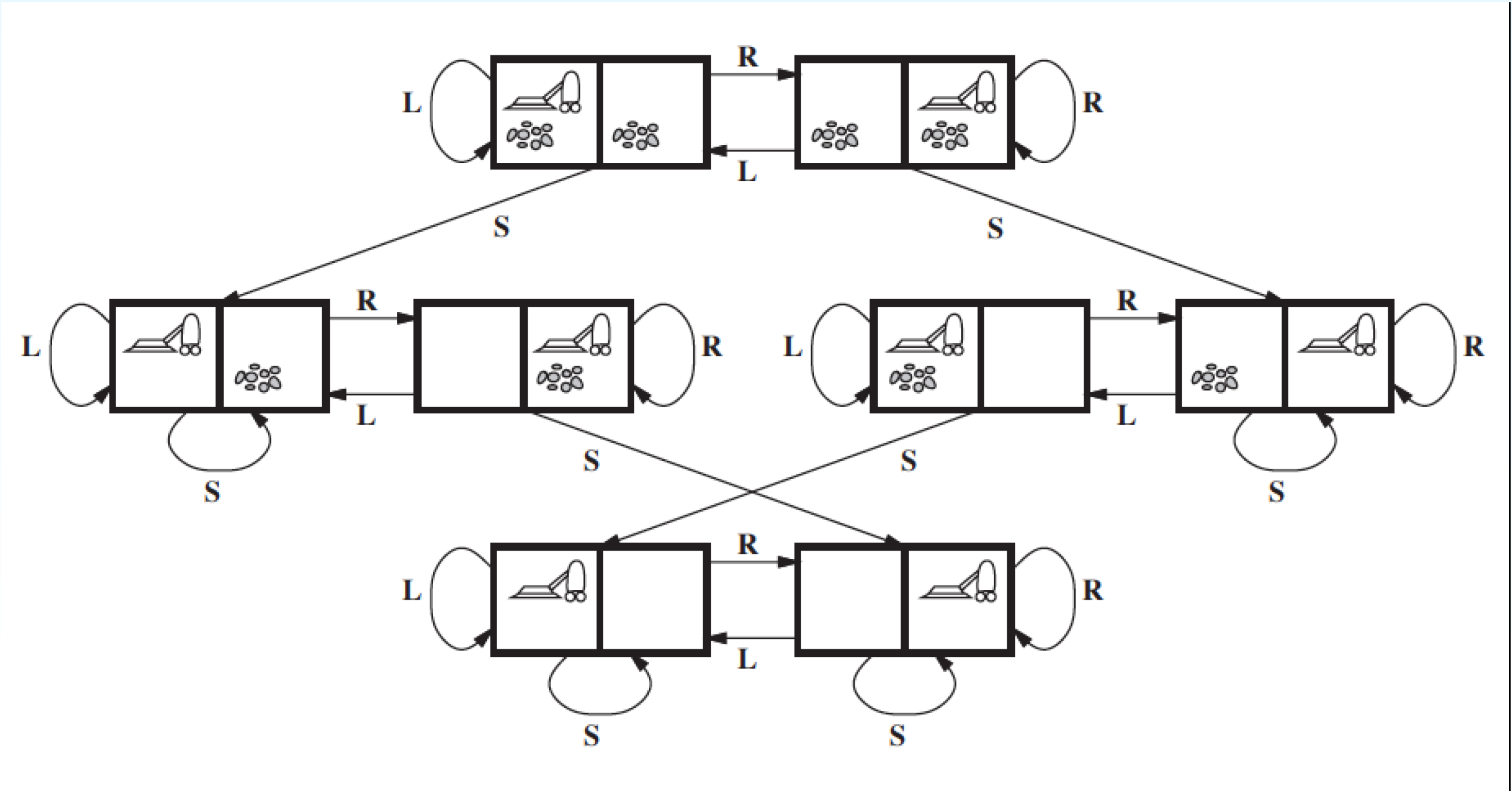
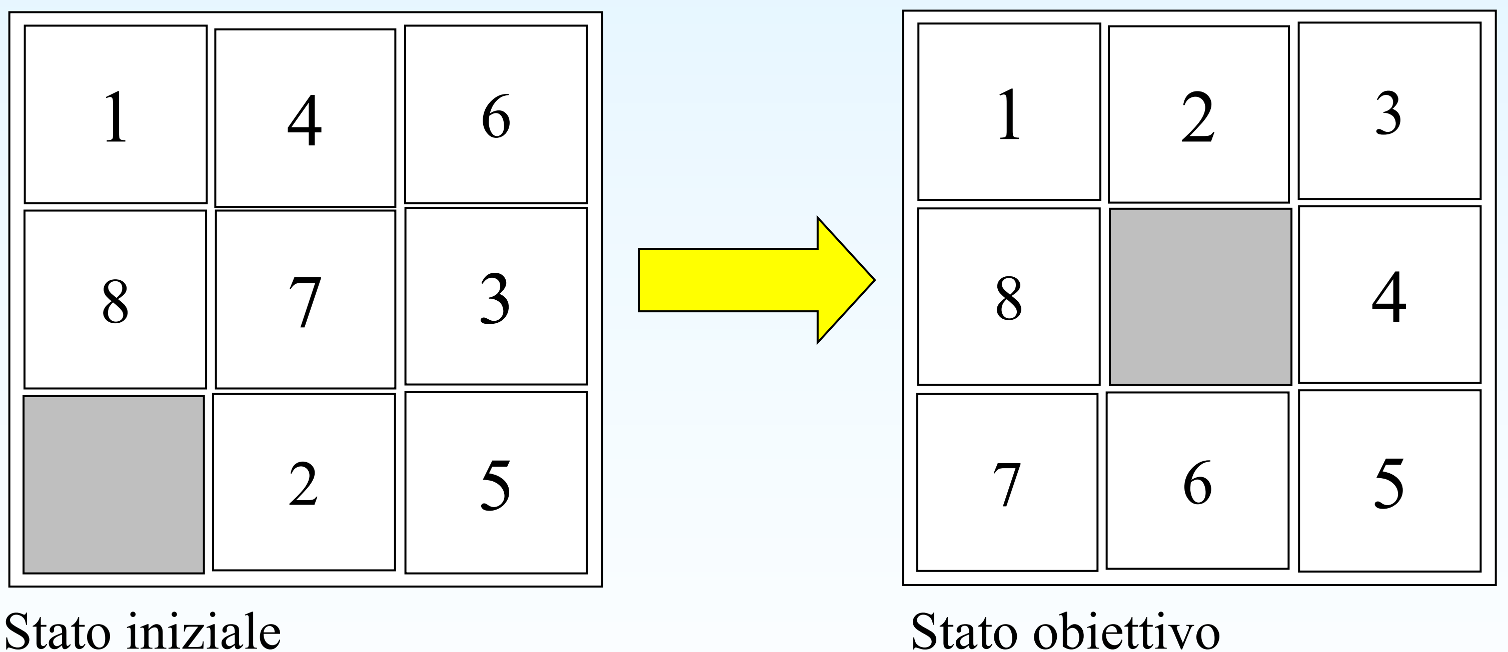


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Esempio: “Rompicapo dell’8”



Stati = configurazione delle caselle

Info irrilevanti: colore, dimensione, peso, materiale, ecc.

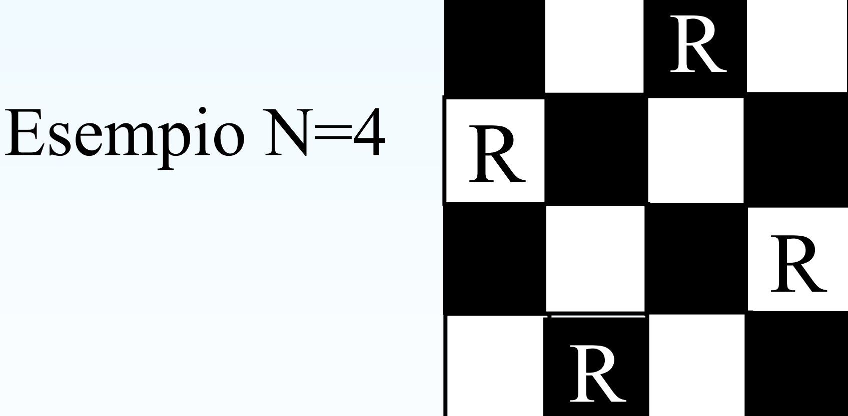
Operatori = muovere ciascuna casella in alto, basso, sinistra, destra (oppure muovere casella vuota; **è meglio???**)

Quanti stati (possibili) ?

- Stato possibile = stato “raggiungibile” attraverso una sequenza di operatori
- 8 tasselli: 181440 (molto facile)
- 15 tasselli: 1.3×10^{12} (facile)
- 24 tasselli: 10^{25} (molto difficile)

Esempio: “Problema delle N Regine”

Problema: *Disporre N regine su una scacchiera NxN senza che queste si attacchino*. NB: Mi interessa trovare uno stato con determinate proprietà e non una sequenza di azioni.



Stati = configurazioni di una o più regine sulla scacchiera

Stato iniziale = nessuna regina sulla scacchiera

Stati obiettivo = N regine sulla scacchiera e nessun attacco

Problema delle N Regine

Operatori (2 possibilità):

- mettere una regina qualsiasi sulla scacchiera (8x8)

  se 8 regine, 64^8 sequenze di azioni (circa)

- regina i-esima sulla colonna i-esima (senza attacco alle precedenti)

Ma il problema resta non banale per n grande (se n=100, ho 10^{52} possibili sequenze!)

Problemi di ricerca e Automi

- Che relazione c’è tra un problema di ricerca ben definito e una Automa a stati finiti?

Problema a Stati Multipli

Spesso l'ambiente non è completamente accessibile

<u>*Caso limite*</u>: completa ignoranza (no sensori)

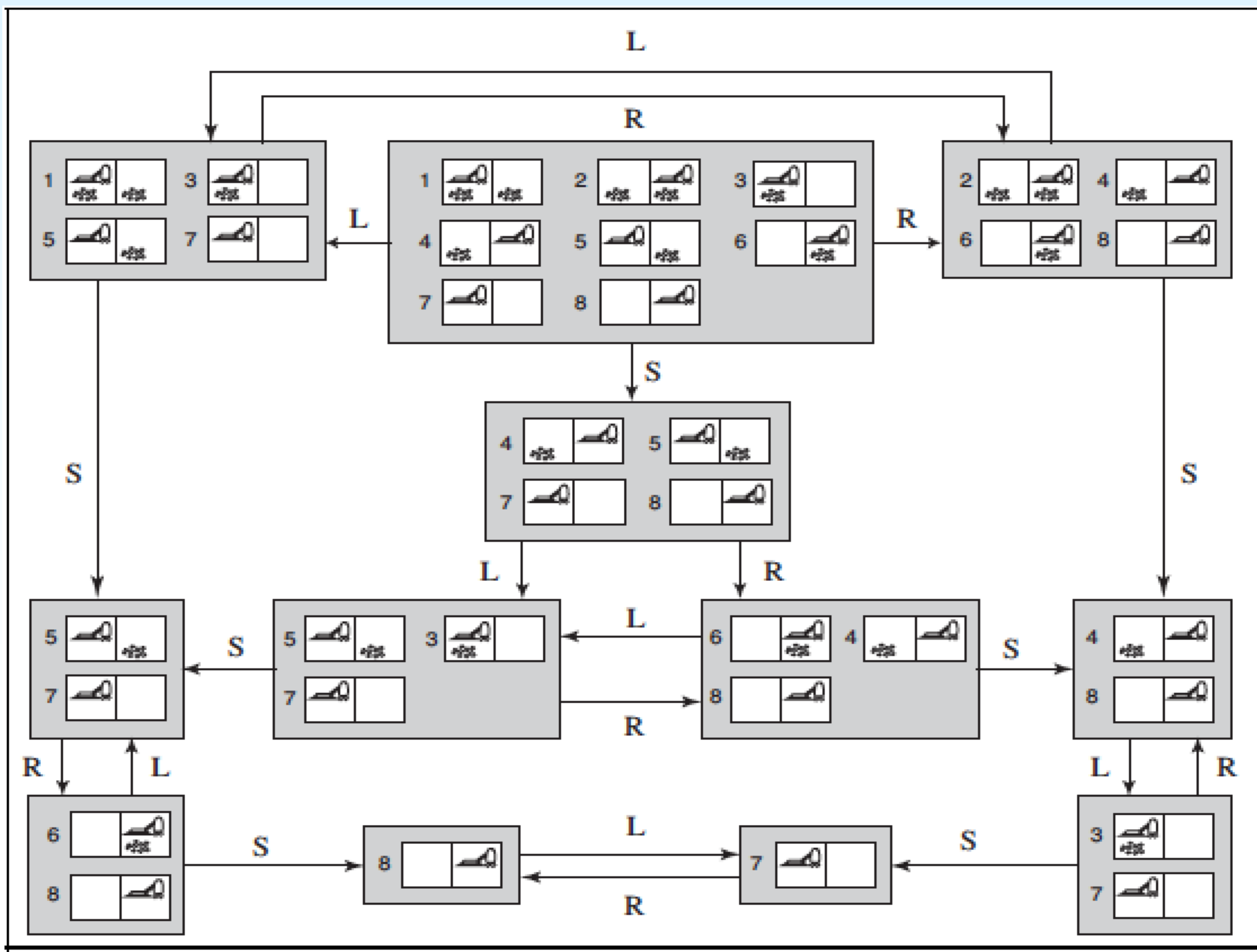
Esempio robot aspirapolvere:

Stato iniziale: uno imprecisato degli 8 possibili

Ogni azione trasforma un belief state in un altro

Soluzione: Trasforma l'insieme iniziale di stati in un insieme contenente solo soluzioni (in questo caso un solo stato):

Diagramma degli stati



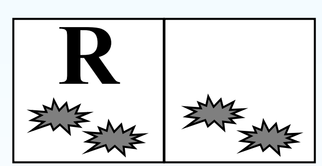
Problema con Contingenze

Problemi in cui la soluzione deve contenere azioni "sensoriali" per acquisire informazioni necessarie

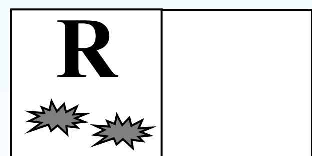
Esempio robot aspirapolvere

- Ambiente parzialmente accessibile (2 sensori)

Stato Iniziale:



oppure



- Informazioni incompleta su effetti azioni (non determinismo):

  Aspira *talvolta deposita polvere se non c'è*

Soluzione: **??? Aspira-Destra-Aspira ???**

E' necessaria un'azione sensoriale per sapere se c'è polvere!

Formalizzare un problema di ricerca

- Un' "arte" in cui si cerca un buon **livello di astrazione** per modellare il problema:
  - Informazioni irrilevanti sugli stati (limitare il numero di stati)
  - Informazioni (precondizioni/effetti) irrilevanti su azioni
  - Azioni irrilevanti
- Esercizio di formalizzazione: il problema dei **Missionari e Cannibili** (Amarel 1968)

  http://www.plastelina.net/game2.html

  - Quali informazioni sono rilevanti e quali irrilevanti?
  - Stati?
  - Operatori?
  - Goal test?
  - Costo soluzione?
- **Come risolvereste "a mano" questo problema di ricerca?**

Esercizio (per casa)

- Formalizzare il gioco “Family crisis“ disponibile on-line: http://www.plastelina.net/game3.html

https://archive.org/details/family-crisis

Algoritmi di Ricerca

Ragionamento come esplorazione dello **spazio degli stati** possibili

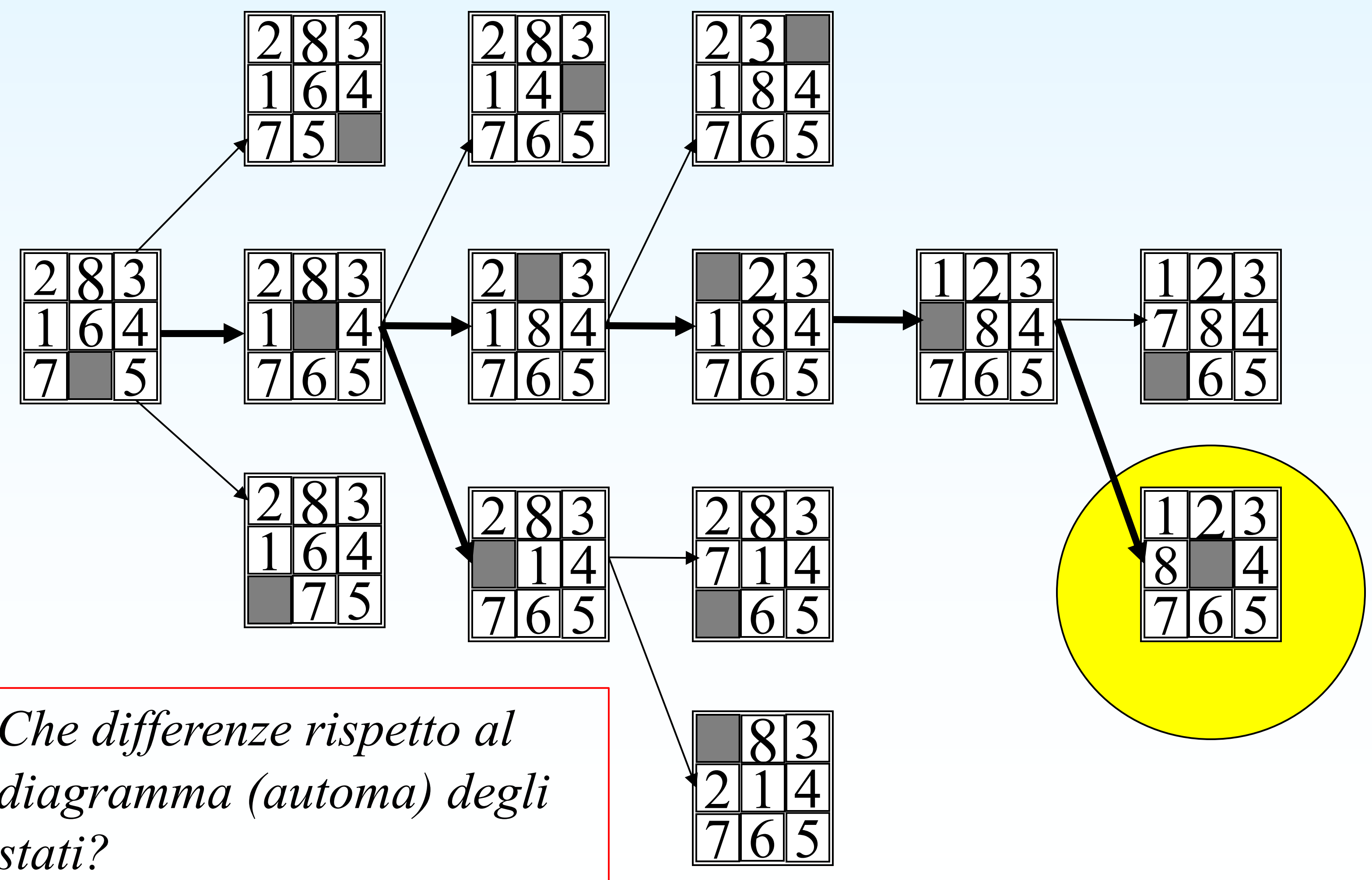
Qualità Soluzione (per scegliere tra più possibili)

- Tempo richiesto per derivarla: **costo off-line**
- Numero e "costo" azioni (mosse): **costo on-line**

Concetti e definizioni preliminari

- Spazio degli stati, transizioni possibili, stati raggiungibili
- Albero di ricerca
- Espansione di un nodo
- Profondità di un nodo
- Fattore di diramazione (branching factor)
- Ricerca come esplorazione dell'albero
- Algoritmo generale di ricerca/esplorazione
- Strategie di esplorazione: cieche e euristiche (esempi per il "rompicapo dei tasselli")
- Proprietà fondamentali di una strategia/algoritmo di ricerca: correttezza, complessità (tempo e spazio), ottimalità

Esempio Albero di Ricerca (*parziale*)



Dimensione albero di ricerca a stati: Rompicapo dell'8

- Fattore diramazione medio: 3
- Costo medio soluzione problema random: 22
- Nodi visitati da ricerca esaustiva con profondità 22: 3^{22}
- Se uso memoria per evitare ripetizioni = 180000 nodi (stati differenti)

Dimensione albero ricerca e stati: rompicapo del 15

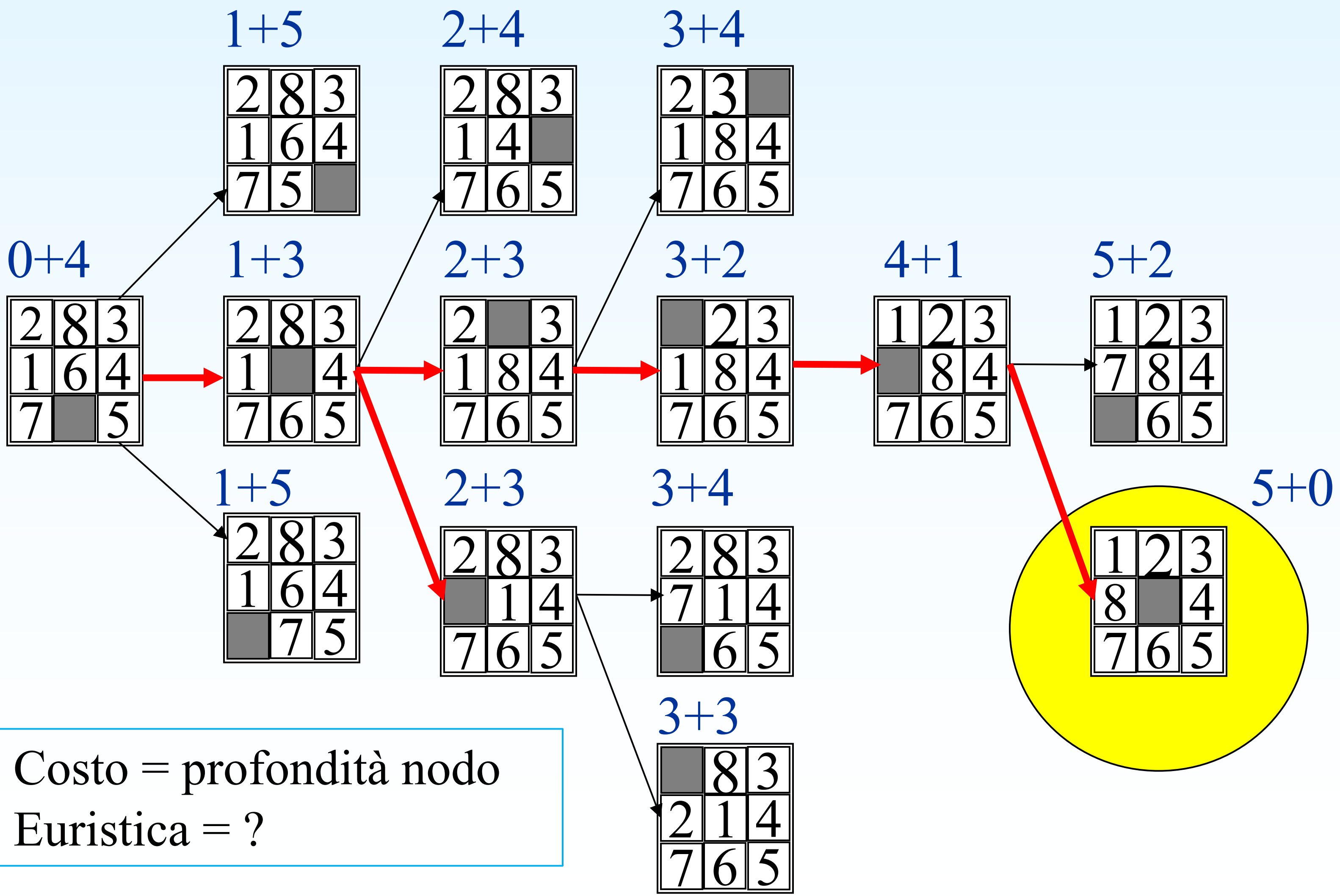
- Fattore diramazione medio: 3
- Costo medio soluzione problema random: molto maggiore di 22
- Stati differenti: circa 10^{13}

Algoritmo Generico di Ricerca

1. L = lista dei nodi/stati iniziali del problema
2. IF L è vuota THEN return 'fail'
   ELSE N = *ScegliNodo(L)*
3. IF GoalTest(N) THEN return N e cammino da nodo iniziale
4. L = L – N + *AggiungiFigli(N,L)*
5. GOTO 2.

- L rappresenta la *lista dei noti/stati da visitare/espandere;* i nodi in L sono *etichettati con il relativo cammino dalla radice*
- **ScegliNodo** e **AggiungiFigli** sono funzioni da specializzare
- **Strategie cieche**: ScegliNodo dipende solo dalla posizione di N
- **Strategie informate**: ScegliNodo dipende anche da informazioni specifiche dal dominio del problema (**euristica**)

Esempio Albero di Ricerca (con costo + euristica sui nodi)



Ricerca in profondità e ampiezza

Profondità: il prossimo nodo da esaminare è un figlio dell'ultimo visitato (se esiste)

Ampiezza: i nodi a profondità i devono essere visitati subito dopo aver visitato tutti i nodi a profondità i -1 (con $i > 0$)

– Come implementare queste ricerche?
– Esempi di visita alberi di ricerca
– Complessità degli algoritmi

Esempi tempo e memoria ricerca in ampiezza

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

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Altri quattro algoritmi non informati

- A profondità limitata
- A profondità iterativa (iterative deepening)
- Bidirezionale
- A costo uniforme

Iterative Deepening Search

1. c = 1, Stop = true
2. L = lista formata dal nodo (stato) iniziale
3. IF L è vuota THEN

   IF Stop == false THEN

   c = c + 1, Stop = true, GOTO 2

   ELSE Return('fail')
4. N = First-node(L)
5. IF Goal-test(N) THEN Return(N) ELSE L = L – N
6. IF Depth(N) < c THEN

   L = Aggiungi-davanti(Figli(N),L)

   ELSE IF |Figli(N)| > 0 THEN Stop = false
7. GOTO 3.

Esempi Ricerca a Costo Uniforme

Albero determinato da (G nodo goal):
(s,1,a) (s,5,b) (s,15,c) (a,10,G) (b,5,G) (c,5,G)

Albero determinato da (G nodo goal):
(s,5,a) (s,8,b) (a,1,c) (c,1,d) (d,1,G) (d,-2,a) (b,1,e)

Proprietà algoritmi di ricerca standard non informati

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^a	Yes^{a,b}	No	No	Yes^a	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^c	Yes	No	No	Yes^c	Yes^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

Evitare di visitare stati già visitati

- Quelli *correntemente* nella lista L
- Quelli sul cammino del nodo appena visitato
- Tutti quelli già incontrati

Come implementarli?

Quali sono i pro e contro dei tre metodi?

Rispondere considerando

- Ricerca in profondità
- Ricerca in ampiezza
- Ricerca a costo uniforme

Esempio problemi in cui sono utili (Griglie)

Best-first search

Considero sia il costo per arrivare a un nodo **n** sia il costo stimato per arrivare a un nodo goal da **n**

$$\mathbf{f(n) = g(n) + h(n)}$$

- Lista L ordinata per f(n)=g(n)+h(n) crescente
- g(n) = costo miglior percorso trovato dalla radice a n
- h(n) = funzione euristica che stima la distanza di n al nodo goal più vicino (minore profondità) a n

A cosa serve h(n)?

Per arrivare prima al nodo goal (visitando meno nodi/stati).
Se e h(n) fosse perfetta ci arriverei subito…

Esempi alla lavagna

Best-first search e A*

- Lista L ordinata per $f(n)=g(n)+h(n)$ crescente
- $g(n)$ = costo miglior percorso trovato dalla radice a n
- $h(n)$ = funzione euristica che stima la distanza di n al nodo goal più vicino (minore profondità) a n
- Funzione **h(n) ammissibile**: sia $h^*(n)$ la reale distanza di n al nodo goal più vicino, diciamo che $h(n)$ è ammissibile se è *sempre ottimista*, ovvero

  Per ogni n, $h(n) <= h^*(n)$

A* = Best-first search con h(n) ammissibile

Ottimalità di A*

- G è nodo goal ottimo → $f(G) = g(G) = f^*$
- G2 è nodo goal subottimo → $f(G2) =$ **$g(G2) > f^*$**
- $h(G) = h(G2) = 0$
- S = stato iniziale

(1) $f^* < g(G2)$ perché $f^* < f(G2)$

Supponiamo per assurdo G2 selezionato prima di G da L
→ Esiste un nodo n sul cammino ottimo da S a G non ancora visitato che è in L quando G2 viene selezionato da L

(2) $f^* >= f(n)$ (h è ammissibile)
(3) $f(n) >= f(G2)$ (G2 è preferito a n)
(4) $f^* >= f(G2)$ (combinando 2 e 3)
(5) $f^* >= g(G2)$!! (assurdo)

Ottimalità di A*

Assunzione sottostante: fattore diramazione è finito!

E se ci fossero azioni con costi nulli o negativi (guadagni)?

Progettare euristiche

Le euristiche sono molto, molto, molto, ….
importanti!

The possibili approcci generali:

1. Metodo del **problema rilassato**
2. Metodo del **sottoproblema**

Differenti euristiche possono essere combinate

Progettare euristiche: problema rilassato

Trovo la soluzione di un rilassamento del problema e uso il costo di tale soluzione come valutazione euristica del problema originale

Esempi nel rompicapo dell'8

1. Numero caselle mal poste nella configurazione corrente
2. Somma della distanza di Manhattan delle caselle nella configurazione corrente rispetto alla posizione finale

Perché 1 e 2 sono soluzioni di un rilassamento del problema originale?

Se la soluzione del problema rilassato è <u>**ottima**</u> l'euristica è ammissibile!

Progettare euristiche: tecnica dei sottoproblemi

- Considero il problema originale come un insieme di sotto-problemi (tipicamente non indipendenti!)
- Uso il costo della soluzione di un sotto-problema come euristica del problema originale
- *Se la soluzione del sotto-problema è ottima, l'euristica è ammissibile*

Esempi alla lavagna con rompicapo dell'8

L'euristica dei **pattern database** memorizza in un database il costo ottimo per tutti i sotto-problemi identificati da un 'pattern' (nel rompicapo dell'8 è la posizione di un sottoinsieme di caselle)

Esempi alla lavagna…

Progettare euristiche: Combinazione di più euristiche

- Se ho più euristiche h1, h2, .., hk le posso combinare e usare insieme. Ma come?

- Il **Min** dei loro valori? Non ha molto senso…

- La **Somma** dei loro valori? Non è un'euristica ammissibile

- Il **Max** dei loro valori? È un'euristica ammissibile se tutte le singole euristiche sono ammissibili

- Ci sono altre combinazioni *più potenti e ammissibili*?

Si, ad es. l'euristica dei **pattern database disgiunti**

Progettare euristiche: Pattern database disgiunti

h-p1: euristica per pattern database caselle 1,2,3 (pattern p1)
h-p2: euristica per pattern database caselle 4,5 (pattern p2)
h-p3: euristica per pattern database caselle 6,7,8 (pattern p3)

h-pd = hp1'+ hp2' + hp3' **Pattern Database disgiunti**

Con hp1'/p2'/p3' = numero di mosse nella soluzione ottima per il pattern p1/p2/p3 **scontato** dal numero di mosse/azioni che **non** riguardano il pattern p1/p2/p3

SE soluzione per p1 = 1right, 6(*)up, 2down, 5(*)left, 3right
la soluzione ha costo originale 5 e scontato 3 → **hp1' = 3**

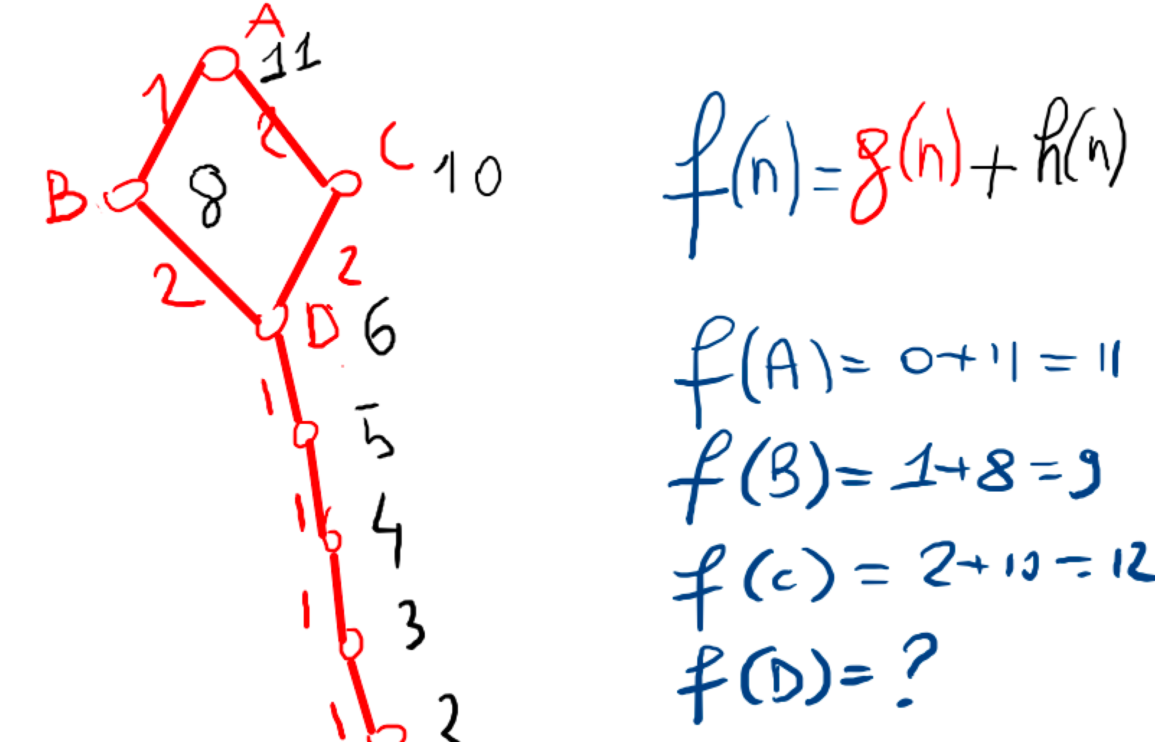
h(n) Consistente (o Monotona)

- h(n) è consistente o monotona se vale:

  per ogni n, a **h(n) <= c(n,a,n') + h(n')**

- c(n,a,n') = costo dell'azione a applicata nello stato di n ottenendo uno stato rappresentato dal nodo successore n'
- E' una forma di disuguaglianza triangolare
- Garantisce h(n) *non* decrescente
- A* con h(n) monotona + lista CLOSED garantisce **ottimalità per grafi di ricerca**
- Teorema: ogni funzione monotona è ammissibile; (NON vale viceversa!)

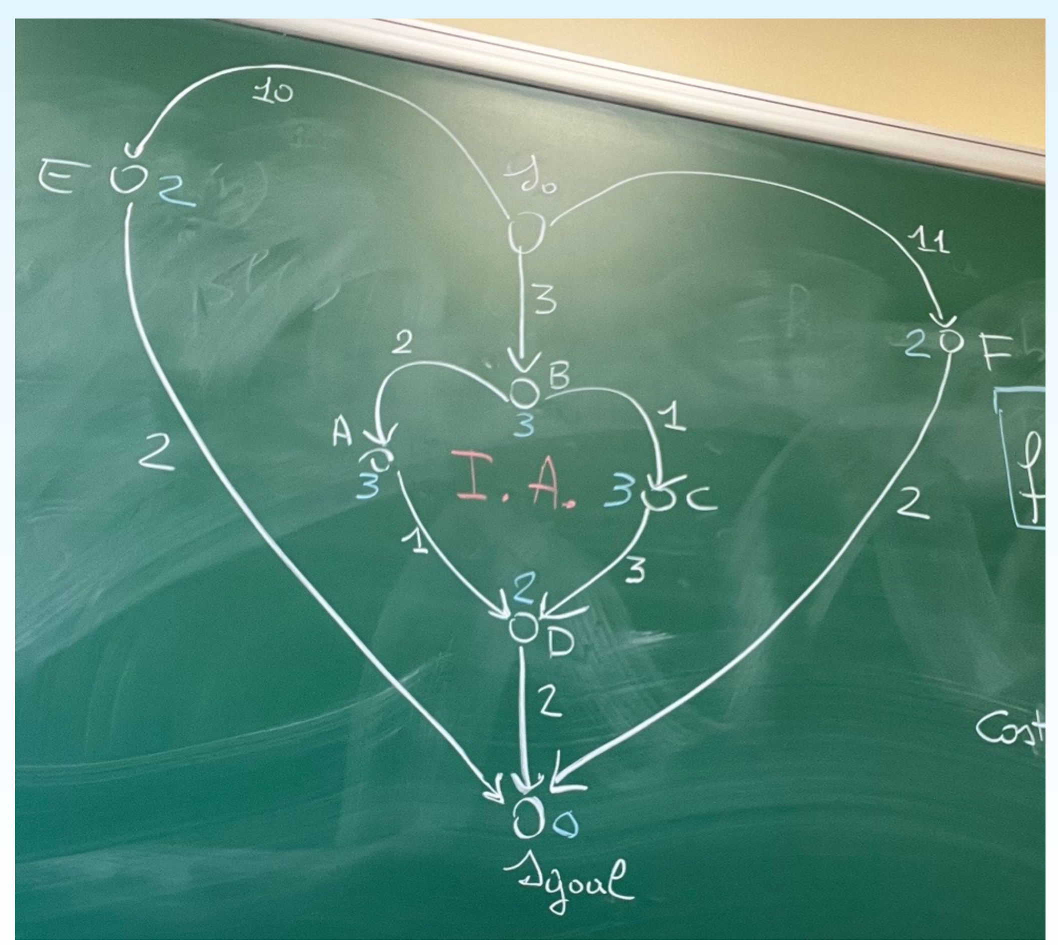
Esempio ‘aquilone’



E se costo da A a B = 2
e costo da A a C = 1
quanti nodi si visitano?
Ci sono nodi con h
inconsistente? Quali?

Quanti nodi vista A*
h è ammissibile?
h è consistente?

Esempio ‘cuore’



H(n) è ammissibile?
Consistente?
Scrivere i valori di N, Open e Closed ad ogni iterazione di A* (stato iniziale s_o, gola s_goal)

Confrontare Euristiche: metodo teorico

- h1: numero di caselle malposte
- h2: somma distanza di Manhattan delle caselle

  E’ meglio h1 o h2??

Confronto teorico per due euristiche ammissibili h1 e h2:

h2 migliore di h1 (h2 domina h1) se e solo se per ogni nodo/stato n, h1(n) <= h2(n)

Non sempre facile da dimostrare formalmente!

Confrontare Euristiche: metodo sperimentale

– N= numero reale nodi generati da A* in un run

– d = profondità dove A* trova nodo goal in un run

– N+1 = *numero nodi in un albero uniforme con fattore di diramazione b* e profondità d*:

$$N+1 = 1 + b^* + (b^*)^2 + \ldots\ldots + (b^*)^d$$

– b* = incognita = *fattore di diramazione effettivo*

– Calcolo b* facendo una media su molti run

Confrontare Euristiche: metodo sperimentale (esempio)

Se A* trova soluzione con $d = 5$ e $N = 52$
$b^* = 1{,}92$

Se l'euristica è buona b* è vicino a 1

Preferisco euristiche con b* più basso

Esempio h1 e h2 per rompicapo dell'8

Confrontare Euristiche: metodo sperimentale (rompicapo dell'8)

caselle sbagliate e la h2 e
quella che conta la DM

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A^* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

Iterative deepening A*/BFS

Best-first search ha complessità esponenziale (sia tempo che spazio)

→ La troppa memoria usata può far fallire la ricerca

Possiamo affrontare questo problema rendendo la ricerca iterativa come in iterative deepening search?

A* ad approfondimento iterativo (IDA*)

Come decido il taglio C della ricerca ad ogni iterazione?

C iniziale = f(nodo radice),

C successivo = **valore minimo di f per un nodo generato che supera il C corrente**

Ricerca locale: l'algoritmo più semplice

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
     neighbor ← a highest-valued successor of current
     if neighbor.VALUE ≤ current.VALUE then return current.STATE
     current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Ricerca locale: difficoltà

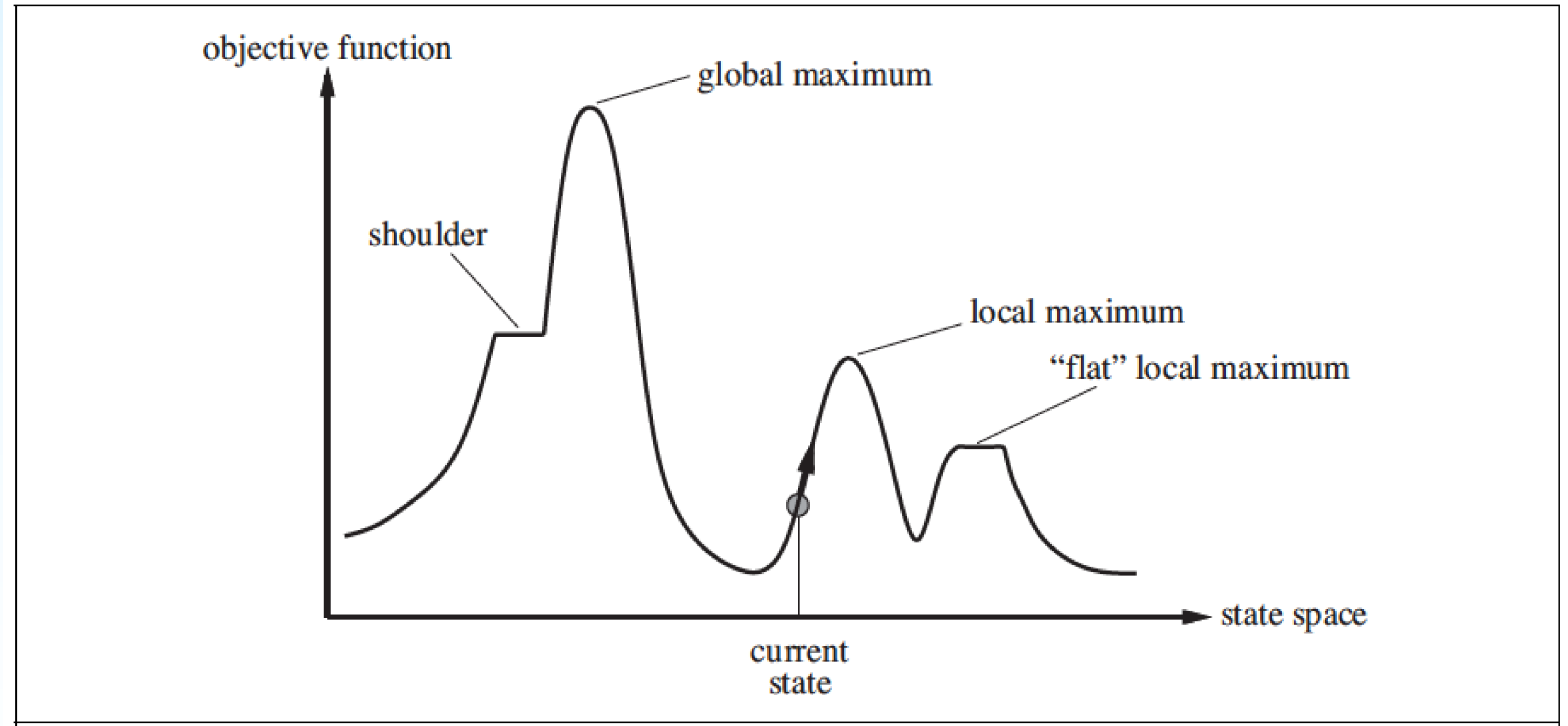
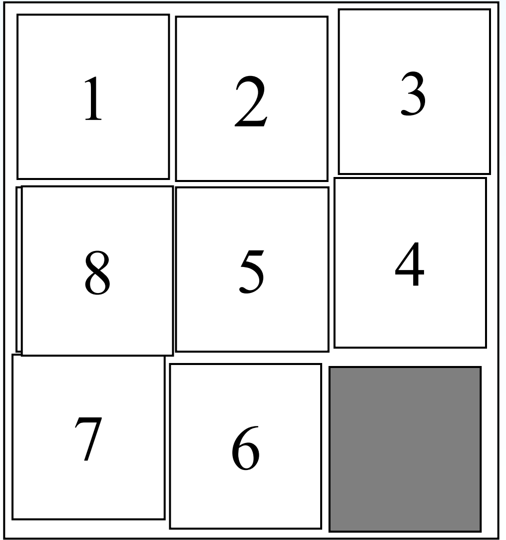


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

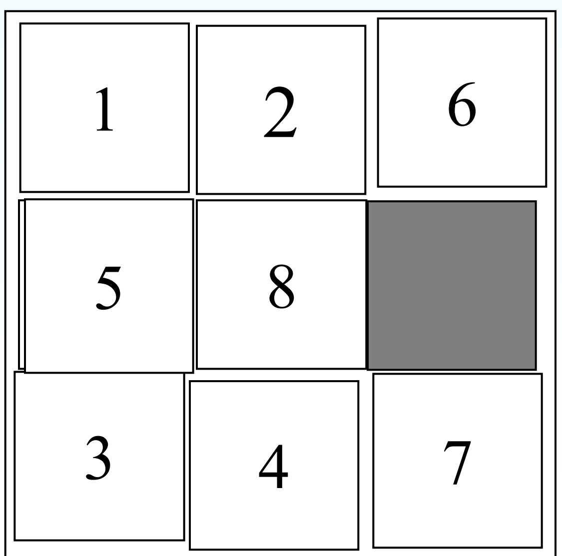
Ricerca locale: esempi stati problematici

h(n): caselle malposizionate in n

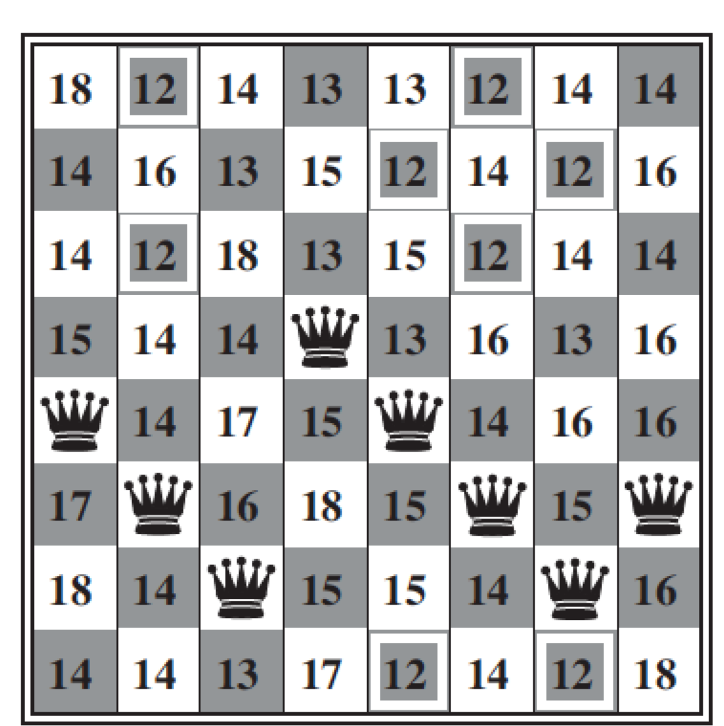
Minimo locale



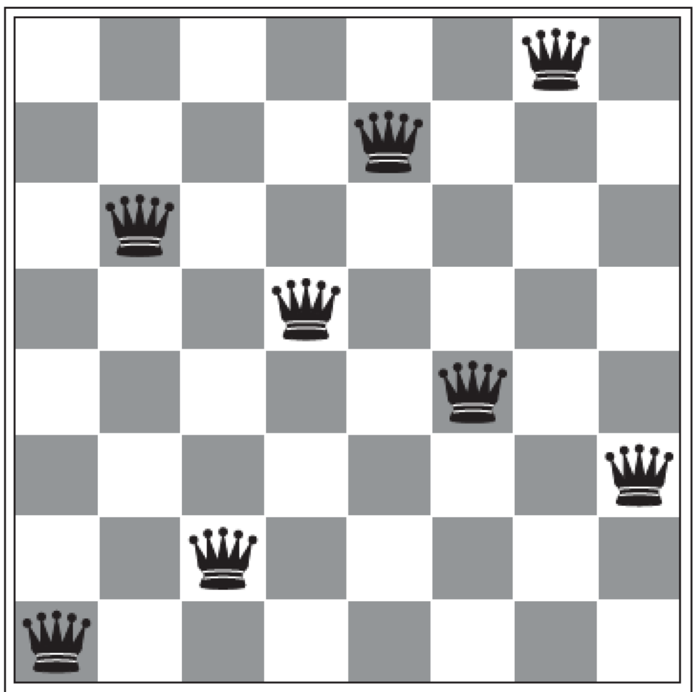
Pianura (plateaux)



Ricerca locale: minimo locale?



(a)



(b)

Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

Ricerca locale

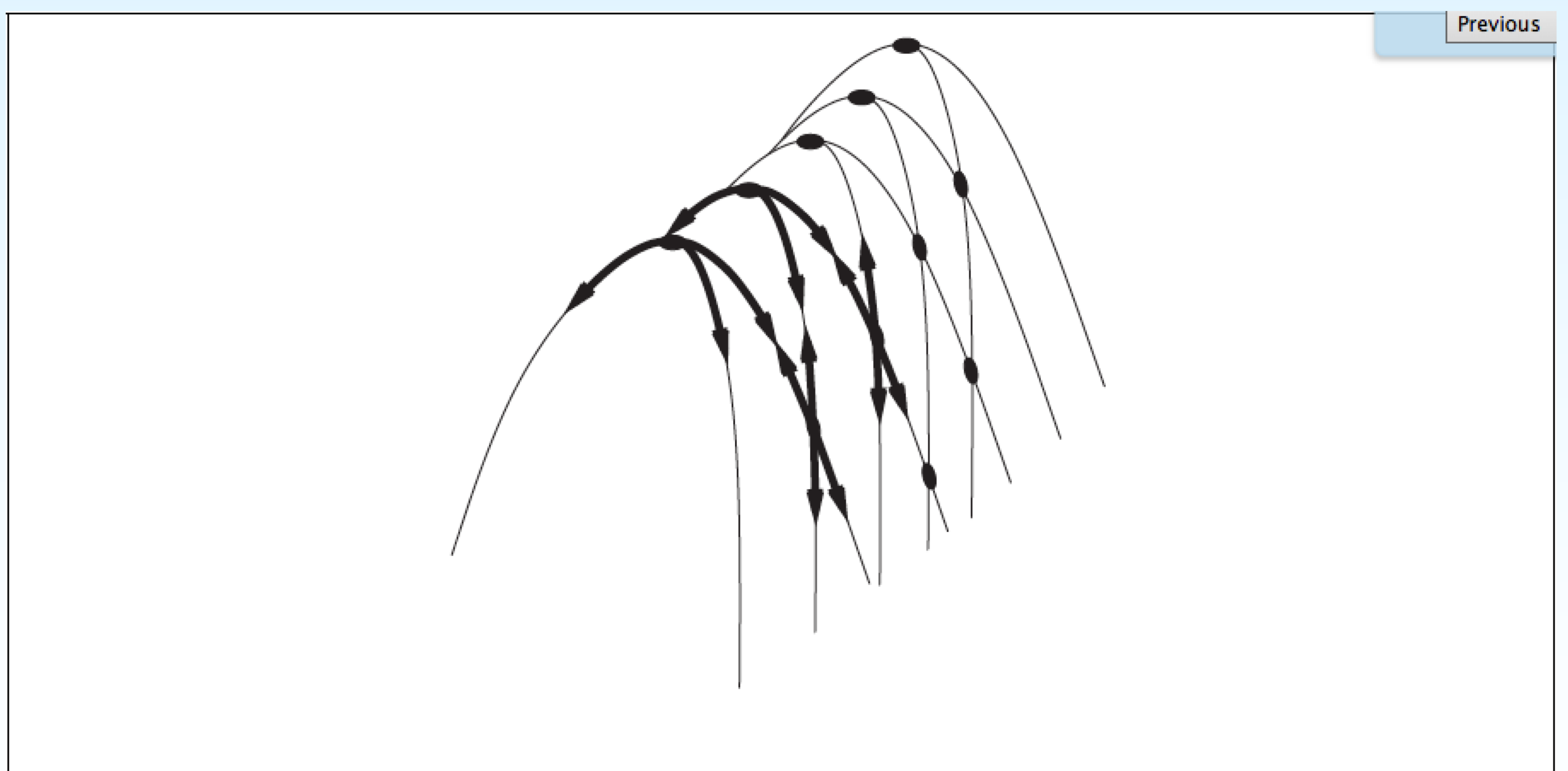


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Algoritmi di ricerca locale

1) **Hill-climbing** (o discesa del gradiente)
   - 8regine: 14% risolti con 4 step; 86% fail con 4 step
   - 8regine = 8^8 stati = 14M stati circa
2) **HC** con mosse laterali (per plateaux)
   - 8regine: 94% risolti con 21 step; 16% fail con 64 step
3) **HC stocastico-a** con probabilità di selezione più alta per stati successori migliori
4) **HC stocastico-b** con prima scelta migliorativa random
   - per grandi vicinati
5) **HC stocastico-c** con riavvi casuali (+ mosse laterali)
   - 8regine: 100% risolti – trova soluzione per 3MegaRegine!

Algoritmi di ricerca locale

6) **Simulated annealing**
7) **Local beam search**
8) **Tabu search**
9) **Enforced Hill-climbing**
10) **Algoritmi genetici**
11) **Local search online**
12) **Local search con apprendimento di h(n)**
13) **Enforced Hill-climbing nella pianificazione automatica (il sistema Fast Forward)**
14) **Walksat** per ragionamenti logici
15) **Walkplan** per pianificazione

Ricerca Locale: Simulated Annealing

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
     T ← schedule(t)
     if T = 0 then return current
     next ← a randomly selected successor of current
     ΔE ← next.VALUE – current.VALUE
     if ΔE > 0 then current ← next
     else current ← next only with probability e^(ΔE/T)
```

e = 2,71828
Num di Eulero

Ricerca Locale: Simulated Annealing

Se temperatura decresce abastanza lentamente converge a un ottimo globale con prob. 1

Dal 1980 usato con successo nello scheduling in fabbrica e problemi di ottimizzazione di grandi dimensioni

Ricerca Locale Local Beam

- Parto da K>1 stati iniziali
- K ricerche locali in ‘‘parallelo’’ (ad s. HC)
- Si selezionano i K migliori successori tra **tutti** e si distribuiscono alle K ricerche come ‘new current state’
- Percorsi scadenti tendono ad essere abbandonati per privilegiare percorsi dove ci sono migliori progessi
- Per garantire esplorazione di porzioni differenti dello spazio di ricercare uso ulteriore randomizzazione nella scelta del successore.

Ricerca Locale con Memoria Breve (Tabu Search)

Hill climbing modificato:

1. sceglie il migliore successore (anche se non migliora lo stato corrente)
2. uso memoria ultimi k stati visitati (**tabu list**)
3. Il successore non deve essere nella tabù list

- k scelto considerando il costo di 3 (ad es. k=10)
- Tecnica molto utile quando i minimi (massimi) locali hanno profondità (altezza limitata con k)
- Esempio alla lavagna

Enforced Hill-Climbing

Hill Climbing modificato:

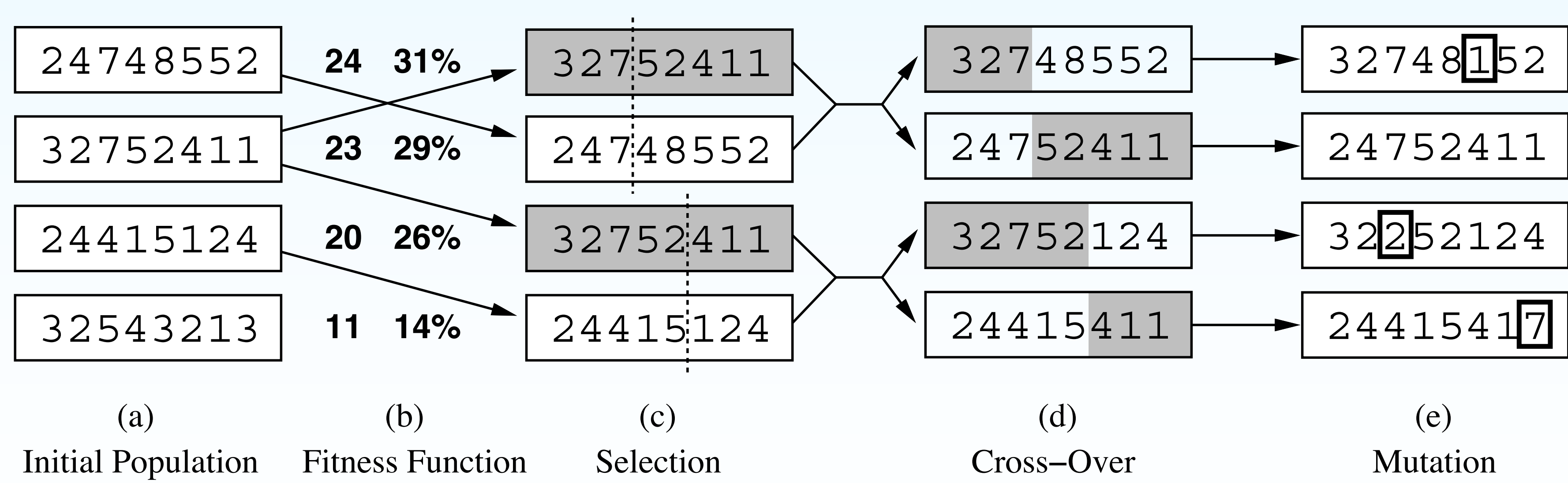
- Per ogni stato visitato *s* diditato da HC, uso una <u>ricerca in ampiezza</u> con stato iniziale *s* per trovare uno stato <u>discendente</u> *s'* che ha valore euristico (funzione h) migliore di *s*
- Se trovo *s'* applico la <u>sequenza</u> di azioni da *s* a *s'* (quella del path da *s* a *s'* nel albero costruito dslla ricerca in ampiezza)

Esempio alla lavagna

Problema memora……

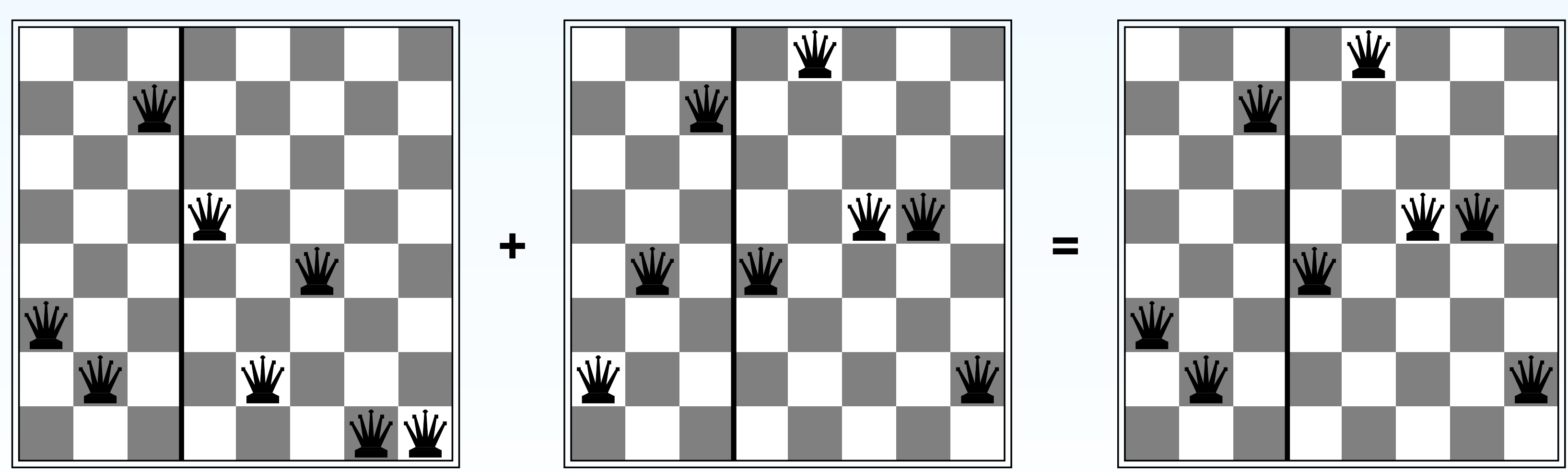
Algoritmi Genetici: Esempio

Stato 8-Regine = vettore di interi da 1 a 8 = righe regine su colonne
Popolazione di 8 individui (stati)



Algoritmi Genetici: Esempio

Accopiamento/generazione del discendente (figlio) e perdita di informazione nei genitori



Algoritmi Genetico Generale

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     loop for i from 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Agenti con Ricerca “Online”

Ragionamento/computazione e azione si alternano: determino azione, eseguo, osservo ambiente, ….

Quando?

- No tempo per “pensare” a una soluzione intera
- Conoscenza parziale ambiente e/o proprie azioni
- Ambiente dinamico (altri agenti, azioni non-deterministiche, eventi esterni)

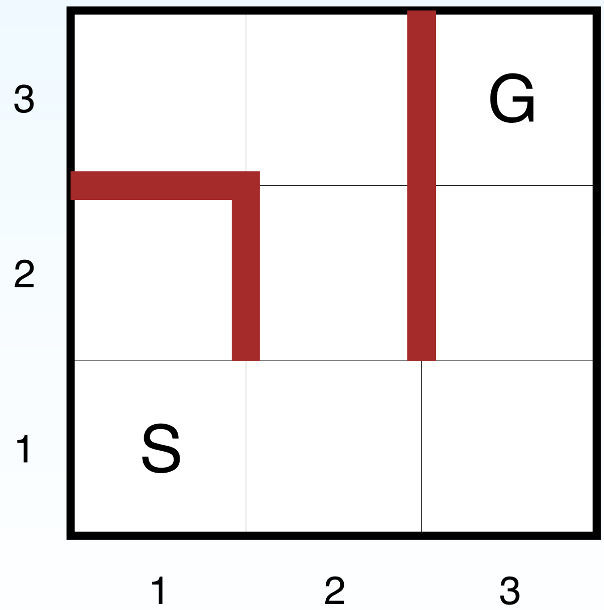
Input di un problema tipico online:

- Azioni(s): lista azioni applicabili in s
- C(*s,a,s’*): funzione costo azione *a* da *s* a *s’ (s’ conosciuto DOPO esecuzione) --> usabile dopo aver eseguito l’azione*
- *Goal-test(s)*

=> *Stati successori si conoscono solo* ***dopo*** *aver eseguito TUTTE le azioni applicabili*

Esempio Ricerca Online

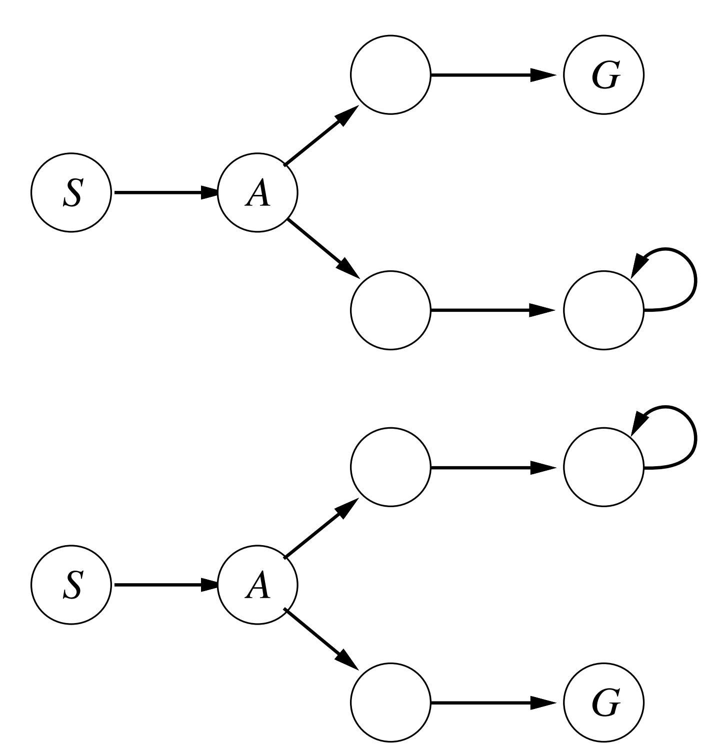
L'agente deve andare da S a G non sapendo nulla dell'ambiente e delle proprie azioni
Assunzioni: azioni deterministiche + ricorda stati già visitati



L'agente (aspirapolvere) potrebbe conoscere le conseguenze ***attese*** delle proprie azioni e avere una euristica (DM), ma non conoscere gli ostacoli presenti….

Altro esempio

L’agente ha attraversato S e A. I seguenti due diagrammi degli stati sono per lui identici, ma in realtà molto differenti tra loro….



Rapporto di Competitività

Usato per misurare prestazioni algoritmi online:

Costo soluzione trovata online (ambiente/diagramma stati sconosciuto) diviso costo soluzione ottima offline (ambiente conosciuto)

Può essere infinito! (vicoli ciechi/deadends che mi fanno stare sempre nella stessa posizione/stato)

I due esempi precedenti contengono deadends

Spazio degli stati **esplorabile in modo sicuro**:

Non ci sono deadends e tutti gli operatori/azioni sono ***reversibili*** *(come ad es. in un tipico labirinto)*

Algoritmi di ricerca online

Assunzione: *dopo ogni azione l'agente osserva il mondo e conosce lo stato del mondo in cui è aggiornando la sua conoscenza dell'ambiente (diagramma degli stati)*

Possiamo usare uno degli algoritmi completi già visti?

- A*?
- Ricerca in ampiezza?
- Ricerca in profondità?

NB: la ricerca dell'algoritmo deve progredire *insieme* all'esecuzione dell'agente ...

Ricerca in profondità online

Nota: creazione di punti di **backtracking** risalita a tali punti durante la ricerca (esempio lavagna)

La ricerca in profondità può essere resa online se:

- le operazioni di "backtracking" sono simulabili attraverso (sequenze di) azioni eseguite dall'agente che mi riportano in uno stato del mondo corrispondente allo stato della ricerca dove si è in contrato il punto di backtracking
- Il grafo/albero di ricerca è bidirezionale

Ricerca in profondità online

function ONLINE-DFS-AGENT(s') **returns** an action
  inputs: s', a percept that identifies the current state
  static: *result*, a table, indexed by action and state, initially empty
    unexplored, a table that lists, for each visited state, the actions not yet tried
    unbacktracked, a table that lists, for each visited state, the backtracks not yet tried
    s, a, the previous state and action, initially null

  if GOAL-TEST(s') **then return** *stop*
  if s' is a new state **then** *unexplored*[s'] ← ACTIONS(s')
  if s is not null **then do**
    result[a, s] ← s'
    add s to the front of *unbacktracked*[s']
  if *unexplored*[s'] is empty **then**
    if *unbacktracked*[s'] is empty **then return** *stop*
    else a ← an action b such that *result*[b, s'] = POP(*unbacktracked*[s'])
  else a ← POP(*unexplored*[s'])
  s ← s'
  return a

NB: unbacktrack[s’] = non è aggiornato se *a* è azione di “reverse

Ricerca in profondità online esercizio

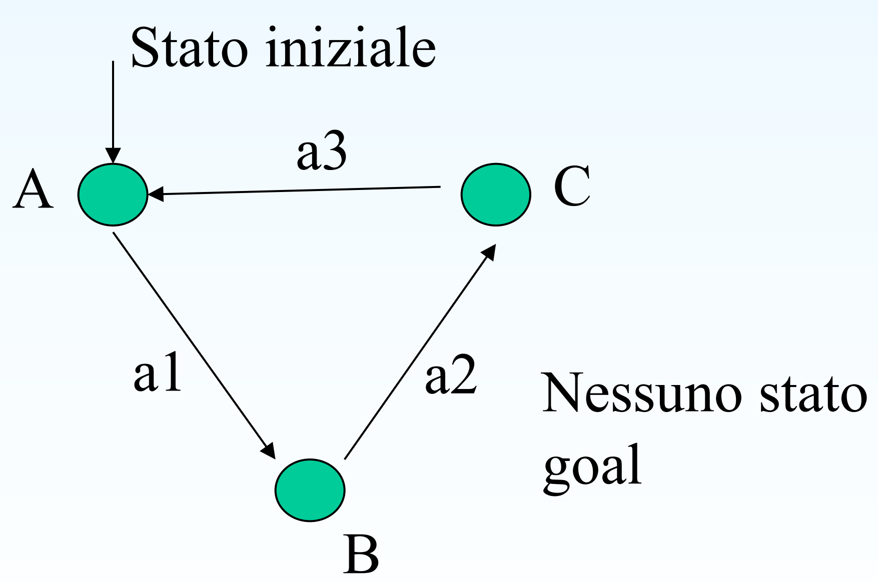


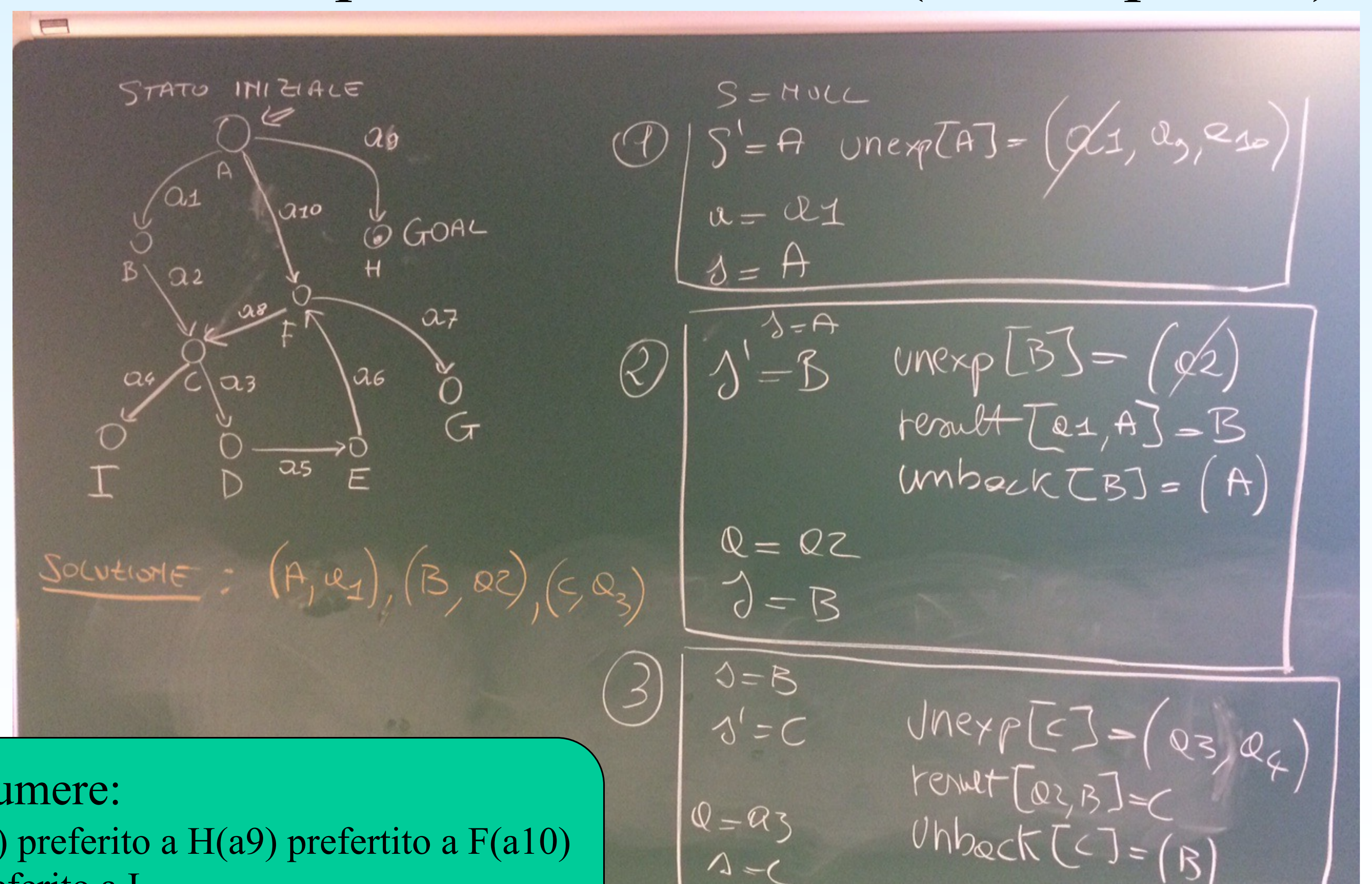
Diagramma degli stati (senza azioni reverse)

s=null, s'=A, Unex[A]=(a1)
a=a1 e Unex[A]=()

s=A, s'=B, Unex[B]=(a2)
Unback[B]=(A)
a=a2 e Unex[B]=()

…. Continuare …

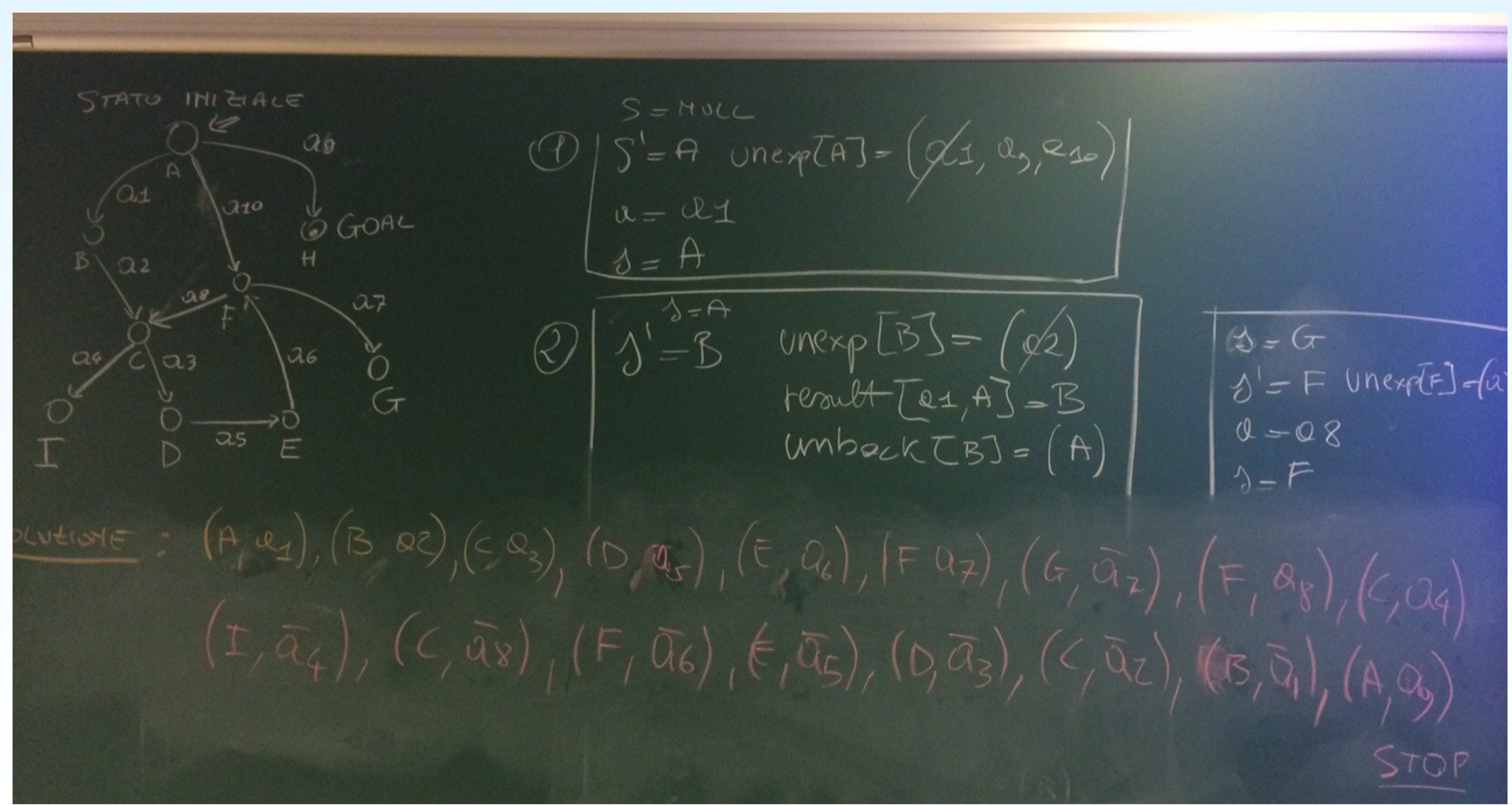
Ricerca in profondità esercizio (da completare)



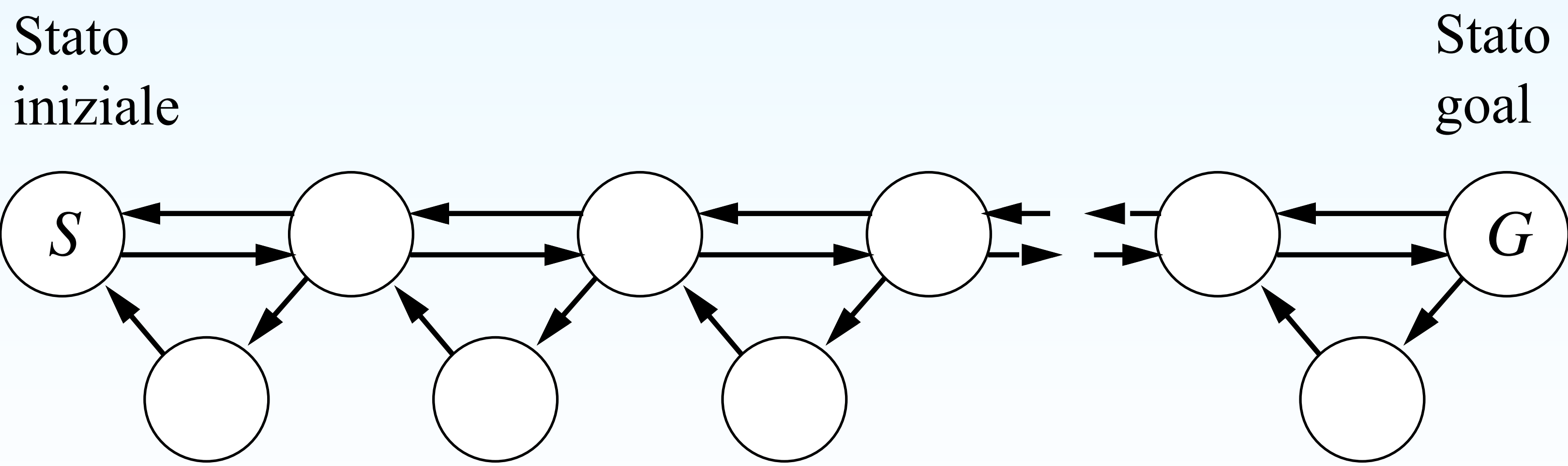
Assumere:
B(a1) preferito a H(a9) prefertito a F(a10)
D preferito a I
G preferito a C

Qual è il valore del rapporto di competitività ottenuto?

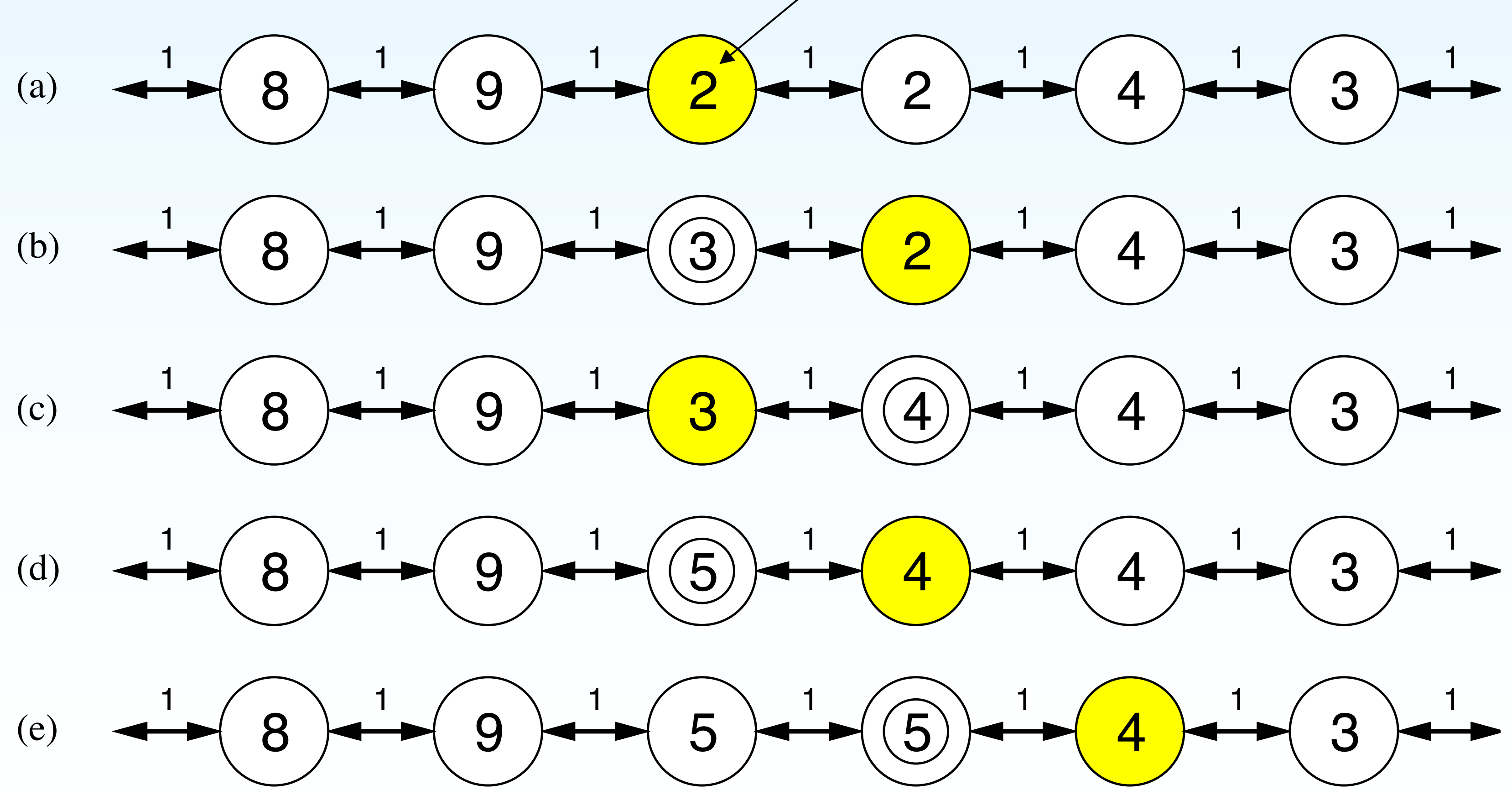
Ricerca in profondità esercizio (soluzione)



Ricerca locale online: esempio “random walk”



Ricerca locale online con apprendimento di h(n): esempio



Ricerca locale online

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          H, a table of cost estimates indexed by state, initially empty
          s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  unless s is null
     result[a, s] ← s'
     H[s] ←        min        LRTA*-COST(s, b, result[b, s], H)
            b ∈ ACTIONS(s)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[b, s'], H)
  s ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return c(s, a, s') + H[s']
```

Esercizio LRTA* (in aula)

$H[s] \leftarrow \min_{b \in Actions(s)} LRTA^*\text{-}cost(s, b, result[b, s], H)$

s —a1→ s'
s —a2→ s''
s —a3→ s'''

$Actions(s) = \{a_1, a_2, a_3\}$

Ipotesi: →
stato corrente = s'
stato precedente = s
s'' già visitato
s''' non visitato

H[s'] e H[s''] definiti
H[s'''] non definito
$result[s, a_1] = s'$
$result[s, a_2] = s''$
$result[s, a_3] = null$

come viene aggiornato il valore di H[s] usando ?

Esercizio LRTA* (in aula)

$$H[s] = \min \{ \text{LRTA*-cost}(s, a_1, s', H), \text{LRTA*-cost}(s, a_2, s'', H), \text{LRTA*-cost}(s, a_3, s''', H) \}$$

$$= \min \{ cost(s, a_1, s') + H[s'], cost(s, a_2, s'') + H[s''], h(s) \}$$ ← perché s''' non è stato visitato

$h(s) = 4$

$cost(s, a_1, s') = 2$

$cost(s, a_2, s'') = 1$

$H[s'] = 3$

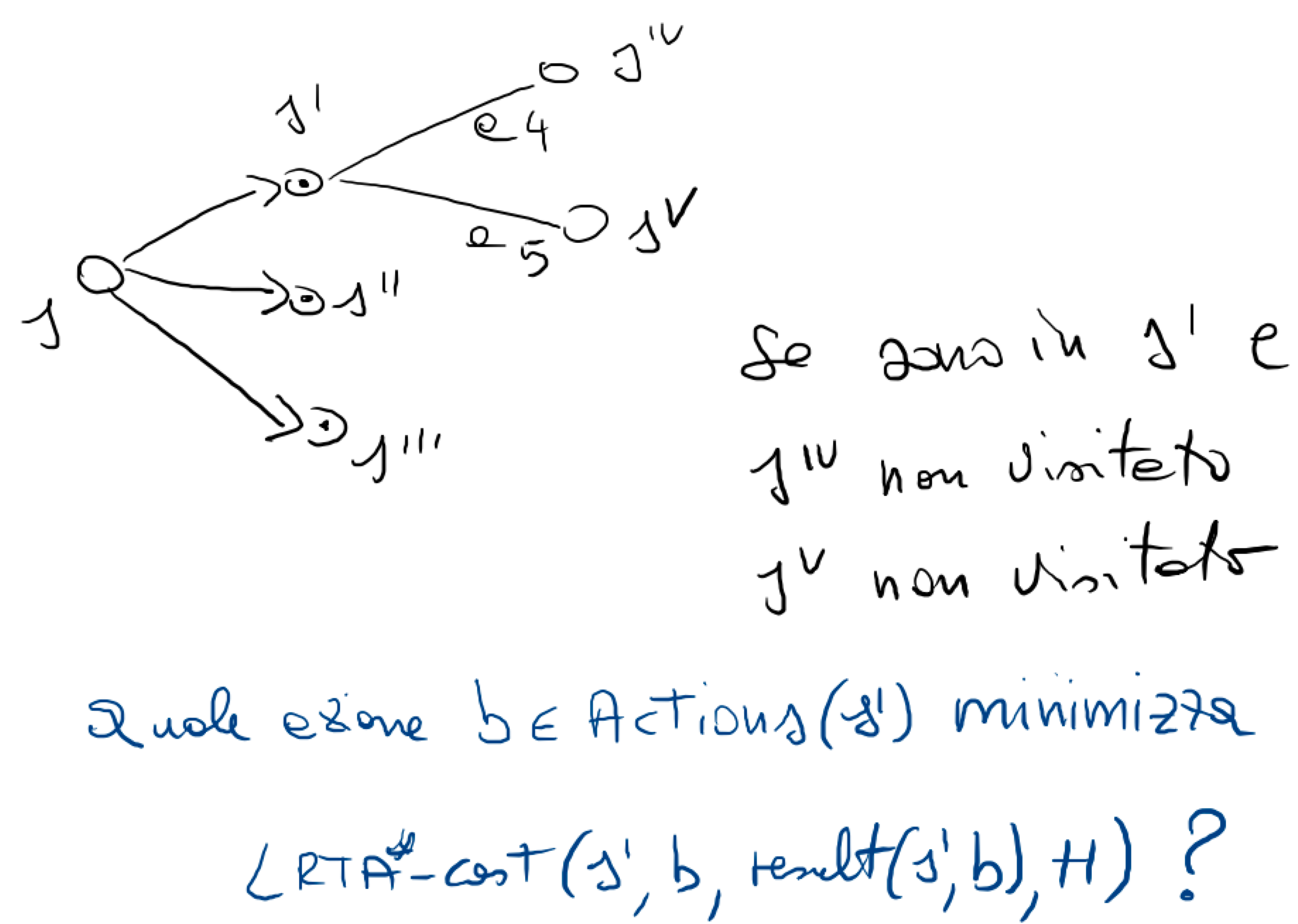
$H[s''] = 4$

$$= \min \{ 2+3, 1+4, 4 \} = 4$$

$\Rightarrow H[s]$ non cambia (4 è valore iniziale)

E se avessimo già visitato anche s''' con $H[s'''] = 5$?

Esercizio LRTA* (in aula)



Si considerino differenti scenari in cui result[s’,a4] e/o result[s’,a5] sono indefiniti

Problemi con Azioni Non-Deterministiche

Stati possibili nell'esempio Robot Aspirapolvere

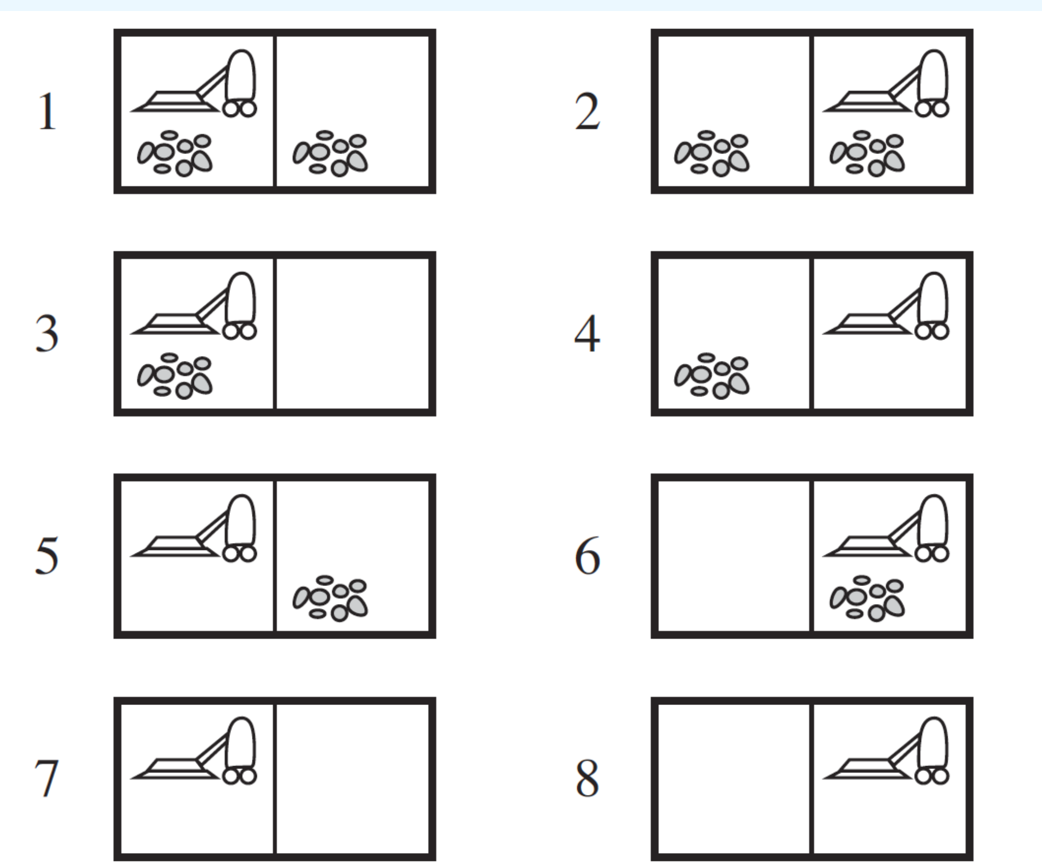


Diagramma degli stati (azioni deterministiche)

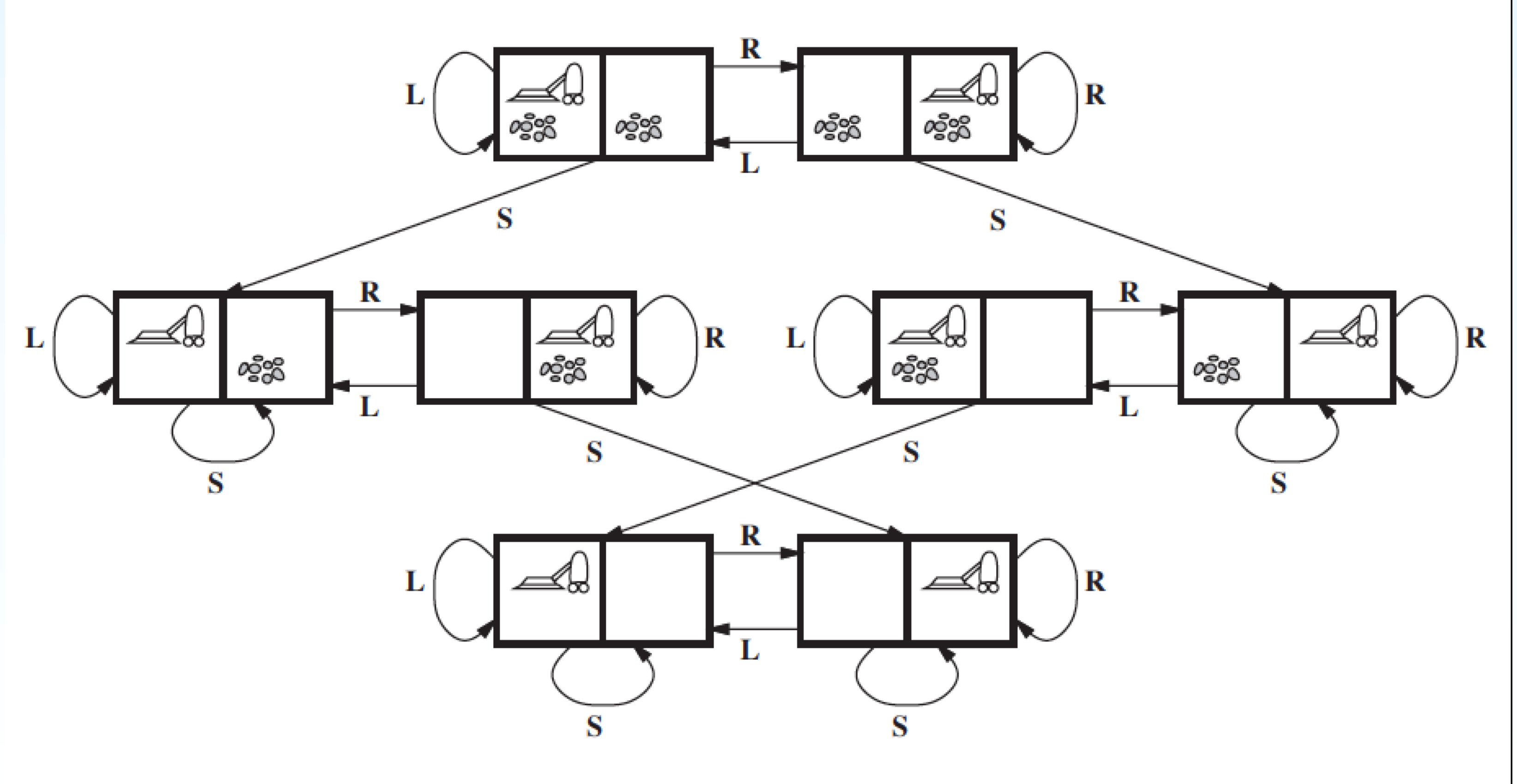


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Esempio robot aspirapolvere con azione non deterministica (aspira)

- ‘Aspira’ (suck) pulisce la locazione in cui e’ il robot e a volte polusce anche quella adiacente
- ‘Aspira’ in a locazione pulita a volte sporca la locazione

Struttura delle soluzioni:

- non può essere una sequenza, è un albero (sequenze + **if-then-else**)

  [aspira, if state = 5 then [vadestra, aspira] else []]

- Gli if-the-else possono essere annidati

NB: gli stati sono fully observable

AND-OR Search Tree

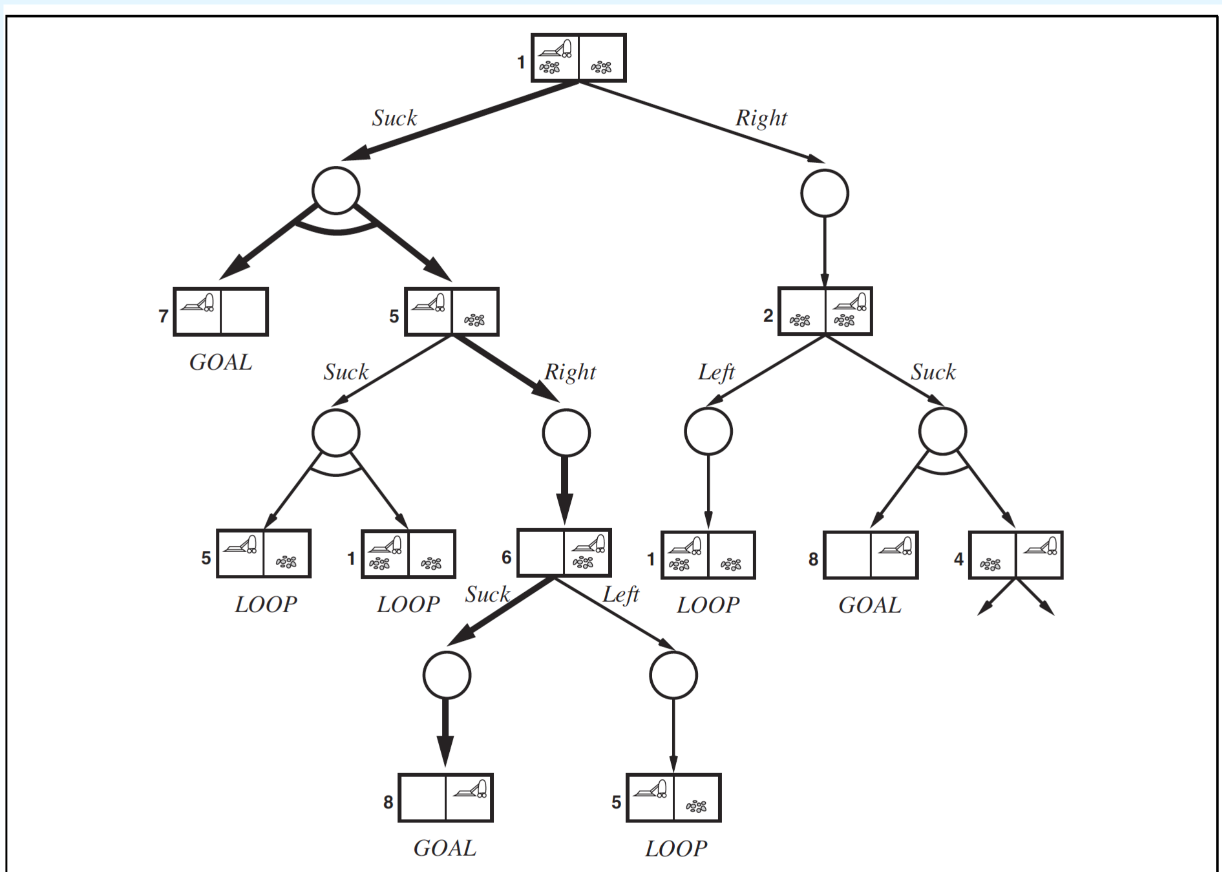


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

Pruning dei cicli: se uno stato è già presente sul path che lo raggiunge dalla radice, viene eliminato (è nodo terminale). Se esiste una soluzione che passa da quello stato, la possiamo trovare ottenere passando dal nodo 'uguale' precedente sul path.

Algoritmo AND-OR Search

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
   OR-SEARCH(problem.INITIAL-STATE, problem, [ ])
---------------------------------------------------------------------------
function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
   if problem.GOAL-TEST(state) then return the empty plan
   if state is on path then return failure
   for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
   return failure
---------------------------------------------------------------------------
function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
   for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
   return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

Estensioni di AND-OR Search

- Uso di euristiche per guidare la costruzione dell'albero AND-OR
- Soluzioni cicliche: ogni effetto di qualsiasi azione prima o poi accade se continuo a ripeterne l'esecuzione
- Struttura soluzioni cicliche: sequenze + if-then-else + cicli (**while**)
- Esempio aspirapolvere: le azioni di movimento a volte non spostano il robot (aspira resta invece deterministica).
  - Esempio: [aspira, **while** state = 5 **do** GoRight, aspira]

AND-OR Search Tree con cicli

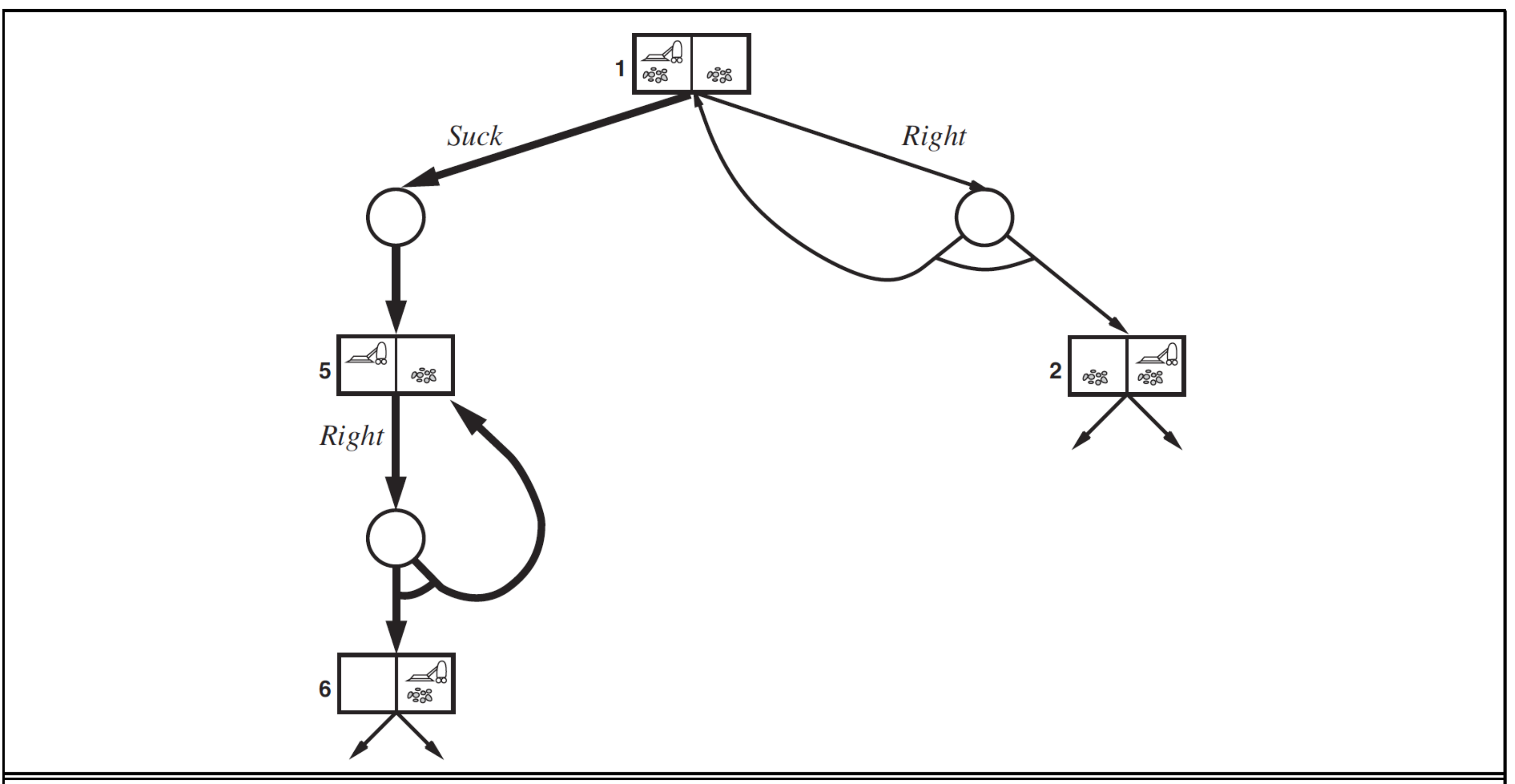


Figure 4.12 Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.