



UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA



Intelligenza Artificiale

Risolvere i problemi con la ricerca

Outline

- ▶ Agenti per la risoluzione di problemi
- ▶ Tipi di problemi
- ▶ Formulazione dei problemi
- ▶ Esempi di problemi
- ▶ Algoritmi di ricerca non-informati
- ▶ Algoritmi di ricerca informati

Agenti *risolutori di problemi*

- ▶ Adottano il paradigma della **risoluzione di problemi come ricerca** in uno spazio di stati
- ▶ Sono particolari agenti con obiettivo, che pianificano l'intera sequenza di mosse prima di agire
- ▶ Gli stati sono privi di una struttura interna (atomici)
- ▶ Passi da seguire:
 1. Determinazione obiettivo (un insieme di stati)
 2. Formulazione del problema
 3. Determinazione della soluzione mediante ricerca
 4. Esecuzione del piano

Agenti *risolutori di problemi*

FORMULAZIONE DEL PROBLEMA

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

||
problema



RICERCA DI UNA SOLUZIONE

SEARCH

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

||
soluzione = sequenza di azioni

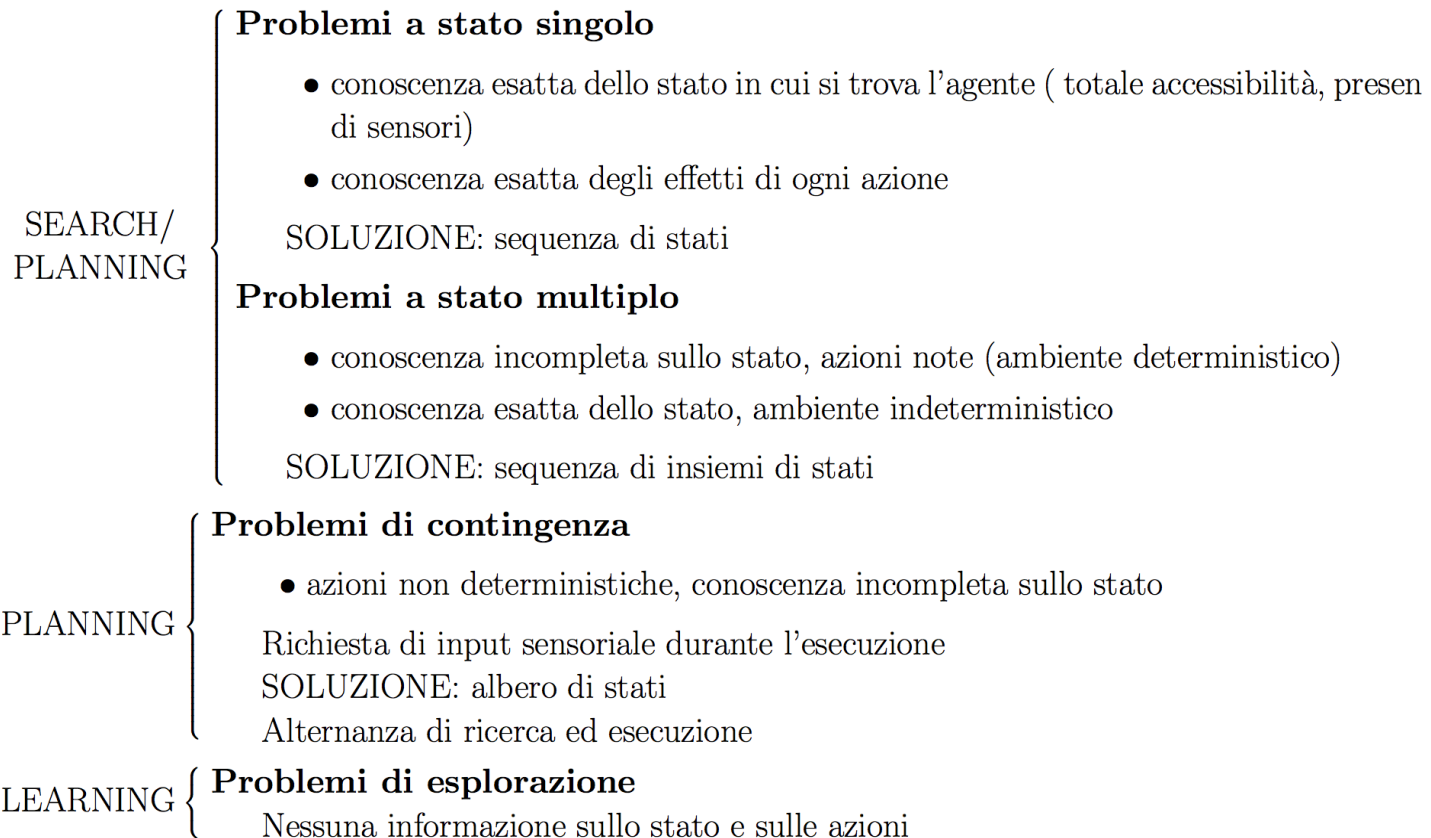


ESECUZIONE

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)



Tipi di problemi

- ▶ Deterministico, pienamente osservabile → **problema a singolo stato**
 - ▶ L'agente conosce esattamente in quale stato si troverà; la soluzione è una sequenza
- ▶ Non-osservabile → **problema senza sensore**
 - ▶ L'agente potrebbe non sapere dov'è; la soluzione è una sequenza
- ▶ Non deterministico e/o parzialmente osservabile → **problema di contingenza**
 - ▶ Le percezioni forniscono **nuove** informazioni sullo stato corrente
 - ▶ Spesso **alternano** ricerca ed esecuzione
- ▶ Spazio degli stati sconosciuto → **problema di esplorazione**

Problemi di search (a stato singolo o multiplo)

► Componenti che definiscono un problema:

1. Stato iniziale
2. Azioni o operatori
3. Test obiettivo (goal test)
4. Funzione costo: associa un costo a ogni operatore

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

Algoritmi di ricerca

*Gli algoritmi di ricerca prendono in input un problema e restituiscono un **cammino soluzione**, cioè una sequenza di azioni che portano dallo stato iniziale a uno stato goal*

► *Misura delle prestazioni*

Trova una soluzione? quanto costa trovarla? quanto efficiente è?

Costo totale = costo della ricerca +
 costo del cammino soluzione

Algoritmi di ricerca

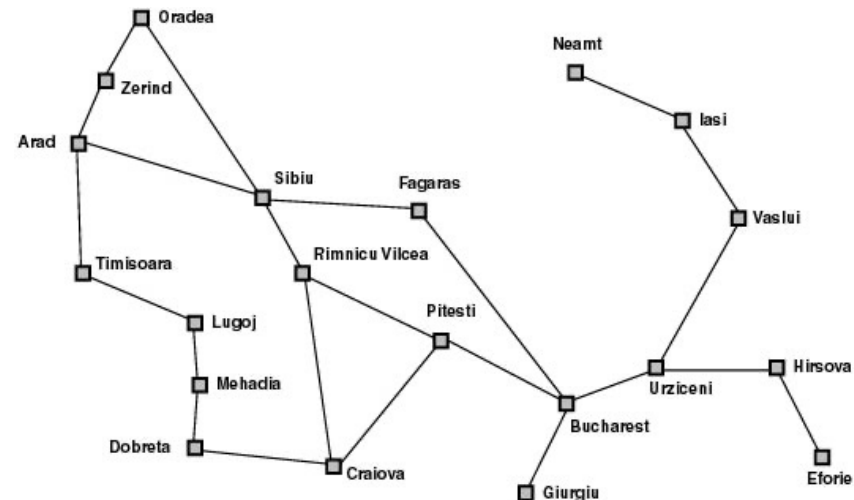
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation

  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then do
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
  action  $\leftarrow$  FIRST(seq)
  seq  $\leftarrow$  REST(seq)
  return action
```

Esempio: Romania

- ▶ In vacanza in Romania; ora in Arad.
- ▶ L'aereo parte domani da Bucarest

- ▶ Formulazione dell'obiettivo:
essere a Bucarest
- ▶ Formulazione del problema:
 - ▶ **stati**: le varie città
 - ▶ **azioni**: guidare tra le città

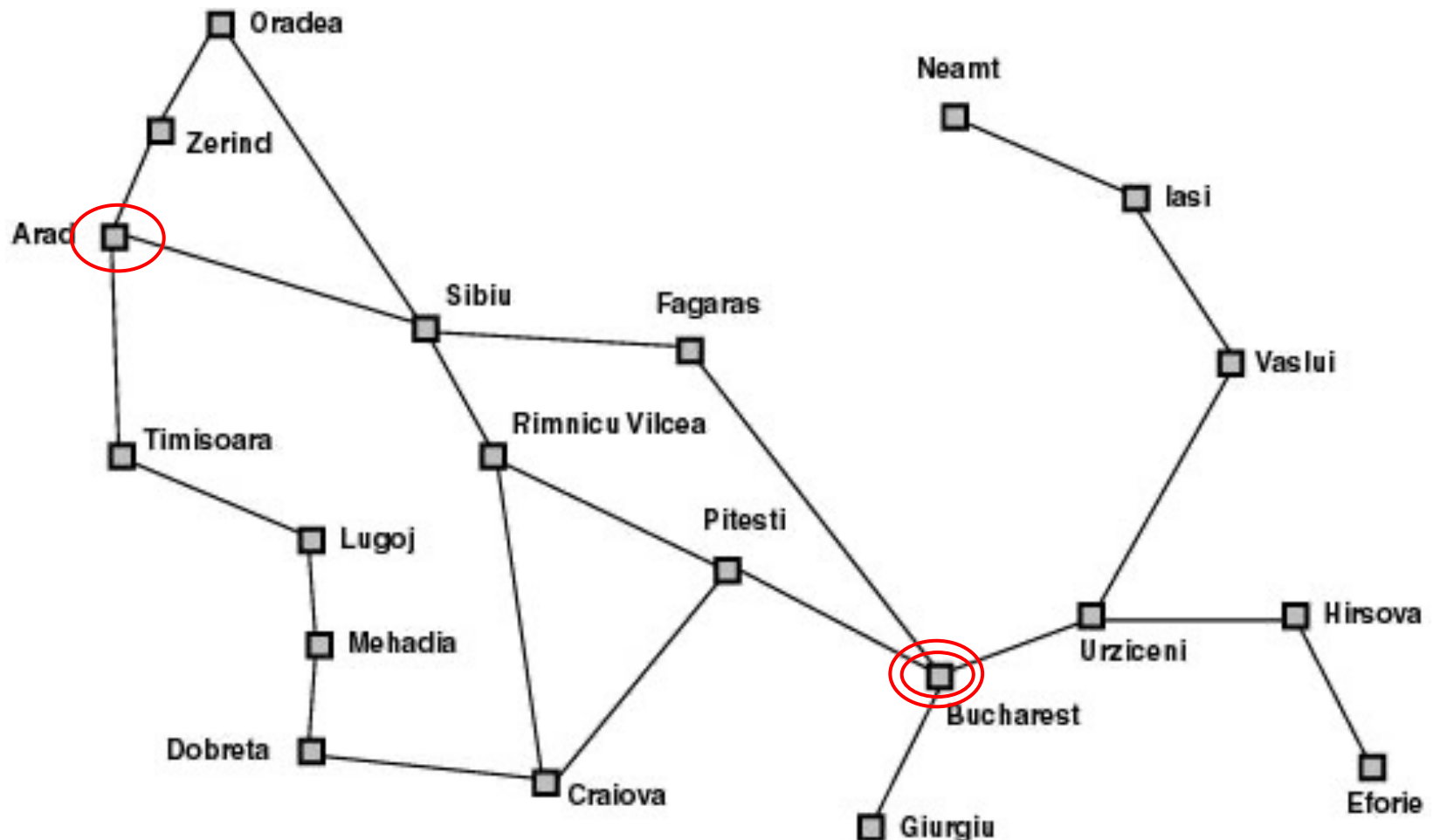


- ▶ Trovare una soluzione:
 - ▶ Sequenze di città, ad esempio, Arad, Sibiu, Fagaras, Bucarest

Che tipo di assunzioni?

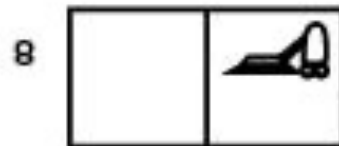
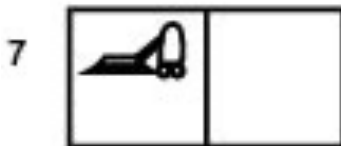
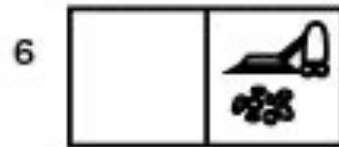
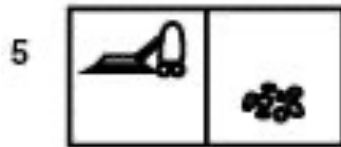
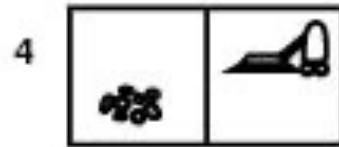
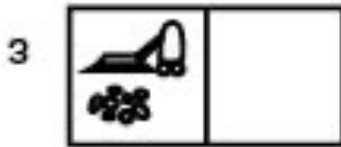
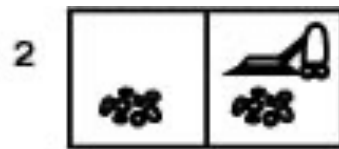
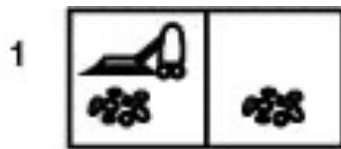
- ▶ L'ambiente è statico
- ▶ Osservabile
- ▶ Discreto
 - ▶ un insieme finito di azioni possibili
- ▶ Deterministico
 - ▶ Si assume che l'agente possa eseguire il piano “ad occhi chiusi”

Route finding: il problema



Vacuum world: il problema

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



Percezioni:

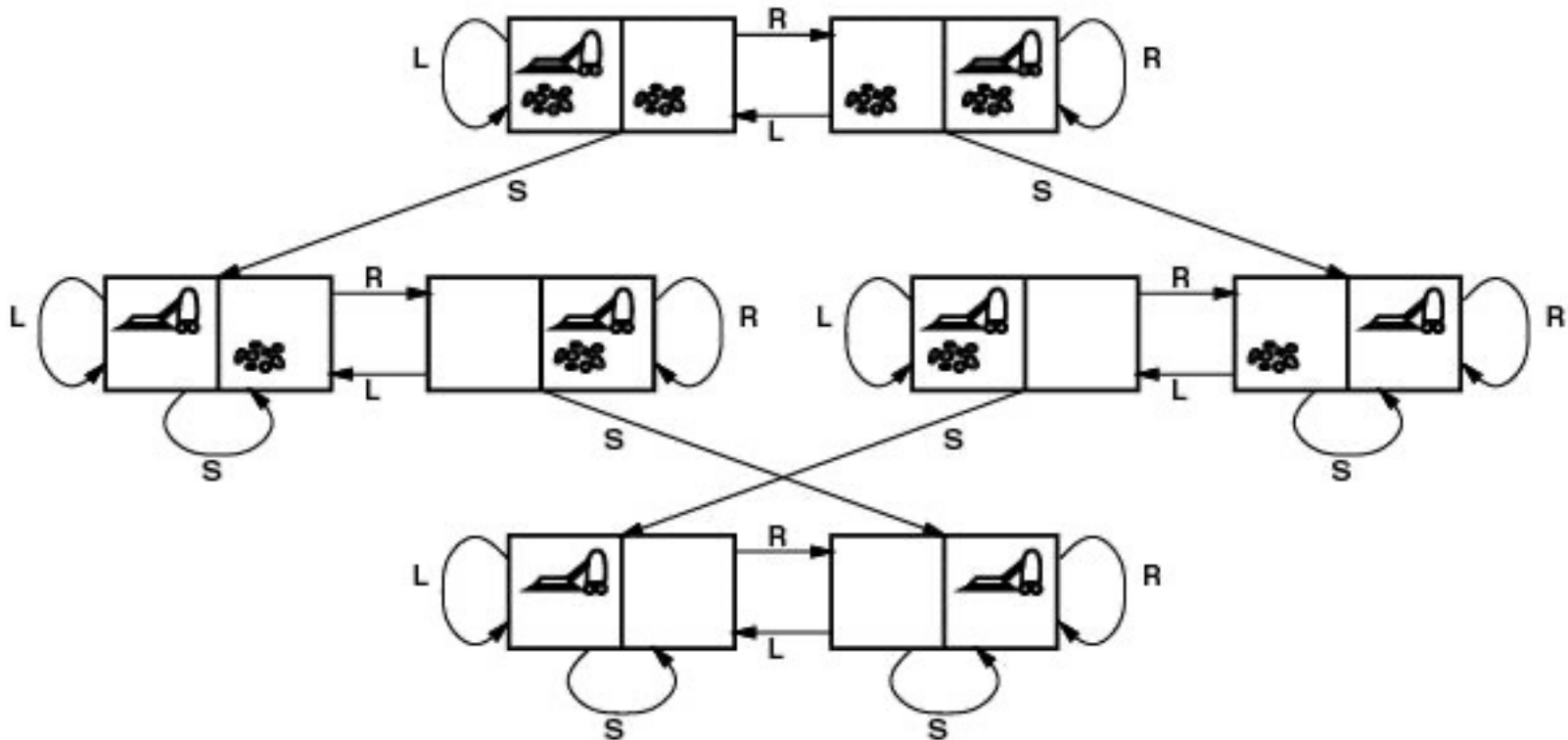
Dirt, No_dirt

Azioni:

Left, Right, Suck

Vacuum world: formulazione

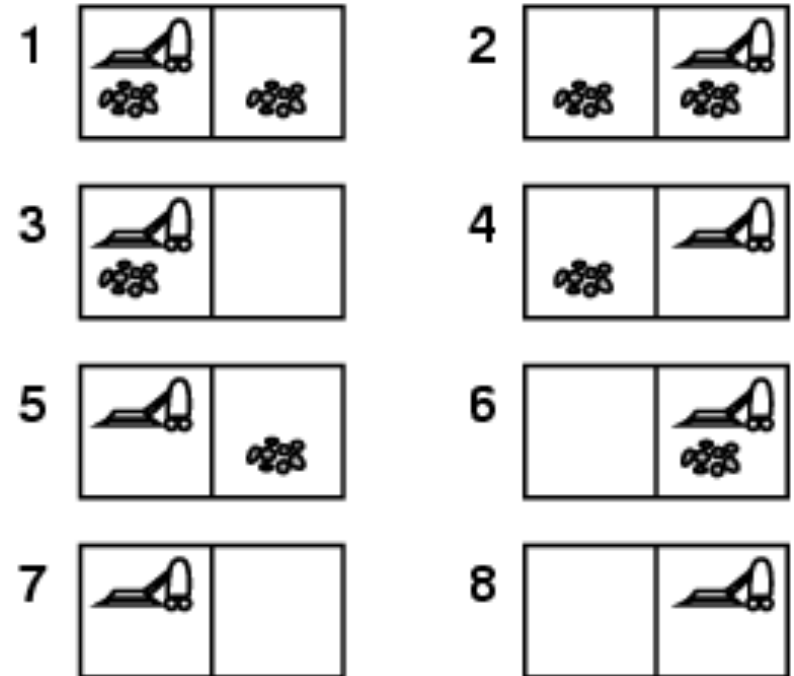
- *Obiettivo*: rimuovere lo sporco { 7, 8 }
- *Funzione di costo*: ogni azione ha costo 1
- *Spazio degli stati* :



Esempio: vacuum world

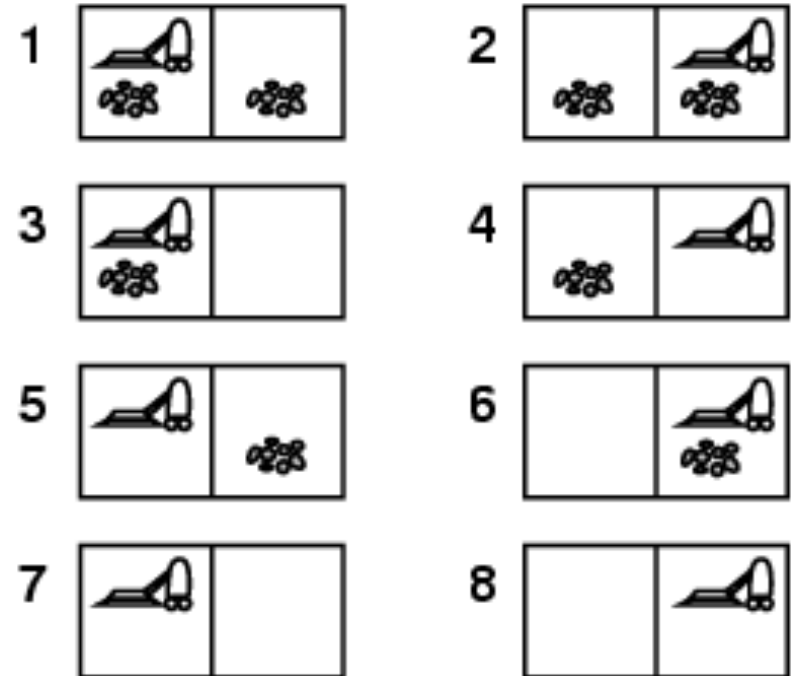
- ▶ Stato singolo, inizia in #5.

Soluzione?



Esempio: vacuum world

- ▶ **Stato singolo**, inizia in #5.
Soluzione? [*Right, Suck*]
- ▶ **Senza sensore**, inizia in $\{1,2,3,4,5,6,7,8\}$ ad esempio,
Right va in $\{2,4,6,8\}$
Soluzione?



Esempio: vacuum world

- ▶ **Senza sensore**, inizia in $\{1,2,3,4,5,6,7,8\}$ ad esempio, *Right* va in $\{2,4,6,8\}$

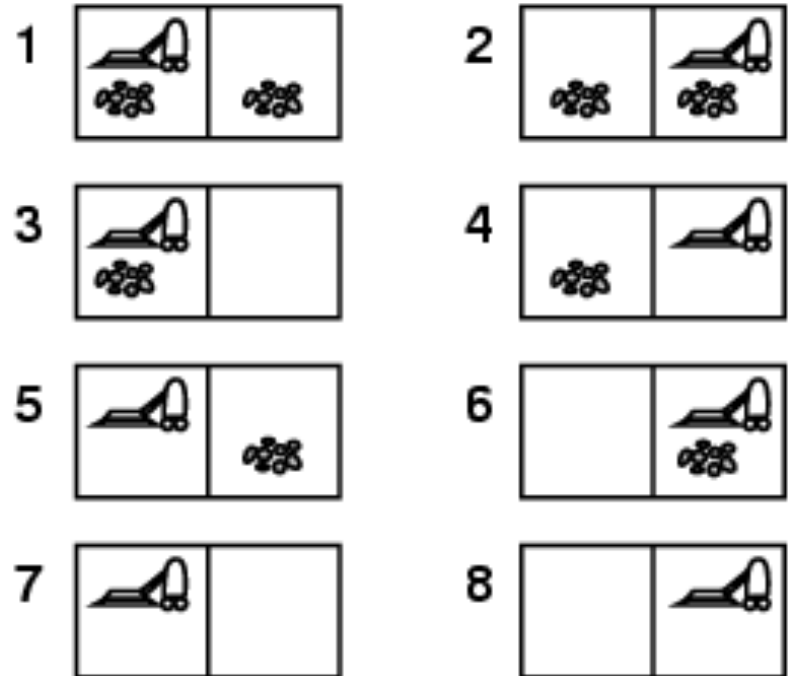
Soluzione?

[Right, Suck, Left, Suck]

- ▶ **Contingenza**

- ▶ Non deterministico: *Suck* può sporcare una cella pulita
- ▶ Parzialmente osservabile: posizione, sporco nella cella attuale.
- ▶ Percepisce: *[L, Clean]*, cioè inizia in #5 or #7

Soluzione?



Esempio: vacuum world

- ▶ **Senza sensore**, inizia in $\{1,2,3,4,5,6,7,8\}$ ad esempio, *Right* va in $\{2,4,6,8\}$

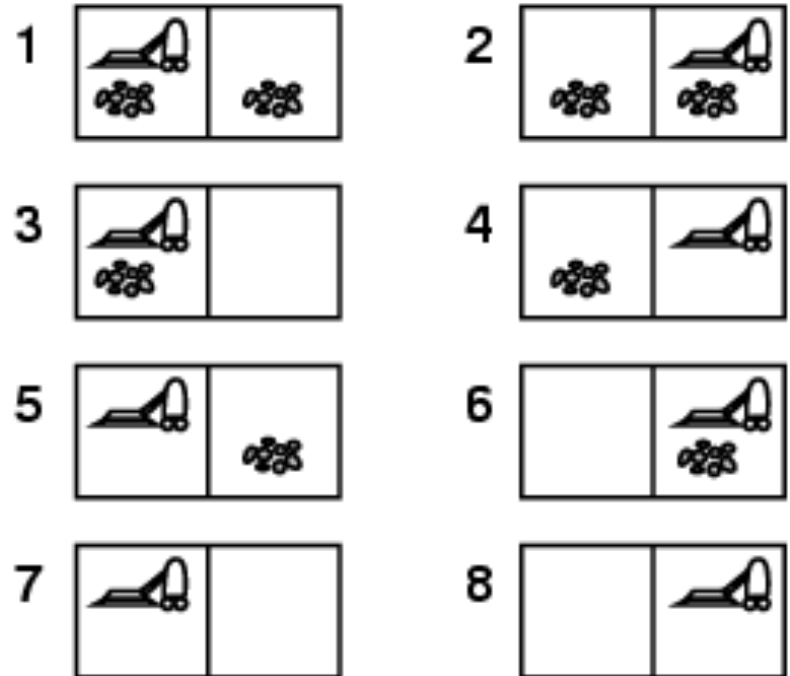
Soluzione?

[Right, Suck, Left, Suck]

- ▶ **Contingenza**

- ▶ Non deterministico: *Suck* può sporcare una cella pulita
- ▶ Parzialmente osservabile: posizione, sporco nella cella attuale.
- ▶ Percepisce: *[L, Clean]*, cioè inizia in #5 or #7

Soluzione? *[Right, if dirt then Suck]*



Formulazione problema singolo stato

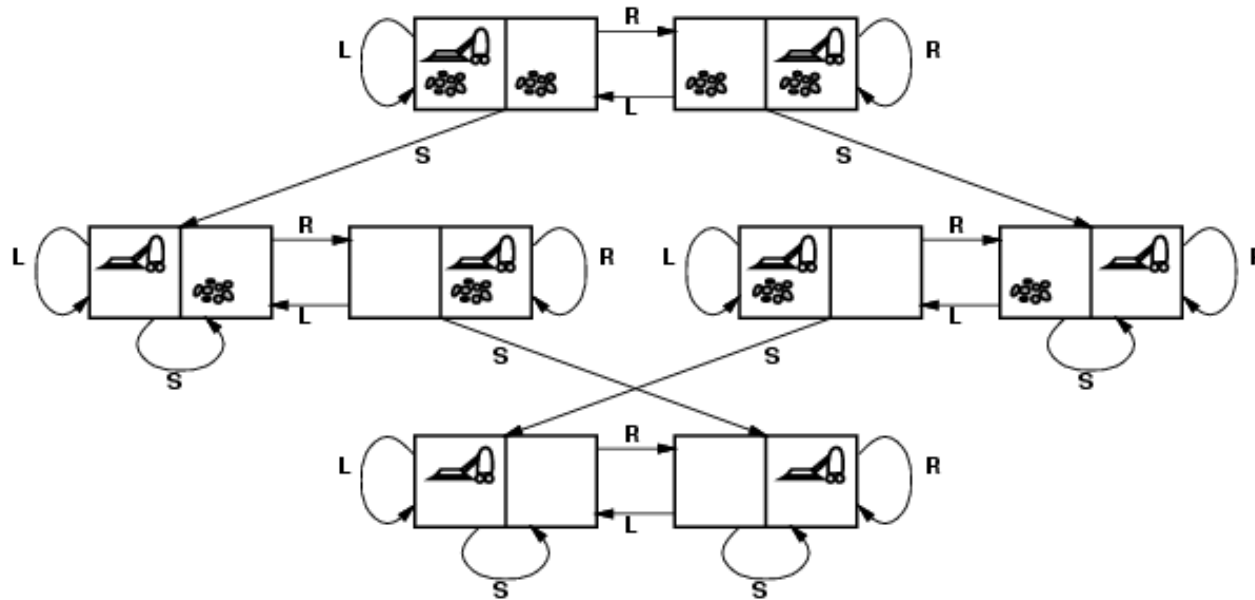
Un **problema** è definito da quattro elementi:

1. **Stato iniziale** ad esempio, "ad Arad"
 2. **Azioni e funzione successore** $S(x)$ = insieme di coppie azione–stato
 - ▶ Ad esempio, $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots \}$
 3. **Test obiettivo**, può essere
 - ▶ **esplicito**, cioè elenco di stati, ad esempio, $x = "In(\text{Bucarest})"$
 - ▶ **implicito**, ad esempio, $NoDirt(x)$, *scaccomatto*
 4. **Costo di cammino** (somma)
 - ▶ ad esempio, somma di distanze, numero di azioni eseguite, etc.
 - ▶ $c(x, a, y)$ è il **costo di un passo**, che si assume essere ≥ 0
- ▶ Una **soluzione** è una sequenza di azioni che portano da uno stato iniziale ad uno stato obiettivo

Selezionare uno spazio di stati

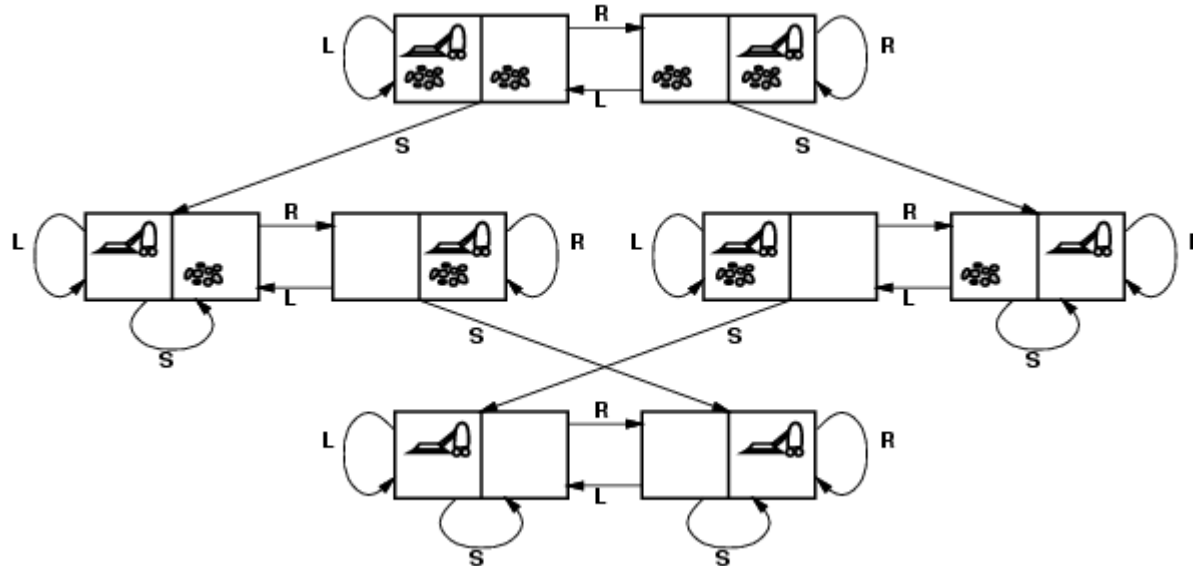
- ▶ Il mondo reale è molto complesso
 - Bisogna effettuare un'**astrazione** dello spazio degli stati per risolvere problemi
- ▶ stato (astratto) = insieme di stati reali
- ▶ azione (astratta) = combinazione complessa di azioni reali
 - ▶ e.g., "Arad → Zerind" rappresenta una serie complessa serie di possibili percorsi, deviazioni, soste, etc.
- ▶ Per garantire la realizzabilità, un qualsiasi stato vero "in Arad" deve raggiungere un qualsiasi stato vero "in Zerind "
- ▶ Soluzione (astratta) =
 - ▶ Un insieme di percorsi reali che sono soluzioni nel mondo reale
- ▶ Ogni azione astratta dovrebbe essere "più facile" rispetto al problema originale

Vacuum world: spazio degli stati



- ▶ stati?
- ▶ azioni?
- ▶ test obiettivo?
- ▶ costo cammino?

Vacuum world: spazio degli stati



- ▶ stati? intero dirt e posizione robot
- ▶ azioni? *Left, Right, Suck*
- ▶ test obiettivo? no dirt in tutte le posizioni
- ▶ costo cammino? 1 per ogni azione

Route finding: la formulazione

- ▶ *Stati*: le città
- ▶ *Stato iniziale*: la città da cui si parte
- ▶ *Obiettivo*: la città destinazione
- ▶ *Azioni*: spostarsi su una città vicina collegata
- ▶ *Funzione di costo*: somma delle lunghezze delle strade
- ▶ Lo spazio degli stati coincide con la rete di collegamenti tra città

Il puzzle dell'otto

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

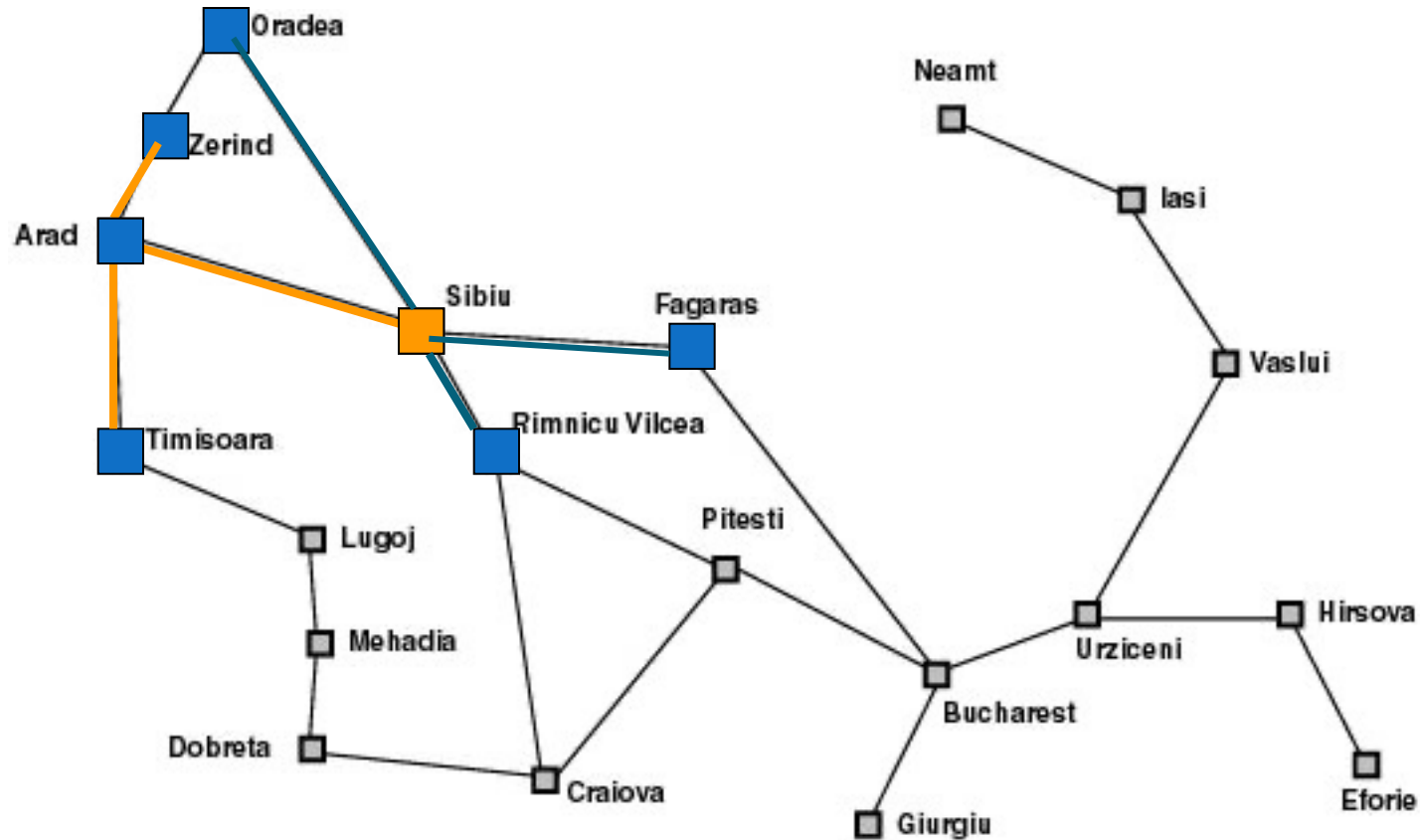
Puzzle dell'otto: formulazione

- ▶ *Stati*: configurazioni della scacchiera
- ▶ *Stato iniziale*: una certa configurazione
- ▶ *Obiettivo*: una certa configurazione
- ▶ *Successori*: mosse della casella bianca
 - in sù: ↑
 - in giù: ↓
 - a destra: →
 - a sinistra: ←
- ▶ *Goal-Test*: Stato obiettivo? →
- ▶ *Path-Cost*: ogni passo costa 1
- ▶ Lo spazio degli stati è un grafo con possibili cicli.
- ▶ [NB: trovare la soluzione ottima per n -Puzzle è NP-hard]

1	2	3
8		4
7	6	5

Ricerca della soluzione

Generazione di un albero di ricerca sovrapposto allo spazio degli stati



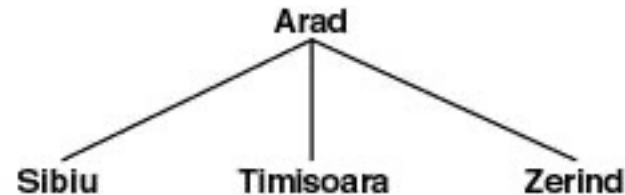
Ricerca della soluzione

Generazione di un albero di ricerca sovrapposto allo spazio degli stati

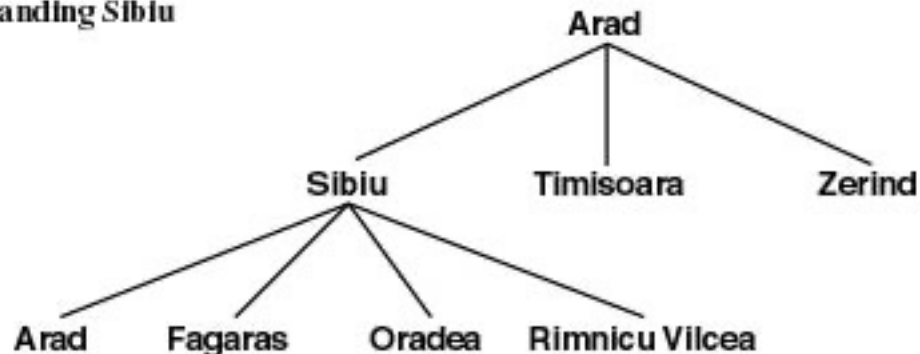
(a) The Initial state

Arad

(b) After expanding Arad

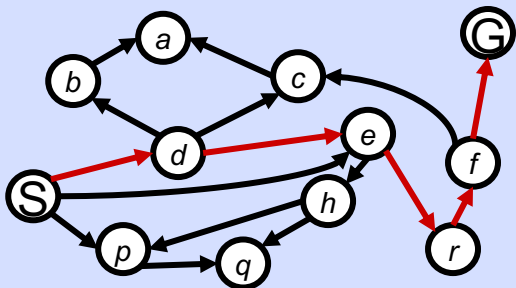


(c) After expanding Sibiu



State Space Graphs vs. Search Trees

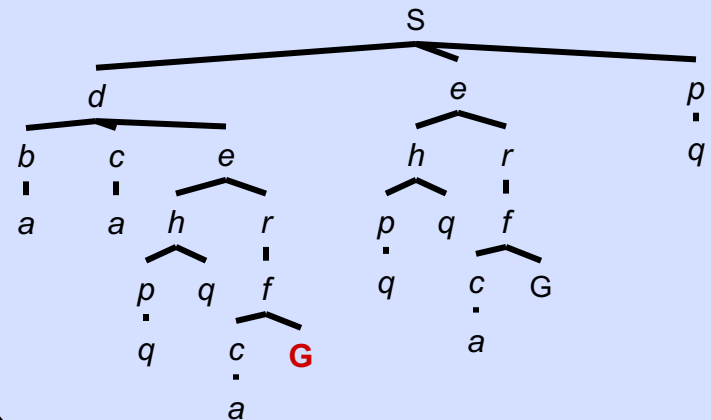
State Space Graph



*Ogni NODO
nell'albero di
ricerca è un
intero
PERCORSO nel
grafo dello
spazio degli
stati.*

*Entrambi si
costruiscono
su richiesta – e
si costruiscono
più piccoli
possibili.*

Search Tree



Strategia di ricerca

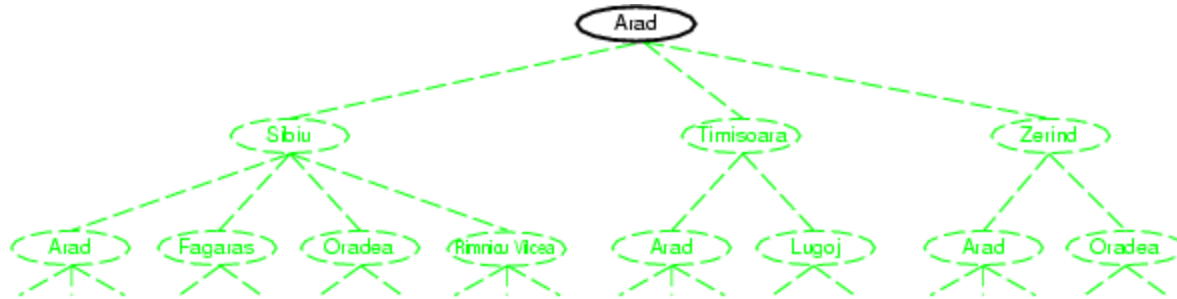
1. Scegliere (tra le foglie dell'albero) un nodo da “espandere”, secondo una data strategia
 2. Controllare se il nodo scelto è un obiettivo
 3. Se non lo è, “espandere” il nodo: generare i suoi figli, ciascuno dei quali contiene uno stato risultante dall'applicazione di un operatore allo stato del nodo 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** o **frontiera**.

Strategia di ricerca

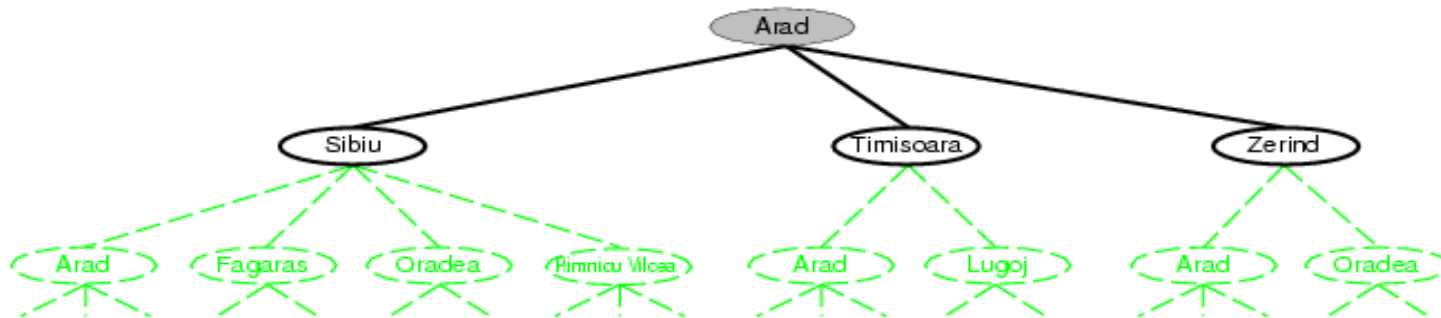
Ovvero: quale nodo espandere tra quelli foglia?

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

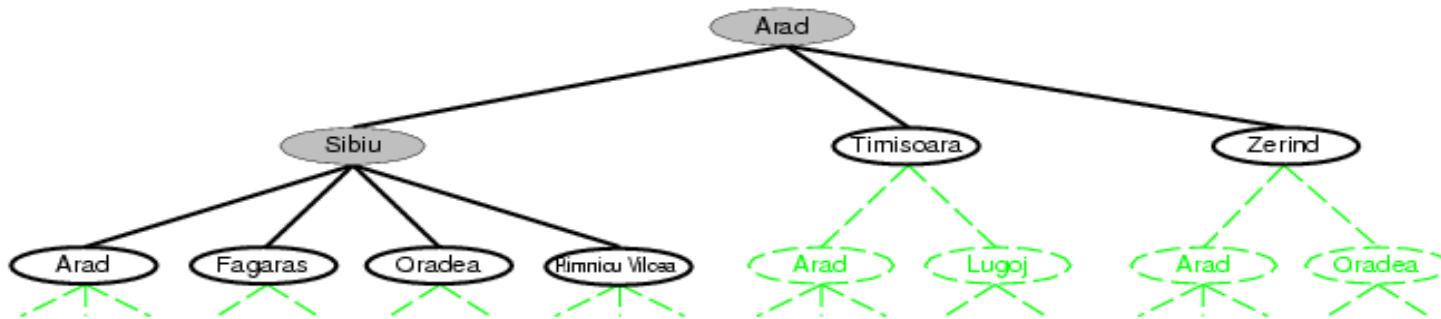
Esempio tree search



Esempio tree search



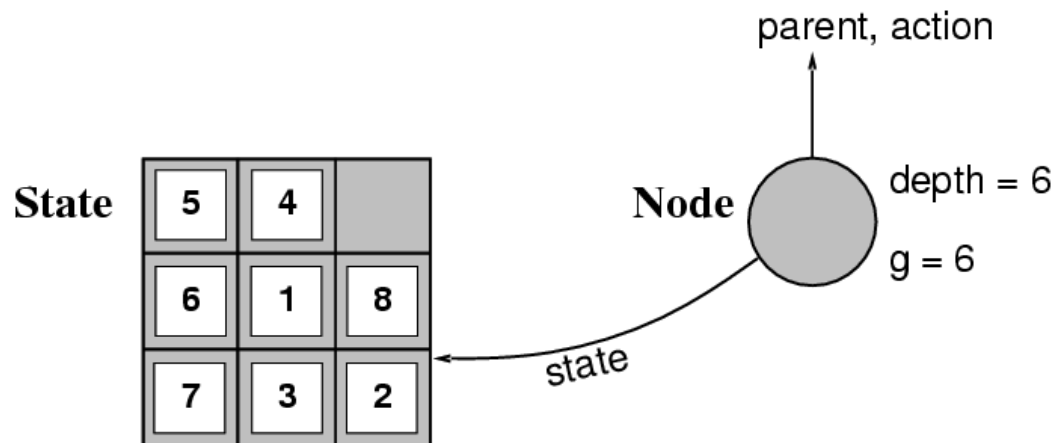
Esempio tree search



I nodi dell'albero di ricerca

Un *nodo* ha cinque componenti:

- ▶ Uno stato
- ▶ Il nodo genitore
- ▶ L'azione eseguita per generarlo
- ▶ La profondità del nodo
- ▶ Il costo del cammino dal nodo iniziale al nodo $g(x)$



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 con operazioni:
 - ▶ Make-Queue
 - ▶ Empty?
 - ▶ First, Rest, RemoveFront
 - ▶ Insert, InsertAll
- ▶ I diversi tipi di coda hanno diverse **Insert**

Tree search algorithm

```
function TREE-SEARCH(problem, fringe) return a solution or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem] applied to STATE[node] succeeds
      then return SOLUTION(node)
    fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Tree search algorithm (2)

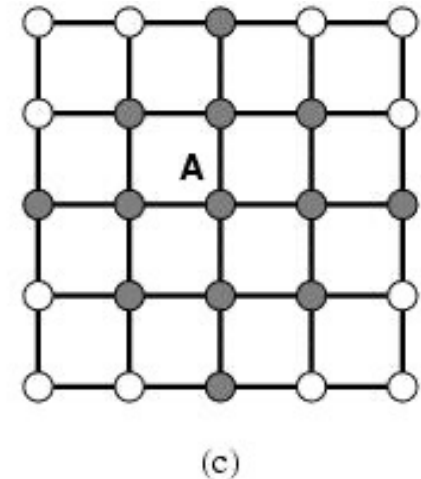
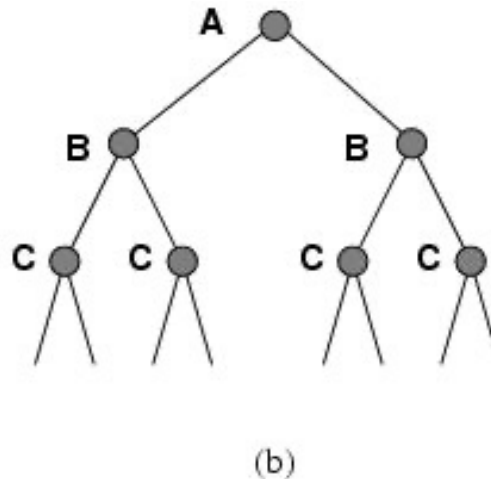
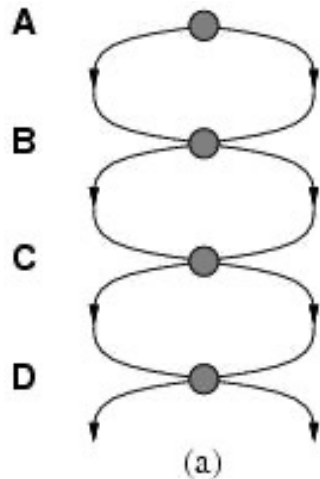
```
function EXPAND(node, problem) return a set of nodes
    successors  $\leftarrow$  the empty set
    for each  $\langle$ action, result $\rangle$  in SUCCESSOR-FN[problem](STATE[node]) do
        s  $\leftarrow$  a new NODE
        STATE[s]  $\leftarrow$  result
        PARENT-NODE[s]  $\leftarrow$  node
        ACTION[s]  $\leftarrow$  action
        PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s]  $\leftarrow$  DEPTH[node] + 1
        add s to successors
    return successors
```

Evitare stati ripetuti

Su spazi di stati a grafo si generano più volte gli stessi nodi nella ricerca.

Albero di ricerca con profondità d

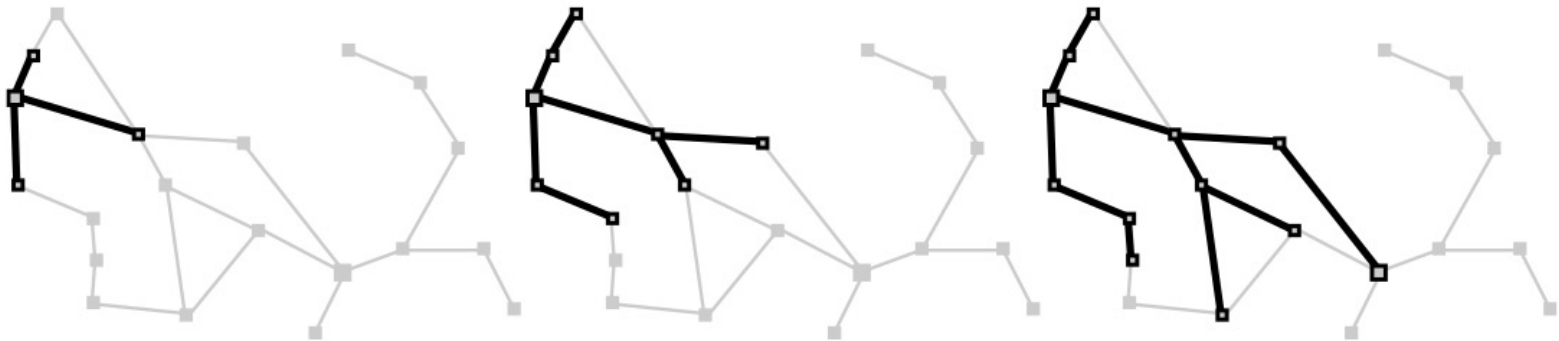
- Ha 4^d foglie
- Ma solo $2d^2$ stati distinti



Come evitare di visitare nodi già visitati?

Compromesso tra spazio e tempo

- ▶ 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!*



Ricerca su grafi

- ▶ Mantiene una lista dei nodi visitati (*closed*)
- ▶ Prima di espandere un nodo si controlla se lo stato era stato già incontrato visitando un altro nodo.
- ▶ Se questo succede il nodo appena trovato non viene espanso
- ▶ Ottimale se il costo del nuovo cammino è sempre maggiore

L'algoritmo di ricerca su grafi

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Valutazione di una strategia

- ▶ Una strategia di ricerca è definita scegliendo l'**ordine di espansione dei nodi**
- ▶ Le strategie vengono valutate secondo le seguenti dimensioni:
 - ▶ **Completezza**: se la soluzione esiste viene trovata
 - ▶ **Ottimalità** (ammissibilità): trova la soluzione migliore, con costo minore
 - ▶ **Complessità nel tempo**: tempo richiesto per trovare la soluzione
 - ▶ **Complessità nello spazio**: memoria richiesta
- ▶ Le complessità di tempo e spazio si misurano in termini di
 - ▶ **b** : fattore di ramificazione massima dell'albero di ricerca
 - ▶ **d** : la profondità della soluzione a costo minimo
 - ▶ **m** : massima profondità dello spazio degli stati (potrebbe essere ∞)

Strategie non informate

- ▶ Usano solo l'informazione disponibile nella definizione del problema. Possono generare nuovi stati e possono distinguere uno stato obiettivo. Se uno stato non è un obiettivo non sanno capire quanto è promettente.
 - ▶ Ricerca in ampiezza
 - ▶ Ricerca di costo uniforme
 - ▶ Ricerca in profondità
 - ▶ Ricerca in profondità limitata
 - ▶ Ricerca con approfondimento iterativo
 - ▶ Ricerca bidirezionale
- ▶ Vs strategie di ricerca euristica (o informata): fanno uso di informazioni riguardo alla distanza stimata dalla soluzione

Confronto delle strategie

Criterio	BF	UC	DF	DL	ID	Bidir
Tempo	b^{d+1}	$b^{1+\lfloor C^*/\varepsilon \rfloor}$	b^m	b^l	b^d	$b^{d/2}$
Spazio	b^{d+1}	$b^{1+\lfloor C^*/\varepsilon \rfloor}$	b^m	b^l	b^d	$b^{d/2}$
Ottimale?	si(*)	si(**)	no	no	si(*)	si
Completa?	si	si(**)	no	si (+)	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 $l > d$)

Strategie informate

- ▶ Best first-search
 - ▶ Algoritmo Greedy
 - ▶ Algoritmo A
- ▶ A* search
 - ▶ Beam search
 - ▶ A* con approfondimento iterativo (IDA*)
 - ▶ Ricerca best-first ricorsiva (RBFS)
 - ▶ A* con memoria limitata (MA*) in versione semplice (SMA*)
- ▶ Euristiche

Ricerca euristica

- ▶ La ricerca esaustiva non è praticabile in problemi di complessità esponenziale
- ▶ Potremmo usare conoscenza del problema ed esperienza per riconoscere i cammini più promettenti.
- ▶ La conoscenza euristica (dal greco “eureka”) aiuta a fare scelte “oculate”
 - ▶ non evita la ricerca ma la **riduce**
 - ▶ consente in genere di trovare una **buona soluzione** in **tempi accettabili**
 - ▶ sotto certe condizioni garantisce **completezza** e **ottimalità**

Funzioni di valutazione euristica

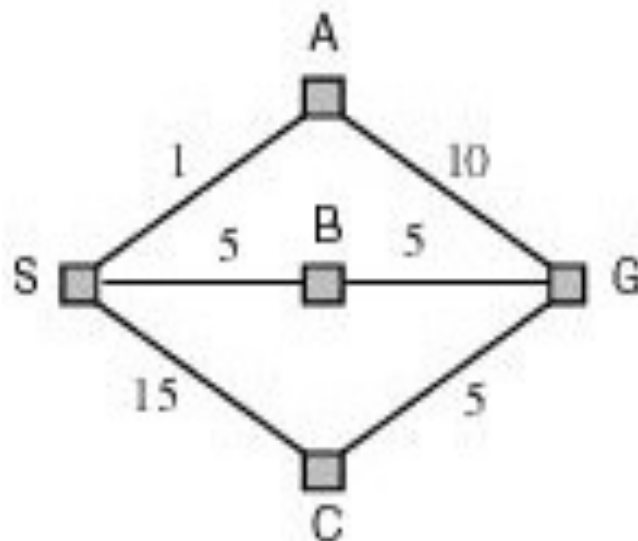
Conoscenza data tramite una *funzione di valutazione* dello stato, detta funzione di valutazione euristica:

$$f: n \rightarrow \mathbb{R}$$

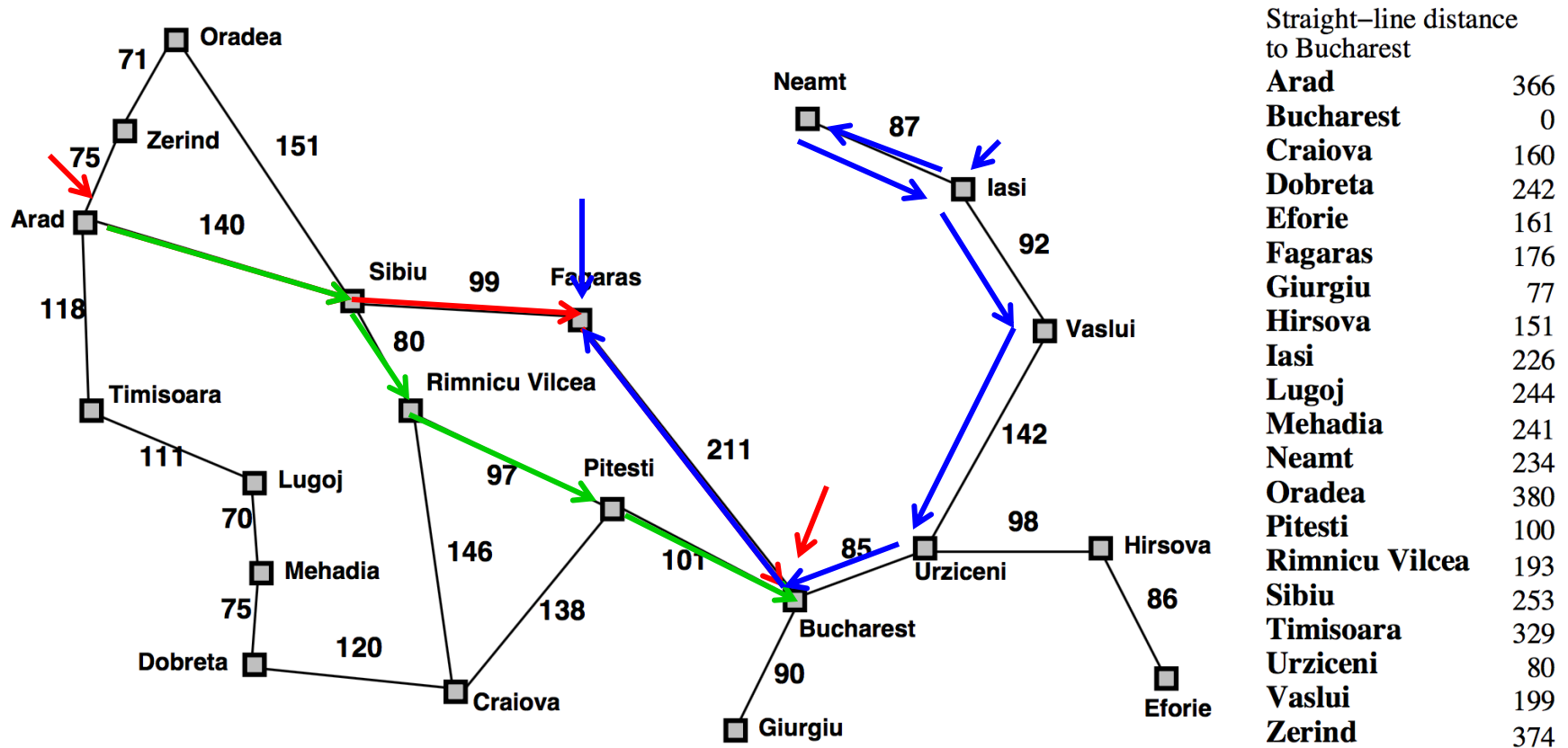
La funzione dipende solo dallo stato, fornendo una stima del costo di una soluzione che passa per il nodo

Esempi di euristica

- ▶ La città più vicina (o la città più vicina alla mèta in linea d'aria) nel *route-finding*
- Il numero delle caselle fuori posto nel gioco dell'otto
- Il vantaggio in pezzi nella dama o negli scacchi



Ricerca greedy: esempio



Da Arad a Bucarest ...

Greedy: Arad, Sibiu, Fagaras, Bucharest (450)

Ottimo: Arad, Sibiu, Rimnicu, Pitesti, Bucharest (418)

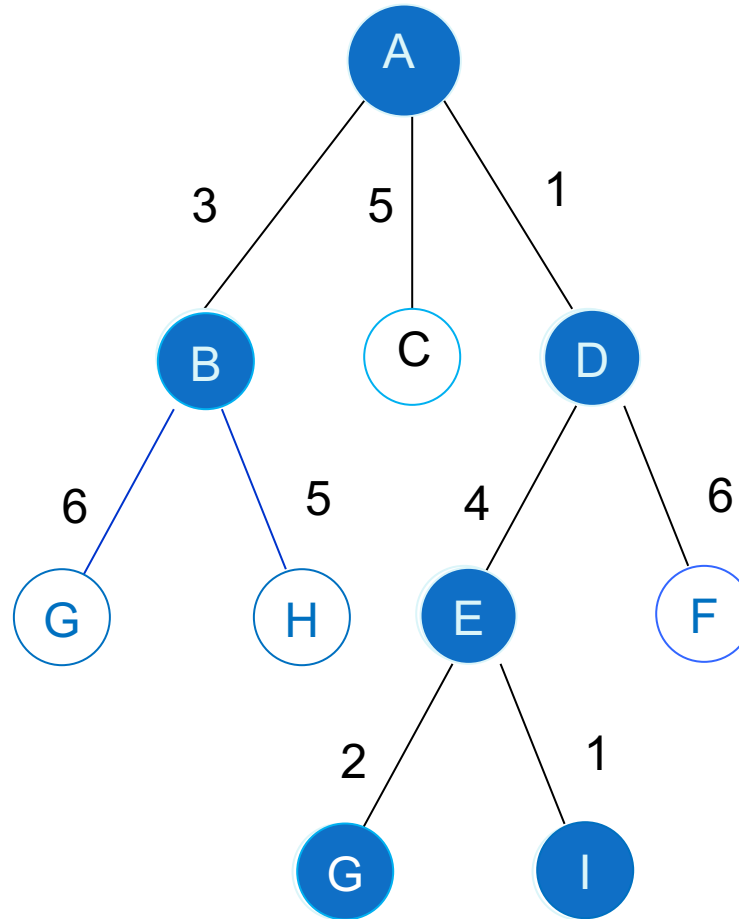
Da Iasi a Fagaras: ... **falsa partenza**

Algoritmo Best-first

- ▶ Ad ogni passo si sceglie il nodo sulla frontiera per cui il valore della f è migliore (il nodo più promettente).
- ▶ Migliore significa 'minore' in caso di stima della distanza della soluzione
- ▶ Implementata da una *coda con priorità* che ordina in base al valore della funzione di valutazione euristica
- ▶ Diverse funzioni di valutazione determinano diverse versioni della ricerca BEST-FIRST
- ▶ La ricerca guidata dal costo si può considerare un caso particolare di ricerca best-first, in cui $f=g$
 - ▶ La funzione di valutazione è $g(n)$: il costo del cammino dallo stato iniziale a n . Informazione euristica nulla

Strategia best-first: esempio

Passo 7

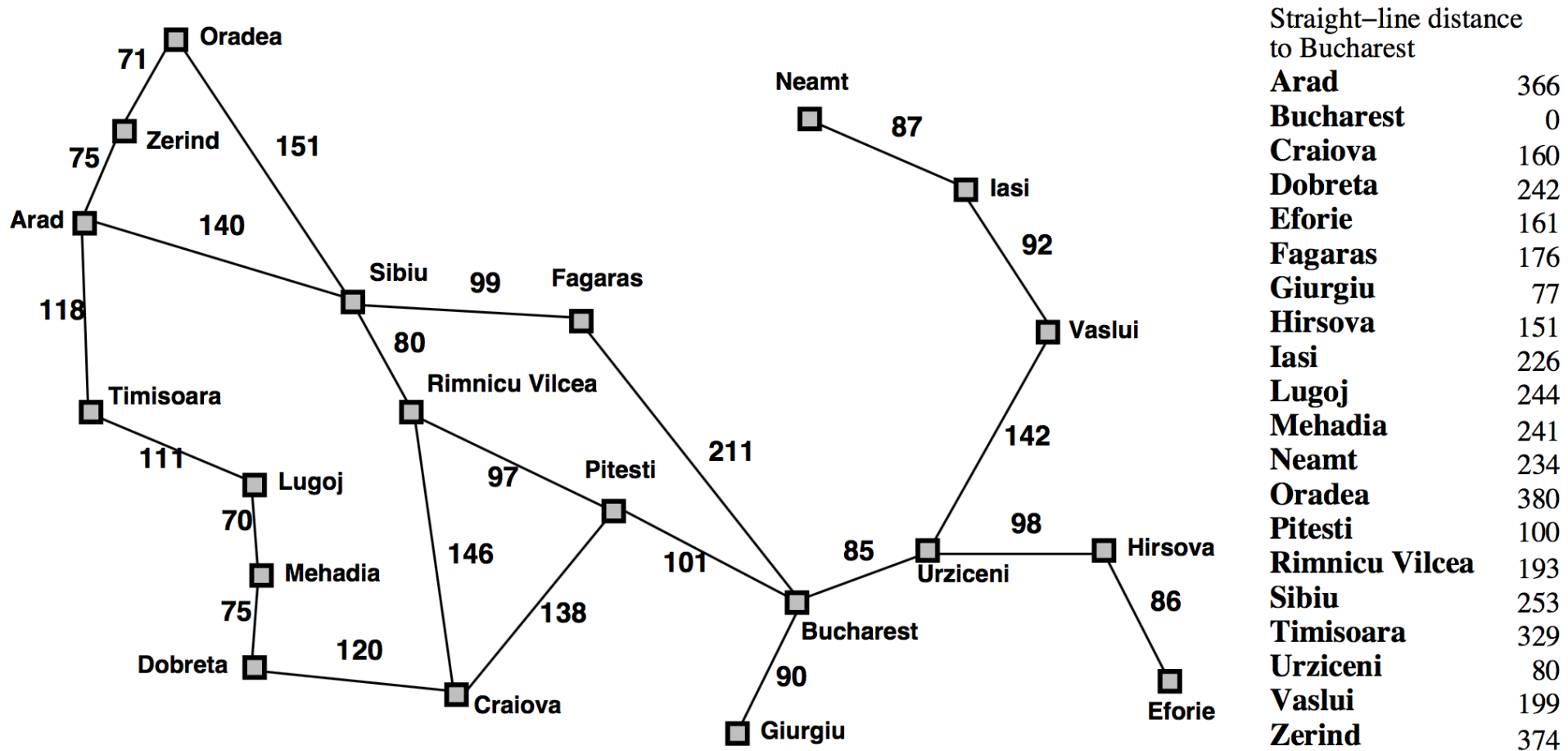


La Best First non è in generale completa, né ottimale

Ricerca greedy best-first

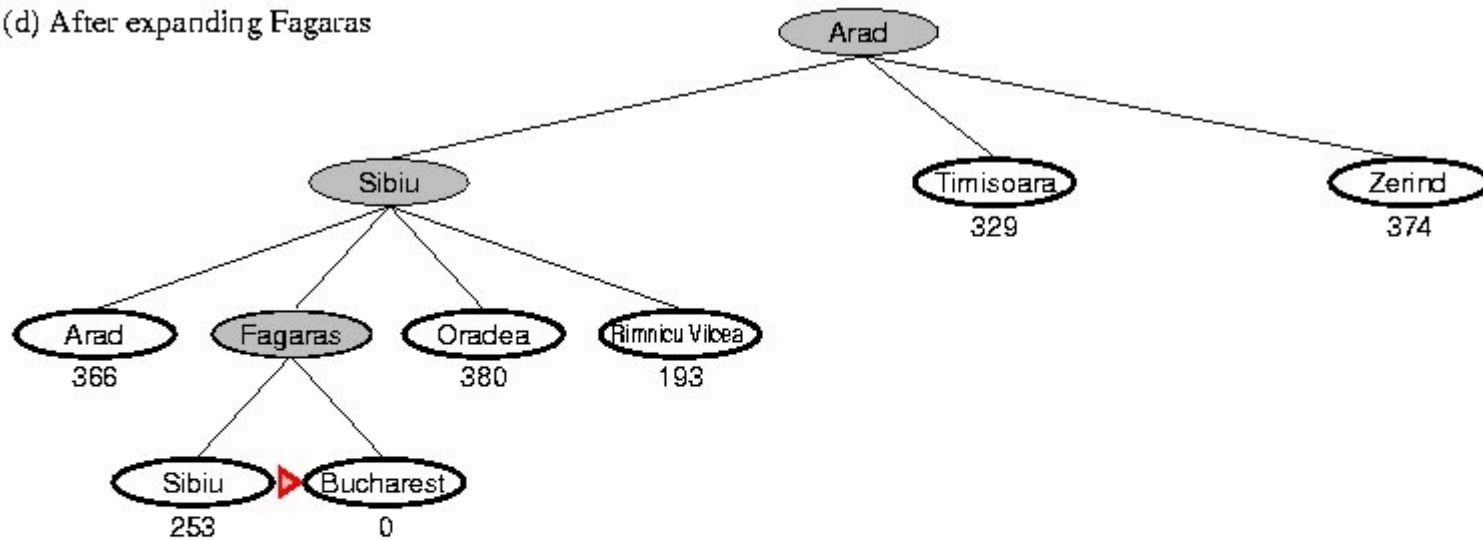
- ▶ Si usa come euristica una stima della distanza della soluzione, da ora in poi $h(n)$ [$h \geq 0$]
- ▶ *Esempio*: ricerca greedy per Route Finding
 $h(n)$ = distanza in linea d'aria tra lo stato di n e la destinazione

Ricerca greedy: esempio

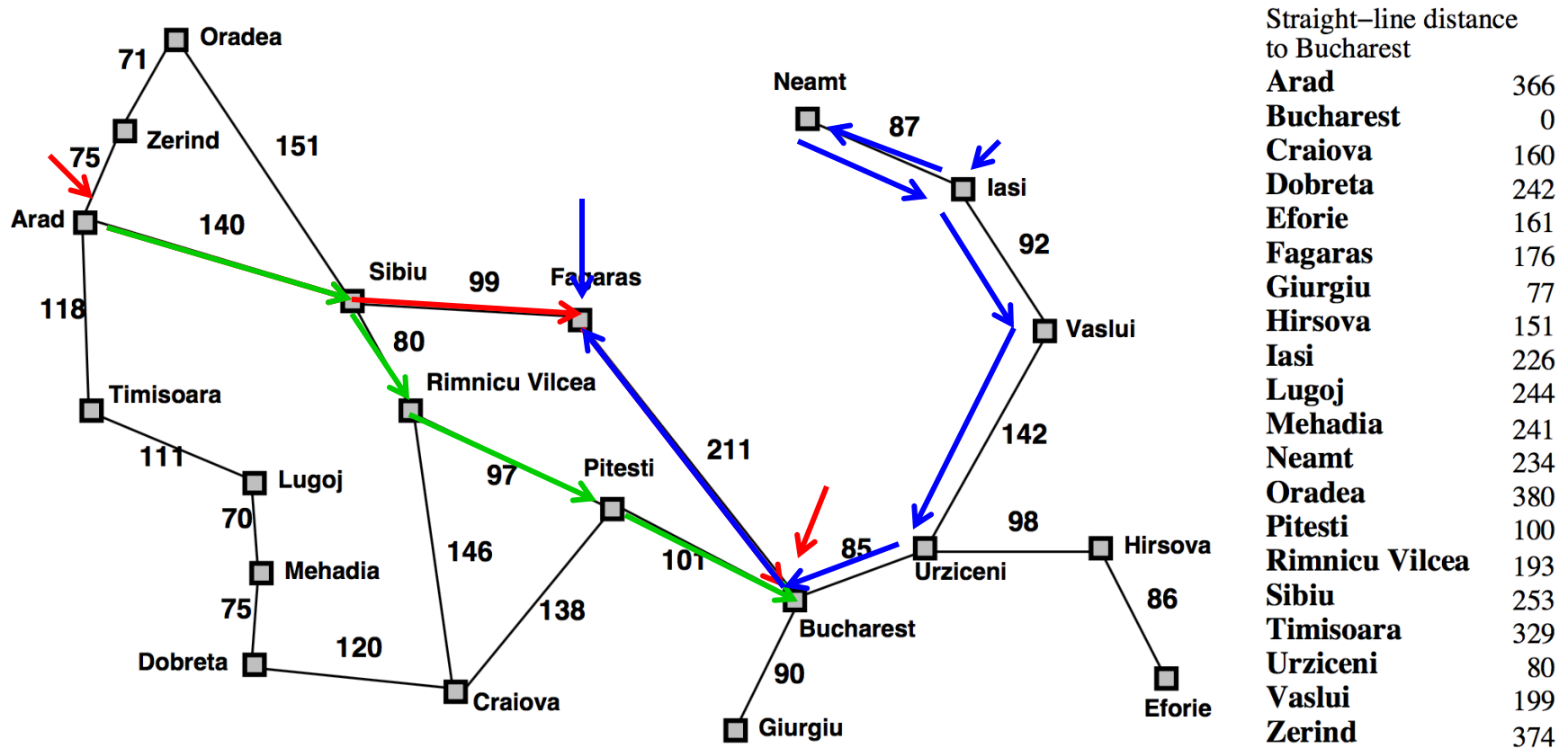


Route finding con Greedy Best-First Search

(d) After expanding Fagaras



Ricerca greedy: esempio



Da Arad a Bucarest ...

Greedy: Arad, Sibiu, Fagaras, Bucharest (450)

Ottimo: Arad, Sibiu, Rimnicu, Pitesti, Bucharest (418)

Da Iasi a Fagaras: ... **falsa partenza**

Algoritmo A: definizione

- ▶ Si può dire qualcosa di f per avere garanzie di completezza e ottimalità?
- ▶ Un algoritmo A è un algoritmo Best First con una funzione di valutazione dello stato del tipo:
$$f(n) = g(n) + h(n), \text{ con } h(n) \geq 0 \text{ e } h(goal)=0$$
 - ▶ $g(n)$ è il costo del cammino percorso per raggiungere n
 - ▶ h una stima del costo per raggiungere da n un nodo goal

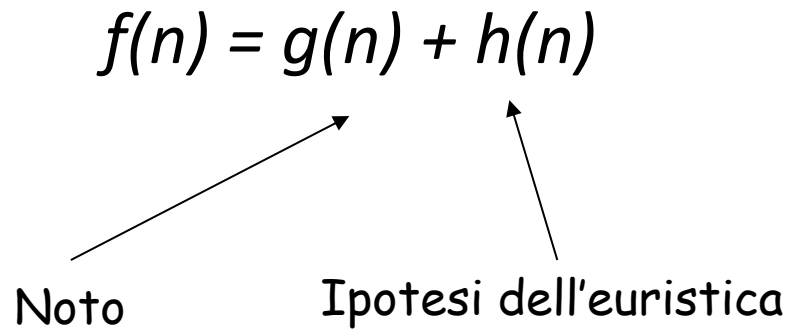
Casi particolari dell'algoritmo A:

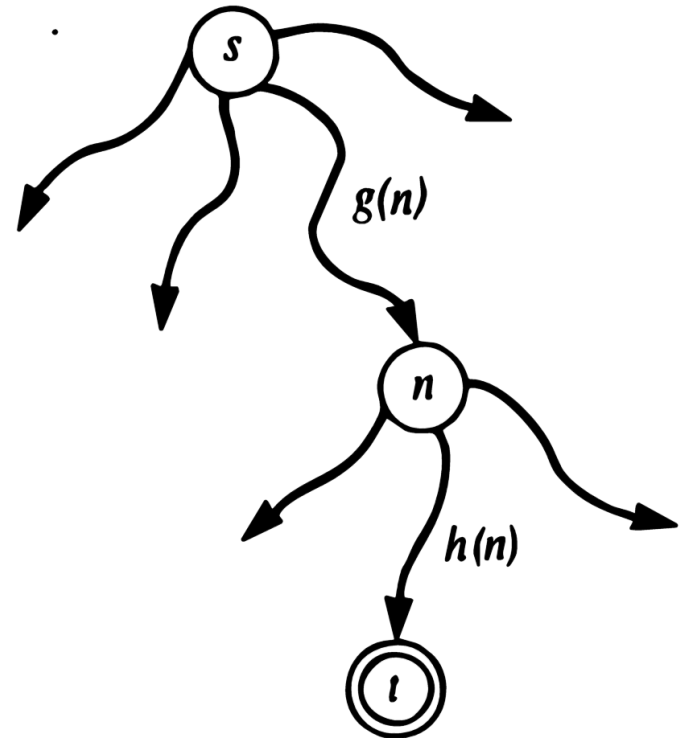
- ▶ Se $h(n) = 0$ [$f(n) = g(n)$] si ha Ricerca a Costo Uniforme
- ▶ Se $g(n) = 0$ [$f(n) = h(n)$] si ha Greedy Best First

Algoritmo A: definizione

$$f(n) = g(n) + h(n)$$

Noto Ipotesi dell'euristica





Algoritmo A: esempio

Esempio nel gioco dell'otto

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

$$f(n) = \# \text{mosse} + \# \text{caselle-fuori-posto}$$

$$f(\text{Start}) = 0 + 7 \qquad \text{Dopo } \leftarrow, \downarrow, \rightarrow, \uparrow \quad f = 4 + 7$$

Algoritmo A*: la stima ideale

Funzione di valutazione ideale (*oracolo*):

$$f^*(n) = g^*(n) + h^*(n)$$

$g^*(n)$ costo del cammino minimo da radice a n

$h^*(n)$ costo del cammino minimo da n a goal

$f^*(n)$ costo del cammino minimo da radice a goal,
attraverso n

Normalmente:

$g(n) \geq g^*(n)$ e $h(n)$ è una stima di $h^*(n)$

Algoritmo A*: definizione

Definizione: Euristica ammissibile

$\forall n . h(n) \leq h^*(n)$ *h è una sottostima*

Es. l'euristica della distanza in linea d'aria

Definizione: Algoritmo A*

Un algoritmo A in cui *h* è una funzione euristica ammissibile.

Teorema: Gli algoritmi A* sono ottimali.

Corollario: BF e UC sono ottimali ($h(n)=0$)

Outline

- ▶ Ricerca Locale
 - ▶ Hill-climbing
 - ▶ Simulated annealing
 - ▶ Algoritmi genetici
- ▶ Ricerca con azioni non deterministiche
 - ▶ Alberi AND-OR
- ▶ Online
 - ▶ Online DFS
 - ▶ LRTA*

Algoritmi di ricerca locale

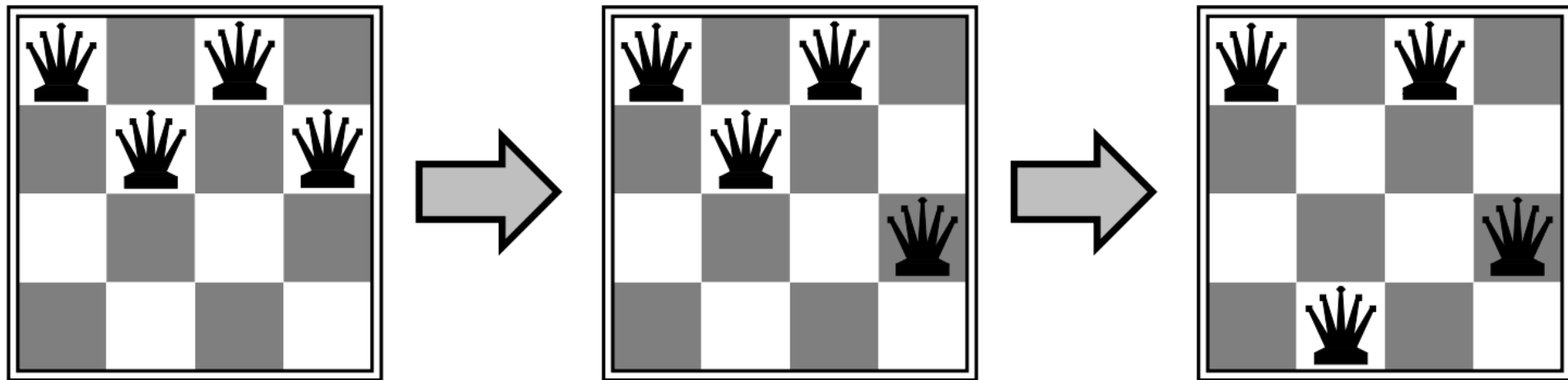
- ▶ In molti problemi di ottimizzazione, il **cammino** dallo stato iniziale a quello obiettivo è irrilevante
 - ▶ La soluzione è costituita dallo stato goal stesso
- ▶ Quindi lo spazio degli stati è dato dall'insieme delle configurazioni “complete”
 - ▶ Trovare una configurazione **ottima**, es. TSP (problema del commesso viaggiatore), oppure
 - ▶ Trovare una configurazione che soddisfi dei vincoli, es. orario
- ▶ In tali casi, si possono usare gli algoritmi di **miglioramento iterativo**
 - ▶ Mantengono uno stato corrente e tentano di migliorarlo

Algoritmi di ricerca locale

- ▶ La ricerca locale è basata sull'esplorazione iterativa di “soluzioni vicine”, che possono migliorare la soluzione corrente mediante modifiche locali.
- ▶ Struttura dei “vicini” (neighborhood). Una struttura dei “vicini” è una funzione F che assegna a ogni soluzione s dell'insieme di soluzioni S un insieme di soluzioni $N(s)$ sottoinsieme di S .
- ▶ La scelta della funzione F è fondamentale per l'efficienza del sistema e definisce l'insieme delle soluzioni che possono essere raggiunte da s in un singolo passo di ricerca dell'algoritmo.
- ▶ Tipicamente è definita attraverso le possibili mosse.
- ▶ La soluzione trovata da un algoritmo di ricerca locale non è detto sia ottima globalmente, ma può essere ottima rispetto ai cambiamenti locali.

Esempio n-regine

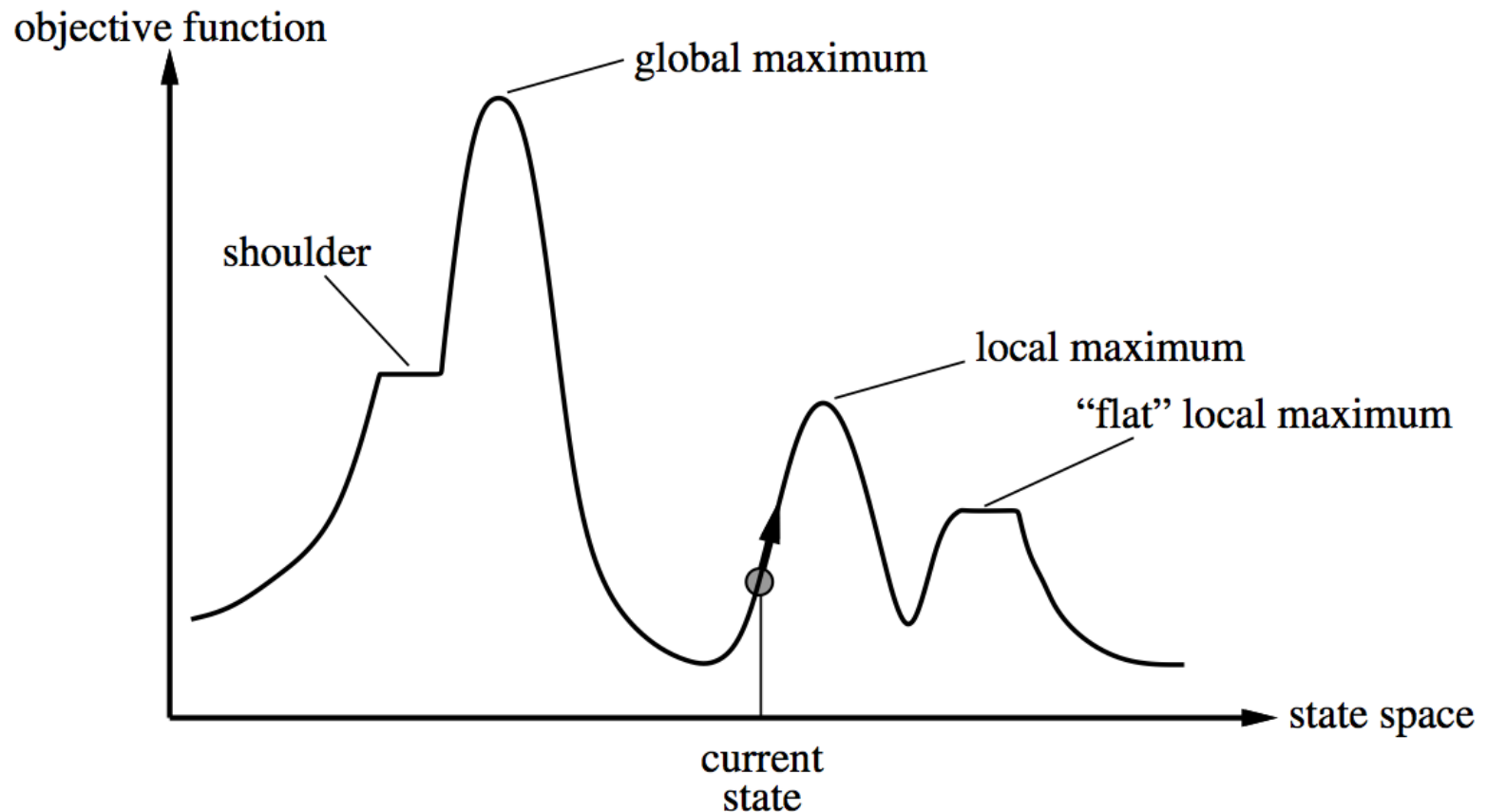
- ▶ Disporre n regine su una scacchiera $n \times n$ senza che si minaccino
- ▶ Muovere una regina in modo da minimizzare il numero di minacce



Algoritmi di ricerca locale

- ▶ Efficienti in occupazione di memoria: tengono traccia solo dello stato corrente (non necessario un puntatore al padre) e si spostano su stati adiacenti
- ▶ Per problemi in cui:
 - ▶ il cammino che si segue non è importante: basta trovare la soluzione
 - ▶ tutti gli elementi della soluzione sono nello stato ma alcuni vincoli sono violati. Es. le regine nella versione completa.
- ▶ Gli stati come punti su una superficie su cui l'algoritmo provoca movimento: i picchi sono massimi locali o soluzioni ottimali (se la f è da ottimizzare).

Algoritmi di ricerca locale



Ricerca in salita (Hill climbing)

- ▶ Vengono generati i successori e valutati; viene scelto un nodo, che migliora la valutazione dello stato attuale:
 - ▶ il primo (salita semplice)
 - ▶ il migliore (salita rapida)
 - ▶ uno a caso (stocastico)
- ▶ Se non ce ne sono l'algoritmo termina (non si tiene traccia degli altri)

Esempio Hill-climbing

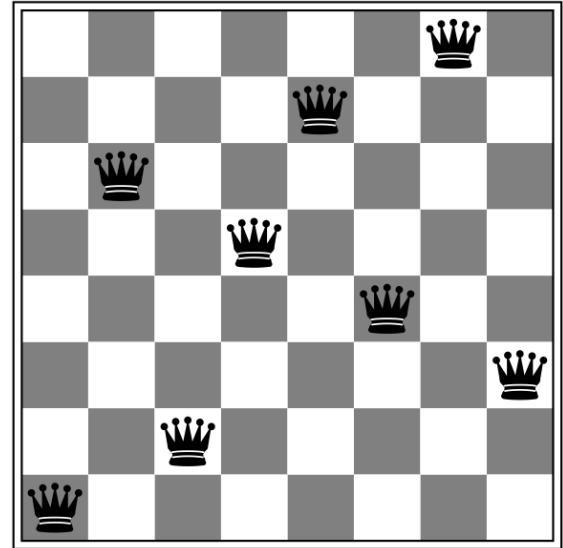
- ▶ Problema 8-regine (formulazione a stato completo).
- ▶ Funzione successore: muove una singola regina in un altro posto nella stessa colonna.
- ▶ Funzione euristica $h(n)$:
il numero di coppie di regine che si minacciano tra di loro
(direttamente o indirettamente).

Esempio Hill-climbing

a)

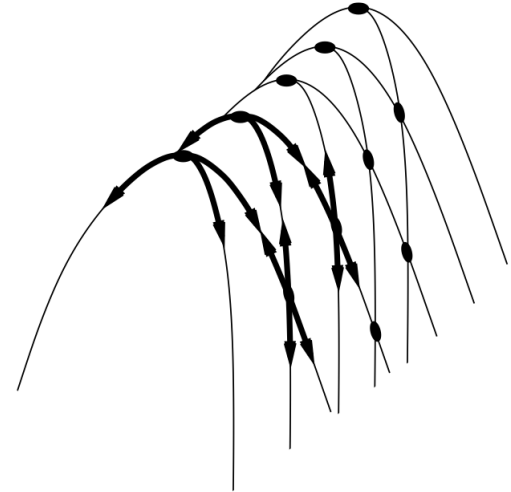
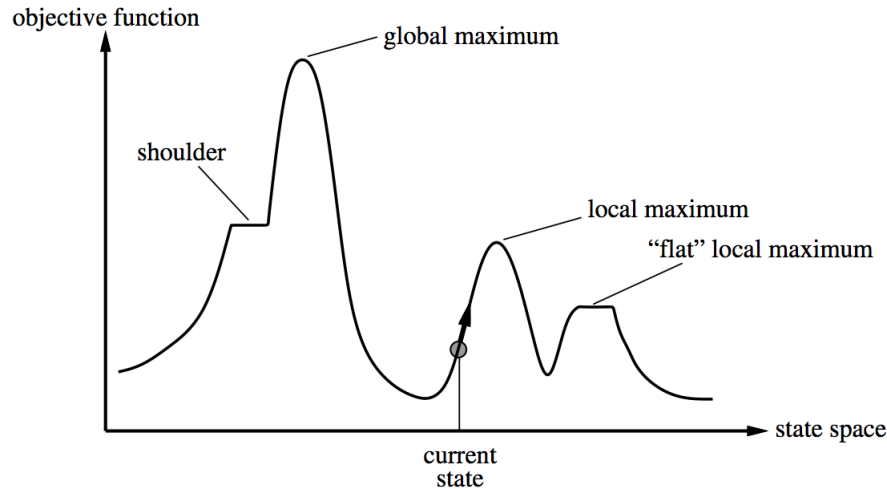
18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♔	13	16	13	16
♔	14	17	15	♔	14	16	16
17	♔	16	18	15	♔	15	♔
18	14	♔	15	15	14	♔	16
14	14	13	17	12	14	12	18

b)



- a) Mostra uno stato con $h=17$ ed il valore h per ogni possibile successore. Quanti successori sono?
- b) Un minimo locale nello spazio degli stati delle 8 regine ($h=1$), raggiunto dopo 5 passi

Problemi

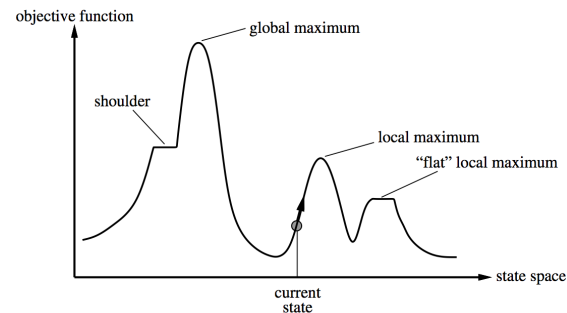


- ▶ **Massimi locali** = picco più alto degli stati vicini ma inferiore al massimo globale
- ▶ **Cresta** = sequenza di massimi locali molto difficili da esplorare per gli algoritmi greedy
- ▶ **Plateau** = un'area piatta del panorama dello spazio degli stati (la funzione di valutazione è piatta).
- ▶ Problema delle regine: si blocca l'86% delle volte.

Mossa Laterale

- ▶ Se si raggiunge un plateau (il successore ha lo stesso valore dello stato corrente) si potrebbe continuare la ricerca facendo una mossa laterale:

- ▶ **Spostamento in uno stato**
- ▶ **con identico valore di h**



- ▶ Bisogna stare attenti ad evitare cicli, specialmente nel caso di massimi (minimi) locali piatti
- ▶ Tipica soluzione: porre un limite massimo al numero consecutivo di mosse laterali
 - ▶ Ad esempio per il problema delle regine se si pone il limite a 100 mosse laterali la percentuale di successo passa da 14% a 94%
 - ▶ Ovviamente aumentano i costi (da 3-4 mosse si passa a 21, max 64 se fallisce)

Varianti Hill-climbing

- ▶ Hill-climbing stocastico
 - ▶ Selezione casuale tra tutte le mosse che vanno verso l'alto.
 - ▶ La probabilità della scelta può essere influenzata dalla “pendenza” delle mosse.
- ▶ Hill-climbing con prima scelta
 - ▶ Come hill climbing stocastico generando casualmente i successori finché non si ottiene uno migliore
 - ▶ Utile quando ci sono migliaia di successori.
- ▶ Hill-climbing con riavvio casuale
 - ▶ Ripete le ricerche hill-climbing partendo da stati iniziali casuali fino a raggiungere un obiettivo.
 - ▶ Se la probabilità di successo dell'algoritmo è p saranno necessari $1/p$ ripartenze per trovare la soluzione (nel caso delle 8 regine corrisponde a 7 iterazioni)

Ricerca local beam

- ▶ Si tiene traccia di k stati anziché uno solo
- ▶ Ad ogni passo si generano i successori di tutti i k stati
 - ▶ Se si trova un goal ci si ferma
 - ▶ Altrimenti si prosegue con i k migliori
- ▶ Le informazioni vengono passate tra i vari thread di ricerca!!
- ▶ Nella variante *local beam stocastica*, si scelgono k successori a caso con probabilità maggiore per i migliori (selezione naturale).
- ▶ La terminologia:
 - ▶ organismo [stato]
 - ▶ progenie [successori]
 - ▶ fitness [il valore della f], idoneità

Algoritmi evolutivi

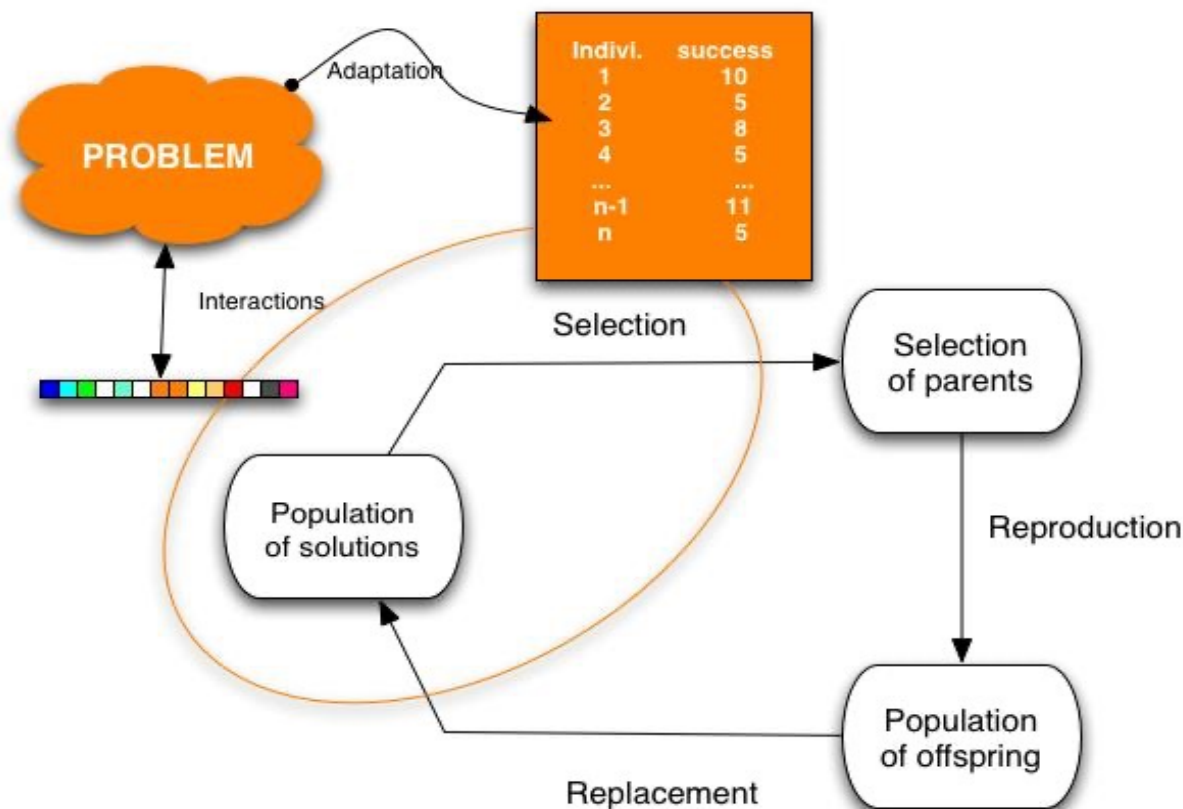
- ▶ Considera il problema di ottimizzazione in analogia con la teoria evolutiva -> Simulazione della selezione naturale
- ▶ Le configurazioni sono viste come individui di una popolazione
- ▶ Gli individui meno adatti muoiono senza riprodursi
- ▶ Consente agli individui migliori di riprodursi più spesso
- ▶ Ogni generazione dovrebbe essere nel complesso migliore rispetto alla precedente
- ▶ Se aspettiamo abbastanza a lungo la popolazione dovrebbe evolvere verso individui migliori (vale a dire, con valori massimi di f)

Algoritmi genetici

- ▶ Popolazione: k stati generati casualmente
- ▶ Ogni individuo è rappresentato come stringa
 - ▶ Es. 24748552 stato delle 8 regine
 - ▶ La stringa è il cromosoma che descrive l'individuo
- ▶ Gli individui sono valutati da una funzione di *fitness*
 - ▶ Es. numero di coppie di regine che non si attaccano
- ▶ Si scelgono gli individui con probabilità proporzionale alla fitness

Algoritmi genetici

- Variazione del local beam stocastico con ricombinazione



Popolazione iniziale

- ▶ La popolazione iniziale viene creata generando gli individui in maniera casuale.
- ▶ Il numero N_p di individui generati è la dimensione della popolazione
- ▶ N_p è scelto in maniera euristica ed è dipendente dalla natura della funzione obiettivo e dalle dimensioni dello spazio di ricerca
- ▶ Negli Algoritmi Genetici standard N_p rimane fisso durante l'evoluzione

Fitness function

- ▶ Nella scelta degli elementi genitori si tendono a favorire elementi della popolazione con un elevato valore di fitness (adattamento).
- ▶ Come ovvia (ma non necessariamente unica) misura di fitness si può pensare al valore della funzione obiettivo f.
- ▶ Ma altri valori possono essere tenuti in considerazione nella definizione della fitness (ad esempio, la diversità di un elemento rispetto agli altri membri della popolazione).

L'operatore di selezione

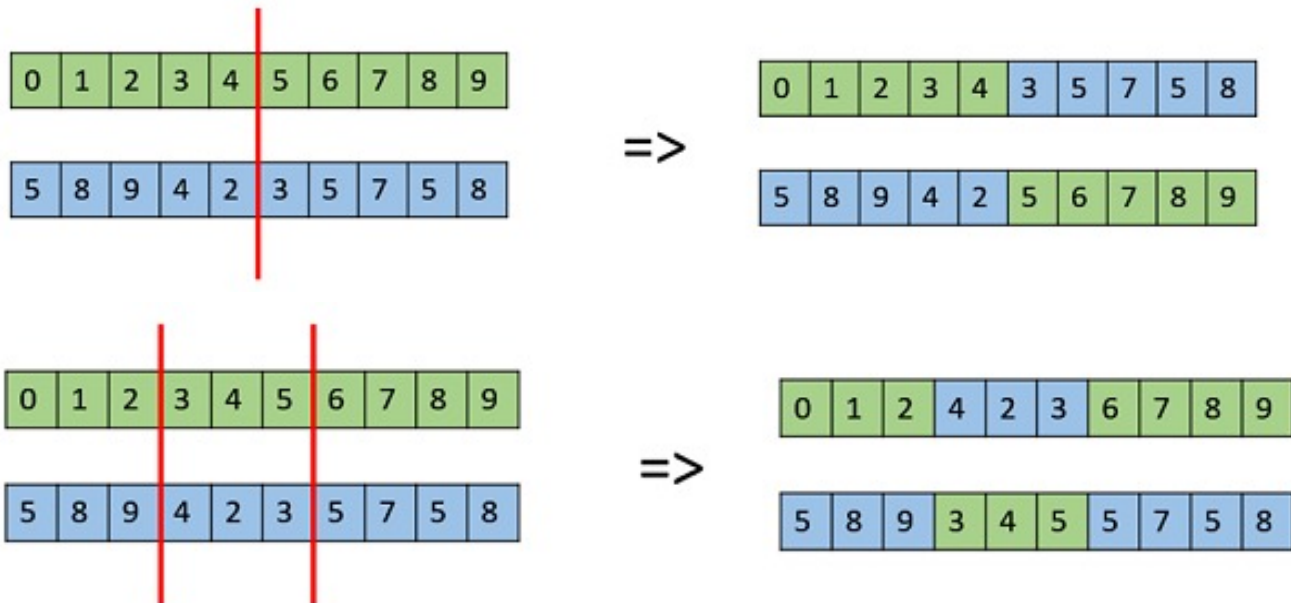
- ▶ La selezione naturale Darwiniana sostiene che gli **individui più “forti”** abbiano maggiori probabilità di sopravvivere nell'ambiente in cui vivono e, dunque, **maggior probabilità di riprodursi**
- ▶ Nel contesto degli algoritmi genetici, **gli individui più forti sono quelli con fitness più alta**, poiché risolvono meglio di altri il problema di ricerca dato; per questo **essi devono essere privilegiati nella fase di selezione** di quegli individui che potranno riprodursi dando luogo a nuovi individui

Selection

- ▶ **roulette wheel:** la popolazione è rappresentata mediante una ruota di roulette con i settori proporzionali alla fitness degli elementi; la pallina viene lanciata N_p volte e gli elementi che hanno fitness migliore hanno probabilità maggiore di essere scelti.
- ▶ **tournament selection:** vengono scelti 2 individui a caso e quello tra i due che ha la fitness migliore viene selezionato per la nuova popolazione; l'operazione viene ripetuta N_p volte; prima della selezione gli individui vengono mescolati (shuffle).

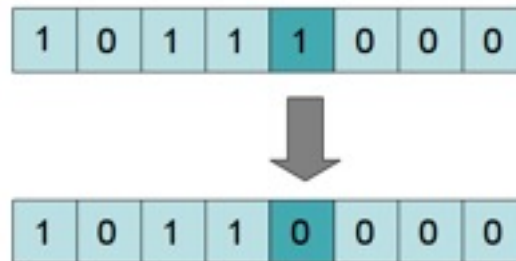
Operatori genetici

- ▶ Per ogni coppia, il crossover viene applicato con probabilità P_c
- ▶ Si scelgono a caso due individui nel mating pool (genitori) e un punto di taglio (punto di crossover) su di essi. Le porzioni di genotipo alla destra del punto di crossover vengono scambiate generando due discendenti.

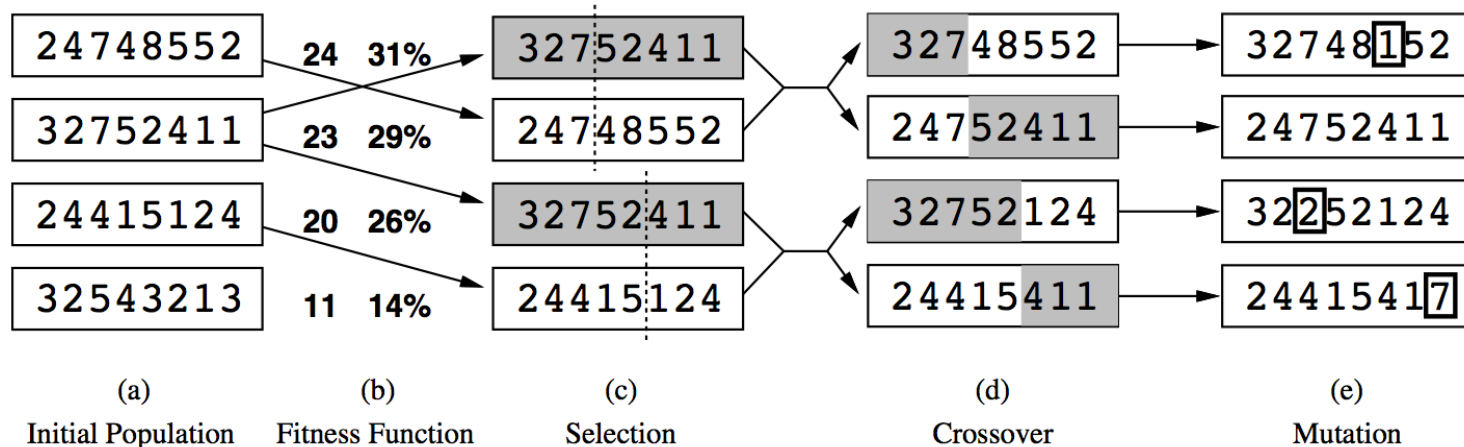


Operatori genetici

- ▶ Per ogni figlio, la mutation viene applicata con probabilità P_m .
- ▶ L'operatore di crossover ricombina il materiale genetico esistente. L'operatore di mutation introduce nuovo materiale genetico.
- ▶ P_c e P_m si scelgono in maniera euristica e in genere $P_m < P_c$ (possono variare durante l'evoluzione).



Esempio



- ▶ Per ogni coppia viene scelto un punto di cross-over e i due genitori producono due figli scambiandosi pezzi
- ▶ Viene infine effettuata una *mutazione* casuale che dà luogo alla prossima generazione.

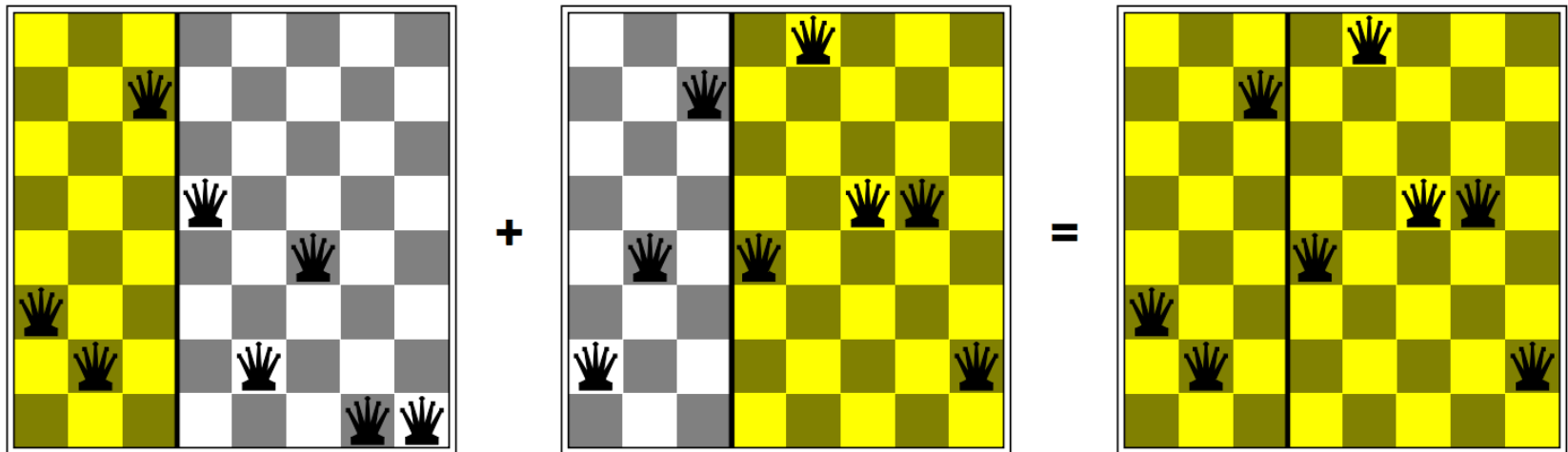
N.B. $24 / (24+23+20+11) = 31\%$

Criterio di arresto

- ▶ Il meccanismo di selezione, ricombinazione e calcolo della fitness viene iterato.
- ▶ L'evoluzione termina quando viene raggiunto l'ottimo, se questo è noto.
- ▶ Altrimenti l'evoluzione termina quando:
 1. viene raggiunto il numero massimo N_g di generazioni; il numero totale N_t di valutazioni della funzione obiettivo
$$N_t = N_p * N_g$$
 2. un indicatore di convergenza (uniformità della popolazione, mancanza di progressi nell'evoluzione) raggiunge un determinato valore

8-regine

- ▶ Gli algoritmi genetici richiedono che gli stati siano codificati come stringhe
- ▶ Il crossover aiuta se e solo se le sottostringhe sono componenti significative



$$3 \ 2 \ 7 \ 5 \ 2 \ 4 \ 1 \ 1 \ + \ 2 \ 4 \ 7 \ 4 \ 8 \ 5 \ 5 \ 2 \ = \ 3 \ 2 \ 7 \ 4 \ 8 \ 5 \ 5 \ 2$$