Oltre la ricerca classica

Cap 4 – *Ricerca locale, cenni di ricerca online*

Alessio Micheli

a.a. 2022/2023

Credits: Maria Simi

Russell-Norvig

Risolutori "classici"

- Gli agenti *risolutori di problemi* "classici" assumono:
  - Ambienti completamente osservabili
  - Ambienti deterministici
  - Sono nelle condizioni di produrre *offline* un piano (una sequenza di azioni) che può essere eseguito senza imprevisti per raggiungere l'obiettivo.

Verso ambienti più realistici - Indice

- La ricerca sistematica, o anche euristica, nello spazio di stati è troppo costosa
  - Metodi di ricerca locale
- Assunzioni sull'ambiente da riconsiderare
  - Azioni non deterministiche e ambiente parzialmente osservabile
    - Piani condizionali, ricerca AND-OR, stati credenza
  - Ambienti sconosciuti e problemi di esplorazione (percezioni forniscono nuove informazioni dopo l'azione)
    - Ricerca *online*

Part 1
Ricerca locale

Assunzioni per ricerca *locale*

- Gli algoritmi visti esplorano gli spazi di ricerca alla ricerca di un goal e restituiscono un ***cammino soluzione***
- Ma a volte lo stato **goal** è la soluzione del problema.
- Gli algoritmi di *ricerca locale* sono adatti per problemi in cui:
  - La sequenza di azioni *non* è importante: quello che conta è unicamente lo stato goal
  - Tutti gli elementi della soluzione sono nello stato ma alcuni vincoli sono violati

    Es. *le regine* nella formulazione a stato completo.

    Ci interessa di ottenere la soluzione ma non il path

Algoritmi di ricerca locale

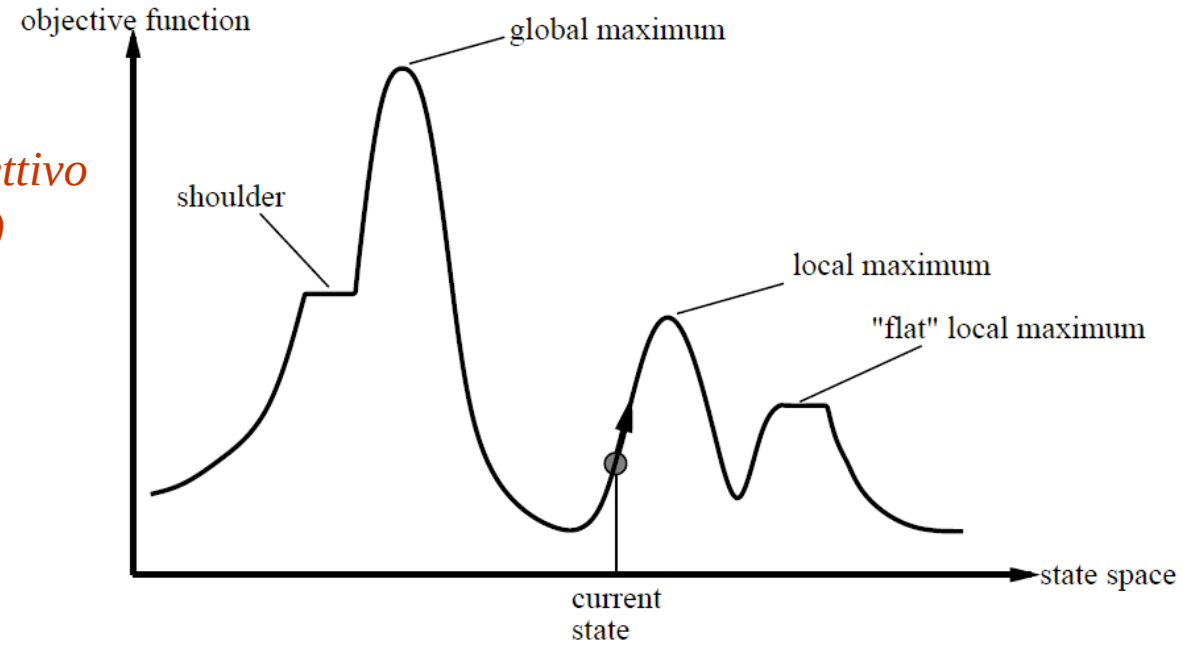
- Non sono sistematici
- Tengono traccia solo del nodo corrente e si spostano su nodi adiacenti
- Non tengono traccia dei cammini (non servono in uscita!)

1. Efficienti in occupazione di memoria
2. Possono trovare soluzioni ragionevoli anche in spazi molti grandi e infiniti, come nel caso di spazi continui

- Utili per risolvere problemi di ottimizzazione
  - *lo stato migliore secondo una funzione obiettivo (f)*
  - *lo stato di costo minore (ma non il path)*
  - *Esempi:*
  - *minimizare numero di regine sotto attacco (f all'obiettivo?)*
  - *training di un modello di Machine Learning*

Panorama dello spazio degli stati

*f euristica di costo della funzione obiettivo (**non** del cammino)*



- Uno stato ha una posizione sulla superficie e una altezza che corrisponde al valore della f. di valutazione (f. obiettivo)
- Un algoritmo provoca movimento sulla superficie
- Trovare l'avvallamento più basso (e.g. min costo) o il picco più alto (e.g. max di un obiettivo)

Ricerca in salita (*Hill climbing*) *steepest ascent/descent

- Ricerca locale *greedy*
- Vengono generati i successori e valutati; viene scelto un nodo che migliora la valutazione dello stato attuale (non si tiene traccia degli altri [no albero di ricerca in memoria]):
  - il migliore ➔ Hill climbing a salita rapida/ripida (*)
  - uno a caso (tra quelli che salgono) ➔ Hill climbing stocastico (anche dipendendo da pendenza)
  - il primo ➔ Hill climbing con prima scelta (il primo generato tra tanti possibili)
- Se non ci sono stati successori migliori l'algoritmo termina con fallimento

L'algoritmo Hill climbing - steepest ascent

function Hill-climbing (*problema)*

returns uno stato che è un massimo locale [esercizio: fare con min.]

nodo-corrente = CreaNodo(*problema*.Stato-iniziale)

loop do

vicino = il successore di *nodo-corrente* di valore più alto

if *vicino*.Valore ≤ *nodo-corrente*.Valore **then**

return *nodo-corrente*.Stato *// interrompe la ricerca*

nodo-corrente = *vicino*

// (altrimenti, se vicino e' migliore, continua)

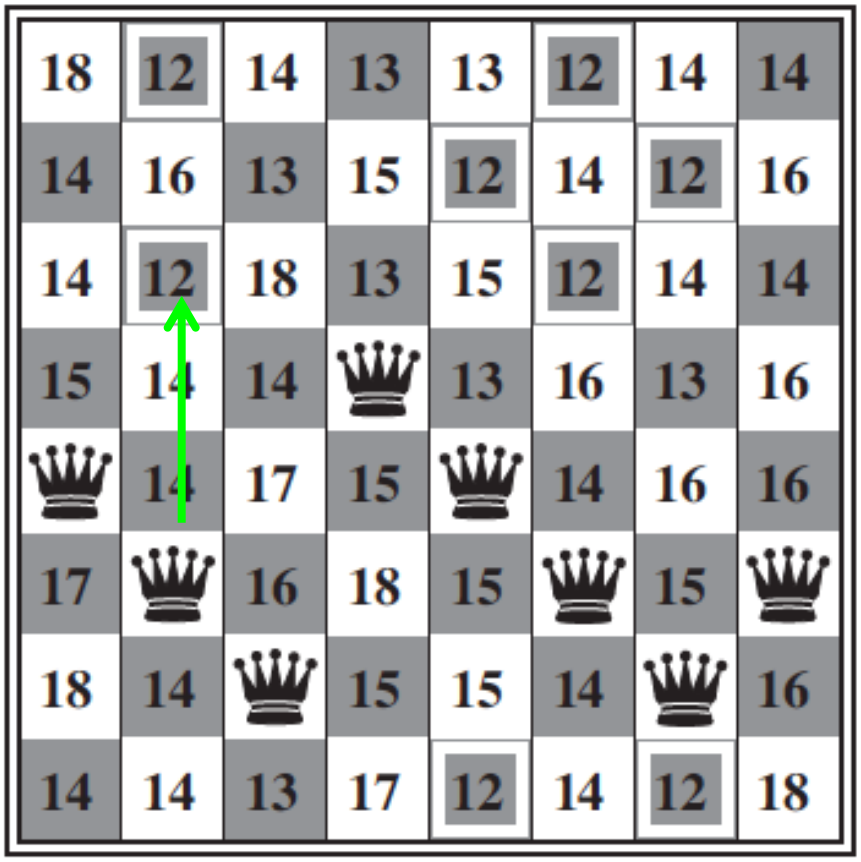
- Nota: si prosegue solo se il vicino (piu alto) è migliore dello stato corrente
  → Se tutti i vicini sono peggiori o uguali si ferma.
- **Non** c'è frontiera a cui ritornare, si tiene 1 solo stato
- Tempo: numero cicli variabile in base al punto di partenza

In Python

```
def hill_climbing(problem):   """ Ricerca locale - Hill-climbing."""
    current = Node(problem.initial_state)
    while True:
        neighbors =  [current.child_node(problem, action) for action in
                      problem.actions(current.state)]
        if not neighbors: # se current non ha successori esci e restituisci current
            break
        # scegli il vicino con valore piu' alto (sulla funzione problem.value)
        neighbor = (sorted(neighbors,key = lambda x:problem.value(x), reverse = True))[0]
        if problem.value(neighbor) <= problem.value(current):
            break
        else:
            current = neighbor     # (altrimenti, se vicino  e' migliore, continua)
    return current
```

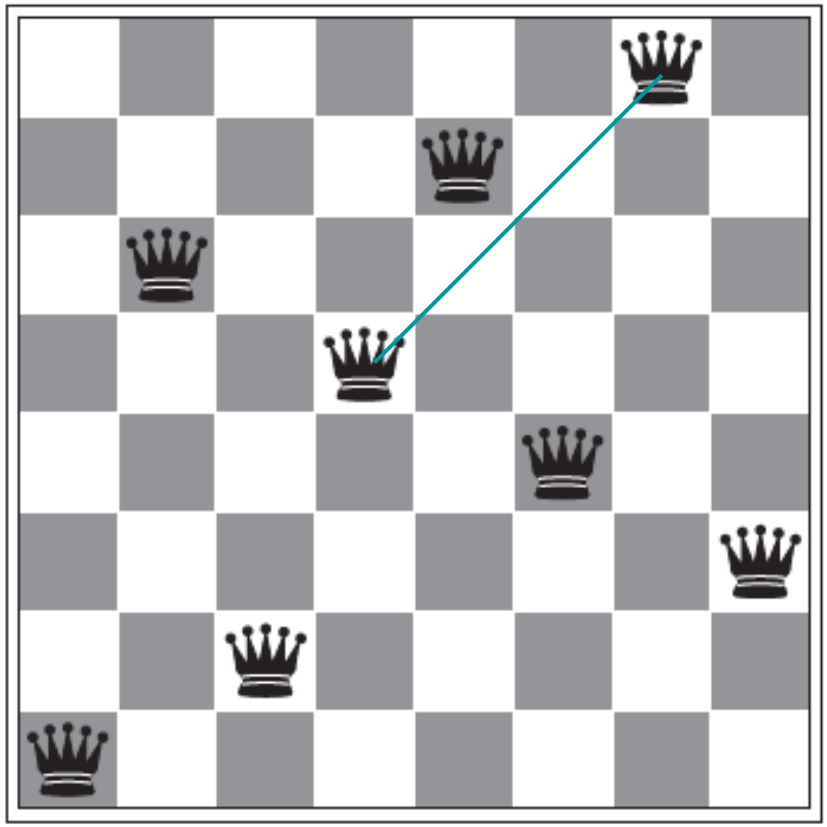
Il problema delle 8 regine (formulazione a stato completo)

- *Costo h* (stima euristica del costo f)*: numero di coppie di regine che si attaccano a vicenda* (valore 17 nell'es. →)
- Si cerca il minimo
- I numeri sono i valori dei successori (7x8) [7 posizioni per ogni regina, su ogni colonna]
- Tra i migliori (di pari valore 12) si sceglie a caso
- Mininimo globale = 0

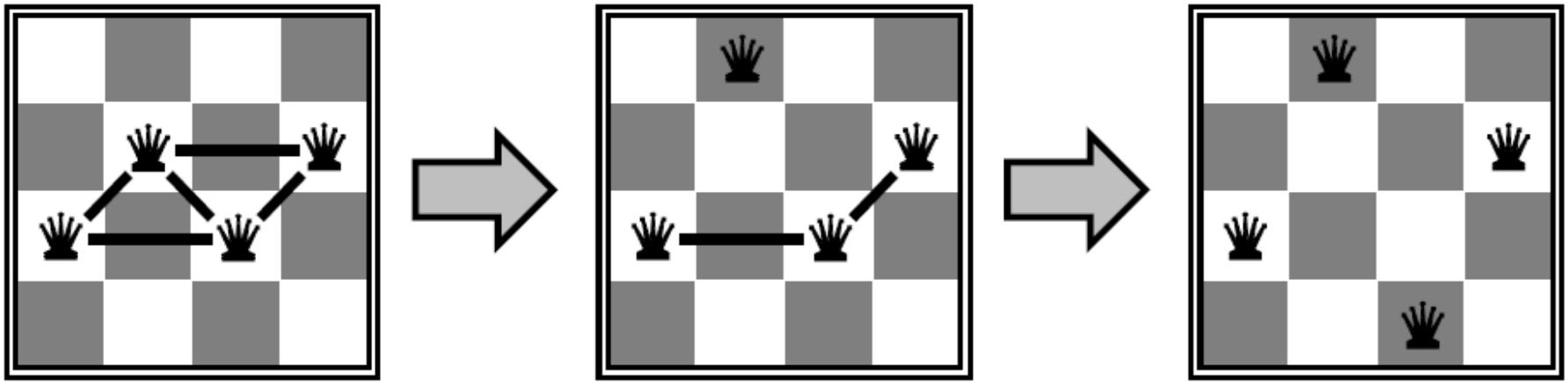


Esempio negativo: un minimo locale

- $h = 1$
- Tutti gli stati successori non migliorano la situazione, *provate!* (minimo locale)
- Per le 8 regine Hill-climbing si blocca l'86% delle volte
- Ma in media solo 4 passi per la soluzione e 3 quando si blocca
- Su $8^8 = 16.8$ milioni di stati



Esempio positivo: successo in tre mosse



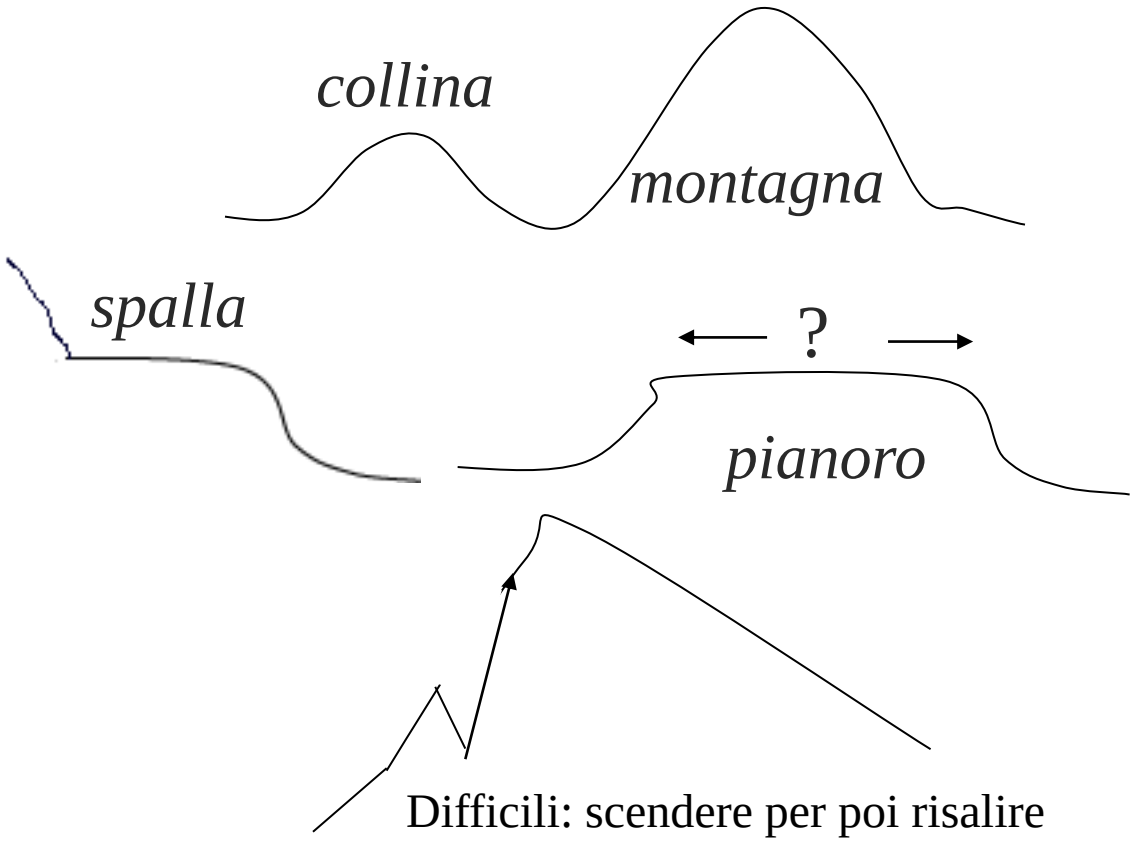
h qui è l'euristica di costo della funzione obiettivo (da minimizzare)

Problemi con Hill-climbing

Se la f. è da ottimizzare i picchi sono massimi locali o soluzioni ottimali

- Massimi locali
- Pianori o spalle

  Plateau
- Crinali (o creste)

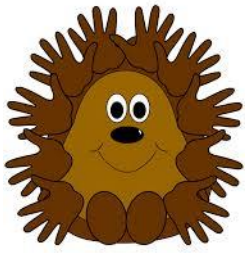


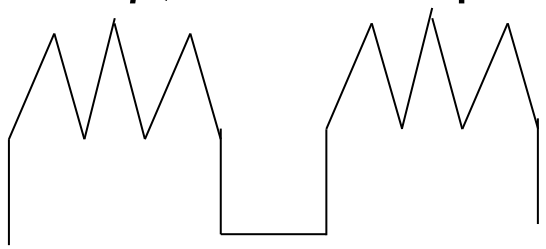
Miglioramenti

1. Consentire (un numero limitato di) mosse *laterali* (ossia ci si ferma per < nell'alg. invece che per ≤ → continua anche a parità di h)
   - L'algoritmo sulle 8 regine ha successo nel 94%, ma impiega in media 21 passi
2. Hill-climbing stocastico: si sceglie a caso tra le mosse in salita (magari tenendo conto della pendenza)
   - Converge più lentamente ma a volte trova soluzioni migliori
3. Hill-climbing con prima scelta
   - Può generare le mosse a caso, uno alla volta, fino a trovarne una migliore dello stato corrente (si prende solo il primo che migliora)
   - Come la stocastica ma utile quando i successori sono molti (e.g. migliaia o ben oltre), evitando una scelta tra tutti.

Miglioramenti (cont.)

4. Hill-Climbing con *riavvio casuale* (*random restart*): ripartire da un punto scelto a caso
   - Se la probabilità di successo è *p* saranno necessarie in media 1/p ripartenze per trovare la soluzione (es. 8 regine, *p*=0.14 → *7* ripartenze [1/p] per avere 6 fail e un successo)
   - Hill-climbing con random-restart è tendenzialmente completo (e.g. insistendo si generano tutte!)
   - Per le regine: caso con 3 milioni di regine in meno di un minuto!
   - Se funziona o no dipende molto dalla forma del panorama degli stati (molti min loc. abbassano *p*, si blocca spesso)

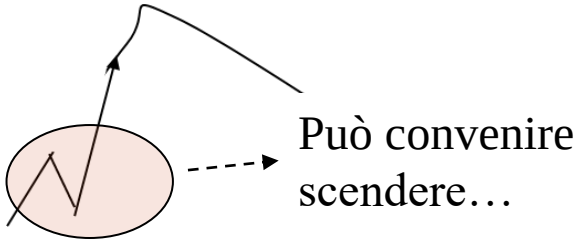
Il porcospino 



Tempra simulata

- L' algoritmo di tempra simulata (*Simulated annealing*) [Kirkpatrick, Gelatt, Vecchi 1983] combina hill-climbing con una scelta stocastica (ma non del tutto casuale, perché poco efficiente…)
- Analogia con il processo di tempra dei metalli in metallurgia
- I metalli vengono portati a temperature molto elevate (alta energia/stocasticità iniziale) e raffreddati gradualmente consentendo di cristallizzare in uno stato a (più) bassa energia.
- (esempio di cross-fertilization tra aree scientifche diverse)

Tempra simulate (per ascesa)



- Ad ogni passo si sceglie un successore *n'* ***a caso***:
  - se migliora lo stato corrente viene espanso
  - se no (caso in cui $\Delta E = f(n') - f(n) \leq 0$) quel nodo viene scelto con probabilità $p = e^{\Delta E/T}$ $[0 \leq p \leq 1]$

    [Si genera un numero casuale tra 0 e 1: se questo è < *p* il successore viene scelto, altrimenti no]
  - Ossia: *p* è inversamente proporzionale al peggioramento
    - Infatti se la mossa peggiora molto, ΔE alto neg., la *p* si abbassa
  - T (temperatura) decresce col progredire dell'algoritmo (quindi anche *p*) secondo un piano definito
    - Col progredire rende improbabili le mosse peggiorative
  - Esercizio: riformulare per cercare un minimo

Tempra simulata: analisi

- La probabilità *p* di una mossa in discesa diminuisce col tempo e l'algoritmo si comporta sempre di più come Hill Climbing.
- Se T viene decrementato abbastanza lentamente con prob. tendentente ad 1 si raggiunge la soluzione ottimale.
- Analogia col processo di tempra dei metalli
  - T corrisponde alla temperatura
  - ΔE alla variazione di energia

Esercizio: pensare a valori estremi di T: che tipi di hill climbing si ottengono?

Tempra simulata: parametri

- Valore iniziale e decremento di T sono parametri
- Valori per T determinati sperimentalmente: il valore iniziale di T è tale che per valori medi di ΔE, $p = e^{\Delta E/T}$ sia all'incirca 0.5

- *NOTA: Presente nell'insieme proposto in Python («simulated_annealing»): enjoy!!!*

Ricerca *local beam*

- La versione locale della *beam search*
- Si tengono in memoria *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 tra questi
  - Note:
    - diverso da K restart (che riparte da zero)
    - diverso da beam search: perche?
    - Esercizio: K =1 o illimitato a cosa portano?

Beam search stocastica

- Si introduce un elemento di casualità … come in un processo di selezione naturale (diversificare la nuova generazione)
- Nella variante stocastica della *local beam*, si scelgono *k* successori, ma con probabilità maggiore per i migliori
- La terminologia:
  - *organismo* [stato]
  - *progenie* [successori]
  - *fitness* [il valore della f], idoneità

Algoritmi genetici/evolutivi: l'idea

- Sono varianti della *beam search stocastica* in cui gli stati successori sono ottenuti combinando *due stati genitore (anziché per evoluzione)*
- La terminologia:
  - popolazione di individui [stati]
  - fitness
  - accoppiamenti + mutazione genetica
  - generazioni

Algoritmi genetici/evolutivi: funzionamento

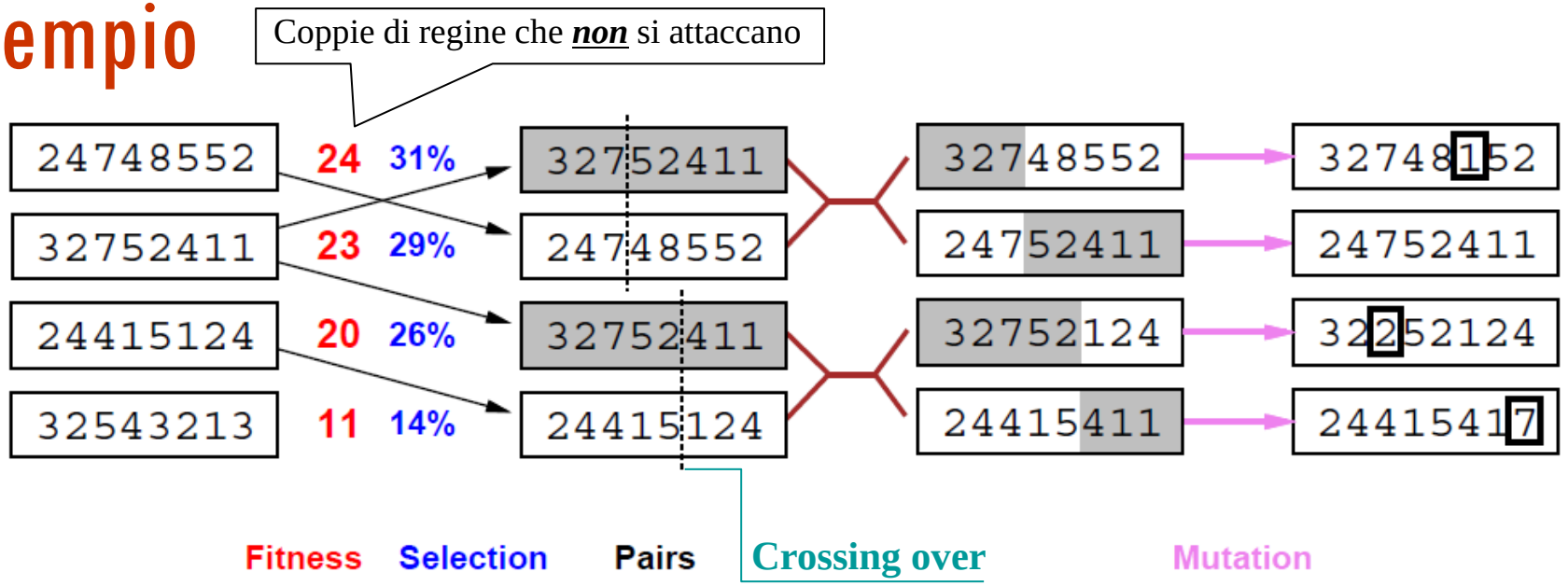
- Popolazione iniziale:
  - *k* stati/individui generati casualmente
  - ogni individuo è rappresentato come una stringa

    Esempio: posizione nelle colonne ("24748552") stato delle 8 regine o con 24 bit*
- Gli individui sono valutati da una funzione di *fitness*
  - Esempio: n. di coppie di regine che ***non*** si attaccano

Algoritmi genetici (cont.)

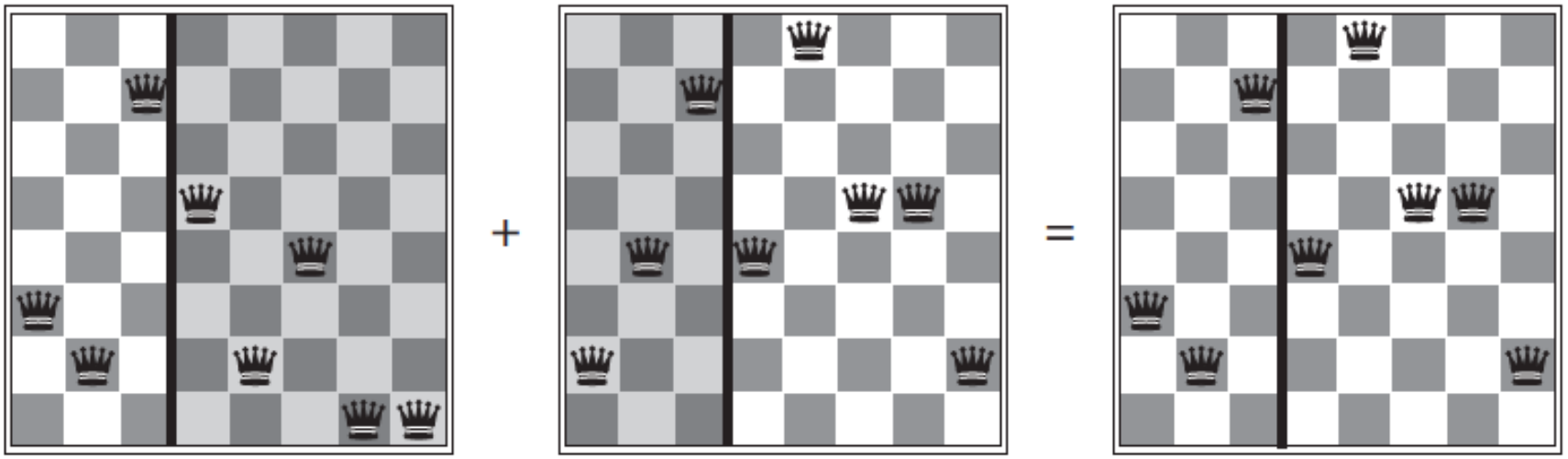
- Si selezionano gli individui per gli "accoppiamenti" con una probabilità proporzionale alla fitness
- Le coppie danno vita alla generazione successiva
  - Combinando materiale genetico (*crossover*)
  - Con un meccanismo aggiuntivo di mutazione genetica (casuale)
- La popolazione ottenuta dovrebbe essere migliore
- La cosa si ripete fino ad ottenere stati abbastanza buoni (stati obiettivo) o finche non miglioriamo più

Esempio



- Per ogni coppia (scelta con probabilità proporzionale alla fitness) viene scelto un punto di *crossing over* e vengono generati due figli scambiandosi pezzi (del DNA)
- Viene infine effettuata una *mutazione* casuale che dà luogo alla prossima generazione
- La *fitness* progressivamente tenderà a favorire generazioni migliori
- Emula i meccanismi genetici ma anche l'evoluzione della specie

Nascita di un figlio



- Le parti chiare sono passate al figlio
- Le parti grigie si perdono
- Se i genitori sono molto diversi anche i nuovi stati sono diversi
- All'inizio spostamenti maggiori che poi si raffinano

Algoritmi genetici

- Suggestivi (area del *Natural computing: e.g. swarm, …)*
- Usati in molti problemi reali e.g. configurazione di circuiti e scheduling di lavori
- Vantaggi: combinano
  - Tendenza a salire della beam search stocastica
  - Interscambio info (indirettamente) tra thread paralleli di ricerca (blocchi utili che si combinano)
- Funziona meglio se il problema (soluzioni) ha componenti significative rappresentate in sottostringhe
- Punto critico: la rappresentazione del problema in stringhe

Spazi continui

- Molti casi reali hanno spazi di ricerca continua e.g. fondamentale per Machine Learning !!!
- Stato descritto da variabili continue, x_1, … x_n , vettore **x**

- Prendiamo ad esempio movimenti in spazio 3D, con posizione data da **x**=(x_1,x_2, x_3)
- Apparentemente ostico, fattori di ramificazione *infiniti* con gli approcci precedenti
- In realtà molti strumenti matematici per spazi continui, che portano ad approcci anche molto efficienti….

Spazi continui- gradient

- Se la f è continua e differenziabile, e.g. quadratica rispetto a **x** (vettore)
- Il minimo o massimo si può cercare utilizzando il **gradiente**, che restituisce la direzione di massima pendenza nel punto
- Data *f* obiettivo su 3D

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3} \right)$$

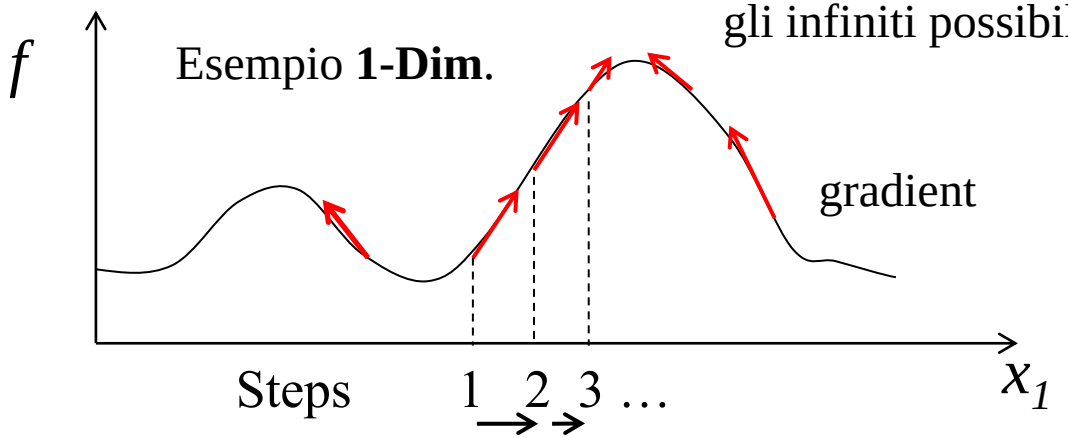
- Hill climbing iterativo: $\mathbf{x}_{new} = \mathbf{x} + \eta \nabla f(\mathbf{x})$

Uso + per salire (maximization)
Uso – per scendere (minimization)

Eta: step size

Quantifica direzione e spostamento, senza cercarlo tra gli infiniti possibili successori!

Nota generale: non sempre è necessario il min/max assoluto: vedremo nel ML

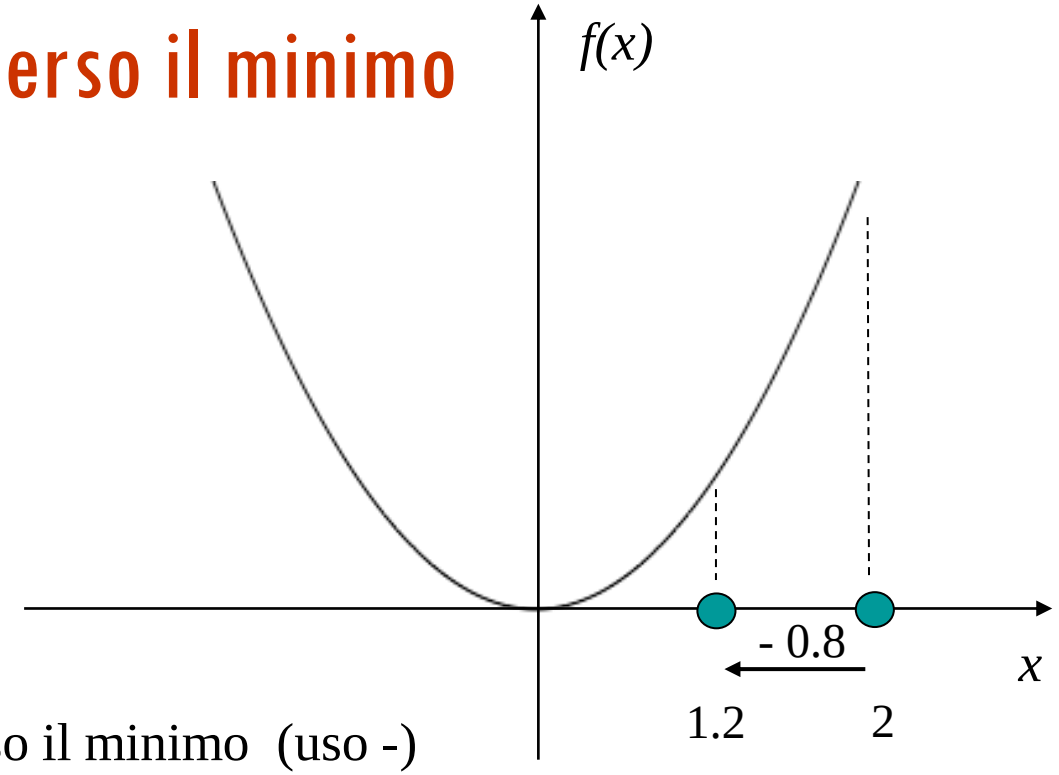


Spostamenti guidati dal gradiente df/dx_1

Esempio: discesa verso il minimo

$f(x)=x^2$

$f'(x)=2x$



Discesa di gradiente verso il minimo (uso -)

$$\mathbf{x}_{new} = \mathbf{x} - \eta \nabla f(\mathbf{x}) \xrightarrow{\text{1-D}} x_{new} = x - \eta f'(x)$$

Mettiamoci in x=2, la derivata vale 2x2=4, mi devo spostare di $-\eta f'(x) = -\eta 4$, quindi ad esempio (η=0.2) ottengo $-\eta f'(x) = -0.2 \times 4 = -0.8$ andando a sinistra al punto $x_{new} = 2 - 0.8 = 1.2$ avvicinandomi quindi al minimo

Part 2
Oltre la ricerca classica

CENNI

Assunzioni sull'ambiente da riconsiderare

- Azioni non deterministiche e ambiente parzialmente osservabile
  - Piani condizionali, ricerca AND-OR, stati credenza
- Ambienti sconosciuti e problemi di esplorazione (percezioni forniscono nuove informazioni dopo l'azione)
  - Ricerca *online*

Ambienti più realistici

- Gli agenti *risolutori di problemi* "classici" assumono:
  - Ambienti completamente osservabili
  - Azioni/ambienti deterministici
  - Il piano generato è una sequenza di azioni che può essere generato *offline* e eseguito senza imprevisti
  - Le percezioni non servono se non nello stato iniziale

Soluzioni più complesse

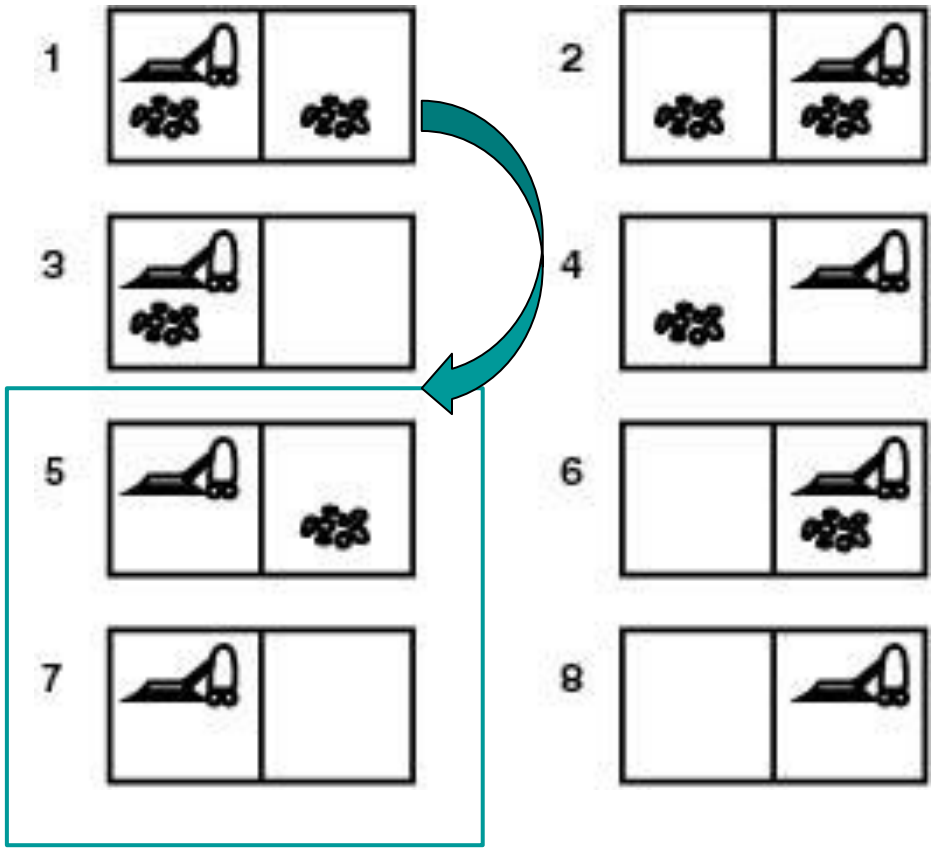
- In un ambiente parzialmente osservabile e non deterministico le percezioni sono importanti
  - restringono gli stati possibili
  - informano sull'effetto dell'azione
- Più che un *piano* l'agente può elaborare una "*strategia*", che tiene conto delle diverse eventualità: un piano con contigenza
- Esempio: l'aspirapolvere con assunzioni diverse
  - Vediamo prima il non determinismo.

Azioni non deterministiche

L'aspirapolvere imprevedibile: ci sono più stati possibili risulato dell'azione

- Comportamento:
  - *Se aspira in una stanza sporca, la pulisce … ma talvolta pulisce anche una stanza adiacente*
  - *Se aspira in una stanza pulita, a volte rilascia sporco*
- Variazioni necessarie al modello
  - Il modello di transizione restituisce un **insieme di stati:** l'agente non sa in quale si troverà
  - Il piano di contingenza sarà un piano condizionale e magari con cicli

Esempio



- Esempio

Risultati(Aspira, 1) = {5, 7}

- Piano possibile

[Aspira,

if stato=5

then [Destra, Aspira]

else [ ]

]

- Soluzione: da sequenza di azioni a piano (albero)

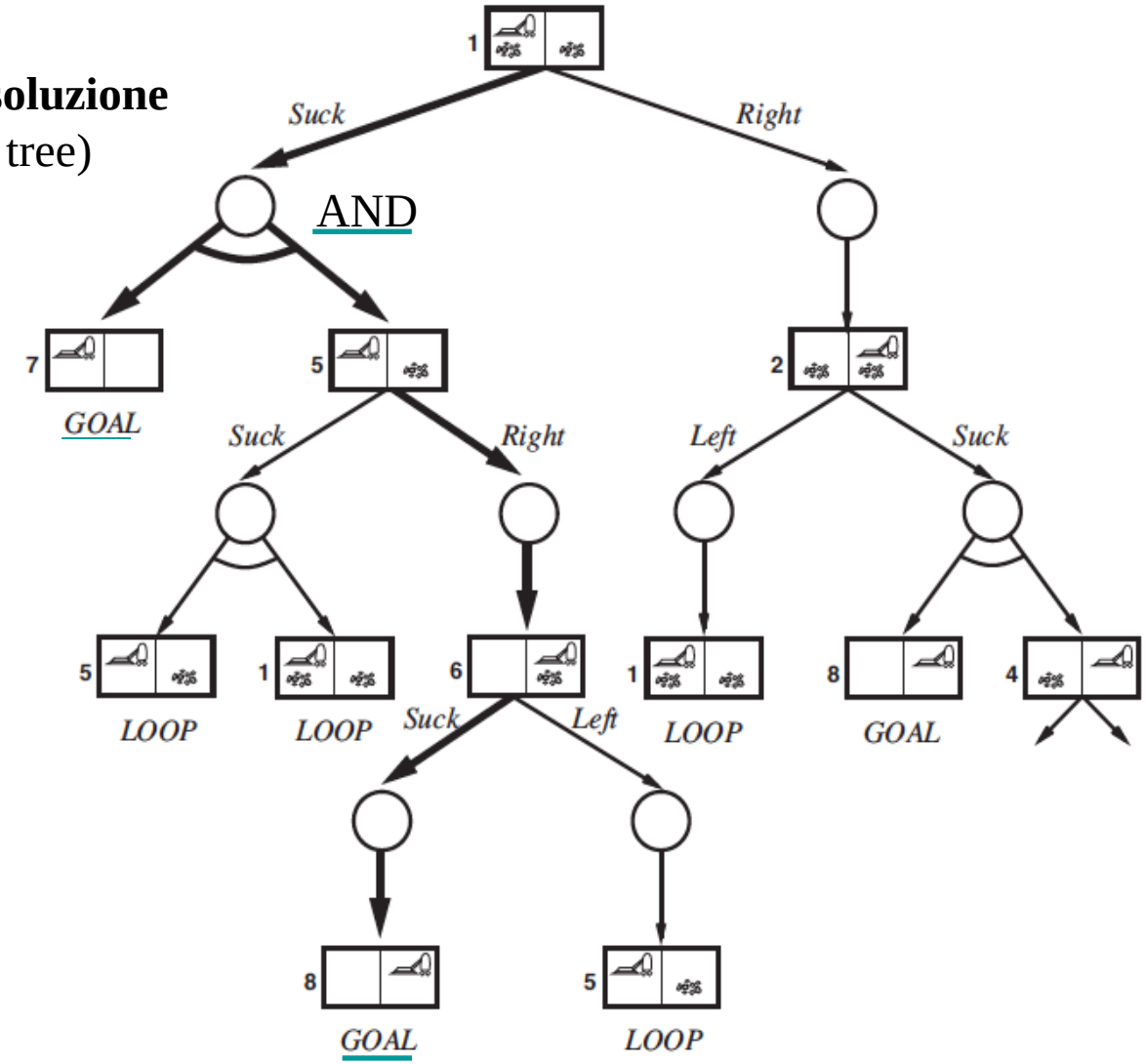
Come si pianifica: *alberi di ricerca AND-OR*

- Nodi OR le scelte dell'agente [1 sola azione]
- Nodi AND le diverse contingenze (le scelte dell'ambiente, piu stati possibili), da considerare tutte
- Una soluzione a un problema di ricerca AND-OR è un albero che:
  - ha un nodo obiettivo in ogni foglia
  - specifica un'unica azione nei nodi OR
  - include tutti gli archi uscenti da nodi **AND** (tutte le contingenze)

Esempio di ricerca AND-OR

Bold= Piano soluzione
(sub tree)

Archi in bold



Piano: [*Aspira*, ***if*** Stato=5 ***then*** [*Destra*, *Aspira*] ***else*** [ ]]

STOP per IIA 6 crediti

Fin qui: AIMA ed. III fino a cap 4.3 incluso

Il resto di questa presentazione vi resta disponibile per consultazione

Algoritmo ricerca grafi AND-OR

function Ricerca-Grafo-AND-OR (*problema)*

returns un piano condizionale oppure *fallimento*

Ricerca-OR(*problema.StatoIniziale, problema,* [ ])

function Ricerca-OR(*stato, problema, cammino*) // nodi OR

returns un piano condizionale oppure *fallimento*

If *problema.*TestObiettivo(*stato*) **then return** [ ] // piano vuoto

If *stato* è su *cammino* **then return** *fallimento* // spezza i cicli

for each *azione* **in** *problema.Azione(stato)* **do**

piano ← Ricerca-AND (*Risultati*(*stato, azione*), *problema,* [*stato*|*cammino*])

If *piano* ≠ *fallimento* **then return** [*azione* | *piano*]

return *fallimento*

Algoritmo ricerca grafi AND-OR

function Ricerca-AND(*stati, problema, cammino*) // nodi AND

  returns un piano condizionale oppure *fallimento*

for each s_i **in** *stati* **do**

  $piano_i$ ← Ricerca-OR(s_i, *problema, cammino*)

  If $piano_i$ = *fallimento* **then return** *fallimento*

return

  [**if** s_1 **then** $piano_1$ **else**

    if s_2 **then** $piano_2$ **else**

      …

        if s_{n-1} **then** $piano_{n-1}$ **else** $piano_n$]

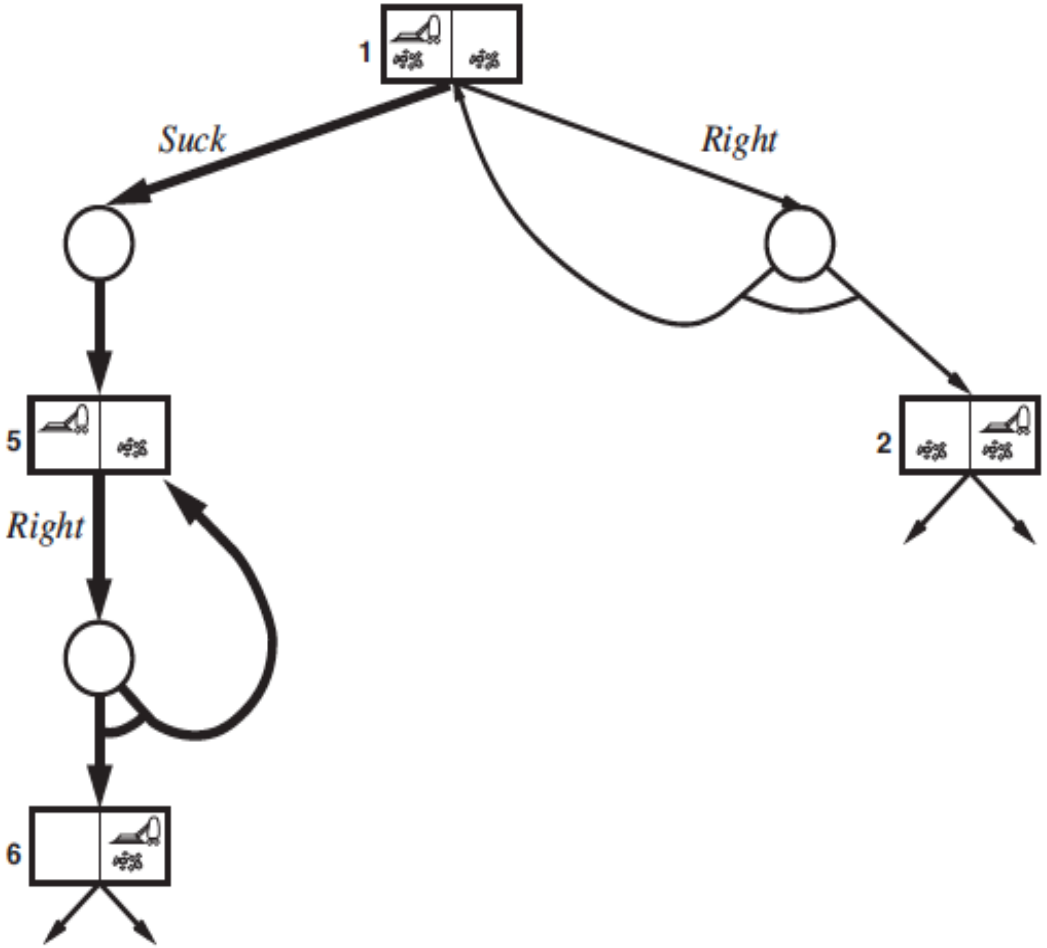
Ritorna un piano condizionale
Con azioni dei nodi OR

Ancora azioni non deterministiche

L'aspirapolvere slittante

- Comportamento:
  - *Quando si sposta può scivolare e rimanere nella stessa stanza*
  - Es. Risultati(Destra, 1) = {1, 2}
- Variazioni necessarie
  - Continuare a provare … [finche' riesce ad andare a destra]
  - Il piano di contingenza potrà avere dei cicli

Aspirapolvere slittante: soluzione



Piano: [*Aspira*, L_1: *Destra*, **if** Stato=5 **then** L_1 **else** *Aspira*]

Osservazione

- Bisogna distinguere tra:
  1. Osservabile e non deterministico (es. aspirapolvere slittante)
  2. Non (o parzialmente) osservabile e deterministico (es. non so se la chiave aprirà la porta)
- In questo secondo caso, se la chiave è sbagliata, si può provare all' infinito ma niente cambierà!

Ricerca con osservazioni parziali

- Le percezioni non sono sufficienti a determinare lo stato esatto, anche se l'ambiente è deterministico.
- Stato credenza: un insieme di *stati possibili* in base alle conoscenze dell'agente
- Problemi senza sensori (*sensorless* o conformanti)
- Si possono trovare soluzioni anche senza affidarsi ai sensori utilizzando stati-credenza

Ambiente non osservabile:

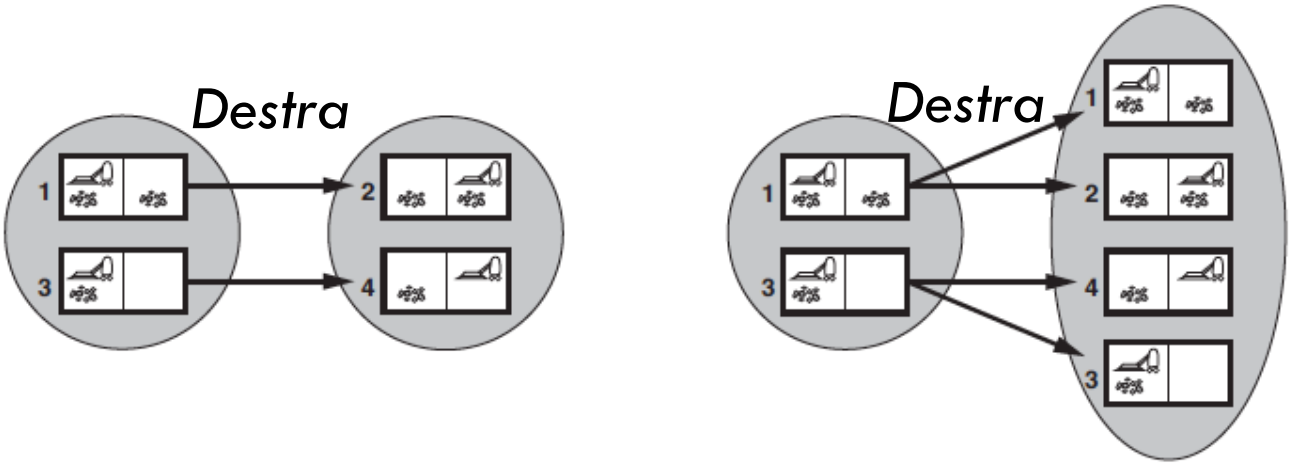
Aspirapolvere senza sensori

- L'aspirapolvere:
  - non percepisce la sua locazione, né se la stanza è sporca o pulita
  - conosce la geografia del suo mondo e l'effetto delle azioni
- Inizialmente tutti gli stati sono possibili
  - Stato iniziale = {1, 2, 3, 4, 5, 6, 7, 8}
- Le azioni riducono gli stati credenza
- Nota: nello spazio degli stati credenza l'ambiente è osservabile (l'agente conosce le sue credenze)

Formulazione di problemi con stati-credenza

- Se N numero stati , 2^N sono i possibili stati credenza
- Stato-credenza iniziale $SC_0 \subseteq$ insieme di tutti gli N stati
- Azioni(b) = unione delle azioni *lecite* negli stati in b (ma se azioni illecite in uno stato hanno effetti dannosi meglio intersezione)
- Modello di transizione: gli stati risultanti sono quelli ottenibili applicando le azioni a uno stato qualsiasi (l' unione degli stati ottenibili dai diversi stati possibili con le azioni eseguibili)

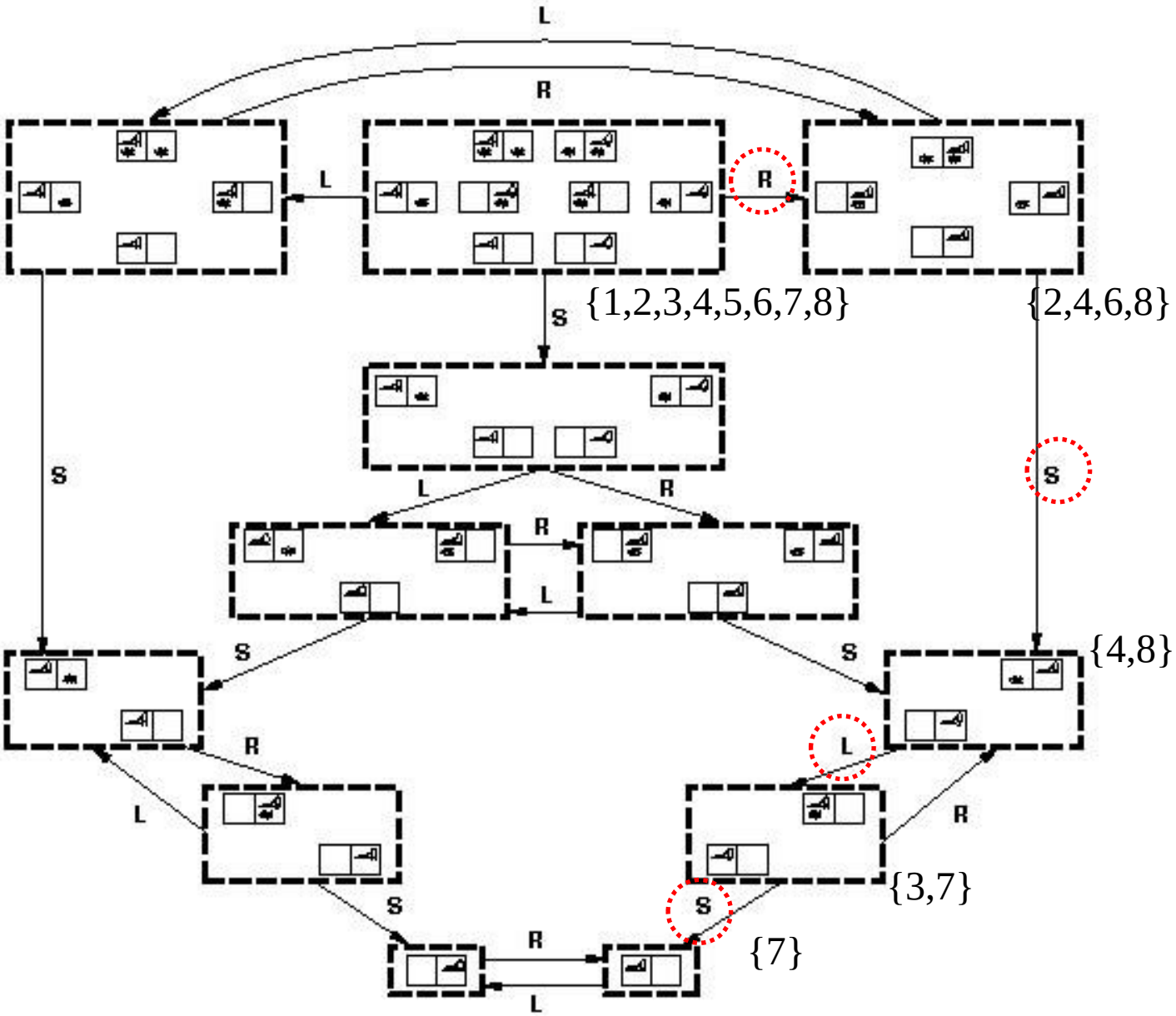
Problemi con stati-credenza (cnt.)



Senza sensori deterministico *Senza sensori e slittante (non det.)*

- Test obiettivo: tutti gli stati nello stato credenza devono soddisfarlo
- Costo di cammino: il costo di eseguire un'azione potrebbe dipendere dallo stato, ma assumiamo di no

Il mondo dell'aspirapolvere senza sensori (determ.)



Ricerca: ottimizzazioni

- Si può effettuare un Ricerca-Grafo e controllare, generando *s*, se si è già incontrato uno stato credenza *s'*=*s* e trascurare *s*
- Si può anche "potare" in modo più efficace in base al fatto che:
  - Se *s'* $\subseteq$ *s*, allora ogni sequenza di azioni che è una soluzione per s lo è anche per s'
    - Se *s'* $\subseteq$ *s* (s' già incontrato) si può trascurare *s*
    - Se s $\subseteq$ s' e da s' si è trovata una soluzione si può trascurare *s*

Soluzione incrementale

- Dovendo trovare una soluzione per {1, 2, 3 …} si cerca una soluzione per stato 1 e poi si controlla che funzioni per 2 e i successivi; se no se ne cerca un'altra per 1 …
- Scopre presto i fallimenti ma cerca un'unica soluzione che va bene per tutti gli stati
- Non è una strategia completa ma è sicuramente più efficiente

Ricerca della soluzione

- Gli stati credenza possibili sono 2^8=256 ma solo 12 sono raggiungibili
- In generale lo spazio di ogni stato può essere molto più grande con gli "stati credenza"
- La rappresentazione atomica obbliga a elencare tutti gli stati. Non è molto "compatta". Non così con una rappresentazione più strutturata (lo vedremo)

Ricerca con osservazioni

- Ambiente parzialmente osservabile
- Esempio: *l'aspirapolvere con sensori locali che percepisce la sua posizione e lo sporco nella stanza in cui si trova* (ma non nelle altre stanze)
- Le percezioni diventano importanti
  - Assumiamo Percezioni(s)

Ricerca con osservazioni parziali

- Le percezioni assumono un ruolo
  - Percezioni(s) = *null* in problemi *sensorless*
  - Percezioni(s) = s, ambienti osservabili
  - Percezioni(s) = percezioni [possibili] nello stato *s*
- Le percezioni restringono l'insieme di stati possibili
  - Esempio: [A, *Sporco*] percezione stato iniziale

    Stato iniziale = {1, 3}

Il modello di transizione si complica

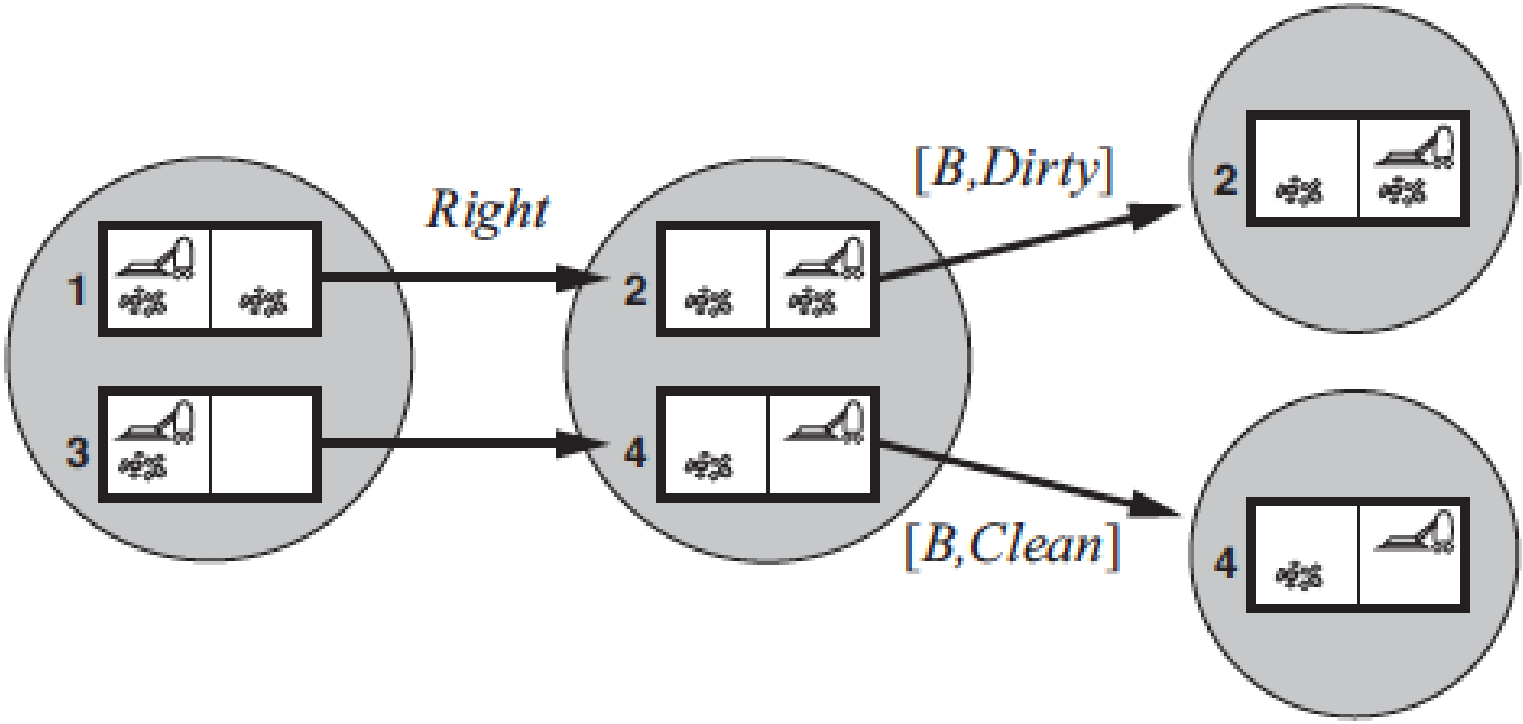
La transizione avviene in tre fasi:

1. Predizione dello stato credenza per effetto delle azioni: *Predizione*(b, a)=b'
2. Predizione dell'osservazione: *Percezioni-possibili*(b')
3. Calcolo *aggiornamento* (insieme di stati credenza compatibili con lo stato credenza predetto e le possibili osservazioni):

   b'' = *Aggiorna*(Predizione(b, a), o)

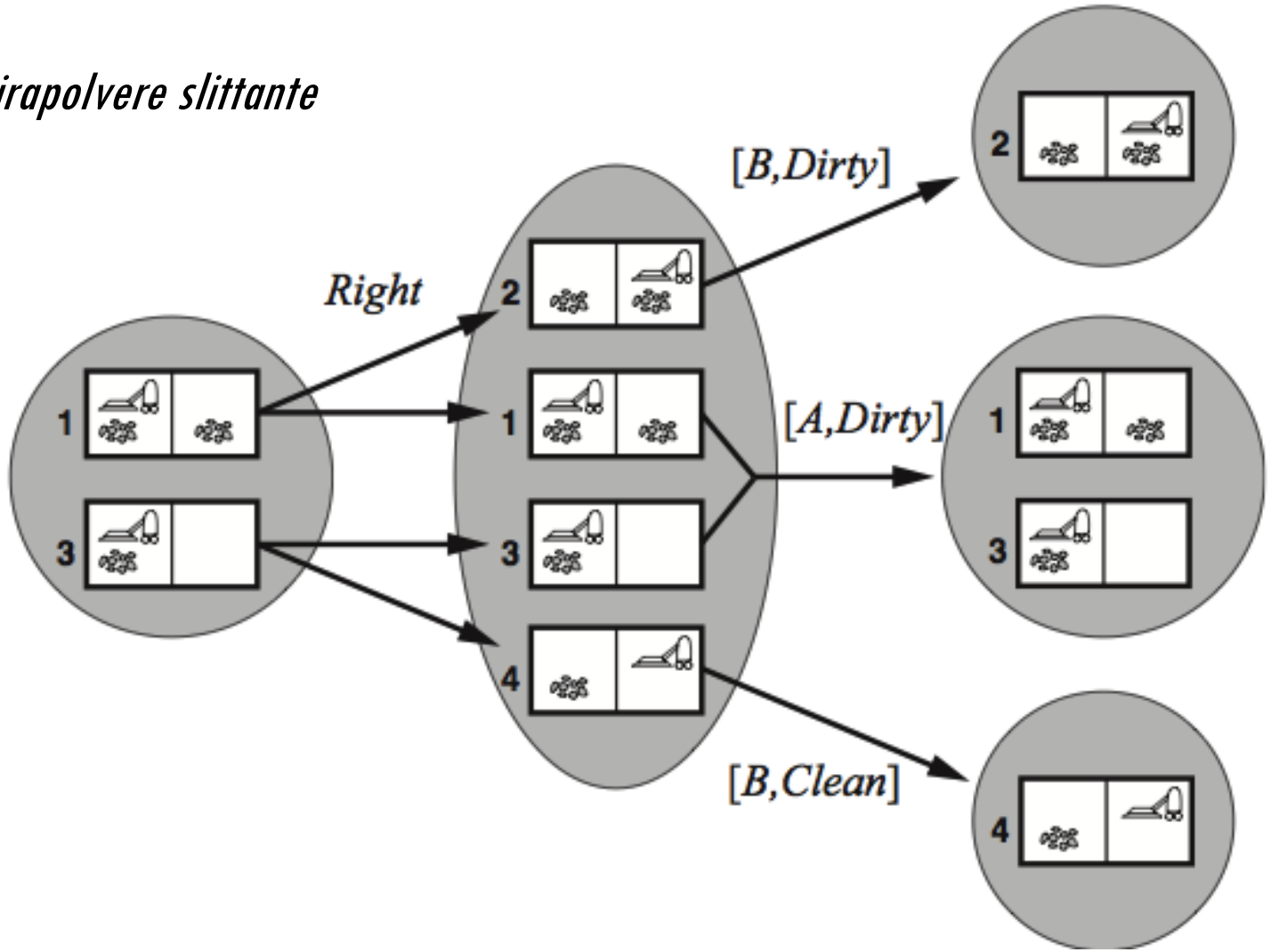
   per ogni possibile osservazione o

Transizione con azioni deterministiche



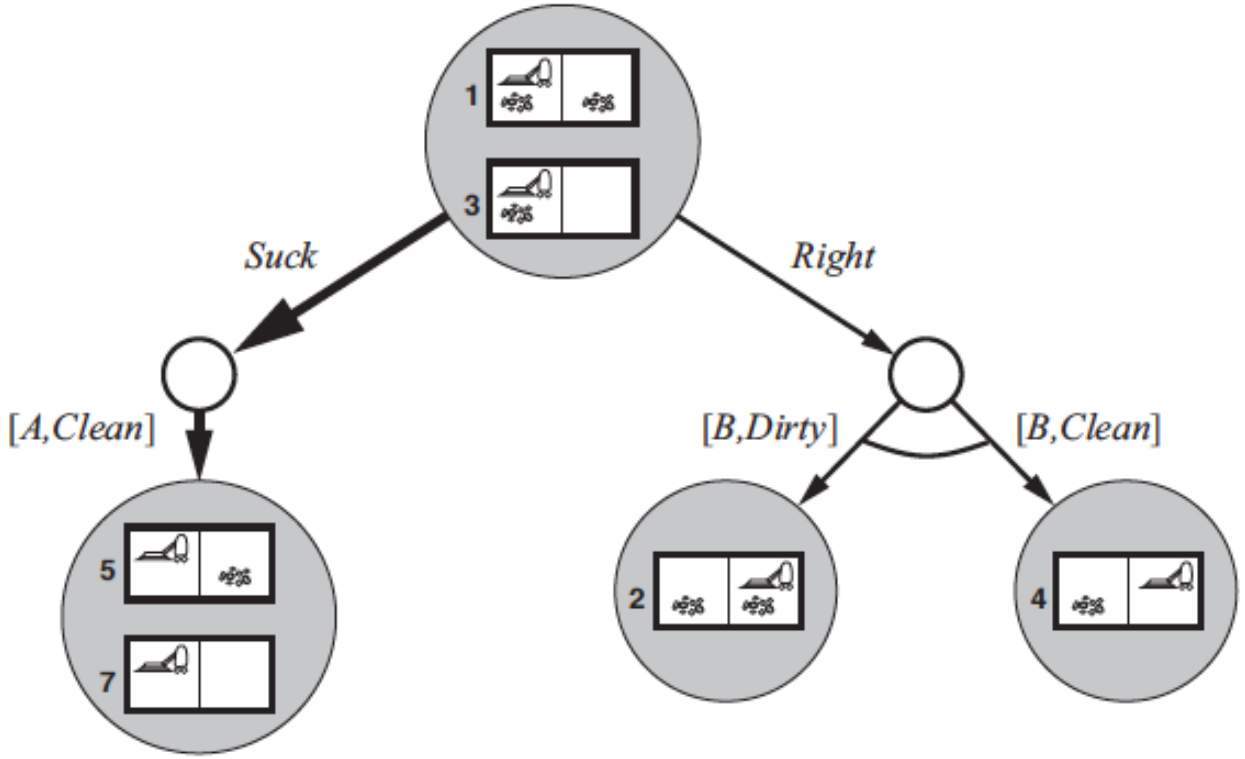
Transizione con azioni non deterministiche

Aspirapolvere slittante



Aspirapolvere con sensori locali

Per pianificare ci servono grafi AND-OR su stati credenza



[*Aspira, Destra,* **if** statoCredenza = {6} **then** *Aspira* **else** []]

Ricerca *online*

- Ricerca *offline* e ricerca *online*
- L'agente alterna pianificazione e azione

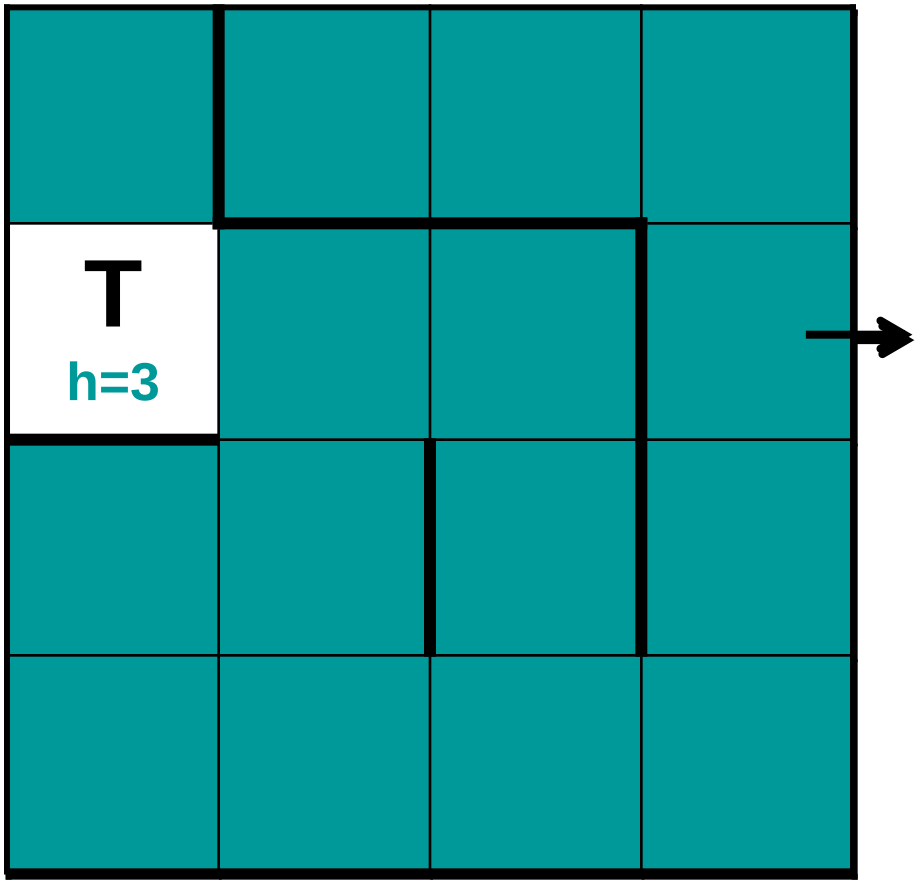
1. Utile in ambienti dinamici o semidinamici
   - Non c'è troppo tempo per pianificare
2. Utile in ambienti non deterministici
   1. Pianificare vs agire
3. Necessaria per ambienti ignoti tipici dei problemi di esplorazione

Problemi di esplorazione

- I problemi di esplorazione sono casi estremi di problemi con contingenza in cui l'agente deve anche pianificare azioni esplorative
- Assunzioni per un problema di esplorazione:
  - Solo lo stato corrente è osservabile, l'ambiente è ignoto
  - Non si conosce l'effetto delle azioni e il loro costo
  - Gli stati futuri e le azioni che saranno possibili non sono conosciute a priori
  - Si devono compiere azioni esplorative come parte della risoluzione del problema
- Il labirinto come esempio tipico

Esempio: Teseo con mappa e senza

- Con mappa
  - applicabili tutti gli algoritmi di pianificazione visti
- Senza mappa
  - l'agente non può pianificare può solo esplorare nel modo più razionale possibile
  - Ricerca online



Assunzioni

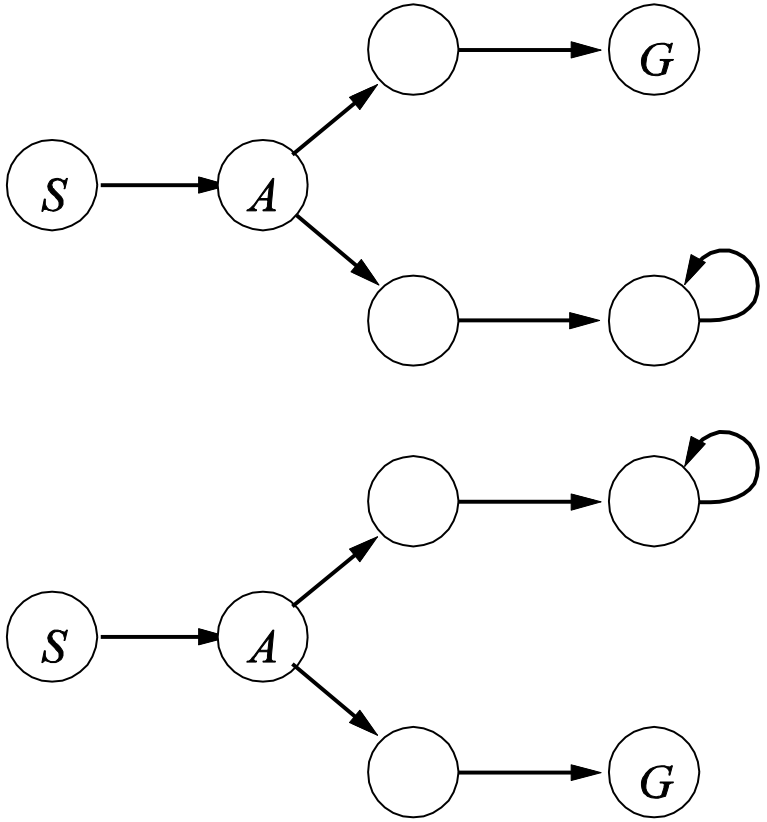
- Cosa conosce un agente *online* in *s* …
  - Le azioni legali nello stato attuale *s*: Azioni (s)
  - Risultato(*s*, *a*), ma dopo aver eseguito *a*
  - Il costo della mossa c(*s*, *a*, *s*'), solo dopo aver eseguito *a*
  - Goal-test(*s*)
  - Puo esserci: la stima della distanza: dal goal: *h*(*s*)

Costo soluzione

- Il costo del cammino è quello effettivamente percorso
- Il rapporti tra questo costo e quello ideale (conoscendo l'ambiente) è chiamato rapporto di competitività
- Tale rapporto può essere infinito
- Le prestazioni sono in funzione dello spazio degli stati

Assunzione ulteriore

- Ambienti esplorabili in maniera sicura
  - non esistono azioni irreversibili
  - lo stato obiettivo può sempre essere raggiunto
  - diversamente non si può garantire una soluzione

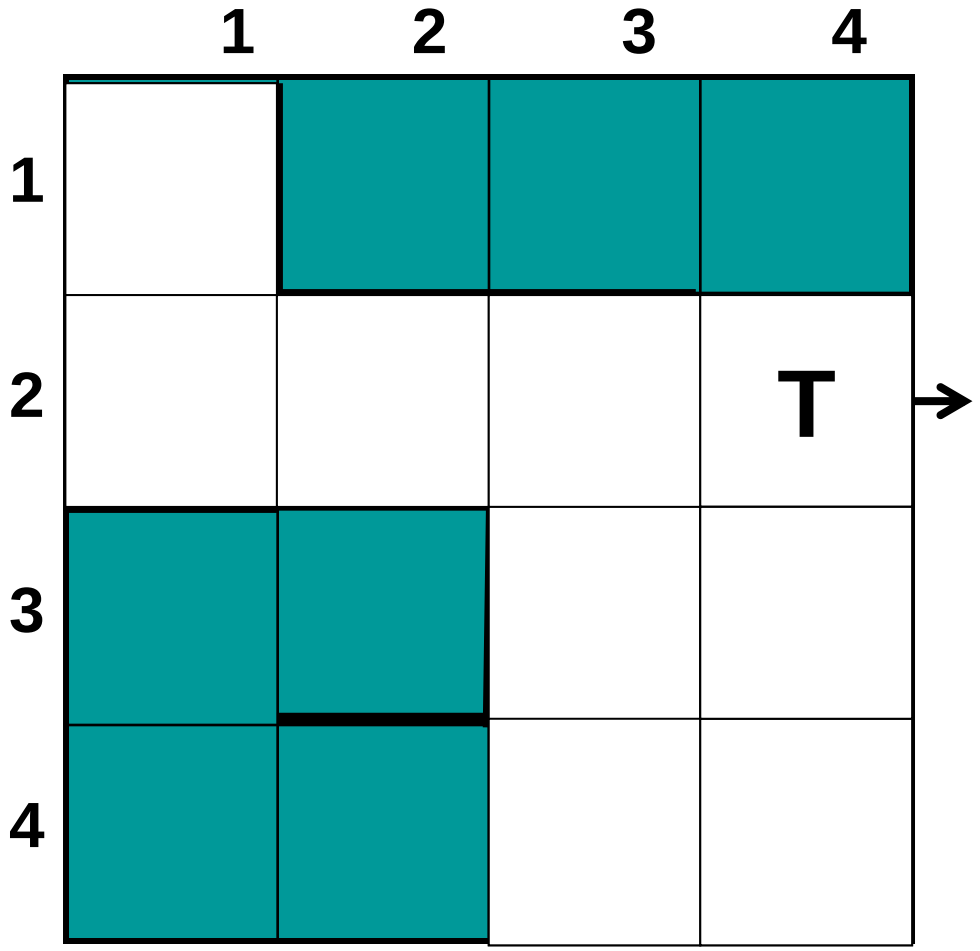


Ricerca in profondità *online*

- Gli agenti *online* ad ogni passo decidono l'azione da fare (non il piano) e la eseguono.
- Procedono solo "localmente", ma anche otrnando indietro e.g.:
- Ricerca in profondità *online*
  - Esplorazione sistematica delle alternative
    - *nonProvate*[s] mosse ancora da esplorare in s
  - È necessario ricordarsi ciò che si è scoperto
    - *Risultato*[*s*, *a*] = *s'*
  - Il backtracking significa tornare sui propri passi
    - *backtrack*[s] stati a cui si può tornare

Esempio

- Sceglie il primo tra (1,1) e (2,2)
- In (1, 1) ha solo l'azione per tornare indietro (POP bactrack[s])
- …
- Nella peggiore delle ipotesi esplora ogni casella due volte



Algoritmo in profondità *online*

function Agente-Online-DFS(*s*) returns un'azione

  static: *Risultato*, *nonProvate*, *backtrack*,

    s- (*stato precedente*), a- (*ultima azione*)

if Goal-Test(*s*) **then return** stop

if s è un nuovo stato **then** *nonProvate*[*s*] ← Azioni(*s*)

if s- *non è null* **then** risultato[s-, a-] ← s; *backtrack*[*s*] ← s-;

if *nonProvate*[*s*] *vuoto* **then**

  if *backtrack*[s] *vuoto* **then return** stop

  else a ← *azione per tornare in* POP(*backtrack*[*s*])

else a ← POP(*nonProvate*[*s*])

s- ← s; return a

Ricerca euristica *online*

- Nella ricerca online si conosce il valore della funzione euristica una volta esplorato lo stato.
- Un algoritmo di tipo Best First non funzionerebbe.
- Serve un metodo locale
- *Hill-climbing* con *random-restart* non praticabile
- Come sfuggire a minimi locali?

Due soluzioni

1. Random-walk
   - si fanno mosse casuali (privilegiando le nuove)
   - in principio completa ma costosa!
2. Ricerca locale con A* (LRTA*):
   - Learning Real Time A*, A* con memoria (apprendimento) in tempo reale
   - esplorando si aggiustano i valori dell'euristica per renderli più realistici
   - In questo modo riesce a superare i minimi locali

Idea dell'algoritmo LRTA*

- H(s): migliore stima trovata fin qui
- Si valutano i successori:

  Costo-LRTA*(s, a, s', H) =

  h(s) *se s' indefinito* (non esplorato)
  H(s') + costo(s,a,s') *altrimenti*
- Ci si sposta sul successore di Costo-LRTA* minore
- Si aggiorna la H dello stato da cui si proviene

LRTA*

function Agente-LRTA*(s) **returns** un'azione

static: risultato, H, s-, a-

if Goal-Test(s) **then return** stop

if s *nuovo (non in H)* **then** H[s] ← h[s]

1. **if** s- non è null *//si aggiusta il costo H del predecessore*

risultato[s-, a-] ← s

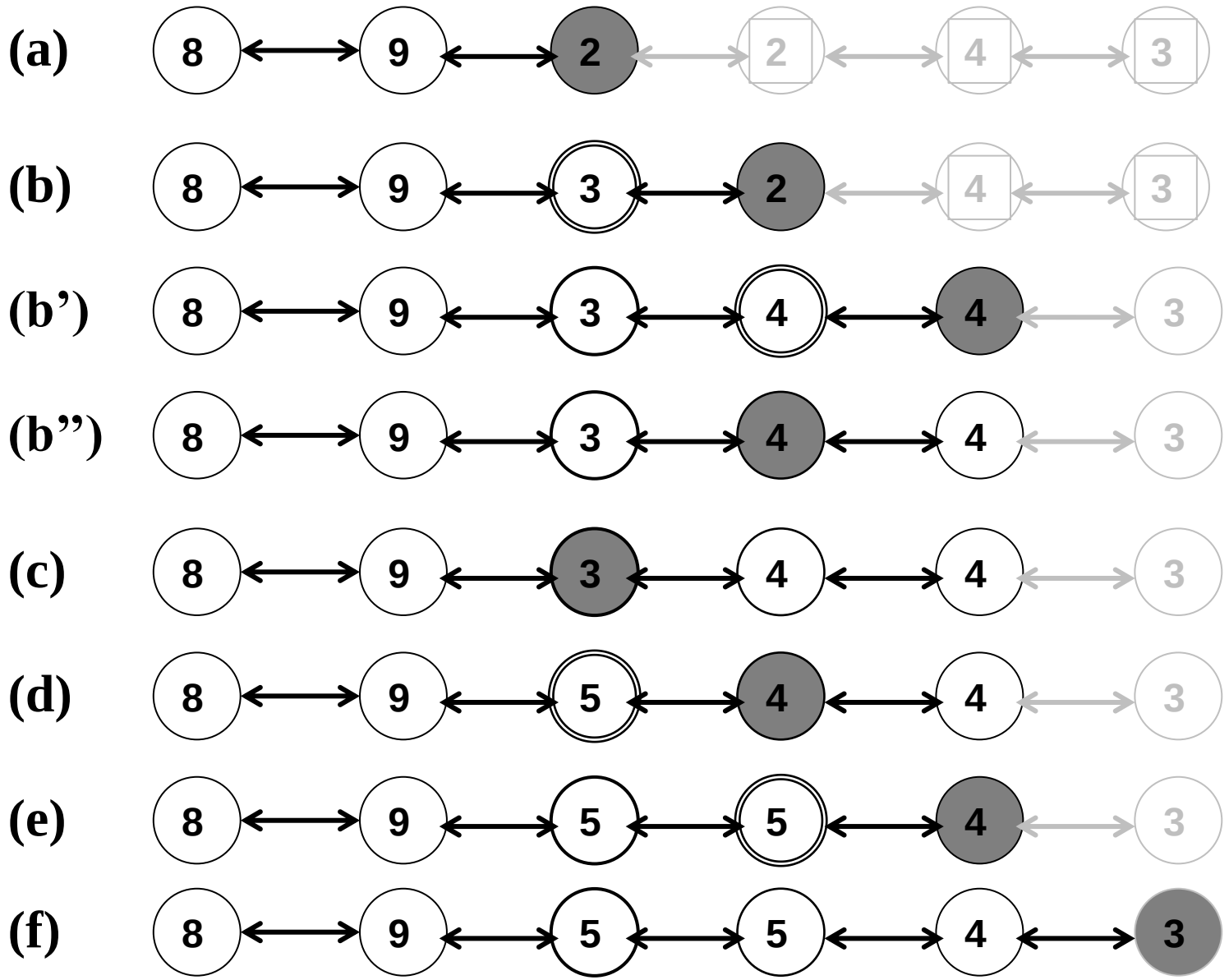
H[s-] ← *min* Costo-LRTA*(s-, b, risultato[s-, b], H)

2. a ← *un'azione b tale che* minimizza b ∈ Azioni(s-)

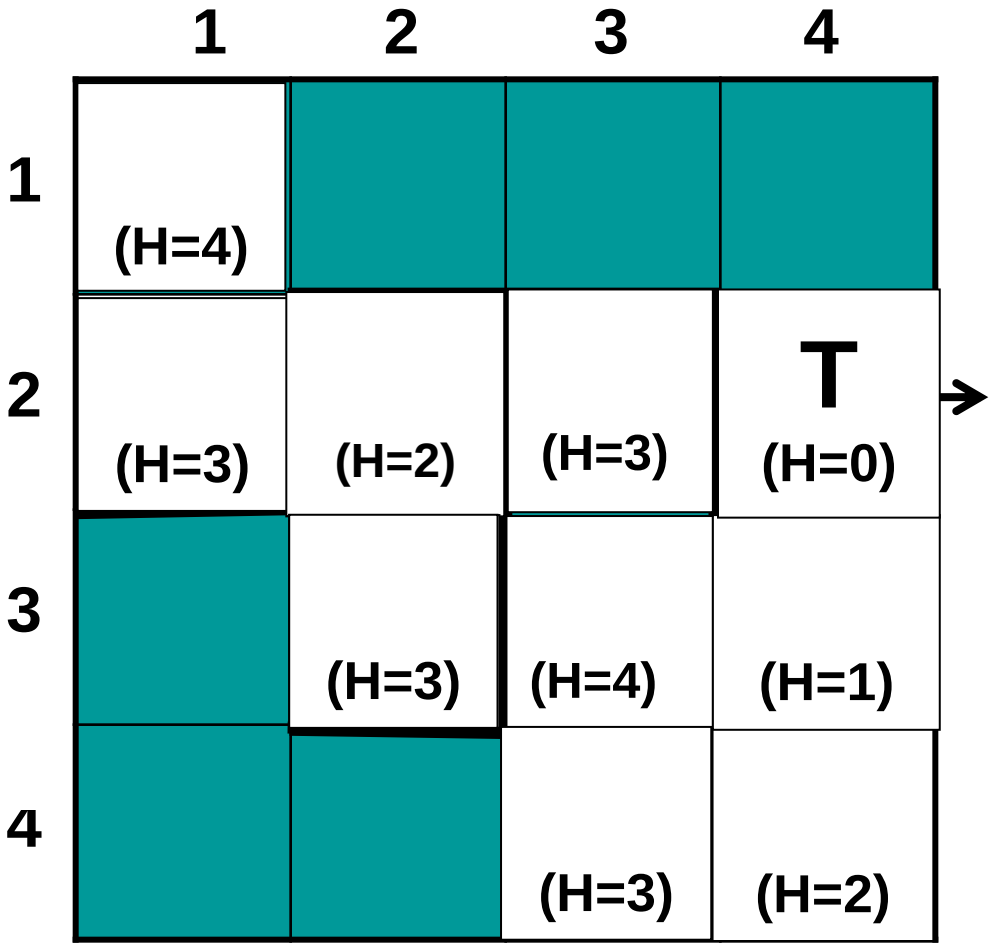
Costo-LRTA*(s, b, risultato[s, b], H)

s- ← s; return a

LRTA* supera i minimi locali (rev)



Esempio di LRTA*



Considerazioni su LRTA*

- LRTA* cerca di simulare A* con un metodo locale: tiene conto del costo delle mosse come può aggiornando al volo la H
- Completo in spazi esplorabili in maniera sicura
- Nel caso pessimo visita tutti gli stati due volte ma è mediamente più efficiente della profondità *online*
- Non ottimale, a meno di usare una euristica perfetta (non basta una $f=g+h$ con h ammissibile)
- Verso l'***apprendimento per rinforzo***!

Per informazioni

Alessio Micheli

micheli@di.unipi.it

Dipartimento di Informatica
Università di Pisa - Italy

Computational Intelligence &
Machine Learning Group