Agenti risolutori di problemi

Risolvere i problemi come ricerca

Maria Simi

a.a 2010/2011

Agenti *risolutori di problemi*

- Adottano il paradigma della risoluzione di problemi come ricerca in uno spazio di stati (*problem solving*).
- Sono particolari agenti con obiettivo, che pianificano l'intera sequenza di mosse prima di agire
- Passi che l'agente segue:
  1. Determinazione obiettivo (un insieme di stati)
  2. Formulazione del problema
  3. Determinazione della soluzione mediante ricerca
  4. Esecuzione del piano

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". Niente può andare storto.

Formulazione del problema

Un problema può essere definito formalmente mediante cinque componenti:

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

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

Formulazione del problema (cnt.)

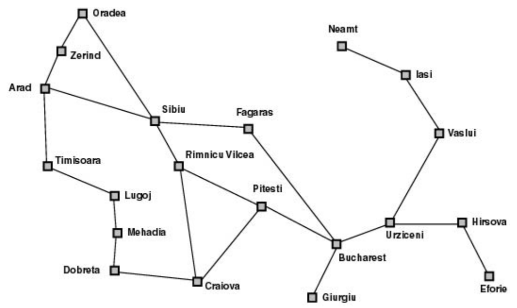
3. Test obiettivo:
   - Un insieme di stati obiettivo
   - Goal-Test: stato → {*true*, *false*}
4. Costo del cammino
   - somma dei costi delle azioni (costo dei passi)
   - costo di passo: c(*s*, *a*, *s*')
   - Il costo di un'azione/passo non è mai negativo

Algoritmi di ricerca

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

- *Misura delle prestazioni*
  Trova una soluzione? Quanto costa trovarla? uanto efficiente è la soluzione?
  Costo totale = costo della ricerca +
  costo del cammino soluzione

Itinerario: il problema

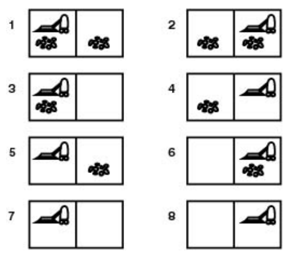


Itinerario: la formulazione

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

Aspirapolvere: il problema

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



Percezioni:

Sporco

NonSporco

Azioni:

Sinistra (L)

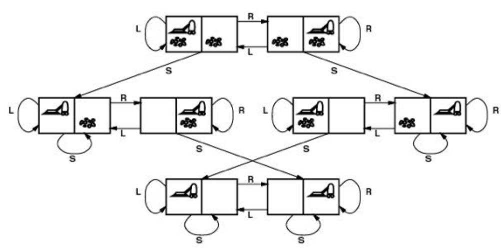
Destra (R)

Aspira (S)

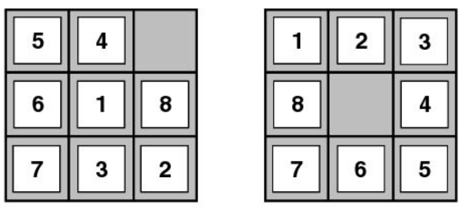
Aspirapolvere: formulazione

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

Spazio degli stati :



Il puzzle dell'otto

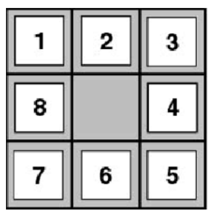


Start State

Goal State

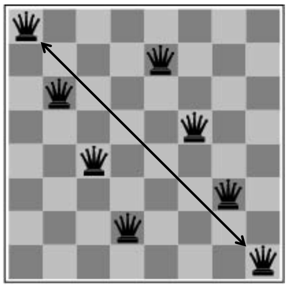
Puzzle dell'otto: formulazione

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



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

Le otto regine: il problema



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

Le otto regine:
Formulazione incrementale 1

- *Stati*: scacchiere con 0-8 regine
- *Goal-Test*: 8 regine sulla scacchiera, nessuna attaccata
- *Costo cammino*: zero
- *Azioni*: aggiungi una regina
  $64 \times 63 \times \ldots \times 57 \sim 3 \times 10^4$
  sequenze da considerare!

Le otto regine:
Formulazione incrementale 2

- *Stati*: scacchiere con 0-8 regine, nessuna minacciata
- *Goal-Test*: 8 regine sulla scacchiera, nessuna minacciata
- *Costo cammino*: zero
- *Azioni*: aggiungi una regina nella colonna vuota più a destra ancora libera in modo che non sia minacciata
  2057 sequenze da considerare

Le 8 regine:
Formulazione a stato completo

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

Dimostrazione di teoremi

- Il problema:
  Dato un insieme di premesse
  $\{s, t, q \Rightarrow p, r \Rightarrow p, v \Rightarrow q, t \Rightarrow r, s \Rightarrow v\}$
  dimostrare una proposizione p
- Nel calcolo proposizionale un'unica regola di inferenza, il *Modus Ponens (MP):*
  Se p e p ⇒ q allora q

Dim. teoremi: formulazione

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

continua

Dim. teoremi: spazio degli stati

{s, t, q⇒p, r⇒p, v⇒q, t⇒r, s⇒v}

{s, t, q⇒p, r⇒p, v⇒q, t⇒ r, s⇒ v, v}

{s, t, q⇒p, r⇒p, v⇒q, t⇒r, s⇒v, r}

{s, t, q⇒p, r⇒p, v⇒q, t⇒r, s⇒v, v, q}

{s, t, q⇒p, r⇒p, v⇒q, t⇒r, s⇒v, r, p}

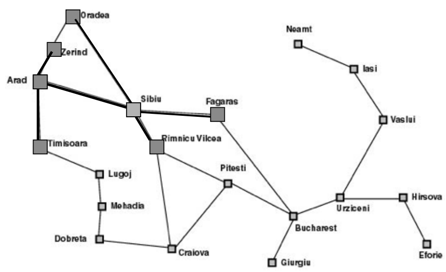
{s, t, q⇒p, r⇒p, v⇒q, t⇒r, s⇒v, v, q, p}

Problemi reali

- Pianificazione di viaggi arei
- Problema del commesso viaggiatore
- Configurazione VLSI
- Navigazione di robot
- Montaggio automatico
- Progettazione di proteine
- …

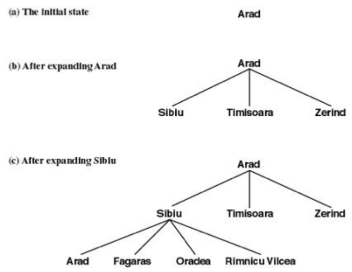
Ricerca della soluzione

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



Ricerca ad albero

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

I nodi dell'albero di ricerca

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

- Uno stato: *n*.stato
- Il nodo padre: *n*.padre
- L'azione effettuata per generarlo: *n*.azione
- Il costo del cammino dal nodo iniziale al nodo: *n*.costo-cammino indicata come g(*n*)

Struttura dati 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:
  - Vuota?(*coda*)
  - POP(coda) estrae il primo elemento
  - Inserisci(elemento, coda)
  - Diversi tipi di coda hanno diverse funzioni di inserimento e implementano strategie diverse

Diversi tipi di strategie

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

Strategie non informate

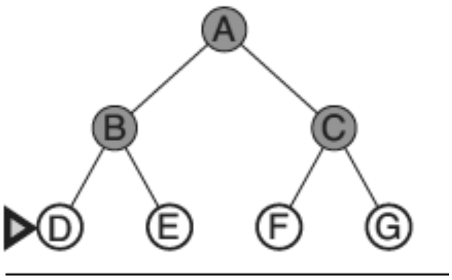
- Ricerca in ampiezza
- Ricerca di costo uniforme
- Ricerca in profondità
- Ricerca in profondità limitata
- Ricerca con approfondimento iterativo

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

Valutazione di una strategia

- C*ompletezza*: se la soluzione esiste viene trovata
- O*ttimalità* (ammissibilità): trova la soluzione migliore, con costo minore
- C*omplessità nel tempo*: tempo richiesto per trovare la soluzione
- C*omplessità nello spazio*: memoria richiesta

Ricerca in ampiezza (BF)



Implementata con una coda che inserisce alla fine (FIFO)

Ricerca in ampiezza (su albero)

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

Analisi spazio-temporale

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

Ricerca in ampiezza: analisi

- Strategia *completa*
- Strategia *ottimale* se gli operatori hanno tutti lo stesso costo k, cioè $g(n) = k \cdot \text{depth}(n)$, dove $g(n)$ è il costo del cammino per arrivare a n
- Complessità nel tempo (nodi generati)
  $T(b, d) = b + b^2 + \dots + b^d \rightarrow O(b^d)$
- Complessità spazio (nodi in memoria): $O(b^d)$

Nota: O notazione per la complessità asintotica

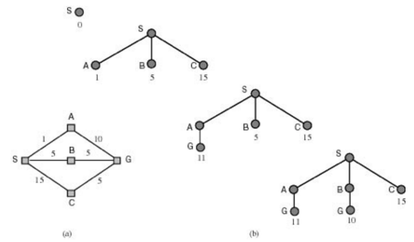
Ricerca in ampiezza: esempio

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

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

Ricerca di costo uniforme (UC)

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



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

Ricerca UC (su albero)

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

Costo uniforme: analisi

Ottimalità e *completezza* garantite purché il costo degli archi sia maggiore di ε>0.

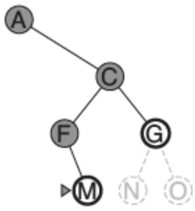
Complessità: $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$

C^* è il costo della soluzione ottima

$\lfloor C^*/\varepsilon \rfloor$ è il numero di mosse nel caso peggiore, arrotondato per difetto

Nota: quando ogni azione ha lo stesso costo UC somiglia a BF ma complessità $O(b^{1+d})$

Ricerca in profondità



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

Ricerca in profondità: analisi

- Se m distanza massima della soluzione nello spazio di ricerca
  b fattore di diramazione
  - Tempo: $O(b^{m+1})$
  - Occupazione memoria: $bm + 1$
- Strategia *non completa* e *non ottimale*.
- Drastico risparmio in memoria:

BF	d=16	10 esabyte
DF	d=16	156 Kbyte

Ricerca in profondità ricorsiva

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

Ricerca in profondità (su albero)

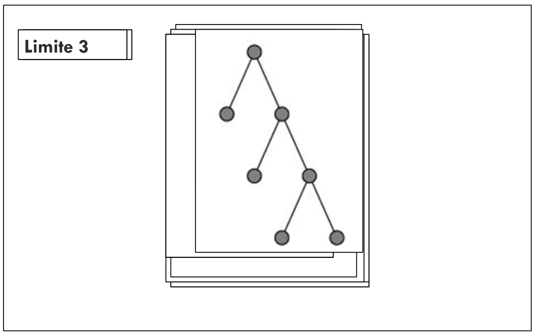
```
function Ricerca-DF-A (problema)
    returns soluzione oppure fallimento
    return Ricerca-DF-ricorsiva(CreaNodo(problema.Stato-iniziale), problema)

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

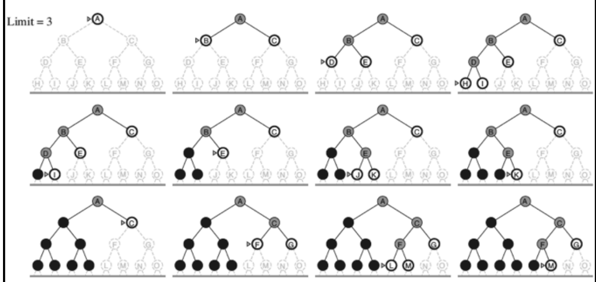
Ricerca in profondità limitata (DL)

- Si va in profondità fino ad un certo livello predefinito ℓ
- Co*mpleta* per problemi in cui si conosce un limite superiore per la profondità della soluzione.
  Es. Route-finding limitata dal numero di città – 1
- Completo: se $d < \ell$
- Non ottimale
- Complessità tempo: $O(b^{\ell})$
- Complessità spazio: $O(b.\ell)$

Approfondimento iterativo (ID)



Approfondimento iterativo (ID)



ID: analisi

- Miglior compromesso tra BF e DF
  BF: $b+b^2+ \ldots +b^{d-1}+b^d$ con b=10 e d=5
  10+100+1000+10.000+100.000=111.110
- ID: I nodi dell'ultimo livello generati una volta, quelli del penultimo 2, quelli del terzultimo 3 … quelli del primo *d* volte
  ID: $(d)b+(d-1)\ b^2+ \ldots +3b^{d-2}+2b^{d-1}+1b^d$
  = 50+400+3000+20.000+100.000=123450
- Complessità tempo: $O(b^d)$ Spazio: $O(b.d)$

Direzione della ricerca

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

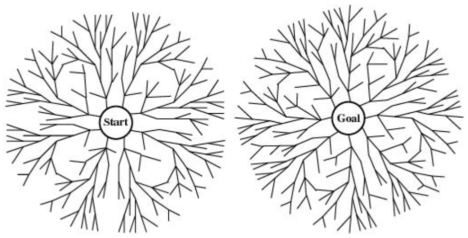
- ricerca *in avanti* o *guidata dai dati*: si esplora lo spazio di ricerca dallo stato iniziale allo stato obiettivo;
- ricerca *all'indietro* o *guidata dall'obiettivo*: si esplora lo spazio di ricerca a partire da uno stato goal e riconducendosi a sotto-goal fino a trovare uno stato iniziale.

Quale direzione?

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

Ricerca bidirezionale

Si procede nelle due direzioni fino ad incontrarsi



Ricerca bidirezionale: analisi

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

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

Confronto delle strategie

Criterio	BF	UC	DF	DL	ID	Bidir
Tempo	b^d	$b^{1+\lfloor C^*/\varepsilon \rfloor}$	b^m	b^ℓ	b^d	$b^{d/2}$
Spazio	b^d	$b^{1+\lfloor C^*/\varepsilon \rfloor}$	bm	$b.\ell$	bd	$b^{d/2}$
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)