Agenti risolutori di problemi

Questi agenti adottano il paradigma della *risoluzione dei problemi come* ricerca *in uno* spazio di stati (Problem solving). Sono Agenti con modello (storia delle percezioni e degli stati) che adottano una rappresentazione *atomica dello stato*. Sono particolari agenti con obiettivo, che pianificano l'intera sequenza di mosse prima di agire.

Il processo di risoluzione

1. Determinazione di un obiettivo (un insieme di stati in cui l'obiettivo è soddisfatto)
2. Formulazione del problema, rappresentando stati e azioni
   Questa fase presenta ancora un *design*
3. Determinazione della soluzione mediante ricerca, si cerca un *piano* ottimale con una soluzione *algoritmica*
4. Esecuzione del piano

> **Danger**
>
> Nel Problem Solving, l'intelligienza è spostata nel design della soluzione, una volta fatto un design opportuno, trovare l'algoritmo è facile.

Formulazione del problema

Un problema può essere definito formalmente mediante 5 componenti:

1. Stato iniziale
2. Azioni possibili in s: $Azioni(s)$
3. *Modello di transizione*:

   $$Risultato : stato \times azione \to stato$$
   $$Risultato(s, a) = s' \text{ uno stato successore}$$

4. *Test obiettivo*: un insieme di stati obiettivo, $Goal - test : stato \to \{true, false\}$
5. *Costo del cammino*: la somma dei costi delle azioni (costo dei passi), dove il costo di ogni passo è $c(s, a, s')$ ed è sempre $c(s, a, s') \geq 0$

Dal punto 1 al punto 3 viene definito implicitamente lo spazio degli stati. Definirlo esplicitamente può essere molto oneroso, come in quasi tutti i problemi di AI. Questo perché l'agente non conosce già tutto in anticipo, per alcuni problemi servirebbe uno spazio enorme. Dobbiamo quindi trovare la soluzione ottima in uno spazio definito implicitamente.

Esempio

Esempio di formulazione del problema - viaggio in Romania

Algoritmi di ricerca

> Ricerca
>
> Il processo che cerca una sequenza di azioni che raggiunge l'obiettivo è detto 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: $\text{costo totale} = \text{costo ricerca} + \text{costo del cammino soluzione}$, misura tre aspetti: trova una soluzione? quanto costa trovarla? quanto costa la soluzione?
Noi valuteremo gli algoritmi sul costo della ricerca, poi ottimizzeremo sul secondo. Uno è il costo per pianificare e l'altro per

fare il viaggio (dobbiamo ottimizare questo). Se il costo della ricerca è troppo più alto può anche essere impossibile trovare una soluzione.

Esempi di problemi

Esempi

Dimostrazione di Teoremi

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 consideriamo un'unica regola di inferenza, il *Modus Ponens* (*MP*):

$$\text{Se } p \text{ e } q \implies q \text{ allora } q$$

Formulazione

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

Esempi di problemi reali

- Pianificazione di viaggi aerei
- Problema del commesso viaggiatore
- Configurazione VLSI
- Navigazione di robot (spazio continuo)
- Montaggio automatico
- Progettazione di proteine
- ...

Ricerca della soluzione

Viene generato un albero di ricerca sovrapposto allo spazio degli stati, generato da *possibili* sequenze di azioni.

Nota

$nodo$ è diverso da $stato$. Possono esistere nodi nell'albero con lo stesso stato.

Esempio di ricerca della soluzione

Algoritmo

```
function Ricerca-Albero (problema) returns solzione oppure fallimento
    inizializza la frontiera con lo stato iniziale del problema
    loop do
        if (la frontiera è vuota) then return fallimento
        Scegli un nodo foglia da espandere e rimuovilo dalla frontiera //(1)
        if (il nodo contiene con uno stato obiettivo) //(2)
            then return la soluzione corrispondente
        Espandi il nodo e aggiungi i successori alla frontiera //(3)
```

- La riga *(1)* non specifica la strategia da usare per scegliere
- La riga *(2)* esamina l'opzione corrente
- La riga *(3)* passa alle opzioni successive in caso non venga trovata la soluzione

I nodi dell'albero di ricerca

Un nodo n è una struttura dati con 4 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 $g(n)$ ($= padre.costo - cammino + costo - passoultimo$).

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.

Tipi di strategie

```
    - <span style="color:#ff82b2"><b>FIFO - First In First Out</b></span> $\rightarrow$ BF (Breadth-first)
Viene estratto l'elemento più vecchio (in attesa di più tempo); i nuovi nodi sono aggiunti alla fine.
```

- LIFO - Last In First Out → DF (Depth-first)
  Viene estratto il più recentemente inserito; i nuovi nodi sono inseriti all'inizio (pila)
- Coda con priorità → UC, e altri successivi
  Viene estratto quello con priorità più alta in base a una funzione di ordinamento; dopo l'inserimento dei nuovi nodi si riordina

Strategie Informate e non

Strategie non informate	Strategie informate
Ricerca in ampiezza (BF), ricerca in profondità(DF), ricerca in profondità limitata (DL), ricerca con approfondimento iterativo (ID), ricerca di costo uniforme (UC)	Le strategia di ricerca euristica (o informata) fanno uso di informazioni riguardo alla distanza stimata dalla soluzione

Valutazione di una strategia

- *Completezza:* Se la soluzione esiste viene trovata
- *Ottimalità (ammissibilità):* trova la soluzione migliore, con costo minore
- *Complessità in tempo:* tempo richiesto per trovare la soluzione
- *Complessità in spazio:* memoria richiesta

Ricerca in ampiezza - BF

Questo algoritmo potrebbe generare nodi con stati già visitati.
Il problema viene ripassato perché la situazione cambia (es. potrebbe cambiare il costo). Implemento una coda di tipo FIFO.
I `nodo.stato` sono goal-tested al momento in cui sono generati, la visita è quindi anticipata, è più efficiente in quanto si ferma appena trova il goal prima di espandere.

Algoritmo 1

```
function Ricerca-Ampiezza-A (problema)
        returns soluzione oppure fallimento
    nodo = un nodo con stato 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)
```

Algoritmo 2

```
function Ricerca-Ampiezza-A (problema)
        returns soluzione oppure fallimento
    nodo = un nodo con stato 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
    esplorati = insieme vuoto
    loop do
        if Vuota?(frontiera) then return fallimento
        nodo = POP(frontiera)
        aggiungi nodo.Stato a esplorati
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione)
            if figlio.Stato non è in esplorati e non è in frontiera then
                if Problema.TestObiettivo(figlio.Stato)
                    then return Soluzione(figlio)
                frontiera = Inserisci(figlio, frontiera)
```

Questa versione *non esplora nodi già esplorati*. É stata aggiunta una lista (molto costosa) per tenere memoria dei nodi già visitati. Quando si fa il `POP` il nodo viene aggiunto alla lista `esplorati`. Prima di fare il test obiettivo e prima di espandere il nodo allargando la frontiera, si controlla che il nodo non sia nella lista degli esplorati o nella frontiera.
Non si può sostituire la lista con una tabella statica che contiene tutti gli stati possibili (con delle flag) perché sono troppi e non avrebbe senso mantenere una tabelle così grande (problema di efficenza).

Analisi della complessità

Dati:

$\rightarrow b =$ fattore di ramificazione (**b**ranching, numero max di successori)
$\rightarrow d =$ profondità del nodo obiettivo più superficiale (**d**epth, più vicino all'iniziale)
$\rightarrow m =$ lunghezza massima dei cammini nello spazio degli stati

La strategia è ottimale se gli operatori hanno tutti lo stesso costo k, cioè se $g(n) = k \cdot depth(n)$, dove $g(n)$ è il costo minimo del cammino per arrivare a n.
Le Complessità in tempo(nodi generati) e quella in spazio(nodi in memoria) sono entrambe $O(b^d)$.

Profondità	Nodi	Tempo	*Memoria*
2	110	0,11 ms	107 kilobyte
4	11.10	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	1 esabyte

Ricerca in Profondità (DF)

Implementata da una coda che mette i successori in testa alla lista (LIFO). L'algoritmo è uguale alla visita in ampiezza ma per ogni nodo visita i figli prima dei fratelli.

Analisi della versione su un albero

Dati m (lunghezza max dei cammini nello spazio degli stati) e b (fattore di diramazione), la complessità in tempo è data da $O(b^m)$ (può essere . $> O(b^d)$) e l'occupazione di memoria da $b \cdot m$.
Questo algoritmo può trovarsi in un ciclo "*senza saperlo*" e rimanerci all'infinito. Il risparmio in memoria è però molto più efficiente.

Analisi della versione su un grafo

In questo caso perde i vantaggi di memoria: la complessita in spazio da $b \cdot m$ torna a tutti i possibili stati per mantenere la lista degli stati già visitati, ma così DF diviene completa in spazi degli stati finiti (al caso pessimo vengono espansi tutti i nodi).
É possibile controllare anche solo i nuovi stati rispetto al cammino radice-nodo corrente senza aggravio in memoria, evitando però solo i cicli in spazi finiti ma non i cammini ridondanti.

Versione ricorsiva

É ancora più efficiente in occupazione di memoria perché *mantiene solo il cammino corrente* (solo m nodi al caso pessimo). Viene 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, generando i nodi fratelli al momento del backtracking.

```
function Ricerca DF-A (problema) returns souzione 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)

Questa versione va in profondità fino a un certo livello predefinito l, è completa per problemi di cui si conosce un limite superiore per la profondità della soluzione (ad es. per il problema della strada in Romania il limite è $\text{numero delle città} - 1$), quindi se $d < l$. La *complessità in tempo* è $O(b^l)$, quella in *spazio* è $O(b \cdot l)$.

Approfondimento iterativo (ID)

Si sfrutta la *ricerca in profondità* ma *non si assume di conoscere* l, quindi l'equivalente di l si incrementa di 1 ogni volta che si finisce l'l-esimo livello di profondità e ad ogni giro controlla tutti i nodi sistematicamente.
Anche se fa molte ripetizioni, dato che ad ogni giro controlla tutti i nodi dell'albero fino al livello l, è un buon compromesso tra BF e DF in termini di efficienza.
Se esiste una soluzione, la *complessità in tempo* è $O(b^d)$ e quella *in spazio* è $O(b \cdot d)$.
É completo ed ottimale se il costo degli operatori è fisso.

Direzione della ricerca

La direzione della ricerca può essere:

- In avanti(o *guidata dai dati*): si esplora lo spazio di ricerca dallo stato iniziale allo stato obiettivo

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

Ricerca bidirezionale

Si precede nelle 2 direzioni fino ad incontrarsi

Analisi

Complessità in Tempo	Complessità in Spazio
$O(b^{\frac{d}{2}})$ - assumendo che il test di intersezione si esegua in tempo costante (es. hash table)	$O(b^{\frac{d}{2}})$ - almeno tutti i nodi sono in una direzione in memoria (es. si usa BF)

Questo approccio non è sempre applicabile, se ad esempio i predecessori non sono definiti o ci sono troppi stati obiettivo.

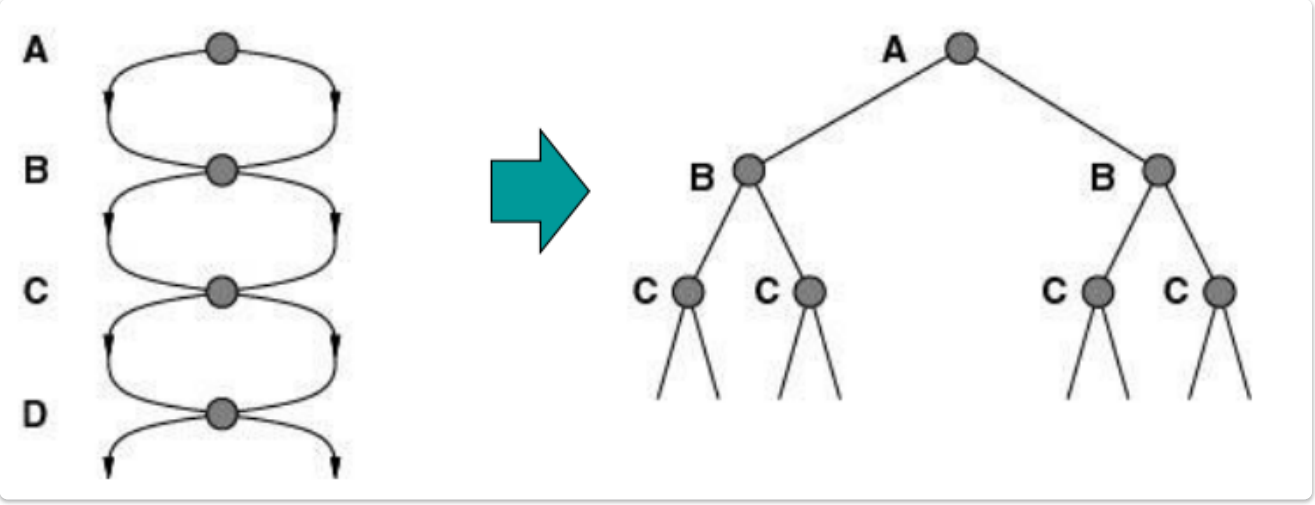
Cammini ciclici

I cammini ciclici rendono gli alberi di ricerca infiniti, anche se hanno spazio degli stati finiti. Nell'esempio del viaggio in romania, $In(Arad)$ può ripetersi all'infinito.

Ridondanze

RICERCA SU GRAFI

Sugli spazi degli stati a grafo si generano più volte gli stessi nodi (i nodi con lo stesso stato) nella ricerca, *anche in assenza di cicli*.



RIDONDANZE NELLE GRIGLIE

Visitare stati giù visitati fa compiere un lavoro inutile. Si può evitare con un *costo di* 4^d ma con $\approx 2d^2$ *stati distinti*.

Compromesso tra spazio e tempo

Ricordare gli stati già visitati è molto costoso in termini di spazio (es. lista esplorati nella visita a grafo), ma ci consente di evitare di visitarli di nuovo. *Gli algoritmi che dimenticano la propria storia sono destinati a ripeterla*.

Tre Soluzioni

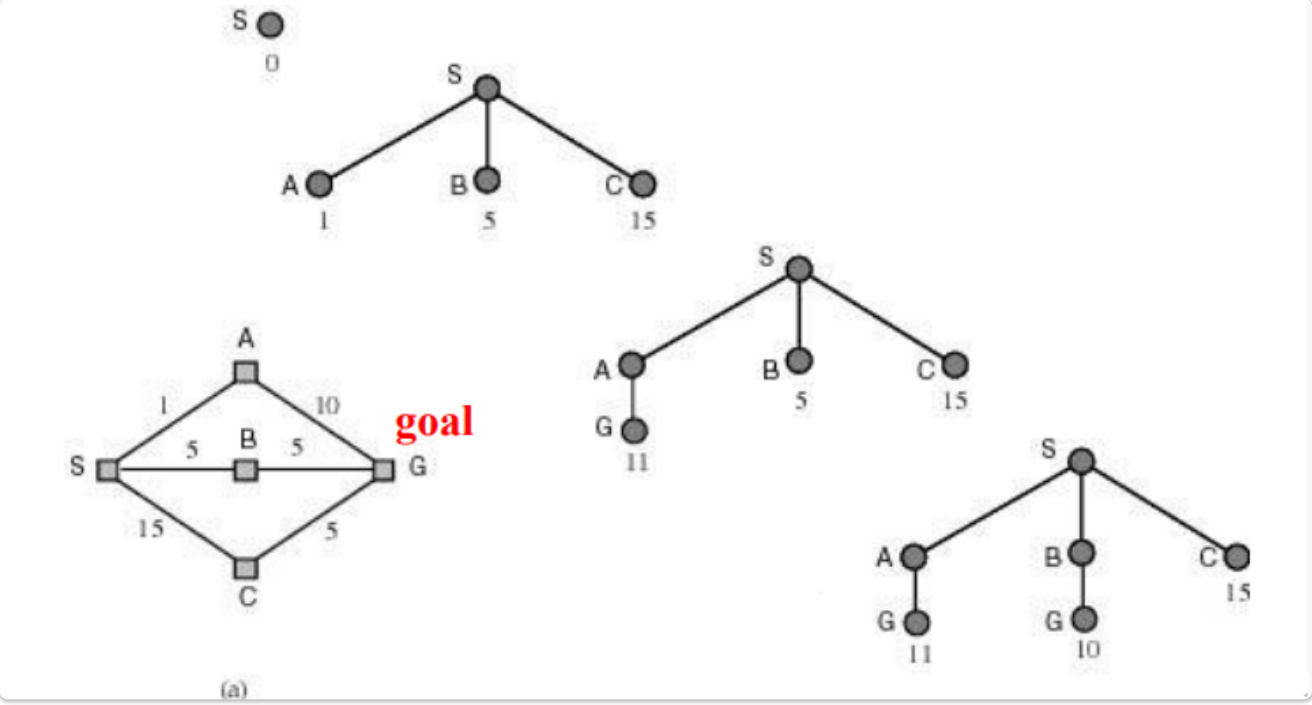
In ordine crescente di costo e di efficacia:

1. *Non tornare nello stato da cui si proviene:* si elimina il genitore dai nodi successivi, non si evitano però i cammini ridondanti
2. *Non creare cammini con cicli:* si controlla che i successori non siano antenati del nodo corrente, è già stato detto per la DF
3. *Non generare nodi con stati già visitati/esplorati:* ogni nodo visistato deve essere tenuto in memoria per una complessità $O(s)$, dove s è il numero di stati possibili (es. hash table per accesso efficiente)
   Repetita: Il costo può essere alto, in caso di DF la memoria torna da bm a tutti gli stati, ma diviene una ricerca completa (per spazi finiti). Ma in molti casi gli stati crescono esponenzialmente.

Ricerca "su grafi" (Repetita)

Mantiene una lista dei nodi (*stati*) visitati/esplorati, detta anche **lista chiusa**. Prima di espandere un nodo si controlla se lo stato è già stato incontrato prima o è già nella frontiera. Se questo succede, il nodo appena trovato non viene espanso. Questo algoritmo è ottimale solo se abbiamo la garanzia che il costo del nuovo cammino sia maggiore o uguale, cioè che il nuovo cammino non conviene.
Esempio del viaggio in romania

Ricerca di costo uniforme (UC)



Questo algoritmo è una *generalizzazione della ricerca in ampiezza*: si sceglie il nodo di costo minore sulla frontiera, si espande sui contorni di *uguale (uniforme) costo*, invece che sui contorni di uguale profondità. É implementata da una coda ordinata per costo di cammino crescente (in cima i nodi di costo minore).

Ricerca UC su albero - Algoritmo

```
function Ricerca-UC-A (problema) returns soluzione oppure fallimento
    nodo = un nodo con
        stato = problema.stato-iniziale
        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
```

In questo algoritmo la visita è posticipata, per vedere il costo minore su `g`. É diverso da DF, ma tipico per i sistemi basati su una coda con priorità.

Ricerca-grafo UC - Algoritmo

```
function Ricerca-UC-G (problema) returns soluzione oppure fallimento
    nodo = un nodo con
        stato = problema.stato-iniziale
        costo-di-cammino=0
    frontiera = una coda con priorità con nodo come unico elemento
    esplorati = insieme vuoto
    loop do
        if Vuota?(frontiera) then return fallimento
        nodo = POP(frontiera)
        if problema.TestObiettivo(nodo.Stato)
            then return Soluzione(nodo)
        aggiungi nodo.Stato a esplorati
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione)
            if figlio.Stato non è in esplorati e non è in frontiera
                then frontiera = Inserisci(figlio, frontiera)
                /* in coda con priorità
            else if figlio.Stato è in frontiera con Costo-cammino più alto
                then sostituisci quel nodo frontiera con figlio
```

Anche qui la visita è posticipata per vedere il costo minore.

Analisi UC

L'ottimalità e la completezza sono garantite purché *il costo degli archi sia maggiore di* $\varepsilon > 0$.
Dato C^* come il *costo della soluzione ottima*, $\lfloor C^*/\varepsilon \rfloor$ è il *numero di mosse nel caso peggiore* arrotondato per difetto. Ad esempio nel caso di andare verso tante mosse di costo ε prima di una che parta più alta, ma poi abbia un path a costo totale più basso. Abbiamo 2 elementi principali

- coda con priorità dei nodi
- presenta un test posticipato (non quando si genera ma quando si estrae), anche se nell'algoritmo è prima.
  La complessità è quindi $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, quando ogni azione ha lo stesso costo, UC somiglia a BF ma complessità $O(b^{1+d})$.

Confronto delle strategie su un albero

Criterio	BF	UC	DF	DL	ID	BiDir
Completa?	si	si (^)	no	si (+)	si	si
Tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor}))$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{\frac{d}{2}})$
Spazio	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor}))$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{\large\frac{d}{2}})$
Ottimale?	si(*)	si (^)	no	no	si (*)	si

Conclusioni

Un agente per il problem solving adotta un paradigma generale di risoluzione dei problemi:

- formaula il problema

- ricerca la soluzione nello spazio degli stati
  Le strategia per la ricerca della soluzione sono "non informate".