

Algoritmi di Ricerca

Algoritmi per l'intelligenza artificiale

Vincenzo Bonnici

Corso di Laurea Magistrale in Scienze Informatiche
Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Università degli Studi di Parma

2025-2026

E' uno dei processi intellettivi che secondo il comportamentismo richiede e definisce l' "attività intellettuale".

- ① Induzione (Apprendimento)
- ② Sussunzione (Riconoscimento)
- ③ Ragionamento (Deduzione)
- ④ Problem Solving (implica tutte le precedenti)

- Il PS è un'attività tipicamente definita come “razionale”.
- Razionale = ciò che si basa su procedimenti esprimibili in formula chiusa (algoritmi)
- L'AI ha dimostrato che più un task richiede “razionalità” in senso stretto più l'uomo perde il confronto con la macchina.
- Hard Computing vs. Soft Computing
 - L'hard computing si basa sulla logica binaria e su istruzioni predefinite come un'analisi numerica e un software veloce e utilizza la logica a due valori.
 - Il soft computing si basa sul modello della mente umana in cui ha ragionamento probabilistico, logica fuzzy e utilizza una logica multivalore

“Come” costruire un Problem Solver ?

- Approccio Human oriented (cognitivist)
- Deve SIMULARE l'attività intelligente
- Risolvere problemi “pensando come un uomo”
- “Bias della razionalità”
- Approccio Machine oriented (comportamentale)
- Deve MANIFESTARE attività intelligente
- Risolvere i problemi al meglio

A cosa “serve” un Problem Solver?

- Spiegare in maniera computazionale come l'uomo risolve i problemi
- Fornire all'uomo uno strumento di supporto

General Problem Solver (Newell, Simon e Shaw 1958)

- Risultato dell'approccio cognitivo
- Esperienza previa con "Logic Theorist"
- Uso di resoconti di Problem Solver umano
- Analisi mezzi fini

Analisi Mezzi-Fini

- ➊ Identificare la più grande differenza tra stato attuale e stato desiderato.
- ➋ Creare un sotto obiettivo che cancelli questa differenza.
- ➌ Selezionare un operatore che raggiunga questo sotto obiettivo
- ➍ SE l'operatore può essere applicato ALLORA lo si applica e, arrivati nello stato successivo, si riapplica l'A-M-F.
- ➎ SE l'operatore non può essere applicato ALLORA si usa l'A-M-F in modo ricorsivo per rimuovere la condizione di blocco.

- Problem Solver che MANIFESTA intelligenza
 - Algoritmi di Ricerca
- Problem Solving = ricerca nello spazio degli stati.
 - PS = Hard Computing
- Il bias della “potenza di calcolo”:
 - Con calcolatori sufficientemente potenti si può “attaccare” ogni tipo di problema.
 - Falso: l'esplosione combinatoria rende futile la forza bruta

Cosa è un problema?

- E' un concetto non definibile, solo esemplificabile. (Nilsson,1982)
- Alcuni esempi:
 - I puzzle "da tavola" \mapsto in genere NP
 - "Commesso viaggiatore"
 - Rompicapo come il Cubo di Rubik
 - SAT, Dimostrazione teoremi
 - Giochi (Dama, Scacchi, etc.)
 - VLSI (Very-large-scale integration)

Formalizzazione

5-tupla di elementi $P = \{X, SCS, x_0, g, t\}$

- X spazio degli stati
- SCS strategia/spazio di ricerca
 - dettata da azioni possibili/operatori
- x_0 stato iniziale
- g funzione costo
- t stato terminale

3	7	4
6	1	
5	2	8

1	2	3
4	5	6
7	8	

I giochi nell'IA e non solo

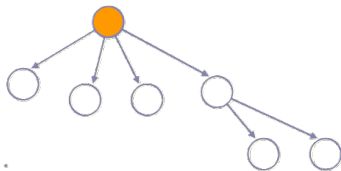
- M. Minsky (1968):
 - “i giochi non vengono scelti perché sono chiari e semplici, ma perché ci danno la massima complessità con le minime strutture iniziali”
- Pungolo Scientifico
 - Matematica: teoria dei grafi e complessità
 - Computer Science: database e calcolo parallelo
 - Economia: teoria dei giochi

Teoria dei giochi - Von Neumann & Morgenstern (1944)

- Teoria della Decisione
 - Analizzare il comportamento Individuale le cui azioni hanno effetto diretto
 - Scommesse & Mondo dei Puzzle
- Teoria dei Giochi
 - Analizzare il comportamento Individuale le cui azioni hanno effetto che dipende dalle scelte degli altri
 - Mondo dei Giochi a più giocatori

"Blind" search (ricerca cieca) - Grafi e strategie

- Spazio di ricerca \mapsto $(SCS(SCS(\dots(x_0))))$
 - Alberi (grafi) e nodi
- Cosa vuol dire trovare una soluzione?
- Cosa è una strategia di ricerca?



Valutare le strategie

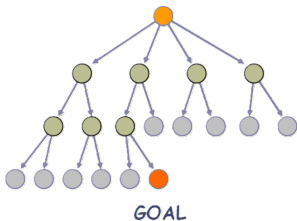
- Criteri fondamentali
 - Completezza
 - Ottimalità
 - Complessità Spaziale
 - Complessità Temporale
- Le “regole d’oro” di J.Pearl (1984)
 - Non dimenticarsi di cercare sotto ogni pietra
 - Non alzare due volte la stessa

- Come espandere un nodo?
- **Coda** dei nodi aperti
 - CODA.insert(node);
 - node = CODA.remove();
- L'ordinamento dei nodi in CODA determina la strategia di ricerca

```
if (goal_test(x0 ) == true ) return SUCCESS
else CODA.insert(x0)
do {
    if (CODA.isEmpty()) return FAILURE
    nodo = CODA.remove()
    figli[] = SCS(nodo)
    CODA.insert(figli)
} while ( goal_test(nodo) == false)
return SUCCESS
```

“Breadth First” - Ricerca in Ampiezza

- Usa una memoria FIFO
- E' un algoritmo “difensivo”
- E' completo e ottimale
- Complessità spaziale: $O(b^d)$
 - b = branching factor (grado medio dei nodi dell'albero)
 - d = profondità della soluzione (nodo dell'albero)
- Complessità temporale: $O(b^d)$

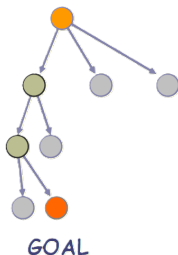


depth	N° nodi	Tempo	Memoria
2	111	1 msec	11 KB
4	11111	0,1 sec	1 MB
6	$>10^6$	10 sec	>100 MB
8	$>10^8$	17 min	>10 GB
10	$>10^{10}$	28 ore	>1 TB
12	$>10^{12}$	116 giorni	>100 TB
14	$>10^{14}$	32 anni	>10000 TB

$b = 10$, velocità ricerca = 100 mila nodi/sec., 100 byte/nodo

“Depth First” - Ricerca in Profondità

- Usa una memoria LIFO
- E' un algoritmo “aggressivo”
- E' non completo e non ottimale
- Complessità temporale: $O(b^d)$
 - b = branching factor (grado medio dei nodi dell'albero)
 - d = profondità della soluzione (nodo dell'albero)
- Complessità spaziale: $O(db)$



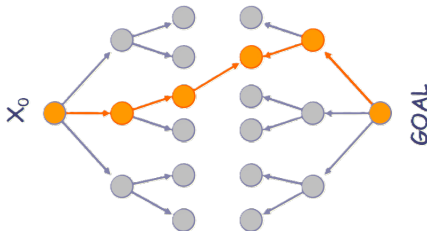
backtracking

Come migliorarli?

- Conoscendo lo stato goal
- Non ripetendo gli stati
 - Evitando di espandere lo stato di provenienza
 - Evitando i cicli
 - In generale: evitando di generare nodi con stati già visitati nella ricerca
- Conoscendo il costo degli operatori

Ricerca Bidirezionale (sfruttare la conoscenza dello stato)

- Ricerca in Ampiezza
 - Dallo stato iniziale verso il goal
 - Dal goal verso lo stato iniziale
- Termina quando le due ricerca si incontrano
- Perché non usare 2 “depth first”?
- E' completa e ottimale
- Complessità spaziale: $O(b^{\frac{d}{2}})$
- Complessità temporale: $O(b^{\frac{d}{2}})$



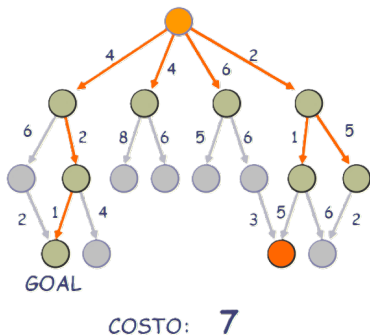
Ricerca a profondità limitata (evitare di cadere in loop)

- Ricerca in Profondità
 - Si stabilisce una profondità massima l
 - Se la coda è vuota al raggiungimento di l si ritorna un fallimento
- Non è completa (se $l < d$) né ottimale
- Complessità spaziale: $O(bl)$
- Complessità temporale: $O(b^l)$
- PRO: evita loop infiniti senza usare memoria!
- CON: richiede conoscenza a priori del problema

- Ricerca in Profondità limitata
 - Passo 1: $l = 0$
 - Passo 2:
 - si applica la ricerca a profondità limitata partendo da x_0
 - se la coda è vuota al raggiungimento di l si reitera il passo 2 aumentando l
- E' ottimale e completa
- Complessità spaziale: $O(bd)$
- Complessità temporale: $O(b^d)$
- CON: si espandono più volte gli stessi stati. Quanto?

Ricerca a costo uniforme (sfruttare la conoscenza del costo degli operatori)

- La “ Breadth First” Search
 - minimizza il costo di cammino della soluzione se la funzione di costo per ogni operatore è costante (es: 1)
- La “ Uniform-Cost” Search
 - minimizza il costo di cammino anche con operatori a costo variabile (es: “commesso viaggiatore”)
 - funzione di costo: $g(n)$
 - Requisito: $g(n) \leq g(SCS(n))$, cioè costo non negativo.
 - Altrimenti non c'è strategia che tenga!
 - E' completa e ottimale.



```
function Dijkstra(Graph, source):  
    for each vertex v in Graph.Vertices:  
        dist[v] = INFINITY  
        prev[v] = UNDEFINED  
        add v to Q  
    dist[source] = 0  
  
    while Q is not empty:  
        u = vertex in Q with min dist[u]  
        remove u from Q  
  
        for each neighbor v of u still in Q:  
            alt = dist[u] + Graph.Edges(u, v)  
            if alt < dist[v]:  
                dist[v] = alt  
                prev[v] = u  
  
    return dist[], prev[]
```

- Sì. Ricerca Uniforme = Algoritmo Dijkstra (modificato)
 - Ricerca Uniforme è una variante dell'algoritmo di Dijkstra per grafi molto grandi.
 - Invece di inserire tutti i vertici in una coda di priorità, inseriamo solo la sorgente, quindi inseriamo uno per uno quando necessario. In ogni passaggio controlliamo se l'elemento è già nella coda di priorità (utilizzando l'array visited). Se sì, eseguiamo il tasto decrementa, altrimenti lo inseriamo.
- E' il progenitore degli algoritmi Best-First
- Complessità temporale Dijkstra $O(X^2)$
- Complessità temporale ricerca uniforme $O(b^d)$

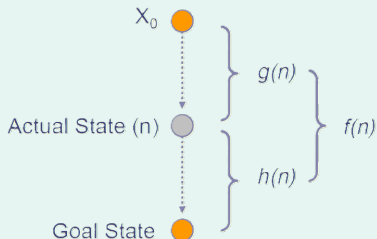
“Heuristic” Search

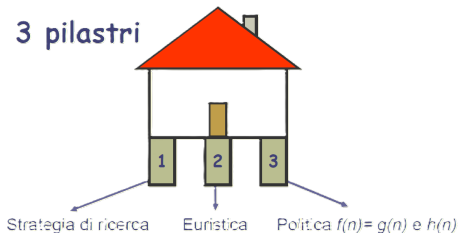
Cosa è una euristica?

- “Qualsiasi cosa” che serva di supporto in un processo decisionale
- E' una conoscenza, magari imperfetta, del dominio in cui ci troviamo
- Un esempio reale: “la Carta di Mercatore
- Tipicamente nel Problem Solving:
 - Valutazione del costo di cammino futuro

Come usare una euristica?

- $g(n)$ costo effettivo dello stato corrente
- $h(n)$ costo previsto per arrivare allo stato finale
- $f(n)$ costo per arrivare allo stato finale. Anche detta funzione di valutazione





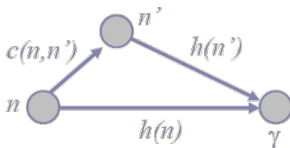
Due esempi di euristiche

- Tessere fuori posto
 - $h_{fp} = 5$
- Distanza di Manhattan
 - $h_M = 11$

3	2	4
8	5	
7	6	1

Proprietà generali delle Euristiche

- **Ammissibilità** (non sovrastima mai il costo effettivamente necessario per raggiungere l'obiettivo):
 - $h(n)$ è ammissibile se $h(n) \leq h^*(n) \forall n$
 - con h^* la migliore delle euristiche (costo del percorso ottimo)
- **Dominanza**:
 - h_2 domina h_1 se
 - h_1 e h_2 sono ammissibili
 - $h_1(n) \leq h_2(n) \forall n$
- **Consistenza**:
 - $h(n)$ è consistente se $h(n) \leq c(n, n') + h(n') \forall (n, n')$
- **Monotonicità**:
 - $h(n)$ è monotona se $h(n) \leq c(n, n') + h(n') \forall n, n' \in SCS(n)$



Dim: consistenza = ammissibilità

- 1 Per def.: $h(n) \leq c(n, n') + h(n') \quad \forall (n, n')$
- 2 Allora possiamo sostituire n' con un nodo risolvante γ
- 3 Quindi, $h(n) \leq c(n, \gamma) + h(\gamma)$
- 4 $h(\gamma) = 0$ e $c(n, \gamma) = h^*(n)$ per $\gamma \in \Pi^*$ (percorso ottimo)
- 5 da 3 e 4 abbiamo che $h(n) \leq h^*(n)$

Dim: monotonicità = consistenza

- 1 Per def.: $h(n) \leq c(n, n') + h(n') \quad \forall n, n' \in SCS(n)$
- 2 e anche $h(n') \leq c(n', n'') + h(n'') \quad \forall n', n'' \in SCS(n')$
- 3 ripetendo il punto 2 con: $n' \leftarrow n''$ e $c(n, n'') \leftarrow c(n, n') + c(n', n'')$
rimane garantito che $h(n) \leq c(n, n'') + h(n'')$
 $\forall n, n'' \in SCS(\dots(SCS(n)))$
- 4 quindi, $h(n) \leq c(n, n') + h(n') \quad \forall (n, n')$

Ammissibilità e Ottimalità

- SE $h(n)$ è ammissibile
- SE l'algoritmo usato è Best-First
- SE $h(n)$ non viene MAI pesato più di $g(n)$
- ALLORA la Ricerca è ottimale

Formalmente: $f(n) \leq f^* \quad \forall n$

con f^* il costo effettivo minimo, a partire da qualsiasi x_0 , per arrivare al nodo finale

Altre conseguenze

- se un'euristica è monotona allora è ammissibile
- se un'euristica è monotona allora $f(.)$ non decresce (per cui vale $f(n) \leq f^* \forall n$)
- se h è non ammissibile la si può rendere ammissibile garantendone il vincolo di monotonicità!
- se h_1 domina h_2 tutti i nodi espansi usando BEST-FIRST e h_1 sono espansi usando h_2
- Ogni nodo n' espanso da un algoritmo BEST -FIRST con h ammissibile costruisce un percorso ottimo tra n_0 e n'

Come ottenere un'euristica ammissibile?

Analizziamo e rilassiamo i vincoli sugli operatori.

Per muovere una tessera da A a B:

3	7	4
6	1	
5	8	2

- ① A e B devono invertire le proprie posizioni
- ② B deve essere una casella vuota
- ③ A e B devono essere confinanti
- ④ Un solo movimento per volta

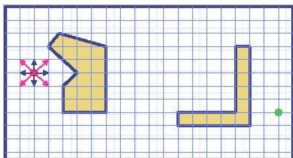
- Rilassando 1,2,3,4: $h(n) = 1 \mapsto$ non c'è informazione
- Rilassando 1,2,3: $h_1(n) = 7 \mapsto h_{fp}(n)$
- Rilassando 1,2,: $h_2(n) = 17 \mapsto h_m(n)$
- Non rilassando: si deve risolvere il problema stesso!

Esempi di euristiche ammissibili

- Numero di tessere fuori posto
- Distanza di Manhattan
 - Somma delle distanze ortogonali delle parti (le tessere nel Puzzle di Sam Loyd) dalle loro posizioni nel goal state.
 - E' ammissibile
 - E' monotona.
 - Rispetta la parità di $h^*(n)$
 - E' pienamente informata quando siamo vicini all'obiettivo
- $h_3 = h_{fp} + h_m$ non è ammissibile!

3	7	4
6	1	
5	8	2

Navigazione robot tra ostacoli.



$h(n)$ = distanza in linea retta

SE il costo degli step è 1 per movimento ortogonale e $\sqrt{2}$ per movimento in diagonale

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Per ogni permutazione delle tessere “fringe” si risolve il sottoproblema, considerando il pivot e le altre tessere.

Si memorizza in un DB il numero di mosse.

Si fa lo stesso per l'altro Pattern.

- Pattern Database è la prima euristica memory based.
- Vi sono $N!/(N! - n!)$ permutazioni delle tessere fringe compreso il pivot. Fringe del 15 puzzle = 519 milioni (495 MB memoria).
- Usando solo il fringe: speedup su $h_m = 6$, nodi = 1/346
- Usando il doppio Pattern: speedup = 12, nodi = 1/2000
- Grandi miglioramenti con Disjoint Pattern DB (Korf & Taylor, '02)

Vantaggi delle Euristiche:

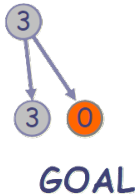
- Nella pratica: migliorano i tempi della ricerca.
- In teoria: solo un'euristica “oracolare” elimina la complessità esponenziale!
 - Constant Absolute Error: $h^*(n) - h(n) = \text{costante} \mapsto$ costo di ricerca lineare $b \cdot d$
 - Constant Relative Error: $h^*(n) - h(n) = \text{dipende da } h^* \mapsto$ costo di ricerca b^d
- La ricerca è asintoticamente cieca.

Algoritmi di Ricerca Euristica:

- Hill Climbing
- Best-First Search
 - Algoritmi Greedy
 - Algoritmo A*
 - Algoritmo generale: WA*
- Memory Bounded Search
 - Linear search (IDA*)
 - memory bounded A* (SMA*)
- Bidirectional Heuristic Search
 - Front-to-end (BHPA*)
 - Front-to-front (perimeter search: BIDA*)

Hill Climbing

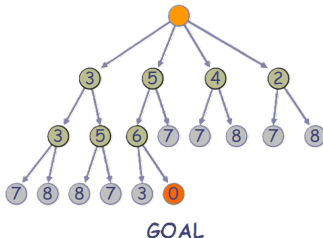
- Si usa unicamente la funzione euristica
- Non c'è backtracking
- Non si usa memoria
- Non è ottimale
- Non è completo
 - Minimi locali



“Greedy” Search

- Ricerca Best First che minimizza $f(n) = h(n)$
- Memorizza tutti i nodi frontiera

```
if ( goal_test(x0) == true ) return SUCCESS
else CODA.insert(x0 , h(x0))
do {
    /* if (CODA.isEmpty()) return FAILURE */
    nodo = CODA.remove(0)
    figli[] = SCS(nodo)
    CODA.insert(figli, h(figli))
} while ( goal_test(nodo) == false )
return SUCCESS
```



Proprietà:

- Non Ottimale
- Non Completo (senza l'Hash)
- complessità tempo e spazio $O(b^d)$

Miglioramenti per non ripetere gli stati:

- check sulla CODA dei nodi da espandere
- aggiunta di tabella HASH per nodi già espansi
- lista dei nodi espansi \mapsto CLOSED
- lista dei nodi da espandere \mapsto OPEN

Best-First Ottimale: A^* (Hart , Nilsson and Raphael, 1968)

- A^* = un nome ambizioso
- Funzione di valutazione $f(n) = g(n) + h(n)$
- Caratteristiche
 - Ottimale
 - Completo (anche senza l'Hash!!)
 - Complessità spaziale e temporale $O(b^d)$
 - Ottimamente efficiente

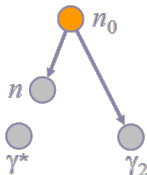
Best-First Ottimale: A^*

```
if ( goal_test(x0) == true ) return SUCCESS
else OPEN.insert(x0, g(x0)+h(x0))
do {
    if (OPEN.isEmpty()) return FAILURE
    nodo = OPEN.remove()
    CLOSED.insert(nodo)
    figli[] = SCS(nodo)
    for all figli{
        if ( (!OPEN.contains(figlio))
            /* se figlio è in OPEN e ha f() minore, la si aggiorna*/
            AND if (!CLOSED.contains(figlio)) )
            OPEN.insert(figlio, g(figlio)+h(figlio) )
    }
}while ( goal_test(nodo)== false )
return SUCCESS
```

A^* = algoritmo ottimale

Per ASSURDO:

A^* espande da OPEN γ_2 e γ_2 non è la soluzione ottima



- 1 per definizione $g(\gamma_2) > f^*$
- 2 sia $n \in \Pi^*$ nodo foglia (in OPEN)
- 3 se h è ammissibile allora $f(n) \leq f^*$
- 4 γ_2 viene preferito a n quindi $f(n) \geq f(\gamma_2)$
- 5 da 3 e 4 abbiamo che $f^* \geq f(\gamma_2)$
- 6 dato che γ_2 è finale allora $h(\gamma_2) = 0$ e $f(\gamma_2) = g(\gamma_2)$
- 7 da 5 e 6 abbiamo che $f^* \geq g(\gamma_2)$ che contraddice il punto 1

A^* = algoritmo completo

Per ASSURDO:

A^* ritorna un insuccesso o non termina.

- ① A^* ritorna un insuccesso se OPEN è vuota
- ② OPEN si svuota se nessuna foglia ha figlia
- ③ se esiste un Π tra n_0 e γ allora per ogni $n \in \Pi$ esiste un figlio
- ④ da 2 a 3 deriva che se esiste Π allora OPEN non si svuota e A^* non ritorna un insuccesso
- ⑤ se Π è di lunghezza finita allora A^* termina anche n grafi infiniti grazie all'uso di $g(n)$: perché $g(n) \leq \inf \forall n$
- ⑥ due condizioni per la completezza:
 - costo di un Π infinito = inf
 - Π^* non infinito

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

Robot Navigation

$h(n)$ = Distanza Manhattan

$g(n, SCS(n)) = 1$

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

8+3	7+4	6+3	5+6	4+7	3+8	2+9	3+10	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2+9	1+10	0+11	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Best-First Generale: WA^* (Ira Pohl, 1970)

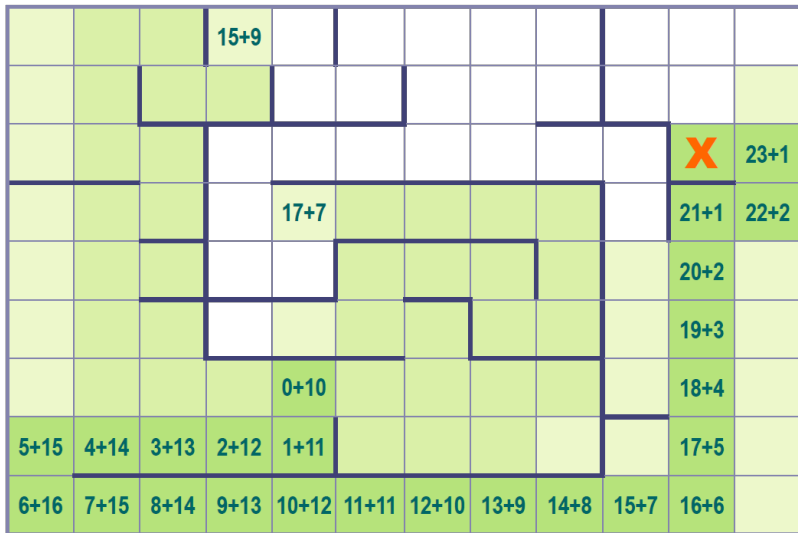
- Funzione di valutazione $\mapsto f(n) = (1 - w)g(n) + wh(n)$
 - $w = 0 \mapsto$ ricerca breadth-first
 - $w = 0.5 \mapsto$ ricerca A^*
 - $w = 1 \mapsto$ ricerca Greedy
 - Come cambia il costo della ricerca?
 - $w < 0.5$ non ha senso, quindi:
- Funzione di valutazione $f(n) = g(n) + wh(n)$
 - Crescendo w la ricerca diventa sempre più “greedy”
 - Il costo delle soluzioni è limitato superiormente da: wC^*

WA^* : alcuni risultati sul 15-puzzle

w	Mosse	Nodi
1	52,7	380×10^6
1,5	58,6	500×10^3
2	63,5	79×10^3
6	103,3	10500
99	145,3	7000

WA* Esempio: maze problem

$w = 1$



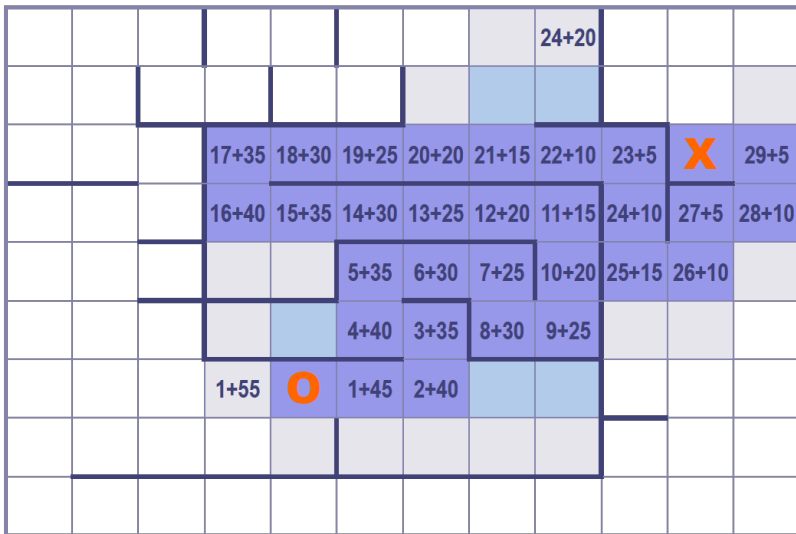
WA* Esempio: maze problem

$w = 2$

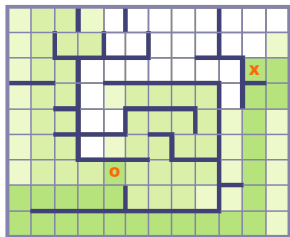
	11+22	12+20	15+18	16+16	19+14	20+12	21+10				
	10+20	13+18	14+16	17+14	18+12		22+8				
	9+18	8+16		24+12			23+6	24+4	25+2	X	31+2
	6+20	7+18							26+4	29+2	30+4
	5+22								27+6	28+4	
	4+24	3+22									
		2+24	1+22	O							
5+30											

WA* Esempio: maze problem

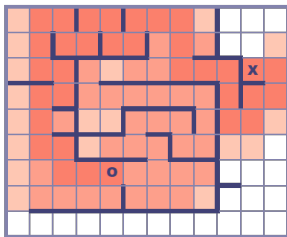
$w = 5$



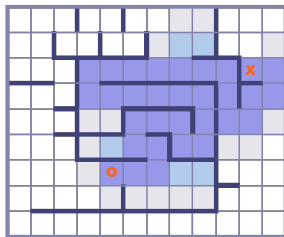
WA* Esempio: maze problem



$w = 1$
CLOSED=41
OPEN=18
 $d = 24$
 $wC^* = 24$



$w = 2$
CLOSED=66
OPEN=18
 $d = 32$
 $wC^* = 48$



$w = 5$
CLOSED=37
OPEN=16
 $d = 30$
 $wC^* = 120$

Iterative Deepening A^* (IDA^*) (Korf,1985)

- Una innovazione “attesa”
- 1985: prime soluzioni ottime del gioco del 15
- Eredita due qualità:
 - linear space search: $O(bd)$ da DFID
 - ottimalità di A^*
- E' completo: complessità temporale $O(b^d)$
- Come funziona
 - H a una soglia di costo: *threshold*
 - Funzione di valutazione $\mapsto f(n) = g(n) + h(n)$
 - Ha la LISTA di nodi LIFO
 - SE $f(n) \leq threshold$ si espande il nodo
 - SE la LISTA è vuota si ricomincia da capo la ricerca aggiornando *threshold*

```

if (goal_test(x0)== true) return SUCCESS
soglia = g(x0)+h(x0)
LISTA.insert(x0)
do {
    nodo = LISTA.remove()
    figli[] = SCS(nodo)
    for all figli{
        if (g(figlio)+h(figlio) <= soglia)
            LISTA.insert(figlio)
    }
}while( goal_test(nodo)== false and !LISTA.isEmpty())
if(goal_test(nodo)== true) return SUCCESS
else{
    update(soglia)
    /*La nuova soglia è data dal minore f(n) fuori
    soglia nella iterazione precedente*/
    GOTO 3
}

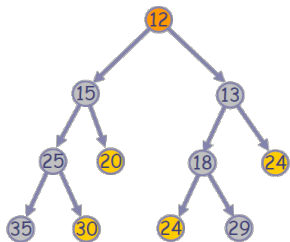
```

- Non è ottimamente efficiente:
 - Ripete i nodi delle iterazioni precedenti (incide poco, specie se b è grande)
 - Ripete nodi nella stessa iterazione
$$(d+1)1 + (d)b + (d-1)b^2 + \dots + (1)b^d = b^d \left(\frac{b}{b-1}\right)^2$$
- Funziona solo se:
 - costo degli operatori del problema hanno valore discreto e (quasi) costante
 - le valutazioni euristiche hanno valore discreto
 - altrimenti $O(b^{(2d)})$

- A*
 - A parità di $f(n)$: si deve preferire $\min(h(n))$
 - Implementare OPEN con Array di Hashtable: Si prelevano i nodi da $OPEN[\min f(n)]$
- IDA*
 - Sfruttare l'informazione delle iterazioni precedenti
 - Espansione nodi: ordinamento dei figli
 - Tenere in memoria una transposition table dei nodi già visitati per evitare ripetizioni
 - -40% di visite sul 15-puzzle

Simplified Memory-Bounded A^* (SMA^*) (Russel, '92)

- SMA^* è un A^* che adatta la ricerca alla quantità di memoria disponibile.
- Se la mem è esaurita si “dimenticano” dei nodi n della frontiera e più in alto nell'albero si fa backup del minimo $f(n)$ dei nodi dimenticati.
- Completata l'espansione di n si aggiorna $f(n)$ al minimo $f(.)$ osservato tra i nodi successori di n
- E' ottimale se c'è mem sufficiente per il cammino risolutivo ottimale
- Se c'è memoria per tutto l'albero SMA^* coincide con A^*
- Rispetto ad IDA^* : quando conviene e quando no?

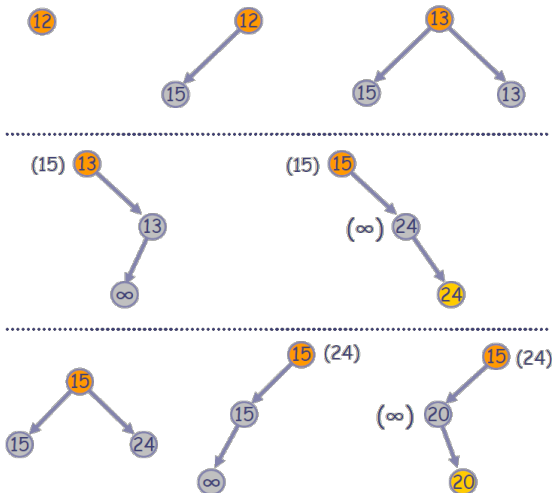


ES:

MAX num nodi = 3

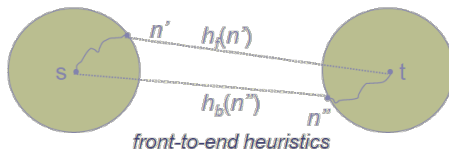
In giallo i nodi GOAL

I valori sono gli $f(n)$



BHPA* - Bi-directional Heuristic Path Algorithm (Pohl '71)

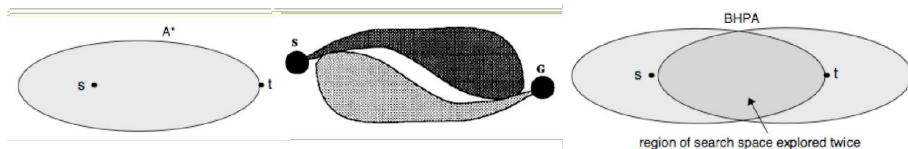
- Compie due ricerche A^* “simultanee” in direzioni opposte usando 2 euristiche: una forward (h_f) e una backward (h_b)



- Ad ogni espansione la direzione è scelta con il criterio della cardinalità:
`if |OPEN_f| < |OPEN_b| then dir=f else dir=b`
- L'algoritmo termina se la migliore soluzione trovata L_{min}
$$L_{min} \leq \max[\min_{n \in OPEN_f} f_f(n), \min_{n \in OPEN_b} f_b(n)]$$
- Questo vincolo di terminazione rende BHPA* poco efficiente!

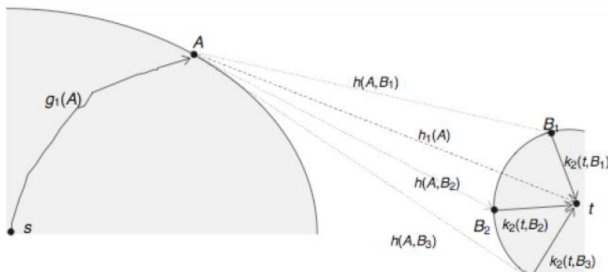
Il problema della ricerca front-to-end

- A^* è più efficiente di $BHPA^*$! Perché?
 - $\#(BHPA) \approx 2(\#A^*)$ e non $\#(BHPA) \approx \sqrt{\#(A^*)}$ come dovrebbe risultare dalla blind search theory.
- “Metafora del Missile”?
 - Le due ricerche si sorpassano senza toccarsi, come due missili?
 - Questa congettura (Pohl, Nilsson) è falsa!
- Le due ricerche vanno una attraverso l'altra. Il vero problema è la garanzia di ottimalità.



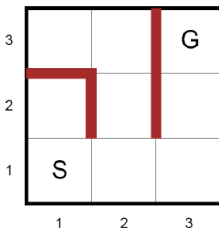
Front-to-front perimeter search ($BIDA^*$, Manzini '95)

- $BIDA^*$ si basa su due step distinti
 - una ricerca breadth-first visita tutti i nodi attorno al target e poi vengono memorizzati i nodi della frontiera (perimeter) con i loro valori di h^* value (k nella figura sotto)
 - ricerca IDA^* forward usando front-to-front evaluations calcolate come
 - $\max(h_1(A), \min_{i \in \text{frontier}} (h(A, B_i) + k(B_i)))$
 - Per maggior velocità i nodi “non necessari ” possono essere rimossi



- La perimeter search è efficiente solo in 2 casi
 - quando IDA^* è “feasible” (operatori a costo unitario)
 - quando la soluzione ottima non è molto profonda
 - Ottimi risultati con i puzzle, pessimi nel path finding

- La differenza rispetto alla offline search è che l'agente conosce $SCS(n)$ (e i suoi costi) solo quando SI TROVA in n .
Nell'esempio sotto l'agente (senza lookahead) scopre che l'operatore UP porta da (1,1) a (2,1) solo dopo averlo applicato!
- L'agente deve muoversi con 2 scopi:
 - esplorare lo spazio
 - arrivare allo stato finale
- Il costo è dato dalla somma degli operatori applicati in tutta l'esplorazione



- VALUTAZIONE ALGORITMI:

- Competitive Ratio (CR) = $g \text{ cammino percorso} / g \text{ cammino ottimo}$
(E' molto difficile riuscire a porre bound su CR!)

- PROBLEMI:

- DEAD END: se gli operatori sono irreversibili allora non è possibile garantire completezza (spazio non "safely explorable") \mapsto CR unbounded
- BLOCCO AVVERSARIO: (ambiente dinamico) un avversario potrebbe porci degli ostacoli in modo da rendere il cammino altamente inefficiente

- VANTAGGI:

- La ricerca online può attaccare problemi in un ambiente dinamico (non avverso!)
(es: aggirare ostacoli che si muovono)

- Hill Climbing:
dato che è una ricerca locale può essere usata come algoritmo online (ha il difetto di bloccarsi in un minimo locale)
- Online Depth First Search (Online DFS):
usa il principio della ricerca in profondità con backtracking. Se ad ogni stato tutte le azioni sono state provate: backtracking fisico allo stato predecessore!
 - Richiede di mappare lo spazio “visitato” in una tabella:
 $result[A, S_1] \leftarrow S_2$
 - Richiede di monitorare le zone dello spazio “da esplorare” con una tabella indicizzata per ogni stato $unexplore[S]$
 - Richiede di monitorare le zone dello spazio in cui poter fare backtracking con una tabella $unbacktracked[S]$

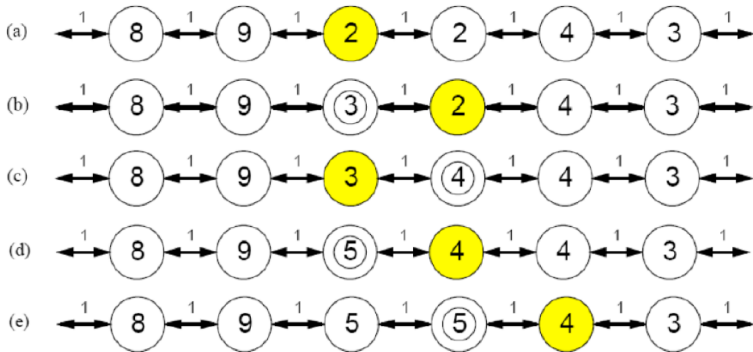
Online Depth First Search

```
ONLINE_DFS(s_a):  
    if (goal_test(s_a) == true ) return STOP  
    if (UNEXPLORED[s_a] == null ){  
        figli[] = SCS(s_a)  
        for all (figli != s_p) UNEXPLORED[s_a].add(figlio)  
    }  
    RESULT[a_p,s_p] = s_a  
    if (UNEXPLORED[s_p].is_empty() == false)  
        UNBACKTRACKED[s_a].add(s_p)  
    if (UNEXPLORED[s_a].is_empty() == true ){  
        if (UNBACKTRACKED[s_a].is_empty() == true ) return STOP  
        back_state = UNBACKTRACKED[s_a].pop()  
        for all (actions of s_a)  
            if (RESULT[ action,s_a ] == back_state)  
                new_action = action  
    }  
    else new_action = UNEXPLORED[s_a].pop()  
    s_p = s_a / a_p = new_action  
    return new_action
```

- *LRTA** (Learning Real-Time A^* , Korf '90):
 - E' dato da: Hill Climbing + Memoria + aggiornamento funzione euristica
 - Possiede una tabella $H[S]$ dei costi aggiornati per ogni stato.
Il criterio di aggiornamento è dato dal classico $f(n) = g(n) + h(n)$
- Funzionamento (l'agente entra in uno stato s_a)
 - 1 se $H[S_a]$ è null allora $H[S_a] \leftarrow h(S_a)$
 - 2 se s_a ha un predecessore allora
 - $result[a_p, s_p] \leftarrow s_a$
 - $H[s_p] \leftarrow \min_{a \in action[s_p], s = result[a, s_p]} \{g(s) + H[s]\}$ (update)
 - 3 scegli l'azione aa che porta allo stato s vicino con $\min\{g(s) + H[s]\}$
(se è null allora si prende $h(s)$, aggiorna $s_p \leftarrow s_a$ e ricomincia)

Spazio monodimensionale

I valori all'interno dei cerchi corrispondono a $H[S]$



- In spazi “sicuri” trova una soluzione in tempo finito
- Complessità = $O(n^2)$
- Non è completo in spazi infiniti (a differenza di A^*)
- E' possibile dimostrare che il valore di $H[s]$ converge ad $h^*(s)$ durante il cammino
- Se aumento il lookahead migliora il cammino, ma aumenta il costo computazionale per passo. Bisogna fare una scelta nel TRADEOFF: costo simulazione / costo esecuzione