La *ricerca esaustiva non è praticabile* in problemi di complessità esponenziale. Usiamo conoscenza del problema ed esperienza ed esperienza per riconoscere i cammini più promettenti: usiamo una stima del costo futuro ed evitiamo di generare gli altri.
La conoscenza euristica aiuta a fare scelte oculate: non evita la ricerca ma la riduce, consente in genere di trovare una buona soluzione in tempi accettabili e sotto certe condizioni garantisce completezza e ottimalità.

## Funzione di valutazione euristica

La conoscenza del problema è data da una funzione di valutazione $f$, che include una funzione $h$ detta funzione di valutazione euristica:

$$h : n \to \mathbb{R}$$

La funzione si applica al nodo ma dipende solo dallo stato ( `n.Stato` )

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

dove $g(n)$ è il costo del cammino visto con UC. Manteniamo la notazione in `n` per uniformità con $g$.

### Esempi di h

- Nell'Esempio del viaggio in romania, la città più vicina, o la città più vicina alla meta in linea d'area (tabella esterna)
- Il numero di caselle fuori posto nel gioco dell'otto
- Il vantaggio in pezzi nella dama o negli scacchi

## Algoritmi Best-First

Resta l'algoritmo UC, da ora in poi cambierà solo la funzione che assegna le probabilità ai nodi.
In questo caso $f$ (*stima di costo*) viene usata per la coda di priorità, questa determina la strategia di ricerca e ad ogni passo fa sì che si scelga il nodo sulla frontiera per cui il valore della $f$ è migliore.

> ⚠ Nota
>
> - In caso di un'euristica che stima la distanza della soluzione, "*migliore*" significa *minore*
> - Greedy best-first è un caso speciale: $f = h$, quindi si usa solo $h$

Esempio del viaggio in romania

## Algoritmo A

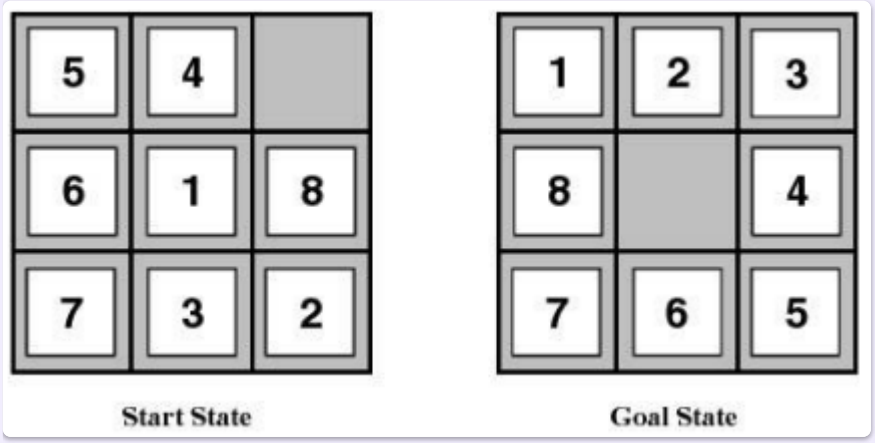
> Algoritmo A
>
> É un algoritmo Best First con una funzione di valutazione dello stato del tipo
>
> $$f(n) = g(n) + h(n) \quad \text{, con } h(n) \geq 0 \text{ e } h(goal) = 0$$
>
> Dove:
>
> - $g(n)$ è il costo del cammino percorso per raggiungere $n$
> - $h(n)$ è una stima del costo per raggiungere da $n$ un nodo $goal$

Vedremo casi particolari dell'algoritmo A:

- Se $h(n) = 0$ ($f(n) = g(n)$) si ha la *Ricerca Uniforme* (UC)
- Se $g(n) = 0$ ($f(n) = h(n)$) si ha il *Gready best-first*

Esempio di A per f=g+h

Esempio nel gioco dell'otto




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

$f(start = 0 + 7)$ Dopo $\leftarrow, \downarrow, \uparrow, \rightarrow$ $f = 4 + 7$

$f(\text{goal state}) = ? + 0$ stesso stato, $g$ è cambiata

## Completezza di A

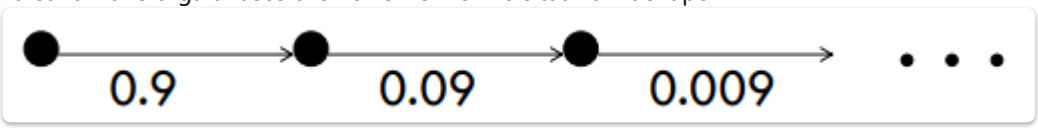
**Theorem**

L'algoritmo $A$ con la condizione

$$g(n) \geq d(n) \cdot \varepsilon \ (\varepsilon > 0 \text{ costo minimo arco})$$

è completo

La condizione ci garantisce che non si verifichino situazioni del tipo




e che il costo lungo un cammino non cresca "*abbastanza*", se cresce abbastanza possiamo fermare quel path per costo alto di $g$.

DIMOSTRAZIONE

Sia $[n_0 n_1 n_2 \dots n' \dots n_k = goal]$ un cammino soluzione, sia $n'$ un nodo della frontiera su un cammino soluzione: prima o poi $n'$ sarà espanso, infatti esistono solo un numero finito di nodi $x$ che possono essere aggiunti alla frontiera con $f(x) \leq f(n')$. Questo viene garantito dalla condizione sulla crescita di $g$ nel teorema, t.c. non esista una catena infinita di archi e nodi che possa aggiungere con costo sempre $\leq f(n')$.
Se non si trova una soluzione prima, $n'$ verrà espanso e i suoi successori verranno aggiunti alla frontiera. Tra questi anche il suo successore sul cammino soluzione. Il ragionamento si può ripetere fino a dimostrare che anche il nodo $goal$ sarà selezionato per l'espansione.

# Algoritmo A*

## La stima ideale

La funzione di valutazione ideale (*oracolo*) è

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

con:

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

- $h^*(n)$: costo del cammino minimo da $n$ a $goal$
- $f^*(n)$: costo del cammino minimo da $radice$ a $goal$, passando da $n$
  Normalmente vale $g(n) \geq g^*(n)$ (costo cammino $\geq$ cammino migliore) e $h(n)$ è una stima di $h^*(n)$, si può andarein sottostima (e.g. linea d'area) o in sovrastima della soluzione

## Definizione

> **Euristica Ammissibile**
>
> $\forall n | h(n) \leq h^*(n)$, $h$ è una sottostima

Ad esempio l'euristica della distanza in linea d'area

> **Algoritmo A***
>
> Un algoritmo A in cui $h$ è una funzione euristica ammissibile

> **Theorem**
>
> Gli algoritmi $A^*$ sono ottimali
> Corollario: $BF^{(+)}$ e UC sono ottimali ($h(n) = 0$)

Esempio del viaggio in romania

## Osservazioni

- Rispeto a Greedy Best-First, la componente $g$ fa si che abbandonino cammini che vanno troppo in profondità
- *Sottostime* e *Sovrastime*
  - Una *sottostima* ($h$) può farci compiere del lavoro inutile (tenendo di conto anche candidati non buoni), però non ci fa perdere il cammino migliore. Una funzione che
  - Una funzione che qualche volta *sovrastima* può farci perdere la soluzione ottimale. Viene fatto un taglio di cammini per sovrastima perché non penso che quel cammino sia promettente, ma potrei tagliare una possibile soluzione di costo minimo

## Ottimalità di A*

Nel casi di ricerca a/su albero, *l'uso di un'euristica ammissibile è sufficiente a garantire l'ottimalità* di $A^*$.
Nel caso di ricerca su grafo (con UC come visto) serve una *proprietà più forte*: la **consistenza** (o **monotonicità**):
Serve per evitare il rischio di scartare candidati ottimi se sono già stati esplorati, si vuole evitare di non considerare/far sparire al momento dell'espansione alcuni candidati ottimali.
Vorremmo assicurarci che il primo espanso sia sempre il migliore, così possiamo tenere la lista degli esplorati e non rischiamo di perdere l'ottimo.
In UC c'è l'`if` finale che non ci fa rischiare di perdere l'ottimo.

## Euristica Consistente (o monotona)

> **Euristica consistente**
>
> - $h(goal) = 0$
> - $\forall n | h(n) \leq c(n, a, n') + h(n')$ dove $n'$ è un successore di $n$, ne segue che $f(n) \leq f(n')$

> **Nota**

Se $h$ è consistente la $f$ non decresce mai lungo i cammini, da cui il termine monotona

É quindi una distanza che rispetta la disuguaglianza trianfolare.

### Proprietà

> **Theorem**
>
> Un'euristica monotona è ammissibile

Ne esistono anche di non ammissibili che sono monotone ma solo rare.

- Le euristiche monotone garantiscono che *la soluzione meno costosa venga trovata per prima* e quindi sono ottimali anche nel caso di ricerca sul grafo
- Non si devono recuperare fra gli antenati nodi con costo minore
- C'è una *lista degli esplorati*, se uno stato già esplorato è sul cammino ottimo, posso evitare di inserire il corrente ripetuto senza perdere l'ottimalità
  `if figlio.Stato` non è nella lista e non è in frontiera `then frontiera=Inserisci(figlio, frontiera)`
- Per la *frontiera*, volendo evitare stati ripetuti, resta l'`if` finale di UC
  `if figlio.Stato` è in frontiera con `Costo-cammino` più alto `then` sostituisci quel nodo frontiera col figlio

## Ottimalità di A* (con h consistente)

1. Se $h(n)$ è consistente, i valori di $f(n)$ lungo un cammino sono decrescenti:

$$\begin{aligned}
&\text{Se } h(n) \leq c(n,a,n') + h(n') && \text{def. consistenza}\\
&g(n) + h(n) \leq g(n) + c(n,a,n') + h(n) && \text{sommando } g(n)\\
&\text{ma siccome } g(n) + c(n,a,n') = g(n')\\
&g(n) + h(n) \leq g(n') + h(n')\\
&f(n) \leq f(n') && \rightarrow f \text{ monotona}
\end{aligned}$$

2. Ogni volta che $A^*$ seleziona un nodo ($n$) per essere espanso, il cammino ottimo a tale nodo è stato trovato: se così non fosse, ci sarebbe un altro nodo $m$ della frontiera sul cammino ottimo (a $n$, ancora da trovare con un cammino ottimo), con $f(m)$ minore, ma ciò non è possibile perché tale nodo sarebbe già stato espanso
3. Quando si seleziona il nodo goal il cammino è ottimo $[h = 0, f = C^*]$,

## Perché A* è vantaggioso

- $A^*$ espande tutti i nodi con $f(n) < C^*$ ($C^*$ = costo ottimo)
- $A^*$ espande alcuni nodi con $f(n) = C^*$
- $A^*$ *non espande alcun nodo con* $f(n) > C^*$
  Quindi alcuni nodi (e loro sottoalberi) non verranno considerati per l'espansione, ma resteranno ottimali.
  Puring: scegliamo un'$h$ opportuna, più alta possibile tra le ammissibili, che ci fa tagliare molto.
  Più è alta $h$ più $f(n)$ deborda e più sono i nodi che non espandiamo perché superano $C^*$, però $h$ *deve essere ammissibile*.

> **Nota**
>
> Più $f$ è aderente alla stima ottimale, più si taglia (gli ovali sono più stretti). Cercheremo quindi una $h$ il più alta possibile tra le ammissibili. Se la troviamo molto bassa molti nodi resteranno minori di $C^*$, quindi li espandiamo tutti.
> Il puring dei sotto-alberi è il punto focale: non li abbiamo già in memoria e evitiamo di generarli.

## Riassunto A*

- L'algoritmo è quello degli schemi UC

- Si usa $f = g + h$ per la *coda con priorità*
  - $h$ e $g$ soddisfano il teorema di completezza
  - $h$ è una funzione euristica ammissibile

## Bilancio su A*

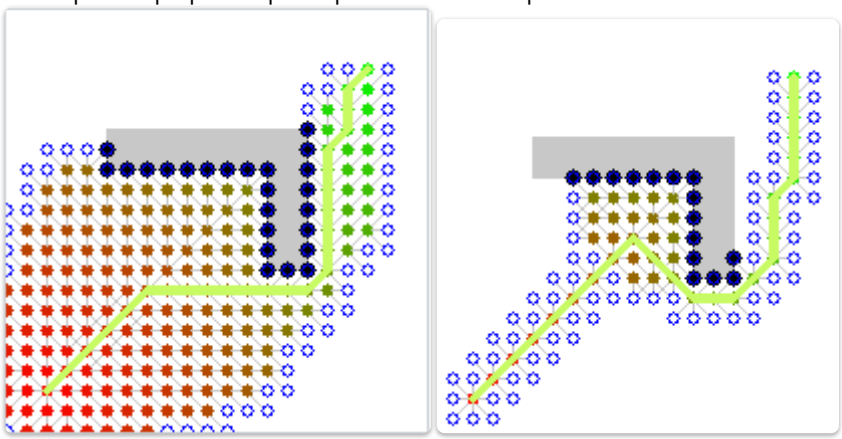
- $A^*$ è **completo**: discende dalla completezza di $A$ ($A^*$ è un algoritmo $A$ particolare)
- $A^*$ con euristica monotona è **ottimale**
- $A^*$ è **ottimamente efficiente**: a parità di euristica nessun altro algoritmo espande meno nodi (senza rinunciare a ottialità)
  **Problemi:** l'*occupazione di memoria* nel caso pessimo resta $O(b^{d+1})$, a causa della frontiera

## Sottocasi speciali di A

- Se $h(n) = 0$ $[f(n) = g(n)]$ si ha UC - $g$ non basta (si può migliorare)
- Se $g(n) = 0$ $[f(n) = h(n)]$ si ha Greedy best-first - $h$ non basta (già visto all'inizio)

### Dijkstra - UC vs. A^*

Entrambi gli algoritmi si scontrano con il muro, ma *$A^*$ con l'euristica* va direttamente verso il percorso ottimo. Dikstra ci mette più tempo perché prova più strade contemporaneamente.



## Costruire le euristiche di A*

### Valutazione di funzioni euristiche

A parità di ammissibilità, un'euristica può essere più efficiente di un'altra nel trovare il cammino soluzione migliore (quindi visita meno nodi). Questo dipende da *quanto informata è l'euristica* (dal grado di informazione posseduto)

$$\begin{array}{ll} h(n) = 0 & \text{minimo di informazione (BF o UC)} \\ h^*(n) & \text{massimo di informazione (oracolo)} \end{array}$$

In generale, per le euristiche ammissibili

$$0 \leq h(n) \leq h^*(n)$$

### Più inforamta, più efficiente

> **Theorem**
>
> Se $h_1 \leq h_2$, i nodi espansi da $A^*$ con $h_2$ sono un sottoinsieme di quelli espansi da $A^*$ con $h_1$.
> $A^*$ espande tutti i nodi con $f(n) = g(n) + h(n) < C^*$, e sono meno per un'$h$ maggiore
> Se $h_1 \leq h_2$, $A^*$ con $h_2$ è almeno efficiente quanto $A^*$ con $h_1$

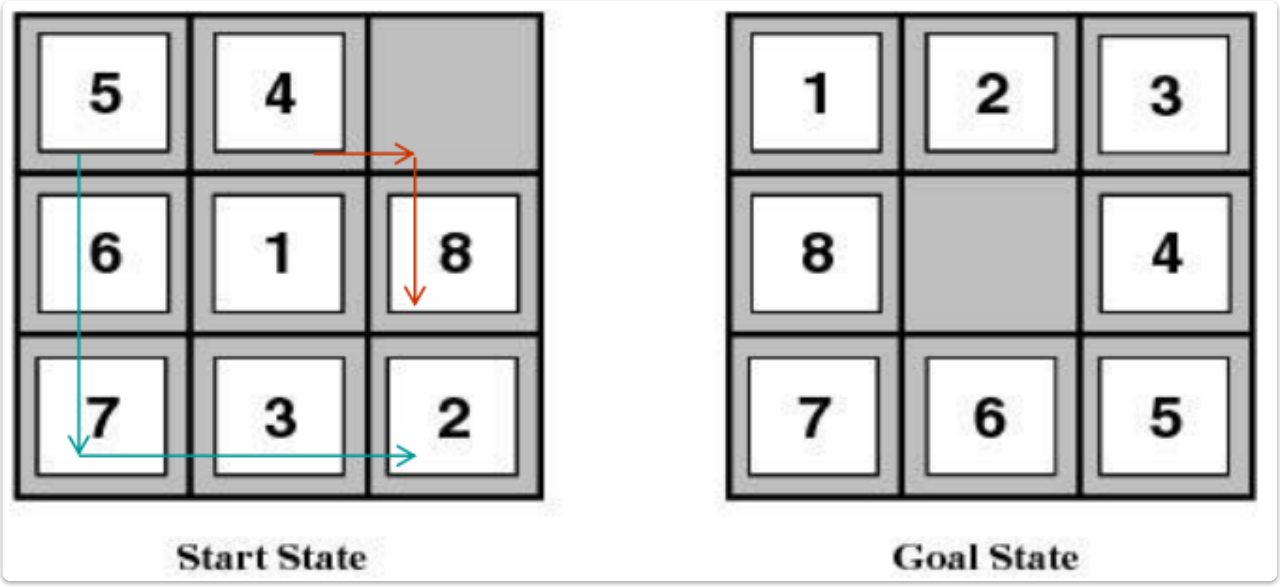
Un'euristica più informata (accurata) riduce lo spazio di ricerca, ma è tipicamente più costosa da calcolare.

### Confronto di euristiche ammissibili

Due possibili euristiche ammissibili per il gioco dell'8:

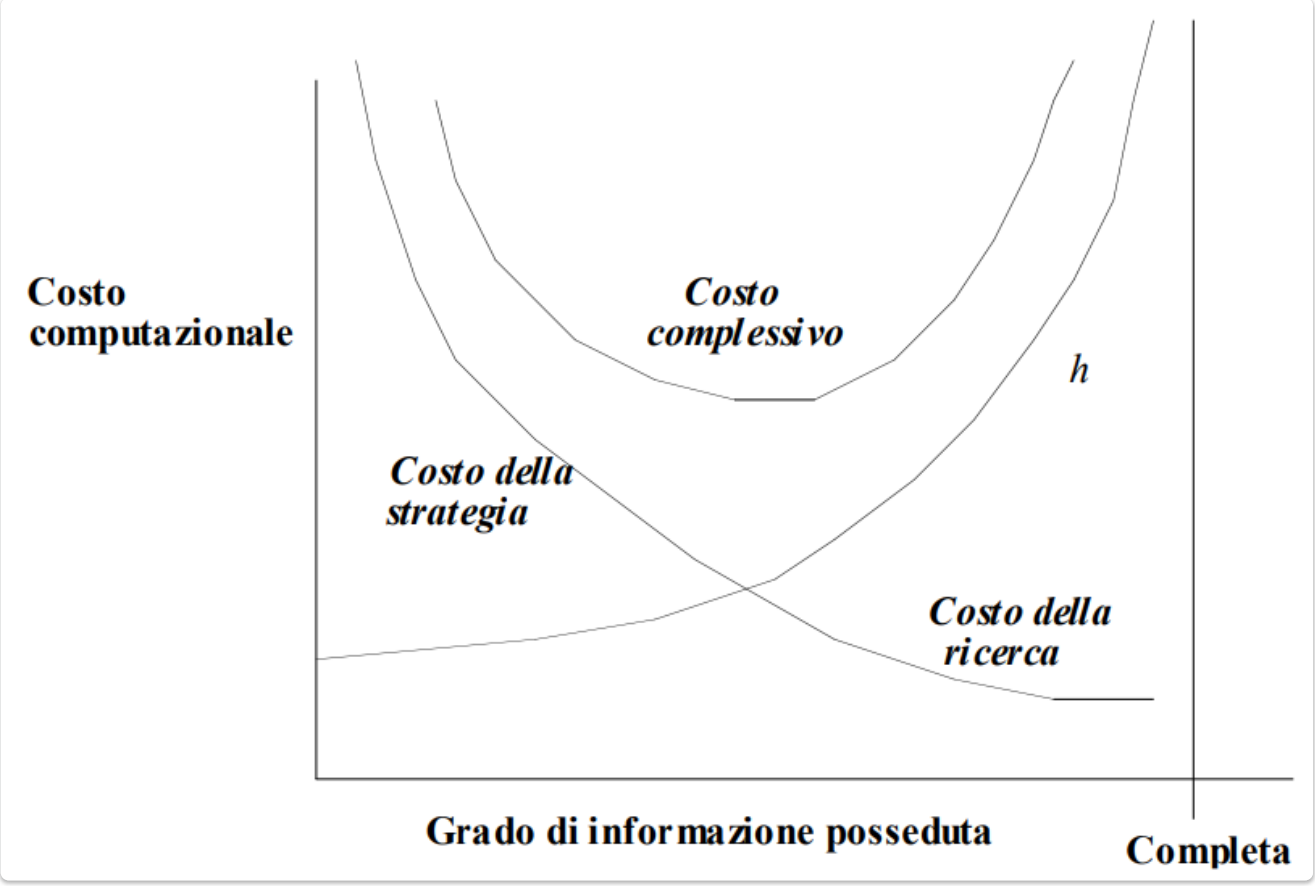
- $h_1$: conta il numero delle caselle fuori posto
- $h_2$: somma delle distanze Manhattan delle caselle fuori posto

$h_2$ è più informata di $h_1$, infatti $\forall n | h_1(n) \leq h_2(n)$. Diciamo che $h_2$ denomina $h_1$




$h_1 = 7$
$h_2 = 4 + 2 + 2 + 2 + 2 + 0 + 3 + 3 = 18$

## Costo ricerca vs. Costo euristica




## Misura del potere euristico

Fattore di diramazione effettivo: b*

$N$: numero di nodi generati

$d$: profondità della soluzione

b* è il fattore di diramazione di un albero uniforme con $N + 1$ nodi, e risolve l'equazione

$$N + 1 = b^* + (b^*)^2 + \cdots + (b^*)^d$$

Sperimentalmente una buona euristica ha un $b^*$ abbastanza vicino a 1 ($\leq 1.5$)

> **Example**
>
> $d = 5\ N = 52$
> $\mathrm{b}^* = 1.92$

> **Example: gioco dell'otto**

| $d$ | ID | $A^*(h_1)$ | $A^*(h_2)$ |
|---|---|---|---|
| 2 | $10(2.43)$ | $6(1.79)$ | $6(1.79)$ |
| 4 | $112(2.87)$ | $13(1.48)$ | $12(1.45)$ |
| 6 | $680(2.73)$ | $29(1.34)$ | $18(1.30)$ |
| 8 | $6384(2.80)$ | $39(1.33)$ | $25(1.24)$ |
| 10 | $47127(2.79)$ | $93(1.38)$ | $39(1.22)$ |
| 12 | $3644035(2.78)$ | $227(1.42)$ | $73(1.24)$ |
| 14 | Nodi generati: $\mathrm{b}^*$ | $539(1.44)$ | $113(1.23)$ |
| ... | - | ... | ... |

Sono riportati i nodi generati e il fattore di diramazione effettivo ($\mathrm{b}^*$, pink). I dati sono mediati, per $d$, su 100 istanze del problema.

### Capacità di esplorazione

Migliorando di poco l'euristica si riesce, a partità di nodi espansi, a raggiungere una *profondità doppia* di esplorazione mosse, quindi:

- tutti i problemi di IA (o quasi) sono di complessità esponenziale, ma ce ne sono di vari ordini di grandezza
- L'euristica può migliorare di molto la capacità di esplorazione dello spazio degli stati rispetto alla ricerca cieca
- Migliorando anche di poco l'euristica si riesce ad esplorare uno spazio molto più grande (più in profondità)

## Inventare un'euristica

Alcune strategia per ottenere euristiche ammissibili sono:

- Rilassamento del problema
- Massimizzazione di euristiche
- Euristiche da sottoproblemi
- Database di pattern disgiunti
- Combinazione lineare
- Apprendere dall'esperienza

### Rilassamento del problema

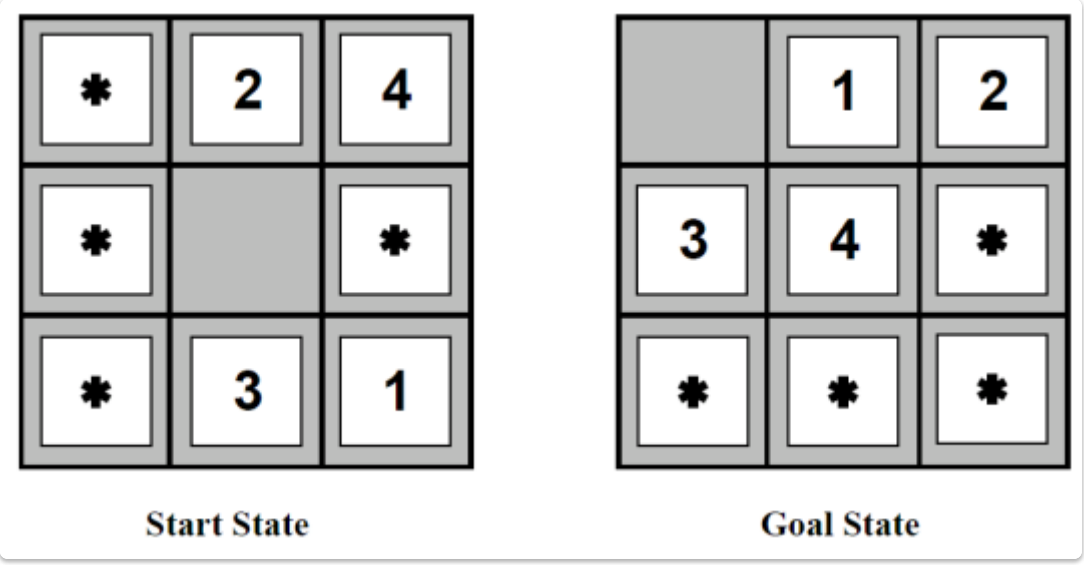
Nel gioco dell'otto, è possibile muovere una tessera da A a B se B è adiacente ad A e se B è libera
$h_1$ e $h_2$ sono *calcoli della distanza esatta della soluzione* in versioni semplificate del puzzle:

- $h_1$: sono sempre ammessi scambia piacimenti tra le caselle
  Si eliminano entrambe le restrizioni, il costo di soluzione (numero di mosse) è il numero delle caselle fuori posto
- $h_2$: sono ammessi spostamenti anche su caselle occupate, purché adiacenti
  Si elimina solo una delle due restrizioni, il costo di soluzione è la somma delle distanze di Manhattan

### Massimizzazione di euristiche

Se si hanno una serie di euristiche ammissibili $h_1, h_2, \ldots, h_k$ *senza che nessuna "domini" un'altra*, allora conviene prendere il massimo dei loro valori $\Rightarrow h(n) = max\{h_1(n), h_2(n), \ldots, h_k(n)\}$.
Se le $h_i$ sono ammissibili, allora anche $h$, che domina tutte le altre, lo è.

## Euristiche da sottoproblemi




Il costo della soluzione ottima al sottoproblema (ordinare 1,2,3,4) è una sottostima del costo per il problema nel suo complesso
Databese di pattern: memorizzare ogni istanza del sottoproblema con il relativo costo della soluzione.
Si può questo database per calcolare $h_{DB}$, estraendo dal database la configurazione corrispondente allo stato completo corrente

SOTTOPROBLEMI MULTIPLI

Potremmo fare la cosa per gli altri sottoproblemi (5-6-7-8, 2-4-6-8, ...) ottenendo altre euristiche ammissibili, per poi prendere il valore massimo, anch'esso ammissibile. Si ouò sommarle ed ottenere un'euristica ancora più accurata?

## Database di pattern disgiunti

In generale non si possono sommare euristiche di sottoproblemi per ottenerne una più accurata, perché le soluzione ai sottoproblemi interferiscono e *la somma delle euristiche in generale non è ammissibile* (potremmo sovrastimare avendo avuto aiuti mutui).
Si deve eliminare il costo delle mosse che contribuiscono all'altro sottoproblema. I database di pattern disgiunti consentono di sommare costi (euristiche additive).

## Combinazione lineare

Quando diverse caratteristiche influenzano la bontà di uno stato, si può usare una combinazione lineare:

$$h(n) = c_1 x_1(n) + c_2 x_2(n) + \cdots + c_k x_k(n)$$

> **Example**
>
> gioco dell'8: $h(n) = c_1 \# fuori - posto + c_2 \# coppie - scambiate$
> scacchi: $h(n) = c_1 \cdot vant - pezzi + c_2 \cdot pezzi - attac. + c_3 \cdot regina + \ldots$

Il peso dei coefficienti può essere aggiustato con l'esperienza, anche qui apprendendo automaticamente da esempi di gioco.
$h(goal) = 0$, ma questo non rende automatiche l'ammissibilità e la consistenza.

## Apprendere dall'esperienza

Si fa girare il programma raccogliendo come dati delle coppie $< stato, h^* >$, questi dati vengono usati per apprendere a predire la $h$ con *algoritmi di apprendimento induttivo* (da istanze note stimiamo $h$ in generale).

Questi algoritmi di apprendimento si concentrano su caratteristiche salienti dello stato $(feature, x_i)$

# Algoritmi evoluti basati su A* - Migliorare l'occupazione di memoria

## Beam search

Nel Best First viene tenuta tutta la frontiera, se *l'occupazione di memoria è eccessiva* si può ricorrere a questa variante.
La **beam search** *tiene ad ogni passo solo i $k$ nodi più promettenti*, dove $k$ è detto *l'ampiezza del raggio* (beam).
La beam search **non è completa**.

## IDA - *A* con approfondimento iterativo

Si combina $A^*$ con ID: ad ogni iterazione ricerca *in profondità* con un limite (*cut off*) dato dal valore della *funzione $f$* (e non dalla profondità).
Il limite $f - limit$ viene aumentato ad ogni iterazione, fino a trovare la soluzione.
Il **punto critico** è di quanto viene aumentato $f - limit$
É cruciale la scelta dell'incremento per garantirne **l'ottimalità**:

- *costo delle azioni fisso*: il limite viene incrementato del costo delle azioni
- *costi delle azioni variabili*: si potrebbe ad ogni passo fissare il limite successivo al valore minimo delle $f$ scartate all'iterazione precedente, in quanto superavano il limite.

### Analisi di IDA*

É **completo** e **ottimale**:

- Se le azioni hanno *costo costante $k$* (caso tipico 1) e $f$-limit viene incrementato di $k$
- Se le azioni hanno *costo variabile* e l'incremento di $f$-limit è $\leq \varepsilon$ (minimo costo degli archi)
- Se il nuovo $f$-limit$=$ $\text{minimo valore } f \text{ dei nodi generati ed esclusi all'iterazione precedente}$
  L'**occupazione in memoria** è di $\mathbf{O(bd)}$ (da DF).

## Best-First ricorsivo (RBFS)

É simile al DF ricorsivo: cerca di usare meno memoria facendo del lavoro in più.
Tiene traccia ad ogni livello del *migliore percorso alternativo*. Invece di fare backtraking in caso di fallimento interrompe l'esplorazione quando trova un nodo promettente (secondo $f$).
Nel tornare indietro si ricorda il miglior nodo che ha trovato nel sottoalbero esplorato, per poterci eventualmente tornare.
La *memoria* è lineare nella profondità della soluzione ottima.
Esempio del viaggio in romania

### Algoritmo

```
function Ricerca-Best-First-Ricorsiva(problema)
        returns soluzione oppure fallimento
    // all'inizio f-limite è un valore primo molto grande
    return RBFS(problema, CreaNodo(problema.Stato-iniziale), infty)

function RBFS (problema, nodo, f-limite)
        returns soluzione oppure falliento e un nuovo limite all'f-costo
    if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
    successori = []
    for each azione in problema.Azioni(nodo.Stato) do
        //genera i successori
        aggiungi Nodo-Figlio(problema, nodo, azione) a successori
    if successori è vuoto then return fallimento, infty
    for each s in successori do
        // valuta i successori
        s.f=max(s.g + s.h, nodo.f) // modo per rendere monotona f
    loop do
        migliore = il nodo con f minimo tra i successori
        if migliore.f > f_limite then return fallimento, migliore.f
        alternativa = nodo con f minimo tra i successori
```

```
        risultato, migliore.f = RBFS (problema, migliore.f, min(f_limite, alternativa))
        if risultato != fallimento then return risultato
```

## A* con memoria limitata - Versione semplice

L'idea è quella di *utilizzare al meglio la memoria diponibile*: $\mathbf{SMA}^*$ procede come $A^*$ fino ad esaurimento della memoria disponibile. A questo punto *"dimentica" il nodo peggiore*, dopo avere aggiornato il valore del padre.
A parità di $f$ si sceglie il nodo miglio più recente e si dimentica il peggore più vecchio.
É **ottimale** se *il cammino soluzione sta in memoria*.

In algoritmi a memoria limitata ($IDA^*$ e $SMA^*$) le limitazioni della memoria possono portare a compiere molto lavoro inutile. É difficiel stimare la complessità temporale effettiva. Le limitazioni di memoria possono rendere un problema intrattabile dal punto di vista computazionale.