

# Intelligenza Artificiale

UniVR - Dipartimento di Informatica

**Fabio Irimie**

1° Semestre 2025/2026

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Tipi di intelligenza artificiale . . . . .	2
1.1.1	Autonomous agents . . . . .	2
1.1.2	Data analysis . . . . .	2
1.1.3	Machine Learning . . . . .	2
1.1.4	Time series analysis . . . . .	3
1.1.5	Intelligent Agents . . . . .	3
1.2	Markov Decision Process (MDP) . . . . .	3
1.3	Generative AI . . . . .	4
<b>2</b>	<b>Agenti e ambiente</b>	<b>4</b>
2.1	Razionalità . . . . .	5
2.2	PEAS . . . . .	6
2.3	Tipi di ambienti . . . . .	7
2.4	Agenti di problem solving . . . . .	7
<b>3</b>	<b>Ricerca nello spazio degli stati</b>	<b>9</b>
3.1	Ricerca generale . . . . .	9
3.1.1	Tree search . . . . .	9
3.1.2	Stato e nodo . . . . .	9
3.1.3	Tree search generale . . . . .	9
3.1.4	Stati ripetuti . . . . .	10
3.2	Ricerca non informata . . . . .	10
3.2.1	Breadth-first search . . . . .	11
3.2.2	Uniform-cost search . . . . .	11
3.2.3	Depth-first search . . . . .	12
3.2.4	Iterative deepening search . . . . .	12
3.3	Ricerca informata . . . . .	14
3.3.1	Best-first search . . . . .	14
3.3.2	Greedy best-first search . . . . .	14
3.3.3	A* search . . . . .	15
3.3.4	Consistenza e ammissibilità . . . . .	15
3.3.5	Euristiche . . . . .	16
3.4	Ricerca locale . . . . .	17
3.4.1	Hill climbing . . . . .	18
3.4.2	Simulated annealing . . . . .	19
3.4.3	Local beam search . . . . .	20
3.4.4	Algoritmi genetici . . . . .	20

# 1 Introduzione

Nel 1950 Alan Turing pubblica un articolo intitolato "Computing Machinery and Intelligence" in cui propone un esperimento per determinare se una macchina può essere considerata intelligente. L'esperimento, noto come "test di Turing", coinvolge un interrogatore umano che comunica con due entità nascoste: una macchina e un essere umano. L'interrogatore deve fare domande a entrambe le entità e, basandosi sulle risposte, deve determinare quale delle due è la macchina. Se l'interrogatore non riesce a distinguere tra le risposte della macchina e quelle dell'essere umano, la macchina è considerata intelligente.

In futuro l'attenzione si è spostata sulla ricerca di metodi per risolvere problemi che richiedono intelligenza umana, utilizzando algoritmi e modelli matematici fino ad arrivare alle reti neurali e intelligenza artificiale.

**Definizione 1.1.** L'intelligenza artificiale è una disciplina che studia come **simulare** l'intelligenza umana in scenari complessi

## 1.1 Tipi di intelligenza artificiale

### 1.1.1 Autonomous agents

Sono sistemi che percepiscono l'ambiente e agiscono in modo autonomo per raggiungere obiettivi specifici.

### 1.1.2 Data analysis

Utilizzo di algoritmi per analizzare grandi quantità di dati e estrarre informazioni utili e correlazioni complesse.

### 1.1.3 Machine Learning

È lo sviluppo di algoritmi che permettono a dei modelli di apprendere dai dati di esempio e migliorare le loro prestazioni nel tempo senza essere esplicitamente programmati. Ad esempio riconoscimento di immagini.

L'apprendimento automatico è diviso in tre categorie principali:

- **Unsupervised learning:** il modello viene addestrato su un insieme di dati non etichettati, dove l'obiettivo è scoprire strutture nascoste o pattern nei dati senza avere risposte corrette predefinite.
- **Supervised learning:** il modello viene addestrato su un insieme di dati etichettati, dove ogni esempio di input è associato a una risposta corretta. L'obiettivo è che il modello impari a mappare gli input alle risposte corrette.
- **Reinforced learning:** il modello impara attraverso interazioni con l'ambiente, ricevendo ricompense o penalità in base alle azioni intraprese. L'obiettivo è massimizzare la ricompensa totale nel tempo.

#### 1.1.4 Time series analysis

L'analisi delle serie temporali è un'area dell'apprendimento automatico che si concentra sull'analisi di dati collezionati nel tempo. Le serie temporali sono sequenze di dati misurati a intervalli regolari, come temperatura giornaliera, prezzi delle azioni o dati di vendita mensili. L'obiettivo dell'analisi delle serie temporali è identificare pattern, tendenze e stagionalità nei dati per fare previsioni future.

Gli approcci comuni per l'analisi delle serie temporali includono:

- **Riconoscimento di anomalie e cause:** è un processo di identificazione di dati o eventi che si discostano significativamente dal comportamento normale o atteso. Queste anomalie possono indicare problemi, errori o situazioni insolite che richiedono attenzione.
- **Generative transformers:** sono una classe di modelli che permettono di predire il prossimo elemento in una sequenza di dati partendo dagli elementi precedenti, come ad esempio la parola successiva in una frase o il pixel successivo in un'immagine. Si sfrutta il concetto di **attenzione** per pesare l'importanza relativa delle diverse parti della sequenza di input durante la generazione dell'output.

#### 1.1.5 Intelligent Agents

Un agente intelligente è un sistema che percepisce l'ambiente circostante attraverso sensori e agisce su l'ambiente per raggiungere un obiettivo specifico. Gli elementi chiave di un agente intelligente includono:

- **Performance measure:** misura il successo dell'agente nel raggiungere i suoi obiettivi
- **Rationality:** l'agente deve agire in modo da massimizzare la sua performance measure attesa

### 1.2 Markov Decision Process (MDP)

Un MDP è un modello matematico utilizzato per rappresentare problemi di decisione sequenziali. Gli elementi principali sono:

- **State:** rappresenta l'ambiente in un dato momento
- **Actions:** insieme delle azioni che l'agente può intraprendere
- **Transition model:** effetto che le azioni hanno sull'ambiente (potrebbero essere parzialmente incognite)

$$T : (\text{state}, \text{action}) \rightarrow \text{next\_state}$$

- **Reward:** valore **immediato** dell'esecuzione di un'azione

$$R : (\text{state}, \text{action}, \text{next\_state}) \rightarrow \text{real\_number}$$

- **Policy:** strategia che l'agente utilizza per decidere quale azione intraprendere in ogni stato con l'obiettivo di massimizzare la ricompensa totale attesa nel tempo

$$\pi : (\text{state}) \rightarrow \text{action}$$

### 1.3 Generative AI

L'intelligenza artificiale generativa si riferisce a una classe di modelli di intelligenza artificiale che sono in grado di generare nuovi contenuti, come testo, immagini, musica o video, a partire da dati di addestramento. Questi modelli hanno miliardi di parametri e sono **preaddestrati** su grandi quantità di dati. In sostanza questi modelli "predicono il futuro" basandosi sui dati su cui sono stati addestrati e un **prompt** (input dell'utente).

## 2 Agenti e ambiente

Gli agenti includono umani, robot, softbot, termostati ecc... La funzione dell'agente mappa lo storico di percezioni in azioni:

$$f : \mathcal{P}^* \mapsto \mathcal{A}$$

Il programma dell'agente è eseguito su architettura fisica per produrre la funzione  $f$ .

**Esempio 2.1.** Un esempio potrebbe essere un insieme di stanze  $\{A, B\}$  e un robot aspirapolvere che può percepire la sua posizione e il contenuto della stanza. L'agente potrebbe quindi percepire  $[A, \text{Sporco}]$  se ci fosse dello sporco nella stanza A. Le azioni potrebbero essere di movimento o pulizia. Tutto questo dipende dalla sequenza di percezioni, ad esempio in una tabella:

Percezione	Azione
$[A, \text{Pulito}]$	Vai a B
$[A, \text{Sporco}]$	Pulisci
$[B, \text{Pulito}]$	Vai ad A
$[B, \text{Sporco}]$	Pulisci
$[A, \text{Pulito}], [A, \text{Pulito}]$	Vai a B
$[A, \text{Pulito}], [A, \text{Sporco}]$	Pulisci

Tabella 1: Esempio di tabella di percezioni e azioni

Non possiamo dire se questa è una funzione corretta perchè non abbiamo una **performance measure** che ci dica se l'agente sta facendo un buon lavoro.

**Definizione 2.1.** Se un agente ha  $|\mathcal{P}|$  possibili percezioni, allora al tempo  $T$  avrà:

$$\sum_{t=1}^T |\mathcal{P}|^t$$

Se lo storico di percezioni è irrilevante, cioè se ad ogni percezione è associata un'azione la funzione viene chiamata **Reflex**.

## 2.1 Razionalità

Per definire l'intelligenza di un agente si utilizza una misura di performance che valuta la sequenza di percezioni.

**Esempio 2.2.** Tornando all'esempio del robot aspirapolvere si potrebbero assegnare i seguenti punteggi:

- Un punto per ogni stanza pulita per ogni unità di tempo
- Meno un punto per ogni mossa
- Penalizzazione per ogni stanza sporca

**Esempio 2.3.** Un altro esempio è il seguente ambiente:

- Ci sono 3 stanze (A, B, C) e due robot ( $r_1, r_2$ )
- $r_1$  può sorvegliare solo A e B e  $r_2$  solo B e C
- $r_1$  inizia dalla stanza A e  $r_2$  dalla C
- Il tempo di percorrenza tra le stanze è 0
- Performance measure: minimizza il tempo in cui una stanza non è sorvegliata, cioè il tempo totale in cui una stanza non è visitata da nessun robot

Un possibile comportamento razionale potrebbe essere il seguente (alternata):

Stato	A	B	C	Tempo
[A, C]	0	1	0	1
[B, C]	1	0	0	2
[A, C]	0	1	0	3
[A, B]	0	0	1	4
Average idleness	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{4}$	Tot: $\frac{1}{3}$

Un altro comportamento potrebbe essere (fissata):

Stato	A	B	C	Tempo
[A, C]	0	1	0	1
[B, C]	1	0	0	2
[A, C]	0	1	0	3
[B, C]	1	0	0	4
Average idleness	$\frac{1}{2}$	$\frac{1}{2}$	0	Tot: $\frac{1}{3}$

Entrambi i comportamenti hanno la stessa performance measure, ma il primo è migliore del secondo perchè penalizza meno una singola stanza rispetto alle

altre. Per capirlo bisogna non solo minimizzare la performance measure, ma anche minimizzare la varianza.

## 2.2 PEAS

Per progettare un agente intelligente bisogna definire l'ambiente in cui opera:

- **Performance measure:** come viene valutato il successo dell'agente
- **Environment:** il contesto in cui l'agente opera
- **Actuators:** i mezzi attraverso cui l'agente agisce sull'ambiente
- **Sensors:** i mezzi attraverso cui l'agente percepisce l'ambiente

**Esempio 2.4.** Prendiamo ad esempio un taxi automatico, il PEAS potrebbe essere:

- Performance measure:
  - Soddisfazione del cliente
  - Sicurezza
  - Efficienza del carburante
  - Rispetto delle leggi stradali
- Environment:
  - Traffico stradale
  - Condizioni meteorologiche
  - Segnali stradali
  - Pedoni e altri veicoli
- Actuators:
  - Volante
  - Acceleratore
  - Freni
  - Indicatori di direzione
- Sensors:
  - Telecamere
  - Lidar
  - Radar
  - Sensori di velocità
  - GPS

## 2.3 Tipi di ambienti

Gli ambienti possono essere classificati in base a diverse caratteristiche:

- **Osservabile:** se l'agente può percepire completamente lo stato dell'ambiente in ogni momento
- **Deterministico:** se l'azione dell'agente determina in modo univoco il prossimo stato dell'ambiente
- **Episodico:** se l'esperienza dell'agente è divisa in episodi indipendenti, cioè l'azione in un episodio non influisce sugli episodi successivi
- **Statico:** se l'ambiente non cambia mentre l'agente sta prendendo una decisione
- **Discreto:** se l'insieme di stati, azioni e percezioni è finito o numerabile
- **Singolo agente:** se l'agente opera da solo nell'ambiente senza la presenza di altri agenti

**Esempio 2.5.** Prendiamo ad esempio i seguenti ambienti provando a classificarli:

	Crossword	Robo-selector	Poker	Taxi
Osservabile	Sì	Parziale	Parziale	Parziale
Deterministico	Sì	No	No	No
Episodico	No	Sì	No	No
Statico	Sì	No	Sì	No
Discreto	Sì	No	Sì	No
Singolo agente	Sì	Sì	No	No

Il tipo di ambiente cambia radicalmente la soluzione del problema:

- **Deterministico, completamente osservabile:** Single-state problem
- **Completamente non osservabile:** Conformant problem, l'agente non sa in che stato si trova, ma potrebbe trovare una soluzione
- **Non deterministico e/o parzialmente osservabile:** Contingency problem, l'agente deve prevedere le possibili situazioni future e agire di conseguenza
- **Spazio degli stati sconosciuto:** Exploration problem, l'agente deve esplorare l'ambiente per scoprire gli stati e le azioni disponibili

## 2.4 Agenti di problem solving

È una forma ristretta di agenti che formulato un problema e un obiettivo partendo da uno stato cerca una soluzione ignorando le percezioni, siccome ci si trova in un single-state problem. Questo si chiama Offline problem solving perchè l'agente ha completa conoscenza dell'ambiente. Online problem solving è quando l'agente non ha completa conoscenza dell'ambiente.



**Esempio 2.6.** Il seguente è un esempio di problem solving agent:

```
1 function Simple-Problem-Solving-Agent(percept) returns action
2   static: seq, an action sequence, initially empty
3           state, some description of the current world state
4           goal, a goal, initially null
5           problem, a problem formulation
6
7   state <- Update-State(state, percept)
8
9   if seq is empty then
10     goal <- Formulate-Goal(state)
11     problem <- Formulate-Problem(state, goal)
12     seq <- Search( problem)
13
14   action <- First(seq)
15   seq <- Rest(seq)
16   return action
17
```

**Esempio 2.7.** Consideriamo il problema "Vacanze in Romania". Bisogna formulare un viaggio da Arad a Bucarest sapendo che l'aereo parte domani.

- **Goal:** Arrivare a Bucarest
- **Formulazione del problema:**
  - Stati: città della Romania
  - Azioni: volare tra le città
- **Soluzione:** Sequenza di città

Si potrebbe usare una mappa per trovare il percorso più breve (visione completa del mondo) e trovare una soluzione ottimale. Questo problema è definito da 4 componenti:

- **Stato iniziale:** ad esempio "ad Arad"
- **Funzione di transizione:** insieme di coppie (stato, azione) che mappano uno stato in un altro, ad esempio:

$$S(A) = \{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots\}$$

- **Test dell'obiettivo:** una funzione che verifica se lo stato corrente soddisfa l'obiettivo, ad esempio:

$$\text{Goal-Test}(s) = \begin{cases} \text{true} & \text{se } s = \text{Bucarest} \\ \text{false} & \text{altrimenti} \end{cases}$$

- **Path cost:** è una funzione che assegna un costo (additivo) a ogni azione, ad esempio la somma di distanze o il numero di azioni:

$$c(x, a, y) \geq 0$$

- **Soluzione:** Una sequenza di azioni che portano dallo stato iniziale allo stato obiettivo.

## 3 Ricerca nello spazio degli stati

### 3.1 Ricerca generale

#### 3.1.1 Tree search

Un algoritmo di ricerca ad albero esplora lo spazio degli stati partendo dallo stato iniziale e generando nuovi stati (successori) applicando le azioni disponibili, cioè **espandendo** gli stati:

```

1 function Tree-Search(problem, strategy) function Tree-Search(
  problem, strategy) returns a solution, or failure
2 initialize the search tree using the initial state of problem
3 loop do
4   if no candidates for expansion then return failure
5   choose a leaf node for expansion according to strategy
6   if node contains a goal state then return the solution
7   else add successor nodes to the search tree (expansion)
8 endreturns a solution, or failure
9   initialize the search tree using the initial state of problem
10  loop do
11    if no candidates for expansion then return failure
12    choose a leaf node for expansion according to strategy
13    if node contains a goal state then return the solution
14    else add successor nodes to the search tree (expansion)
15 end

```

#### 3.1.2 Stato e nodo

Stato e nodo non sono la stessa cosa, infatti:

- **Stato:** rappresenta una configurazione dell'ambiente
- **Nodo:** è una struttura dati che costituisce una parte dell'albero di ricerca e include informazioni aggiuntive come il genitore, l'azione che ha portato a quello stato, il costo del percorso o la profondità nell'albero, ecc...

#### 3.1.3 Tree search generale

Espandere un nodo significa generare i suoi figli, cioè i nodi successori e tutti i nodi non esplorati sono chiamati **frontiera**.

```

1 function Tree-Search( problem, frontier) returns a solution, or
  failure
2   frontier <- Insert(Make-Node(problem.Initial-State))
3   while not IsEmpty(frontier) do
4     node <- Pop(frontier)
5     if problem.Goal-Test(node.State) then return node

```

```

6     frontier <- InsertAll(Expand(node, problem))
7   end loop
8   return failure

```

La strategia è quella di scegliere l'ordine in cui i nodi vengono espansi, cioè come viene gestita la frontiera. Le strategie sono valutate in base a:

- **Completezza:** se garantisce di trovare una soluzione quando esiste
- **Complessità di tempo:** numero di nodi generati o espansi
- **Complessità di spazio:** numero massimo di nodi memorizzati in memoria
- **Ottimalità:** se garantisce di trovare la soluzione migliore

Le complessità di spazio e di tempo sono misurate in termini di:

- *b*: maximum branching factor, numero massimo di figli per nodo
- *d*: profondità della soluzione meno costosa
- *m*: profondità massima dell'albero di ricerca (potrebbe essere infinita)

### 3.1.4 Stati ripetuti

Fallire nel riconoscere stati ripetuti può trasformare un problema lineare in un problema esponenziale. Bisogna quindi mantenere una lista di stati già visitati e non espandere nodi che portano a stati già visitati:

```

1 function Graph-Search( problem, frontier) returns a solution, or
   failure
2   explored <- an empty set
3   frontier <- Insert(Make-Node(problem.Initial-State))
4   while not IsEmpty(frontier) do
5     node <- Pop(frontier)
6     if problem.Goal-Test(node.State) then return node
7     if node.State is not in explored then
8       add node.State to explored
9       frontier <- InsertAll(Expand(node, problem))
10    end if
11  end loop
12  return failure

```

## 3.2 Ricerca non informata

Gli algoritmi di ricerca non informata utilizzano soltanto i dati disponibili nella definizione del problema e i principali sono:

- Breadth-first search
- Uniform-cost search (Dijkstra)
- Depth-first search
- Depth-limited search
- Iterative deepening search

### 3.2.1 Breadth-first search

Questo algoritmo espande il nodo non esplorato più superficiale, cioè il nodo più vicino alla radice. Utilizza una coda FIFO per la frontiera e i nuovi successori vengono aggiunti alla fine della coda.

```
1 function BFS( problem) returns a solution, or failure
2   node <- node with State=problem.Initial-State, Path-Cost=0
3   if problem.Goal-Test(node.State) then return node
4   explored <- empty set frontier <- FIFO queue with node as the
      only element
5   loop do
6     if frontier is empty then return failure
7     node <- Pop(frontier)
8     add node.State to explored
9     for each action in problem.Actions(node.State) do
10      child <- Child-Node(problem,node,action)
11      if child.State is not in (explored or frontier) then
12        if problem.Goal-Test(child.State) then return child
13        frontier <- Insert(child)
14      end if
15    end for
16  end loop
```

Questo tipo di ricerca è:

- **Completa:** Sì, soltanto se  $b$  è finito, cioè se il branching factor è limitato
- **Complessità di tempo:**  $b + b^2 + b^3 + \dots + b^d = O(b^d)$
- **Complessità di spazio:**  $O(b^d)$ , perchè bisogna memorizzare tutti i nodi generati
- **Ottimale:** Sì, soltanto se il costo delle azioni è uniforme

### 3.2.2 Uniform-cost search

Questo algoritmo espande il nodo non esplorato con il **costo del percorso più basso**. La frontiera è una coda di priorità ordinata in base al costo del percorso. Questo tipo di ricerca è:

- **Completa:** Sì, se il costo minimo delle azioni  $\geq \epsilon$  (con piccola ma  $\epsilon > 0$ )
- **Complessità di tempo:** Numero di nodi  $g \leq$  del costo del percorso ottimale  $C^*$ .  $O(b^{1+\lceil C^*/\epsilon \rceil})$
- **Complessità di spazio:**  $O(b^{1+\lceil C^*/\epsilon \rceil})$
- **Ottimale:** Sì perchè i nodi vengono espansi in ordine di costo del percorso

Ci sono due modifiche principali rispetto alla BFS che garantiscono l'ottimalità:

1. Il goal test viene fatto quando il nodo viene estratto dalla frontiera, non quando viene generato. (Questo elemento spiega il  $+1$  nella complessità)
2. Controllare se un nodo generato è già presente nella frontiera con un costo più alto e in tal caso sostituirlo con il nuovo nodo a costo più basso

### 3.2.3 Depth-first search

Questo algoritmo espande il nodo non esplorato più profondo, cioè il nodo più lontano dalla radice. Utilizza una pila LIFO per la frontiera e i nuovi successori vengono aggiunti all'inizio. Questo tipo di ricerca è:

- **Completa:** No, perchè può rimanere bloccata in un ramo infinito, a meno che l'albero di ricerca non abbia una profondità limitata. Si potrebbero evitare loop modificando l'algoritmo per evitare stati ripetuti sul percorso corrente
- **Complessità di tempo:**  $O(b^m)$ , dove  $m$  è la profondità massima dell'albero di ricerca
- **Complessità di spazio:**  $O(bm)$ , bisogna memorizzare soltanto il percorso corrente e i nodi fratelli
- **Ottimale:** No, perchè non garantisce di trovare la soluzione migliore

### 3.2.4 Iterative deepening search

Questo algoritmo combina i vantaggi della BFS e della DFS. Esegue una serie di ricerche in profondità limitata, aumentando progressivamente il limite di profondità fino a trovare una soluzione.

```
1 # Depth-Limited Search
2 function DLS(problem, limit) returns soln/fail/cutoff
3   R-DLS(Make-Node(problem.Initial-State), problem, limit)
4
5
6 function R-DLS(node, problem, limit) returns soln/fail/cutoff
7   if problem.Goal-Test(node.State) then return node
8   else if limit = 0 then return cutoff # raggiunta la profondità
9     massima
10  else
11    # flag: c'e' stato un cutoff in uno dei sottoalberi?
12    cutoff-occurred? <- false
13    for each action in problem.Actions(node.State) do
14      child <- Child-Node(problem, node, action)
15      result <- R-DLS(child, problem, limit-1)
16      if result = cutoff then cutoff-occurred? <- true
17      else if result = failure then return result
18    end for
19    if cutoff-occurred? then return cutoff else return failure
20  end else
21
22 # Iterative Deepening Search
23 function IDS(problem) returns a solution
24   inputs: problem, a problem
25   for depth <- 0 to infinity do
26     result <- DLS(problem, depth)
27     if result = cutoff then return result
28   end
```

Questo tipo di ricerca è:

- **Completa:** Sì
- **Complessità di tempo:**  $db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- **Complessità di spazio:**  $O(bd)$

- **Ottimale:** Sì, se il costo delle azioni è uniforme

**Esercizio 3.1.** Assumi:

1. Un albero di ricerca ben bilanciato, tutti i nodi hanno lo stesso numero di figli
2. Il goal state è l'ultimo che viene espanso nel suo livello (il più a destra)
3. Se il branching factor è 3, la soluzione più superficiale è a profondità 3 (la radice è a profondità 0) e si utilizza la ricerca in ampiezza quanti nodi vengono generati?
4. Se il branching factor è 3, la soluzione più superficiale è a profondità 3 (la radice è a profondità 0) e si utilizza la iterative deepening quanti nodi vengono generati?

**Esercizio 3.2.** Un uomo ha un lupo, una pecora e un cavolo. L'uomo è sulla riva di un fiume con una barca che può trasportare solo lui e un altro oggetto. Il lupo mangia la pecora e la pecora mangia il cavolo, quindi non può lasciarli insieme da soli.

1. Formalizza il problema come un problema di ricerca
2. Usa BFS per risolvere il problema

**Soluzione:**

Formalizziamo gli stati come una tupla:

$$\langle W, S, C, M, B \rangle$$

dove:

- W: posizione del lupo
- S: posizione della pecora
- C: posizione del cavolo
- M: posizione dell'uomo
- B: stato della barca

La posizione può essere 0 (left) o 1 (right).

Lo stato iniziale è:

$$\langle 0, 0, 0, 0, 0 \rangle$$

Lo stato obiettivo è:

$$\langle 1, 1, 1, 1, 1 \rangle$$

Le azioni possibili sono:

- Porta il lupo (CW)
- Porta la pecora (CS)
- Porta il cavolo (CC)
- Porta niente (CN)

Operatore	Precondizione	Funzione
CW	$M = B, M = W, S \neq C$	$\langle W, S, C, M, B \rangle \mapsto \langle \bar{W}, S, C, \bar{M}, \bar{B} \rangle$
CS	$M = B, M = S$	$\langle W, S, C, M, B \rangle \mapsto \langle W, \bar{S}, C, \bar{M}, \bar{B} \rangle$
CC	$M = B, M = C, W \neq S$	$\langle W, S, C, M, B \rangle \mapsto \langle W, S, \bar{C}, \bar{M}, \bar{B} \rangle$
CN	$M = B$	$\langle W, S, C, M, B \rangle \mapsto \langle W, S, C, \bar{M}, \bar{B} \rangle$

Notiamo che in tutte le precondizioni c'è  $M = B$  perchè l'uomo deve essere sempre con la barca, quindi si possono unire i due stati in uno solo  $M$ .

### 3.3 Ricerca informata

Gli algoritmi di ricerca informata utilizzano informazioni aggiuntive (euristiche) per guidare la ricerca verso la soluzione in modo più efficiente.

#### 3.3.1 Best-first search

Questo algoritmo usa una **funzione di valutazione** per ogni nodo che stima la "desiderabilità". La frontiera è una coda ordinata in ordine decrescente di desiderabilità. A seconda di come viene definita la desiderabilità si ottengono diversi algoritmi:

- Greedy best-first search
- A\*

#### 3.3.2 Greedy best-first search

Questo algoritmo espande il nodo che sembra essere il più vicino alla soluzione secondo una funzione di valutazione euristica  $h(n)$  che stima il costo rimanente per raggiungere l'obiettivo da un nodo  $n$ .

**Esempio 3.1.** In una mappa di una città, la funzione di valutazione potrebbe essere la distanza in linea d'aria dal nodo corrente alla destinazione. In questo modo, l'algoritmo esplora prima i nodi che sembrano più vicini alla destinazione, riducendo il numero di nodi esplorati rispetto a una ricerca non informata.

Questo tipo di ricerca è:

- **Completa:** No, perchè può rimanere bloccata in un ciclo infinito. È completo se lo spazio di ricerca è finito e ci sono controlli per evitare stati ripetuti
- **Complessità di tempo:**  $O(b^m)$  nel peggiore dei casi, ma può essere molto più veloce con una buona euristica

- **Complessità di spazio:**  $O(b^m)$ , bisogna memorizzare tutti i nodi generati
- **Ottimale:** No

### 3.3.3 A\* search

Questo algoritmo evita di espandere cammini che sono già molto costosi e ha come funzione di valutazione:

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

dove:

- $g(n)$ : costo del percorso dal nodo iniziale a  $n$
- $h(n)$ : stima del costo rimanente per raggiungere l'obiettivo da  $n$
- $f(n)$ : stima del costo totale del percorso passando per  $n$

L'euristica, per poter garantire l'ottimalità, deve essere **ammissibile**, cioè per ogni nodo la stima di quel nodo deve essere minore o uguale del vero costo per arrivare all'obiettivo, quindi non deve **sovrastimare** il costo rimanente:

$$h(n) \leq h^*(n) \quad h(n) \geq 0 \rightarrow h(G) = 0$$

dove  $h^*(n)$  è il costo effettivo del percorso da  $n$ .

**Teorema 3.1.** Per A\* l'euristica ammissibile implica l'ottimalità

Questo tipo di ricerca è:

- **Completa:** Sì, tranne se ci sono infiniti nodi con  $f \leq f(G)$
- **Complessità di tempo:** Esponenziale in errore relativo in  $h \times$  lunghezza del numero di passi della soluzione ottimale. (Se l'euristica è buona, la complessità sarà molto più bassa)
- **Complessità di spazio:**  $O(b^d)$ , bisogna memorizzare tutti i nodi generati
- **Ottimale:** Sì, ma richiede assunzioni sull'euristica (ammissibilità, consistenza) e una strategia di ricerca (ricerca ad albero o grafo)

### 3.3.4 Consistenza e ammissibilità

**Definizione 3.1.** Un euristica è **consistente** se:

$$h(n) \leq c(n, a, n') + h(n')$$



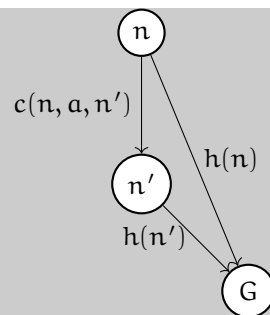


Figura 1: Esempio di euristica consistente

- Si può dimostrare che se  $h$  è consistente, allora  $f(n)$  non decresce lungo qualsiasi cammino
- A\* espande i nodi in ordine crescente di  $f$ , quindi trova sempre la soluzione ottimale

Quindi si espande sempre prima un cammino ottimo rispetto a un cammino non ottimo.

La consistenza implica l'ammissibilità e può essere dimostrato per induzione sul cammino verso il goal. L'ammissibilità però non implica la consistenza.

Consistenza  $\rightarrow$  Ammissibilità

Ammissibilità  $\nrightarrow$  Consistenza

- Tree-Search + euristica ammissibile  $\rightarrow$  A\* ottimale
- Graph-Search + euristica ammissibile  $\nrightarrow$  A\* ottimale (può scartare il cammino ottimale per un nodo ripetuto)
- Graph-Search + euristica consistente  $\rightarrow$  A\* ottimale

### 3.3.5 Euristiche

Le euristiche possono essere create in diversi modi, prendiamo ad esempio l'8-puzzle:

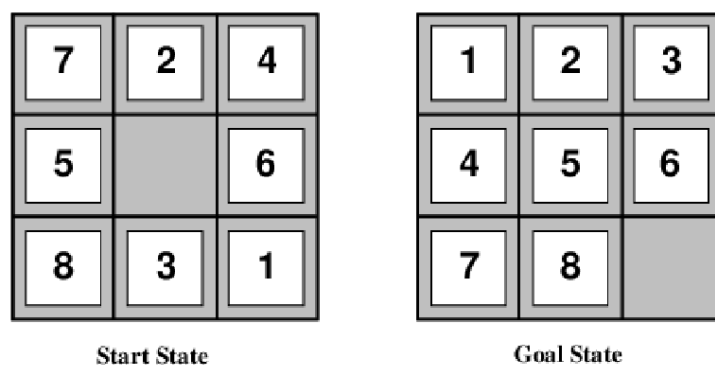


Figura 2: Esempio di 8-puzzle

Per questo problema si potrebbe utilizzare come euristica:

- $h_1(n)$  = numero di pezzi fuori posto
- $h_2(n)$  = somma delle distanze di Manhattan (numero di mosse orizzontali e verticali necessarie per portare ogni pezzo alla sua posizione obiettivo)

Entrambe le euristiche sono ammissibili, ma  $h_2$  è più precisa di  $h_1$  perchè fornisce una stima più vicina al costo reale per raggiungere l'obiettivo.

In questo caso si dice che  $h_2$  **domina**  $h_1$  se sono entrambe ammissibili e  $h_2(n)$  è sempre maggiore o uguale a  $h_1$ :

$$h_2(n) \geq h_1(n) \quad \forall n$$

**Teorema 3.2.** Date due qualsiasi euristiche **ammissibili**  $h_a$  e  $h_b$ , allora l'euristica definita come:

$$h(n) = \max(h_a(n), h_b(n))$$

è anch'essa ammissibile e domina sia  $h_a$  che  $h_b$

Le euristiche ammissibili possono essere derivate dall'esatto costo della soluzione di un problema **rilassato**, cioè un problema simile a quello originale ma con restrizioni rimosse. Ad esempio, per l'8-puzzle si potrebbe rilassare il problema permettendo di muovere una casella ovunque (in questo caso  $h_1(n)$  da la soluzione migliore), oppure permettendo di muovere una casella in qualsiasi casella adiacente ( in questo caso  $h_2(n)$  da la soluzione migliore)

**Definizione utile 3.1.** Il costo della soluzione ottimale di un problema rilassato non è maggiore del costo della soluzione ottimale del problema reale.

### 3.4 Ricerca locale

La ricerca locale è una tecnica di ricerca che si concentra su una soluzione cercando di migliorarla iterativamente. Gli algoritmi più comuni sono:

- Hill climbing
- Simulated annealing
- Algoritmi genetici

Questo tipo di ricerca è utile quando il percorso per arrivare alla soluzione non è importante, ma solo la soluzione finale. Ci sono due approcci principali:

- Trovare la configurazione ottimale (ad esempio (TSP - Traveling Salesman Problem))
- Trovare una configurazione che soddisfa dei vincoli (ad esempio il problema delle  $n$  regine)

Si possono anche usare algoritmi di **iterative improvement** che partono da una configurazione iniziale e cercano di migliorarla iterativamente fino a raggiungere un punto di ottimo locale.

**Esempio 3.2.** Un esempio di ricerca locale è il Traveling Salesman Problem (TSP), dove l'obiettivo è trovare il percorso più breve che visita un insieme di città esattamente una volta e ritorna alla città di partenza.

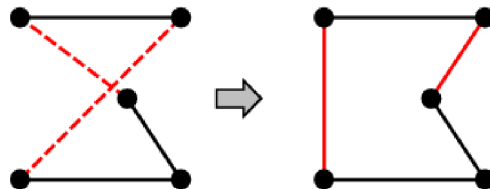


Figura 3: Esempio di TSP

Varianti di questo approccio arrivano fino a 1% della soluzione ottimale in tempi ragionevoli per migliaia di città.

**Esempio 3.3.** Un altro esempio è il problema delle  $n$  regine, dove l'obiettivo è posizionare  $n$  regine su una scacchiera  $n \times n$  in modo che nessuna regina minacci un'altra.

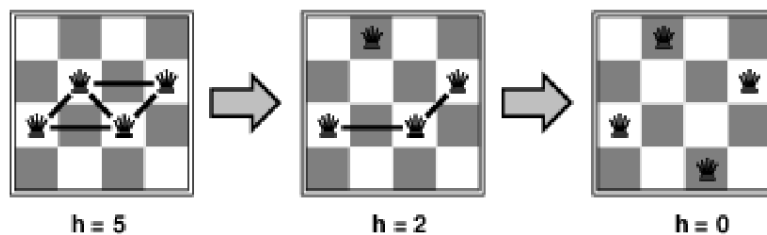


Figura 4: Esempio di  $n$ -regine

Risolve quasi sempre il problema quasi istantaneamente per  $n$  molto grande, ad esempio  $n = 1.000.000$ .

### 3.4.1 Hill climbing

Hill climbing è un algoritmo di ricerca locale che parte da una soluzione iniziale e cerca di migliorarla iterativamente spostandosi verso la soluzione migliore nella sua vicinanza. L'algoritmo continua a muoversi finché non trova un punto dove nessuna delle soluzioni vicine è migliore della soluzione corrente. Ritorna un massimo locale, che potrebbe non essere il massimo globale.

```

1 function Hill-Climbing(problem) returns a state that is a local
   maximum
2   inputs: problem, a problem
3   local variables: current, a node
4                   neighbor, a node
5   current <- Make-Node(problem.Initial-State)

```

```

6  loop do
7    neighbor <- a highest-valued successor of current
8    if neighbour.Value <= current.Value then return
9      current.State
10   end if
11   current <- neighbor
12 end

```

La rappresentazione del problema assumendo di avere tutti gli stati sull'asse x e il valore della funzione obiettivo sull'asse y è chiamata **state space landscape**:

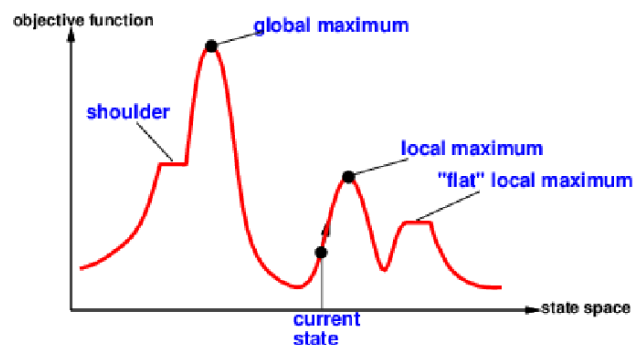


Figura 5: Esempio di state space landscape per hill climbing

- **Random-restart hill climbing**: esegue hill climbing più volte da stati iniziali casuali per aumentare le probabilità di trovare il massimo globale
- **Random sideways moves**: permette di fare mosse che non migliorano la soluzione corrente per evitare di rimanere bloccati in una sezione piatta

### 3.4.2 Simulated annealing

Simulated annealing è un algoritmo di ricerca locale ispirato al processo di raffreddamento dei metalli. L'algoritmo parte da una soluzione iniziale e cerca di migliorarla iterativamente, ma a differenza di hill climbing, permette di **accettare soluzioni peggiori** con una certa probabilità che **diminuisce nel tempo**. Questo aiuta a evitare di rimanere bloccati in massimi locali.

```

1  function Simulated-Annealing( problem, schedule) returns a solution
    state
2    inputs: problem, a problem
3           schedule, a mapping from time to 'temperature'
4    local variables: current, a node
5                   next, a node
6                   T, a 'temperature' controlling prob. of downward
    steps
7    current <- Make-Node(problem.Initial-State)
8    for t <- 1 to infinity do
9      T <- schedule(t) // temperature at time t
10     if T = 0 then return current
11     next <- a randomly selected successor of current
12     deltaE <- next.Value - current.Value
13     if deltaE > 0 then current <- next
14     else current <- next only with probability e-(deltaE/T)

```

Se la "temperatura"  $T$  diminuisce lentamente abbastanza, allora l'algoritmo converge **sempre** alla soluzione ottimale  $x^*$ . Questo perchè:

$$e^{\frac{E(x^*)}{kT}} / e^{\frac{E(x)}{kT}} = e^{\frac{E(x^*) - E(x)}{kT}} \gg 1 \quad \text{per } T \rightarrow 0$$

### 3.4.3 Local beam search

Local beam search è un algoritmo di ricerca locale che mantiene **un insieme di soluzioni candidate** e cerca di migliorarle iterativamente. In ogni iterazione, l'algoritmo **seleziona casualmente i successori di  $k$ , ma con un bias verso i migliori** per formare il nuovo insieme di soluzioni candidate.

### 3.4.4 Algoritmi genetici

Gli algoritmi genetici sono una classe di algoritmi di ricerca ispirati ai processi di evoluzione biologica. Questi algoritmi utilizzano meccanismi simili alla selezione naturale, alla mutazione e al crossover per evolvere una popolazione di soluzioni candidate verso soluzioni migliori. Gli algoritmi genetici hanno bisogno di stati encodati come stringhe di caratteri. Il crossover combina due stringhe per creare una nuova stringa e ha senso solo se le sottostringhe hanno un significato indipendente.

**Esempio 3.4.** Prendiamo ad esempio il problema delle  $n$  regine in cui si ha un encoding della scacchiera come un numero in cui la cifra in posizione  $i$  rappresenta la riga in cui si trova la regina nella colonna  $i$ .

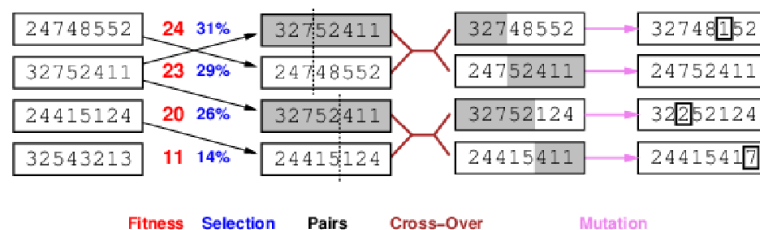


Figura 6: Esempio di algoritmo genetico per il problema delle  $n$ -regine

Questo genera una configurazione nuova a partire da coppie di configurazioni esistenti, selezionate in base alla loro "fitness" (un valore che misura quanto una configurazione si avvicina alla soluzione ottimale).

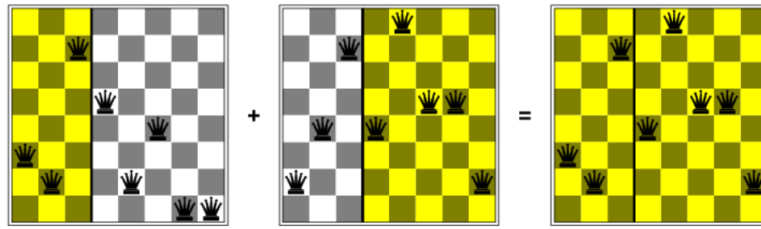


Figura 7: Rappresentazione della scacchiera

In questo caso la fitness è calcolata come il numero di regine che si minacciano a vicenda. Il caso peggiore è quello in cui tutte le regine si minacciano a vicenda, quindi la fitness è 28 (per  $n = 8$ ) perchè:

$$\text{num\_minacce} = \frac{n(n-1)}{2} = \frac{8 \cdot 7}{2} = 28$$

quindi:

$$\text{fitness} = 28 - \text{num\_minacce}$$