Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso a Libera Scelta - 6 CFU

Introduzione all'Intelligenza Artificiale

Professore:
Prof. Alessio Micheli
Prof. Claudio Gallicchio

Autore:
Filippo Ghirardini

Anno Accademico 2023/2024

Contents

Introduzione all'Intelligenza Artificiale

Realizzato da: Ghirardini Filippo

A.A. 2023-2024

---

1 Introduzione

1.1 Obiettivi dell'IA

1.1.1 Modellare

Modellare fedelmente l'essere umano:

- **Agire umanamente**: Test di Turing[1]
- **Pensare umanamente**: modelli cognitivi per descrivere il funzionamento della mente umana

1.1.2 Risultati

Raggiungere i risultati ottimali:

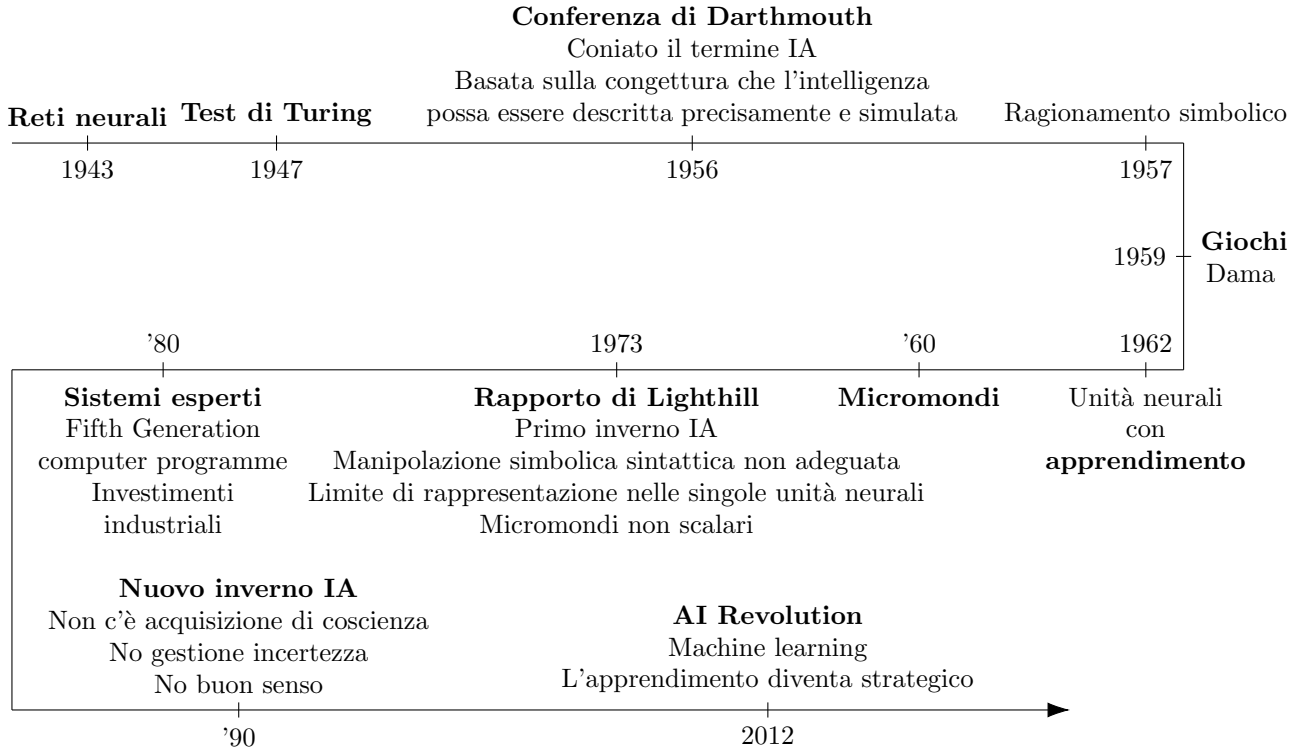
- Pensare razionalmente
- Agenti razionali: percepiscono l'ambiente, operano autonomamente e si adattano. Fanno la cosa giusta agendo in modo da ottenere il miglior risultato calcolando come agire in modo efficace e sicuro in una varietà di situazioni nuove. Ha alcuni vantaggi:

  1. Estendibilità e generalità
  2. Misurabilità dei risultati rispetto all'obiettivo

  I limiti dipendono dai rischi, dall'etica e dalla complessità computazionale.

1.2 Storia dell'IA

Nasce sin dall'antichità con il desiderio dei filosofi di sollevare l'uomo dalle fatiche del lavoro. Dal 1940 c'è un esplosione di popolarità che però si alterna tra periodi di crisi e di grandi avanzamenti.



[1] Ci sono due umani e una macchina. Tutti questi conversano tramite un computer. Se l'esaminatore non riesce a distinguere l'essere umano dalla macchina allora vince quest'ultima.

Esempio 1.2.1 (Scacchi)**.** Un esempio propedeutico è quello dell'applicazione dell'IA al gioco degli scacchi, definita **IA debole**. Negli anni '60 c'erano principalmente due opinioni al riguardo:

- Newell e Simon sostenevano che in 10 anni le macchine sarebbero state campioni negli scacchi
- Dreyfus sosteneva che una macchina non sarebbe mai stata in grado di giocare a scacchi

Nel 1997 la macchina Deep Blue sconfigge il campione mondiale di scacchi Kasparov. Viene naturale farsi alcune domande...

- Ha avuto **fortuna**?
- Ha avuto un **vantaggio psicologico**? La macchina eseguiva le mosse immediatamente e Kasparov si sentiva come l'ultimo baluardo umano.
- **Forza bruta**? La macchina calcolava 36 miliardi di posizioni ogni 3 minuti

Oggi l'Intelligenza Artificiale eccelle in tutti i giochi. L'ultimo a "cadere" è stato il Go nel 2016. Allo stesso tempo però il livello delle persone è aumentato giocando contro le macchine.

Definizione 1.2.1 (IA debole)**.** *Al contrario dell'IA forte, non ha lo scopo di possedere abilità cognitive generali, ma piuttosto di essere in grado di risolvere esattamente un singolo problema.*

1.3 Reti neurali

Le reti neurali sono caratterizzate da:

- **Flessibilità**: capacità di acquisizione automatica di conoscenza e di adattamento automatico a contesti diversi e dinamici
- **Robustezza**: capacità di trattare incertezza e rumorosità del mondo reale
- Rappresentazione appresa dai dati in forma **sub-simbolica**
- Possibilità di usare più strati di reti neurali con diversi livelli di astrazione (**Deep Learning**)

1.3.1 Deep Learning

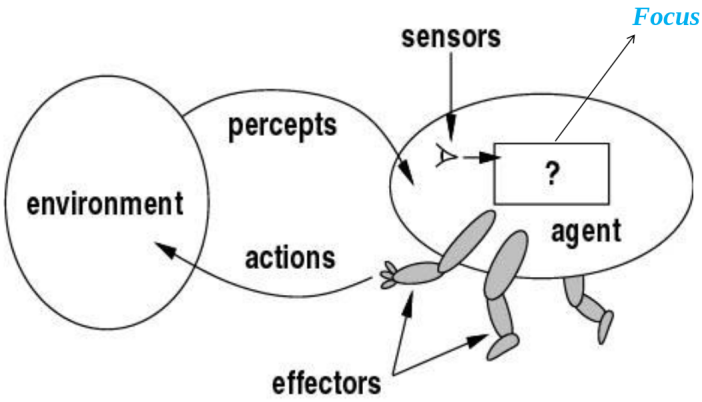
Abbinando alla capacità dei modelli di machine learning una grande quantità di dati e degli High Performance Computer, si è favorito molto il deep learning.
Dal 2010 le reti neurali profonde hanno iniziato a diffondersi molto nelle grandi industrie, riscuotendo successo ad esempio:

- **Computer vision**: ad esempio la classificazione del cancro della pelle
- **Natural Language Processing**: ad esempio IBM Watson o Google DeepL

Questa tecnologia ha raggiunto prestazioni a livello di quelle umane.

2 Agenti intelligenti

L'approccio moderno dell'IA (AIMA) è quello di costruire degli **agenti intelligenti**. La visione ad agenti offre n quadro di riferimento e una prospettiva più generale. È utile anche perché è **uniforme**.



Ciclo *percezione- azione*

Noi ci concentreremo sul programma che sta al centro dell'agente e che consiste in un ciclo di percezione-azione.

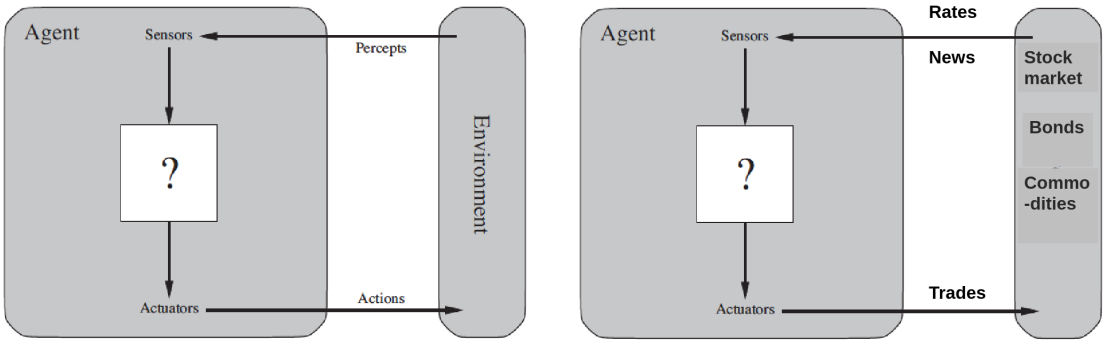
2.1 Caratteristiche

Un agente ha alcune caratteristiche:

- **Situati**: ricevono *percezioni* da un ambiente e agiscono mediante **azioni** (attuatori)

2.1.1 Percezioni e azioni

Le percezioni corrispondono agli **input** dai sensori. La **sequenza percettiva** sarà la storia completa delle percezioni.
La scelta dell'azione è *funzione* unicamente della sequenza percettiva ed è chiamata **funzione agente**.
Il compito dell'IA è costruire il programma agente.



2.2 Agente razionale

Definizione 2.2.1 (Agente razionale)**.** *Un agente razionale interagisce con il suo ambiente in maniera* ***efficace*** *(fa la cosa giusta).*

Si rende quindi necessario un **criterio di valutazione** oggettivo dell'effetto delle azioni dell'agente. La valutazione della prestazione deve avere le seguenti caratteristiche:

- **Esterna**
-

•

Definizione 2.2.2 (Agente razionale)**.** *Per ogni sequenza di percezioni compie l'azione che massimizza il valore atteso della misura delle prestazioni, considerando le sue percezioni passate e la sua conoscenza pregressa.*

Osservazione 2.2.1. Si basa sulla razionalità e non sull'onniscenza e onnipotenza: non conosce alla perfezione il futuro ma può apprendere e hai dei limiti nelle sue azioni.

Raramente tutta la conoscenza sull'ambiente può essere fornita a priori dal programmatore. L'agente razionale deve essere in grado di modificare il proprio comportamento con l'esperienza. Può **migliorare** esplorando, apprendendo, aumentando l'autonomia per operare in ambienti differenti o mutevoli.

Definizione 2.2.3 (Agente autonomo)**.** *Un agente è autonomo nella misura in cui il suo comportamento dipende dalla sua capacità di ottenere esperienza e non dall'aiuto del progettista.*

2.3 Ambienti

Definire un problema per un agente significa innanzitutto caratterizzare l'ambiente in cui opera. Viene utilizzata la descrizione **PEAS**:

- **P**erformance
- **E**nviroment
- **A**ctuators
- **S**ensors

Prestazione	Ambiente	Attuatori	Sensori
Arrivare alla destinazione, sicuro, veloce, ligio alla legge, viaggio confortevole, minimo consumo di benzina, profitti massimi	Strada, altri veicoli, pedoni, clienti	Sterzo, acceleratore, freni, frecce, clacson, schermo di interfaccia o sintesi vocale	Telecamere, sensori a infrarossi e sonar, tachimetro, GPS, contachilometri, accelerometro, sensori sullo stato del motore, tastiera o microfono

L'ambiente deve avere le seguenti proprietà:

- Osservabilità:
  - Se è **completamente osservabile** l'apparato percettivo è in grado di dare conoscenza completa dell'ambiente o almeno tutto ciò che è necessario per prendere l'azione
  - Se è **parzialmente osservabile** sono presenti limiti o inaccuratezze dell'apparato sensoriale
- Agente singolo o multi-agente:
  - L'ambiente ad agente **singolo** può anche cambiare per eventi, non necessariamente per azioni di agenti
  - Quello **multi-agente** può essere *competitivo* (scacchi) o *cooperativo*
- Predicibilità:
  - **Deterministico**: quando lo stato successivo è completamente determinato dallo stato corrente e dall'azione (e.g. scacchi)
  - **Stocastico**: quando esistono elementi di incertezza con associata probabilità (e.g. guida)
  - **Non deterministico**: quando si tiene traccia di più stati possibili risultato dell'azione ma non in base ad una probabilità

- Episodico o sequenziale:
  - **Episodico**: quando l'esperienza dell'agente è divisa in episodi atomici indipendenti in cui non c'è bisogno di pianificare (e.g. partite diverse)
  - **Sequenziale**: quando ogni decisione influenza le successive (e.g. mosse di scacchi)
- Statico o dinamico:
  - **Statico**: il mondo non cambia mentre l'agente decide l'azione (e.g cruciverba)
  - **Dinamico**: cambia nel tempo, va osservata la contingenza e tardare equivale a non agire (e.g. taxi)
  - **Semi-dinamico**: l'ambiente non cambia ma la valutazione dell'agente sì (e.g. scacchi con timer)
- Valori come lo stato, il tempo, le percezioni e le azioni possono assumere valori **discreti** o **continui**. Il problema è combinatoriale nel discreto o infinito nel continuo.
- **Noto** o **ignoto**: una distinzione riferita alla conoscenza dell'agente sulle leggi fisiche dell'ambiente (le regole del gioco). È diverso da osservabile.

Definizione 2.3.1 (Simulatore)**.** *Un simulatore è uno strumento software che si occupa di:*

- *Generare stimoli*
- *Raccogliere le azioni in risposta*
- *Aggiornare lo stato*
- *Attivare altri processi che influenzano l'ambiente*
- *Valutare la prestazione degli agenti (media su più istanze)*

Gli esperimenti su classi di ambienti con condizioni variabili sono essenziali per ***generalizzare****.*

2.4 Programma agente

L'agente sarà quindi composto da un'architettura e da un programma. Il programma dell'agente implementa la funzione agente $Ag : Percezioni \rightarrow Azioni$.

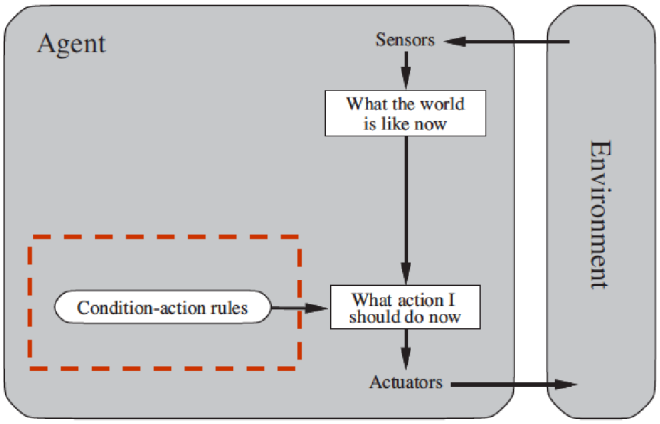
```
function Skeleton-Agent (percept) returns action
   static: memory, agent memory of the world
   memory <- UpdateMemory(memory, percept)
   action <- Choose-Best-Action(memory)
   memory <- UpdateMemory(memory, action)
   return action
```

2.4.1 Tabella

Un agente basato su tabella esegue una scelta come un accesso ad una tabella che associa un'azione ad ogni possibile sequenza di percezioni.
Ha una **dimensione ingestibile**, è difficile da costruire, non è autonomo ed è di difficile aggiornamento (apprendimento complesso).

2.4.2 Agenti reattivi

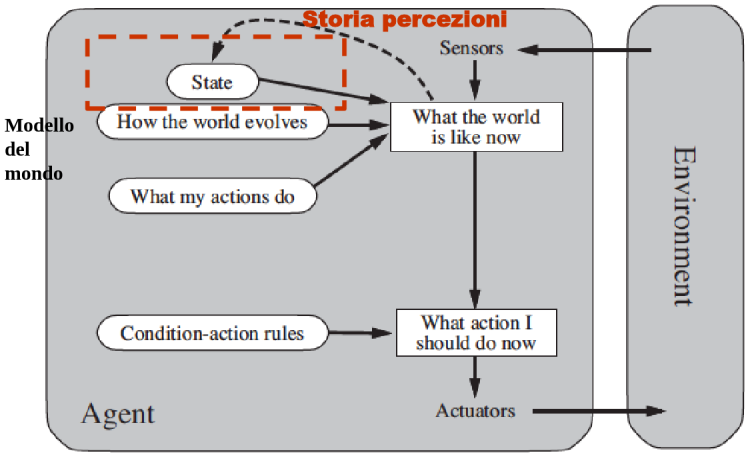
L'agente agisce in base a quello che percepisce senza salvare nulla in memoria.




```
function Agente-Reattivo-Semplice (percezione)
   returns azione
   persistent: regole, un insieme di regole
   condizione-azione (if-then)
   stato <- Interpreta-Input(percezione)
   regola <- Regola-Corrispondente(stato, regole)
   azione <- regola.Azione
   return azione
```

2.4.3 Agenti basati su modello

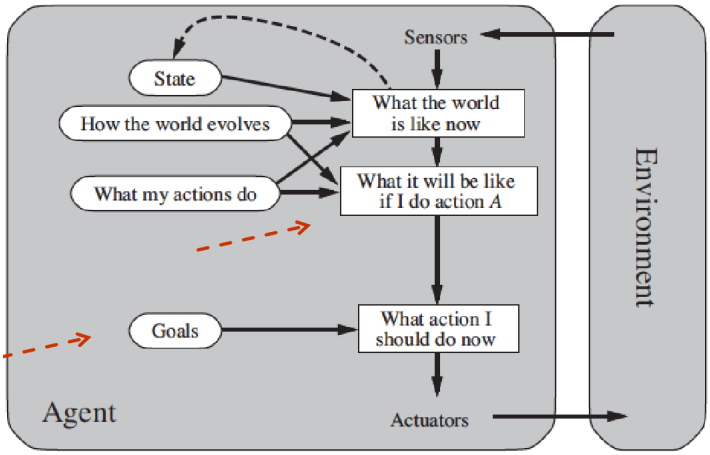
L'agente ha uno stato che mantiene la storia delle percezioni e influenza il modello del mondo.




```
function Agente-Basato-su-Modello (percezione)
   returns azione
   persistent: stato, una descrizione dello stato corrente
               modello, conoscenza del mondo
               regole, un insieme di regole condizione-azione
               azione, azione piu recente
   stato <- Aggiorna-Stato(stato, azione, percez., modello)
   regola <- Regola-Corrispondente(stato, regole)
   azione <- regola.Azione
   return azione
```

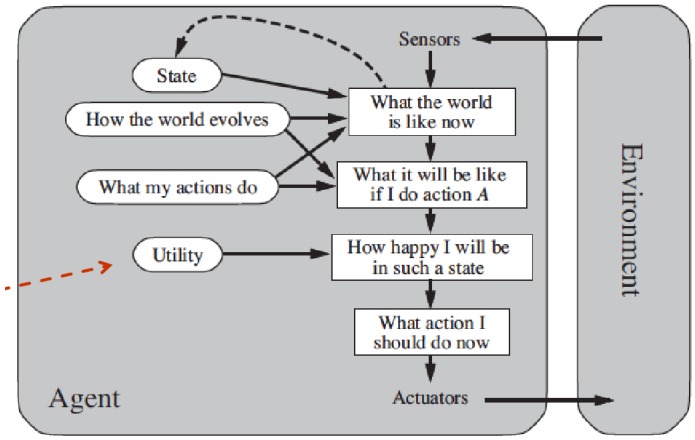
2.4.4 Agenti con obiettivo

Fin'ora l'agente aveva un obiettivo predeterminato dal programma. In questo caso invece viene specificato anche il **goal** che influenza le azioni. Abbiamo quindi più **flessibilità** ma meno efficienza.



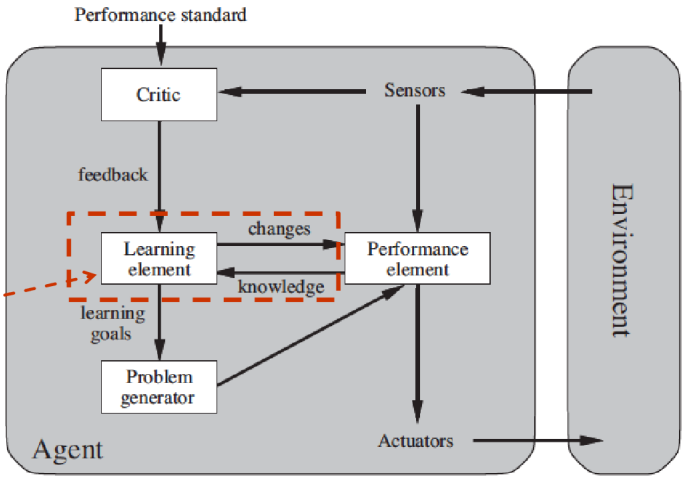
2.4.5 Agenti con valutazione di utilità

In questo caso ci sono **obiettivi alternativi** o più modi per raggiungerlo. L'agente deve quindi decidere verso dove muoversi e si rende necessaria una **funzione utilità** che associ ad un obiettivo un numero reale. La funzione terrà anche conto della probabilità di successo (**utilità attesa**).



2.4.6 Agenti che apprendono

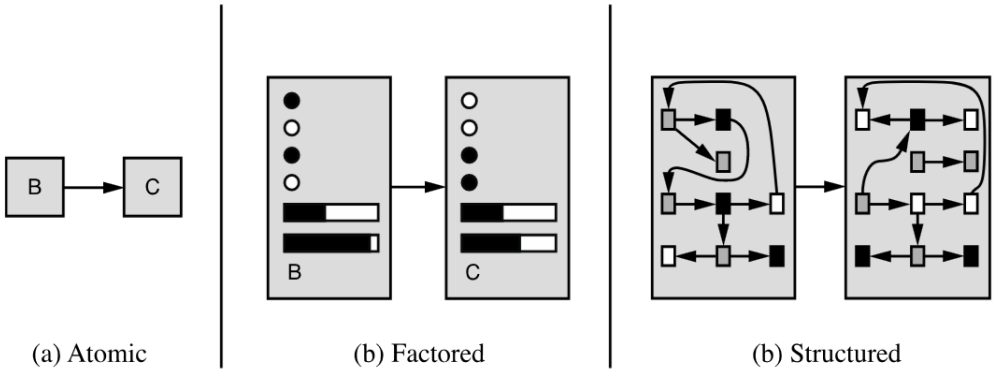
Questo tipo di agente include la capacità di **apprendimento** che produce cambiamenti al programma e ne migliora le prestazioni, adattando i comportamenti.
L'elemento **esecutivo** è il programma stesso, quello **critico** osserva e dà feedback ed infine c'è un generatore di problemi per suggerire nuove situazioni da esplorare.



2.4.7 Tipi di rappresentazione

Gli stati e le transizioni possono essere rappresentati in tre modi:

- **Atomica**: solo con gli stati
- **Fattorizzata**: con più variabili e attributi
- **Strutturata**: con l'aggiunta delle relazioni



3 Agenti risolutori di problemi

Gli agenti risolutori di problemi adottano il paradigma della risoluzione di problemi come **ricerca** in uno **spazio di stati**. Sono agenti con **modello** (storia percezioni e stati) che adottano una rappresentazione **atomica** degli stati.
Sono particolari gli agenti con **obiettivo** che pianificano l'intera sequenza di mosse prima di agire.

3.1 Processo di risoluzione

I passi da seguire sono i seguenti:

1. **Determinazione di un obiettivo**, ovvero un insieme di stati in cui l'obiettivo è soddisfatto
2. **Formulazione** del problema tramite la rappresentazione degli stati e delle azioni
3. Determinazione della **soluzione** mediante la ricerca
4. **Esecuzione** del piano

Esempio 3.1.1 (Viaggio con mappa)**.** Supponiamo di voler fare un viaggio. Il processo di risoluzione sarebbe il seguente:

1. Raggiungere Bucarest
2. - Azioni: guidare da una città all'altra
   - Stato: città su mappa

3.2 Assunzioni

Assumiamo che l'ambiente in questione sia **statico**, **osservabile**, **discreto** e **deterministico** (assumiamo un mondo ideale).

3.3 Formulazione del problema

Un problema può essere definito formalmente mediante 5 componenti:

1. **Stato iniziale**
2. **Azioni** possibili
3. **Modello di transizione**: $ris : stato \times azione \to stato$, uno stato *successore* $ris(s, a) = s'$
4. **Test obiettivo** per capire tramite un insieme di stati obiettivo se il goal è raggiunto $test : stato \to \{true, false\}$
5. **Costo del cammino**: composto dalla somma dei costi delle azioni, dove un passo ha costo $c(s, a, s')$. Un passo non ha mai costo negativo.

I punti 1, 2 e 3 definiscono implicitamente lo **spazio degli stati**. Definirlo esplicitamente può essere molto costoso.

3.4 Algoritmo di ricerca

Gli algoritmi di ricerca prendono in input un problema e restituiscono un **cammino soluzione**.
Dobbiamo misurare le **prestazioni**: trova una soluzione? Quanto costa trovarla? Quanto è efficiente?

$$costo\_totale = costo\_ricerca + costo\_cammino\_sol$$

Esempio 3.4.1 (Arrivare a Bucarest)**.** Partiamo con la formulazione del problema:

1. **Stato iniziale**: la città di partenza, ovvero Arad
2. **Azioni**: spostarsi in una città collegata vicina

```
Azioni(In(Arad))={Go(Sibiu),Go(Zerind),...}
```

3. **Modello di transizione**:

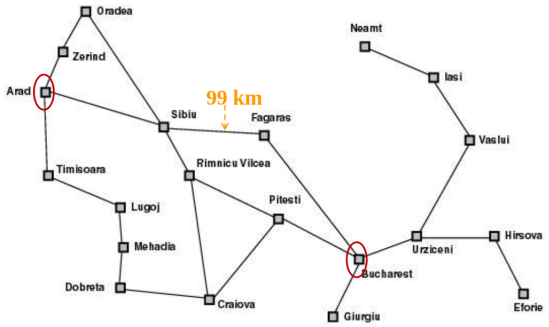
```
Risultato(In(Arad), Go(Sibiu)) = In(Sibiu)
```

4. **Test obiettivo**:

```
{In(Bucarest)}
```

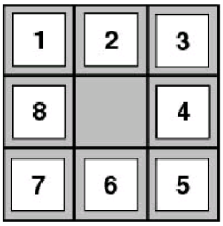
5. **Costo del cammino**: somma delle lunghezze delle strade

In questo esempio lo spazio degli stati coincide con la rete dei collegamenti tra le città.



Esempio 3.4.2 (Puzzle dell'8)**.** Partiamo con la formulazione del problema:

1. **Stati**: tutte le possibili configurazioni della scacchiera
2. **Stato iniziale**: una configurazione tra quelle possibili
3. **Obiettivo**: una configurazione del tipo



4. **Azioni**: le mosse della casella vuota
5. **Costo cammino**: ogni passo costa 1

In questo esempio lo spazio degli stati è un grafo con possibili cicli (ci possiamo ritrovare in configurazioni già viste). Il problema è NP-completo: per 8 tasselli ci sono $\frac{9!}{2} = 181.000$ stati.

Esempio 3.4.3 (8 regine)**.** Supponiamo di dover collocare 8 regine su una scacchiera in modo tale che nessuna regina sia attaccata da altre.

1. **Stati**: tutte le possibili configurazioni della scacchiera con 0-8 regine
2. **Goal test**: avere 8 regine sulla scacchiera, di cui nessuna è attaccata
3. **Azioni**: aggiungi una regina

In questo esempio lo spazio degli stati sono le possibili scacchiere, ovvero $64 \times 63 \times \ldots \times 57 \simeq 1.8 \times 10^{14}$.
Proviamo ad utilizzare una formulazione diversa:

1. **Stati**: tutte le possibili configurazioni della scacchiera in cui *nessuna regina è minacciata*
2. **Goal test**: avere 8 regine sulla scacchiera, di cui nessuna è attaccata
3. **Azioni**: aggiungere una regina nella colonna vuota più a destra ancora libera in modo che non sia minacciata

Lo spazio degli stati passa a 2057, anche se comunque rimane esponenziale per k regine.
Vediamo infine un'ultima formulazione:

1. **Stati**: scacchiere con 8 regine, una per colonna
2. **Goal test**: nessuna delle regine già presenti è attaccata
3. **Azioni**: sposta una regina nella colonna se minacciata
4. **Costo cammino**: zero

Qui lo spazio degli stati è di qualche decina di milione.
Capiamo quindi che formulazioni diverse del problema portano a spazi di stati di dimensioni diverse.

Esempio 3.4.4 (Dimostrazione di teoremi)**.** Dato un insieme di premesse:

$$\{s, t, q \Rightarrow p, r \Rightarrow p, v \Rightarrow q, t \Rightarrow r, s \Rightarrow v\} \tag{1}$$

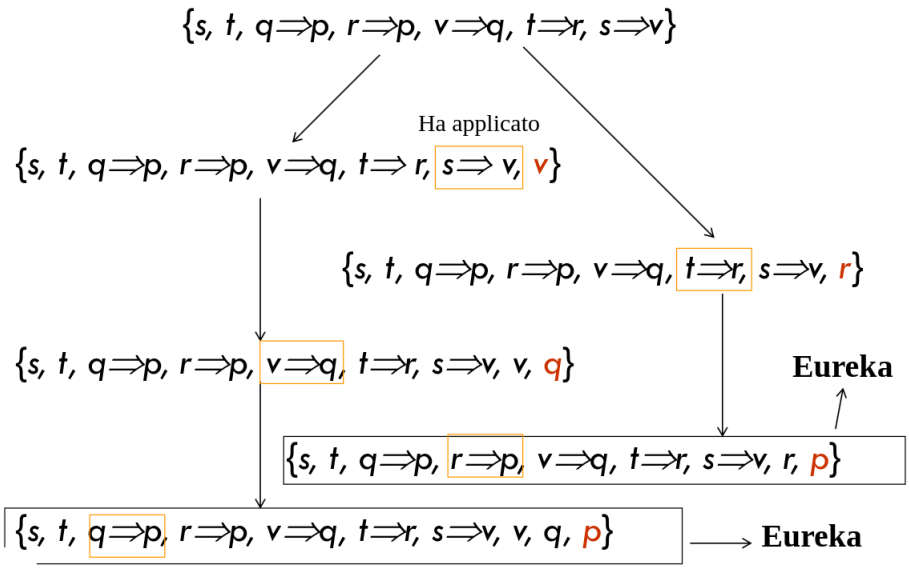
dimostrare una proposizione p utilizzando solamente la regola di inferenza *Modus Ponens*:

$$(p \wedge p \Rightarrow q) \Rightarrow q$$

Scriviamo la formulazione del problema:

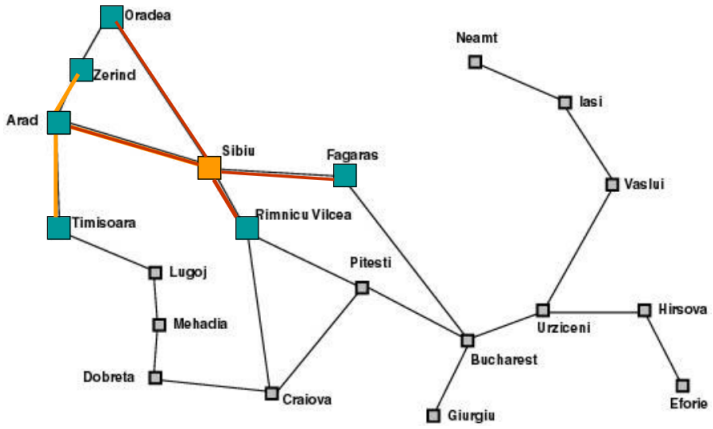
- **Stati**: insieme di proposizioni
- **Stato iniziale**: le premesse
- **Stato obiettivo**: un insieme di proposizioni contenente il teorema da dimostrare
- **Operatori**: l'applicazione del Modus Ponens

Lo spazio degli stati è quindi il seguente:



3.5 Ricerca della soluzione

La ricerca della soluzione consiste nella generazione di un **albero di ricerca** a partire dalle possibili sequenze di azioni che si sovrappone allo spazio degli stati.
Ad esempio per il caso di Bucarest:



Espandiamo ogni nodo con i suoi possibili successori (frontiera).

Definizione 3.5.1 (Frontiera)**.** *Lista dei nodi in attesa di essere espansi (le foglie dell'albero di ricerca).*

Osservazione 3.5.1. Si noti che un nodo dell'albero è diverso da uno stato. Infatti possono esitere nodi dell'albero di ricerca con lo stesso stato (si può tornare indietro).

3.6 Strategie di ricerca

Ci sono diversi tipi di strategia per la ricerca della soluzione:

- **FIFO**
- **LIFO**
- **Coda con priorità**

3.6.1 Breadth First

Come esplorare il grafo dello spazio degli stati a livelli progressivi di stessa profondità.
Per ogni nodo lo espandiamo, analizziamo i suoi figli (senza scendere ulteriormente di livello) e dopo averli fatti tutti scende di livello seguendo il principio FIFO.
Il seguente è il codice della **ricerca ad albero**, ovvero dove non si torna su un nodo già visitato.

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

Il seguente è invece quello della **ricerca su grafo**:

```
function Ricerca-Ampiezza-g
   returns soluzione oppure fallimento
   nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
   if problema.Test-Obiettivo(nodo.Stato) then return Soluzione(nodo)
   frontiera = una coda FIFO con nodo come unico elemento
   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 e in esplorati e non in frontiera then
   if Problema.TestObiettivo(figlio.Stato) then return Soluzione(figlio)
   frontiera = Inserisci(figlio, frontiera) /* in coda
end
```

4 Ricerca euristica

5 Ricerca locale

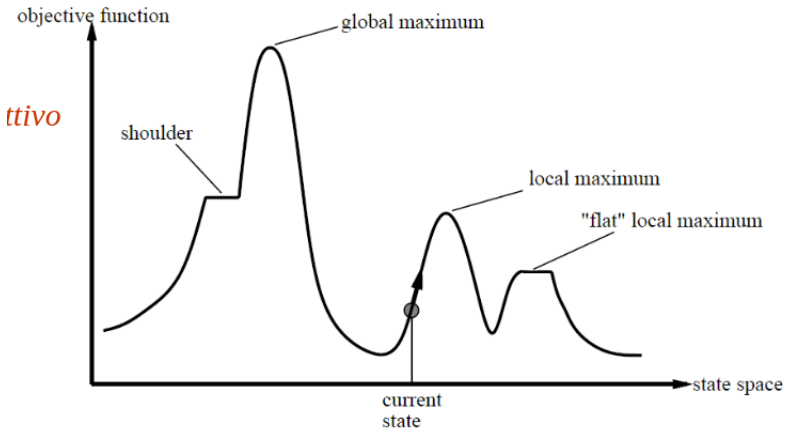
La ricerca *euristica* nello spazio di stati è troppo costosa ed è quindi necessario utilizzare metodi diversi.
Se prima gli algoritmi restituivano un cammino soluzione per raggiungere un goal, ora il goal è la soluzione stessa al problema. Gli algoritmi di ricerca locale sono adatti per problemi in cui:

- La sequenza di azioni non è importante ma conta solo lo stato goal
- Tutti gli elementi della soluzioni sono nello stato ma alcuni vincoli sono violati

Questi algoritmi non sono sistematici e tengono traccia solo del nodo corrente spostandosi su quelli adiacenti.
Non tengono traccia dei cammini: rendono più efficiente l'occupazione della memoria e possono trovare soluzioni anche in spazi di stati molto grandi o infiniti.
Sono utili per risolvere problemi di **ottimizzazione**:

- Stato migliore secondo una funzione obiettivo f
- Lo stato di costo minore (non il cammino)

Data la funzione euristica del costo dell'obiettivo



uno stato ha una posizione sulla superficie e un'altezza che corrisponde al valore della valutazione della funzione obiettivo. Un algoritmo provoca movimento sulla superficie e l'obiettivo è raggiungere un punto in particolare (e.g. massimo locale).

5.1 Hill climbing

Sfrutta un principio di ricerca locale greedy dove vengono generati i successori e vengono valutati. Viene scelto un nodo che migliora lo stato attuale e scartati gli altri:

- **Salita rapida** (o discesa): viene scelto il migliore
- **Stocastico**: scelta random
- **Prima scelta**: viene scelto il primo

Se non ci sono successori che migliorano lo stato, l'algoritmo termina con fallimento.

```
def hill_climbing(problem):
   current = Node(problem.initial_state)
   while True:
      neighbors = [current.child_node(problem, action) for action in
          problem.actions(current.state)]
      if not neighbors: # se current non ha successori esci e restituisci current
         break
```

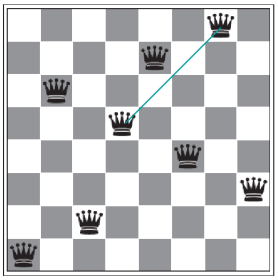
```
        # scegli il vicino con valore piu' alto (sulla funzione problem.value)
        neighbor = (sorted(neighbors,key = lambda x:problem.value(x), reverse = True))[0]
        if problem.value(neighbor) <= problem.value(current):
            break
        else:
            current = neighbor # (altrimenti, se vicino migliore, continua)
    return current
```

Non c'è frontiera a cui ritornare e si tiene un solo stato, quindi efficiente per la memoria. Il tempo necessario è variabile e dipende dal punto di partenza.

5.1.1 8 regine

Nel problema già descritto delle 8 regine, poniamo come funzione da minimizzare h il numero di coppie di regine che si attaccano a vicenda. Bisogna minimizzare h. Ogni regina può fare 7 mosse quindi abbiamo $7 \cdot 8 = 56$ possibili stati successivi. Tra i migliori con lo stesso valore di h si sceglie a caso.

Esempio 5.1.1 (8 regine)**.** Nel caso delle 8 regine:



Possiamo migliorare l'algoritmo in alcuni modi:

1. Consentire un numero limitato di **mosse laterali**, ovvero l'algoritmo si ferma solo quando è peggiore la soluzione e non peggiore o uguale (sulle 8 regine 94% di successo ma in media 21 passi)
2. Hill-climbing **stocastico** (più lento ma soluzioni migliori)
3. Hill-climbing **prima scelta**: genera mosse a caso fino a trovarne una migliore.
4. Implementiamo un **riavvio casuale** che fa ripartire l'algoritmo da un punto a caso. Se la probabilità di successo è p, saranno necessarie $\frac{1}{p}$ iterazioni. Con molti minimi locali nella funzione obiettivo, p si abbassa e aumentano il numero di volte in cui si blocca.

5.2 Tempra simulata

Questo algoritmo combina hill-climbing con una scelta stocastica non totalmente casuale.
Ad ogni passo si sceglie un successore n' a caso:

- Se **migliora** lo stato corrente, viene espanso
- Se lo **peggiora** ($\Delta E = f(n') - f(n) \leq 0$) quel nodo viene scelto con probabilità $p = e^{\frac{\Delta E}{T}} \quad 0 \leq p \leq 1$.

Questo significa che p è inversamente proporzionale al peggioramento. Con il progredire dell'algoritmo rende improbabili le mosse peggiorative.

5.2.1 Scelta dei parametri

I parametri sono il valore iniziale e il decremento di T. Il valore iniziale dovrebbe essere tale che per i valori medi di ΔE p sia circa 0.5.

5.3 Local beam

Dato l'algoritmo *beam*, vengono salvati in memoria solo k stati. Ad ogni passo si generano i successori di tutti i k stati e:

- Se si trova un goal, ci si ferma
- Altrimenti si prosegue con i k migliori tra questi

Note 5.3.1. È diverso da k restart, in quanto non si riparte da 0, e dal *beam search* perché non si tengono tutti gli stati.

5.3.1 Versione stocastica

Si introduce un elemento di casualità: i k successori vengono scelti con una probabilità maggiore per i migliori ma non tutti. Introduciamo della terminologia:

- **Organismo**: lo stato
- **Progenie**: i successori
- **Fitness**: il valore della funzione obiettivo

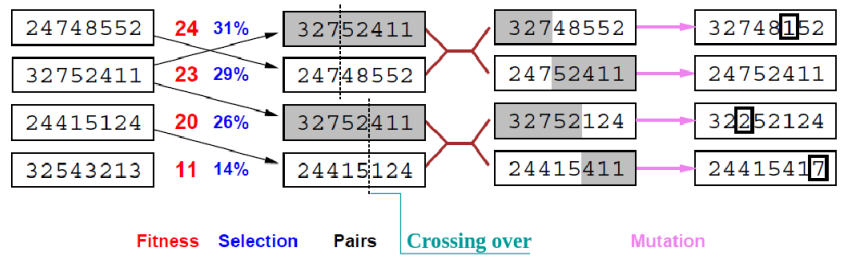
5.3.2 Algoritmi genetici ed evolutivi

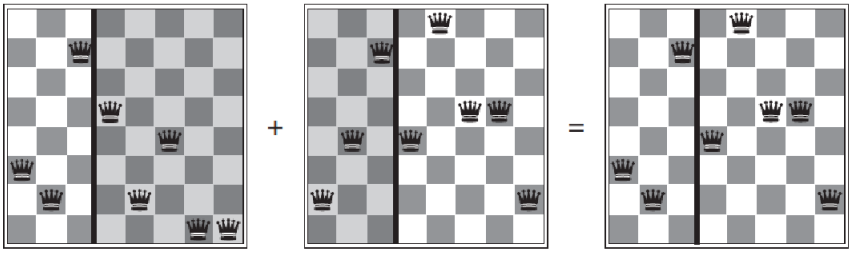
Sono una variante della *beam search stocastica* in cui gli stati successori sono ottenuti combinando due stati genitore invece che per evoluzione. La **popolazione** iniziale è composta da k **individui** generati casualmente e rappresentati come una stringa. Gli individui sono valutati da una funzione di **fitness**. Vengono poi selezionati quelli per l'**accoppiamento** che danno vita alla generazione successiva in due modi:

- **Crossover**: combinando il materiale genetico
- **Casuale**: con un meccanismo di mutazione genetica

Ogni generazione dovrebbe essere migliore della precedente.

Esempio 5.3.1 (8 regine)**.** Nel problema delle 8 regine abbiamo una popolazione di queste, dove le loro posizioni sono descritte da una stringa (ogni cifra è la riga in cui c'è la regina in quella colonna). La funzione di fitness è il numero di coppie di regine che non si attaccano. Per ogni coppia di combinazioni sulla scacchiera (scelta con la probabilità proporzionale alla fitness) viene scelto un punto di **crossing over** in maniera casuale e vengono generati due figli scambiandosi dei pezzi. Alla fine viene fatta una mutazione causale.





Questi algoritmi fanno parte del **Natural computer** e come vantaggi hanno:

- Tendenza a salire della beam search stocastica
- Interscambio delle informazioni tra thread paralleli di ricerca in maniera indiretta

Questo tipo di algoritmi sono più efficaci se il problema ha componenti significative rappresentate in stringhe; è proprio la rappresentazione ad essere il punto critico.

5.4 Spazi continui

Lo stato è descritto da variabili **continue** in un vettore $x = x_1, \ldots, x_n$. Un esempio è lo spazio tridimensionale.
L'apparente difficoltà dovuta ai fattori di ramificazione infiniti è affrontata tramite strumenti matematici quali il *gradiente*. Ad esempio l'**hill climbing iterativo** diventa:

$$x_{new} = x \pm \eta \nabla f(x)$$

sfruttando la direzione e lo spostamento che ci fornisce il gradiente invece di cercarlo tra gli infiniti successori.

Esempio 5.4.1. Prendiamo la funzione $f(x) = x^2$ con derivata prima $f'(x) = 2x$. Cerchiamo il minimo con

$$x_{new} = x - \eta f'(x)$$

Partendo ad esempio da $x = 2$ con $\eta = 0.2$, otteniamo come primo risultato $x_{new} = 2 - 0.8 = 1.2$.

5.5 Ambienti realistici

A differenza dei problemi classici, il nostro ambiente è **parzialmente osservabile** e **non deterministico**. Qui le **percezioni** sono importanti in quanto restringono gli stati possibili e informano sull'effetto dell'azione.
L'agente deve elaborare una strategia con un piano di contingenza che tenga conto delle diverse eventualità.

Esempio 5.5.1 (Aspirapolvere)**.** Un aspirapolvere imprevedibile ha due comportamenti:

- Se aspira in una stanza sporca la pulisce ma a volte pulisce anche una stanza adiacente
- Se aspira in una stanza pulita, a volte la sporca

La soluzione non è più una sequenza ma è un albero che gestisce il piano di di contingenza.

5.5.1 Albero AND-OR

È un albero che ha come nodi *OR* le scelte dell'agente e come nodi *AND* le diverse contingenze da considerare.

Esempio 5.5.2 (Aspirapolvere)**.** Nell'esempio 5.5.1 l'albero sarebbe: