Intelligenza Artificiale

Università di Verona
Imbriani Paolo -VR500437
Professor Alessandro Farinelli

November 6, 2025

Contents

1 Introduzione

Alle origini dell'intelligenza artificiale vi è un bisogno diverso da quello che abbiamo oggi. Alan Turing, negli anni 50 si era chiesto se le macchine potessero pensare, creando un test famoso ancora ora come "test di Turing" dove un interrogatore umano si deve interfacciare con un umano e una macchina e doveva capire chi dei due fosse chi. Nel 1956 ci fu uno studio fatto da il progetto di ricerca di Dartmouth, che aveva l'intento di risolvere compiti che richiedeva l'intelligenza di una persona attraverso una macchina, comprendendo che le *anche le macchine possono imparare*. La definizione più "accettata" di Intelligenza Artificiale è quella dove viene vista come una complessa e affascinante *disciplina* che studia come simulare l'intelligeza in scenari complessi usando come strumenti agenti autonomi per delle task ripetitive, sporche e pericolose che sfruttano l'analisi dei dati (predizione e classificazione).

Definizione 1.1

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

Bisogna distinguere machine learning e programmazione:

- **Programmazione**: macchine programmate per ogni task che devono eseguire (il concetto chiave è **il programma**)
- **Machine Learning**: insegnare alla macchina (attraverso esempi) come risolvere task più complesse (il concetto chiave è il **modello**)

1.1 Machine Learning

L'idea di far apprendere una macchina si possono dividere in tre paradigmi contraddisti:

- Unsupervised learning
- Supervised learning
- Reinforcement learning

Esistono poi i trasformatori, che sono modelli di machine learning probabilistici che si basano sul concetto di attenzione, che sono alla base di modelli come GPT. Il concetto dell'attenzione è quello di dare più importanza ad alcune parole rispetto ad altre in un contesto, per esempio in una frase. La potenza di questi trasformatori è che riescono a fare un'analisi del contesto molto più profonda rispetto ai modelli precedenti, permettendo di fare analisi di immagini come per esempio riconoscere oggetti in un'immagine o riconoscere dove è presente l'acqua all'interno di una foto.

1.2 Agenti intelligenti

Un agente intelligente è un'entità che percepisce il suo ambiente attraverso dei sensori e agisce su di esso attraverso degli attuatori.

- Percepisce l'ambiente attraverso dei **sensori**
- Agisce sull'ambiente attraverso degli **attuatori**
- Ha un **obiettivo** da raggiungere

Come dovrebbe comportarsi un agente intelligente?

- **Razionale**: agisce per massimizzare il raggiungimento dell'obiettivo
- **Performance measure**: misura di quanto bene l'agente sta raggiungendo l'obiettivo

Quando vogliamo ragionare sul Reinforcement Learning, è utilire usare il *Markov Decision Process*.

Definizione 1.2

Un **Markov Decision Process (MDP)** è una tupla (S, A, P, R) dove:

- S è un insieme di stati
- A è un insieme di azioni
- $P(s'|s, a)$ è la probabilità di transizione dallo stato s allo stato s' eseguendo l'azione a
- $R(s, a, s')$ è la ricompensa ottenuta eseguendo l'azione a nello stato s e transizionando nello stato s'

Poi si ha la *policy* che è una funzione che mappa uno stato in un'azione.

2 Risolvere problemi con la ricerca

2.1 Agents and enviroments

Gli agenti includono umani, robot, softbot, termostati, ecc. La funzione agente mappa la storia delle percezioni in azioni.

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

Il *programma dell'agente* viene eseguito su un'architettura fisica che produce f.

Esempio 2.1

Immaginiamo di avere un agente aspirapolvere che percepisce il luogo e i suoi contenuti.

- **Percezioni**: bump, Dirty e location (A o B)
- **Azioni**: left, right, suck, noOp

un esempio di sequenza percepita potrebbe essere:

$$(A, Dirty), Suck, (A, Dirty), Suck, (A, Clean), Right$$

$$(B, Dirty), Suck, (B, Clean), Left, (A, Clean), NoOp$$

Cosa fa la funzione *Right*? Può essere implementata in un piccolo programma agente? Se un agentes ha $|\mathcal{P}|$ possibili percezioni, quante "entries" avrà la tabella della funzione agente dopo T time steps?

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

L'obiettivo dell'IA è quello di progettare **piccoli** programmi agenti che permettono di rappresentare grandi funzioni agenti.

```
function Reflex-Vacuum-Agent([location,status]) returns an action
    if status = dirty then return suck
    else if location = A then return right
    else if location = B then return left
```

2.1.1 Multi-robot Patrolling

Esempio 2.2

Considerate il seguente ambiente:

- Tre stanze (A,B,C) e due robot (r_1, r_2)
- r_1 può pattugliare A e B, r_2 può pattugliare B e C
- r_1 inizia da A e r_2 inizia da C
- Il tempo di viaggio tra le stanze è 0
- Performance Measure: minimizzare il tempo medio di inattività tra le stanze
- Media di inattività: somma degli intervalli nella quale la stanza non è stata visitata da nessun robot

- Quale potrebbe essere un comportamento razionale di questo ambiente?

Quello che succede in maniera ragionevole è la seguente, dove S è la tupla in cui i robot sono posizionati: TODO
Nei diversi casi si ha che il miglior modo per fare girare i robot è quello di farli muovere alternando chi entra nella stanza B minimizzando anche la varianza nelle varie stanze perché dobbiamo stare attenti a non penalizzare troppo una stanza.

2.2 Tipi di ambiente

Il tipo di ambiente determina la progettazione di un agente? Nel mondo reale è ovviamente parzialmente visibile, stocastico, sequenziale, dinamico, continuo, multi-agente.

- **Completamente osservabile vs parzialmente osservabile**: un agente ha accesso completo allo stato dell'ambiente in ogni istante di tempo?
- **Deterministico vs stocastico**: il prossimo stato dell'ambiente è completamente determinato dallo stato corrente e dall'azione eseguita dall'agente?
- **Episodico vs sequenziale**: l'esperienza dell'agente è divisa in episodi indipendenti?
- **Statico vs dinamico**: l'ambiente può cambiare mentre l'agente sta pensando?
- **Discreto vs continuo**: il numero di stati, percezioni e azioni è finito o infinito?
- **Singolo agente vs multi-agente**: l'agente agisce da solo o ci sono altri agenti che possono influenzare l'ambiente?

	Crosswords	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

- Se il problema è deterministico e completamente osservabile, è un **single-state problem**
- Se il problema non è osservabile, è un **conformant problem**
- Se il problema è non deterministico o parzialmente osservabile, è un **contingency problem**
- Quando non conosco lo spazio degli stati è un **exploration problem**

2.3 Problem Solving Agents

Una forma ristretta di agente generale sono i: **Goal Based Agent**

- Formula un goal e un problema partendo dallo stato corrente
- Cerco una soluzione a questo problema
- Eseguo la soluzione ignorando le percezioni

Notiamo che questo si chiama anche offline problem; la soluzione viene eseguita ad "occhi chiusi".

```
function Simple-Problem-Solving-Agent(percept) returns an action
    static: solution, state, problem, action
    state <- Update-State(state, percept)
    if seq is empty then
        goal <- Formulate-Goal(state)
        problem <- Formulate-Problem(state, goal)
        seq <- Search(problem)
    action <- First(seq)
    seq <- Rest(seq)
    return action
```

Esempio 2.3 (Vacanze in Romania)

In viaggio in Romania, se attulamente ad Arad. Il viaggio parte domani da Bucharest.

- **Formulate Goal:** essere a Bucharest
- **Formulate Problem:** stati: varie città, azioni: guidare tra le città
- **Search:** trovare una sequenza di azioni che portano da Arad a Bucharest
- **Esempio di Soluzione:** Arad, Sibiu, Fagaras, Bucharest

2.3.1 Tree Search Algorithm

Idea base: offline, esplorazione simulata di spazio di stati, generando successori di stati già esplorati.

```
function Tree-Search(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        node <- Pop an element from the frontier
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
```

```
        expand node, adding the resulting nodes to the frontier
    end
```

Definizione 2.1

Uno **stato** è una rappresentazione di una configurazione fisica.

Definizione 2.2

Un **nodo** è una struttura dati che contiene:

- uno stato
- un puntatore al nodo genitore
- l'azione che ha generato lo stato
- il costo del cammino dal nodo radice a questo nodo

Gli stati non hanno parenti, azioni, figli, costi e profondità!

```
function Expand(node, problem) returns a set of nodes
    successors <- an empty list
    for each action in problem.ACTIONS(node.STATE) do
        child <- CHILD-NODE(problem, node, action)
        add child to successors
    return successors
```

2.4 Strategie di ricerca

Una strategia è definita dal scegliere l'ordine dei nodi di espansione. Strategie vengono valutate insieme alle seguenti metriche:

- Completezza: la strategia trova una soluzione se esiste
- Tempo: tempo di esecuzione della strategia
- Spazio: memoria usata dalla strategia
- Optimalità: la strategia trova la soluzione ottima?

Tempo e spazio sono misurati in termini di:

- b branching factor (numero massimo di figli per nodo)
- d profondità della soluzione più superficiale
- m profondità massima dell'albero di ricerca (potrebbe essere infinito)

2.4.1 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:

```
function Graph-Search( problem, frontier) returns a solution, or failure
  explored <- an empty set
  frontier <- Insert(Make-Node(problem.Initial-State))
  while not IsEmty(frontier) do
    node <- Pop(frontier)
    if problem.Goal-Test(node.State) then return node
    if node.State is not in explored then
      add node.State to explored
      frontier <- InsertAll(Expand(node, problem))
    end if
  end loop
  return failure
```

2.5 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

2.5.1 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.

```
function BFS( problem) returns a solution, or failure
  node <- node with State=problem.Initial-State,Path-Cost=0
  if problem.Goal-Test(node.State) then return node
  explored <- empty set frontier <- FIFO queue with node as the only element
  loop do
    if frontier is empty then return failure
    node <- Pop(frontier)
    add node.State to explored
    for each action in problem.Actions(node.State) do
```

```
      child <- Child-Node(problem,node,action)
      if child.State is not in (explored or frontier) then
        if problem.Goal-Test(child.State) then return child
        frontier <- Insert(child)
      end if
    end for
  end loop
```

Questo tipo di ricerca è:

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

2.5.2 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 $\geq \varepsilon$ (con piccola ma $\varepsilon > 0$)
- **Complessità di tempo**: Numero di nodi $g \leq$ del costo del percorso ottimale C^*. $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- **Complessità di spazio**: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- **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

2.5.3 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

2.5.4 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.

```
# Depth-Limited Search
function DLS(problem, limit) returns soln/fail/cutoff
  R-DLS(Make-Node(problem.Initial-State), problem, limit)


function R-DLS(node, problem, limit) returns soln/fail/cutoff
  if problem.Goal-Test(node.State) then return node
  else if limit = 0 then return cutoff # raggiunta la profondita' massima
  else
    # flag: c'e' stato un cutoff in uno dei sottoalberi?
    cutoff-occurred? <- false
    for each action in problem.Actions(node.State) do
      child <- Child-Node(problem, node, action)
      result <- R-DLS(child, problem, limit-1)
      if result = cutoff then cutoff-occurred? <- true
      else if result 6 = failure then return result
    end for
    if cutoff-occurred? then return cutoff else return failure
  end else

# Iterative Deepening Search
function IDS(problem) returns a solution
  inputs: problem, a problem
  for depth <- 0 to infinity do
    result <- DLS(problem, depth)
    if result 6 = cutoff then return result
  end
```

Questo tipo di ricerca è:

- **Completa**: Sì
- **Complessità di tempo**: $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- **Complessità di spazio**: $O(bd)$
- **Ottimale**: Sì, se il costo delle azioni è uniforme

Esempio 2.4

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?

Esempio 2.5

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:

$$< W, S, C, M, B >$$

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:

$$< 0, 0, 0, 0, 0 >$$

Lo stato obiettivo è:

$$< 1, 1, 1, 1, 1 >$$

Le azioni possibili sono:

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

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

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.

2.6 Ricerca informata

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

2.6.1 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*

2.6.2 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 2.6

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

2.6.3 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'obbiettivo, 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 2.6.1

Per A* l'euristica ammissibile implica l'ottimalità

Questo tipo di ricerca è:

- **Completa**: Sì, tranne se ci sono nodi infiniti con $f \leq f(G)$
- **Complessità di tempo**: Esponenziale in errore relativo in $h\times$ lunghezza del numeo 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, se l'euristica è ammissibile e consistente

2.7 Ricerca locale

In molte problemi di ottimizzazione il "path" è irrilevante, il traguardo è importante. In questi casi, allora lo spazio degli stati è un insieme di configurazioni:

- Trovare la configurazione ottimale (TSP (Travelling Salesperson Problem), etc...)
- Trovdare una configurazione che soddisfi dei vincoli (n-Queens, per esempio, dove ci sono 8 regine su una scacchiera e per trovare la configurazione dove nessuna delle 8 è sotto attacco, parto da una configurazione "base" e sposto le regine finché non trovo la configurazione traguardo, etc...)

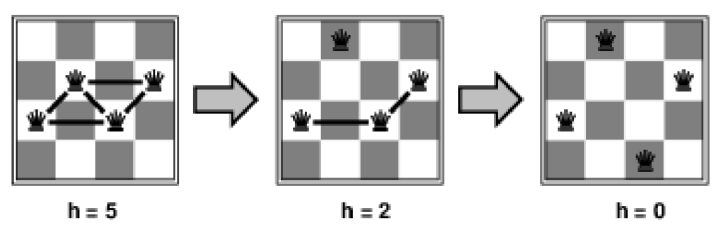
Si possono usare algoritmi di "iterative improvement":

- Mantenere un singolo stato corrente
- Cercare di migliorarlo

Spazio costante, fatto apposta per online e offline search. Varianti di questo approccio arrivano fino a 1% di soluzione ottimali.

Esempio 2.7 (Problema delle n regine)

- Inserire n regine su una scacchiera $n \times n$ in modo che nessuna regina possa attaccarne un'altra (quindi due regine non devono essere sulla stessa riga, colonna o diagonale).
- Muovi una regina per volta, cercando di ridurre il numero di conflitti.

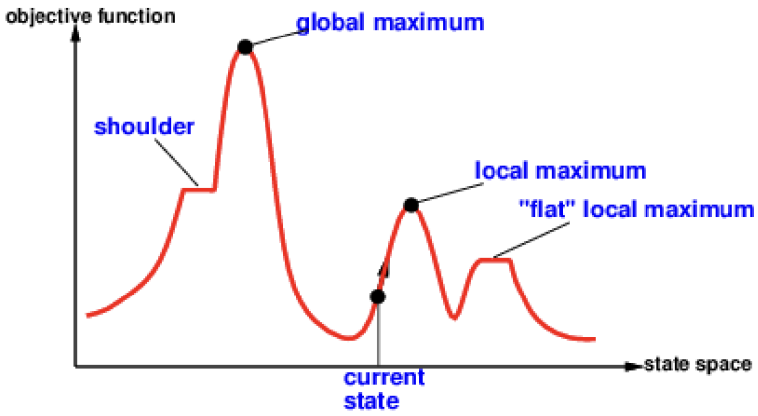


Quasi sempre si solve una problema di questo tipo in pochi passi, anche per $n = 1$ milione.

Ecco ora l'algoritmo di "hill-climbing" (come scalare il monte everest in una fitta nebbia con amnesia):

```
function Hill-Climbing(problem) returns a state that is a local maximum
    inputs:  problem, a problem
    local variables: current, a node
                     neighbor, a node
    current <- MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor <- a highest-value neighbor of current
        if neighbor.VALUE <= current.VALUE then return
          current.STATE
        current <- neighbor
```

Utile per considerare lo *state scape landscape*:



Ci sono varianti di questo algoritmo:

- **Random-restart hill climbing** è una variante che supera il massimo locale, trivialmente completo.
- **Random sideways moves** è buono perché esce dalle *shoulder* ma non completamente perché può rimanere bloccato in un ciclo infinito su ”flat local maxima”.

2.7.1 Simulated annealing

Simulated annealing è un algoritmo di ottimizzazione ispirato al processo di raffreddamento dei metalli. L'idea è di permettere occasionalmente mosse che peggiorano la soluzione corrente per evitare di rimanere bloccati in massimi locali.

- Inizia con una temperatura alta che permette molte mosse peggiorative
- La temperatura diminuisce gradualmente, riducendo la probabilità di accettare mosse peggiorative

- Alla fine, la temperatura raggiunge zero e l'algoritmo si comporta come hill-climbing
- La scelta della schedule di raffreddamento è cruciale per le prestazioni dell'algoritmo

```
function Simulated-Annealing(problem, schedule) returns a solution state
    inputs: problem, a problem
    schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps
    current <- Make-Node(problem.Initial-State)
    for t <- 1 to infinity do
      T <- schedule(t)
      if T = 0 then return current
      next <- a randomly selected successor of current
      deltaE <- next.Value - current.Value
      if deltaE > 0 then current <- next
      else current <- next only with probability e^{delta E/T}
```

A temperatura fissata T, la probabilità di accettare una mossa che peggiora la soluzione di ΔE è $e^{\Delta E/T}$.

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

Decrescendo T abbastanza, si può garantire la convergenza alla soluzione ottimale. Perché

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

2.7.2 Local beam search

Local Beam Search è un algoritmo di ricerca locale che mantiene k stati invece di uno solo. Inizia con k stati casuali e ad ogni iterazione:

- Genera tutti i successori di tutti i k stati correnti
- Seleziona randomicamente i k migliori successori tra tutti quelli generati
- Ripete fino a quando non viene trovata una soluzione o non ci sono più miglioramenti
- Se tutti i k stati convergono allo stesso punto, si può introdurre diversità sostituendo alcuni stati con nuovi stati casuali

2.8 Ricerca locale in spazio continuo

La ricerca locale può essere estesa a spazi di stato continui. Per risolvere questi problemi si possono utilizzare tecniche come:

- **Discretizzazione:** suddividere lo spazio continuo in una griglia di punti discreti e applicare algoritmi di ricerca locale su questi punti

- **Randomiche Perturbazioni:** introdurre piccole perturbazioni casuali alle soluzioni correnti per esplorare lo spazio delle soluzioni con metodi come il simulated annealing (il prossimo stato è scelto randomicamente)
- **Gradiente**: utilizzare il gradiente della funzione obiettivo per guidare la ricerca verso direzioni di miglioramento (il prossimo stato è scelto in base alla direzione del gradiente). Il metodo del gradiente calcola:

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

  Per trovare la direizione di massima crescita della funzione obiettivo si pone il gradiente uguale a zero:

$$\nabla f(x) = 0$$

A volte però non riusciamo a risolvere $\nabla f(x) = 0$ analiticamente, quindi possiamo migliorarla localmente:

- Si performa un update nella direzione della salita per ogni coordinata
- Più la funzione è ripida più si fanno passi grandi

Aggionrare una coordinata viene effettuato tramite una funzione generale $g(x_1, x_2)$

$$x_1 \leftarrow x_1 + \alpha \frac{\partial g(x_1, x_2)}{\partial x_1} \quad x_2 \leftarrow x_2 + \alpha \frac{\partial g(x_1, x_2)}{\partial x_2}$$

Oppure in forma vettoriale:

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad nablag(X) = \begin{bmatrix} \frac{\partial g(x_1, x_2)}{\partial x_1} \\ \frac{\partial g(x_1, x_2)}{\partial x_2} \end{bmatrix}$$

$$X \leftarrow X + \alpha \nabla g(X)$$

Dove α è lo step size, cioè la dimensione del passo da fare:

- Se è troppo grande si rischia di saltare soluzioni
- Se è troppo piccolo la convergenza sarà molto lenta

2.8.1 Algoritmo di Newton-Raphson

È una tecnica generale per trovare le radici di una funzione cioè risolvere un'equazione $f(x) = 0$. Per farlo si trova un'approsimazione iniziale $\bar{x}_0$ della soluzione e iterativamente si aggiorna l'approissimazione usando la formula:

$$\bar{x}_{n+1} = \bar{x}_n - \frac{f(\bar{x}_n)}{f'(\bar{x}_n)}$$

dove:

$$g'(x) = \frac{d}{dx} g(x)$$

Esempio 2.8

Consideriamo la funzione $f(x) = x^2 - a$.

- Mostrare che il metodo di Newton conduce a:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

- Fissato $a = 4, x_0 = 1$ cacolare le prime tre iterazioni. $(x_i = \{1, 2, 3\})$

Quindi abbiamo

$$f(x) = x^2 - a$$

$$x_{n+1} = \bar{x}_n - \frac{f(\bar{x}_n)}{f'(\bar{x_n})}$$

$$x_{n+1} = \bar{x}_n - \frac{\bar{x}_n - a}{2\bar{x}_n}$$

2.9 Constrained Satisfaction Problem

Un **Constrained Satisfaction Problem (CSP)** è un problema definito da:

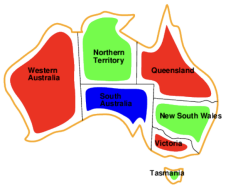
- Un insieme di variabili $X = \{X_1, X_2, \ldots, X_n\}$
- Un insieme di domini $D = \{D_1, D_2, \ldots, D_n\}$ dove ogni D_i è l'insieme dei valori possibili per la variabile X_i
- Un insieme di vincoli $C = \{C_1, C_2, \ldots, C_m\}$ che specificano le relazioni tra le variabili

Assunzioni: single agent, azioni deterministiche, stato completamente osservabile

Esempio 2.9 (Map-Coloring)

Il Map-coloring è un problema specifico di Graph coloring. Dato un insieme di regioni geografiche e un insieme di colori, assegnare un colore a ogni regione in modo che regioni adiacenti non abbiano lo stesso colore.

- Variabili: regioni geografiche (es. WA, NT, Q, NSW, V, SA, T)
- Domini: colori (es. rosso, verde, blu)
- Vincoli: regioni adiacenti non possono avere lo stesso colore



Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Esempio 2.10 (N-Queens Problem)

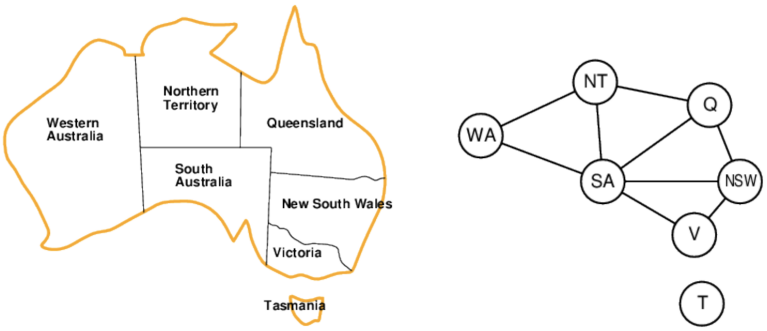
Il N-Queens Problem è un problema di posizionamento di N regine su una scacchiera $N \times N$ in modo che nessuna regina possa attaccarne un'altra.

- Variabili: posizioni delle regine sulle righe della scacchiera
- Domini: colonne della scacchiera (es. 1, 2, ..., N)
- Vincoli: questa volta formuliamo i vincoli in maniera più complessa. Un vincolo per ogni coppia di variabili specificando le posizioni "permesse" PER OGNI ogni due regine.

Questa formulazione rende alcuni vincoli impliciti, per esempio, non è possibile assegnare due regine la stessa colonna quindi non ci sta bisogno di controllare.

2.9.1 Grafo dei vincoli

Un **grafo dei vincoli** chiamato anche grafo primale consiste nel costruire un nodo per ogni variabile e un arco per ogni vincoli tra due variabili.



2.9.2 Ipergrafi e Grafi duali

Le relazioni tra ipergrafi e grafi binari:

- Si può sempre convertire un ipergrafo in un grafo binario
- Ogni variabile ha un dominio esponenzialmente grande

2.9.3 Problemi combinatori

Dato un insieme di possibili soluzioni bisogna trovare quella migliore che soddisfa i vincoli. Alcuni esempi:

- Decisionali: colorare un grafo con k colori
- Ottimizzazione: trovare la colorazione con il minor di conflitti
- Ottimizzazione Multi-obiettivo: portfolio investment, minimizzare il rischio e massimizzare il guadagno
- Modelli grafici:
  - Insieme di variabili, domini e funzioni locali (vincoli)
  - Funzioni globali è un aggregazione di funzioni locali
  - Soluzioni: l'assegnamento di variabili che ottimizza la funzione globale

Definizione 2.3: Rete a vincoli

Una tupla di tre elementi CN $= (X, D, C)$ dove:

- $X = \{X_1, X_2, \ldots, X_n\}$ è un insieme di variabili
- $D = \{D_1, D_2, \ldots, D_n\}$ è un insieme di domini associati alle variabili
- $C = \{C_1, C_2, \ldots, C_m\}$ è un insieme di vincoli che specificano le relazioni tra le variabili
- Ogni vincolo C_i è una tupla (S_i, R_i) dove:
  - $S_i \subseteq X$ è lo scopo, l'insieme delle variabili coinvolte in R_i
  - R_i sottoinsieme del prodotto cartesiano delle variabili in S_i
  - R_i specifica le tuple permesse su S_i
- **Soluzione:** assegnamento di tutte le variabili che soddisfano tutti i vincoli.
- Obiettivo: consistency check, trovare una o tutte le soluzioni, ottimizzare una funzione obiettivo.

Ci si può avvicinare alla soluzione attraverso una soluzione parziale:

- Soluzione parziale consistente: soluzione parziale che soddisfa tutti i vincoli di cui lo scope non contiene variabili non assegnate
- Una soluzione parziale consistente non è necessariamente estendibile a una soluzione completa

2.9.4 Tree Decomposition

Definizione 2.4: Cycle Cutset

Dato un grafo non orientato, un sottoinsieme di nodi nel grafo è un cycle cutset se la rimozione di questi nodi rende il grafo aciclico.

Il concetto è:

- Una volta che una variabile viene assegnata può essere rimossa dal grafo
- Se rimoviamo un cycle cutset allora il grafo rimanente è un albero
- Si può usare arc-consistency per risolvere l'albero rimanente
- Dobbiamo controllare ogni possibile assegnazione delle variabili del cycle cutset e fare propagazione negli archi
- La complessità è comunque esponenziale ma nella dimensione del cycle cutset.

3 Logical Agents

Perché costruire un agente basato sulla logica? Solitamente hanno due componenti:

- **Knowledge base (KB):** insieme di proposizioni che rappresentano ciò che l'agente sa sul mondo
- **Inference engine:** meccanismo per dedurre nuove proposizioni

La knowledge base è un insieme di proposizioni in un linguaggio formale. Un approccio dichiarativo per costruire un agente:

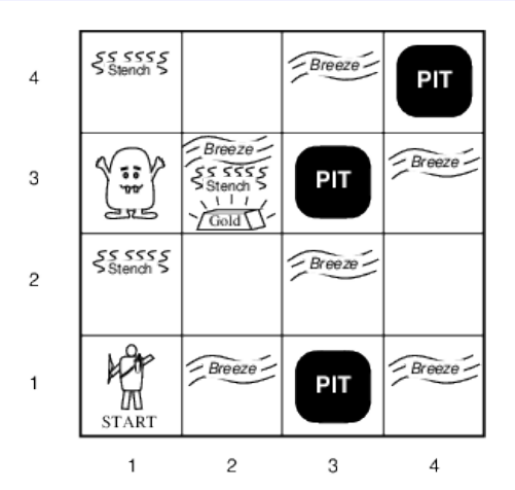
- Dire cosa sa l'agente
- Poi può chiedersi da solo cosa fare e le risposte dovrebbero seguire la knowledge base.

Gli agenti possono essere visti al livello di conoscenza i.e cosa sanno, non come sono implementati. O a livello di implementazione, cioè come sono costruiti.

```
function KB-Agent(percept) returns an action
    inputs: percept, a percept
    static: KB, a knowledge base, initially empty
            action, an action, initially null
            t, a counter initially 0
    KB <- Tell(KB, Percept-To-Sentence(percept))
    action <- Ask(KB, Action-Sentence())
    KB <- Tell(KB, Action-To-Sentence(action))
    t <- t + 1
    return action
```

Esempio 3.1 (Wumpus World PEAS)

Un esempio famoso di agente logico è l'agente Wumpus World.



- **Performance measure:** +1000 per uscita, -1 per ogni azione, -1000 per essere mangiati dal Wumpus o cadere in un buco
- **Environment:**
  - Celle adiacenti al wumpus sono maleodoranti
  - Celle adiacenti a un buco sono ventose
  - C'è scintillio se l'oro è nella stessa cella
  - Sparare uccide il wumpus se si è rivolti verso di lui
  - Sparare consuma la sola freccia
  - Prendere raccoglie l'oro se si è nella stessa cella

- Rilasciare lascia cadere l'oro nella stessa cella

- **Actuators:** muovi, gira a sinistra, gira a destra, spara, prendi oro, esci
- **Sensors:** brivido, puzza, scintilla, bump, urlo, sensore di glitter, sensore di impatto

Questo problema è:

- Osservabile? No—solo percezione locale
- Deterministico? Sì—i risultati sono esattamente specificati
- Episodico? No—sequenziale a livello di azioni
- Statico? Sì—Wumpus e buchi non si muovono
- Discreto? Sì
- Single-agent? Sì—Wumpus è essenzialmente una caratteristica naturale

3.1 Logica in generale

La logiche sono linguaggi formali per rappresentare informazioni come per trarre conclusioni. Come sappiamo la logica è divisa in:

- **Sintassi:** definisce le frasi nel linguaggio
- **Semantica:** definisce il "significato" delle frasi; cioè definisce la verità di una frase in un mondo

Un esempio è il linguaggio dell'aritmetica:

- $x + 2 \geq y$ è una frase; $x2 + y >$ non è una frase
- $x + 2 \geq y$ è vera se il numero $x + 2$ non è minore del numero y
- $x + 2 \geq y$ è vera in un mondo dove $x = 7, y = 1$
- $x + 2 \geq y$ è falsa in un mondo dove $x = 0, y = 6$

3.1.1 Entailment

L'entailment è una relazione tra frasi in un linguaggio logico e ha a che fare con i modelli non con la prova formale.

$$KB \models \alpha$$

Knowledge base KB entaila la frase α se e solo se *alpha* è vera in ogni mondo dove KB è vera.

Esempio 3.2

Per esempio se KB contiene "La Juventus ha vinto" e "Roma ha vinto" allora KB entaila "O la Juventus o Roma ha vinto". Oppure se $x + y = 4$ allora $4 = x + y$. Entailmente è una relazione tra frasi (cioè sintassi) basata sulla semantica. I computer sono molto bravi a processare regole sintattiche.

3.1.2 Modelli

I logici solitamente ragionano in termini di modelli, che sono formalmente strutturati in mondi rispetto alla verità che deve essere valutata. Diciamo che m è un modello per una frase *alpha* se *alpha* è vera in m. $M(\alpha)$ è l'insieme di tutti i modelli per *alpha*. Allora $KB \models \alpha$ se e solo se $M(KB) \subseteq M(\alpha)$.

Esempio 3.3

Prendendo l'esempio di prima se KB contiene "La Juventus ha vinto e la Roma ha vinto" allora α può essere "La Juventus ha vinto".

3.2 Inferenza

$KB \vdash_i \alpha$ vuol dire che α può essere derivata da KB con una procedura i. Le conseguenze di KB sono un pagliaio; α è un ago. Entailmente = ago nel pagliaio; inferenza = trovarlo.

- **Soundness:** i è corretto (sound) se dove $KB \vdash_i \alpha$, è anche vero che $KB \models \alpha$
- **Completeness:** i è completo se dove $KB \models \alpha$, è anche vero che $KB \vdash_i \alpha$

Preview: la logica del primo ordine è abbastanza espressiva da poter dire quasi tutto ciò che ci interessa, ed esiste una procedura di inferenza corretta e completa. Quindi, la procedura risponderà a qualsiasi domanda la cui risposta segua da ciò che è noto dalla KB.

3.3 Logica proposizionale

3.3.1 Sintassi

La logica proposizionale è il linguaggio logico più semplice. I simboli P_1, P_2 sono proposizioni atomiche.

- Se S è una proposizione, allora $\neg S$ è una proposizione
- Se S_1 e S_2 sono proposizioni, allora $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \Rightarrow S_2$, $S_1 \Leftrightarrow S_2$ sono proposizioni

3.3.2 Semantica

Ogni modello specifica se qualcosa è vero o falso per ogni simbolo proposizionale. Le regole per valutare rispetto ad un modello M sono:

- $\neq S$ è vero in M se e solo se S è falso in M
- $S_1 \wedge S_2$ è vero in M se e solo se sia S_1 che S_2 sono veri in M
- $S_1 \vee S_2$ è vero in M se e solo se almeno uno tra S_1 e S_2 è vero in M
- $S_1 \Rightarrow S_2$ è vero in M se e solo se S_1 è falso in M o S_2 è vero in M
- $S_1 \Longleftrightarrow S_2$ è vero in M se e solo se entrambi sono veri in M

Esempio 3.4 (Wumpus World Sentences)

Consideriamo $P_{i,j}$ sia vero se ci sta un pozzo nella cella (i, j) e $B_{i,j}$ sia vero se c'è del vento nella cella (i, j).

- $R_1 : \neg P_{1,1}$ (non c'è un pozzo nella cella (1,1))
- $R_2 : \neg B_{1,1}$
- $R_3 : B_{1,2}$

"Il pozzo è in una cella adiacente alla cella con vento":

$$R_4 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

$$R_5 : B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{1,3} \vee P_{2,2})$$

Una cella è ventosa **se e solo se** c'è un pozzo in una cella adiacente.

```
function TT-Entails(KB,a) returns true or false
  inputs: KB, the knowledge base, a sentence in prop. logic
  a, the query, a sentence in prop. logic
  symbols <- a list of the proposition symbols in KB and a
  return TT-Check-All(KB,a,symbols,[])
```