Intelligenza Artificiale

Università di Verona
Imbriani Paolo -VR500437
Professor Alessandro Farinelli

October 23, 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	$M = B, M = W, S \neq C$	$\langle W, S, C, M, B \rangle \mapsto \langle \bar{W}, S, C, \bar{M}, \bar{B} \rangle$
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.

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...)

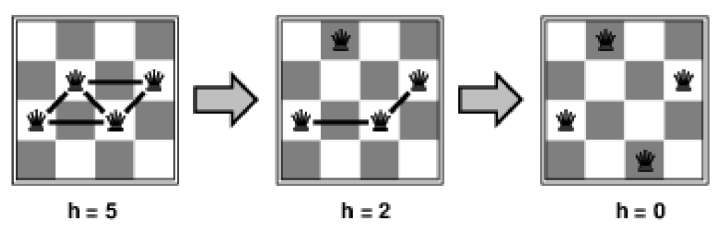
Si possono usare algoritmi di "iterative improvement":

- 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.

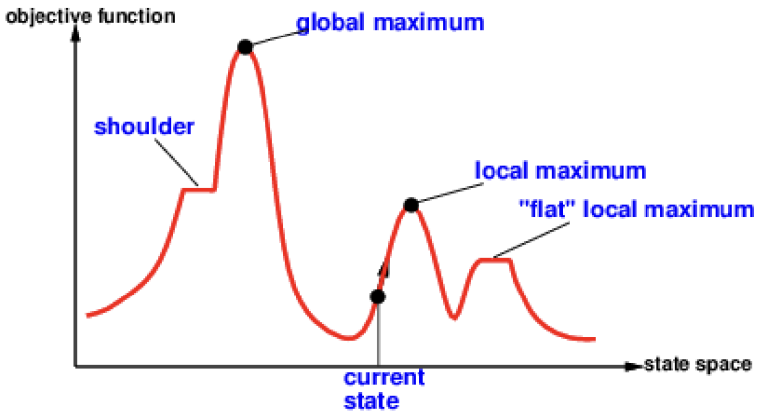


Quasi sempre si solve una problema di questo tipo in pochi passi, anche per $n = 1$ milione.

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