Intelligenza Artificiale

2024-2025

L'intelligenza artificiale è una disciplina ampia con diverse tecniche e diversi obiettivi.

Ragionamento: risolvere problemi nuovi sfruttando conoscenze. Intelligenza $\neq$ personalità.

L'intelligenza artificiale è una branca dell'informatica che progetta sistemi con capacità generalmente considerate come riservate all'uomo. È scienza (studio teorico, anche comparativo con l'intelligenza umana) e ingegneria (produce applicazioni concrete utili).

Le intelligenze artificiali ottenute tramite *machine learning* sono diventate così complesse da essere studiate come *black-box*, tecnica generalmente riservata ai fenomeni naturali.

Alan Turing dimostrò che è possibile costruire una macchina per ogni algoritmo definito. Ideò anche il test di Turing per intelligenze artificiali nel 1950. Neumann, oltre a definire la famosa architettura, fu il primo a teorizzare programmi autoreplicanti e capaci di evolversi come gli organismi naturali.

La *cibernetica* confronta i sistemi artificiali agli animali.

Nel 1943 McCulloch e Pitts creano il primo modello matematico di *neuroni artificiali*. Nel 1949 Hebb teorizza l'apprendimento autonomo. Solo di recente questi concetti hanno iniziato ad avere utilità pratica grazie all'aumento di disponibilità di dati e potenza di calcolo.

Il termine *intelligenza artificiale* viene definito nel 1956 a Dartmouth da McCarthy, Minsky, Rochester, Shannon ed Elwood. Nascono due paradigmi:

- paradigma di simulazione
- paradigma prestazionale o di emulazione

I primi programmi AI creati dal team soffrono di problemi di scalabilità. Viene per la prima volta riconosciuto il problema della necessità di conoscenza.

Dendral (1969) è il primo esempio di *sistema esperto* per la classificazione di sostanze chimiche basato sui dati spettrometrici. I sistemi esperti sono agenti basati sulle conoscenze e sono costituiti da regole codificate manualmente da esperti del settore.

I sistemi esperti sono esempi di *intelligenza artificiale simbolica*. Sono costruiti con oggetti astratti (*simboli*) e regole logiche.

Un agente opera in maniera autonoma, percependo l'ambiente, adattandosi ai cambiamenti e perseguendo obiettivi. La definizione di ambiente è flessibile e si può riferire anche ad un ambiente completamente virtuale, come l'interno di un computer.

Un esempio di regola è la seguente:

> Un professore può lavorare per un solo ateneo per volta.

In termini logici:

$$\text{Professore}(x) \land \text{LavoraPer}(x, y) \land \text{LavoraPer}(x, z) \land x \neq z \rightarrow \bot$$

Mettiamo di avere la seguente conoscenza pregressa (*stato*):

$$KB = \{\text{Professore}(P),\ \text{LavoraPer}(P, U1)\}$$

e di ricevere una nuova informazione:

$$I = \{\text{LavoraPer}(P, U2)\}$$

Possibili approcci:

- cambiare KB per rispettare la regola in funzione di I, due mutuazioni:
  - P non è più un professore
  - P si è licenziato da $U1$ per lavorare presso $U2$
- ignorare I a favore della conoscenza pregressa in KB

Se non è definito in modo chiaro il percorso da prendere, il sistema si blocca. Gli umani, invece, sono adattati a lavorare con stati inconsistenti e incerti senza bloccarsi perché hanno una conoscenza probabilistica della realtà.

Il ragionamento serve per mantenere lo stato consistente dopo nuove percezioni, e per decidere azioni verso un obiettivo.

Gli agenti hanno un ambito limitato e specifico. Esempi storici includono MYCIN per la diagnostica del sangue, con metriche probabilistiche di incertezza, SHRDLU per il linguaggio naturale, o il sistema LUNAR per le rocce lunari.

Il 1979 vede la nascita delle scienze cognitive. Nel 1981 il Giappone lancia il piano di investimenti Fifth Generation. Negli anni '80 Xerox, Texas Instruments e Symbolics realizzano workstation per il linguaggio LISP. Nel 1986 quattro diversi gruppi indipendenti riscoprono l'algoritmo di retropropagazione. Nel 1988 viene inventato l'apprendimento probabilistico. Gli anni 2000 vedono la nascita del *big data*, grandi dataset raccolti tramite Internet. Il 2011 marca l'apertura dell'epoca del *deep learning*.

L'AI tradizionale (simbolica) ha una visione algoritmica e deduttiva sia dei sistemi artificiali che della mente umana, mentre l'approccio moderno è induttivo e basato su processi incrementali. Negli anni recenti si afferma l'idea che i sistemi migliori nascano dall'unione di sistemi simboli e sistemi ad apprendimento automatico, con un'unione di informazione e semantica.

La questione di fondo rimane intoccata: come rappresentare la conoscenza interna? Quali processi razionali rappresentare?

Si sviluppano metodi e teorie: rappresentazione della conoscenza, ragionamento automatico, pianficiazione... Al contempo nascono soluzioni pratiche: computer vision, programmazione in linguaggio naturale, sistemi esperti e modelli.

Agenti razionali

Concetti legati alle percezioni:

- *percezione*: dati raccolti dai sensori dell'agente
- *sequenza percettiva*: storia completa delle percezioni dell'agente

Il comportamento dell'agente dipende dalla *conoscenza integrata* e dall'intera sequenza percettiva.

La *funzione agente* è l'astrazione utilizzata per rappresentare il comportamento dell'agente, specificando l'azione scelta dall'agente in base alla sequenza percettiva. È implementata concretamente nel *programma agente*.

Un agente è razionale quando *fa la cosa giusta*. L'azione corretta è definita in modo consequenzialista: valutare il comportamento dell'agente in base alle conseguenze delle sue azioni. L'ambiente, in seguito alle azioni dell'agente, procede secondo una sequenza di stati. Se essa è *desiderabile*, il comportamento dell'agente è valutato come positivo.

Per misurare la desiderabilità è necessario definire una misura di prestazione. La razionalità diventa dunque dipendente da quattro fattori:

1. misura di prestazione (definisce il criterio di successo)
2. conoscenza pregressa dell'ambiente
3. azioni possibili
4. sequenza percettiva

L'agente è dunque razionale se, per ogni possibile sequenza percettiva, è un grado di scegliere un'azione che massimizzi il valore atteso della sua metrica di prestazione, date le informazioni contenute nella sequenza percettiva e ogni conoscenza ulteriore nota all'agente. La metrica può talvolta penalizzare la correttezza della scelta per velocizzare il tempo di decisione o l'uso di risorse (anche computazionali).

La raccolta di informazioni (information gathering) è importante nel determinare la razionalità dell'agente. Un esempio è l'esplorazione dell'ambiente. L'agente deve poi essere in grado di apprendere qualcosa dalle informazioni raccolte. Un agente con questa capacità (di apprendimento) è detto *autonomo*, ovvero in grado di aggiornare le sue conoscenze iniziali sulla base dell'informazione raccolta.

Il programma agente è eseguito su un *architettura agente*, il sistema fisico dell'agente. L'agente è dato dall'unione del programma e dell'architettura.

Programma agente

Il programma agente ha sempre input e output come parte della propria struttura base. Se l'agente si basa sulla sequenza percettiva, anche solo parzialmente, avrà bisogno di memorizzarla.

Esistono alcuni tipi base di programma agente:

- *agente reattivo semplice*
  - ignora la sequenza percettiva e si basa soltanto sulla percezione corrente
  - si basa su *regole condizione-azione* (`if-then-else`)
  - funziona solo se ambiente è *completamente osservabile*, ovvero abbastanza informativo da poter indurre l'azione corretta sulla base della sola percezione corrente
- *agente reattivo basato su modello*
  - ambiente non completamente osservabile, necessità di memorizzare stato
  - *modello di transizione* del mondo: conoscenza dell'evoluzione dell'ambiente, sia causata dall'agente che per cause esterne ad esso
  - *modello sensoriale*: come lo stato del mondo ha effetto sulle percezioni dell'agente (ad esempio mancanza di luce = no informazione visiva)
- *agente basato su obiettivi*
  - di fronte a diverse possibili azioni, la corretta è da scegliere secondo un *obiettivo* invece che sulla base di regole generali
  - informazione = stato + obiettivo
  - necessità di tecniche di ricerca e pianificazione
- *agente basato sull'utilità*
  - i semplici obiettivi possono essere insufficienti
  - utilità definita da funzione che associa un punteggio numerico al grado di soddisfazione legato agli stati raggiungibili con le azioni possibili

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

```
function Agente-Reattivo-Basato-Su-Modello(percezione)returns un'azione
    persistent: stato: la concezione corrente dello stato del mondo
                modello_transizione:
                modello_sensoriale:
                regole:
                azione:
    stato <- Aggiorna-Stato(stato, azione, percezione, modello_transizione, modello_sensoriale)
    regola <-Regola-Corrispondente(stato, regole)
    azione <- Regola.Azione
    return azione
```

Classificazione dell'ambiente:

- completamente o parzialmente osservabile
- deterministico (stato successivo = stato corrente + effetti delle azioni dell'agente) o non deterministico
- episodico (l'azione dell'agente influenza solo l'episodio corrente) o sequenziale (azioni correnti hanno effetto percepibile in istanti futuri)
- statico (influenzato solo dalle azioni dell'agente) o dinamico
- discreto o continuo (in base alla cardinalità del tempo e delle azioni)

Complessità degli algoritmi

Algoritmo: sequenza finita di operazioni non ambigue ed effettivamente calcolabili che, una volta eseguite, producono un risultato in una quantità di tempo finita.

Questo implica che serva una condizione d'uscita finita nei cicli contenuti nell'algoritmo.

Un algoritmo è indipendente dal modello di calcolo che lo concretizza.

Gli algoritmi devono avere due caratteristiche fondamentali:

- *correttezza*: produzione del risultato desiderato
- *efficienza*: minimizzazione del tempo di esecuzione e dell'occupazione di memoria

Si analizzano gli algoritmi e non i programmi, perché l'analisi teorica è più generica e affidabile di quella sperimentale. Permette di scegliere tra diverse soluzioni alternative e di scegliere l'implementazione più conveniente in base a fattori teorici.

Si rappresentano il tempo e lo spazio in funzione delle dimensioni del problema: $t(n)$, $s(n)$. Per evitare l'influsso di istruzioni spurie, fattori additivi e moltiplicativi, si esegue un'*analisi asintotica* dell'impiego di risorse degli algoritmi.

Le notazioni O e Ω rappresentano, rispettivamente, i limiti superiori e inferiori dell'andamento di una funzione. La notazione Θ rappresenta un limite stretto.

- $f(n) = O(g(n))$ se $\exists\ c > 0,\ n > 0\ /\ f(n) \leq cg(n)\ \forall n \geq 0$ (limite superiore asintotico)
- $f(n) = \Omega(g(n))$ se $\exists\ c > 0,\ n > 0\ /\ f(n) \geq cg(n)\ \forall n \geq 0$ (limite inferiore asintotico)
- $f(n) = O(g(n))$ se $\exists\ c_1 > 0,\ c_2 > 0,\ n > 0\ /\ c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq 0$ (limite stretto asintotico)

Avviene un abuso di notazione: sarebbe più corretto scrivere $f(n) \in O(g(n))$ perché si tratta di famiglie di funzioni.

La notazione O gode di proprietà transitiva. Valgono per esse alcune regole di semplificazione:

- *eliminazione*: $O(kg(n)) = O(g(n))$
- *somma*: $O(g_1(n)) + O(g_2(n)) = O(\max(g_1(n), g_2(n)))$
- *prodotto*: $O(g_1(n)) \cdot O(g_2(n)) = O(g_1(n)g_2(n))$

Le principali classi sono:

- $O(1)$: *costante*
- $O(\log(n))$: *logaritmico*
- $O(n)$: *lineare*
- $O(n\log(n))$: *pseudolineare*
- $O(n^2)$: *quadratica*
- $O(k^n)$: *esponenziale*
- $O(n^k)$: *polinomiale*

La complessità intrinseca di un problema è collegata ma non coincidente alla complessità di uno specifico algoritmo risolutivo. La soluzione trovata, infatti, potrebbe essere più complessa rispetto a quanto richiesto dal problema, ma non può mai essere inferiore. Formalmente:

> Un algoritmo ha complessità $O(f(n))$ se $\exists$ un algoritmo risolutivo con delimitazione superiore $O(f(n))$.
>
> Un algoritmo ha complessità $\Omega(f(n))$ se tutti gli algoritmi risolutivi hanno delimitazione inferiore $\Omega(f(n))$.

Se il problema ha delimitazione inferiore $\Omega(f(n))$ e consideriamo una soluzione $O(f(n))$, allora tale soluzione è ottimale a meno di costanti.

Problemi di decisione

I *problemi di decisione* ammettono una soluzione booleana determinata. Un *algoritmo di decisione* è un algoritmo che termina per ogni possibile input. Un problema di decisione si dice *decidibile* se esiste almeno un algoritmo di decisione per esso. Esistono problemi dimostrabilmente indecidibili.

I problemi si analizzano sulla base delle prestazioni dell'algoritmo migliore nel caso peggiore. Si fa riferimento al modello di calcolo astratto della macchina di Turing.

- TIME($f(n)$): famiglia dei problemi risolvibili in tempo $O(f(n))$
- SPACE($f(n)$): famiglia dei problemi risolvibili con occupazione di memoria $O(f(n))$

Classe di complessità: insieme dei problemi risolvibili al massimo in $C(n)$.

Un problema è *polinomiale nel tempo* (appartenente alla classe PTime o P) se esiste un algoritmo di classe $O(n^k)$ deterministico.

Un problema è *trattabile* se è possibile trovare agevolmente soluzioni per n grande.

Un problema è *non deterministico* / appartenente alla classe NPTime o NP se il suo algoritmo risolutivo migliore è esponenziale. Questo porta alla necessità di alberi di ricerca molto estesi e ramificati, o all'uso di una macchina di Turing non deterministica (non realmente esistente).

I problemi NP sono intrattabili. Un possibile modello che approssimi la macchina di Turing non deterministica consiste nel trovare soluzioni casuali la cui ottimalità possa essere verificata in tempo polinomiale (approccio *guess-and-check*).

Si ipotizza (ma non è ancora stato dimostrato) che $P \neq NP$.

Un'altra classe è $CoNP$, che include i problemi il cui complemento sia in NP. Il complemento consiste nel determinare se una data istanza ha risposta negativa. Una soluzione e quella del suo complemento possono essere diverse.

Si ipotizza (ma non è ancora stato dimostrato) che $CoNP \neq P \neq NP$.

Altre classi:

- PSpace (che può essere $NPTime$)

  - problemi considerati più difficili degli NP
  - risolvibili in spazio polinomiale da una macchina di Turing deterministica
- ExpTime: TIME(k^n) per macchina di Turing deterministica
- NExpTime: TIME(k^n) per macchina di Turing non deterministica

Classi inferiori a P:

- LogSpace vs NLogSpace: risolvibili in spazio logaritmico da macchine di Turing qualsiasi
  - si conta solo lo spazio di lavoro, non quello occupato dall'input
- $AC0$: $O(k)$ da n^k processori in parallelo
  - è la classe di complessità delle operazioni sui database

Agenti risolutori di problemi e ricerca

Quando non è immediatamente evidente l'azione corretta da compiere, all'agente diventa necessario considerare un *cammino* (sequenza di azioni) che colleghino tramite passaggi il suo stato corrente alla risoluzione del problema.

Quest'operazione si chiama *ricerca*.

Formulazione: *problemi di ricerca nello spazio degli stati*.

Si definisce *spazio degli stati* l'insieme di tutti gli stati raggiungibili a partire dallo stato iniziale mediante una qualsiasi sequenza di operatori.

Esso è caratterizzato da:

- stato iniziale (noto all'agente, ma non a priori)
- insieme di azioni possibile (operatore di successione tra stati o funzione successore $S(x)$)

Il processo risolutivo ha 4 fasi:

1. definizione dell'obiettivo
2. formulazione del problema: modello astratto della parte del mondo modificabile tramite le azioni
3. ricerca: l'agente simula sequenze di azioni per trovare quella corretta, o per determinare che non esista soluzione
4. esecuzione

Se l'ambiente è deterministico e il modello dell'agente è esatto, l'agente potrebbe spegnere le proprie percezioni e agire con la garanzia di raggiungere l'obiettivo, in *anello aperto*. Per avere maggiore sicurezza, anche di fronte a comportamento non deterministico o modello non perfetto, si agisce *in anello chiuso* usando le percezioni.

Definizione formale di *problema di ricerca*:

- *spazio degli stati*
- *stato iniziale*
- *stato obiettivo*
- *azioni possibili*: funzione Azioni(s) che restituisce l'insieme finito di azioni possibili nello stato s
- *modello di transizione*: funzione Risultato(s, a) che restituisce lo stato risultante dell'applicazione di a sullo stato s
- *funzione costo-azione*: Costo $-$ Azione(s, a, s') che determina il costo per raggiungere lo stato s' a partire da s applicando l'azione a

Una sequenza di azioni forma un *cammino*. Una *soluzione* è un cammino che porta dallo stato iniziale allo stato obiettivo. Assumiamo costi positivi per le azioni, e che il costo del cammino sia la somma dei costi delle azioni che lo compongono.

La soluzione ottima è quella di costo minimo tra le possibili.

La dimensione dello spazio degli stati, con n possibili elementi (agente compreso) per cella e k celle, è $n \times k^n$.

La ricerca di soluzioni avviene mantenendo ed estendendo un insieme di sequenze dette *soluzioni parziali*. Il processo di *ricerca* consiste nella determinazione di tale sequenza.

È possibile costruire un *albero di ricerca* che rappresenti le ramificazioni del processo e le soluzioni parziali. I nodi rappresentano gli stati e gli archi rappresentano le azioni. Durante l'esplorazione viene definita *frontiera* l'insieme dei nodi raggiunti ma non ancora espansi.

Un nodo dell'albero di ricerca contiene:

- lo stato a cui il nodo corrisponde
- un riferimento al proprio padre all'interno dell'albero
- l'azione applicata al padre per giungere al nodo stesso
- il costo totale del cammino dalla radice al nodo stesso

L'*algoritmo di ricerca* riceve in ingresso il problema di ricerca e dà in uscita la soluzione o un'indicazione di fallimento.

La ricerca ha un costo, che in alcuni casi può essere superiore rispetto a quello dell'esecuzione di un'azione non ottimale.

La frontiera è invece rappresentata tramite una coda con i seguenti metodi:

- un metodo booleano che indica se la coda è vuota
- un metodo di estrazione in testa (con rimozione)
- un metodo di visualizzazione del nodo in testa (senza rimozione)
- aggiunta di nodi

Possiamo definire vari tipi, in base alla politica di accesso:

- *con priorità*: elemento in uscita definito dinamicamente tramite una funzione valutazione
- *FIFO: first in, first out*
- *LIFO: last in, first out*, detta anche pila o *stack*

Per le frontiere esiste una strategia chiamata *best-first*: scegliere un nodo n a cui corrisponde il valore minimo di una funzione di valutazione $f(n)$.

La procedura si basa sui seguenti passaggi:

1. estrarre dalla frontiera il nodo n per il quale il valore $f(n)$ associato è il minimo
2. se lo stato rappresentato da n è uno stato obiettivo, restituire n
3. in caso contrario, applicare un'espansione per generare nodi figli
4. ogni nodo figlio di n viene aggiunto alla frontiera se:
    1. non è stato precedentemente raggiunto
    2. è stato raggiunto tramite un cammino di costo inferiore ai precedenti
5. l'algoritmo fallisce oppure porta a trovare un nodo che rappresenta un cammino fino ad un obiettivo

L'algoritmo ovviamente dipende dal tipo di $f(n)$ utilizzata.

Si osservi come esempio la seguente implementazione in pseudocodice:

```
function Ricerca-Best-First(problema, f) returns un nodo soluzione o fallimento
    nodo <- Nodo(Stato=problema.StatoIniziale)
    frontiera <- una coda con priorità ordinata in base a f, con nodo come elemento iniziale
    raggiunti <- una tabella di lookup, con un elemento con chiave problema.StatoIniziale e valore nodo
    while not Vuota?(frontiera)
        nodo <- Pop(frontiera)
        if nodo.Stato == problema.Obiettivo then return nodo
        for each figlio in Espandi(problema, nodo) do
            s <- figlio.Stato
            if s not in raggiunti or figlio.Costo-Cammino < raggiunti[s].Costo-Cammino then
                raggiunti[s] <- figlio
                aggiungi figlio a frontiera
    return fallimento
```

Innanzitutto si può notare che l'algoritmo restituisce un nodo anche se è fatto per cercare un cammino. Questo perché ripercorrendo la lista dei predecessori da un nodo si può ricostruire l'intero cammino con facilità.

```
function Espandi(problema, nodo) return un insieme di nodi
    s <- nodo.Stato
    successori <- emptyset
    for each azione in problema.Azioni(s) do
        s' <- problema.Risultato(s, azione)
        costo <- nodo.Costo-Cammino + problema.Costo-Azione(s, azione, s')
        successori.add(new Nodo(Stato=s', Padre=nodo, Azione=azione, Cost-Cammino=costo))
    return successori
```

Le prestazioni degli algoritmi di ricerca si valutano secondo quattro metriche:

1. *completezza*: capacità di trovare una soluzione quando questa esiste, e di riportare il fallimento in caso contrario
2. *ottimalità rispetto al costo*: capacità di trovare la soluzione ottimale tra le ammissibili
3. *complessità temporale*: tempo impiegato in funzione della complessità del problema
4. *complessità spaziale*: occupazione di memoria

I problemi di ricerca si possono dividere in due classi in base alla conoscenza posseduta dall'agente:

1. *ricerca non informata*: nessuna informazione sul numero di passi o sul costo del cammino

2. *ricerca informata (euristica)*: le conoscenze dell'agente permettono di avere preferenze (non deterministiche) nella scelta degli stati

Esercizio 4.1

Fornire una formulazione completa del seguente problema. Scegliere una formulazione che sia abbastanza precisa da essere implementata.

Ci sono sei scatole di vetro in fila, ognuna con una serratura. Ognuna delle prime cinque scatole contiene una chiave che sblocca la scatola successiva; l'ultima scatola contiene una banana. Tutte le scatole sono chiuse; l'agente possiede la chiave della prima scatola e vuole la banana.

- *stati*: Ci sono sei scatole di vetro in fila, ognuna con una serratura. Ognuna delle prime cinque scatole contiene una chiave che sblocca la scatola successiva; l'ultima scatola contiene una banana.
- *stato iniziale*: tutte le scatole sono chiuse; l'agente possiede la chiave per aprire la prima scatola.
- *azioni*: apri, prendi
- *modello di transizione*:
  - *apri*, apre la scatola se l'agente ha la chiave
  - *prendi*: prende la banana se la scatola è aperta
- *stati obiettivo*: l'agente ha preso la banana
- *costo azione*: —

4.2

C'è un pavimento formato da $n \times n$ piastrelle. Inizialmente ogni piastrella può essere dipinta o non dipinta. L'agente inizialmente si trova su una piastrella non dipinta. Si può dipingere solo la piastrella sotto di sé e ci si può spostare solo su una piastrella adiacente se questa è non dipinta. Si vuole dipingere l'intero pavimento.

- *stati*: pavimento formato da $n \times n$ piastrelle. Inizialmente ogni piastrella può essere dipinta o non dipinta
- *stato iniziale*: tutti gli stati con almeno una piastrella non dipinta; agente su piastrella non dipinta
- *azioni*: dipingi, avanti, indietro, destra, sinistra
- *modello di transizione*:
  - dipingi: cambia stato da non dipinta a dipinta per la piastrella su cui si trova l'agente
  - avanti, indietro, destra, sinistra: movimento verso caselle adiacenti non dipinte
- *stati obiettivo*: tutte le piastrelle dipinte
- *costo di azione*: ogni azione costa 1

Possibili problemi: incastrarsi tra piastrelle dipinte senza riuscire a completare il lavoro.

4.3

Un robot si trova dentro un labirinto e deve raggiungere la postazione di ricarica. Il robot può muoversi solo in avanti verso la direzione in cui è rivolto. Ogni volta che incontra un incrocio (un punto in cui è possibile prendere una direzione differente da quella attuale) o un muro si ferma. Inizialmente il robot si trova in un incrocio ed è rivolto verso nord. La postazione di ricarica si trova in un corridoio senza uscite.

1. fornire una formulazione completa del problema
2. indicare la dimensione dello spazio degli stati
3. nel descrivere il problema sono state fatte delle semplificiazioni. Indicare almeno 3 delle semplificazioni fatte

- *stati*: un labirinto di n incroci e m vicoli ciechi, alla fine di uno di essi si trova la postazione di ricarica. Il robot si trova sempre ad un incrocio o in fondo ad un vicolo cieco, ed è ruotato in una delle 4 possibili direzioni (i punti cardinali)
- *stato iniziale*: tutti gli stati in cui il robot è ad un incrocio ed è rivolto verso nord
- *azioni*: avanti, ruota
- *modello di transizione*:
- *stati obiettivo*: il robot è alla postazione di ricarica
- *costo di azione*:

In ogni stato il robot si trova ad uno degli n incroci o degli m vicoli ciechi, rivolto in una delle quattro direzioni:

$4 \times (n + m)$

Semplificazioni fatte:

1. robot rivolto in solo 4 direzioni
2. robot può capire se è incrocio o alla fine del vicolo cieco
3. il robot può percorrere qualunque distanza senza bisogno di ricaricarsi
4. nessun ostacolo o danneggiamento o imprevisto nei movimenti
5. capacità di riconoscere la stazione di ricarica e di collegarvisi

Strategie di ricerca non informata

Un algoritmo di ricerca non informata non riceve alcuna informazione relativa alla vicinanza degli stati all'obiettivo.

Quando tutte le azioni hanno lo stesso costo, una possibile strategia appropriata è la ricerca in ampiezza, livello per livello. La ricerca è sistematica e completa anche su spazi degli stati infiniti.