Intelligenza Artificiale

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

Ragionamento: risolvere problemi nuovi sfruttando conoscenze. Intelligenza ≠ 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.

- $\mathrm{TIME}(f(n))$: famiglia dei problemi risolvibili in tempo $O(f(n))$
- $\mathrm{SPACE}(f(n))$: 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
- $\mathrm{ExpTime}$: $\mathrm{TIME}(k^n)$ per macchina di Turing deterministica
- $\mathrm{NExpTime}$: $\mathrm{TIME}(k^n)$ per macchina di Turing non deterministica

Classi inferiori a P:

- $\mathrm{LogSpace}$ vs $\mathrm{NLogSpace}$: 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 $\mathrm{Azioni}(s)$ che restituisce l'insieme finito di azioni possibili nello stato s
- *modello di transizione*: funzione $\mathrm{Risultato}(s, a)$ che restituisce lo stato risultante dell'applicazione di a sullo stato s
- *funzione costo-azione*: $\mathrm{Costo} - \mathrm{Azione}(s, a, s')$ 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.

Pseudocodice dell'algoritmo di ricerca in ampiezza:

```
function Ricerca-In-Ampiezza(problema) returns un nodo soluzione o fallimento
    nodo <- NODO(problema.StatoIniziale)
    if nodo.Stato == problema.Obiettivo then return nodo
    frontiera <- una coda FIFO, con nodo come elemento iniziale
    raggiunti <- {problema.StatoIniziale}
    while not Vuota?(frontiera) do
        nodo <- Pop(frontiera)
        for each figlio in Espandi(problema, nodo) do
            s <- figlio.Stato
            if s == problema.Obiettivo then return figlio
            if s not in raggiunti then
                add s a raggiunti
                add figlio a frontiera
    return fallimento
```

Si noti che nell'algoritmo si confrontano gli *stati* e non i *nodi*. Questo perché si può giungere allo stesso nodo secondo diverse strade, con risultati anche radicalmente diversi.

Se il costo di ogni ramo è costante, le soluzioni "in alto" sono più efficienti di quelle equivalenti trovate più in basso, per via del costo minore. Per rami a costo variabile, il controllo deve essere effettuato istanza per istanza, ogni volta che si individua uno stato obiettivo.

La frontiera è implementata tramite coda FIFO. L'algoritmo fornisce una ricerca completa e ottimale.

Il numero di nodi generati, con un fattore di ramificazione b (b figli per ogni nodo), è:

$$1 + b + b^2 + b^3 + \ldots + b^d$$

per d livelli, ovvero l'algoritmo ha complessità spaziale (tutti i nodi restano in memoria) e temporale (tutti i nodi sono visitati esattamente una volta) $O(b^d)$.
Questa complessità, essendo esponenziale, inizia ad esplodere verso $d = 10$.

Ricerca a costo uniforme

Quando le azioni hanno costi diversi conviene utilizzare la ricerca best-first usando come metrica il costo del cammino dalla radice al nodo corrente. Questo porta alla formulazione dell'algoritmo di Dijkstra.

La ricerca in ampiezza si diffonde per ondate di ampiezza, mentre la ricerca a costo uniforme si muove a ondate di costo.

Usa una coda con priorità per implementare la frontiera. È *completa*, cioè trova una sola soluzione quando essa esiste. È anche *ottimale* perché, tra più soluzioni, le trova in ordine di costo, quindi giunge subito alla soluzione migliore.

Ricerca in profondità

Espande sempre il primo dei nodi fino a raggiungere il livello più profondo (foglia). Nodi di uguale profondità vengono selezionati arbitrariamente. Arrivata ad una foglia, la ricerca fa *backtracking* ed espande nodi a livelli più superficiali.

La funzione di ricerca utilizza una coda LIFO (*stack*) come meccanismo di inserimento.

L'algoritmo non è ottimo perché non impiega alcun criterio che favorisca soluzioni a costo minimo. Non è nemmeno completo perché potrebbe perdersi in un percorso di profondità infinita privo di soluzioni.

La complessità, dato il fattore di ramificazione b e massima profondità m è $O(b^m)$, esponenziale come la ricerca in ampiezza. La complessità spaziale è però polinomiale: $O(b \cdot m)$, molto inferiore alla ricerca in ampiezza.

Variante: *ricerca a profondità limitata*. Porre una profondità massima l per produrre un algoritmo di costo temporale $O(b^l)$ e spaziale $O(b \cdot l)$. Anch'esso è incompleto.

```
function Ricerca-Profondità-Limitata(problema, soglia) returns un nodo soluzione o
fallimento o soglia
    frontiera <- una coda LIFO con Nodo(problema.StatoIniziale) come elemento iniziale
    risultato <- fallimento
    while not Vuota?(frontiera) do
        nodo <- Pop(frontiera)
        if nodo.Stato == problema.Obiettivo then return nodo
        if Profondità(nodo) > soglia then
            risultato <- soglia
        else if not È-Ciclo(nodo) do
            for each figlio in Espandi(problema, nodo) do
                aggiungi figlio a frontiera
    return risultato
```

La ricerca in profondità, non conservando memoria dei nodi visitati, potrebbe rivisitare più volte i nodi membri di un ciclo. Si può parzialmente risolvere aggiungendo una procedura che controlla la presenza di cicli controllando i padri per k passi, numero arbitrario.

Ricerca ad approfondimento iterativo: provare la ricerca a profondità limitata iterativamente, con L crescente.

```
function Ricerca-Approfondimento-Iterativo(problema) returns un nodo soluzione o
fallimento
    for profondità = o to infty do
        risultato <- Ricerca-Profondità-Limitata(problema, profondità)
        if risultato != profondità then return risultato

            # se risultato è diverso dal valore di soglia allora è la soluzione
            oppure fallimento
```

L'algoritmo riunisce molti vantaggi sia della ricerca in profondità che in ampiezza.

- complessità temporale:
  - $O(b^d)$, se esiste soluzione
  - $O(b^m)$, nel caso peggiore
- complessità spaziale:
  - $O(b \cdot d)$, se esiste soluzione
  - $O(b \cdot m)$, nel caso peggiore

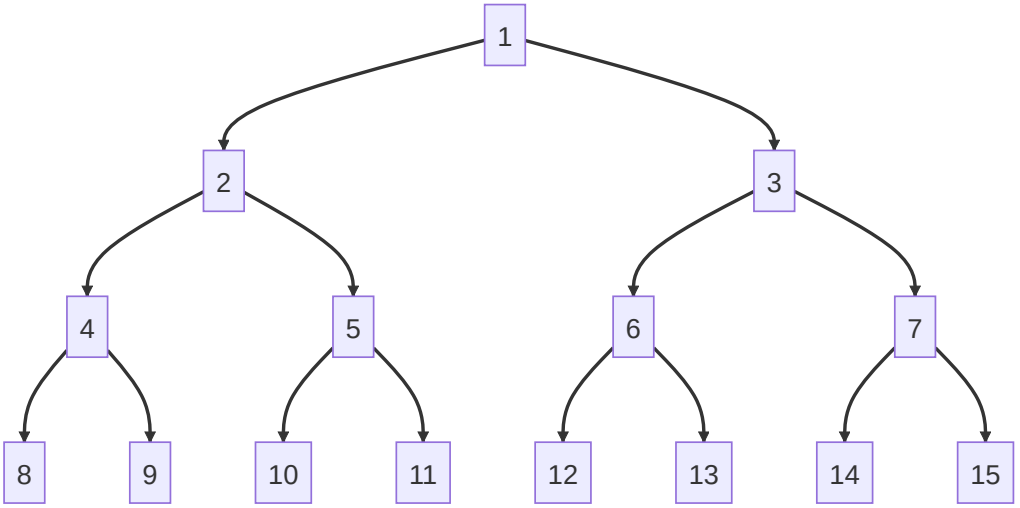
Esercizi

5.1

> Si consideri uno spazio degli stati in cui lo stato iniziale è indicato con 1 ed ogni stato k ha esattamente due stati successori denotati con $2k$ e $2k + 1$.

1. Disegnare la porzione dello spazio degli stati che contiene gli stati che vanno da 1 a 15
2. Supponendo che lo stato obiettivo sia 11, indicare l'ordine di visita nello spazio degli stati effettuata dalla ricerca in ampiezza
3. Supponendo che lo stato obiettivo sia 11 indicare l'ordine di visita dello spazio degli stati effettuata dalla ricerca a profondità limitata con $L = 3$
4. Supponendo che lo stato obiettivo sia 11, indicare l'ordine di visita dello spazio degli stati effettuata dalla ricerca a profondità iterativa limitata con $L = 3$.

1. Disegno:



2. Ordine di visita in ampiezza:

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11$$

3. Ordine di visita a profondità limitata:

$$1, 2, 4, 8, (4), 9, (4), 2, 5, 10, (5), 11$$

4. Ordine di visita iterativa a profondità limitata:

Iterazione 0:

Iterazione 1:

$$1, 2, (1), 3$$

Iterazione 2:

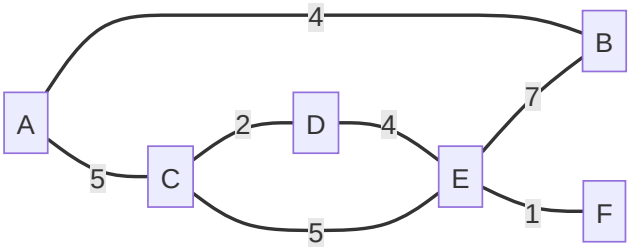
$$1, 2, 4, (2), 5, (2), (1), 3, 6, (3), 7$$

Iterazione 3:

$$1, 2, 4, 8, (4), 9, (4), (2), 5, 10, (5), 11$$

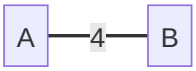
5.2

> Si consideri lo spazio degli stati rappresentato in figura. Supponiamo che un agente partendo dallo stato iniziale **A** debba raggiungere lo stato obiettivo **F** minimizzando il costo. Il costo di ogni azione per spostarsi da uno stato ad uno stato adiacente è indicato sull'arco che li unisce. Descrivere i passi indicati dalla ricerca uniforme ed indicare la soluzione individuata con il relativo costo.



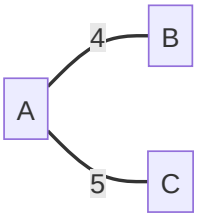
Passo 1:

- risolti: $\emptyset$
- frontiera: $\{B(4),\ C(5)\}$
- scelta: B a costo 4



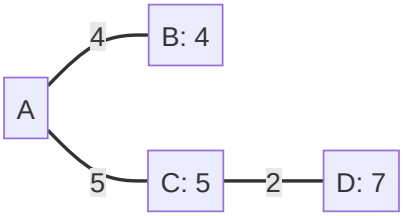
Passo 2:

- risolti: $\{B(4)\}$
- frontiera: $\{C(5),\ E(7+4=11)\}$
- scelta: C a costo 5



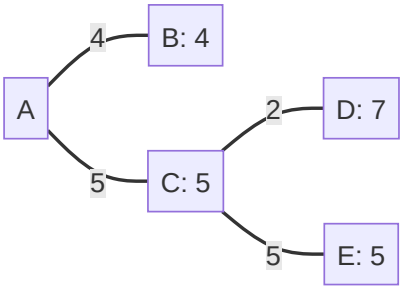
Passo 3:

- risolti: $\{B(4),\ C(5)\}$
- frontiera: $\{E(11),\ D(5+2=7),\ E(5+5=10)\}$
- scelta: D a costo 7



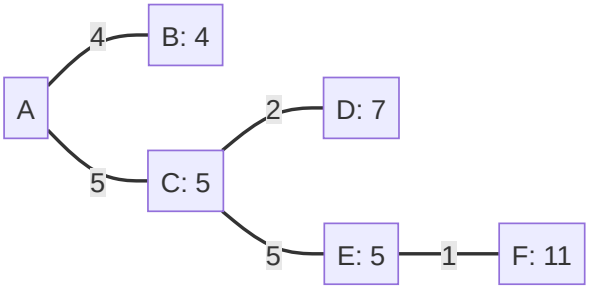
Passo 4:

- risolti: $\{B(4),\ C(5),\ D(7)\}$
- frontiera: $\{E(11),\ E(10)\}$
- scelta: E a costo 10



Passo 5:

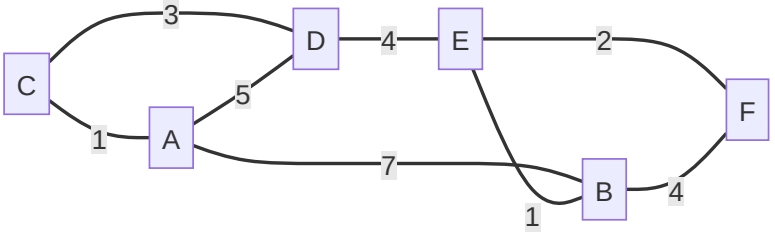
- risolti: $\{B(4),\ C(5),\ D(7),\ E(10)\}$
- frontiera: $\{F(10+1=11)\}$
- scelta: F a costo 11



$\Rightarrow$ soluzione migliore: $A,\ C,\ E,\ F$

5.3

Si consideri lo spazio degli stati rappresentato in figura. Supponiamo che un agente partendo dallo stato iniziale **A** debba raggiungere lo stato obiettivo **F** minimizzando il costo. Il costo di ogni azione per spostarsi da uno stato ad uno stato adiacente è indicato sull'arco che li unisce. Descrivere i passi indicati dalla ricerca uniforme ed indicare la soluzione individuata con relativo costo.



$$F[1] = \{A[0]\}$$

$$\{C[1], D[5], B[7]\}$$
$$\{D[4], B[7]\}$$
$$\{B[7], F[8]\}$$
$$\{E[8], F[11]\}$$
$$\{F[10]\}$$

$A - B - E - F$, costo complessivo 10

Ricerca informata

La ricerca non informata trova soluzioni, spesso inefficienti, generando sistematicamente nuovi stati che sono poi verificati rispetto all'obiettivo.

La ricerca informata utilizza invece euristiche per scegliere ad ogni passo i nodi più convenienti da esplorare, per arrivare ad una soluzione (non necessariamente ottima) in modo più efficiente.

> **Euristica**: *regola, strategia, semplificazione o qualunque altro strumento che drasticamente limita la ricerca di soluzioni nello spazio degli stati del problema.*
> *Le euristiche non garantiscono soluzioni ottimali; non garantiscono soluzioni in ogni caso. Una euristica utile è ciò che offre soluzioni che sono abbastanza buone il più delle volte.*
> Feigenbaum e Feldman, 1963

Negli algoritmi di ricerca l'euristica prende la forma di una funzione $h(n)$ che fornisce una stima del costo della soluzione. Nei sistemi esperti le euristiche sono regole empiriche definite dagli esperti del dominio. Se espresse come regole, si parla di *sistemi basati su regole*.

Generalmente le euristiche sono altamente specifiche e non adattabili a nuovi problemi.

Punti principali di applicazione dell'informazione euristica:

- determinazione dei nodi da espandere
- decisione dei successori da generare
- decisione dei nodi da scartare dall'albero di ricerca

Ricerca best-first *greedy*

Viene espanso per primo il nodo con il valore più basso di $h(n)$. I risultati sono potenzialmente peggiori rispetto alla ricerca esaustiva.

```
function Ricerca-Best-First-Greedy(problema) returns un nodo soluzione o fallimento
    return Ricerca-Best-First(problema, h(n))
```

La ricerca best-first greedy si comporta come una ricerca in profondità. Non è ottimale e può essere incompleta.

- *complessità temporale*: $O(b^d)$ con profondità d e ramificazione b nel caso peggiore
- *complessità spaziale*: $O(b^d)$ come quella temporale perché tiene in memoria tutti i nodi

Una buona euristica permette di ridurre sostanzialmente la complessitò.

Ricerca A^*

La ricerca A^* è un algoritmo di ricerca informata che invece di considerare solo la distanza dall'obiettivo, considera anche il *costo* nel raggiungere il nodo n dalla radice.

Si tratta di una ricerca best-first che utilizza la funzione di valutazione

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

dove $g(n)$ il costo del cammino dal nodo iniziale al nodo n e $h(n)$ rappresenta il costo stimato euristico del cammino più breve da n a uno stato obiettivo. Si ha dunque che $f(n)$ sia il costo stimato del cammino migliore che continua da n fino a un obiettivo.

La ricerca A^* è *completa* in spazi degli stati finiti. L'ottimalità rispetto al costo dipende all'euristica. Indicando con $h(n)$ la vera distanza tra n e l'obiettivo, si dice che l'euristica $h'(n)$ sia *ammissibile* se

$$\forall n h'(n) \leq h(n).$$

L'euristica perfetta ($h' = h$) e l'euristica banale ($h' = 0$) sono sempre ammissibili.

> **Teorema:** se $h'(n) \leq h(n)$ per ogni nodo, allora A^* troverà sempre il nodo obiettivo ottimale.

Dimostrazione:

- si supponga per assurdo che l'algoritmo abbia raggiunto lo stato obiettivo G_2 seguendo un percorso subottimale e che l'abbia inserito nella frontiera
- sia n un nodo non espanso nella frontiera tale da essere sul percorso più breve verso l'obiettivo ottimo G
  - allora $f(G_2) = g(G_2)$ perché $h'(G_2) = 0$ (G_2 è soluzione)
  - allora $g(G_2) > g(G)$ perché G_2 è subottimale
  - $f(G) = g(G)$ perché $h'(G) = 0$ (G è soluzione)
- quindi
  - $f(G_2) > f(G)$ da sopra
  - $h'(n) \leq h(n)$ perché h' è ammissibile
- quindi
  - $g(n) + h'(n) \leq g(n) + h(n)$
  - $g(n) + h(n) = f(n)$ per definizione
  - $g(n) \leq f(G)$ dato che n è sulla strada che porta a G e quindi
- $f(G_2) > f(n)$, e A^* non selezionerà mai G_2 per l'espansione (contraddizione)

L'algoritmo A^* può essere implementato come chiamata di `ricerca-best-first` con la funzione euristica $f(n) = g(n) + h(n)$ come funzione di valutazione:

```
function Ricerca-Best-First-Greedy(problema) returns un nodo soluzione o fallimento
    return Ricerca-Best-First(problema, f(n))
```

Esistono però anche altre implementazioni più efficienti.

La ricerca A^* è *ottimale* e *completa* se non esistano infinti nodi con n tale che $f(n) < g(G)$.

- *complessità temporale*: $O(b^d)$ nel caso peggiore, con profondità d e ramificazione b
- *complessità spaziale* $O(b^d)$ coincidente con la complessità temporale perché tutti i nodi sono tenuti in memoria

È possibile abbassare la complessità rinunciando all'ottimalità.

Ricerca in spazi complessi

Ricerca basata su euristiche più generiche rispetto a quelle della ricerca informata.

Algoritmo Simulated Annealing

Hill climbing combinato ad una scelta stocastica (non del tutto casuale).

Viene aggiunto un parametro, la temperatura T, che rappresenta il grado con cui si accettano scelte casuali anche se non migliorative.

Procedimento:

- si sceglie un successore a caso
- se migliora lo stato corrente, viene espanso
- se non lo migliora, viene potenzialmente scelto con probabilità $p = e^{\frac{\Delta E}{T}}$ con $o \leq p \leq 1$, dove $\Delta E = f(n') - f(n)$

La probabilità p è dunque inversamente proporzionale al peggioramento, per ΔE negativo. La temperatura T decresce secondo un piano definito con il progredire delle iterazioni.

La soluzione ottima è raggiungibile de T decresce abbastanza lentamente.

```
function Simulated-Annealing(problema, velocità_raffreddamento) returns uno stato
soluzione
    corrente <- problema.StatoIniziale
    for t <- 1 to infty do
        T <- velocità_raffreddamento[t]
        if T = 0 then return corrente
        successivo <- un successore di corrente scelto a caso
        DeltaE <- Valore(successivo) - Valore(corrente)
        if DeltaE > 0 then corrente <- successivo
        else corrente <- successivo solo con probabilità e^(DeltaE/T)
```

Algoritmi genetici

Si parte da una popolazione (generata casualmente) di stati di cui ognuno è denotato da una stringa rappresentativa della sua struttura. Ad ogni stato è legato un valore della funzione di *fitness*.

Tre principali operatori:

- *selezione*: selezionare gli individui che diventano genitori della generazione successiva (in base al valore di fitness)
- *mutazione*: un certo tasso di mutazioni casuali nelle generazioni figlie rispetto alla stretta ereditarietà dai genitori
- *ricombinazione*: parti di stato dei genitori vengono prese e messe nel codice dei figli

Procedimento:

- la popolazione è ordinata in base alla funzione di fitness, per produrre coppie di futuri genitori
- per ciascuna coppia viene effettuato un punto di cross-over; i figli vengono prodotti scambiando i pezzi dei genitori sul punto di cross-over
- viene effettuata una mutazione casuale sui figli

All'inizio del processo la popolazione è altamente diversificata. Con il procedere delle generazioni, la popolazione si stabilizza e le differenze di fitness si riducono.

```
function Riproduzione(genitore1, genitore2) returns un individuo
    n <-Lunghezza(genitore1)
    c <- numero casuale fra 1 e n
    return Concatena(Sottostringa(gentitore1,1,c), Sottostringa(gentitore2,c+1,n))
```

```
function Algoritmo-Genetico(popolazione, fitness) returns un individuo
    repeat
        pesi <- Pesato-Da(popolazione, fitness)
        popolazione2 <- lista vuota
        for i=1 to Dimensione(popolazione) do
            genitore1, genitore2 <- Selezione-Casuale-Pesata(popolazione, pesi, 2)
            figlio <- Riproduzione(genitore1, genitore2)
            if(piccola probabilità casuale) then figlio <- Mutazione(figlio)
            aggiungi figlio a popolazione2
        popolazione <- popolazione2
        until esiste in popolazione un individuo con fitness sufficientemente alto, o è
passato abbastanza tempo
        return l'individuo migliore nella popolazione in base al fitness
```

Gli algoritmi genetici funzionano al meglio in situazioni in cui sia possibile sviluppare schemi utili in una varietà maggiore di situazioni.

Problemi di soddisfacimento di vincoli

Un *problema di soddisfacimento di vincoli* (*CSP*, *Constraint Satisfaction Problem*) è risolto quando ogni variabile ha un valore che soddisfa tutti i vincoli su di essa.

Si introduce dunque una rappresentazione *fattorizzata* di ogni stato, parametrizzato mediante variabili, invece che fornire stati atomici.

Gli algoritmi di ricerca CSP sfruttano la struttura degli stati (ora nota) ed euristiche generali per individuare le soluzioni. L'idea base è eliminare le porzioni (ampie) dello spazio degli stati nelle quali i vincoli non sono rispettati. Il vantaggio principale è la possibilità di dedurre azioni e modello di transizione direttamente dalla descrizione del problema.

Formalmente un problema CSP è definito da tre componenti:

- X: insieme delle variabili $\{X_1, \dots, X_n\}$
- D: insieme dei domini di ciascuna variabile $\{D_1, \dots, D_n\}$
  - ogni dominio D_i contiene i valori $\{v_1, \dots, v_k\}$ ammissibili per la variabile X_i
- C: insieme dei vincoli
  - le relazioni tra variabili possono essere espresse in forma $\langle \text{ambito}, \text{rel} \rangle$, dove rel può essere espresso:
    - per elencazione di valori ammissibili
    - formalmente come condizione
  - per esempio, $X_1 > X_2$, per $D_{X_1} = D_{X_2} = \{1, 2, 3\}$, si può esprimere come:
    - $\langle (X_1, X_2) \{(3, 1), (3, 2), (2, 1)\} \rangle$
    - $\langle (X_1, X_2), X_1 > X_2 \rangle$
  - si noti che nella seconda formulazione l'ambito diventa superfluo

Una soluzione di un CSP è un assegnamento *completo* e *consistente* (o *legale*). È possibile trovare più soluzioni.

- *completo*: a tutte le variabili è assegnato un valore
- *consistente*: l'assegnamento non viola alcun vincolo
  Rappresentazione dell'assegnamento:

$$\{X_1 = v_1, X_2 = v_2, \dots\}$$

In caso di vincoli non esprimibili tramite funzione, come accade per i vincoli di relazione, può essere conveniente usare una rappresentazione a grafo.

I vincoli si dividono in *unari* (applicati ad una sola variabile) e *binari* (relativi alla relazione tra una coppia di variabili). I vincoli relativi ad un numero arbitrario di variabili superiore a 2 sono detti *globali*. Un esempio è il vincolo *TutteDiverse* che è un modo compatto per indicare che tutte le variabili debbano avere valori diversi tra loro.

Gli algoritmi CSP possono procedere in due diversi modi:

1. generare successori tramite nuovi assegnamenti di variabile
2. fare inferenza tramite *propagazione dei vincoli*, ovvero ridurre i valori legali per una variabile sperando che questo riduca i valori legali anche per altre variabili

La propagazione dei vincoli cerca di mantenere la *consistenza locale*. Questo si può ottenere in vari modi.

Una variabile è detta *nodo-consistente* se tutti i i valori del suo dominio soddisfano i suoi vincoli unari. Un grafo è detto *nodo-consistente* se ogni sua variabile è nodo-consistente.
Ridurre il dominio delle variabili con vincoli unari iniziali permette di eliminare tutti i vincoli unari successivi.

Una variabile è detta *arco-consistente* se ogni valore del suo dominio soddisfa i vincoli binari a cui partecipa. Formalmente:

> *Una variabile X_i è detta arco-consistente rispetto ad un'altra variabile X_j se per ogni valore nel dominio corrente D_I c'è almeno un valore nel dominio D_j che soddisfa il vincolo binario sull'arco (X_i, X_j).*

Un grafo è *arco-consistente* se per ogni arco (X_i, X_j) la variabile X_i è arco-consistente rispetto a X_j e viceversa.

AC-3 è l'algoritmo più noto per verificare la consistenza d'arco. Esso opera su grafi orientati, quindi è necessario di rendere orientati anche i grafi di vincoli che non lo sono. Solitamente si assegna il verso dal maggiore al minore per i vincoli di disuguaglianza, e si inserisce un contro-vincolo che rappresenta la relazione nel verso opposto.

```
algorimto AC-3(csp) returns false se viene trovata un'inconsistenza, o altrimenti true
    Arc <- un insieme di archi orientati, inizialmente tutti quelli nel csp
    while Arc non è vuota do
        (X_i, X_j) <- Estrai(Arc) # Estrai e rimuovi un arco da Arc
        if Rimuovi-Valori-Inconsistenti(csp, X_i, X_j) then
            if dimensione di D_i = 0 then return false
            for each X_k in X_j.Adiacenti do
                aggiungi (X_k, X_i) ad Arc
    return true
```

```
function Rimuovi-Valori-Inconsistenti(csp, X_i, X_j) returns true se e solo se viene
rimosso un valore
    rimosso <- false
    for each x in D_i do
        if nessun valore y in D_j permette a (x, y) di soddisfare il vincolo tra X_i e
X_j then
            rimuovi x da D_i
            rimosso <- true
    return rimosso
```

Si parte da un insieme di archi orientati Arc, contenente all'inizio tutti gli archi del problema. Ad ogni iterazione viene rimosso un arco orientato (X_i, X_j) (l'ordine non conta) e lo si rende arco consistente. Se il dominio D_i resta invariato, si passa all'arco successivo. In caso contrario, D_i si riduce e si aggiungono all'insieme Arc tutti gli (X_k, X_i).
Si noti che Arc è un insieme, quindi gli elementi già presenti non vengono ri-aggiunti.

Se il dominio è ridotto all'insieme vuoto, allora l'intero problema non ammette una soluzione consistente. In caso contrario, si continua fino a che non ci sono più archi in coda.

AC-3 restituisce un problema equivalente all'originale, avente le stesse soluzioni, ma con domini di variabili più piccoli. Può capitare che AC-3 riduca ogni dominio ad un singolo valore, trovando dunque una soluzione al problema.

Complessità computazionale:

- problema con n variabili (n nodi) ognuna con dominio di dimensione $\geq d$
- c vincoli binari (c archi)
- ogni arco può essere inserito in `Arc` soltanto d volte
- controllo di consistenza in $O(d^2)$
  Quindi complessivamente $O(cd^3)$.

Consistenza di cammino: restrizione dei vincoli binari utilizzando vincoli impliciti che sono inferiti considerando triplette di variabili. Formalmente:

> *Un insieme di variabili* $\{X_i, X_j\}$ *è* cammino-consistente *rispetto a una terza variabile* X_m *se, per ogni assegnamento* $\{X_i = a,\ X_j = b\}$ *consistente con i vincoli (se esistono) su {Xi, Xj}, esiste un assegnamento di* X_m *che soddisfa i vincoli su* $\{X_i, X_m\}$ *e* $\{X_m, X_j\}$.
> *Il termine* consistenza di cammino *si riferisce alla consistenza complessiva del cammino da* X_i *a* X_j *con* X_m *nel mezzo.*

Un CSP è k-*consistente* se, per ogni insieme di k–1 variabili e ogni loro assegnamento consistente, è sempre possibile assegnare un valore consistente a ogni k-esima variabile.

- la *1-consistenza* significa che, dato l'insieme vuoto, possiamo rendere consistente ogni insieme di una sola variabile: è il caso della *consistenza di nodo*
- la *2-consistenza* corrisponde alla *consistenza d'arco*
- la *3-consistenza* corrisponde alla *consistenza di cammino*

Un CSP è detto *fortemente* k-*consistente* se è k-consistente e anche $(k$–$1)$-consistente, $(k$–$2)$-consistente, ... , fino a 1-consistente.

Ricerca della soluzione dei CSP

Spesso dopo la propagazione dei vincoli rimangono variabili con più valori possibili. Si ricerca la soluzione tramite algoritmi di ricerca con *backtracking*.

Idea di base: effettuare assegnamenti ricorsivi fino a trovare soluzione o determinare fallimento. Passaggi:

- scegliere ripetutamente una variabile senza valore assegnato
  - assegnare sistematicamente tutti i suoi valori del dominio
  - cercare di estendere ogni assegnamento ad una soluzione tramite chiamate ricorsive
- se la chiamata ha successo, restituire soluzione; se fallisce, si riporta l'assegnamento allo stato precedente e si tenta il valore successivo
- se non si trova nessun valore accettabile, restituire fallimento

```
function Ricerca-Backtracking(csp) returns una soluzione, o fallimento
    return Backtracking(csp, { })
```

```
function Backtracking(csp, assegnamento) returns una soluzione, o fallimento
    if assegnamento è completo then return assegnamento
    var <- Scegli-Variabile-non-Assegnata(csp, assegnamento)
    for each valore in Ordina-Valori-Dominio(csp, var, assegnamento) do
        if valore è consistente con assegnamento then
            aggiungi {var = valore} ad assegnamento
            inferenze <- Inferenza(csp, var, assegnamento)
            if inferenze != fallimento then
                aggiungi inferenze a csp
                risultato <- Backtracking(csp, assegnamento)
                if risultato != fallimento then return risultato
                rimuovi inferenze da csp
            rimuovi {var = valore} da assegnamento
    return fallimento
```

L'algoritmo `Backtracking` è modellato sulla ricerca in profondità ricorsiva.

`Backtracking` deve fare delle scelte euristiche:

- quali valori assegnare nel passo successivo?
- in quale ordine provare i possibili valori del dominio?

Le funzioni `Scegli-Variabile-non-Assegnata` e `Ordina-Valori-Dominio` sono usate per implementare delle euristiche di uso generale.

La funzione `Inferenza` può essere usata, facoltativamente, per imporre consistenza d'arco, di cammino o k-consistenza. Se la scelta di un valore porta al fallimento (notato da `Inferenza` o `Backtracking`), allora gli assegnamenti di valori (inclusi quelli effettuati da `Inferenza`) sono annullati e si prova con un nuovo valore.

Usare ordini prefissati o casuali per la scelta delle variabili non assegnate non è ottimale. Scelte migliori:

- euristica *Minimum Remaining Values* (*MRV*): priorità alla variabile con meno valori legali
- euristica di grado: scegliere la variabile coinvolta nel maggior numero di vincoli
  Di solito l'euristica di grado si usa per risolvere pareggi nell'ordinamento MRV.

Per l'ordinamento dei valori nel dominio si usa l'euristica LCV:

- *Least Constraining Values* (*LCV*): priorità ai valori che lasciano il più grande sottoinsieme di valori legali alle variabili non assegnate
  Questa euristica è utile solo quando si vuole convergere velocemente ad una soluzione tra più possibili. Se si desidera raggiungere tutte le possibili soluzioni, o se il problema non ne ha, LCV non dà alcun vantaggio.

Tecniche *forward checking*: applicare la tecnica di consistenza d'arco (quella alla base di AC-3) anche durante il problema e non solo all'inizio. Dopo ogni assegnamento, si stabilisce la consistenza d'arco derivante dall'aver fissato quella variabile.

La coppia MRV + Forward Checking è la combinazione migliore per la maggior parte dei problemi CSP.

Anche il backtracking stesso ammette più possibili tecniche

- *backtracking cronologico*: tecnica più semplice, tornare alla variabile precedente e tentare un altro valore. Non sempre efficace
- *conflict set*: risalire fino all'assegnamento che causa conflitti con il dominio della variabile considerata al momento del fallimento
  Costruzione del set per *forward chaining*:
- ogni volta che un assegnamento $X = x$ causa la restrizione del dominio di un'altra variabile Y, si aggiunge quell'assegnamento al conflict set della seconda variabile Y
- se il dominio di questa Y viene svuotato portando al fallimento del problema, tutti gli elementi del conflict set di Y devono essere messi nel conflict set di X
- a questo punto è noto che $X = x$ è da evitare per proteggere la non-contraddizione di Y
  Backjumping: alternativa al backtracking cronologico basata sul salto alla variabile che ha causato il conflitto, usando le informazioni del conflict set.

Si definisce *agente basato sulla conoscenza* un agente in grado di decidere quali azioni intraprendere secondo un processo di ragionamento basato su una rappresentazione interna di conoscenza. Un *agente logico* è un agente basato sulla conoscenza la cui rappresentazione interna è basata sulla logica.

Informalmente, la *base di conoscenza* (KB, *knowledge base*) è costituita da un insieme di formule espresse in linguaggio formale. Ogni formula rappresenta un'asserzione sul mondo. Si definisce *assioma* una formula assunta come vera senza essere derivata da altre formule. Per essere utilizzabile da un agente, una KB deve prevedere almeno le due modalità di interazione elencate di seguito:

- un meccanismo per aggiunta di nuove formule alla KB (*tell*)
- un meccanismo per effettuare interrogazioni sulla KB (*ask*)
  L'*inferenza*, ovvero la derivazione di nuove formule a partire da quelle conosciute, può essere una conseguenza di entrambi i meccanismi. Se presente, la conoscenza iniziale (pregressa all'avvio dell'agente) prende il nome di *background knowledge*.

Definiamo ora un generico agente basato sulla conoscenza. Data in ingresso una percezione, l'agente la aggiunge alla KB. Successivamente, interroga quest'ultima per capire quale sia la migliore azione da intraprendere. Infine, l'agente comunica alla KB di aver intrapreso l'azione scelta. Il processo è descritto dal seguente pseudocodice:

```
function Agente-KB(percezione) returns un’azione
    persistent: KB: una base di conoscenza
                    t: un contatore inizializzato a 0 che indica il tempo
    Tell(KB, Costruisci-Formula-Percezione(percezione, t))
    azione ← Ask(KB, Costruisci-Interrogazione-Azione(t))
    Tell(KB, Costruisci-Formula-Azione(azione, t))
```

```
    t←t+1
    return azione
```

Le funzioni `Costruisci-Formula-Percezione`, `Costruisci-Interrogazione-Azione` e `Costruisci-Formula-Azione` implementano l'interfaccia tra il sistema interno di *rappresentazione e ragionamento* (*knowledge representation and reasoning*) e la parte dell'agente che interagisce con l'ambiente (sensori e attuatori). In particolare, `Costruisci-Formula-Percezione` costruisce una formula che asserisce la percezione ricevuta dall'agente in un dato istante temporale. `Costruisci-Interrogazione-Azione` costruisce una formula per interrogare la KB sull'azione da intraprendere in un dato momento. Infine, `Costruisci-Formula-Azione` costruisce una formula finalizzata ad asserire che l'azione scelta sia stata effettivamente eseguita. Le funzioni `Tell` e `Ask` astraggono i dettagli del meccanismo di inferenza.

Esistono due approcci principali per costruire un agente basato sulla conoscenza. Il primo modo, noto come *approccio dichiarativo*, consiste nel fornire la conoscenza necessaria all'agente tramite una serie di istruzioni `Tell` che lo informino sull'ambiente nel quale dovrà operare. L'approccio alternativo, detto *procedurale*, consiste invece nel codificare direttamente in un programma i comportamenti desiderati. La determinazione dell'approccio migliore tra i due, oggetto di dibattiti negli anni '70 e '80, è oggi risolta dall'opinione comune che sia buona pratica combinare i due approcci. Ad esempio, la conoscenza dichiarativa può essere compilata in codice procedurale per ragioni di efficienza.

La *conoscenza* può essere definita come una relazione tra un *conoscitore* e una *proposizione* (ovvero un'asserzione dichiarativa). Le *proposizioni*, a loro volta, possono essere definite semplicemente come entità astratte che possono essere vere o false. La conoscenza assume implicitamente che la proposizione considerata sia vera. Si parla di *credenza* quando la proposizione è vera *secondo il conoscitore*.
Definiamo ora il concetto di *rappresentazione*. Una rappresentazione è una relazione tra due domini, nei quali gli elementi del primo (*rappresentatore*) vengono usati al posto di quelli del secondo. Solitamente il rappresentatore è più accessibile o immediato del secondo. Solitamente si impiega un rappresentatore formale, ovvero un carattere o un gruppo di caratteri provenienti da un alfabeto predeterminato. Un insieme di simboli può rappresentare una proposizione. La frase simbolica è concreta mentre la proposizione è astratta.

La *rappresentazione della conoscenza* è la disciplina che si occupa dell'uso di simboli formali per rappresentare un insieme di proposizioni credute da un agente. Può esistere un numero infinito di proposizioni credute a fronte di un numero finito di proposizioni rappresentate. È il *ragionamento* a colmare il divario tra il rappresentato e il creduto.

La *Knowledge Representation Hypothesis* proposta dal filosofo Brian Smith è alla base del campo della rappresentazione della conoscenza. Essa suggerisce che le rappresentazioni simboliche usate da agent basati sulla conoscenza efficaci debbano godere delle seguenti due proprietà:

1. *interpretabilità umana*: gli osservatori esterni (umani) devono essere in grado di capire che i simboli rappresentino proposizioni specifiche;
2. *comportamento del sistema*: il comportamento dell'agente deve essere direttamente influenzato dalle rappresentazioni simboliche.
   L'ipotesi porta allo sviluppo di *sistemi basati sulla conoscenza* (*KB systems*), progettati in modo che la posizione intenzionale sia fondata su rappresentazioni simboliche. Tali rappresentazioni (insiemi di simboli che rappresentano le proposizioni date) sono chiamate basi di conoscenza.

A questo punto, è possibile definire il *ragionamento* come una manipolazione formale dei simboli che rappresentano le proposizioni per produrre rappresentazioni di nuove proposizioni. Il ragionamento porta quindi a determinare una proposizione che sia *conseguenza logica* delle sue premesse (proposizioni iniziali). L'uso dei simboli facilita l'applicazione della logica nelle macchine. Il ragionamento, nella visione presentata, è infatti una forma di calcolo che lavora con i simboli invece che con i numeri. Il ragionamento è fondamentale per gli agenti basati sulla conoscenza perché questi ultimi devono agire in base alla loro comprensione del

mondo, non solo sulla base delle informazioni esplicitamente rappresentate. In questo modo, gli agenti possono fare inferenze e prendere decisioni che superano la conoscenza esplicita. Un concetto chiave del ragionamento è l'*implicazione logica*, definibile informalmente nel modo seguente: le proposizioni rappresentate da un insieme di frasi S implicano una proposizione p se la verità di p è implicita nella verità delle frasi in S. L'implicazione richiede l'uso di un linguaggio formale con una chiara definizione di cosa significhi che una frase sia vera o falsa.

Un linguaggio di *knowledge representation* deve essere conciso, il più possibile *espressivo*, *non ambiguo*, *indipendente dal contesto* ed *efficace*, ovvero deve esistere una procedura di inferenza che sia corretta e implementabile. Alcune possibili scelte sono:

- *linguaggio naturale*: espressivo ma ambiguo e non conciso;
- *linguaggi di programmazione*: precisi, strutturati ma poco espressivi;
- *logica*: precisa, concisa ed espressiva.

La base di conoscenza è ridefinibile come un insieme di proposizioni scritte in un linguaggio formale. La logica è la scienza che fornisce all'uomo gli strumenti per verificare la rigorosità di un ragionamento. Essa fornisce gli strumenti formali per:

- analizzare inferenze in termini di operazioni su espressioni simboliche;
- dedurre le conseguenze di date premesse;
- studiare la verità o falsità di certe proposizioni data la verità o falsità di altre proposizioni;
- stabilire la consistenza e validità di una data teoria.

In informatica, la logica è utilizzata:

- in intelligenza artificiale, come linguaggio formale per la rappresentazione di conoscenza:
  - semantica non ambigua;
  - sistemi formali di inferenza;
- per sistemi di dimostrazione automatica di teoremi e studio di meccanismi efficienti per la dimostrazione;
- per la progettazione di reti logiche;
- nei database relazionali come potente linguaggio per l'interrogazione intelligente;
- come linguaggio di specifica di programmi per eseguire prove formali di correttezza;
- come un vero e proprio linguaggio di programmazione (programmazione logica e linguaggio PROLOG).

Come ogni linguaggio formale, i linguaggi logici sono formati da una *sintassi* e da una *semantica*. La sintassi è un insieme di regole che specificano quali formule appartengono al linguaggio (formule ben formate). La semantica è l'insieme delle regole che permettono l'interpretazione delle formule di un linguaggio formale in un modo che attribuisca loro un significato. La semantica definisce la *verità* delle formule rispetto ad ogni mondo possibile. Una base di conoscenza ha infatti l'obiettivo di rappresentare la realtà (il mondo) attraverso un insieme di formule. Data una KB, la rappresentazione da essa fornita potrebbe descrivere più mondi. È per questa ragione che si parla dunque di "mondi possibili". In modo più formale, questo concetto si può definire *interpretazione*. Se i mondi possibili possono essere considerati ambienti potenzialmente reali nei quali l'agente potrebbe o non potrebbe trovarsi, allora le interpretazioni sono astrazioni matematiche. Ognuna di esse deve avere in tal caso un valore di verità fissato per ogni formula della KB.

Data un'interpretazione m e una frase α, si dice che m è un *modello* di α se α è vera in m. Si denota con $M(\alpha)$ l'insieme dei modelli di α. La nozione può essere estesa anche ad una KB nel modo seguente. Data una base di conoscenza (insieme di frasi) K, si dice che un'interpretazione m è un modello per K se ogni frase $\alpha \in K$ è vera in m. Si denota con $M(K)$ l'insieme dei modelli per K. Nella teoria di Russell e Norvig il termine *modello* può indicare sia l'interpretazione che i mondi possibili.

Per indicare che una formula logica β è *conseguenza logica* di un'altra formula α, scriviamo che $\alpha \models \beta$. Formalmente, $\alpha \models \beta$ se e solo se ogni modello di α è anche modello di β, ovvero $M(\alpha) \subseteq M(\beta)$. Il concetto si può estendere per applicarlo ad una base di conoscenza K e ad una formula α nel modo seguente: $K \models \alpha$ se e solo se $M(K) \subseteq M(\alpha)$. Per verificare se $K \models \alpha$, si calcolano tutti i modelli di K e si verifica che lo siano anche per α (*model checking*).

Alternativamente, si può usare un *sistema* (o *procedura*) *di inferenza*. Si tratta di un sistema di regole che permettono di fare inferenza, ovvero dedurre nuova conoscenza da quella già posseduta. Se α può essere derivato da K attraverso la procedura di inferenza i, si denota $K \vdash_i \alpha$. Un sistema di inferenza è *corretto* se la conclusione a cui permette di giungere è sempre una conseguenza logica delle premesse, ovvero se $K \vdash_i \alpha$ allora $K \models \alpha$. Un sistema si dice *completo* se attraverso la sua applicazione si possono derivare tutte le conseguenze logiche delle premesse, ovvero, quando $K \models \alpha$, allora $K \vdash_i \alpha$.

Le regole di inferenza descrivono processi di ragionamento in cui è garantito che le conclusioni siano vere in un qualsiasi mondo che avveri le premesse. Se una KB è vera nel modo reale, allora ogni formula da essa derivata con un processo di inferenza corretto è vera nel mondo reale.

Logica classica

La logica classica si divide in due classi principali di linguaggi logici: *logica proposizionale* e *logica dei predicati del primo ordine*. Esse permettono di esprimere proposizioni e le relazioni tra esse. La principale differenza tra le due è nell'espressività. La logica dei predicati, al contrario della logica proposizionale, permette di esprimere variabili e quantificazioni. Questo permette dunque di analizzare gli elementi che compongono le proposizioni. La logica dei predicati è dunque più espressiva della logica proposizionale.

Logica proposizionale

La logica proposizionale studia *formule complesse* generate dalla concatenazione di *proposizioni atomiche* (singoli letterali che possono essere veri o falsi) mediante *connettivi*. I connettivi più comuni sono la congiunzione $\wedge$, la disgiunzione $\vee$, la negazione $\neg$, l'implicazione $\Rightarrow$ e il "se e solo se" $\Leftrightarrow$.

La *semantica* è lo studio delle regole che stabiliscono il valore di verità all'interno di un'*interpretazione*. Per interpretazione si intende una funzione $I : \mathcal{L} \to \{T, F\}$ di assegnazione di un valore di verità (o falsità) per ogni simbolo atomico presente in una formula. Data una formula composta da n simboli atomici, sono disponibili per essa 2^n possibili interpretazioni.

Il *model checking* è una tecnica per verificare se una *knowledge base* è modello per una proposizione data, ovvero per asserire se $K \models \alpha$. Per prima cosa, bisogna determinare e raccogliere tutte le possibili interpretazioni dei simboli proposizionali sia di K che di α. Successivamente, si esamina il modello $M(K)$ determinando quali interpretazioni sono modello per K. Infine, si verifica se ogni modello di K è anche modello di α.

Una tecnica alternativa al *model checking* è il metodo della *dimostrazione di teoremi*. Si procede dalle premesse cercando di ricavare conclusioni. Ogni dimostrazione ha struttura del tipo

$$\frac{A_1 \dots A_n}{A}$$

dove A_i sono le premesse e A è la conclusione, ovvero, più semplicemente,

$$\frac{\text{Premesse}}{\text{Conclusioni}}.$$

A questo fine, è utile introdurre tre concetti: *equivalenza logica*, *validità di una formula*, *soddisfacibilità di una formula*. Due formule sono logicamente equivalenti se e solo se la prima è modello della seconda e la seconda è modello della prima:

$$\alpha \equiv \beta \Leftrightarrow \alpha \models \beta \wedge \beta \models \alpha.$$

Una formula è *valida* (o è una *tautologia*) se essa è vera per qualunque interpretazione. Una formula è *soddisfacibile* se esiste almeno un'interpretazione secondo la quale essa sia vera. La determinazione della soddisfacibilità, o problema SAT, è stato il primo problema $NP - complete$ ad essere individuato.

L'unione di validità ed equivalenza logica permette di costruire il *teorema di deduzione*. Esso asserisce che

$$\alpha \models \beta \Leftrightarrow (\alpha \Rightarrow \beta);$$

inoltre, α valida se e solo se $\neg\alpha$ è insoddisfacibile; α è soddisfacibile se e solo se $\neg\alpha$ non è valida.

Esistono alcune regole, dette *regole di inferenza logica*, che permettono di ricavare conseguenze da proposizioni e premesse. La prima, detta *modus ponens*, procede come segue:

$$\frac{\alpha \Rightarrow \beta,\ \alpha}{\beta}.$$

Un'ulteriore regola, detta *and elimination*, permette di semplificare le congiunzioni:

$$\frac{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_m}{\alpha_i}.$$

Dove conveniente, è possibile sostituire $\alpha \Rightarrow \beta$ con $\neg\alpha \vee \beta$.

Una proprietà importante della logica proposizionale è la *monotonicità*: l'aumentare delle proposizioni in una *knowledge base* non può mai ridurre il numero delle conseguenze da essa ricavabili. Formalmente, se $K \models \alpha$ allora $K \cup \{\beta\} \models \alpha$ per ogni α e β.

Il *problema di ricerca delle dimostrazioni* consiste nel dimostrare la conseguenza logica di una proposizione da una *knowledge base* ($K \models \alpha$). Esso contiene quattro componenti:

1. *stato iniziale* (la KB K)
2. *azioni*: insieme di tutte le regole di inferenza che, per ogni proposizione appartenente alla *knowledge base*, corrispondono alle premesse delle regole
3. *risultato*: aggiunta a K di tutte le conclusioni dell'inferenza applicata
4. *obiettivo*: stato che contiene α

Il *principio di risoluzione* asserisce che, date due clausole m e l contenenti letterali complementari (ad esempio $l_i = \neg m_j$), è possibile costruire un'unica clausola equivalente con i termini complementari elisi. In formula:

$$\frac{l_1 \vee \ldots \vee l_k,\ m_1 \vee \ldots \vee m_n}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k,\ m_1 \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n}$$

Si definisce *forma normale congiuntiva* (CNF) una rappresentazione basata sulla congiunzione di clausole, che a loro volta sono disgiunzioni di letterali:

$$\gamma_1 \wedge \gamma_2 \wedge \ldots \wedge \gamma_n$$

dove

$$\gamma_i = l_1 \vee l_2 \vee \ldots \vee l_k.$$

Nella forma normale congiuntiva non sono dunque ammesse bicondizionali e implicazioni. Le negazioni possono essere applicate solo a singoli letterali. Tutte le proposizioni della logica proposizionale possono essere riscritte in forma normale congiuntiva, utilizzando le regole di semplificazione e distribuzione.

L'*algoritmo di risoluzione* è una tecnica per risolvere il problema di ricerca delle dimostrazioni. Per dimostrare $K \models \alpha$, ovvero che α sia conseguenza logica di K, si procede per assurdo. Si cerca di dimostrare che $(K \wedge \neg\alpha)$ sia insoddisfacibile. Bisogna dunque trovare una contraddizione in $(K \wedge \neg\alpha)$. La si converte

innanzitutto in CNF. Si procede applicando il principio di risoluzione ovunque possibile. Se il risultato finale è una clausola vuota, allora si è dimostrato che $K \models \alpha$. In caso contrario, se il risultato non è vuoto e non si riesce a procedere oltre, si è scoperto che $K \not\models \alpha$. Lo pseudocodice della procedura è il seguente:

```
function PL-Risoluzione(K, α) returns true oppure false
    inputs: una base di conoscenza K, una formula proposizionale α
    clausole ← l'insieme di clausole nella rappresentazione CNF di (K ∧ ¬α)
    nuove ← { }
    loop do
        for each coppia di clausole Ci, Cj in clausole do
            risolventi ← PL-Risolvi(Ci, Cj)
            if risolventi contiene la clausola vuota then return true
            nuove ← nuove ∪ risolventi
        if nuove ⊆ clausole then return false
        clausole ← clausole ∪ nuove
```

La funzione `PL-Risolvi` restituisce l'insieme di tutte le possibili clausole ottenute risolvendo i due input.

Una *clausola di Horn* è un tipo particolare di clausola con delle restrizioni aggiuntive. Se ne distinguono due tipi. Il primo, detto *clausola di Horn definita*, contiene un solo letterale non negato, mentre tutti gli altri sono negati (ad esempio $\neg A \vee B \vee \neg C$). Il secondo detto *clausola di Horn obiettivo*, contiene solamente letterali negati (ad esempio $\neg A \vee \neg B \vee \neg C$). Un singolo letterale è chiamato *fatto*. La maggiore restrittività delle clausole di Horn permette di applicare algoritmi di risoluzione più efficienti rispetto a quelli applicabili su clausole generiche. Gli algoritmi basati sulle clausole di Horn permettono infatti di fare inferenza con complessità temporale $O(n)$ data una KB da n clausole. In una KB contenente solamente clausole definite, è possibile procedere per *concatenazione* (in avanti o all'indietro) per fare inferenza, dopo aver trasformato tutte le clausole in implicazioni. Ad esempio, $(A \vee \neg B \vee \neg C)$ è trasformabile in $((B \wedge C) \Rightarrow A)$. $(B \wedge C)$ è detta *corpo* dell'implicazione mentre A prende il nome di *testa*.

L'algoritmo di *concatenazione in avanti* determina se un singolo simbolo proposizionale q è conseguenza logica della base di conoscenza K composta da clausole definite. L'algoritmo comincia dai fatti noti presenti in K. Controlla poi se grazie a tali fatti è possibile verificare tutte le premesse di un'implicazione (applicazione del *modus ponens*), la sua conclusione è quindi aggiunta all'insieme dei fatti noti. Questo processo continua finché non viene aggiunta la stessa query q o non è più possibile effettuare alcuna inferenza. Segue lo pseudocodice dell'algoritmo:

```
function PL-CA-Consegue?(K, q) returns true oppure false
    inputs: una base di conoscenza K, un simbolo proposizionale q
    conto ← una tabella, dove conto[c] è il numero di simboli della premessa della
clausola c
    inferiti ← una tabella, dove inferiti[s] è inizialmente false per tutti i simboli
    coda ← una coda di simboli, che contiene inizialmente quelli noti come veri in K (i
fatti noti)
    while coda non è vuota do
        p ← Pop(coda)
        if p = q then return true
        if inferiti[p] = false then
            inferiti[p] ← true
            for each clausola c in K con p in c.Premessa do
                conto[c] ← conto[c] - 1
```

```
                if conto[c] = 0 then aggiungi c.Conclusione a coda
    return false
```

La *concatenazione all'indietro* parte dalla *query* q e procede a ritroso. Se è già noto che la *query* è vera, non occorre fare nulla. In caso contrario, l'algoritmo trova tutte le implicazioni nella base di conoscenza che hanno q come conclusione. Se tutte le premesse di una di quelle implicazioni possono essere dimostrate vere mediante la concatenazione all'indietro, allora q è vera. Come nel caso della concatenazione in avanti, un'implementazione efficiente si esegue in tempo lineare.

La concatenazione in avanti è un esempio del concetto generale di *ragionamento guidato dai dati*, un tipo di ragionamento in cui l'attenzione parte dei fatti conosciuti. All'interno di un agente, quest'approccio può essere usato per derivare conclusioni partendo dalle percezioni in ingresso, spesso anche senza avere in mente un quesito specifico. È spesso usato per task quali *object recognition* e *routine recognition*.

La concatenazione all'indietro è una forma di ragionamento basato sugli obiettivi. È utile per rispondere a questioni specifiche di *problem solving* come "Cosa devo fare adesso?" oppure "Dove avrò lasciato le chiavi?". Spesso il costo della concatenazione all'indietro è molto meno che lineare nelle dimensioni della base di conoscenza, perché il processo coinvolge solo i fatti rilevanti.

Logica del primo ordine

La logica proposizionale è dichiarativa, espone *fatti* ma non spiega come derivarne conoscenza. Permette informazioni parziali e negate. È composizionale e indipendente dal contesto, nonché poco concisa. È limitata e non permette di rappresentare alcuni concetti, ad esempio "ogni numero naturale è pari oppure dispari".

Nella logica del primo ordine, invece dei semplici fatti della logica proposizionale, ci sono tre diversi tipi di entità: gli *oggetti*, che rappresentano delle costanti; le *relazioni* e *proprietà* (predicati); le funzioni che producono oggetti (anch'esse relazioni, ma univoche). Le relazioni sono caratterizzare dall'*arietà*, ovvero dal loro numero di parametri. La differenza tra predicati e funzioni è la seguente: i predicati sono relazioni tra individui (oggetti), mentre le funzioni *definiscono* individui (sempre oggetti).

Un'altra novità introdotta dalla logica del primo ordine è la *quantificazione*, ovvero la possibilità di introdurre quantificatori universali ($\forall$, "per ogni"), esistenziali ($\exists$, "esiste") e variabili ($\forall$ davanti ad una disgiunzione).

I simboli logici includono: le costanti logiche $\bot$ e $\top$ (*false* e *true*); i connettivi logici proposizionali $\wedge$, $\vee$, $\neg$, $\Rightarrow$ e $\Leftrightarrow$; i quantificatori $\forall$ e $\exists$, un insieme numerabile di simboli di variabile Var, solitamente x_1, x_2, ...; opzionalmente, il simbolo di predicato d'uguaglianza $=$. I simboli non logici includono: un insieme numerabile Const di costanti c_1, c_2, ...; un insieme numerabile Funct di funzioni f_1, f_2, ..., ognuna con arietà $\mathrm{ar}(f_i)$; un insieme numerabile Pred di simboli di predicato o di relazione P_1, P_2, ..., ognuno con arietà $\mathrm{ar}(P_i)$. I simboli non logici dipendono dal dominio rappresentato e devono avere un significato specifico in tale contesto. Ad esempio, per il dominio dei numeri naturali, si possono definire funzioni di arietà 2 per indicare operazioni di somma, relazioni di maggioranza o di successione.

Definiamo ora l'insieme Terms induttivamente come segue. Prima di tutto $\mathrm{Var} \cup \mathrm{Const} \subseteq \mathrm{Terms}$. Inoltre, se $t_1, \ldots t_n \in \mathrm{Terms}$ e $f_{/n} \in \mathrm{Funct}$ allora $f(t_1, \ldots, t_n) \in \mathrm{Terms}$. Nient'altro appartiene a Terms. Grazie a questa definizione preliminare, è ora possibile definire l'insieme delle formule ben formate in logica del primo ordine, rappresentato dall'insieme Forms. Anch'esso si definisce per induzione. Innanzitutto, se $t_1, \ldots, t_n \in \mathrm{Terms}$ e $P_{/n} \in \mathrm{Pred}$, allora $P(t_1, \ldots, t_n)$. Inoltre, $t_1, t_2 \in \mathrm{Terms}$ allora $(t_1 = t_2) \in \mathrm{Forms}$. Infine, se $f_1, f_2 \in \mathrm{Forms}$ e $x \in \mathrm{Vars}$, allora anche $(f_1 \wedge f_2)$, $(f_1 \vee f_2)$, $(f_1 \Rightarrow f_2)$, $(f_1 \Leftrightarrow f_2)$, $\neg(f_1)$, $\exists x(f_1)$ e $\forall x(f_1)$ appartengono a Forms. Nient'altro oltre a quanto precedentemente elencato appartiene all'insieme.

A livello sintattico l'uso dei connettivi logici è funzionalmente indistinguibile tra logica proposizionale e logica del primo ordine. Un predicato di arietà nulla equivale ad una variabile proposizionale. Per questo motivo, la logica del primo ordine può essere vista come un'estensione della logica proposizionale. Inoltre, la logica del

primo ordine permette di parlare di *oggetti*, concetto non presente nella logica proposizionale. Costanti e variabili rappresentano specifici oggetti, mentre i quantificatori permettono di esprimere informazioni sulla loro esistenza e sulle loro proprietà.

Le variabili si distinguono in *legate* e *libere* a seconda che siano, o meno, in una sottoformula che è nel campo d'azione di un quantificatore. Le *leggi di ridenominazione* permettono di cambiare il nome di variabili legate al fine di ridurre ambiguità in caso di quantificazioni a più livelli. Le formule si dividono invece in *ground* (senza variabili), *aperte* (contenenti solo variabili libere) e *chiuse* (contenenti solo variabili legate). Le formule *ground* sono un tipo di formule chiuse.

Anche nella logica del primo ordine si trova il concetto di *interpretazione*, seppure in modo diverso rispetto alla logica proposizionale. L'interpretazione assegna una semantica alle formule chiuse. Il *dominio del discorso* è l'insieme degli oggetti, che vanno collegati all'interpretazione. Per definire più formalmente il concetto di relazione è necessario dare una definizione formale di *relazione*.