Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso a Libera Scelta - 6 CFU

Introduzione all'Intelligenza Artificiale

Professore:
Prof. Alessio Micheli
Prof. Claudio Gallicchio

Autore:
Filippo Ghirardini

Anno Accademico 2023/2024

Contents

Introduzione all'Intelligenza Artificiale

Realizzato da: Ghirardini Filippo

A.A. 2023-2024

---

1 Introduzione

1.1 Obiettivi dell'IA

1.1.1 Modellare

Modellare fedelmente l'essere umano:

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

1.1.2 Risultati

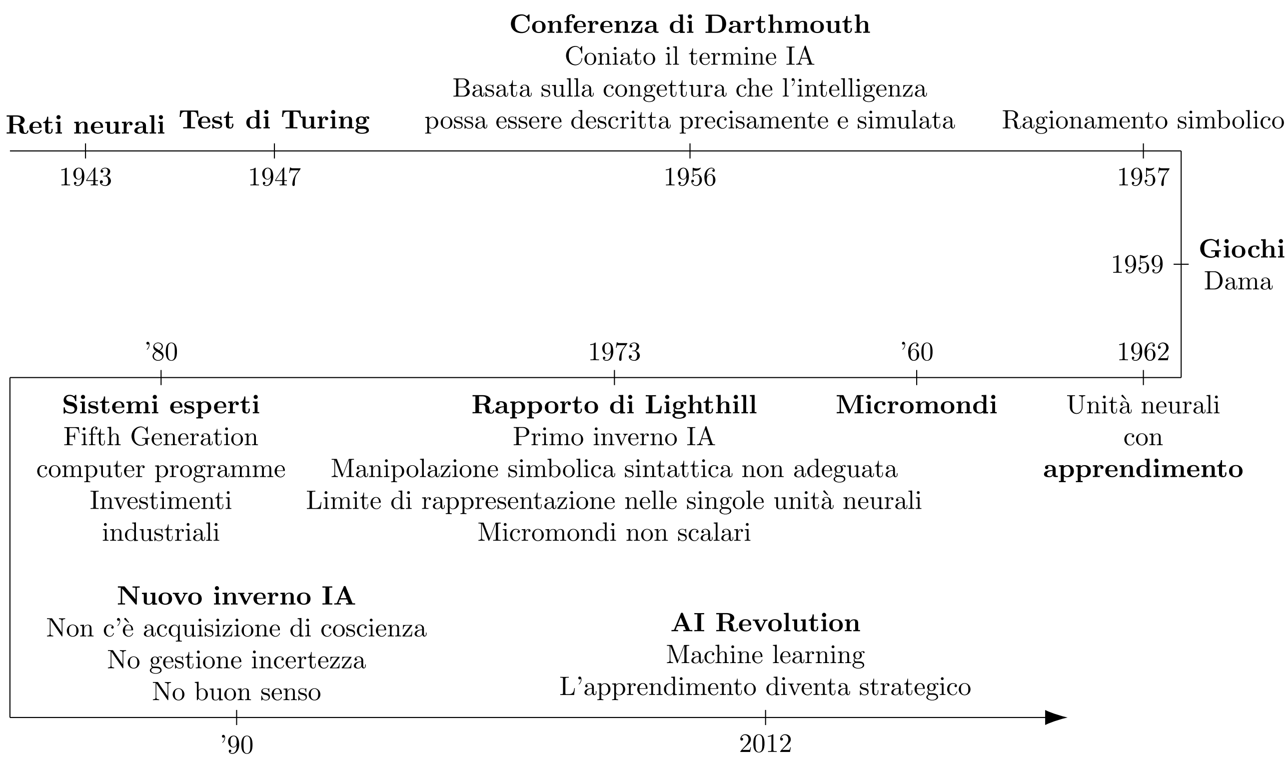
Raggiungere i risultati ottimali:

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

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

1.2 Storia dell'IA

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



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

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

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

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

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

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

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

1.3 Reti neurali

Le reti neurali sono caratterizzate da:

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

1.3.1 Deep Learning

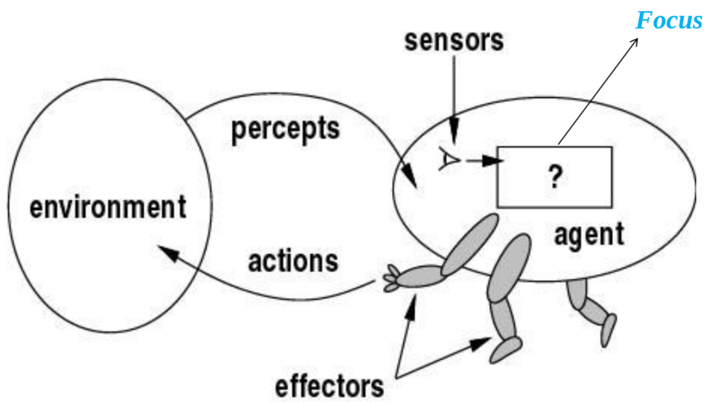
Abbinando alla capacità dei modelli di machine learning una grande quantità di dati e degli High Performance Computer, si è favorito molto il deep learning.
Dal 2010 le reti neurali profonde hanno iniziato a diffondersi molto nelle grandi industrie, riscuotendo successo ad esempio:

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

Questa tecnologia ha raggiunto prestazioni a livello di quelle umane.

2 Agenti intelligenti

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



Ciclo *percezione- azione*

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

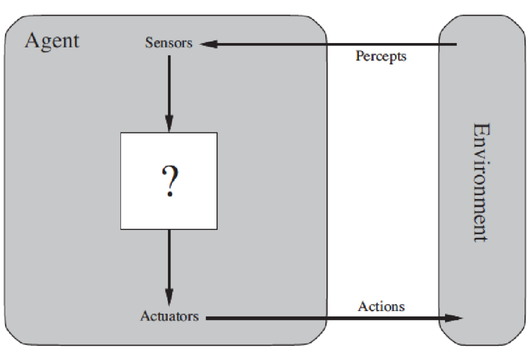
2.1 Caratteristiche

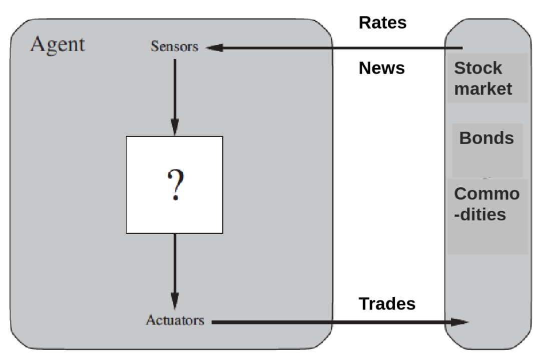
Un agente ha alcune caratteristiche:

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

2.1.1 Percezioni e azioni

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





2.2 Agente razionale

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

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

- **Esterna**
-

•

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

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

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

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

2.3 Ambienti

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

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

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

L'ambiente deve avere le seguenti proprietà:

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

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

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

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

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

2.4 Programma agente

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

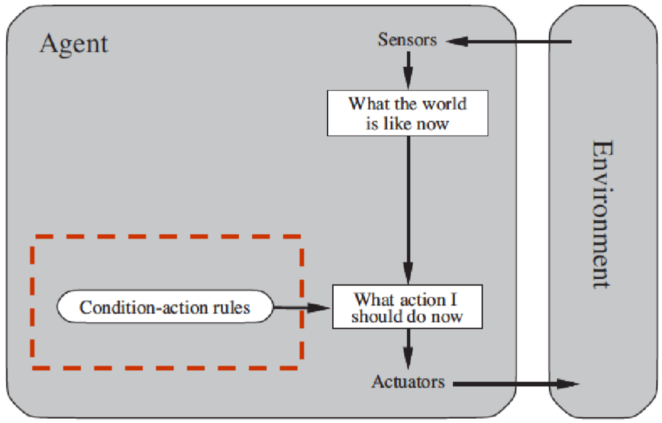
```
function Skeleton-Agent (percept) returns action
   static: memory, agent memory of the world
   memory <- UpdateMemory(memory, percept)
   action <- Choose-Best-Action(memory)
   memory <- UpdateMemory(memory, action)
   return action
```

2.4.1 Tabella

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

2.4.2 Agenti reattivi

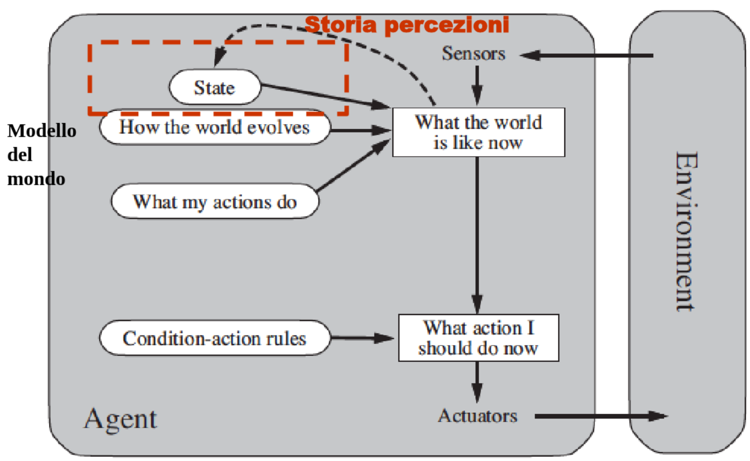
L'agente agisce in base a quello che percepisce senza salvare nulla in memoria.




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

2.4.3 Agenti basati su modello

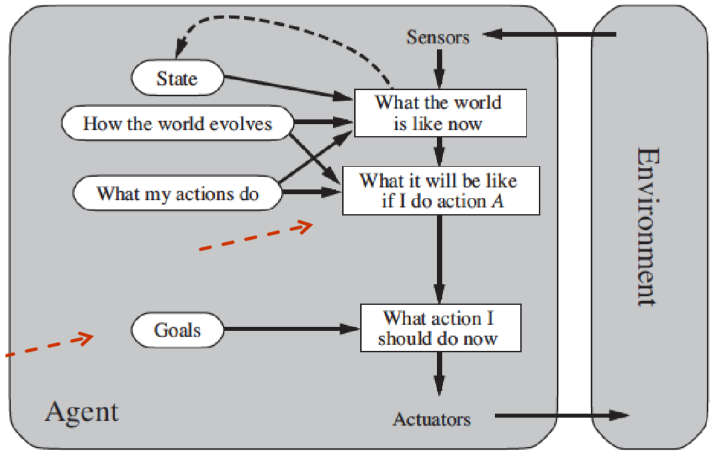
L'agente ha uno stato che mantiene la storia delle percezioni e influenza il modello del mondo.




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

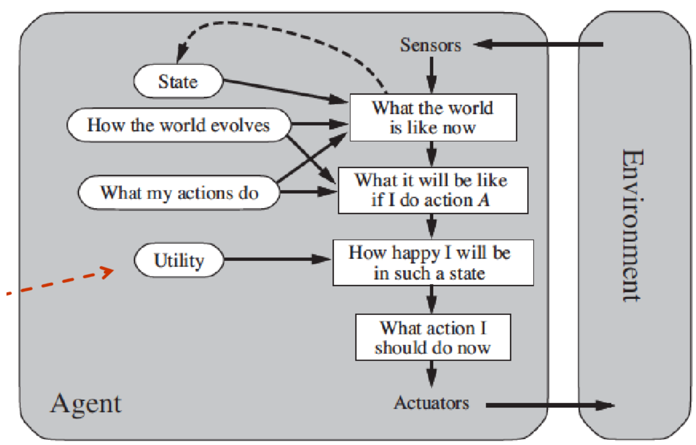
2.4.4 Agenti con obiettivo

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



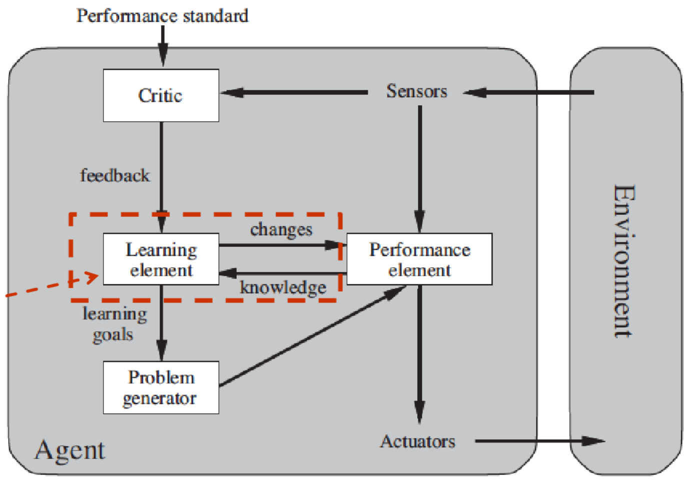
2.4.5 Agenti con valutazione di utilità

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



2.4.6 Agenti che apprendono

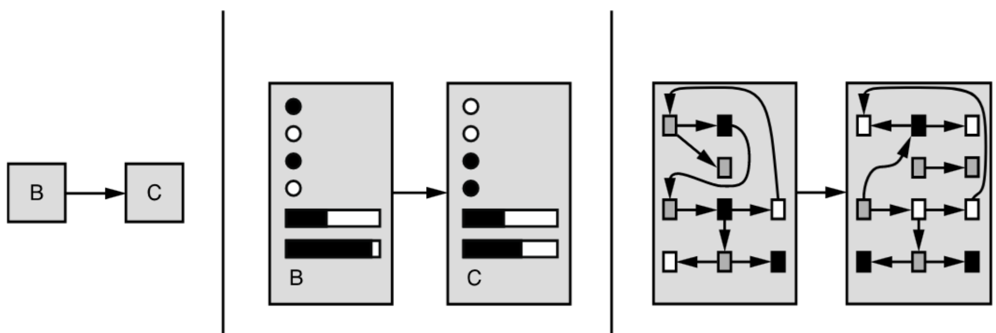
Questo tipo di agente include la capacità di **apprendimento** che produce cambiamenti al programma e ne migliora le prestazioni, adattando i comportamenti.
L'elemento **esecutivo** è il programma stesso, quello **critico** osserva e dà feedback ed infine c'è un generatore di problemi per suggerire nuove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



2.4.7 Tipi di rappresentazione

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

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



(a) Atomic (b) Factored (b) Structured

3 Agenti risolutori di problemi

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

3.1 Processo di risoluzione

I passi da seguire sono i seguenti:

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

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

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

3.2 Assunzioni

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

3.3 Formulazione del problema

Un problema può essere definito formalmente mediante 5 componenti:

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

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

3.4 Algoritmo di ricerca

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

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

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

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

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

3. **Modello di transizione**:

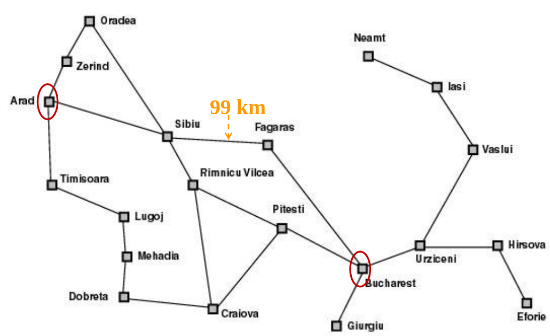
```
Risultato(In(Arad), Go(Sibiu)) = In(Sibiu)
```

4. **Test obiettivo**:

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

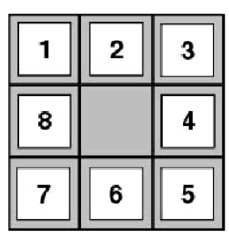
5. **Costo del cammino**: somma delle lunghezze delle strade

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



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

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



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

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

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

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

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

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

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

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

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

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

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

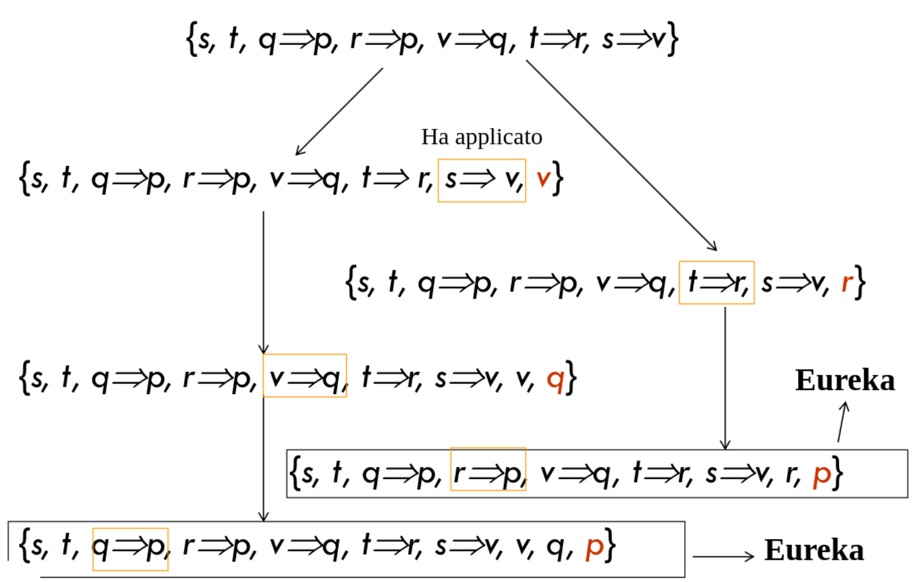
dimostrare una proposizione p utilizzando solamente la regola di inferenza *Modus Ponens*:

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

Scriviamo la formulazione del problema:

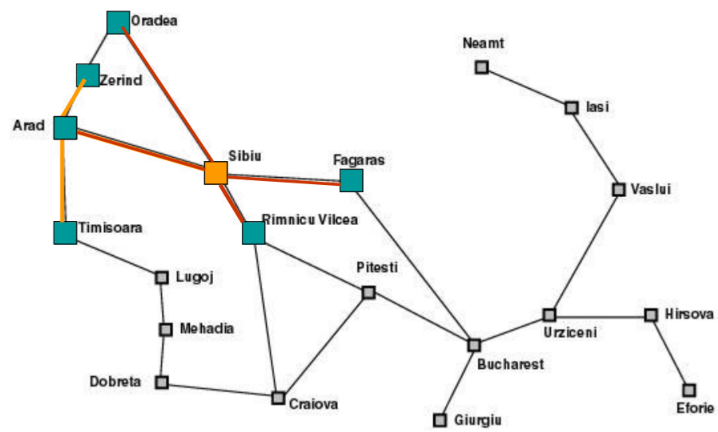
- **Stati**: insieme di proposizioni
- **Stato iniziale**: le premesse
- **Stato obiettivo**: un insieme di proposizioni contenente il teorema da dimostrare
- **Operatori**: l'applicazione del Modus Ponens

Lo spazio degli stati è quindi il seguente:



3.5 Ricerca della soluzione

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



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

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

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

3.6 Strategie di ricerca

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

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

3.6.1 Breadth First

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

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

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

```
function Ricerca-Ampiezza-g
   returns soluzione oppure fallimento
   nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
   if problema.Test-Obiettivo(nodo.Stato) then return Soluzione(nodo)
   frontiera = una coda FIFO con nodo come unico elemento
   esplorati = insieme vuoto
loop do
   if Vuota?(frontiera) then return fallimento
   nodo = POP(frontiera); aggiungi nodo.Stato a esplorati
   for each azione in problema.Azioni(nodo.Stato) do
   figlio = Nodo-Figlio(problema, nodo, azione)
   if figlio.Stato non e in esplorati e non in frontiera then
   if Problema.TestObiettivo(figlio.Stato) then return Soluzione(figlio)
   frontiera = Inserisci(figlio, frontiera) /* in coda
end
```

Analizziamone la complessità partendo dalle seguenti assunzioni:

- Fattore di **branching** b: numero massimo di successori
- **Depth** del nodo obiettivo più superficiale
- Lunghezza **massima** dei cammini nello spazio degli stati

La strategia è ottimale se tutti gli operatori hanno lo stesso costo k, ovvero se $g(n) = k \cdot depth(n)$, dove $g(n)$ è il costo del cammino per arrivare ad n.
La complessità nel *tempo* (nodi generati) sarà

$$T(b, d) = 1 + b + b^2 + \ldots + b^d \longrightarrow O(b^d)$$

mentre in *spazio* (nodi in memoria):

$$O(b^d)$$

È chiaro che l'algoritmo scali male, sopratutto per quanto riguarda lo spazio.

3.6.2 Depth first

In questo algoritmo si parte da un nodo e si scende nel primo figlio, procedendo appunto in profondità. Arrivati alle foglie si torna indietro ai figli precedentemente non visitati. In memoria tengo solamente i fratelli del path corrente ed elimino i rami già esplorati.
Possono esserci tre versioni possibili:

- **Albero**: data m la lunghezza massima dei cammini nello spazio degli stati e b il fattore di diramazione, la **complessità** in *tempo* è $O(b^m)$ (può essere maggiore di $O(b^d)$) mentre in *spazio* è $b \cdot m$. Rispetto al Breadth First, non è né completo né ottimale, ma ci garantisce un notevole risparmio in memoria
- **Grafo**: la memoria corrisponde a tutti i possibili stati, diventando quindi completo nello spazio finito (non in quello infinito)
- **Ricorsiva**: ancora più efficiente per la memoria perché mantiene solo il cammino corrente ($O(m)$). Viene realizzata con un algoritmo di *backtracing* che salva lo stato su uno stack a cui torna in caso di fallimento.

```
function Ricerca-DF-A (problema)
   returns soluzione oppure fallimento
   return Ricerca-DF-ricorsiva(CreaNodo(problema.Stato-iniziale), problema)

function Ricerca-DF-ricorsiva(nodo, problema)
   returns soluzione oppure fallimento
   if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
   else
   for each azione in problema.Azioni(nodo.Stato) do
      figlio = Nodo-Figlio(problema, nodo, azione)
      risultato = Ricerca-DF-ricorsiva(figlio, problema)
      if risultato != fallimento then return risultato
   return fallimento
```

3.6.3 Depth Limited

La ricerca in profondità limitata arriva fino ad un dato livello l. È completa solo se si conosce il limite superiore d per la profondità della soluzione e $d < l$. Non è ottimale e ha complessità in tempo $O(b^l)$ e in spazio $O(b \cdot l)$

3.6.4 Iterative Depth

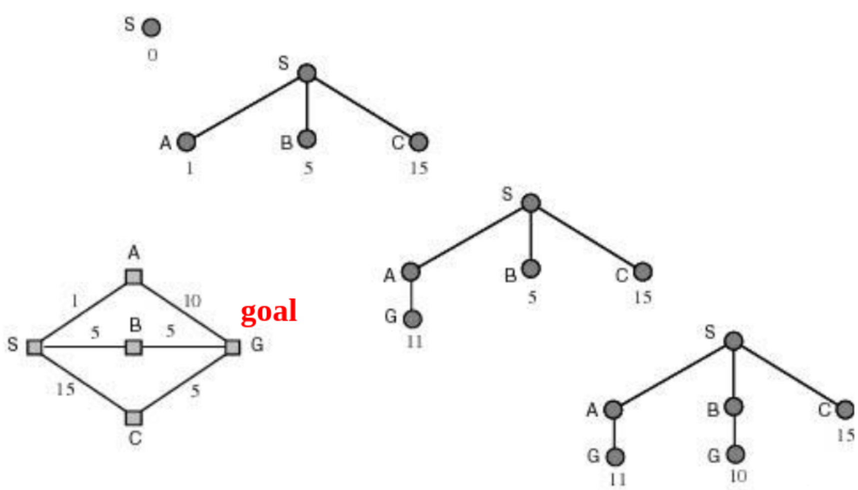
Questo approccio prevede di provare l'algoritmo depth limited con limite di profondità $l = 0, 1, \dots$ fino a trovare la soluzione. È il miglior compromesso tra breadth first e depth first:

- Complessità in **tempo** $O(b^d)$ se ammette soluzione
- Complessità in **spazio** $O(b \cdot d)$ se ammette soluzione

Quindi ha la *completezza* e l'*ottimalità* del breadth first e la complessità in *spazio* della depth first.

3.6.5 Uniform Cost

Partendo da una ricerca in ampiezza, la generalizziamo: si sceglie il nodo di costo minore sulla frontiera e si espande sui contorni di costo uguale.



Codice per la ricerca su albero:

```
function Ricerca-UC-A (problema)
     returns soluzione oppure fallimento
     nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
     frontiera = una coda con priorita con nodo come unico elemento
  loop do
     if Vuota?(frontiera) then return fallimento
     nodo = POP(frontiera)
     if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
     for each azione in problema.Azioni(nodo.Stato) do
        figlio = Nodo-Figlio(problema, nodo, azione)
        frontiera = Inserisci(figlio, frontiera) /* in coda con priorita*/
end
```

Codice per la ricerca su grafo:

```
function Ricerca-UC-G (problema)
     returns soluzione oppure fallimento
     nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
     frontiera = una coda con priorita con nodo come unico elemento
     esplorati = insieme vuoto
  loop do
     if Vuota?(frontiera) then return fallimento
     nodo = POP(frontiera);
     if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
     aggiungi nodo.Stato a esplorati
     for each azione in problema.Azioni(nodo.Stato) do
        figlio = Nodo-Figlio(problema, nodo, azione)
        if figlio.Stato non in esplorati e non in frontiera then
           frontiera = Inserisci(figlio, frontiera) /* in coda con priorita
        else if figlio.Stato in frontiera con Costo-cammino piu alto then
           sostituisci quel nodo frontiera con figlio */
end
```

Questo algoritmo è **ottimo** e **completo** purché il costo degli archi sia $\epsilon > 0$. Assunto C^* come costo della soluzione ottima, $\lfloor \frac{C^*}{\epsilon} \rfloor$ è il numero di mosse nel caso peggiore. La complessità è quindi $O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$.

Note 3.6.1. Quando ogni azione ha lo stesso costo, la complessità si avvicina a quella della breadth first: $O(b^{1+d})$.

3.7 Direzione

Un problema importante è quello della **direzione** della ricerca, che può essere:

- In **avanti** o guidata da *dati*: si esplora lo spazio di ricerca dallo stato iniziale all'obiettivo
- All'**indietro** o guidata dall'*obiettivo*: si esplora lo spazio di ricerca partendo da uno stato goal e riconducendosi ad un sotto-goal fino a trovare uno stato iniziale

Per scegliere la direzione bisogna tenere in conto di quale ha il **fattore di diramazione** minore. Si preferisce la ricerca all'*indietro* quando l'obiettivo è ben definito (e.g. theorem proving) mentre quella in *avanti* quando ci sono molteplici obiettivi (e.g. design).

3.7.1 Ricerca bidirezionale

Nella ricerca bidirezionale si procede in entrambe le direzioni fino ad incontrarsi. La **complessità** è:

- *Tempo*: $O(\sqrt{b^d})$ assumendo che il test dell'intersezione delle due direzioni sia costante

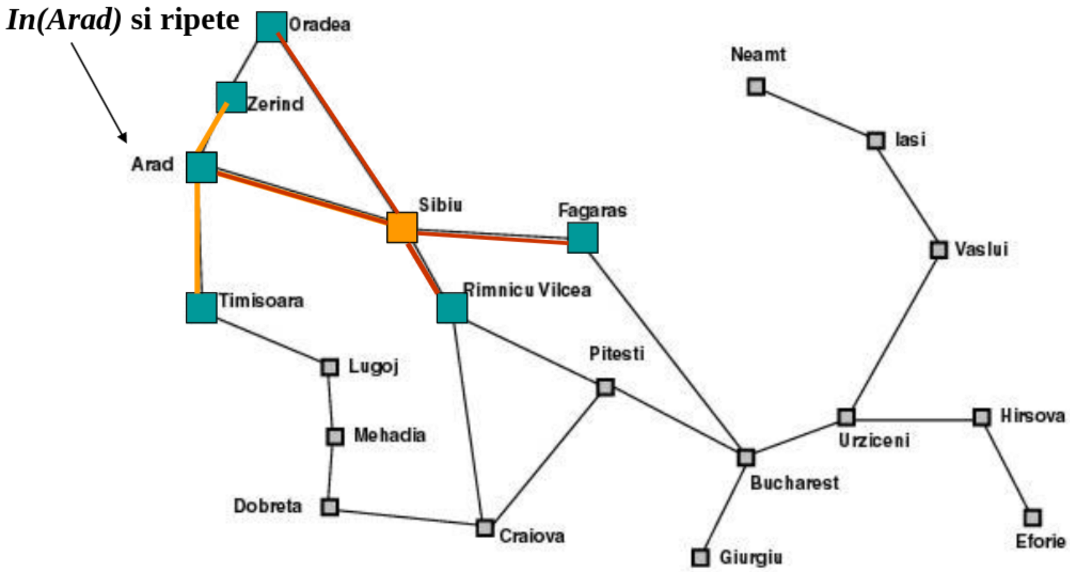
- *Spazio*: $O(\sqrt{b^d})$, poiché almeno tutti i nodi di una direzione saranno in memoria

Si noti che non sempre è applicabile, come nel caso in cui i predecessori non siano definiti o ci siano troppi stati obiettivo.

3.8 Problematiche

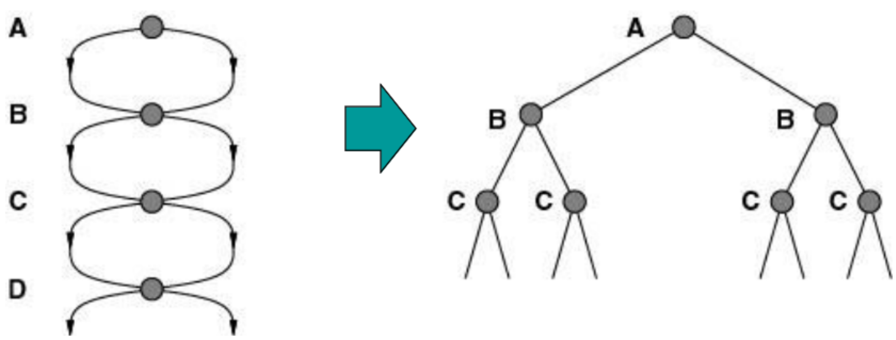
3.8.1 Cicli

I cammini ciclici rendono gli alberi di ricerca *infiniti* anche quando lo spazio degli stati è finito.



3.8.2 Ridondanze

Su spazi di stati a grafo si possono generare più volte nodi con lo stesso stato nella ricerca, anche in assenza di cicli.



Visitare questi stati è lavoro inutile. Per evitarlo serve **ricordare** gli stati già visitati, occupando ovviamente più spazio. Tre possibili soluzioni sono:

1. Non tornare nel nodo **genitore**, eliminandolo dai successori (non evita i cammini ridondanti)
2. Per evitare i **cammini ciclici** si controlla che i successori non siano antenati del nodo corrente
3. Non generare nodi con **stati già esplorati**: ogni nodo visitato deve essere salvato in memoria

Il costo può essere alto, ad esempio nella depth first la complessità in spazio torna ad essere pari a tutti gli stati.
La **ricerca** sul grafo avverrà quindi come segue:

1. Mantiene una lista di stati esplorati (lista chiusa)
2. Prima di espandere un nodo si controlla se era già stato incontrato o se è già nella frontiera
3. In quel caso, non viene espanso

Questa tecnica è ottimale solo se abbiamo la garanzia che il costo del nuovo cammino sia maggiore o uguale, ovvero non convenga.

3.9 Confronto

Confronto	**BF**	**UC**	**DF**	**DL**	**ID**	**BDir**
Completa	Si	Si(*)	No	Si(**)	Si	Si(***)
Tempo	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(\sqrt{b^d})$
Spazio	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(b \cdot m)$	$O(b \cdot l)$	$O(b \cdot d)$	$O(\sqrt{b^d})$
Ottimale	Si(****)	Si(*)	No	No	Si(****)	Si(***)

Legenda:

- *: se costo archi $\geq \epsilon \geq 0$
- **: se si conosce il limite alla profondità della soluzione ($l > d$)
- ***: se si utilizza UC o BF
- ****: se gli archi hanno tutti lo stesso costo

4 Ricerca euristica

5 Ricerca locale

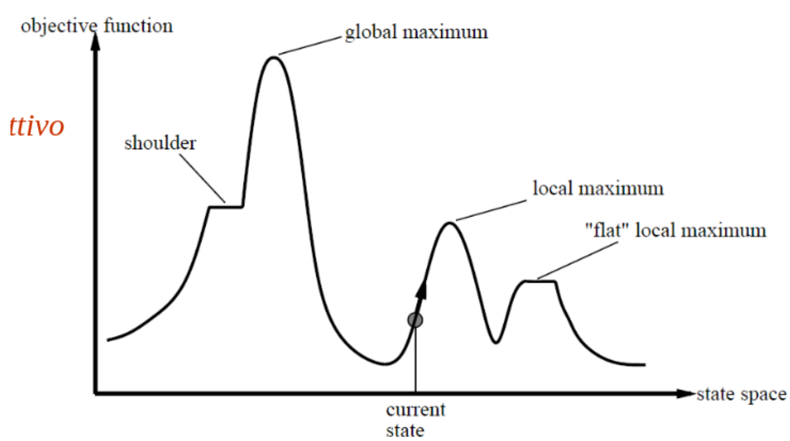
La ricerca *euristica* nello spazio di stati è troppo costosa ed è quindi necessario utilizzare metodi diversi.
Se prima gli algoritmi restituivano un cammino soluzione per raggiungere un goal, ora il goal è la soluzione stessa al problema. Gli algoritmi di ricerca locale sono adatti per problemi in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

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

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

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

Data la funzione euristica del costo dell'obiettivo



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

5.1 Hill climbing

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

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

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

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

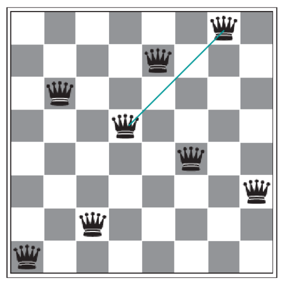
```
        # scegli il vicino con valore piu' alto (sulla funzione problem.value)
        neighbor = (sorted(neighbors,key = lambda x:problem.value(x), reverse = True))[0]
        if problem.value(neighbor) <= problem.value(current):
            break
        else:
            current = neighbor # (altrimenti,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

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

5.1.1 8 regine

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

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



Possiamo migliorare l'algoritmo in alcuni modi:

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

5.2 Tempra simulata

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

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

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

5.2.1 Scelta dei parametri

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

5.3 Local beam

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

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

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

5.3.1 Versione stocastica

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

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

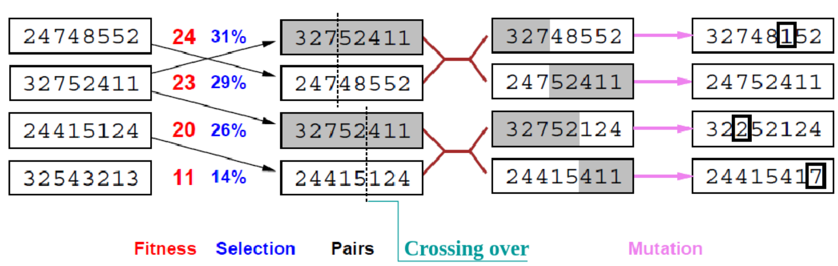
5.3.2 Algoritmi genetici ed evolutivi

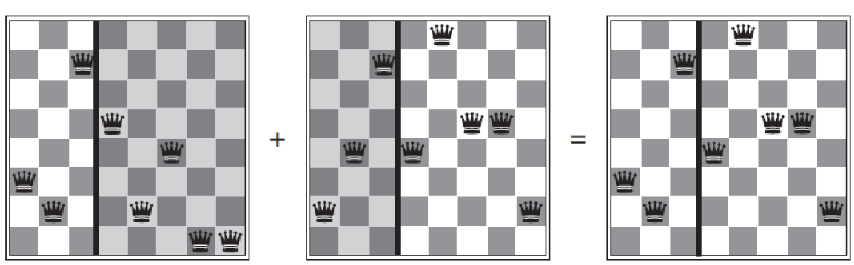
Sono una variante della *beam search stocastica* in cui gli stati successori sono ottenuti combinando due stati genitore invece che per evoluzione. La **popolazione** iniziale è composta da k **individui** generati casualmente e rappresentati come una stringa. Gli individui sono valutati da una funzione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

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

Ogni generazione dovrebbe essere migliore della precedente.

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





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

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

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

5.4 Spazi continui

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

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

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

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

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

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

5.5 Ambienti realistici

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

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

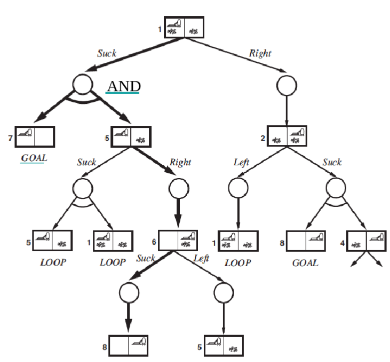
- Se aspira in una stanza sporca la pulisce ma a volte pulisce anche una stanza adiacente
- Se aspira in una stanza pulita, a volte la sporca

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

5.5.1 Albero AND-OR

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

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



6 Agenti basati su conoscenza

C'è bisogno di rappresentare la conoscenza in maniera parziale e incompleta (gli ambienti sono parzialmente osservabili). Ci servono quindi dei linguaggi più espressivi e con **capacità inferenziali**.

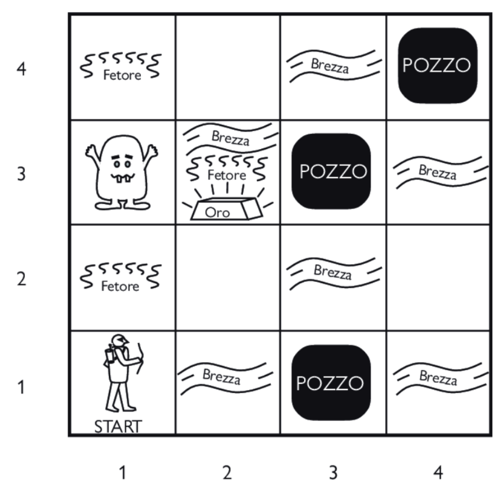
6.1 Knowledge Base

L'insieme di tutta la conoscenza necessaria a decidere un'azione da compiore è la **knowledge base** e può essere definita in due modi:

- *Dichiarativo*: all'agente viene detto cosa deve sapere, partendo da una conoscenza di base vuota e aggiungendo progressivamente formule (TELL)
- *Procedurale*: si scrive un programma che definisca il processo decisionale una volta per tutte

Definizione 6.1.1 (Knowledge Base)**.** *Un insieme di* enunciati *(formule) espressi in un linguaggio di rappresentazione.*

Esempio 6.1.1 (Wumpus World)**.** Il mondo del Wumpus è una caverna fatta di stanze connesse tra loro. All'interno c'è questa bestia puzzolente che mangia chiunque entri nella stanza in cui si trova. Questo può essere ucciso dall'agente che ha una freccia a disposizione.
Ci sono delle stanze con degli *ostacoli*: pozzi, in cui se l'agente entra, muore. In una delle stanze si trova l'*obiettivo*, ovvero un lingotto d'oro.
L'agente non conosce l'ambiente e la sua posizione, se non all'inizio.



Definiamo le **misure di prestazione**:

- +1000 se trova l'oro, torna in [1,1] ed esce
- -1000 se muore
- -1 per ogni azione
- -10 se usa la freccia

Invece l'**ambiente** è una griglia 4x4 circondata da pareti di delimitazione. L'agente inizia sempre nella posizione [1,1] rivolto verso destra (la prima casella è sempre safe). Le posizioni dell'oro e della bestia sono casuali e tutti i riquadri hanno una probabilità di 0.2 di contenere un pozzo.
L'agente può fare le seguenti **azioni**:

- Andare avanti
- Ruotare a destra o a sinistra di 90
- Afferrare un oggetto

- Scagliare la freccia
- Uscire

Il nostro agente puo **percepire** le seguenti cose:

- *Fetore* nelle caselle adiacenti alla bestia
- *Brezza* nelle caselle adiacenti ai pozzi
- *Luccichio* nella casella con l'oro
- *Urlo* se la bestia viene uccisa

e vengono rappresentati come una quintupla, che ad esempio nella prima casella vale:

$$[none, none, none, none, none]$$

Di conseguenza sappiamo che nelle caselle adiacenti non ci sono né pozzi né la bestia.

6.1.1 Tell-Ask

L'agente interagisce con la knowledge base tramite un'interfaccia funzionale di tipo Tell-Ask:

- *Tell*: aggiungere nuovi enunciati
- *Ask*: interagire con la knowledge base
- *Retract*: eliminirare enunciati

Gli enunciati nella KB rappresentano le credenze dell'agente e le risposte α devpomp essere tali per cui queste discendano necessariamente dalla KB.
Il problema fondamentale è quindi capire, data una base di conoscenza KB, come dedurre che un certo fatto α è vero di conseguenza.

$$KB \models \alpha \tag{2}$$

Un programma basilare è il seguente:

```
function Agente-KB (percezione) returns azione
  persistent: KB, una base di conoscenza
    t, un contatore, inizialmente a 0, che indica il tempo
  TELL(KB, Costruisci-Formula-Percezione(percezione, t ))
  azione = ASK(KB, Costruisci-Query-Azione( t ))
  TELL(KB, Costruisci-Formula-Azione(azione, t ))
  t = t + 1
  return azione
```

6.1.2 Analisi

A differenza di una *base di dati*, la base di conoscenza non contiene solo fatti specifici da recuperare ma anche fatti generali, oregole, espressi in maniera esplicita in un linguaggio compatto. Questo le conferisce la **capacità inferenziale**, ovvero derivare nuovi fatti da quelli memorizzati.
Il lato negativo è che, avendo un linguaggi più espressivo, è **meno efficiente** il meccanismo inferenziale. Serve quindi trovare il giusto bilanciamento da *espressività* del linguaggio e *complessità* del meccanismo inferenziale.

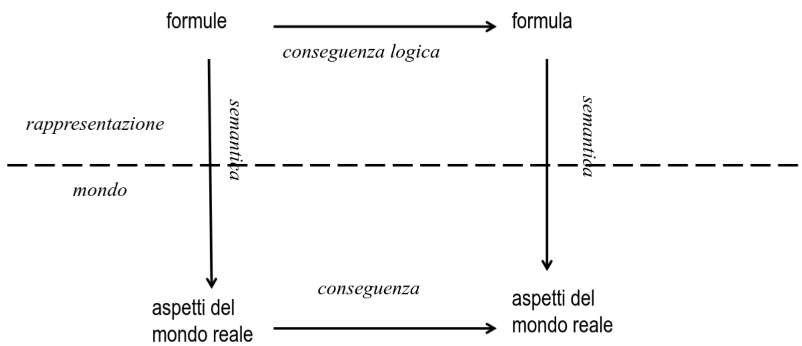
6.2 Logica

Le KB sono costituite da enunciati espresse secondo le regole della **sintassi**. La **semantica** invece ne esprime il significato. Un **modello** è una configurazione dei valori di verità che si possono assegnare alle variabili di una formula.

6.2.1 Formalismo

Un formalismo per la rappresentazione della conoscenza si compone di:

- Una **sintassi**: un linguaggio composto da un vocabolario e da regole per la formulazione degli enunciati
- Una **semantica**: stabilisce una corrispondenza tra gli enunciati e ifatti del mondo
- Un **meccanismo inferenziale** che ci consente di inferire nuovi fatti



Facendo il paragone con l'agente, le formule sono le sue configuraziini fisiche e il ragionamento è il processo di costruzione di nuove configurazioni a partire dalle vecchie. Il ragionamento logico deve assicurare che le nuove configurazioni siano effettive conseguenze sul mondo causate dalle vecchie configurazioni.

7 Logica proposizionale

7.1 Sintassi

La sintassi è la seguente, rappresentata in BNF:

$$\begin{aligned} formula &\to formulaAtomica | formulaComplessa \\ formulaAtomica &\to True | False | simbolo \\ simbolo &\to P | Q | R | \dots \\ formulaComplessa &\to \neg formula \\ &| (formula \wedge formula) \\ &| (formula \vee formula) \\ &| (formula \Rightarrow formula) \\ &| (formula \Leftrightarrow formula) \end{aligned} \tag{3}$$

7.2 Semantica

La logica proposizionale segue una semantica **composizionale**, dove il significato di una frase è determinato dal significato dei suoi componenti a partire dai *simboli proposizionali*. Di seguito la ravola di verità:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

7.3 Conseguenza logica

Definizione 7.3.1 (Conseguenza logica)**.** *Una formula α è una conseguenza logica di un insieme di formule KB se e solo se in ogni modello di KB, anche α è vera ($KB \models \alpha$).*

Indichiamo con $M(KB)$ i modelli dell'insieme di formule in KB e con $M(\alpha)$ l'insieme delle interpretazioni che rendono α vera, ovvero i suoi **modelli**.

$$KB \models \alpha \Leftrightarrow M(KB) \subseteq M(\alpha) \tag{4}$$

7.3.1 Model checking

Un modo per determinare la conseguenza logica è quello di enumere i *modelli* e mostrare che la formula α vale in tutti quelli in cui è vera la KB.

Esempio 7.3.1 (Wumpus World)**.** Partendo dall'esempio 6.1.1 abbiamo che la KB iniziale, KB_0, è costituita dalle regole descritte nella definizione dell'esercizio:

$$\begin{gathered} \neg W_{1,1} \quad \neg P_{1,1} \\ B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2,} \vee P_{3,1}) \\ B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) \\ \vdots \end{gathered}$$

Il primo passo dell'agente è spostarsi in $[2,1]$ dato che in $[1,1]$ non ha percepito niente. Abbiamno quindi:

$$KB_1 = KB_0 \cup \{\neg B_{1,1}, B_{2,1}, \neg F_{1,1}, \neg F_{2,1}, \dots\}$$

e rappresentiamo le domande sulla presenza o meno di pozzi come:

$$KB_1 \models \neg P_{1,2}$$
$$KB_1 \models \neg P_{2,2}$$
$$KB_1 \models \neg P_{3,1}$$

Sapendo da KB_0 che non ci sono pozzi nella casella $[1,1]$ e che c'è un pozzo nella stanza adiacente solo se ci percepisce la brezza, formuliamo le seguenti proposizioni:

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

e concludiamo che non c'è brezza in $[1,1]$ e c'è in $[2,1]$, ovvero $\neg B_{1,1}$ e $B_{2,1}$.
Ci rimangono quindi tre configurazioni possibili dato che abbiamo:

$$KB_1 \models \neg P_{1,2}$$
$$KB_1 \models P_{2,2} \vee P_{3,1}$$

e sono quelle in cui i pozzi sono in $[3,1]$ oppure in $[2,2]$ oppure in entrambi.i

7.3.2 SAT

Un altro approccio alla dimostrazione della conseguenza logica si basa su tre principi:

- **Equivalenza logica**: due formule α e βsono equivalenti se sono vere nello stesso insieme di modelli
$$\alpha \equiv \beta \Leftrightarrow \alpha \models \beta \wedge \beta \models \alpha \tag{5}$$
  Alcune leggi fondamentali per l'equivalenza sono:
  - *Commutatività*: $(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad (\alpha \vee \beta) \equiv (\beta \vee \alpha)$
  - *Associatività*: $((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad ((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$
  - *Eliminazione della doppia negazione*: $\neg(\neg\alpha)$
  - *Contrapposizione*: $(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$
  - *Eliminazione dell'implicazione*: $(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$
  - *Eliminazione del bicondizionale*: $(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$
  - *De Morgan*: $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$
  - *Distributività*: $(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma) \quad (\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$
- **Validità**: una formula α è valida se e solo se è vera in tutte le sue interpretazioni. In quel caso sono anche dette **tatutologie**.

  Teorema 7.3.1 (Teorema di deduzione e refutazione)**.** Date due formule α e β, allora $\alpha \models \beta \Leftrightarrow (\alpha \Rightarrow \beta)$. Possiamo riscriverlo, usando le leggi appena elencate, anche come $\alpha \models \beta \Leftrightarrow (\alpha \wedge \neg\beta)$, che ci permette di fare la dimostrazione per **assurdo**.

- **Soddisfacibilità**: una formula α è soddisfacibile se e solo se esiste una interpretazione in cui α è vera (ovvero se esiste un modello di α). La determinazione della soddisfacibilità è il problema **SAT**.

Si noti che *validità* e *soddisfacibilità* sono connesse:

- α è valida se e solo se $\neg\alpha$ è insoddisfacibile
- α è soddisfacibvile se e solo se $\neg\alpha$ non è valida

Definizione 7.3.2 (Forma a clausole)**.** *La forma a clausole è la* ***forma normale congiuntiva*** *(CNF), ovvero una congiunzione di disgiunzioni di letterali (un simbolo o la sua negazione). È sempre possibile ottenerla con trasformazioni che preservano l'equivalenza logica.*

Per eseguire una trasformazione in forma a clausole bisogna seguire i seguenti passi:

1. Eliminazione del $\Leftrightarrow$
2. Eliminazione del $\Rightarrow$
3. Portare le negazioni all'interno tramite De Morgan
4. Distribuire $\vee$ su $\wedge$

Esempio 7.3.2. Partendo dall'esempio 6.1.1, trasformiamo $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$:

1. $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
2. $(\neg B_{1,1} \vee (P_{1,2} \vee P_{2,1})) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$
3. $(\neg B_{1,1} \vee (P_{1,2} \vee P_{2,1})) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$
4. $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

che possiamo riscrivere come

$$\{\neg B_{1,1}, P_{1,2}, P_{2,1}\}\{\neg P_{1,2}, B_{1,1},\}\{\neg P_{2,1}, B_{1,1}\}$$

7.3.3 Deduzione

Un altro modo per dimostrare la conseguenza logica è utilizzare un **sistema di deduzione**, che denotiamo come $KB \vdash A$. La deduzione avviene specificando delle **regole di inferenza** con le seguenti caratteristiche:

- Devono derivare **solo** formule che sono conseguenza logica
- Devono derivare **tutte** le formule che sono conseguenza logica

Definizione 7.3.3 (Correttezza)**.** *Tutto ciò che è derivabile è conseguenza logica, le regole preservano la verità.*

$$KB \vdash \alpha \Rightarrow KB \models \alpha \tag{6}$$

Definizione 7.3.4 (Completezza)**.** *Tutto ciò che è conseguenza logica è ottenibile tramite il sistema di deduzione.*

$$KB \models \alpha \Rightarrow KB \vdash \alpha \tag{7}$$

Alcune regole di inferenza sono:

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta} \quad \text{Modu ponens} \tag{8}$$

$$\frac{\alpha \wedge \beta}{\alpha} \quad \text{Eliminazione dellÁND} \tag{9}$$

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{Introduzione della doppia implicazione} \tag{10}$$

$$\frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta} \quad \text{Eliminazione della doppia implicazione} \tag{11}$$

Esempio 7.3.3 (Wumpus)**.** Partendo dalle stesse assunzioni fatte nell'esempio 7.3.1, voglio chiedermi se posso dimostrare con le regole di inferenza che non c'è un pozzo in $[1, 1]$, ovvero $\neg P_{1,2}$.

$$\begin{aligned}
R_6 : (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) &\qquad (R_2, \Leftrightarrow E)\\
R_7 : (P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1} &\qquad (R_6, \wedge E)\\
R_8 : \neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1}) &\qquad (R_7, \text{contrapposizione})\\
R_9 : \neg(P_{1,2} \vee P_{2,1}) &\qquad (R_4, R_8, \text{Modus ponens})\\
R_{10} : \neg P_{1,2} \wedge \neg P_{2,1} &\qquad (R_9, \text{De Morgan})\\
R_{11} : \neg P_{1,2} &\qquad (R_10, \wedge E)
\end{aligned}$$

Anche la deduzione può quindi essere visto come problema di ricerca, dove vanno definite:

- **Direzione** della ricerca: nella dimostrazione di teoremi conviene procedere all'*indietro*
- **Strategia** della ricerca:
  - *Completezza*: le regole della deduzione naturale sono un un insieme completo, se lo è anche l'algoritmo siamo a posto
  - *Efficienza*: è un problema decidibile ma NP-Completo

In generale per risolvere una proposizione meno regole abbiamo e meglio è, senza però rinunciare alla completezza.
Dati l e m letterali positivi o negativi e l_i e m_j di segno opposto, la regola di risoluzione possiamo scriverla in generale come:

$$\frac{\{l_1, \dots, l_i, \dots, l_k\}\{m_1, \dots, m_j, \dots, m_n\}}{\{l_1, \dots, l_{i-1}, l_{i+1}, \dots, l_k\}\{m_1, \dots, m_{j-1}, m_{j+1}, \dots, m_n\}} \tag{12}$$

da cui poi possiamo costruirci un **grafo di risoluzione**.

7.4 Algoritmi

Di seguito alcuni algoritmi per determinare se è vera una conseguenza logica a partire da una KB.

7.4.1 TV-Consegue

Questo algoritmo enumera tutte le possibili interpretazioni di KB, e per ciascuna interpretazione se soddisfa la KB controlla che soddisfi anche α. Basta trovare una singola interpretazione che soddisfa la KB ma non α per determinare una risposta negativa. Avremo quindi, dati k simboli, 2^k possibili interpretazioni.

```
function TV-Consegue?(KB, a) // Restituisce true oppure false
inputs: KB, la base di conoscenza, una formula della logica proposizionale
a, la query, una formula della logica proposizionale
simboli = una lista dei simboli proposizionali contenuti in KB e a
return TV-Verifica-Tutto(KB, a, simboli, { })

function TV-Verifica-Tutto(KB, a, simboli, modello) // Restituisce true oppure false
if Vuoto?(simboli) then
if PL-Vero?(KB, modello) then return PL-Vero?(a, modello)
else return true // Quando KB false, restituisce sempre true
else do
P = Primo(simboli); resto = Resto(simboli)
return TV-Verifica-Tutto(KB, a, resto, modello = {P = true})
and
TV-Verifica-Tutto(KB, a, resto, modello = {P = false})
```

Esempio 7.4.1. Supponiamo di voler verificare la seguente conseguenza logica:

$$(\neg a \vee b) \wedge (a \vee c) \models (b \vee c)$$

Ci costruiamo la tabella di verità: Per poi selezionare solo le righe in cui la KB è vera e verificare se

a	b	c	$\neg a \vee b$	$a \vee c$
T	T	T	T	T
T	T	F	T	T
T	F	T	F	T
T	F	F	F	T
F	T	T	T	T
F	T	F	T	F
F	F	T	T	T
F	F	F	T	F

la nostra formula è sempre vera: Quindi la risposta è sì.

a	b	c	$\neg a \vee b$	$a \vee c$	$b \vee c$
T	T	T	T	T	T
T	T	F	T	T	T
F	T	T	T	T	T
F	F	T	T	T	T

Applicando l'algoritmo 7.4.1 abbiamo la seguente esecuzione:

```
TV-VERIFICA-TUTTO(KB, formula, [a, b, c], { })
TV-VERIFICA-TUTTO(KB, formula, [b, c], {a=T})
TV-VERIFICA-TUTTO(KB, formula, [c], {a=T, b=T})
TV-VERIFICA-TUTTO(KB, formula, [ ], {a=T, b=T, c=T}) // OK
TV-VERIFICA-TUTTO(KB, formula, [ ], {a=T, b=T, c=F}) // OK
TV-VERIFICA-TUTTO(KB, formula, [c], {a=T, b=F})
```

```
TV-VERIFICA-TUTTO(KB, formula, [ ], {a=T, b=F, c=T}) // OK
TV-VERIFICA-TUTTO(KB, formula, [ ], {a=T, b=F, c=F}) // OK
TV-VERIFICA-TUTTO(KB, formula, [b, c], [a=F])
etc...
```

7.4.2 DPLL

Questo algoritmo parte da una KB in forma a clausole e prende in input una formula in CNF ed enumera ricorsivamente in profondità tutte le possibili interpretazioni alla ricerca di un modello. Per avere un miglioramento sull'algoritmo 7.4.1 applico tre clausole:

- **Terminazione anticipata**: si può decidere sulla verità di una clausola anche con interpretazioni parziali, ovvero quando ho degli OR basta che un simbolo sia vero mentre quando ho degli AND basta che unpo sia falso per rendere falsa l'intera interpretazione
- **Euristica dei simboli puri**: un simbolo puro è un simbolo che appare con lo stesso segno in tutte le clausole (trascurando eventualmente quelle già rese vere). Possono poi essere assegnati a True se il letterale è positivo o a False se è negativo
- **Euristica delle clausole unitarie**: una clausola in cui è rimasto un solo letterale non assegnato

```
function DPLL-Soddisfacibile?(s) returns true oppure false
inputs: s, una formula della logica proposizionale
clausole = insieme di clausole nella rappresentazione CNF di s
simboli = una lista di tutti i simboli proposizionali in s
return DPLL(clausole, simboli, { })

function DPLL(clausole, simboli, modello) returns true oppure false
if ogni clausola in clausole vera in modello then return true
if qualche clausola in clausole falsa in modello then return false
P, valore = Trova-Simbolo-Puro(simboli, clausole, modello)
if P diverso da null then return DPLL(clausole, simboli - P, modello = {P = valore})
P, valore = Trova-Clausola-Unitaria(clausole, modello)
if P diverso da null then return DPLL(clausole, simboli-P, modello = {P = valore})
P = Primo(simboli); resto = Resto(simboli)
return  DPLL(clausole, resto, modello = {P = true})
or
DPLL(clausole, resto, modello = {P = false})
```

Esempio 7.4.2. Supponiamo di voler verificare la seguente conseguenza logica:

$$\{\neg B_{1,1}, P_{1,2}, P_{2,1}\}\{\neg P_{1,2}, B_{1,1}\}\{\neg P_{2,1}, B_{1,1}\}\{\neg B_{1,1}\} \models \{\neg P_{1,2}\}$$

Aggiungiamo alla KB la clausola $\{P_{1,2}\}$ e verifichiamo con SAT se l'insieme è insoddisfacibile:

1. La clausola $\{P_{1,2}\}$ è unitaria, quindi $P_{1,2} = True$. Di conseguenza $\{\neg B_{1,1}, P_{1,2}, P_{2,1}\}$ e $\{P_{1,2}\}$ sono soddisfatte e rimaniamo con
$$\{\neg P_{1,2}, B_{1,1}\}\{\neg P_{2,1}, B_{1,1}\}\{\neg B_{1,1}\}$$
2. $P_{2,1}$ è un simbolo puro ed essendo negativo sarà uguale a False, quindi la clausola $\{\neg P_{2,1}, B_{1,1}\}$ è soddisfatta e rimaniamo con
$$\{\neg P_{1,2}, B_{1,1}\}\{\neg B_{1,1}\}$$

Dato che non esistono modelli possiamo dire che $\neg P_{1,2}$ è conseguenza logica della KB

Questo algoritmo è **completo** e **termina sempre**. Alcuni miglioramenti sono:

- Se possibile scomporre in sotto problemi indipendenti (quando non hanno simboli in comune)
- Ordinare le variabili per frequenza di comparizione
- Backtracing intelligente

7.4.3 WalkSAT

Definiamo la formulazione di un problema SAT in ambito locale:

- **Stati**: sono le interpretazioni, assegnamenti completi
- **Obiettivo**: un assegnamento che soddisfa tutte le clausole (*modello*)

Si parte da un assegnamento *casuale* e ad ogni passo si cambia il valore di un simbolo proposizionale (**flip**). La valutazione di uno stato avviene controllando il numero di clausole soddisfatte.

Ad ogni passo viene scelta a caso una clausola non soddisfatta e individua un simbolo da modificare, scegliendo con probabilità p tra:

- **Random walk**: il simbolo è scelto a caso
- **Ottimizzazione**: viene scelto il simbolo che rende più clausole soddisfatte

Dopo un certo numero di flip predefinito, l'algoritmo si arrende.

```
function WalkSAT(clausole, p, max_flips) returns un modello o fallimento
modello = assegnamento casuale di valori di verita ai simboli in clausole

for i = 1 to max_flips do
if modello soddisfa clausole then return modello
clausola = una clausola, falsa in modello, scelta casualmente nell'insieme clausole
if Random(0, 1) =< p then inverti il valore in modello di un simbolo scelto
casualmente in clausola
else inverti il valore di verita del simbolo in clausole che massimizza il numero
di clausole soddisfatte

return fallimento
```

Esempio 7.4.3 (WalkSAT)**.** L'obiettivo è quello di massimizzare il numero di clausole soddisfatte tra le seguenti:

$$\{\neg B_{1,1}, P_{1,2}, P_{2,1}\}\{\neg P_{1,2}, B_{1,1}\}\{\neg P_{2,1}, B_{1,1}\}\{\neg B_{1,1}\}$$

Un esempio di esecuzione dell'algoritmo è il seguente:

1. Configurazione di *partenza*: $[B_{1,1} = F, P_{1,2} = T, P_{2,1} = T]$
2. *Random walk*: la prima e la quarta clausola sono soddisfatte, scelgo seconda e faccio un flip a caso di $B_{1,1}$ ottenendo $[B_{1,1} = T, P_{1,2} = T, P_{2,1} = T]$
3. L'unica non soddisfatta è la quarta, posso solo fare un flip di $B_{1,1}$ ottenendo $[B_{1,1} = F, P_{1,2} = T, P_{2,1} = T]$
4. *Random walk*: la prima e la quarta clausola sono soddisfatte, scelgo la seconda e faccio un flip a caso di $P_{1,2}$ ottenendo $[B_{1,1} = F, P_{1,2} = F, P_{2,1} = T]$
5. *Ottimizzazione*: l'unica non soddisfatta è la terza, faccio un flip di $P_{2,1}$ ottenendo $[B_{1,1} = F, P_{1,2} = F, P_{2,1} = F]$

Se il limite $max_flips = \infty$ e l'insieme di clausole è soddisfacibile, prima o poi termina, ma se non lo è non terminerà mai. Non possiamo quindi usarlo per verificare l'insoddisfacibilità.

7.4.4 Confronto

Se un problema è **sotto-vincolato** (ha molte soluzioni) è più probabile che *WalkSAT* trovi una soluzione in tempi brevi.

Esempio 7.4.4 (3-SAT)**.** Dato il seguente problema 3-SAT, ovvero con clausole di 3 letterali:

$$\{\neg D, \neg B, C\}\{B, \neg A, \neg C\}\{\neg C, \neg B, E\}\{E, \neg D, B\}\{B, E, \neg C\}$$

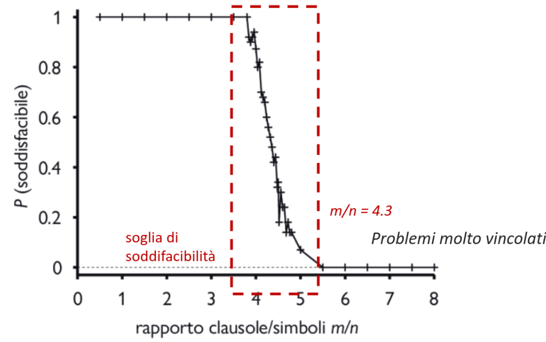
Abbiamo 32 possibili interpretazioni con 16 possibili soluzioni. Questo sarebbe facile da risolvere con *Walk-SAT*.

È importante nel capire il livello di difficoltà di un problema SAT il rapporto tra numero di **clausole** e numero di **simboli**

$$\frac{m}{n} \tag{13}$$

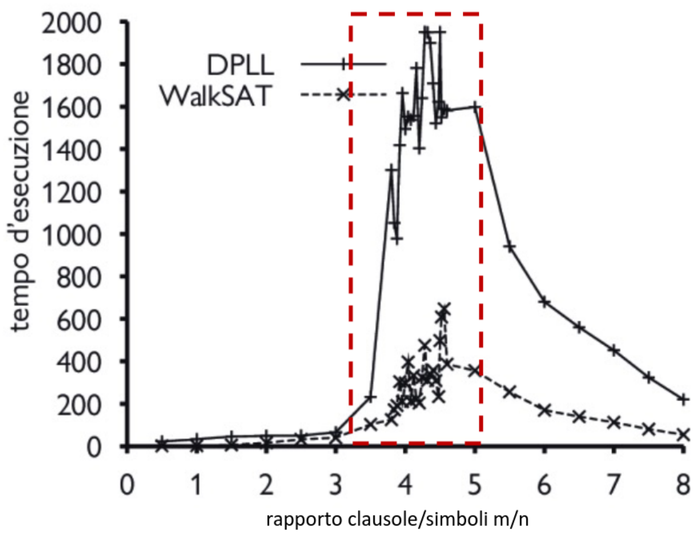
Infatti, più è grande il rapporto e più vincolato è il problema.

Esempio 7.4.5. Supponiamo di avere $n = 50$ simboli e di avere m clausole che variano.



Su 100 problemi generati a caso vediamo che 4.3 è la soglia oltre la quale un problema diventa difficile da risolvere.

Vediamo il **confronto** tra l'algoritmo DPLL e WalkSAT:



Notiamo che i problemi vicini alla *soglia di soddisfacibilità* sono molto più difficili da risolvere rispetto a quelli più lontani. Inoltre vediamo che quando un problema è poco vincolato i due algoritmi performano allo stesso modo mentre intorno alla soglia *WalkSAT* è nettamente migliore.

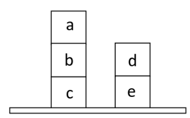
8 Logica del prim'ordine

Il problema principale della logica proposizionale è che non ha la potenza espressiva per descrivere un ambiente con molti oggetti in modo conciso. Nella logica del prim'ordine abbiamo assunzioni più ricche riguardo la natura della realtà: *oggetti*, *relazioni*, *proprietà*.

8.1 Concettualizzazione

Il primo passo è quello di decidere quali sono le cose di cui vogliamo parlare, dobbiamo quindi definire gli **oggetti**, che possono essere identificati da *simboli* o da *funzioni* che li mettono in relazione con altri oggetti. L'insieme degli oggetti rilevanti costituiscono il **dominio del discorso**, che potrebbe anche essere infinito.
Ci sono poi le **relazioni**, le **proprietà** (unarie) e le funzioni (biettive).

Esempio 8.1.1 (Mondo dei blocchi)**.** Siamo in un mondo di blocchi, che ad esempio può essere strutturato nel seguente modo:



Per prima cosa definiamo il *dominio* composto dai blocchi presenti:

$$\{a, b, c, d, e\}$$

Poi individuiamo le *funzioni* rilevanti ad identificare gli oggetti, nel nostro caso solo quella che, dato un blocco, identifica quello che ci sta sopra:

$$Hat(b) = a$$

Infine abbiamo le *relazioni*:

$$\begin{gathered} On = \{< a, b >, < b, c >, < d, e >\} \\ Clear = \{a, d\} \\ Table = \{c, e\} \\ Block = \{a, b, c, d, e\} \end{gathered}$$

Riassumiamo la nostra **concettualizzazione** come la seguente tupla composta da *dominio*, *funzioni* e *relazioni*:

$$< \{a, b, c, d, e\}, \{Hat\}, \{On, Clear, Table, Block\} >$$

8.2 Sintassi

8.2.1 Simboli

Gli elementi sintattici di base sono i simboli usati per indicare oggetti, relazioni e funzioni. Ne abbiamo di tre tipi:

- **Simboli di costante**: rappresentano gli oggetti
- **Simboli di predicato**: rappresentano le relazioni
- **Simboli di funzione**: rappresentano le funzioni

Ogni simbolo degli ultimi due tipi ha una specifica **arità**, che ne indica il numero di *argomenti*.
Quando gli oggetti non sono specificati, usiamo le **variabili**, che ci consentono di formulare predicati che possono essere veri o falsi a seconda di che valore assegnamo.

8.2.2 Termini

Un termine è un'espressione logica che si riferisce ad un oggetto. Ha la seguente sintassi:

$$Termine \rightarrow Costante | Variabile | Funzione(Termine, \ldots)$$

Esempio 8.2.1. Alcuni esempi di termini ben formati:

$$\begin{gathered} f(x, y) \quad +(2,3) \quad Padre-di(Giovanni) \\ x, A, B, 2 \quad Prezzo(Banane) \quad Hat(A) \end{gathered}$$

Il simbolo di **uguaglianza** si usa per dire che due termini fanno *riferimento* allo stesso oggetto:

$$\exists x, y \quad Fratello(x, Riccardo) \wedge Fratello(y, Riccardo) \wedge \neg(x = y)$$

8.2.3 Formule

Una formula **atomica** è l'espressione più semplice e *indivisibile* che afferma una relazione tra oggetti del dominio. È composta da un *predicato* seguito da una *lista di termini* che corrispondono alla sua arità. Le formule atomiche non contengono quantificatori, connettivi logici o altre formule come componenti. Ha la seguente sintassi:

$$\begin{aligned} \text{Formula-atomica} \rightarrow & True | False | \\ & \text{Termine} = \text{Termine} \mid \\ & \text{Predicato(Termine}, \ldots) \end{aligned}$$

Ci sono poi le formule **complesse**, nelle quali possiamo usare *connettivi logici* e *quantificatori.* Hanno la seguente sintassi:

$$\begin{aligned} \text{Formula} \rightarrow & \text{Formula-atomica} \mid \\ & \text{Formula} \quad \text{Connettivo} \quad \text{Formula} \mid \\ & \text{Quantificatore} \quad \text{Variabile} \quad \text{Formula} \mid \\ & \neg\text{Formula} \mid (\text{Formula}) \end{aligned}$$

Esempio 8.2.2. Alcune formule *atomiche* ben formate:

$$\begin{gathered} Ama(Giorgio, Lucia) \quad On(A, B) \quad Madre-di(Luigi) = Silvana \\ Amico(Padre-di(Giorgio), Padre-di(Elena)) \quad +(2,3) = 5 \quad x = 5 \end{gathered}$$

Alcune formule *complesse* ben formate:

$$\begin{gathered} On(A, B) \wedge On(B, C) \\ Studia(Paolo) \Rightarrow Promosso(Paolo) \end{gathered}$$

8.2.4 Quantificatori

Esistono due tipi di quantificatori:

- **Universale**: la relazione si applica a tutti gli elementi del dominio (e.g. $\forall x \quad Ama(x, Gelato)$)

- **Esistenziale**: la relazione si applica ad almeno un elemento del dominio (e.g. $\exists x \quad Mela(x) \wedge Rossa(x)$)

Note 8.2.1. L'ordine dei quantificatori è importante.

Ogni quantificatore ha un suo **scope**, ad esempio nel seguente predicato

$$\forall x (\exists y \quad Ama(x, y))$$

$\exists y$ ha come scope $Ama(x, y)$ mentre $\forall x$ ha come scope $(\exists y \quad Ama(x, y))$.

8.2.5 Linguaggio

Il linguaggio della logica del prim'ordine è descritto dal seguente vocabolario:

$$
\begin{aligned}
&\text{Connettivo} \rightarrow \wedge|\vee|\neg| \Rightarrow | \Leftrightarrow | \Leftarrow \\
&\text{Quantificatore} \rightarrow \forall|\exists \\
&\text{Variabile} \rightarrow x|y|\dots|a|\dots|s|\dots \\
&\text{Costante} \rightarrow A|B|\dots|Mario|Pippo|\dots|1|\dots \\
&\text{Funzione} \rightarrow Hat|Padre-di|+|-|\dots \\
&\text{Predicato} \rightarrow On|Clear|\geq|<|\dots
\end{aligned}
$$

Quando utilizziamo le variabili nell'ambito dei quantificatori sono definite **legate**, altrimenti sono **libere**.

Definizione 8.2.1 (Formula chiusa)**.** *Una formula che non contiene occorrenze di variabili libere.*

Definizione 8.2.2 (Formula aperta)**.** *Una formula che contiene occorrenze di variabili libere.*

Definizione 8.2.3 (Formula ground)**.** *Una formula che non contiene variabili.*

Osservazione 8.2.1 (Precedenza)**.** Nel linguaggio della logica del prim'ordine è importante la precedenza tra gli operatori logici. Ad esempio:

$$
\begin{aligned}
\forall x \quad Persona(x) \Rightarrow Sesso(x) = M \vee Sesso(x) = F \vee Sesso(x) = NB \\
\vee Sesso(x) = GQ \vee Sesso(x) = GF \vee Sesso(x) = A
\end{aligned}
$$

deve essere interpretata come:

$$
\begin{aligned}
\forall x \quad (Persona(x) \Rightarrow ((Sesso(x) = M) \vee (Sesso(x) = F) \vee (Sesso(x) = NB) \\
\vee (Sesso(x) = GQ) \vee (Sesso(x) = GF) \vee (Sesso(x) = A)))
\end{aligned}
$$

8.3 Semantica

La logica del prim'ordine usa una semantica di tipo **dichiarativo**, che consiste nello stabilire una corrispondenza tra:

- I *termini* del linguaggio e gli *oggetti* del mondo
- Le *formule chiuse* e i *valori di verità*

8.3.1 Componenti

Le componenti principali della semantica sono:

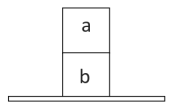
- **Universo** del discorso: un insieme non vuoto di elementi che rappresenta l'insieme di tutti gli *oggetti* che si prendono in considerazione
- **Assegnazione di valori**:
  - Alle *costanti* si assegna un elemento specifico dell'universo
  - Alle *variabili* si possono assegnare elementi dell'universo attraverso una *funzione*
  - Alle *funzioni* si assegna una mappatura da una sequenza di elementi dell'universo ad un singolo elemento dell'universo (rispettando la sua *arità*)
  - Ai *predicati* si assegna una relazione sull'universo (rispettando la sua *arità*)
- **Verità di formule**:
  - Una formula *atomica* è vera se l'interpretazione assegna ai suoi termini una sequenza che rientra nella relazione specificata dal predicato
  - Una formula *complessa* viene valutata sulla base dei suoi componenti usando il significato dei *connettori logici* e dei *quantificatori*

8.3.2 Interpretazione

Una interpretazione I stabilisce una *corrispondenza* tra elementi atomici del linguaggio ed elementi della concettualizzazione. Interpreta:

- I simboli di *costante* come elementi del dominio D
- I simboli di *funzione* come funzioni da n-uple di D in D
- I simboli di *predicato* come insiemi di n-unple (relazioni)

Esempio 8.3.1. Partiamo da un'altra versione del mondo dei blocchi:



concettualizzato come segue:

$$
\begin{gathered}
On(A, B) \\
Clear(A) \\
Table(B)
\end{gathered}
$$

Una possibile interpretazione è la seguente:

$$
\begin{aligned}
& I(A) = a \quad I(B) = b \\
& I(On) = \{< a, b >\} \\
& I(Clear) = \{a\} \\
& I(Table) = \{b\}
\end{aligned}
$$

ma lo è anche:

$$
\begin{aligned}
& I(A) = a \quad I(B) = b \\
& I(On) = \{< b, a >\} \\
& I(Clear) = \{b\} \\
& I(Table) = \{a\}
\end{aligned}
$$

Quindi il significato di un termine o di una formula composta è determinato in funzione del significato dei suoi componenti.
Nel caso dei quantificatori:

- **Universale**: $\forall x\, A(X)$ è vera se lo è per ciascun elemento del dominio di A. Se il dominio è *finito* equivale ad una serie di $\wedge$. Data la forza del quantificatore universale, spesso si restringe il suo campo d'azione mediante $\Rightarrow$
- **Esistenziale**: $\exists x\, A(x)$ e vera se esiste almeno un elemento del dominio per cui A è vera. Se il dominio è *finito* equivale ad una serie di $\vee$. Data la debolezza del quantificatore esistenziale, di solito si usa con $\wedge$

Da questo possiamo derivare alcune proprietà che mettono in relazione i due quantificatori (a destra le proprietà della logica da cui sono derivate):

$$
\begin{array}{ll}
\forall x\, \neg P(x) \equiv \neg \exists x\, P(x) & \neg P \wedge \neg Q \equiv \neg (P \vee Q) \\
\neg \forall x\, P(x) \equiv \exists x\, \neg P(x) & \neg (P \wedge Q) \equiv \neg P \vee \neg Q \\
\forall x\, P(x) \equiv \neg \exists x\, \neg P(x) & P \wedge Q \equiv \neg (\neg P \vee \neg Q) \\
\neg \forall x\, \neg P(x) \equiv \exists x\, P(x) & P \vee Q \equiv \neg (\neg P \wedge \neg Q)
\end{array}
$$

Note 8.3.1. La semantica standard, che segue la logica classica, è spesso molto prolissa anche per esprimere concetti molto semplici. Per combattere questo problema esiste la semantica dei **database** che parte dalle seguenti ipotesi per essere più concisa:

- *Nomi unici*: simboli e oggetti distinti, ogni costante fa riferimento ad un oggetto distinto
- *Mondo chiuso*: tutto ciò di cui non si sa che è vero, è falso, quindi le formule atomiche non conosciute come vere sono false
- *Chiusura del dominio*: esistono solo gli oggetti di cui si parla. Ogni modello contiene un numero di elementi del dominio non superiore a quello degli elementi denominati dai simboli di costante

8.3.3 Knowledge Base

Le interazioni con la Knowledge Base in logica del prim'ordine sono fatte sfruttando:

- **Asserzioni**, ad esempio
$$TELL(KB, Re(Giovanni))$$
- **Interrogazioni**, ad esempio
$$ASK(KB, Persona(Giovanni)) \longrightarrow Si$$

Esempio 8.3.2 (Wumpus World)**.** Possiamo implementare l'esempio del Wumpus World (6.1.1) in FOL, descrivendo tutto in maniera più precisa. Alcuni casi:

- **Percezioni**: possiamo rappresentarle come un predicato binario $Percezione(5 - upla, t)$
- **Regole** per le percezioni: ad esempio per lo scintillio
$$\forall t, s, b, m, c\ Percezione([s, b, Scintillio, m, c], t) \Rightarrow Scintillio(t)$$
$$\forall t, s, b, m, c\ Percezione([s, b, None, m, c], t) \Rightarrow \neg Scintillio(t)$$
- Descrizione della **mappa**: ad esempio l'adiacenza di una casella
$$\forall x, y, a, b Adiacente([x, y], [a, b]) \Leftrightarrow (x = a \wedge (y = b-1 \vee y = b+1)) \vee (y = b \wedge (x = a-1 \vee x = a+1))$$

8.4 Inferenza

Dobbiamo eliminare i quantificatori. Introduciamo prima il concetto di **sostituzione**: $A[x/g]$ è il risultato della sostituzione di g per x in A.

8.4.1 Istanziazione

L'instanziazione **universale** prevede di inferire tutte le formule ottenute sostituendo un termine *ground* g a una variabile quantificata universalmente x.

$$\frac{\forall x\ A[x]}{A[x/g]} \tag{14}$$

Esempio 8.4.1. Data la proposizione

$$\forall x\ Re(x) \wedge Avido(x) \Rightarrow Malvagio(x)$$

si possono ottenere:

$$Re(Giovanni) \wedge Avido(Giovanni) \Rightarrow Malvagio(Giovanni)$$
$$Re(Padre(Giovanni)) \wedge Avido(Padre(Giovanni)) \Rightarrow Malvagio(Padre(Giovanni))$$

L'instanziazione **esistenziale** prevede di sostituire una variabile quantificata esistenzialmente con un unico nuovo simbolo costante.

$$\frac{\exists x\ A[x]}{A[x/k]} \tag{15}$$

Se $\exists$ non compare nello scope di $\forall$, k è una costante nuova chiamata **costante di Skolem**, altrimenti va introdotta una **funzione di Skolem** nelle variabili quantificate universalmente. Alcuni esempi:

$$\exists x\ Padre(x, G) \longrightarrow Padre(K, G)$$
$$\forall x \exists y\ Padre(x, y) \longrightarrow \forall x\ Padre(x, p(x))$$

8.4.2 Grounding

La riduzione a inferenza proposizionale è detta **grounding** e prevede i seguenti passaggi:

1. Istanziazione universale
2. Istanziazione esistenziale
3. Sostituire le formule *atomiche ground* con simboli proposizionali

Il problema che rimane prima di poter applicare gli algoritmi già visti è che anche se le costanti sono in numero finito, in presenza di funzioni le istanze da creare sono infinite: ad esempio:

$$Giovanni, Padre(Giovanni), Padre(Padre(Giovanni)), \dots$$

Teorema 8.4.1 (Teorema di Herbrand)**.** Se $KB \models \alpha$, allora c'è una dimostrazione che coinvolge solo un sotto-insieme finito della KB proposizionalizzata.

Per applicare il teorema appena enunciato si può procedere *incrementalmente*:

1. Creare le istanza con le costanti
2. Creare le istanze con un solo livello di annidamento
3. Procedere livello per livello finché non siamo in grado di costruire la dimostrazione proposizionale della formula che è conseguenza logica

Se $KB \not\models$ il processo non termina. Il problema è quindi **semidecidibile**.

8.4.3 Forma a clausole

Per estendere alla logica del prim'ordine il metodo di risoluzione dobbiamo prima estendergli anche la trasformazione in forma a clausole.
Costanti, *funzioni* e predicati sono come definiti ma escludendo formule atomiche del tipo $t_1 = t_2$. Definiamo quindi una clausola come un insieme di *letterali* che rappresenta la loro disgiunzione:

$$Clausola \rightarrow \{Letterale, \dots, Letterale\} \tag{16}$$
$$Letterale \rightarrow Formula_atomica \mid \neg Formula_atomica \tag{17}$$

Una Knowledge Base è quindi un insieme di clausole.

Teorema 8.4.2. Per ogni formula chiusa α del FOL è possibile trovare in maniera **effettiva**un insieme di clausole $FC(\alpha)$ che è soddisfacibile se e solo se α lo è. Allo stesso modo per l'insoddisfacibilità.

Definizione 8.4.1 (Effettivo)**.** *Esiste una procedura che può essere eseguita per trasformare la formula α in un insieme di clausole $FC(\alpha)$ tale che valga questo risultato.*

Esempio 8.4.2 (Trasformazione in forma a clausole)**.** Vediamo passo per passo il processo di trasformazione appena descritto applicato alla frase:

$$\forall x(\forall y\, Animale(y) \Rightarrow Ama(x, y)) \Rightarrow (\exists y\, Ama(y, x))$$

1. Eliminazione delle implicazioni

$$\forall x \neg(\forall y\, Animale(y) \Rightarrow Ama(x, y)) \lor (\exists y\, Ama(y, x))$$
$$\forall x \neg(\forall y\, \neg Animale(y) \lor Ama(x, y)) \lor (\exists y\, Ama(y, x))$$

2. Negazioni all'interno

$$\forall x(\exists y \neg(\neg Animale(y) \lor Ama(x, y))) \lor (\exists y\, Ama(y, x))$$
$$\forall x(\exists y(\neg\neg Animale(y) \land \neg Ama(x, y))) \lor (\exists y\, Ama(y, x))$$
$$\forall x(\exists y(Animale(y) \land \neg Ama(x, y))) \lor (\exists y\, Ama(y, x))$$

3. Standardizzazione delle variabili: facciamo in modo che ogni quantificatore usi una variabile diversa
$$\forall x(\exists y(Animale(y) \land \neg Ama(x, y))) \lor (\exists z\ Ama(z, x))$$

4. Skolemizzazione: eliminazione dei quantificatori esistenziali
$$\forall x(Animale(F(x)) \land \neg Ama(x, F(x)))) \lor Ama(G(x), x)$$

5. Eliminazione quantificatori universali
$$(Animale(F(x)) \land \neg Ama(x, F(x)))) \lor Ama(G(x), x)$$

6. Applico la forma normale congiuntiva
$$(Animale(F(x)) \land Ama(G(x), x)) \land (\neg Ama(x, F(x)) \lor Ama(G(x), x))$$

7. Applico la notazione a clausole
$$\{Animale(F(x)) \land Ama(G(x), x)\}\{\neg Ama(x, F(x)) \lor Ama(G(x), x)\}$$

8. Separazione delle variabili: clausole diverse devono avere variabili diverse
$$\{Animale(F(x_1)) \land Ama(G(x_1), x_1)\}\{\neg Ama(x_2, F(x_2)) \lor Ama(G(x_2), x_2)\}$$

8.4.4 Unificazione

Definizione 8.4.2 (Unificazione). *Operazione mediante la quale si determina se due espressioni possono essere rese identiche mediante una sostituzione di termini a variabili. Il risultato è la* ***sostituzione*** *che rende le due espressioni identiche, detta* ***unificatore****, o FAIL, se le espressioni non sono unificabili.*

Definizione 8.4.3 (Sostituzione). *Un insieme finito di associazioni tra variabili e termini in cui ogni variabile compare una sola volta sulla sinistra. Data una sostituzione σ e un'espressione A, $A\sigma$ è un'istanza generata dalla sostituzione delle variabili con le corrispondenti espressioni.*

L'idea è di trovare l'unificatore più generale di tutti, il **Most General Unifier** (MGU).

Teorema 8.4.3. L'unificatore più generale è unico, a parte i nomi delle variabili (l'ordine non conta).

L'algoritmo di unificazione prende in input due espressioni p e q e restituisce un MGU θ se esiste, altrimenti FAIL. Appena trova espressioni non unificabili fallisce. Una causa di fallimento sono sostituzioni circolari del tipo $x = f(x)$; questo controllo si chiama **occurr check**.

```
function Unify(x, y, t =vuoto) returns una sostituzione che rende x e y identici, o
    fallimento
  if t = fallimento then return fallimento
  else if x = y then return t // caso di successo
  else if Variabile?(x) then return Unify-Var(x, y, t)
  else if Variabile?(y) then return Unify-Var(y, x, t)
  else if Composta?(x) and Composta?(y) then // es. Op(F(A,B)) = F Args(F(A,B)=(A,B)
    return Unify(Args(x), Args(y), Unify(Op(x), Op(y), t))
  else if Lista?(x) and Lista?(y) then
    return Unify(Resto(x), Resto(y), Unify(Primo(x), Primo(y), t))
  else return fallimento
```

Esempio 8.4.3. Vediamo un esempio di applicazione dell'algoritmo:

1. $UNIFY(P(A, y, z), P(x, B, z), \{\})$
2. $UNIFY((A, y, z), (x, B, z), UNIFY(P, P, \{\}))$

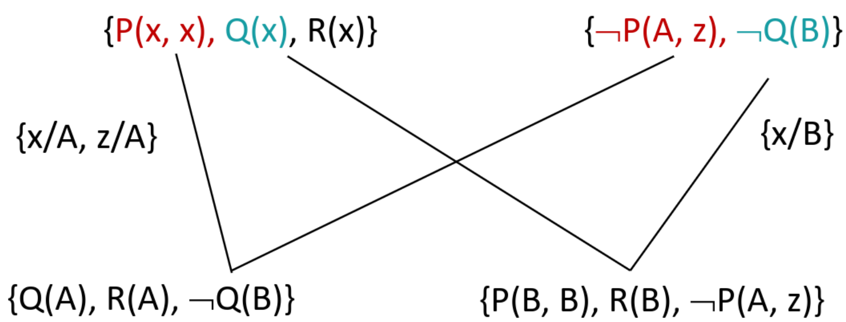
3. $UNIFY((A, y, z), (x, B, z), \{\})$
4. $UNIFY((y, z), (B, z), UNIFY(A, x, \{\}))$
5. $UNIFY((y, z), (B, z), UNIFY(x, A, \{\}))$
6. $UNIFY((y, z), (B, z), UNIFY - VAR(x, A, \{\}))$
7. $UNIFY((y, z), (B, z), \{x/A\})$
8. $UNIFY(z), (z), \{y/B, x/A\})$
9. $UNIFY z, z, \{y/B, x/A\})$
10. $\{y/B, x/A\}$

8.4.5 Risoluzione

Data una clausola ϕ che contiene A, una clausola ψ che contiene $\neg B$ e l'unificatore $\gamma = MGU(A, B)$ definiamo la **risolvente** come:

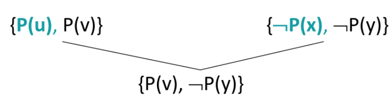
$$((\phi \ \{A\}) \subset (\psi \ \{\neg B\}))\gamma \tag{18}$$

Esempio 8.4.4. Ad esempio:

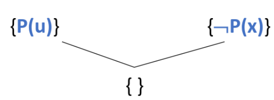


Definizione 8.4.4 (Fattori)**.** *Se un sottoinsieme dei letterali di una stessa clausola può essere unificato, allora la clausola ottenuta dopo tale unificazione si dice **fattore** delle clausole.*

Osservazione 8.4.1 (Problema dei fattori)**.** Ad esempio nel seguente caso le seguenti clausole dovrebbero produrre la clausola vuota, ma invece no.



È quindi necessario applicare il metodo di risoluzione ai fattori delle clausole.

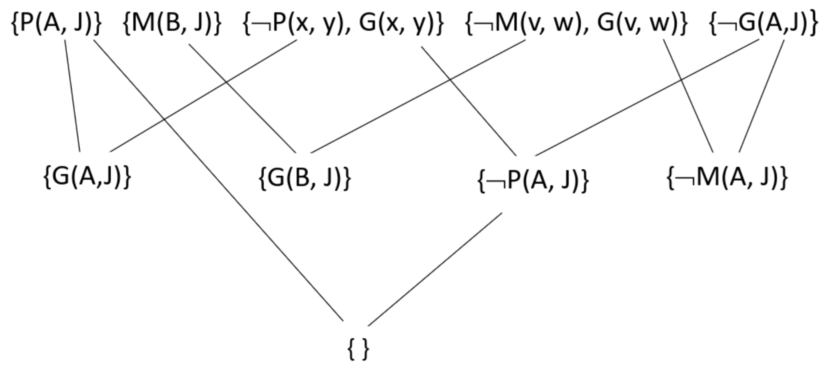


La deduzione per risoluzione è **corretta** ma **non completa**. Per ottenere la completezza si può risolvere per **confutazione**.

Esempio 8.4.5. Partiamo dalla seguente KB:

$$\{P(A, J)\}$$
$$\{M(B, J)\}$$
$$\{\neg P(x, y), G(x, y)\}$$
$$\{\neg M(v, w), G(v, w)\}$$
$$\{\neg G(A, J)\}$$

In questo caso il grafo è il seguente:



8.5 Programmazione logica

8.5.1 Clausola di Horn

Una clausola di Horn è una **disgiunzione di letterali** che contiene al massimo un letterale positivo. È un modo potente ed efficiente per rappresentare conoscenze in logica del prim'ordine. È efficace in particolare per rappresentare regole e fatti in un sistema basato su regole.

$$\{Q, \neg P_1, \ldots, \neg P_k\} \qquad k \geq 0 \tag{19}$$

Possiamo quindi scrivere la knowledge base a regole come:

$$\begin{aligned} P_1 \wedge \ldots \wedge P_k \Rightarrow Q \quad &\text{(regola)} && k > 0 \\ Q \quad &\text{(fatto)} && k = 0 \end{aligned}$$

Se la knowledge base contiene solo *clausole Horn* definite, i meccanismi inferenziali sono più semplici (lineari per il caso proposizionale) senza dover rinunciare alla completezza.

8.5.2 Inferenza

I metodi usati nei sistemi basati su regole sono di due tipi:

- Inferenza in **avanti**: si inizia con le *premesse* e si procede verso le conclusioni partendo da un insieme di fatti noti e applicando le regole per dedurre nuove informazioni. Si continua fino a quando non si raggiunge un obiettivo o non si possono fare più inferenze
- Inferenza all'**indietro**: si inizia dalle *conclusioni* e si lavora a ritroso per trovare le premesse che supportano la conclusione. Si verifica passo per passo se l'obiettivo corrente può essere dedotto dalle regole e se necessario si cerca all'indietro per dedurre o dimostrare i fatti richiesti dalle premesse.

8.5.3 Programmazione

I programmi logici sono KB costituiti di clausole Horn definite espressi come fatti e regole:

- **Fatto**: un fatto è rappresentato da una singola clausola di Horn senza premesse
- **Regola**: sono implicazioni logiche che descrivono i come i fatti sono in relazione tra loro

Note 8.5.1 (Convenzioni programmazione logica). Nella programmazione logica le **variabili** sono indicate con le lettere maiuscole e le **costanti** con quelle minuscole.

L'interpretazione può seguire due filosofie:

- **Dichiarativa**: data la regola
$$A : -B_1, \dots, B_n$$
A è vero se sono veri $B_1, \dots, B_n$ in accordo al significato logico dell'implicazione.

- **Procedurale**: sempre nel caso visto nel punto precedente, si considera la testa A come una chiamata di procedura e il corpo come una serie di procedure da eseguire in sequenza

Esempio 8.5.1. Ad esempio definiamo le seguenti regole, fatti e goal:

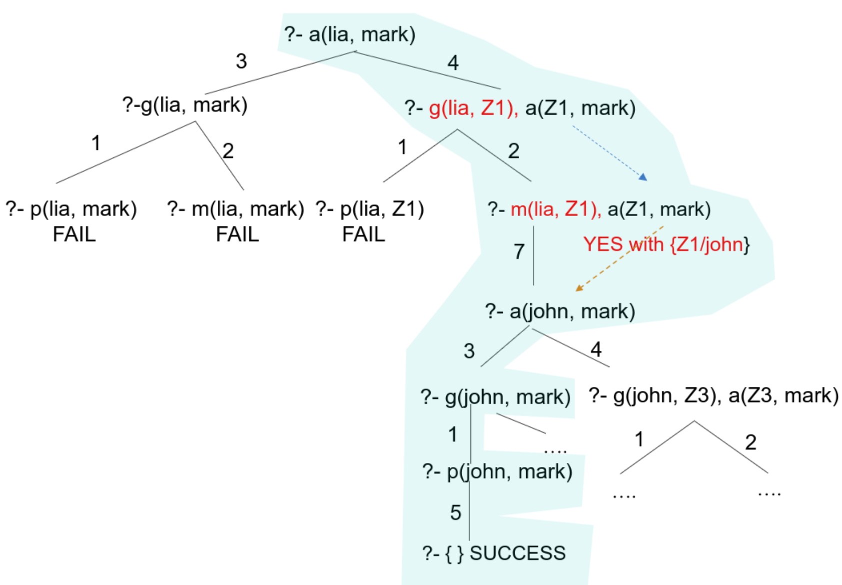
$$genitore(X, Y) : -padre(X, Y).$$
$$genitore(X, Y) : -madre(X, Y).$$
$$antenato(X, Y) : -genitore(X, Y).$$
$$antenato(X, Y) : -genitore(X, Y), antenato(Z, Y).$$
$$padre(gio, mark).$$
$$padre(gio, luc).$$
$$madre(lia, gio).$$
$$: -antenato(lia, mark :)$$

8.5.4 Risoluzione SLD

La risoluzione di tipo **Selection Linear Definite-Clauses** è una strategia **ordinata** e **completa** per le clausole di Horn. A partire da un programma P e un goal G si costruisce l'albero di risoluzione, definito come:

- Ogni **nodo** dell'albero corrisponde ad un goal
- La **radice** è $: -G_1, \dots, G_k$, il goal di partenza
- Data la radice come nodo dell'albero, questo ha tanti **discendenti** quanti sono i fatti e le regole in P la cui testa è unificabile con G_1
- I nodi che sono clausole vuote sono **successi**
- I nodi che non hanno successori sono **fallimenti**

Esempio 8.5.2. Dato l'esempio 8.5.1, il grafo di risoluzione è:



Questa strategia è **completa** per clausole di Horn definite.