Appunti Intelligenza Artificiale

Matteo Berti

A.A. 18/19

Contenuti

1 Introduzione all'IA

La definizione di **intelligenza** comprende altri concetti come il processo di pensiero, il ragionamento, il comportamento e la razionalità. L'IA può essere vista sotto quattro aspetti:

- **Agire umanamente** - il *Test di Turing*.
  Il test di Turing è un criterio per determinare se una macchina sia in grado di pensare, si basa sull'"*imitation game*". Si hanno tre partecipanti, un uomo A, una donna B, e una terza persona C. Quest'ultima è tenuta separata dagli altri due e tramite una serie di domande deve stabilire qual è l'uomo e quale la donna. Dal canto loro anche A e B hanno dei compiti: A deve ingannare C e portarlo a fare un'identificazione errata, mentre B deve aiutarlo. Affinché C non possa disporre di alcun indizio (come l'analisi della grafia o della voce), le risposte alle domande di C devono essere dattiloscritte o similarmente trasmesse. Il test di Turing si basa sul presupposto che una macchina si sostituisca ad A. Se la percentuale di volte in cui C indovina chi sia l'uomo e chi la donna è simile prima e dopo la sostituzione di A con la macchina, allora la macchina stessa dovrebbe essere considerata *intelligente*. Alcune estensioni al test includono l'utilizzo di robotica e visione artificiale.

- **Pensare umanamente** - l'approccio della *Modellazione Cognitiva*.
  Se si vuol dire che un dato programma pensa come un umano è necessario determinare come pensano gli umani. Ci sono tre modi per comprendere come funziona la mente umana:

  1. *Introspezione*: cercare di catturare i nostri pensieri mentre passano.
  2. *Esperimenti psicologici*: osservare un umano in azione.
  3. *Imaging celebrale*: osservare il cervello in azione (EEG).

  Quando si hanno sufficienti informazioni su come funziona la mente si può esprimere in un programma.

- **Pensare razionalmente**: le *"Leggi del Pensiero"* o *approccio logistico*.
  Utilizzo della logica per creare un programma che risolva ogni problema. Diciamo che un sistema è *razionale* se fa la "cosa giusta" (la scelta ideale), date le informazioni in suo possesso. Vi sono tuttavia due problemi in questo approccio:

  1. Non è semplice convertire conoscenza informale (linguaggio) nella conoscenza formale utilizzata dalla logica.
  2. Vi è una forte differenza tra risolvere un problema in teoria e nella pratica.

- **Agire razionalmente**: l'approccio dell'*agente razionale*.
  Un *agente* è un qualcosa che agisce. Un *agente razionale* è un agente che agisce per raggiungere il miglior risultato, o quando c'è incertezza il **miglior risultato atteso**. Il modello dell'agente razionale è il più adatto per il nostro scopo.

1.1 Definizioni

Ufficialmente l'IA è definita come: "*l'abilità di un computer di eseguire attività comunemente associate agli esseri umani*".
L'informatica consiste nel risolvere problemi che sappiamo come risolvere, mentre l'IA nel risolvere problemi in cui non siamo sicuri delle nostre azioni o non siamo in grado di conoscere l'ambiente.
Un'altra definizione di intelligenza è la seguente: "*l'intelligenza è l'abilità di imparare dall'esperienza, applicare conoscenza per risolvere problemi, ed adattarsi e sopravvivere in ambienti (sociali e geografici) differenti*".
Spearman quando parla di *G-Factor* (General Intelligence Factor) intende abilità cognitive matematiche e linguistiche.
Thurstone definisce sei fattori che descrivono l'intelligenza: fluidità verbale, abilità numerica, inferenza, abilità spaziale, velocità di percezione e memoria.

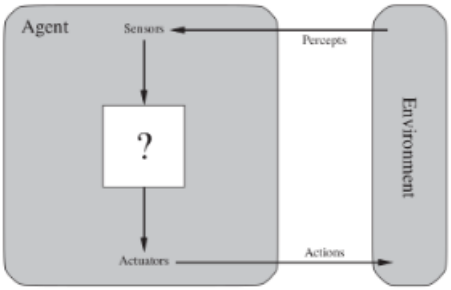
1.2 IA e logica

Esistono vari tipi di logica (proposizionale, del primo ordine, modale, ...) e varie tecniche di rappresentazione della conoscenza e del ragionamento (deduzione, induzione, abduzione, ...).

- **Deduzione**: è un processo di ragionamento che permette di *derivare* conseguenze da ciò che si è assunto vero. B può essere derivato da A se B è una conseguenza logica di A ($A \vDash B$). Data la verità dell'assunzione A segue la verità della conclusione B.
- **Induzione**: il ragionamento induttivo permette di *inferire* B da A dove B non segue necessariamente da A. Le premesse sono viste come forti prove della verità della conclusione. In generale è un procedimento che partendo da singoli casi particolari cerca di stabilire una legge universale.
- **Abduzione**: è una forma di inferenza logica che inizia con un'osservazione o un insieme di osservazioni per trovare la più semplice e più probabile spiegazione per le osservazioni. Se abbiamo una teoria T e un insieme di osservazioni O, l'abduzione è il processo di derivazione di un insieme di spiegazioni per O secondo T, e prendendo una sola spiegazione E. Formalmente:
  - La Teoria e la Spiegazione implicano le Osservazioni :: $T \cup E \vDash O$
  - La Spiegazione è consistente con la Teoria :: $T \cup E$ è *consistente*
- **Sistemi esperti**: sono sistemi basati sulla conoscenza, utilizzati per risolvere problemi in un dominio limitato e specifico. Hanno performance molto buone simili ad esperti del dominio umani. Sono in grado di esplorare grandi database di informazioni ed inferire conclusioni. Le soluzioni sono costruite dinamicamente. La ricerca e il processo di costruzione della soluzione sono basati su regole.

2 Agenti intelligenti

Un **agente intelligente** è tutto ciò che può percepire il proprio *ambiente* attraverso *sensori* ed agire su questo ambiente attraverso *attuatori*.



La **percezione** è l'input dell'agente in ogni dato istante. Mentre una **sequenza percettiva** è lo storico completo di ciò che l'agente ha percepito.
La *scelta di azione* di un agente in ogni dato momento può dipendere dall'intera sequenza percettiva osservata fino a quell'istante, ma non da qualcosa che non è stato percepito.

Il *comportamento* dell'agente è descritto dalla **Funzione Agente** che mappa ogni *sequenza percettiva* in un azione. È possibile caratterizzare questa funzione in due modi:

- *Esternamente*: in una forma tabellare, qualcosa che in realtà non sarebbe possibile in quanto le tabelle sarebbero molto grandi.
- *Internamente*: sottoforma di **Programma Agente** ovvero la concreta implementazione della *Funzione Agente* sulla macchina.

2.1 Agenti razionali

Un **agente razionale** è un agente che fa la cosa giusta, in altre parole la tabella della *Funzione Agente* è riempita correttamente. Rimane il problema di definire quale sia la "cosa giusta". È necessario considerare le *conseguenze* del comportamento dell'agente. Quando un agente è posto all'interno di un ambiente *genera* una sequenza di azioni (sequenza di stati nell'ambiente) basati sulla sequenza percettiva ricevuta. A questo punto se le azioni sono auspicabili si può dire che l'agente si comporti correttamente.
La nozione di **auspicabilità** viene catturata dalla **misura di performance**, in cui si valutano gli stati dell'ambiente, e non lo stato dell'agente.
La *misura di performance* valuta ogni sequenza data di stati dell'ambiente. La regola generale per progettare questa misura è basarsi su come si vuole che risulti l'*ambiente* alla fine, piuttosto che su come si pensa l'agente dovrebbe comportarsi.

Cosa è **razionale** in un dato momento dipende da:

- La *misura di performance* che definisce il criterio di successo.
- La *conoscenza precedente* dell'ambiente da parte dell'agente.
- Le *azioni* che l'agente può eseguire.
- La *sequenza percettiva* dell'agente fino al momento.

Definizione 1 (**Agente Razionale**): per ogni possibile sequenza percettiva un *agente razionale* dovrebbe selezionare un azione che ci si aspetta **massimizzi** le proprie misure di performance, date le prove fornite dalla sequenza percettiva e tutta la conoscenza che l'agente possiede.

L'**omniscenza** consiste nel conoscere il *reale* risultato di un azione e comportarsi di conseguenza.
Gli *agenti razionali* massimizzano la performance attesa mentre gli **agenti perfetti** massimizzano la performance reale. La **raccolta di informazioni** consiste nell'eseguire azioni in modo da modificare le percezioni future. L'**esplorazione** è un tipo di raccolta di informazioni eseguita da un agente che *non* conosce inizialmente l'ambiente. L'agente non solo deve essere in grado di raccogliere informazioni, ma anche di **imparare** il più possibile dalle percezioni.

È possibile raggruppare tutte le informazioni di cui abbiamo bisogno in modo da specificare un agente razionale sotto al concetto di *task environment*. In sostanza il **task environment** consiste nei problemi per i quali gli agenti razionali rappresentano la soluzione.

Definizione 2 (**P.E.A.S. - Performance, Environment, Attuators, Sensors**): Task Environment.
Le descrizioni *PEAS* definiscono i task environments. Alcune proprietà del *task environment* sono:

- **Osservabilità**: quanto è osservabile l'ambiente (totalmente, parzialmente, non osservabile).
- **Singolo o multi-agente**: nel caso di più agenti si ha a che fare con concetti come: competizione, cooperazione, comunicazione e comportamento casuale.
- **Deterministico o stocastico**: un ambiente è *deterministico* quando il suo stato è completamente determinato dallo stato corrente e le azioni eseguite dall'agente, altrimenti è *stocastico*.
- **Incerto**: se l'ambiente non è completamente osservabile o non deterministico.
- **Nondeterministico**: ambiente in cui le azioni sono caratterizzate dal suo possibile risultato ma non vi sono probabilità annesse.
- **Episodico**: l'esperienza dell'agente è suddivisa in episodi atomici e gli episodi successivi non dipendono dalle azioni intraprese nei precedenti episodi.
- **Sequenziale**: in questo caso le decisioni correnti possono influenzare quelle future (es. scacchi).

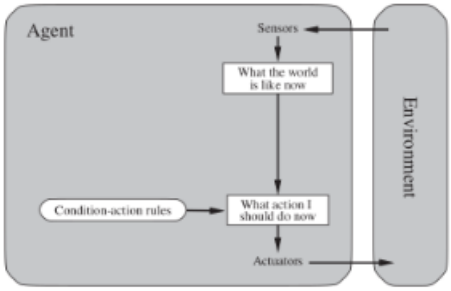
- **Statico o dinamico**, **discreto o continuo**, ...

Un esempio di un ambiente difficile è il taxi driver, in quanto è: parzialmente osservabile, multi-agente, stocastico, sequenziale, dinamico e continuo. Tipo di agente: Taxi Driver; misura di **P**erformance: sicuro, veloce, legale, confortevole, ecc.; **E**nvironment: strade, pedoni, traffico, ecc.; **A**ttuatori: volante, acceleratore, freni, ecc.; **S**ensori: camere, sonar, GPS, ecc.

2.2 Struttura degli agenti

Il lavoro dell'IA è progettare un *programma agente* che implementi la *funzione agente*, ovvero il mapping tra percezione ed azioni. Questo programma verrà eseguito in una parte fisica (sensori, attuatori, ecc) chiamata **architettura**. Agente = Architettura + Programma.
Ci sono 4 tipi base di *programmi agente*:

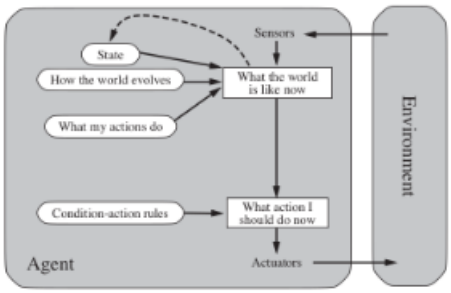
1. **Simple Reflex Agents**: l'agente agisce solo sulla base della situazione ambientale attuale, mappando la percezione corrente in azioni appropriate *ignorando* lo storico delle percezioni. Il processo di mapping potrebbe essere basato su una tabella o su qualsiasi algoritmo di corrispondenza basato su regole. Questo tipo di agenti richiede che l'ambiente sia *completamente osservabile*.
   Un esempio per questa classe è un aspirapolvere robot che si muove in un ciclo infinito, ogni percezione contiene uno stato di una posizione corrente [*pulito*] o [*sporco*] e di conseguenza decide se [*aspirare*] o [*continuare a muoversi*].



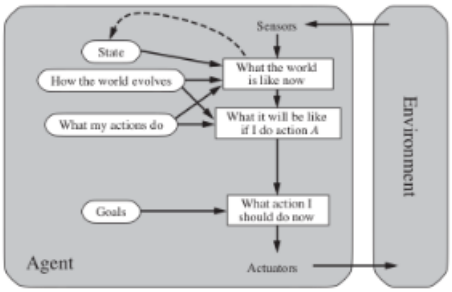
2. **Model-based Reflex Agent**: l'agente ha bisogno di memoria per memorizzare lo storico delle percezioni, usandolo per aiutare a rivelare gli attuali aspetti *non osservabili* dell'ambiente. È quindi indicato in casi di ambiente osservabile solo parzialmente. L'agente deve avere un modello del *mondo*, il quale contiene due tipi di conoscenza:

   - Come si evolve il mondo indipendentemente dall'agente.
   - Come le azioni dell'agente influenzano il mondo.

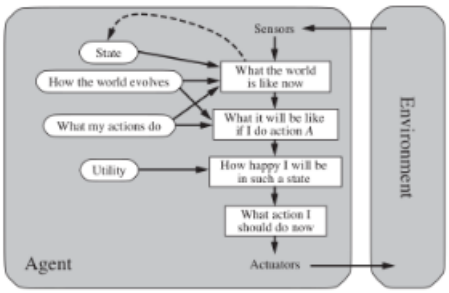
   Un esempio per questa classe è la visione mobile autosterzante in cui è necessario controllare lo storico della percezione per comprendere appieno come il mondo si sta evolvendo (macchine frenano, pedoni avanzano, ecc).



3. **Goal-based Agents**: in questo modello l'agente ha un *obiettivo* e ha una strategia per raggiungere quell'obiettivo, tutte le azioni sono basate sul suo goal e da una serie di azioni possibili seleziona quella che migliora il progresso verso l'obiettivo (non necessariamente il migliore).
   Un esempio per questa classe è qualsiasi robot di ricerca che abbia una posizione iniziale e desideri raggiungere una destinazione.

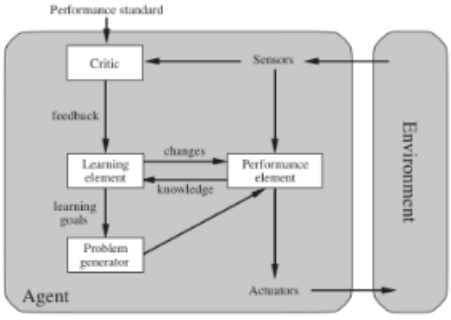


4. **Utility-based Agents**: come l'agente basato sui goal ma con una misura di "*quanto felice*" un'azione renderebbe l'agente, piuttosto che il feedback binario basato sui goal ["*felice*", "*infelice*"], questo tipo di agenti fornisce la soluzione migliore.
   Un esempio per questa classe è un sistema di raccomandazione del percorso che calcola il percorso "migliore" per raggiungere una destinazione.



5. **Learning Agents**: questo agente è in grado di apprendere dall'esperienza, ha la capacità di acquisire e integrare automaticamente le informazioni nel sistema, qualsiasi agente progettato e previsto per avere successo in un *ambiente incerto* è considerato un agente di apprendimento. Ogni agente di questo tipo deve avere i seguenti componenti concettuali:

   - *Learning element*: che è responsabile di apportare miglioramenti.
   - *Performance element*: che è responsabile di selezionare azioni esterne.
   - *Critic*: che valuta le performance dell'azione.
   - *Problem generator*: che suggerisce le azioni che possono portare a nuove esperienze.



3 Problem solving e ricerca

Come citato in precedenza gli *agenti intelligenti* tendono a massimizzare la propria misura di performance, questo risulta più semplice se l'agente adotta un **obiettivo** (goal) e prova a soddisfarlo.
- La prima cosa che l'agente dovrebbe fare è definire l'obiettivo, questo processo è chiamato **goal formulation**, che consiste nel primo passo del problem solving. L'obiettivo è costituito da un insieme di *stati del mondo* che lo soddisfano.
- La seconda cosa da fare definito un obiettivo è la **formulazione del problema**, ovvero il processo di decisione di quali azioni e stati considerare, dato un goal.
- Terzo, assumendo che l'ambiente sia deterministico la soluzione ad ogni problema è una sequenza fissa di azioni. Il processo di ricerca di una sequenza di azioni che raggiunge l'obiettivo è chiamata **search**.
In breve: formulare un **obiettivo**, formulare un **problema** da risolvere, risolvere il problema utilizzando la **ricerca**, **eseguire** la soluzione, ritornare al passo 1.

Un **algoritmo di ricerca** prende in input un problema e ritorna una soluzione sotto forma di sequenza di azioni. Un **problema** può essere definito formalmente tramite cinque componenti:

- **Stato iniziale**: dove l'agente inizia, `In(s)`.
- **Azioni**: una descrizione di tutte le possibili azioni disponibili per l'agente. Dato uno stato `s`, `ACTIONS(s)` ritorna l'insieme di azioni che possono essere eseguite.
- **Modello di transizione**: è una descrizione di cosa fa ogni azione, `RESULT(s, a)`. Questi tre componenti (stato iniziale, azioni e modello di transizione) definiscono lo **spazio degli stati** del problema ed ha la forma di un grafo. Un percorso nel grafo indica una sequenza di stati connessi da una sequenza di azioni.
- **Goal test**: determina se un certo stato è uno **stato goal** o meno.
- **Costo del percorso**: è una funzione che assegna un costo numerico ad ogni percorso, `C(s, a, s')` è chiamata "*step cost*" ed è il costo dell'azione `a` per portare l'agente dallo stato `s` allo stato `s'`.

Una **soluzione** è una *sequenza di azioni* che porta dallo *stato iniziale* allo *stato goal*. Una **soluzione ottima** è la soluzione con il *costo del percorso* più basso tra tutte le possibili soluzioni. L'**astrazione** è il processo di rimozione dei dettagli da una rappresentazione.

Una soluzione abbiamo detto che è una sequenza di azioni che parte dallo stato iniziale e forma un **search-tree**. La ricerca parte dallo stato **radice** ed a ogni stato, chiamato **nodo**, decide quale azione eseguire. Questa operazione viene chiamata **espansione**. Il modo in cui un algoritmo sceglie quale nodo espandere è deciso da *strategie di ricerca*. Viene chiamata **frontiera** l'insieme di tutti i nodi espandibili in ogni momento. Ci sono quattro elementi utilizzati per valutare le performance di un algoritmo:

1. *Completezza*: se l'algoritmo garantisce di trovare una soluzione, quando la trova?
2. *Ottimalità*: la strategia trova la soluzione ottima?
3. *Complessità temporale*: quanto tempo ci mette per trovare una soluzione?
4. *Complessità spaziale*: quanta memoria è necessaria per eseguire la ricerca?

Per valutare l'efficacia di un algoritmo di ricerca solitamente si considera solo il *costo di ricerca* che tipicamente dipende dalla complessità temporale. È possibile raggruppare le strategie di ricerca in strategie **informate** e **disinformate**.

3.1 Strategie di ricerca disinformata

Questo tipo di stategie non hanno informazioni aggiuntive sugli stati oltre a quelle fornite dalla definizione del problema. L'unica cosa che queste strategie possono fare è *generare stati* (ovvero espandere nodi) e distinguere uno stato *goal* da uno *non-goal*. Intuitivamente questi algoritmi, non prendendo in considerazione

la posizione del goal, non sanno dove stanno andando fino a quando non trovano un obiettivo e riportano un successo.

Breadth-First Search
Questo algoritmo espande tutti i nodi in un dato livello prima di espandere ogni nodo sul livello successivo. Viene utilizzata una **coda FIFO** per implementare l'algoritmo.

- *Completezza*: si, trova sempre il nodo goal data una profondità finita (d) e un fattore di branching finito (b).
- *Ottimalità*: non sempre, dipende dai costi dei percorsi, se il costo del percorso è uguale per tutti allora è ottimo.
- *Complessità temporale*: $b + b^2 + b^3 + ... + b^d = O(b^d)$, in cui b è il fattore di branching e d la profondità dell'albero. Questo valore tuttavia è esponenziale, ciò non è consigliabile.
- *Complessità spaziale*: anch'essa esponenziale, $O(b^d)$ che la rende eccessiva, si pensi a valori bassi come $d = 16$ e $b = 10$ necessitano di uno spazio di $10^{16} = 10$ exabyte e 365 anni di tempo.

In linea generale BFS funziona nel seguente modo: si prende il nodo di partenza e si espande, trovando i nodi del livello 1; per ognuno di questi, da sinistra verso destra, si controlla se corrispondono al goal, in caso affermativo si termina la ricerca, altrimenti vengono espansi a loro volta determinandone i figli. Non si procede a valutare i nodi espansi nel livello $n + 1$ fino a quando non sono stati valutati tutti quelli del livello n. Da notare che questo algoritmo non tiene conto di eventuali pesi degli archi, ma solo nel numero di "step" necessari da un nodo partenza ad un nodo goal.

Uniform-Cost Search
Questo algoritmo è un'evoluzione del BFS in cui si tiene conto anche del costo del percorso da un nodo di partenza ad uno di arrivo. Invece che espandere il nodo più in "superficie" come in BFS, viene espanso il nodo n con il costo di percorso $g(n)$ più basso, non si procede più per "livelli" ma si espande il nodo sulla frontiera con costo inferiore. Questo è fatto ordinando la frontiera con una **coda di priorità** ordinata per g (con g funzione di percorso meno costoso). In sostanza la ricerca a costo uniforme espande i nodi in base al loro costo di percorso ottimale, per questo l'algoritmo è caratterizzato dal costo del percorso invece che dalla profondità dei rami.
Ogni nodo espanso viene inserito in una lista di nodi *visited* che serve a evitare di visitare più volte gli stessi nodi. Ogni volta che si deve espandere un nodo si sceglie quello che ha il percorso complessivo di costo inferiore (memorizzando per ognuno il costo dall'origine ad esso). Si procede espandendo i nodi convenienti ed anche se si raggiunge un nodo goal si attende fino a quando tutti i percorsi verso tutti i nodi goal sono stati identificati, scegliendo infine quello con costo inferiore.

- *Completezza*: è possibile se il costo dei nodi è sempre positivo, quindi non avvengono loop.
- *Ottimalità*: è ottimale perchè espande i nodi basandosi sul loro costo di percorso ottimale.
- *Complessità temporale*: $O(b^{1+\frac{c}{\epsilon}})$ in cui c è il costo della soluzione ottimale, ed ϵ una piccola costante positiva.
- *Complessità spaziale*: uguale alla complessità temporale.

Depth-First Search
DFS espande sempre il nodo più profondo nella frontiera corrente dell'albero di ricerca. Questo algoritmo è implementato utilizzando una coda LIFO.

- *Completezza*: no nel caso di spazio degli stati infito, si in caso di stati finiti.
- *Ottimalità*: non è ottimale.
- *Complessità temporale*: $O(b^m)$ in cui m è la massima profondità di ogni nodo.

- *Complessità spaziale*: $O(b \cdot m)$ in cui b è il coefficiente di branching ed m la profondità massima di ogni nodo.

Esiste una variante di DFS chiamata **backtracking search** che ha complessità spaziale $O(m)$.

Depth-Limited Search
Il problmea di DFS con lo spazio degli stati infinito può essere risolto mettendo un limite l alla profondità dell'albero. In questo caso i nodi al livello l sono considerati senza figli. Questo approccio viene chiamato appunto **depth-limited search**.

- *Completezza*: se $l > d$ si, altrimenti no.
- *Ottimalità*: dipende dalla relazione tra l e d.
- *Complessità temporale*: $O(b^l)$.
- *Complessità spaziale*: $O(b \cdot l)$.

Depth-First Search a Profondità Iterativa
Questo algoritmo incrementa il limite di profondità finchè non trova lo stato goal.

- *Complessità temporale*: $O(b^d)$.
- *Complessità spaziale*: $O(b \cdot d)$.

La *profondità iterativa* è il metodo di ricerca disinformata preferito quando lo spazio di ricerca è grande e la profondità della soluzione non è conosciuta. Esiste una variante di questo algoritmo che invece di accrescere la profondità incrementa il limite di costo del percorso. Questo algoritmo è chiamato **Iterative Lengthening Search**.

Ricerca Bidirezionale
È un algoritmo di ricerca che consiste in due ricerche parallele, una che parte dal nodo iniziale e la seconda che parte dal nodo goal. L'algoritmo si ferma quando le due ricerche si incontrano.

3.2 Strategie di ricerca informata (euristica)

Questo tipo di strategie utilizzano la conoscenza specifica del problema oltre alla definizione del problema per trovare una soluzione in un modo efficiente. L'approccio generale è chiamato **Best-First Search**, in cui si esplora un grafo espandendo il nodo più "promettente" secondo una *funzione di valutazione* $f(n)$. È chiaro quindi che la scelta di f determini la strategia di ricerca. Molti degli algoritmi best-first includono come componente di f una *funzione euristica* $h(n)$.
$h(n)$ stima il costo del percorso meno costoso dal nodo n allo stato goal. La performance degli algoritmi di ricerca euristica dipende dalla qualità della *funzione euristica*. È possibile a volte costruire buone euristiche rilassando la definizione del problema (applicando un processo chiamato rilassamento), memorizzando costi di soluzioni pre-calcolate per sotto-problemi in un database di modelli, o imparando dall'esperienza.

Greedy Best-First Search
Si espande il primo figlio del nodo, dopo la generazione di un nodo figlio:
- Se l'euristica del figlio è migliore del suo genitore, il figlio viene inserito in testa alla coda, con il genitore reinserito direttamente dietro di esso e il ciclo ricomincia.
- Altrimenti, il figlio viene inserito nella coda in una posizione determinata dal suo valore euristico. La procedura valuterà poi gli eventuali figli rimanenti del genitore.

L'algoritmo valuta i nodi utilizzando **solo** la seguente funzione euristica:
$\mathbf{f(n) = h(n)}$, in cui $h(n)$ è un'approssimazione dallo stato n allo stato goal.
Ad esempio se si vuole andare da una città a un'altra passando per diverse altre città, si potrebbe scegliere come funzione euristica $g(n)$ la distanza in linea retta tra ogni città e il goal, ed a ogni step si sceglie il nodo con distanza minore dall'obiettivo (sperando sia ottimale).

- *Completezza*: l'algoritmo è incompleto anche con spazio degli stati finito in quanto c'è la possibilità che rimanga bloccato senza uscita.
- *Ottimalità*: il costo di ricerca è minimo ma non ottimale.
- *Complessità temporale*: $O(b^m)$ in cui m è la massima profondità e b il fattore di branching.
- *Complessità spaziale*: $O(b^m)$.

In questo modello la funzione euristica è molto importante, una funzione poco precisa o erronea porterebbe a risultati insoddisfacenti.

A* Search
Valuta i nodi combinando $g(n)$ = costo per arrivare al nodo n ed $h(n)$ = costo per arrivare dal nodo n al nodo goal.
$\mathbf{f(n) = g(n) + h(n)}$ stima il costo della soluzione meno costosa passante per n.
Per avere soluzioni ottimali è necessario utilizzare euristiche che non sovrastimino mai il costo, al limite sottostimino o utilizzino il valore esatto. In generale l'algoritmo A* è una versione modificata (ed informata) della ricerca a costo uniforme. Ad ogni stato è associato un numero che rappresenta una stima del costo da quello stato ad uno dei goal. Se durante la ricerca si trova nuovamente uno stato visitato con valore A* più basso si eslora lo stato, altrimenti lo si ignora e si pota quella parte della ricerca. Al termine dell'esecuzione viene ritornato lo stato goal con costo inferiore.
Si mantengono due liste: *open* e *closed*, nella prima vi sono tutti i nodi che si stanno prendendo in considerazione al momento (frontiera), nella seconda i nodi già visitati. Si parte dal nodo di partenza, si calcola $f(n)$ per ogni successore, inserendoli in open, e si prende in considerazione il nodo con valore $f(n)$ minore tra tutti quelli nella lista di nodi aperti. A questo punto si sposta il nodo padre in closed e si ripete il procedimento con il nodo selezionato. La stima $h(n)$ è molto utile per evitare di percorrere strade a costo basso ma che di fatto allontanerebbero dall'obiettivo.

Condizioni di ottimalità per A*:

- **Ammissibilità**: un'euristica è definita *ammissibile* se non sovrastima mai il costo per raggiungere il goal. Queste euristiche sono chiamate *ottimistiche* in quanto ritengono il costo di risoluzione del problema inferiore a quello che è realmente. Ad esempio una distanza in linea retta tra due punti è un'euristica ammissibile in quanto rappresenta la distanza minima tra i due punti.
- **Consistenza** (monotonicità): è una condizione forte richiesta solo per ricerca sui grafi con A*. Un'euristica $h(n)$ è **consistente** se per ogni nodo n ed ogni successore n' di n generato da ogni azione a, il costo stimato per raggiungere il goal da n non è maggiore del costo dello step per arrivare ad n' più il costo stimato per raggiungere il goal da n'. La condizione è anche chiamata disuguaglianza del triangolo e afferma che ogni lato del triangolo non può essere più lungo della somma degli altri due lati.
  $\mathbf{h(n) \leq c(n, a, n') + h(n')}$, in cui c calcola il costo per andare da n a n'.

La versione dell'albero di ricerca per A* è ottimale se $h(n)$ è *ammissibile*, mentre nella versione della ricerca su grafo è ottimale se $h(n)$ è *consistente*.

- *Completezza*: A* è completo.
- *Ottimalità*: A* è ottimale e ottimamente efficiente.
- *Complessità temporale*: $O(b^d)$.
- *Complessità spaziale*: $O(b^d)$.

Ricerca Euristica con Memoria Delimitata
Alcuni degli algoritmi di questo gruppo includono **iterative-deepening A* (IDA)**, **recursive best-first search (RBFS)** e **memory-bounded A* (MA*)**. La caratteristica comune di questi algoritmi è che

utilizzano davvero poca memoria (lineare in base alla profondità della soluzione ottimale più profonda) e dato abbastanza tempo possono risolvere problemi che algoritmi come A* non possono risolvere per esaurimento di memoria.
In linea generale nell'iterative-deepening A* si fa partire l'algoritmo a profondità $d = 0$ che corrisponde allo stato di inizio, e si controlla se questo è anche il goal. In caso affermativo si termina la ricerca e si ritorna lo stato, altrimenti lo si espande incrementando il livello di profondità d di uno e così via fino a quando non si trova lo stato goal.

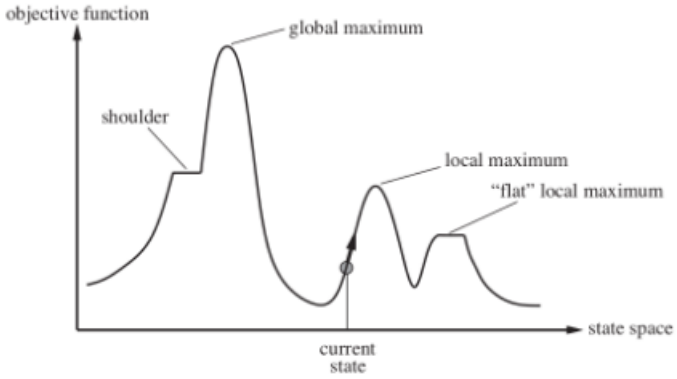
4 Oltre la ricerca classica

4.1 Algoritmi di ricerca locale e problemi di ottimizzazione

Gli algoritmi di ricerca visti nella sezione precedente vengono chiamati *algoritmi di ricerca sistematica* in quanto avanzano esplorando tutti i percorsi finchè non trovano il goal, mantenendoli in memoria. Quando la ricerca ha successo (un goal è trovato) l'intero percorso è la soluzione.
In molti problemi non è necessario conoscere il percorso per arrivare alla soluzione, ma solo la soluzione. In questi casi gli **algoritmi locali di ricerca** sono la scelta migliore in quanto considerano solo il *nodo corrente* e il suo *vicinato* ogni volta. Questi algoritmi presentano tre vantaggi principali:

1. Basso utilizzo di memoria, solitamente una quantità costante.
2. Si possono trovare buone soluzioni in domini molto grandi, anche infiniti.
3. Sono molto adatti per risolvere problemi di *ottimizzazione* in cui il goal non è solo la ricerca di una soluzione ma la ricerca della migliore soluzione basata su una *funzione obiettivo*.

Esempi di *funzioni obiettivo* possono essere: minimizzare il costo di un operazione ad esempio, in questo caso vogliamo trovare il **minimo globale** nell'orizzonte. O viceversa se si vuole massimizzare una funzione si cerca il **massimo globale**.



Hill-Climbing Search
La ricerca *hill-climbing* è uno dei più famosi algoritmi di ricerca locale. È semplicemente un loop in cui ad ogni passo prende il nodo con il valore più alto fino a quando non viene trovato il "picco". Ad ogni passo l'algoritmo memorizza solo il nodo corrente e la funzione obiettivo, questo significa che il consumo di memoria rimane molto basso.

function Hill-Climbing (problem) **returns** uno stato che rappresenti il massimo locale
  current ← Make-Node(problem.Initial-State)
  loop do
    neighbor ← il successore di *current* con valore più alto
    if *neighbor*.Value < *current*.Value **then**
      return *current*.State
    current ← *neighbor*

A volte questo algoritmo viene anche chiamato **Greedy Local Search** in quanto ad ogni passo prende il miglior vicino ma senza sapere cosa avviene dopo. Questa modalità può chiaramente portare ad alcuni problemi:

- *Massimo locale*: l'algoritmo si ferma su un massimo locale invece che su uno globale.
- *Plateaux*: è il caso in cui il vicinato è "piatto" e non ci sono soluzioni migliori per un certo tratto.

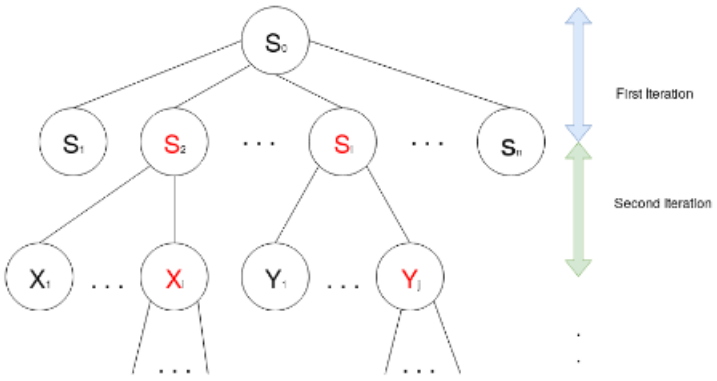
Vi sono tre implementazioni di hill-climbing: **Stochastic Hill-Climbing**: si sceglie casualmente tra tutte le possibili salite. **First-Choice Hill-Climbing**: si itera il processo di selezione casuale di un vicino come candidato per una soluzione e lo si accetta solo se il risultato rappresenta un miglioramento. **Random Restart Hill-Climbing**: se si raggiunge un ottimo locale si riparte da una posizione casuale x. Si cerca quindi l'ottimo locale riguardo ad x, se è migliore del precedente si aggiorna la variabile *current* e si ricomincia. I criteri di arresto sono o dopo un certo tempo di esecuzione, o dopo un certo numero di iterazioni.

Temperamento Simulato
In questo algoritmo di ricerca locale per trovare il massimo globale sono ammesse mosse "sbagliate" verso qualche soluzione non buona (discesa) con la speranza di evitare massimi locali e trovare soluzioni migliori in seguito. Si procede come segue per un numero finito di iterazioni:

- Si campiona un nuovo punto x_I nel vicinato di x, ovvero $N(x)$.
- Si salta nel nuovo punto con probabilità data dalla funzione di probabilità scelta $P(x, x_I, T)$ dove T è positivo.
- Si decrementa T

Se $T \to 0$ l'algoritmo assomiglia all'*hill-climbing*, se $T \to \infty$ assomiglia ad una *passeggiata aleatoria*.

Local Beam Search
In contrasto con *hill-climbing*, che mantiene solo il nodo corrente e la funzione obiettivo in memoria, questo algoritmo mantiene k possibili nodi, dove k è un numero generato casualmente. Ad ogni passo vengono generati tutti i successori di tutti i k nodi e se uno di essi è il goal la computazione si arresta, altrimenti procede in parallelo con tutti i thread che si scambiano informazioni utili sulla migliore soluzione fino al suo raggiungimento. Le soluzioni sub-ottimali sono rimosse e vengono seguite ed espanse solo le soluzioni più promettenti.



Questo modo di procedere restringe enormemente lo spazio di ricerca e l'algoritmo converge velocemente.

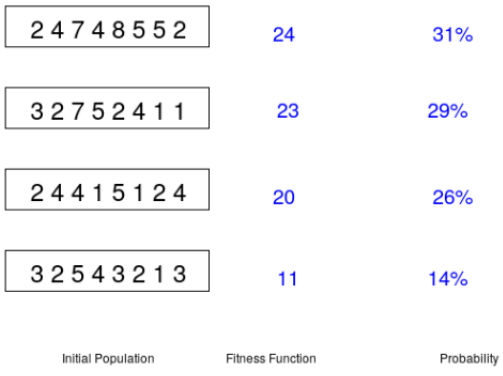
Algoritmi Genetici (GA)
Sono algoritmi che cercano di imitare l'idea biologica dell'evoluzione naturale. In generale i passaggi seguiti dagli algoritmi sono i seguenti

- Si in inizia con k stati (nodi) scelti in modo casuale chiamati **popolazione**.

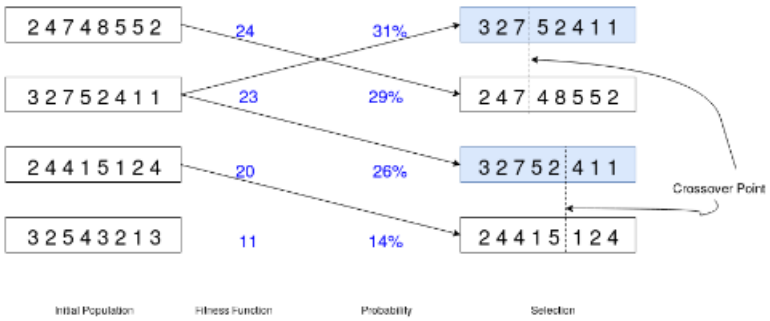
- Ogni stato, chiamato anche **individuo** è rappresentato da una stringa finita in un qualche alfabeto (cromosomi).
- Una funzione obiettivo chiamata *fitness function* è definita per rappresentare gli individui più adatti. Solitamente è meglio indivuarne il numero maggiore possibile.
- Il **crossover** è il processo di riproduzione in cui gli individui più adatti vengono replicati.
- La **mutazione** è il processo di evoluzione, si permette che alcune stringhe vengano modificate per poter evolvere.

Un esempio è il caso delle *8 regine*. Questo è un problema molto famoso e consiste nel posizionare 8 regine in una scacchiera 8x8 in modo tale che non si possano attaccare a vicenda.

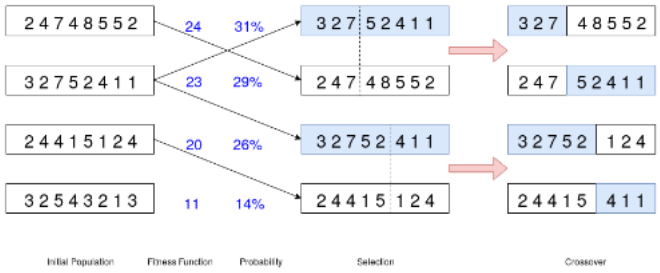
1. Si generano k stati random, in questo caso ne prendiamo in considerazione 2 (popolazione).
2. Si devono rappresentare gli individui come una stringa di numeri in un alfabeto. In questo caso la soluzione migliore è numerare righe e colonne da 1 a 8. La prima scacchiera (il primo individuo) sarà rappresentata dalla stringa $s_1 = 3, 2, 7, 5, 2, 4, 1, 1$ e la seconda (secondo individuo) dalla stringa $s_2 = 2, 4, 7, 4, 8, 5, 5, 2$.
3. Si definisce la *fitness function*, una buona funzione è una che determini il numero più alto di individui adatti. Definiamo la *fitness function* come *il numero di coppie che non si attaccano a vicenda* (ad esempio una funzione definita come il numero di triple o quadruple che non si attaccano a vicenda darebbe meno individui). Nel primo individuo il risultato della funzione è 23, nel secondo 24 coppie. Supponiamo di espandere questo esempio con altri due stati (scacchiere) con *fitness* rispettivamente 20 e 11 (la configurazione di questi due stati è nell'immagine sotto).
   Un altro passo intermedio da fare è calcolare le probabilità che questi individui vengano scelti nei passaggi successivi. Un modo banale per farlo è dividere ogni valore per la somma di tutti i punteggi *fitness*. Quindi: $S = 24 + 23 + 20 + 11 = 77$ la probabilità del primo individuo sarà $24/77 = 31\%$, e così via.



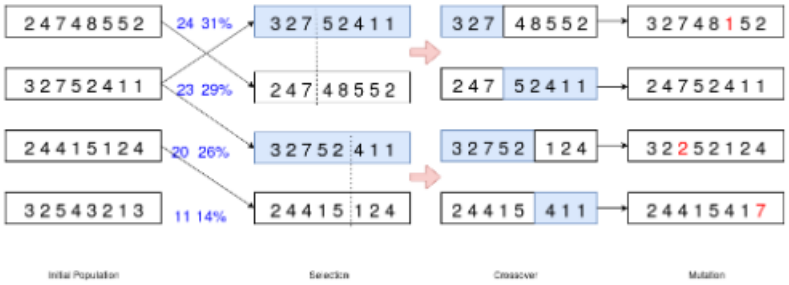
4. A questo punto si prendono gli individui con la maggior probabilità di riprodursi e per ogni coppia di individui si decide randomicamente un *crossover point*.



5. In questa fase si combinano insieme le coppie di individui scelti, più precisamente la parte sinistra del *crossover point* del primo individuo, con il lato destro del *crossover point* del secondo individuo.



6. Nell'ultima fase si sceglie casualmente una mutazione per gli individui finali (mutazioni in rosso).



Infine l'algoritmo ricominica da capo ripetendo tutto fino a quando la condizione di terminazione non viene raggiunta. Nell'esempio sopra la condizione di terminazione è 28 $(7+6+5+4+3+2+1)$, ovvero non vi sono coppie che si attaccano a vicenda.

Le caratteristiche degli algoritmi genetici sono che: lavorano bene con problemi discreti e continui; la probabilità di rimanere bloccati in minimi locali è molto bassa; sono costosi in termini di computazione ma si può risolvere tramite parallelismo; trovare una buona *fitness function* è molto importante e il processo potrebbe essere costoso.

4.2 Ricerca locale in spazi continui

Gli algoritmi descritti in precedenza non possono gestire problemi nel continuo in quanto hanno branching finito (gli alberi di ricerca hanno un numero finito di rami).

Gradiente
Il *gradiente* altro non è che la pendenza che c'è tra *due punti* (su un piano cartesiano 2D), precisamente la variazione sull'asse delle y diviso la variazione sull'asse delle x, per funzioni più complesse di una retta si utilizzano le derivate al posto della semplice "variazione" data dalla sottrazione. È una generalizzazione

multi-variabile del concetto di **derivata** ed è una funzione con valori vettoriali, quindi ha sia direzione che magnitudine. Il gradiente punta verso la direzione del maggiore incremento della funzione e la magnitudine è la pendenza del grafico in quella direzione.
Ad esempio, se si vogliono costruire tre nuovi aereoporti in modo tale che la somma delle distanze quadrate che intercorrono tra ogni ogni città e gli aereoporti sia minima. Ogni aereoporto A_i ha coordinate $A_1 = (x_1, y_1)$, $A_2 = (x_2, y_2)$ e $A_3 = (x_3, y_3)$. Si può calcolare la funzione obiettivo $f = (x_1, y_1, x_2, y_2, x_3, y_3)$ facilmente per ogni stato ma questo significa che si otterrebbe come risultato un *minimo locale*. Per trovare il *minimo* (o il massimo) *globale* è necessario prendere il considerazione il **gradiente**. Il gradiente della funzione obiettivo è il vettore ∇f e fornisce direzione e magnitudine della *pendenza più ripida*, calcolando le derivate parziali del primo ordine della funzione rispetto ad ogni variabile.

Definizione: data una funzione $f(x, y, z)$ in tre variabili, il **gradiente** è dato dalla formula:

$$\nabla f = \frac{\delta f}{\delta x} i + \frac{\delta f}{\delta y} j + \frac{\delta f}{\delta z} k$$

In cui i, j, k sono i vettori unità standard in tre dimensioni.

Nel nostro caso il gradiente è $\nabla f = (\frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta y_1}, \frac{\delta f}{\delta x_2}, \frac{\delta f}{\delta y_2}, \frac{\delta f}{\delta x_3}, \frac{\delta f}{\delta y_3})$. Trovare il massimo locale è facile, basta risolvere l'equazione $\nabla f = 0$, per trovare il massimo globale invece è necessario fare una *steepest-ascent hill-climbing* e possiamo farlo iterando la seguente formula:

Definizione: trovare il massimo globale (**metodo di discesa**)

$$\mathbf{x_{k+1}} \leftarrow \mathbf{x_k} + \alpha_k \nabla f(\mathbf{x_k}), \text{ dove } \alpha \text{ è la dimensione del } \textbf{passo}$$

La costante α è molto importante in quanto misura quanto grandi o piccoli siano i passi verso il massimo globale, tuttavia se si decide di procedere con passi grandi si rischia di superare il picco (goal).

Ci sono molti modi per avvicinarsi al goal ad ogni iterazione, uno dei metodi più conosciuti è il metodo **Newton-Raphson**. Per trovare le radici di una funzione è necessario risolvere l'equazione *g(x) = 0*. Si procede calcolando una nuova stima della funzione ad ogni iterazione utilizzando la formula:

$$\mathbf{x_{k+1}} \leftarrow \mathbf{x_k} - \frac{g'(\mathbf{x_k})}{g''(\mathbf{x_k})}, \text{ dove } g'(x) \text{ e } g''(x) \text{ sono la derivata prima e seconda di } g(x)$$

Nel nostro caso abbiamo a che fare con il calcolo multivariabile, quindi al posto della derivata è necessario usare il *gradiente*. La formula da risolvere è la seguente:

Definizioni: discesa gradiente

$$\mathbf{x_{k+1}} \leftarrow \mathbf{x_k} - H_f^{-1}(\mathbf{x_k}) \nabla f(\mathbf{x_k})$$

In cui $H_f^{-1}(\mathbf{x})$ è la matrice Hessiana delle derivate seconde, i cui elementi H_{ij} sono dati da $\frac{\delta^2 f}{\delta_{x_i} \delta_{x_j}}$.

Risolvendo il problema precedente ad ogni iterazione ci si avvicina al minimo globale. Tuttavia, nello spazio continuo con molte variabili, la soluzione diventa difficile dal punto di vista computazionale a causa delle grandi dimensioni della **matrice Hessiana**.
Ottimizzazione dei vincoli e *programmazione lineare* sono alcune delle tecniche più utilizzate nell'ottimizzazione combinatoria. Qui le soluzioni sono in una certa forma data dai vincoli che devono soddisfare.
Un altro esempio di come calcolare i gradienti: data la funzione $g(x, y, z) = \sqrt{x^2 + y^2 + z^2}$ si calcola $\nabla g(x, y, z)$.
$\nabla g(x, y, z) = \langle \frac{\delta g(x,y,z)}{\delta x}, \frac{\delta g(x,y,z)}{\delta y}, \frac{\delta g(x,y,z)}{\delta z} \rangle = \frac{x}{\sqrt{x^2+y^2+z^2}}, \frac{y}{\sqrt{x^2+y^2+z^2}}, \frac{z}{\sqrt{x^2+y^2+z^2}}$
Consideriamo il gradiente in un punto particolare $P = (2, 6, -3)$. $\nabla g(2, 6, -3) = ?$. Prima si calcola il denominatore che è comune a tutti: $\sqrt{2^2 + 6^2 + (-3)^2} = \sqrt{49} = 7$. Quindi i punti saranno: $\langle \frac{2}{7}, \frac{6}{7}, \frac{-3}{7} \rangle$

4.3 Ottimizzazione numerica

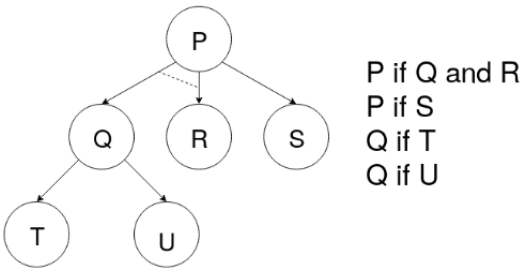
Le **tecniche di ottimizzazione** ci permettono di trovare gli estremi (minimo e massimo) di una *funzione obiettivo* $f : \mathbb{R}^n \to \mathbb{R}$.
Si sa che $max(f(x)) = -min(-f(x))$ quindi è necessario trovare solo uno degli estremi della funzione, diciamo $min(f(x))$ ed è poi possibile calcolare facilmente $max(f(x))$.

Il **Golden Section Search** è un semplice algoritmo che ha lo scopo di trovare il massimo in una funzione con valori definiti all'inteno di un segmento $[a, b]$. Di seguito il funzionamento:

1. Si inizia considerando l'intervallo $[a_0, b_0] := [a, b]$ aggiustando il valore di *tolleranza* ϵ.
2. Si restringe progressivamente l'intervallo $[a_{k+1}, b_{k+a}] \subset [a_k, b_k]$ in modo che il valore minimo x^* continui ad essere dentro al nuovo intervallo.
3. Si ferma l'iterazione quando $|b_k - a_k| < \epsilon$ e si accetta il valore $\frac{b_k - a_k}{2}$ come buona approssimazione del minimo x^*.

4.4 Ricerca con azioni non-deterministiche

Nel caso in cui l'ambiente sia parzialmente osservabile o non-deterministico o entrambi è necessario escogitare un **piano di contingenza** anche chiamato **strategia**. Il metodo più usato in questo caso è l'**And-Or Search Trees**. Gli alberi And-Or sono definiti come una rappresentazione grafica della riduzione del problema in *congiunzioni* o *disgiunzioni* di sotto-problemi.



5 Ricerca avversaria

5.1 Giochi

I *giochi* sono l'esempio tipico di ambienti multi-agente in cui ogni agente deve considerare l'azione degli altri nell'ambiente e come questa azione influenzi il suo benessere. In questa situazione, la **competizione** è la situazione in cui i goal degli agenti sono in conflitto. Questo fenomeno è chiamato **problemi di ricerca avversaria** o **giochi**. Di seguito alcuni concetti da sapere sui giochi:

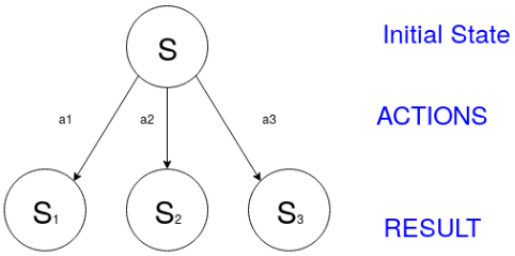
- Giochi competitivi o a somma-zero: un giocatore vince, l'altro perde.
- Informazione perfetta: i giocatori conoscono il risultato di tutte le mosse precedenti
- Informazione imperfetta: i giocatori non conoscono le mosse precedenti, ad esempio il caso in cui giocano simultaneamente.
- Stati semplici: facile descrivere lo stato del gioco in ogni momento (giochi complicati sono il calcio o il basket).

È necessario definire un po' di terminologia:

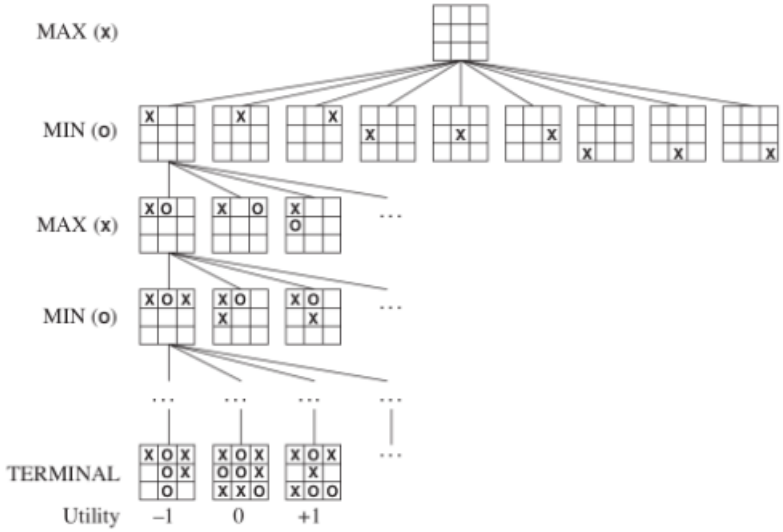
- `S0`: stato iniziale del gioco.
- `PLAYER(s)`: chi è il giocatore nello stato s.
- `ACTIONS(s)`: ritorna l'insieme di mosse legali dallo stato `s`.

- `RESULT(s, a)`: lo stato risultante dopo che è stata effettuata l'azione `a` sullo stato `s`.
- `TERMINAL-TEST(s)`: ritorna **true** se `s` è uno stato terminale.
- `UTILITY(s, p)`: la funzione obiettivo nello stato `s` per il giocatore `p`.

Si immagini di avere due giocatori: **MIN**, **MAX** e due operazioni: `ACTIONS(s)`, `RESULT(s, a)`. Con questi dati possiamo creare un **game tree**, di seguito un esempio.



Nel caso degli scacchi il numero di nodi è 1040 e come si può immaginare non può essere esplorato o memorizzato tutto a causa della complessità computazionale estremamente elevata. In tic-tac-toe questo numero è inferiore a 9! ma comunque molto alto per un gioco così semplice. L'immagine seguente mostra un albero parziale per il gioco. **MAX** è il giocatore che inizia per primo il gioco. **Utility** mostra il risultato del gioco nello stato terminale.



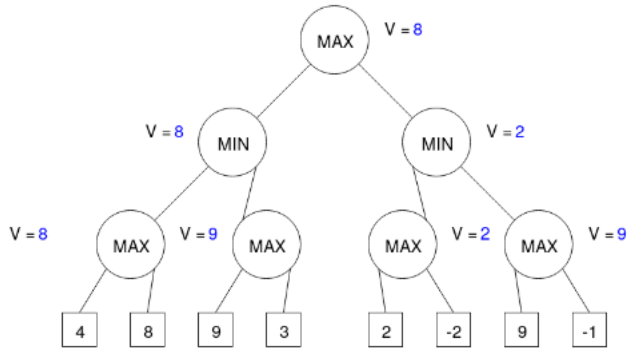
5.2 Algoritmo MINIMAX

La soluzione ottimale in un gioco è una sequenza di azioni che porta a uno stato goal, in altre parole che porta a uno stato vincente. Per vincere un giocatore, deve mettere in atto una strategia. Dato un albero di gioco, la strategia ottimale può essere determinata dal **valore minimax** di ciascun nodo. Questo algoritmo è adatto a giochi deterministici con informazione perfetta. La seguente immagine mostra formalmente l'algoritmo.

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

L'algoritmo **MINIMAX** esegue una **Depth-First Search** dell'albero di gioco. Se la profondità massima dell'albero è m e ci sono b mosse legali (fattore di branching) in ogni punto, allora abbiamo una *complessità temporale* $O(b^m)$ e una *complessità spaziale* $O(bm)$. È *completo* se l'albero è finito, ed *ottimale* se anche

l'avversario si comporta in modo ottimale. Questa complessità molto elevata rende MINIMAX un algoritmo impraticabile anche per giochi semplici.



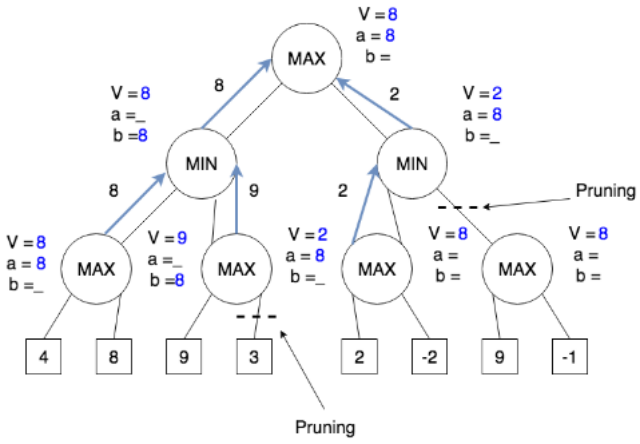
Se si considera MINIMAX per più di due giocatori, avremmo un vettore di possibili valori nella funzione UTILITY e l'algoritmo potrebbe cambiare leggermente. Questa nuova versione è chiamata $\alpha\beta - pruning$.

5.3 Alpha-Beta Pruning

Il problema con la ricerca **MINIMAX** è che il numero di stati del gioco che deve esaminare è esponenziale nella profondità dell'albero. Esistono tecniche che possono alleviare questo problema tagliando la metà del numero di stati che vanno visitati. Queste tecniche sono basate sul concetto di **potatura** (pruning). In $\alpha\beta - pruning$ l'idea è quella di voler potare alcuni nodi per motivi di efficienza. L'algoritmo si basa su 2 valori α e β.

α: è la scelta migliore di MAX fino ad ora (il valore più alto).
β: è la scelta migliore di MIN fino ad ora (il valore più basso).
$[\alpha, \beta]$: ogni nodo tiene traccia dei propri valori α e β.

MIN aggiorna sempre β e MAX aggiorna sempre α. L'algoritmo parte assegnando $\alpha = -\infty$ e $\beta = +\infty$.



La complessità di questo algoritmo può essere ridotta ad $O(b^{\frac{m}{2}})$. Ci sono anche **giochi non-deterministici**, in cui dobbiamo includere un livello di nodi **chance** tra i nodi MAX e MIN. In questa classe sono inclusi giochi come backgammon o i giochi di carte.
Nel caso di **giochi con informazione imperfetta** si calcola il valore minimax per ogni azione in ogni scenario, poi si sceglie l'azione con il valore atteso più alto tra tutti gli scenari.

6 Problemi di soddisfacimento dei vincoli

I problemi di soddisfacimento dei vincoli (CSP) rappresentano una classe molto importante di problemi di ottimizzazione combinatoria. Questi problemi sono molto difficili da risolvere perché hanno una complessità esponenziale. Per risolverli, si utilizzano tecniche diverse, una delle quali è il **constraint programming** (CP). Ovvero un paradigma di programmazione dichiarativa che consente di modellare e risolvere problemi di ottimizzazione combinata (COP) fornendo un costrutto di alto livello come vincoli globali, operatori logici ecc. La tecnica che CP utilizza per risolvere i COP è una sistematica ricerca in backtracking dell'albero.

6.1 Definire CSP

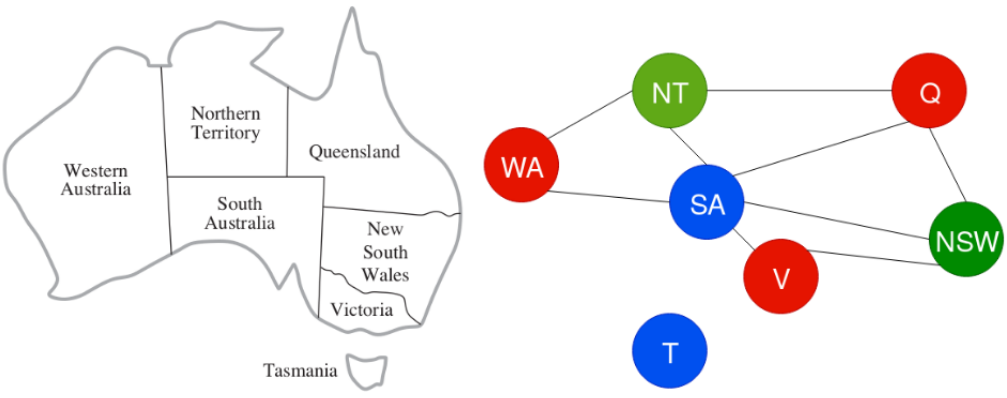
Un problema di soddisfazione dei vincoli è modellato come una tripla di insiemi: (X, D, C) dove:

- X: è un'insieme di variabili chiamate *variabili di decisione*, $X = X_1, X_2, ..., X_n$
- D: è un'insieme di domini, uno per ogni variabile X_i, $D = D_1, D_2, ..., D_n$
- C: è un'insieme di vincoli che specificano i possibili valori che le variabili possono (o non possono) assumere. Un vincolo è una relazione tra variabili $C_i(X_j ... X_k)$.

Una soluzione per un CSP è un'assegnazione di un valore alle variabili che soddisfino contemporaneamente tutti i vincoli. Un esempio di CSP è la colorazione di una mappa: data una mappa di stati, colorare gli stati con colori diversi in modo tale che quelli non adiacenti abbiano lo stesso colore. Consideriamo questo particolare problema con la mappa australiana. Può essere espresso con il seguente CSP:

- $P = (X, D, C)$
- $X = \{WA, NT, Q, NSW, V, SA, T\}$
- $D = \{red, green, blue\}$
- $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$

Le **variabili** possono avere domini discreti, finiti o infiniti. Anche i valori nel **dominio** possono essere di qualsiasi tipo (numeri interi, reali, booleani, ...). Si hanno molti tipi di vincoli, unari, binari, globali, ... Uno dei vincoli globali più importanti è $Alldifferent(X_1, X_2, ...X_n)$ che forza tutte le variabili ad essere differenti l'una dall'altra.

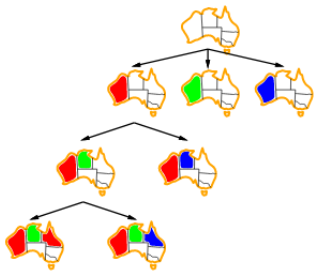


6.2 Consistenza locale e propagazione dei vincoli

Come abbiamo detto in precedenza, CP utilizza la ricerca sistematica per estendere un'assegnazione parziale di valori alle variabili in una sola, completa e coerente. Naturalmente non tutti gli assegnamenti sono coerenti, quindi è necessario trovare un modo per rilevare questo tipo di incarichi incoerenti. La **consistenza locale** è una forma di **inferenza** che consente di rilevare assegnazioni parziali incoerenti. In questo modo è possibile fare meno lavoro nella ricerca. Si chiama consistenza locale perché vengono esaminati i vincoli uno

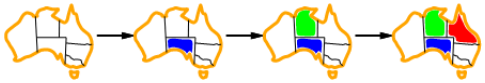
alla volta in quanto la consistenza globale è NP-hard. Ricapitolando la *consistenza locale* è una proprietà che vogliamo garantire e lo facciamo usando **Propagation**. La propagazione sono tutte le tecniche utilizzate per applicare la consistenza locale. Il modo in cui queste tecniche e questi metodi sono implementati sono liberi così che chiunque possa implementarli, importante è l'obiettivo, far rispettare la coerenza locale.
Esistono molti tipi di consistenza, ma le più utilizzate sono le consistenze basate sul dominio, come la **arc consistency**, la **generalized arc consistency** e la **bounds consistency**. Questo tipo di consistenze sono in grado di rilevare incoerenze del tipo: $X_i = j$ e se questo assegnamento è inconsistente si elimina il valore j dal dominio della variabile X_i tramite *propagazione*.

Backtracking Search
La Depth-First Search nei CSPs con assegnamenti a variabile singola è chiamata *backtracking search*, ovvero l'algoritmo disinformato base per i problemi di soddisfacimento dei vincoli. Di seguito un esempio:

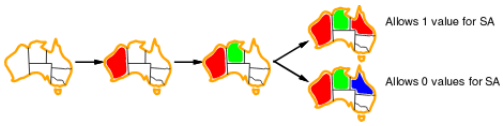


Minimo Valore Rimanente
Viene scelta la variabile con il numero inferiore di valori assegnabili (chiamati valori "legali"). Di seguito un esempio:

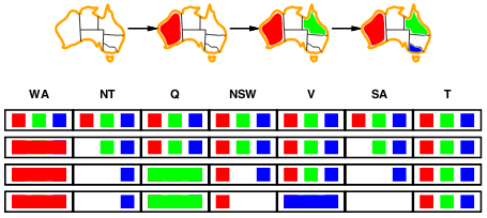
Grado di Euristica
Viene scelta la variabile con il maggior numero di costanti sulle variabili rimanenti. Di seguito un esempio:



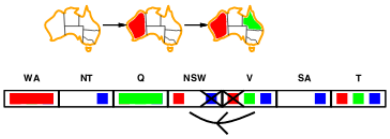
Valore Meno Vincolante
Data una variabile, viene scelto il valore meno vincolante, ovvero quello che esclude il numero minore di valori nelle variabili rimanenti. Di seguito un esempio:



Controllo in Avanti
L'idea è quella di tenere traccia dei valori legali rimanenti per le variabili non assegnate. La ricerca termina quando una variabile non ha più valori legali. Il controllo in avanti propaga informazioni dalle variabili assegnate a quelle non assegnate, ma non fornisce una rilevazione precoce per tutti i fallimenti (es. nella penultima riga è chiaro che NT e SA non possano essere entrambe blu). Di seguito un esempio:



Arc Consistency
Rappresenta la più semplice forma di propagazione e rende ogni arco consistente. $X \to Y$ è consistente sse per ogni valore x di X esiste un qualche valore y permesso. Di seguito un esempio:



Nell'esempio X perde un valore, e i vicini di X vanno ricontrollati.

CSP Strutturati ad Albero
Teorema: se il grafo dei vincoli non ha loop, il CSP può essere risolto in tempo $O(n\ d^2)$.
Gli **algoritmi iterativi per CSP** prevedono che si permetta la presenza di stati con vincoli non soddisfatti e che gli operatori riassegnino i valori delle variabili. La scelta della variabile può essere: *base*, ovvero si sceglie randomicamente una variabile con conflitto, con *euristica min-conflicts*, ovvero si sceglie la variabile che viola meno vincoli (usando una funzione $h(n)$ = numero totale di vincoli violati).

6.3 Constraint Programming

Due *vincoli* sono **equivalenti** se hanno lo stesso insieme di soluzioni.
Un *vincolo* è **soddisfacibile** se ha almeno una soluzione.

Metodo Gauss-Jordan
Si sceglie un equazione (vincolo) c dall'insieme C e si riscrive nella forma $x = e$, poi si sostituisce x con e ovunque in C con e. Questo procedimento va avanti fino a quando tutte le equazioni sono nella forma $x = e$ (ritorna true) o un'equazione è nella forma $d = 0$ (ritorna false).

Vincoli ad Albero
I *vincoli ad albero* rappresentano dati strutturati, una *costante* è un albero, un *costruttore* con una lista di alberi maggiore di 0 è un albero (con n figli). Esempio: *cons(red, purple, cons(blue, cons(green)))*.

Termini Un *termine* è un albero con variabili che sostituiscono sottoalberi. Esempio: *cons(red, X, cons(blue, cons(Y)))*, X e Y possono essere altri sottoalberi.

Solver
Un risolutore è ben costruito se: le risposte dipendono solo dall'insieme di vincoli primitivi, è monotonico ovvero se fallisce per $C1$ fallisce anche per $C1 \wedge C2$, *indipendente* dal nome delle variabili, il risolutore fornisce la stessa risposta indipendentemente da ome vengono chiamate le variabili.
Un risolutore è **completo** se risponde *sempre* true o false, mai unknown.

6.4 MiniZinc

MiniZinc è un linguaggio di modellazione di alto livello utilizzato per indicare i problemi di soddisfacimento dei vincoli. Si usa MiniZinc per scrivere codice di alto livello che modella un CSP, successivamente usa uno dei 20 risolutori per risolvere il CSP. Il processo di compilazione passa attraverso una rappresentazione di basso livello chiamata *FlatZinc* e quindi al risolutore: **MiniZinc** $\to$ **FlatZinc** $\to$ **Solver X**.

Esempio: supponiamo di avere il seguente problema: si vuole massimare $25B+30T$ tali che $\frac{1}{200}B+\frac{1}{40}T \leq 40$; con $0 \leq B \leq 60$ e $0 \leq T \leq 40$. Questo problema è modellabile tramite MiniZinc nel seguente modo:

```
var 0..60: B;
var 0..40: T;
constraint (1/200)*B + (1/140)*T <= 40;
solve maximize 25*B + 30*T;
```

output [”B = ”, show(B), ”T = ”, show(T)];

Alcune peculiarità di MiniZinc sono: la dichiarazione di variabili, l'assegnamento, la struttura *constraint* <*booleanexpression*>, la keyword *solve* (satisfy, minimize, maximize), predicati e test, annotazioni e output. Di seguito il problema dello zaino in MiniZinc:

```
int: n;          % number of objects
int: weight_max; % maximum weight allowed (capacity of the
knapsack)
array[1..n] of int: values;
array[1..n] of int: weights;
array[1..n] of var int: take; % 1 if we take item i; 0
otherwise

var int: profit = sum(i in 1..n) (take[i] * values[i]);
solve maximize profit;

constraint        % all elements in take must be >= 0
forall(i in 1..n) ( take[i] >= 0 )   /\
sum(i in index_set(weights))( weights[i] * take[i] ) <=
weight_max;

output [show(take), "\n"];
```

7 Agenti logici

In questo capitolo studieremo gli **agenti basati sulla conoscenza**, ovvero agenti che hanno una rappresentazione interna della conoscenza e utilizzano un processo chiamato **ragionamento** per accedere e manipolare questa conoscenza. Per poter ragionare dobbiamo creare un formalismo per esprimere la conoscenza. Verrà di seguito introdotta la logica formale (sarà sufficiente la logica proposizionale e del primo ordine).

7.1 Agenti basati sulla conoscenza

Chiamiamo **Knowledge-Based**, (KB) un insieme di **frasi** o **proposizioni** espresse in un linguaggio speciale chiamato *linguaggio di rappresentazione della conoscenza*. Le frasi che non derivano da altre frasi sono chiamate **assiomi**. Possiamo aggiungere frasi in un KB con un processo chiamato **TELL** e interrogare il KB con **ASK**. Usiamo l'**inferenza** per derivare nuove frasi da quelle esistenti. Ogni linguaggio formale ha una **sintassi** ben definita che viene utilizzata per rappresentare tutte le frasi ben formate e una **semantica** che definisce il significato delle frasi. La semantica esprime il valore di verità della frase in un modello (mondo), ovvero un'interpretazione di un insieme di proposizioni, cioè un'associazione tra le proposizioni elementari e le realtà rappresentate.
Se una frase α è vera in un modello M, diciamo che α *soddisfa* M. Una frase β che segue da una frase α è scritta nella forma $\alpha \models \beta$ e rappresenta la nozione di **derivazione logica**. Questa formulazione significa che in ogni modello in cui α è vero anche β è vero.

Definizione: $\alpha \models \beta \iff M(\alpha) \subseteq M(\beta)$

Un **algoritmo di inferenza** è un algoritmo che può trarre conclusioni sulla base di alcune premesse. Vogliamo che questi algoritmi siano sia **sound** che **completi**. La **correttezza** (soundness) significa che ogni formula che può essere dimostrata in un sistema è logicamente valida rispetto alla *semantica* del sistema. La **completezza** esprime il fatto che un insieme di assiomi è sufficiente a dimostrare tutte le verità di una teoria e quindi a decidere della verità o falsità di qualunque enunciato formulabile nel linguaggio della teoria. Queste due proprietà sono molto desiderabili nella logica.
Se KB è vero nel mondo reale, allora qualsiasi frase α derivata da una procedura di inferenza sound è garantita essere vera nel mondo reale. Importante è anche il concetto di **grounding** che effettua la domanda: come facciamo a sapere che il nostro KB è vero nel mondo reale?

7.2 Logica proposizionale

La **logica proposizionale** è il tipo più semplice di logica che si usa per esprimere la conoscenza e formalizzare il linguaggio naturale. Quando definiamo una logica dobbiamo prima definire la **sintassi**, poi la **semantica**. Nel caso della logica proposizionale abbiamo i seguenti elementi:

- **Aiuti simbolici**: usati per aiutarci ad esprimere le frasi in maniera chiara, questi aiuti sono: "(", ")".
- **Connettivi**: permettono di creare frasi complesse combinando insieme frasi atomiche, alcuni esempi sono: "$\neg$", "$\wedge$", "$\vee$", " $\implies$ ", " $\iff$ ".
- **Costanti proposizionali**: top ("$\top$") e bottom ("$\bot$").
- **Variabili proposizionali**: un numero infinito di proposizioni atomiche, $p_0, p_1, ...$

Inoltre è presente un insieme di regole che permette di creare frasi complesse:

Regole: $\omega ::= \top | \bot | p_i | \neg\omega | (\omega \wedge \omega) | (\omega \vee \omega) | (\omega \implies \omega) | (\omega \iff \omega)$

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{True} \mid \textit{False} \mid P \mid Q \mid R \mid \ldots \\
\textit{ComplexSentence} & \rightarrow & (\textit{Sentence}) \mid [\textit{Sentence}] \\
& \mid & \neg \textit{Sentence} \\
& \mid & \textit{Sentence} \wedge \textit{Sentence} \\
& \mid & \textit{Sentence} \vee \textit{Sentence} \\
& \mid & \textit{Sentence} \Rightarrow \textit{Sentence} \\
& \mid & \textit{Sentence} \Leftrightarrow \textit{Sentence}
\end{array}
$$

$$
\textsc{Operator Precedence} \quad : \quad \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
$$

A questo punto, dopo la **sintassi**, è necessario definire la **semantica** della logica proposizionale. La semantica definisce le regole che assegnano il valore di verità a ciascuna frase rispetto a un modello particolare. Usiamo la *tabella di verità* per definire la semantica.

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*

L'**enunciazione proposizionale** è co-NP completo che significa che nel peggiore dei casi la complessità computazionale dell'algoritmo è esponenziale nella dimensione dell'input. Fino ad ora abbiamo verificato la **derivazione** mediante il **controllo del modello**, quindi elencando tutti i modelli in cui la derivazione è vera. C'è un altro metodo chiamato **dimostrazione del teorema**, ovvero fornire derivazione applicando alcune *regole di inferenza* ben definite. Applicheremo direttamente l'inferenza sulle frasi del KB in modo da arrivare a una dimostrazione senza controllare i modelli.

Equivalenza logica ($\equiv$): diciamo che 2 frasi α, β sono logicamente equivalenti se sono vere nello stesso insieme di modelli. Per $\alpha \equiv \beta$ ad esempio $A \wedge B \equiv B \wedge A$.

Equivalenza logica

$$\alpha \equiv \beta \iff \alpha \vDash \beta \wedge \beta \vDash \alpha$$

Validità: una frase è valida se è vera in *tutti* i possibili modelli. Le frasi valide sono chiamate **tautologie** e sono necessariamente vere. Un esempio di tautologia è il seguente: $A \wedge \neg A$. Dato che $\top$ è vero in tutti i modelli ogni frase valida equivale a $\top$.

Teorema di deduzione

Per ogni frase α e β, $\alpha \vDash \beta$ se e solo se la frase $(\alpha \implies \beta)$ è valida.

In conclusione possiamo decidere $\alpha \vDash \beta$ controllando che $\alpha \implies \beta$ sia vera.

Soddisfacibilità: una frase è **soddisfacibile** se è vera o soddisfatta da *alcuni* modelli. Il problema della *soddisfacibilità* (SAT) può essere verificato elencando tutti i modelli fino a quando non se ne trova uno che soddisfi la frase. SAT nella logica proposizionale è NP-completo. La **validità** e la **soddisfacibilità** sono strettamente legati: α è *valido* se e solo se $\neg\alpha$ è *insoddisfacibile* ed α è *soddisfacibile* se e solo se $\neg\alpha$ non è *valido*.

Reductio ad Absurdum

$\alpha \vDash \beta$ se e solo se la frase $(\alpha \wedge \neg\beta)$ è insoddisfacibile.

Dimostrare β partendo da α verificando l'insoddisfacibilità di $\alpha \wedge \neg\beta$ è anche chiamato **confutazione** o **prova per contraddizione**. In generale, utilizziamo le **regole di inferenza** per derivare/inferire una **prova**. Qui ne vedremo alcune:

Modulus Ponens

$\frac{(\alpha \implies \beta)\alpha}{\beta}$

Se $\alpha \implies \beta$ ed α è data, si può inferire β.

Eliminazione-AND

$\frac{(\alpha \wedge \beta)}{\beta}$ or $\frac{(\alpha \wedge \beta)}{\alpha}$

Se è data una congiunzione di due termini si può inferire ognuno di essi separatamente.

Monotonicità: indica che l'insieme di frasi derivate può solo aumentare si aggiungono più informazioni al KB.
Un altro modo per dimostrare le frasi è applicare la regola di **risoluzione**. La quale prende in input una clausola (**disgiunzione dei letterali**) e un nuovo letterale e produce una nuova clausola. Di seguito un esempio di come funziona questa regola:
$\frac{(A \vee B)(\neg A \vee \neg C)}{B \vee \neg C}$
Un dimostratore di teoremi basato su *risoluzione* può, per ogni frase α e β in *Logica Proposizionale*, decidere se $\alpha \vDash \beta$. La risoluzione funziona solo per clausole disgiunzione dei letterali $(\alpha \vee \beta)$. Ogni frase in Logica Proposizionale è logicamente equivalente ad una congiunzione di clausole. Una frase espressa come congiunzione di clausole è chiamata *Conjuctive Normal Form (CNF)*.

Esempio: applicazione di regole logiche per trasformare $B \iff (A \vee C)$ in CNF:

1. $B \iff (A \vee C)$
2. $(B \implies (A \vee C)) \wedge ((A \vee C) \implies B)$
3. $(\neg B \vee A \vee C) \wedge (\neg(A \vee C) \vee B)$
4. $(\neg B \vee A \vee C) \wedge ((\neg A \vee \neg C) \vee B)$
5. $(\neg B \vee A \vee C) \wedge (\neg A \vee B) \wedge (\neg C \vee B)$

L'algoritmo di risoluzione per verificare che $KB \vDash \alpha$ mostri che $(KB \wedge \neg\alpha)$ è *insoddisfacibile*. Come funziona l'algoritmo:

- *Primo*: si trasforma $(KB \wedge \neg\alpha)$ in un CNF e poi lo si usa come input per l'algoritmo.
- *Secondo*: ogni coppia che contiene letterali complementari viene risolta per produrre una nuova clausola la quale è aggiunta all'insieme se non è già presente.
- *Terzo*: questo processo continua fino a quando una delle seguenti cose non avviene:
  - Non ci sono più nuove clausole che possono essere aggiunte, ciò significa che il KB non deriva α.
  - Le ultime due clausole vengono risolte in una *clausola vuota*, ciò significa che KB deriva α.

La **Clausola Definita** è una disgiunzione dei letterali in cui *esattamente uno è positivo*: $\neg A \vee B \vee \neg C$.
La **Clausola di Horn** viene utilizzata per una forma semplificata di risoluzione. Le clausole horn sono una disgiunzione di letterali dove al *massimo uno è positivo*.
La **Clausola Goal** è chiamata clausola *senza valori letterali positivi*.
Vogliamo lavorare con KB che hanno solo *clausole definite* per 3 motivi:

- Ogni clausola definita può essere scritta come un'implicazione la cui *premessa* è una congiunzione di letterali positivi e la cui *conclusione* è un singolo letterale positivo: $(\neg A \vee \neg B \vee C) :: (A \vee B) \implies C$. Nella forma di Horn la *premessa* si chiama corpo e la conclusione si chiama *testa*. Un singolo letterale positivo è chiamato *fatto*.
- Gli algoritmi di concatenamento avanti/indietro vengono utilizzati per effettuare *inferenza* di clausole di Horn. Questi algoritmi sono la base della programmazione logica.
- La complessità della dimostrazione della *derivazione* in Horn è *lineare* sulla dimensione di KB.

L'**algoritmo di concatenamento avanti/indietro** prende come input un KB e una proposizione che chiamata *query* e definisce se la query è derivata dal KB. Questo algoritmo è **sound** e **completo** e alla fine raggiunge il cosiddetto **punto fisso** in cui non è più possibile effettuare calcoli.

8 Logica del primo ordine

Possiamo usare i migliori elementi della *Logica Proposizionale* come il fatto che sia dichiarativa, compositiva, indipendente dal contesto e non ambigua e alcuni aspetti dei *linguaggi naturali*. Possiamo associare nomi a **oggetti**, verbi a **relazioni** e **funzioni**. Una *funzione* è una relazione da un input a un output.

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} & \rightarrow & (\ \textit{Sentence}\ ) \mid [\ \textit{Sentence}\ ] \\
& \mid & \neg\, \textit{Sentence} \\
& \mid & \textit{Sentence} \wedge \textit{Sentence} \\
& \mid & \textit{Sentence} \vee \textit{Sentence} \\
& \mid & \textit{Sentence} \Rightarrow \textit{Sentence} \\
& \mid & \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
& \mid & \textit{Quantifier}\ \textit{Variable}, \ldots\ \textit{Sentence} \\
\\
\textit{Term} & \rightarrow & \textit{Function}(\textit{Term}, \ldots) \\
& \mid & \textit{Constant} \\
& \mid & \textit{Variable} \\
\\
\textit{Quantifier} & \rightarrow & \forall \mid \exists \\
\textit{Constant} & \rightarrow & A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\textit{Predicate} & \rightarrow & \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} & \rightarrow & \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\\
\textsc{Operator Precedence} & : & \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

I modelli nella FOL sono interessanti in quanto contengono oggetti. Chiamiamo **dominio** di un modello l'insieme di oggetti che esso contiene. Gli elementi sintattici di base nella FOL sono i simboli utilizzati per rappresentare oggetti (**simboli costanti**), relazioni (**simboli predicati**) e funzioni (**simboli funzione**). Ogni predicato o funzione ha un numero definito di parametri chiamato **arity**. Oltre a *oggetti*, *predicati* e *funzioni* ogni modello include anche un'**interpretazione**.
Un **termine** è un'espressione lessicale che si riferisce a un oggetto. Possiamo combinare insieme *termini* che si riferiscono a oggetti e *predicati* che si riferiscono a relazioni per formare **frasi atomiche** chiamate anche **atomi** che vengono utilizzate per dichiarare *fatti*. Un *atomo* è formato da un *predicato* seguito da un

elenco di termini, ad esempio *Brother(John, Jim)...* Si possono creare frasi atomiche complesse, ad esempio *Married(Father(John), Mother(John))*.
Le **frasi complesse** sono frasi create combinando insieme frasi atomiche usando i *connettivi*. $\neg Brother(LeftLeg(Richard), Jo$ oppure $Brother(Richard, John) \land Brother(John, Richard)$. I **quantificatori** consentono di esprimere le proprietà di intere raccolte di oggetti. La FOL ha 2 quantificatori:

- Quantificatore *universale* ($\forall$): permette di creare frasi su tutti gli elementi di un dominio. Ad esempio: $\forall x King(x) \implies Person(x)$, questa frase significa: per tutte le variabili x, se x è un Re allora è anche una Persona.
- Quantificatore *esistenziale* ($\exists$): permette di creare frasi su *alcuni* elementi del dominio. Ad esempio: $\exists x Person(x) \land King(x)$, questa frase significa: esiste un x tale che sia una Persona e un Re.

Il connettivo naturale per $\forall$ è $\implies$ e per $\exists$ è $\land$. Si può scrivere ($\forall$, $\implies$) e ($\exists, \land$).

Possiamo avere più di un quantificatore all'interno di una frase, in questo caso si parla di **quantificatori annidati**. I quali possono essere dello stesso tipo: $\forall x \forall y Brothers(x, y) \implies Siblings(x, y)$ o di tipi diversi: $\forall x \exists y Loves(x, y)$.
Una variabile appartiene sempre al quantificatore più interno che la menziona: $\forall x(Crown(x) \lor (\exists x Brother(Richard, x)))$.
Vi è un'altra maniera di creare *frasi atomiche* in FOL, ovvero il **simbolo di uguaglianza**. Questo simbolo è utilizzato per indicare che due oggetti si riferiscono allo stesso oggetto: $Father(John) = Henry$. Di seguito le regole di De Morgan per la logica del primo ordine:

$$\begin{array}{llll} \forall x\ \neg P & \equiv \neg\exists x\ P & \neg(P \lor Q) & \equiv \neg P \land \neg Q \\ \neg\forall x\ P & \equiv \exists x\ \neg P & \neg(P \land Q) & \equiv \neg P \lor \neg Q \\ \forall x\ P & \equiv \neg\exists x\ \neg P & P \land Q & \equiv \neg(\neg P \lor \neg Q) \\ \exists x\ P & \equiv \neg\forall x\ \neg P & P \lor Q & \equiv \neg(\neg P \land \neg Q) \end{array}$$

A questo punto è necessario utilizzare la logica per i nostri scopi, si immagini di avere un KB e di usare TELL per aggiungere frasi nel KB, mentre ASK per ottenere informazioni dal KB.
Asserzione: le frasi aggiunte nel KB tramite l'utilizzo di TELL (es. $TELL(KB, King(John))$).
Query: utilizzate per fare domande al KB (es. $ASK(KB, King(John))$ :: "*true*"). Una variante è ASKVARS che ritorna un insieme di valori, ad esempio "quali valori di x rendono la domanda vera".
Ogni KB può essere visto come un insieme di **assiomi** dai quali si possono derivare teoremi.

FIRST ORDER LOGIC
Termine - un termine è definito ricorsivamente come segue: o è una *costante* o una *variabile*, oppure se f^n è un simbolo di funzione (con *arità* n, ovvero con n argomenti) e $t_1, ..., t_n$ sono termini, allora $f^n(t_1, ..., t_n)$ è un termine.
Formula ben formata (fbf) - una fbf è definita ricorsivamente come segue: se p^n è un simbolo di predicato e $t_1, ..., t_n$ sono termini, allora $p^n(t_1, ..., t_n)$ è una fbf (atomica). Se F e G sono fbf, allora lo sono anche $\neg F$, $F \lor G$, $F \land G$, $F \to G$, $F \leftrightarrow G$. Se F è una fbf ed x una variabile, allora anche $\forall x F$ e $\exists x F$ sono fbf.

Esempio: $\neg\ \exists x(p(x, a) \land f(x))$ dato p predicato, ed f simbolo di funzione, questa non è una fbf in quanto non è possibile mettere in and una formula con un termine (solo entrambe formule), il termine può solo comparire come argomento di un predicato. Questo perchè il termine rappresenta un oggetto del dominio, non un concetto vero/falso come fa la formula.

Regole d'inferenza
Modus Ponens: è l'applicazione $MP\ :\ L^2 \to L$, ovvero da due formule ne produco una, precisamente da **F** e **F** $\to$ **G** deduci **G**. In cui F e $F \to G$ sono le premesse e G la conclusione.
Generalizzazione: è l'applicazione $Gen\ :\ L \to L$, ovvero da **F** deduci $\forall$**xF** in cui F è la premessa e $\forall x F$ la conclusione.

Formule chiuse
In una formula del tipo $\forall x F$ (oppure $\exists x F$) la variabile **x** e la formula **F** sono dette rispettivamente **indice** e **campo d'azione** del quantificatore $\forall$ (oppure $\exists$). Un'occorrenza di una variabile **x** in una formula è detta

vincolata sse x è l'*indice* di un quantificatore oppure appartiene al *campo d'azione* di un quantificatore avente x come indice. Diversamente, l'occorrenza di x è detta **libera**. Una formula **chiusa** è una formula senza occorrenze libere di alcuna variabile. Esempio: $\forall x(p(x) \wedge q(x))$ è chiusa, mentre $\exists x p(x,y) \vee q(x))$ ha occorrenza libera per y e anche per x in quanto $q(x)$ non è "visto" dal quantificatore esistenziale non essendovi parentesi e avendo l'or priorità inferiore all'exists.

Interpretazione
Pre-interpretazione: consiste in un insieme non vuoto **D** detto *dominio della pre-interpretazione*, ovvero un insieme di oggetti. Poi per ogni *costante* del linguaggio L la pre-interpretazione assegna un elemento del dominio. Infine ad ogni *simbolo di funzione n-aria* in L viene assegnata una corrispondenza da D^n a D. In generale quindi la pre-interpretazione da un signifcato alle costanti e ai simboli di funzione.
Interpretazione: si basa su una pre-interpretazione ed associa una corrispondenza da D^n a $\{V, F\}$ (vero o falso) ad ogni *predicato n-ario* del linguaggio. In generale l'interpretazione da un significato ai predicati.
Assegnamento sulle variabili: data una pre-interpretazione J l'assegnamento altro non fa che assegnare ad ogni variabile un elemento del dominio (del tutto identico a quello che fa la pre-interpretazione con le costanti).
Assegnamento sui termini: si associa un oggetto del domino ad ogni termine del linguaggio, precisamente: se il termine è una variabile o una costante allora significa che ciò è già stato fatto in precedenza. Se il termine è una funzione con termini $t_1, ..., t_n$ ai quali sono stati assegnati gli elementi del dominio $d_1, ..., d_n$ ed f' è la corrispondenza da D^n a D assegnata al simbolo di funzione f, allora $f'(d_1, ..., d_n) \in D$ è l'elemento di D assegnato ad $f(t_1, ..., t_n)$.

La **verità di una formula** è data da: per le forume atomiche del tipo $p(t_1, ..., t_n)$ si esamina il valore di $p'(d_1, ..., d_n)$ se p combacia con p' assegnato dall'interpretazione e i d_i con i t_i allora l'esito è **Vero**, altrimenti **Falso**.
In caso di combinazione di formule si utilizzano le tabelle di verità note.
Per i quantificatori: $\exists xF$ ha valore di verità **Vero** se *esiste* $d \in D$ tale che F ha valore di verità vero. Mentre $\forall xF$ ha valore di verità **Vero** se, **per ogni** $d \in D$, F ha valore di verità vero.

Esempio: sia il dominio $D : N^+$ l'insieme dei numeri naturali positivi, e sia $p :<$ ovvero la relazione minore. Questo è un modello per $\forall x \exists y p(x,y)$ ma non lo è per $\exists y \forall x p(x,y)$, le due formule avranno quindi insiemi di modelli diversi e non risultano semanticamente uguali.

Gli **assiomi propri** di una teoria T sono un sottoinsieme di formule chiuse. Un **modello** per una teoria T è un'interpretazione che è modello per ogni suo assioma. Un insieme di formule chiuse S è **soddisfacibile** se esiste un'interpretazione modello per S, **valido** se ogni interpretazione è modello per S, **insoddisfacibile** se nessuna interpretazione è modello per S e **non valido** se esiste interpretazione che non è modello per S.

Una formula chiusa F è **conseguenza logica** di un set di formule chiuse S se ogni interpretazione di I che è un modello per S lo è anche per F; si scrive $S \vDash F$ (se tutte le interpretazioni del linguaggio che rendono vere tutte le formule in S, rendono vera anche F allora si dice che F è conseguenza logica dell'insieme

equivalenze notevoli

- $\neg(\neg\alpha) \equiv \alpha$
- $\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$
- $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$ Leggi di De Morgan
- $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
- $\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$ Leggi di Distribuzione
- $\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
- $\alpha \wedge \beta \equiv \beta \wedge \alpha$ Leggi Commutative
- $\alpha \vee \beta \equiv \beta \vee \alpha$
- $(\alpha \wedge \beta) \wedge \gamma \equiv \alpha \wedge (\beta \wedge \gamma)$ Leggi Associative
- $(\alpha \vee \beta) \vee \gamma \equiv \alpha \vee (\beta \vee \gamma)$
- $\alpha \rightarrow \beta \equiv \neg\beta \rightarrow \neg\alpha$ Legge Contrappositiva
- $\neg\exists x p(x) \equiv \forall x \neg p(x)$
- $\neg\forall x p(x) \equiv \exists x \neg p(x)$
- $\forall x (p(x) \wedge q(x)) \equiv \forall x p(x) \wedge \forall x q(x)$
- $\exists x (p(x) \vee q(x)) \equiv \exists x p(x) \vee \exists x q(x)$

S).

8.1 Numeri, insiemi e liste

La **teoria dei numeri** può essere definita a partire da una serie molto piccola di assiomi. Infatti è necessario un **predicato** che chiamiamo *NatNum*, un **simbolo costante** che chiamiamo *0* e un **simbolo di funzione** che chiamiamo *successore*, *S*. Usiamo gli **assiomi di Peano** per definire i numeri naturali in modo ricorsivo: $NatNum(0); \forall n NatNum(n) \implies NatNum(S(n))$.
Il significato delle asserzioni precedenti è: "0 è un numero naturale", per ogni oggetto n, se n è un numero naturale, anche il successore di $n(S(n))$ è un numero naturale.
Quindi possiamo vedere i numeri naturali come: "0, $S(0)$, $S(S(0))$ ...). Possiamo aggiungere più assiomi per poter implementare operazioni come $(+)$. Gli **insiemi** sono un oggetto matematico molto importante. Possiamo esprimerli usando i costrutti di FOL.

- *Costante*: insieme vuoto.
- *Predicati*: un predicato unario *Set*, due predicati binari: $x \in s$ (x è incluso nell'insieme s) e $s1 \subseteq s2$ ($s1$ è un sottoinsieme di $s2$).
- *Funzioni*: tre funzioni binarie: $s1 \cap s2$ (intersezione), $s1 \cup s2$ (unione) e $x|s$ (un nuovo set che risulta dalla x contigua al set s).

Esistono 8 assiomi usati per implementare gli insiemi, ad esempio uno di questi è: $\forall s Set(s) \iff (s =) \vee (\exists x, s2 Set(s2) \vee s = x|s2)$, per gli altri assiomi fare riferimento al libro.
Le liste hanno alcune altre caratteristiche che devono essere prese in considerazione, ad esempio una lista è ordinata e uno stesso elemento può apparire più volte in una lista. Per definire le liste è necessario:

- *Costante*: *Nil* è un elenco costante senza elementi.
- *Funzioni*: Cons, Append, First, Rest.
- *Predicato*: Find

Usando i letterali precedenti possiamo definire le liste: $[A, B, C] :: Cons(A, Cons(B, Cons(C, Nil)))$.

8.2 Prolog

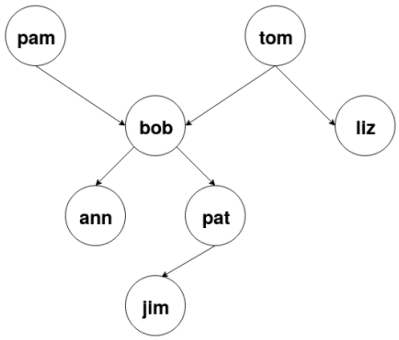
Prolog è un linguaggio di programmazione *logico* e *dichiarativo*. In un linguaggio dichiarativo il programmatore specifica un **goal** e lascia al sistema il modo di risolverlo. I programmi Prolog specificano le **relazioni**

tra oggetti e le proprietà tra gli oggetti. Ad esempio possiamo dire: John ha una bici - si sta dichiarando la relazione di proprietà tra 2 oggetti. Un programma Prolog contiene 3 elementi: **fatti**, **regole**, **interrogazioni**.
I *fatti* descrivono una relazione esplicita tra oggetti e proprietà che gli oggetti potrebbero avere ("John ha una bicicletta", "I capelli sono neri", "Apple è un'azienda" ecc.): **teaches(fabio, uux)**.
Le *regole* definiscono le relazioni implicite tra oggetti (es. relazione fratello) e proprietà dell'oggetto (A è figlio di B se B è padre di A). Le regole sono clausole *non unitarie*, mentre i fatti sono clausole unitarie: **guide(Teacher, Student): - teacher(Teacher, CourseId), studies(Student, CourseId)**.
Le *query* consentono di porre domande sulle proprietà dell'oggetto e dell'oggetto. Le query si basano su fatti e regole, supponiamo che si voglia sapere se andrea insegna il corso uux, si può scrivere: **?-teaches(andrea, uxx)**.

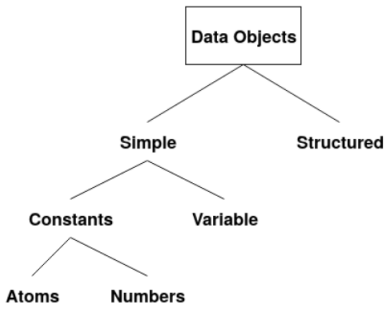
Sintassi delle clausole

- **:-** indica "se" o "è implicato da", il lato sinistro viene chiamato **head**, quello destro **body**.
- **,** indica "and" come congiunzione logica.
- **;** indica "or" come disgiunzione logica.

Relazioni familiari in Prolog: Prolog è un linguaggio di programmazione per il calcolo simbolico, non numerico. Per definire il KB per la relazione familiare, è necessario definire le relazioni aggiungendo **tuple** nel KB. Una volta aggiunte possiamo interrogare il KB sulle relazioni. Ogni clausola nel KB deve terminare con un *punto* ".". In esempi come **parent(bob, tom).**, "parent", "bob" e "tom" sono chiamati **atomi** mentre in **parent(bob, X).** X è una variabile.



parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
mother(X,Y):-parent(X,Y), female(X).
sister(X,Y):-parent(Z,X), parent(Z,Y), female(X), X≠Y.
haschild(X,_).
grandparent(X,Z):-parent(X,Y), parent(Y,Z).
predecessor(X,Y):-parent(X,Y). %base case
predecessor(X,Z):-parent(X,Y), predecessor(Y,Z). %recurssive case



Le liste sono una semplice struttura dati che consente di rappresentare raccolte di elementi, come $L = [blu, rosso, verde, nero]$. Una lista può essere vuota, [] o non vuota. Nel secondo caso il primo elemento della lista si chiama $Head$ e la parte restante si chiama $Tail$ ed è anch'essa stessa una lista.
Membership, controlla se un elemento è presente nell'elenco, è possibile usare anche la funzione build-in *member*.

```
Membership - lmember(X,L).
lmember(X,[X | _]).
lmember(X,[_ | TAIL]):-lmember(X,TAIL).
```

Lunghezza di una lista

```
Length of a list - length(L,N).
length([],0).
length([_ | TAIL], N):-length(TAIL, N1), N is N1+1.
```

Concatenzazione di liste

```
Concatenation of 2 lists - conc(L1,L2,L3).
conc([],L,L).
conc([X1 | L1],L2,[X1 | L3]):-conc(L1,L2,L3).
```

Cancellazione di un elemento X dalla lista

```
Deletion - del(X,L).
del(Y,[Y],[]).
del(X,[X, |L1],L1).
del(X,[Y | L],[Y | L1]):-del(X,L,L1).
```

Inserire un elemento in lista

```
Insert - ins(X, L, R) - using del(X,L))
ins(X,L,R):-del(X,R,L).
```

N-esimo elemento della lista

```
Find N-th element of a list
nlist(1,[X | L], X).
nlist(N,[Y | L], X):-N1 is N - 1, nlist(N1,L,X).
```

A questo punto parliamo di **Matching** o **Unification** in Prolog. *Matching* significa che dati due termini per verificare se sono *identici* o le variabili in entrambi i termini possono avere lo stesso oggetto dopo essere stati istanziati. Ad esempio: **date(D, M, 2006) = date(D1, feb, Y1).** significa: **D = D1, M = feb, Y1 = 2006**. Prima di dare le regole generali per il *matching/unification* è necessario ricordare

che ci sono 3 tipi di termini in Prolog: **costanti** (che possono essere *atomi* o *numeri*), **variabili**, **termini complessi (Strutture)** [functor(term1, term2, ...)].
La regola generale è: il *matching tra due termini* avviene se sono uguali o se contengono variabili che possono essere istanziate in modo tale che i termini risultanti siano uguali.

Regole per Matching/Unification

1. Se il *termine1* e il *termine2* sono costanti, allora il *termine1* e il *termine2* corrispondono se e solo se sono lo stesso atomo o lo stesso numero.
2. Se il *termine1* è una variabile e il *termine2* è qualsiasi tipo di termine, allora il *termine1* e il *termine2* corrispondono e il *termine1* è istanziato al *termine2*. Analogamente, se il *termine2* è una variabile e il *termine1* è qualsiasi tipo di termine, allora il *termine1* e il *termine2* corrispondono e il *termine2* viene istanziato al *termine1*. (Quindi se sono entrambe variabili, sono entrambe istanziate l'una con l'altra e diciamo che condividono valori).
3. Se il *termine1* e il *termine2* sono termini complessi, corrispondono se e solo se:
   (a) Hanno stessa funzione e arità.
   (b) Tutti gli argomenti corrispondenti corrispondono.
   (c) Le istanziazioni di variabili sono compatibili. (Ad esempio, non è possibile istanziare la variabile X su *mia*, quando si abbina una coppia di argomenti, e quindi istanziare X su *vincent*, quando si abbina un'altra coppia di argomenti.)
4. Due termini corrispondono se e solo se segue dalle tre clausole precedenti che corrispondono.

Un altro argomento importante in Prolog è il **Backtracking**. A volte si vuole impedire il backtracking per motivi di efficienza.
Prevenire il backtracking: Prolog farà automaticamente un passo indietro se ciò è necessario a raggiungere un obiettivo. Il backtracking non controllato può causare inefficienza in un programma. Dei "Cut", (!) posssono essere utilizzati per impedire il backtracking.

Esempio 1
Date le seguenti 3 regole:

- R1: **f(X, 0):- X < 3.** - "se X < 3 allora Y = 0".
- R2: **f(X, 2):- 3 =< X, X < 6.** - "se 3 =< X e X < 6 allora Y = 2".
- R3: **f(X, 4):- 6 =< X.** - "se 6 <= X allora Y = 4".

A questo punto ci si chiede: **?-f(1, Y), 2 < Y.** Il primo goal [f(1, Y)] utilizza R1 ed istanzia Y = 0, mentre il secondo goal fallisce in quanto (2 ¡ 0). Prolog effettua backtracking per utilizzare l'altra regola, in modo da risolvere i goal. È possibile notare che tutti i predicati sono mutuamente esclusivi, ovvero quando uno è vero gli altri sono falsi. Quindi in questo caso il backtracking è solo un processo che consuma tempo, e che vorremmo evitare. Possiamo evitarlo utilizzando Cut(!), modificando il codice come segue:

- R1: **f(X, 0):- X < 3, !.**
- R2: **f(X, 2):- 3 =< X, X < 6, !.**
- R3: **f(X, 4):- 6 =< X.**

Esempio 2
max(X, Y, Max), ovvero in Max sarà presente il valore maggiore tra X e Y, in Prolog:
max(X, Y, X):- X >= Y.
max(X, Y, Y):- X < Y.

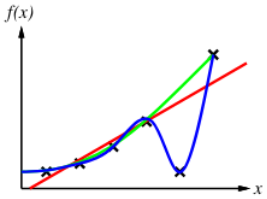
Ora è possibile cambiare l'algoritmo aggiungengo Max.
max(X, Y, X):- X >= Y, !.
max(X, Y, Y).
Un altro modo per scriverlo utilizzando i cut è:
max1(X, Y, Max):- X >= Y, Max = X, !; Max = Y.

Negazione come fallimento: si vuole rappresentare in Prolog la frase "A Mary piacciono tutti gli animali tranne che i serpenti":
like(mary, X):- snake(X), !, fail.
like(mary, X):- animal(X).

Non-relazione: not(Goal) è vero se il goal è falso.
not(P):- P, !, fail; true.

9 Machine Learning

Inductive learning
In breve lo scopo è trovare un'*ipotesi* h che si avvicini il più possibile alla funzione target f, dato un insieme di esempi. Si costruisce/aggiusta h per far si che concordi con f sull'insieme di allenamento. Si parla di **consistenza** se h coincide con f per tutti gli esempi.



Supervised learning
L'agente osserva un training set contenente un'insieme di coppie (Input, Output) ed impara una funzione che mappi gli input negli output. Si parla di **classificazione** quando l'output ha valori *discreti*. Mentre si parla di **regressione** quando l'output ha valori *continui*. L'input invece viene fornito in termini di *features* che rappresentano i dati, dati etichettati.

Reinforcement learning
L'agente impara da *ricompense* o *punizioni*.

Unsupervised learning
L'agente impara pattern nell'input senza alcun feedback dall'output, qui si utilizzano tecniche di *clustering*.

Semi-supervised learning
Pochi esempi etichettati e conosciuti, all'interno di un insieme più grande di dati non etichettati (es. riconoscere l'età di una persona dalla foto).

9.1 Supervised learning: regressione lineare

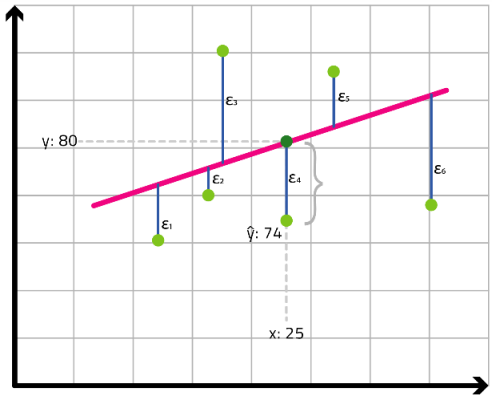
La regressione è un processo statistico che cerca di stabilire una relazione tra due o più variabili, si differenzia nettamente dalla *classificazione*, poiché quest'ultima si limita a discriminare gli elementi in una determinato numero di classi (label), mentre nella prima l'input è un dato e il sistema ci restituisce un output reale (a differenza della classificazione che riceve in input un dato e restituisce in ouput una label, un'etichetta a cui il dato appartiene e in generale quindi un valore discreto).
La formulazione matematica dell'**ipotesi** (ovvero della retta che descrive l'andamento dei dati) è piuttosto semplice, essa infatti, dovendo assumere la forma di una retta, si presenterà nel modo seguente:

$$h_\theta(x_1) = \theta_1 x_1 + \theta_0$$

In cui θ_0 rappresenta l'*intersezione* della retta con l'asse x, mentre θ_1 rappresenta la *pendenza* della retta. Il problema di individuare i migliori parametri θ_0 e $theta_1$ affinché la retta descriva con precisione i dati del nostro training set si può riformulare come un problema di *minimizzazione*. Qui entra in gioco la funzione costo (che ci permette di discriminare tra le infinite possibili rette). Definiamo questa funzione nel modo seguente:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

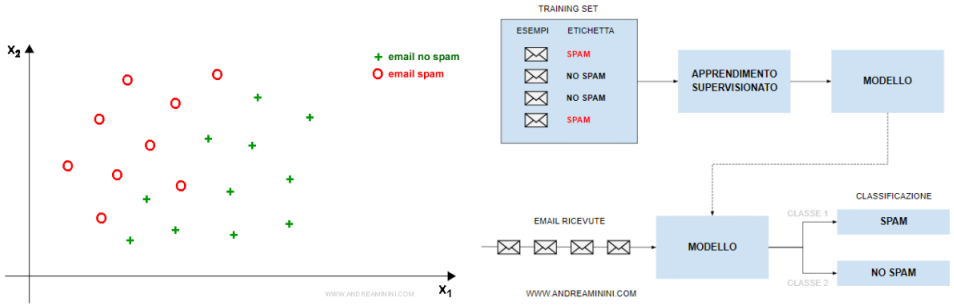
Questa funzione calcola, attraverso una sommatoria, lo scarto quadratico medio di tutti gli errori calcolati per gli m elementi di input, prima di dividere il valore così ottenuto per m = cardinalità del training set. Rispetto a tale funzione, è nostro interesse individuare la somma che restituisce il valore minimo che la funzione J può assumere: $min_{(\theta_0,\theta_1)} J(\theta_0, \theta_1)$.



Minimizzando la funzione costo J ci assicuriamo di minimizzare la somma di tutte le distanze blu che separano la retta dai punti.

9.2 Supervised learning: classificazione

L'algoritmo viene istruito dal supervisore a riconoscere le *categorie* tramite una serie di esempi pratici (training set). In ogni esempio sono forniti alla macchina: le variabili descrittive dell'ambiente (x) un'etichetta per indicare il risultato desiderato (y) ossia la classe di appartenenza dell'esempio. Il sistema elabora gli esempi alla ricerca di una regola generale di classificazione detta modello. Una volta costruito il modello, la macchina lo utilizza per classificare le nuove istanze, sulla base delle osservazioni compiute sull'training set. Di seguito un esempio per la classificazione di email di spam.



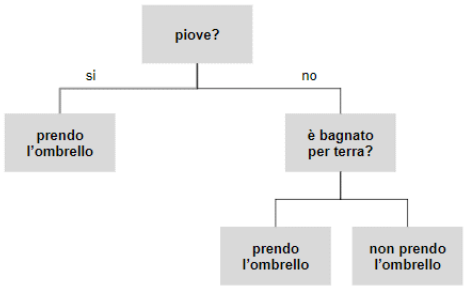
9.3 Unsupervised learning

L'apprendimento non supervisionato è una tecnica di apprendimento automatico che consiste nel fornire al sistema informatico una serie di input (esperienza del sistema) che egli riclassificherà ed organizzerà sulla base di caratteristiche comuni per cercare di effettuare ragionamenti e previsioni sugli input successivi. Al contrario dell'apprendimento supervisionato, durante l'apprendimento vengono forniti all'apprendista solo esempi non annotati, in quanto le classi non sono note a priori ma devono essere apprese automaticamente. Sono due i metodi principali: **analisi dei componenti principali**, si trova una sequenza di combinazioni

lineari delle variabili che hanno varianza massima e sono mutuamente scorrelate. **Clustering**, ricerca di sotto-gruppi di dati non conosciuti, si ricerca una partizione dei dati in modo che le osservazioni all'interno del gruppo siano piuttosto simili tra loro. Tecniche specifiche sono: *clustering k-means* e *clustering gerarchico*.

9.4 Alberi di decisione

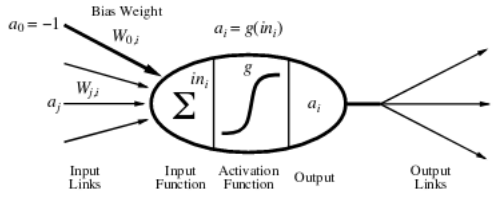
Un albero di decisione è un sistema con n variabili in input e m variabili in output. Le variabili in input (attributi) sono derivate dall'osservazione dell'ambiente. Le variabili in output, invece, identificano la decisione/azione da intraprendere. Il processo decisionale è rappresentato con un albero logico rovesciato dove ogni nodo è una funzione condizionale. Ogni nodo verifica una condizione (test) su una particolare proprietà dell'ambiente (variabile) e ha due o più diramazioni verso il basso in funzione. Il processo consiste in una sequenza di test. Comincia sempre dal nodo radice, il nodo genitore situato più in alto nella struttura, e procede verso il basso. A seconda dei valori rilevati in ciascun nodo, il flusso prende una direzione oppure un'altra e procede progressivamente verso il basso. La decisione finale si trova nei nodi foglia terminali, quelli più in basso. In questo modo, dopo aver analizzato le varie condizioni, l'agente giunge alla decisione finale.



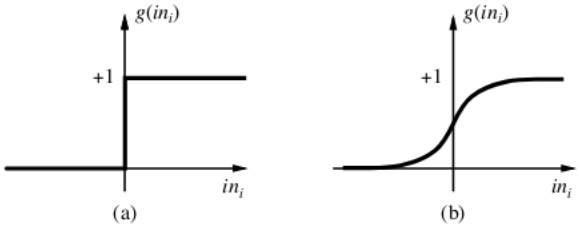
10 Reti neurali

Le **unità** McCulloch-Pitts rappresentano un modello matematico di un singolo neurone, l'output consiste in una funzione lineare dell'input "appiattita". Questa funzione è chiamata **funzione di attivazione**:

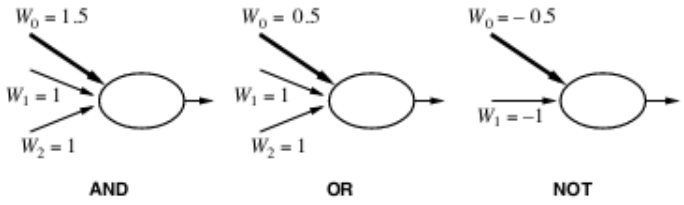
$$a_i \leftarrow g(in_i) = g(\sum_j W_{j,i} a_j)$$



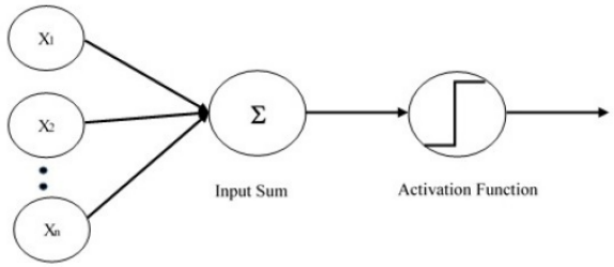
Una **funzione di attivazione** è una funzione che viene attivata al superamento di un certo valore, può essere una *step function* (o *threshold function*) oppure una *funzione sigmoidea*: $1/(1+e^{-x})$. Cambiando il peso di bias $W_{0,i}$ si sposta la posizione della threshold.



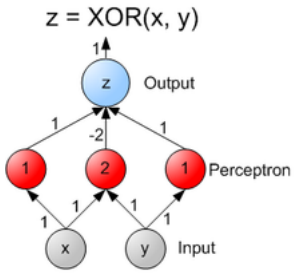
Ad esempio le **funzioni booleane** (o funzioni logiche), possono essere implementate come di seguito:



Rete Neurale Single-Layer: Perceptron
Questo è il modello più semplice di rete neurale (in realtà non è una rete in quanto è presente un solo neurone, il percettrone). Il contenuto della memoria locale del neurone è costituito da un vettore di pesi. Il calcolo di un percettrone a singolo strato viene eseguito sul calcolo della somma di ciascun elemento del vettore di input moltiplicato per l'elemento corrispondente del vettore dei pesi: $\sum_i x_i \cdot W_i$. Il valore visualizzato nell'output sarà l'input per la funzione di attivazione, la quale restituirà o meno un output in base a se viene attivata o meno. Questo flusso: input + sommatoria + funzione di attivazione = output è chiamato *feed-forward*.



Rete Neurale Feed-Forward
Una rete neurale feed-forward è una rete neurale artificiale dove le connessioni tra le unità non formano cicli, differenziandosi dalle reti neurali ricorrenti. Questo tipo di rete neurale fu la prima e più semplice tra quelle messe a punto. In questa rete neurale le informazioni si muovono solo in una direzione, avanti, rispetto ai nodi d'ingresso, attraverso nodi nascosti (se esistenti) fino ai nodi d'uscita. Queste reti non hanno memoria degli input avvenuti in tempi precedenti, per cui l'output è determinato solamente dall'attuale input.



11 Aspetti economici, filosofici ed etici dell'IA

Esistono molte definizioni sull'IA:
AI debole: è possibile costruire macchine in grado di agire come se fossero intelligenti?
AI Forte: è possibile costruire macchine realmente intelligenti?
Per porre questa domanda è necessario prima definire cos'è l'*intelligenza*. In una certa misura possiamo affermare che un'intelligenza artificiale debole è possibile ed esistente ai nostri giorni.

Obiezione di coscienza all'IA: le macchine possono *pensare*? Le macchine che superano il test di Turing possono essere viste come macchine che pensano, tuttavia questa è solo una simulazione del pensiero.

Ci sono due visioni opposte:

1. **Naturalismo biologico**: gli stati mentali sono caratteristiche emergenti di alto livello che risultano da processi fisici di basso livello nei neuroni ed è la proprietà non specificata dei neuroni che conta.

2. **Funzionalismo**: uno stato mentale è una condizione casuale intermedia tra input e output. Qualsiasi due sistemi con processi casuali isomorfi avrebbero gli stessi stati mentali. Pertanto, un programma per computer potrebbe avere gli stessi stati mentali di una persona. Il presupposto è che esiste un certo livello di astrazione al di sotto del quale l'implementazione specifica non ha importanza. Questo fatto è mostrato nell'*esperimento di sostituzione del cervello*.

11.1 Aspetti etici dell'IA

Etica delle macchine: ipotesi computazionali e filosofiche per le macchine che possono prendere decisioni morali *autonome*. Domande come: i veicoli autonomi chi dovrebbero uccidere in caso di incidente? I robot medici dovrebbero dire la verità ai pazienti? Come garantire che le macchine non prendano decisioni immorali?

Un altro problema è che il comportamento umano e il comportamento della macchina sono regolati da leggi diverse: un esempio è il problema del trolley. Il *Trolley Problem* è una versione del dilemma etico che esorta a fare una scelta: sacrificare una vita per salvarne cinque o viceversa? Questo dilemma è legato all'utilitarismo che afferma che la decisione moralmente corretta è quella che massimizza il benessere per il maggior numero di persone.