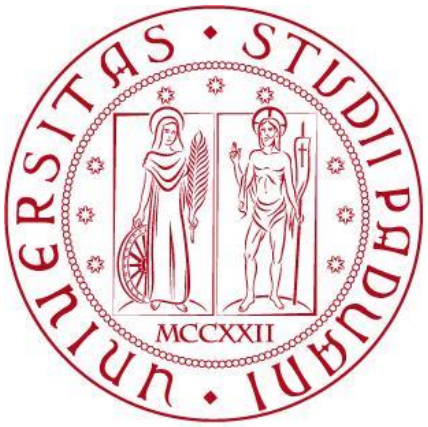


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

INTELLIGENZA ARTIFICIALE

(Versione 17/06/2018)

Stesura a cura di:
Stefano Ivancich

INDICE

Questa dispensa è scritta da studenti senza alcuna intenzione di sostituire i materiali universitari. Essa costituisce uno strumento utile allo studio della materia ma non garantisce una preparazione altrettanto esaustiva e completa quanto il materiale consigliato dall'Università.
Lo scopo di questo documento è quello di riassumere i concetti fondamentali degli appunti presi durante la lezione, riscritti, corretti e completati facendo riferimento alle slide e ai libro di testo: "*S.Russell, P.Norvig, Intelligenza Artificiale. Un approccio moderno. Milano-Torino: Pearson Prentice Hall, 2010. Volume I - Terza Edizione*" per poter essere utilizzato per una veloce consultazione. Non sono presenti esempi e spiegazioni dettagliate, per questi si rimanda ai testi citati e alle slide.

Se trovi errori ti preghiamo di segnalarli qui:
www.stefanoivancich.com
ivancich.stefano.1@gmail.com
Il documento verrà aggiornato al più presto.

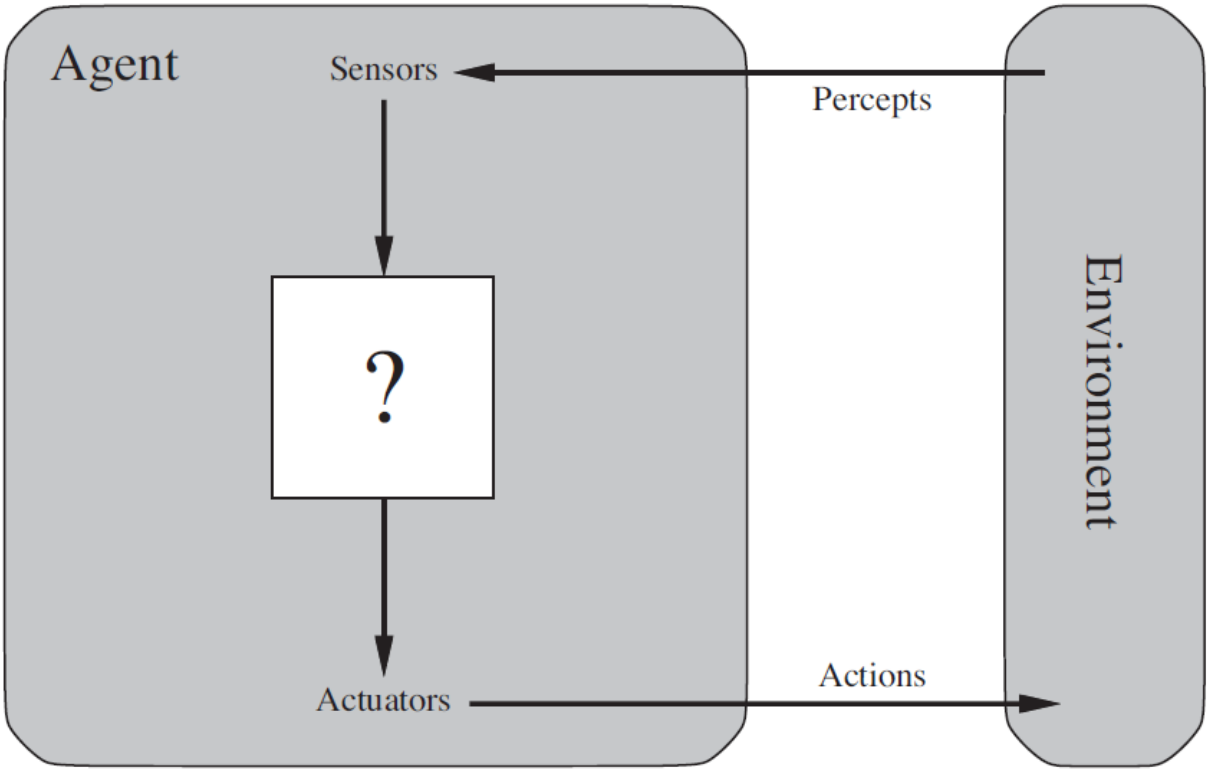
1.Agenti intelligenti

1.1. Agenti razionali

Agente: un sistema che percepisce il suo ambiente attraverso dei **sensori** e agisce su di esso mediante degli **attuatori**.
Percezione: input percettivi in un dato istante.
Sequenza percettiva: la storia completa di tutto quello che l'agente ha percepito nella sua esistenza.
Funzione agente: è una funzione matematica astratta che descrive il comportamento di un agente, descrive la corrispondenza tra una qualsiasi sequenza percettiva e una specifica azione.
Programma agente: implementazione concreta della funzione agente, prendono in input la percezione corrente dei sensori e restituiscono un'azione agli attuatori.
Autonomia: quando un agente NON si basa sulla conoscenza pregressa inserita dal programmatore.
Misure di prestazione: valutano una sequenza di stati dell'ambiente. Vanno progettate in base all'effetto che si desidera ottenere sull'ambiente anzi che su come si pensa che debba comportarsi l'ambiente.
In un dato istante, ciò che è razionale di pende da:

- la misura di prestazione che definisce il criterio di successo;
- la conoscenza pregressa dell'ambiente da parte dell'agente;
- le azioni che l'agente può effettuare;
- la sequenza percettiva dell'agente fino all'istante corrente.

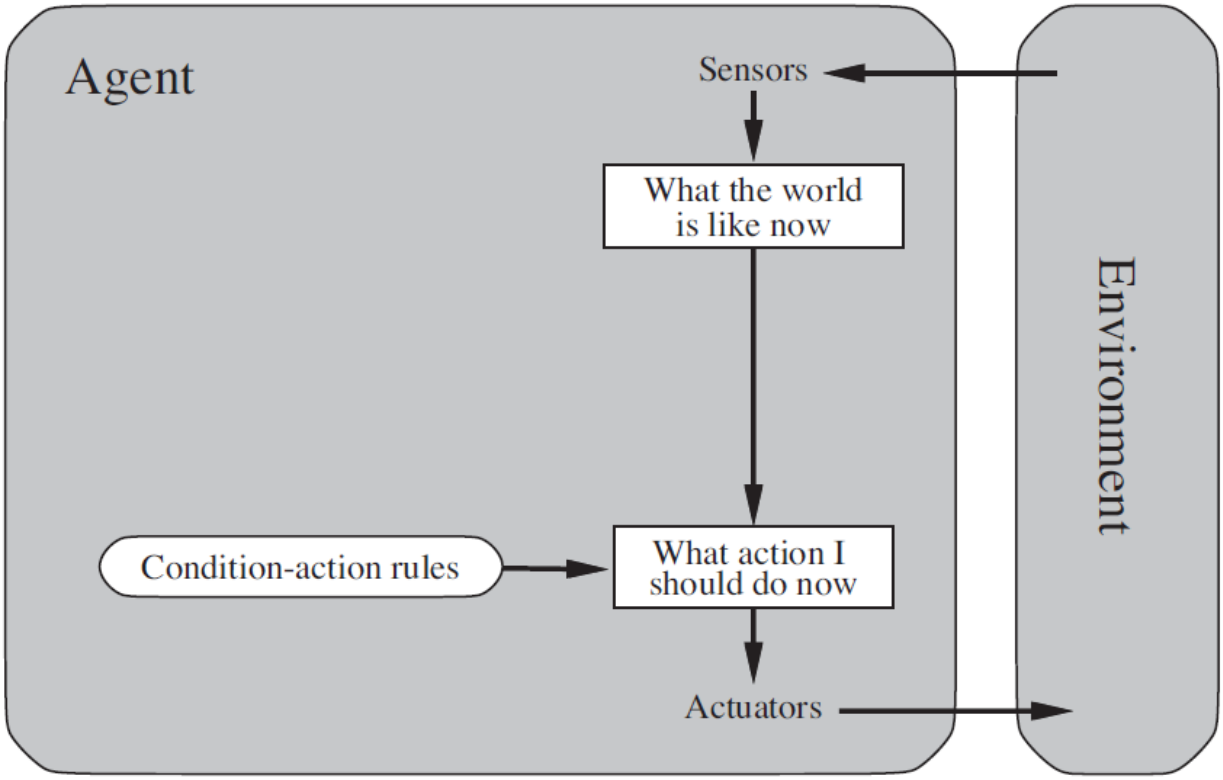
Agente razionale: per ogni possibile sequenza di percezioni, sceglie un'azione che massimizzi il valore atteso della sua misura di prestazione, date le informazioni fornite dalla sequenza percettiva e da ogni ulteriore conoscenza dell'agente.
Non si limita a raccogliere informazioni, ma deve anche essere in grado di apprendere il più possibile sulla base delle proprie percezioni per compensare la sua iniziale conoscenza parziale o erronea.



1.2. La struttura degli agenti

1.2.1. Agenti reattivi semplici

Scelgono le azioni sulla base della percezione corrente, ignorando tuta la storia percettiva.
Sono basati su regole condizione-azione (***if*** *condizione* ***then*** *azione*).
L'ambiente deve essere **completamente osservabile**, anche una minima parte di non-osservabilità può causare grandi problemi.
Spesso non sono in grado di evitare cicli infiniti quando operano in ambienti parzialmente osservabili, questo si può risolvere randomizzando talvolta le azioni.



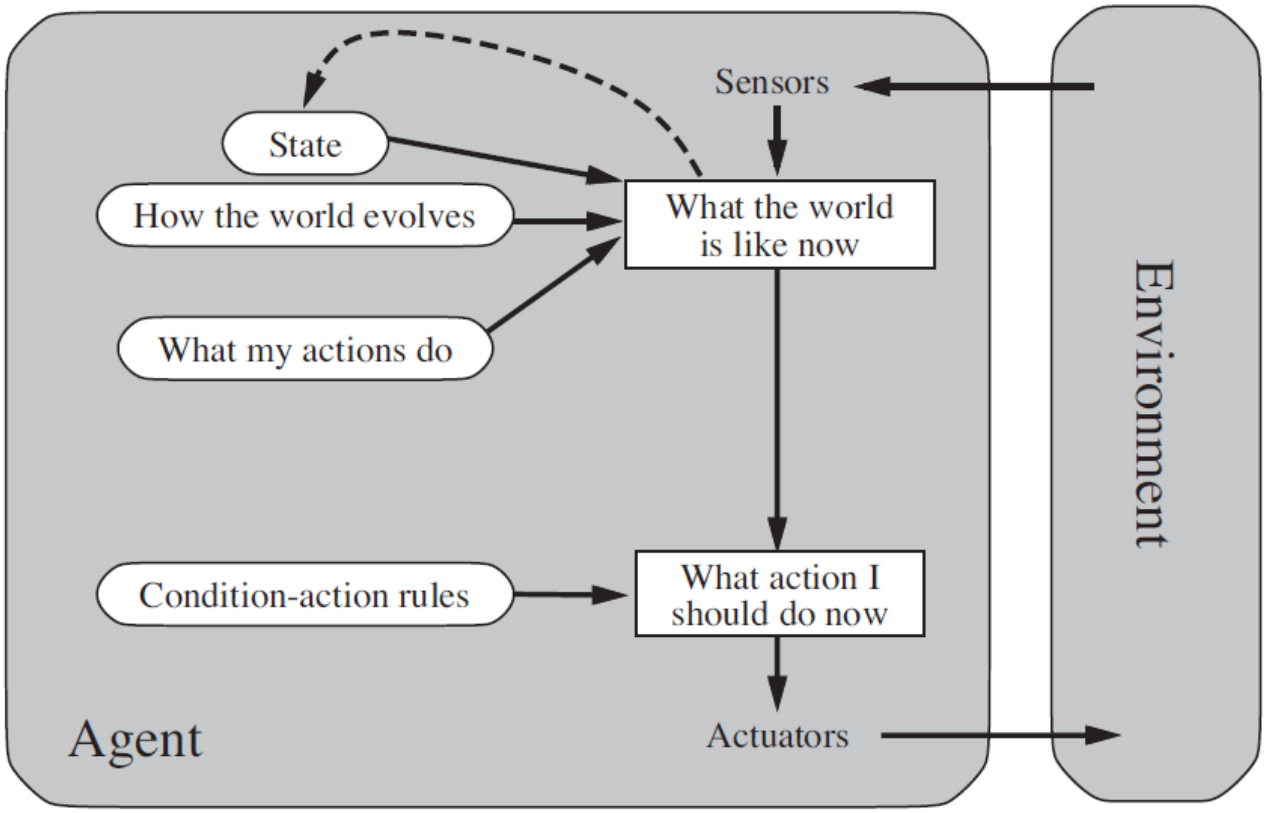
function SIMPLE-REFLEX-AGENT(*percept*) **returns** an action
persistent: *rules*, a set of condition–action rules

state ← INTERPRET-INPUT(*percept*)
rule ← RULE-MATCH(*state*, *rules*)
action ← *rule*.ACTION
return *action*

1.2.2. Agenti reattivi basati su modello

Per gestire l'osservabilità parziale si può tener traccia della parte del mondo che non si può vedere nell'istante corrente.
Quindi l'agente mantiene uno **stato interno** che dipende dalla storia delle percezioni.
Possiede due tipi di conoscenza:

- Informazioni sull'evoluzione del mondo indipendentemente dalle sue azioni.
- Informazioni sull'effetto delle sue azioni sul mondo.

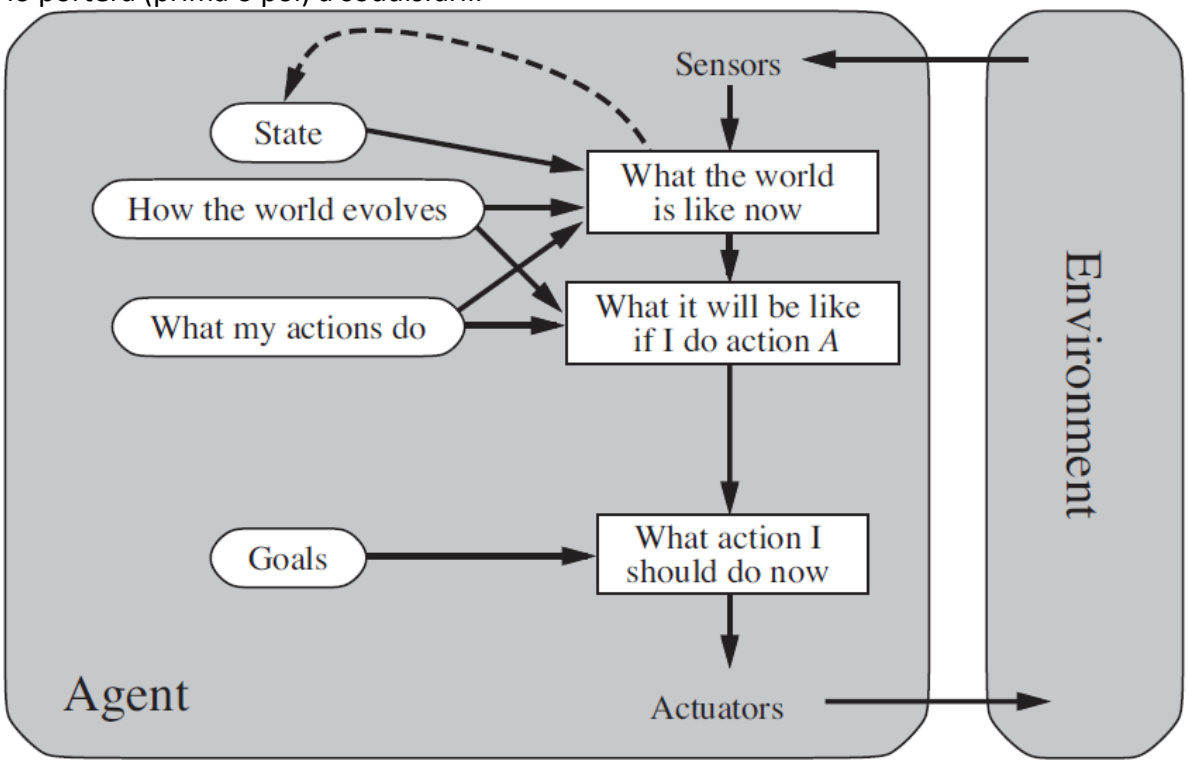


function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action
  persistent: *state*, the agent's current conception of the world state
    model, a description of how the next state depends on current state and action
    rules, a set of condition–action rules
    action, the most recent action, initially none

  state ← UPDATE-STATE(*state*, *action*, *percept*, *model*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← *rule*.ACTION
  return *action*

1.2.3. Agenti basati su obiettivi

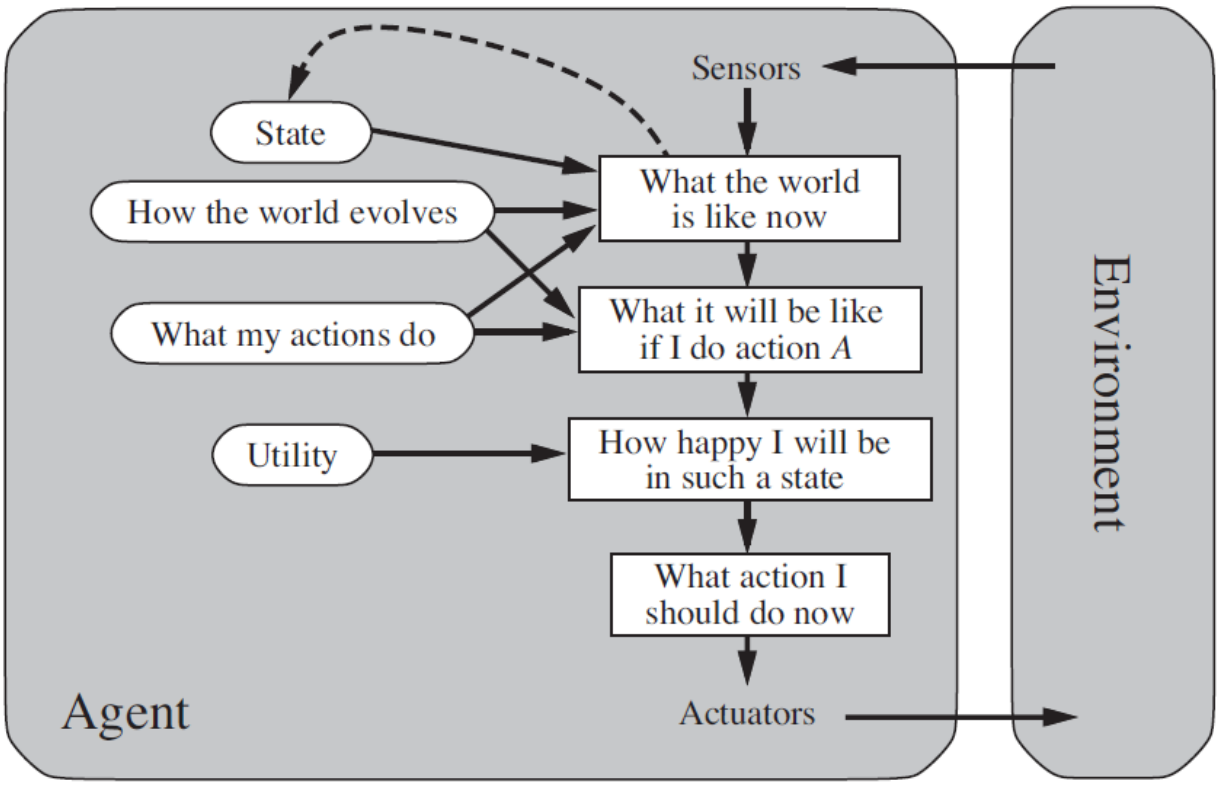
Oltre a tener traccia dello stato dell'ambiente, memorizza un insieme di obiettivi e sceglie l'azione che lo porterà (prima o poi) a soddisfarli.



1.2.4. Agenti basati sull'utilità

Funzione di utilità: è un internalizzazione della misura di prestazione.

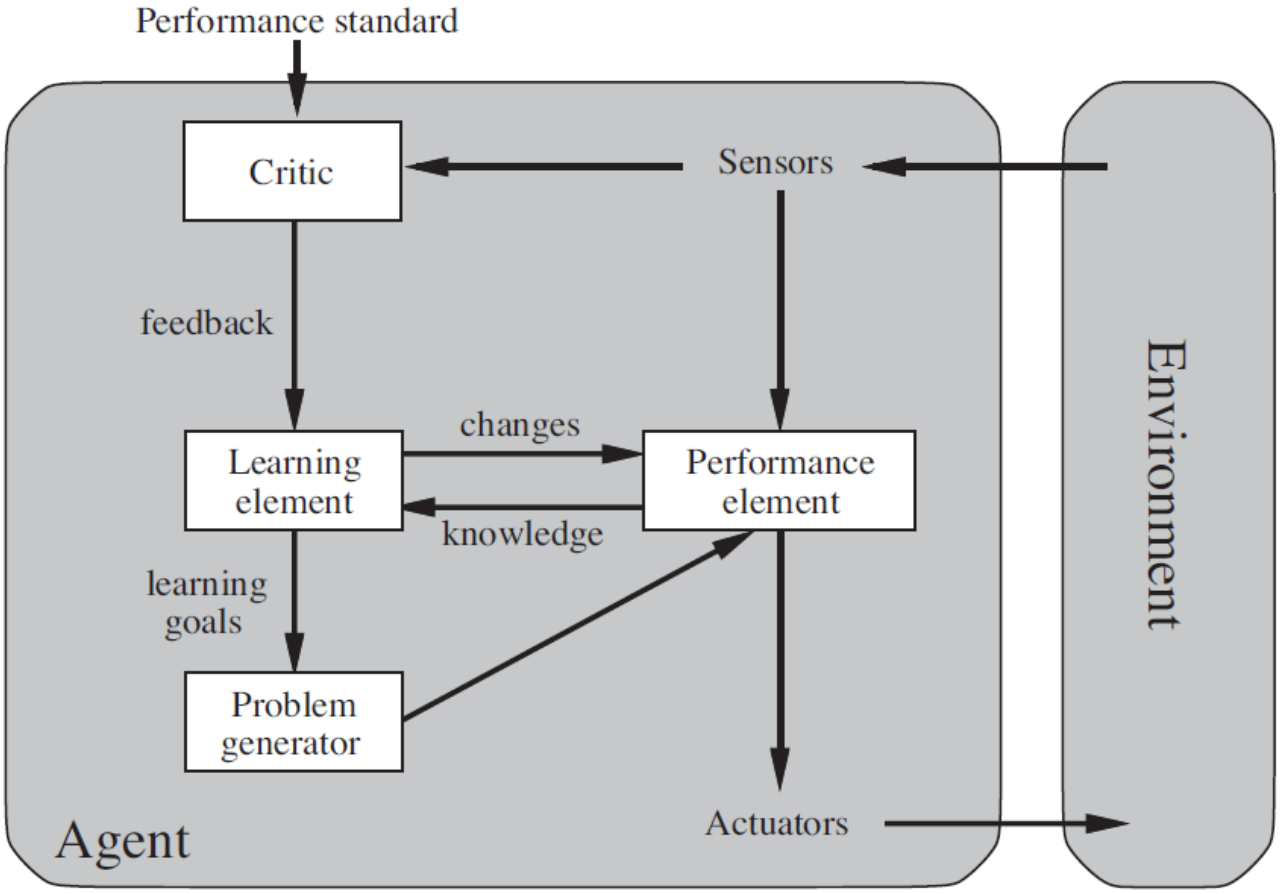
Sceglie l'azione che massimizza l'utilità attesa dei risultati, ovvero l'utilità che l'agente si attende di ottenere.



1.2.5. Agenti capaci di apprendere

Diviso in quattro componenti:

- **Elemento esecutivo:** si occupa della selezione delle azioni esterne, equivale agli agenti descritti precedentemente.
- **Elemento critico:** dice a quello di apprendimento come si sta comportando l'agente rispetto ad uno standard di prestazione prefissato. È necessario perché di per se le percezioni non forniscono alcuna indicazione del successo dell'agente.
- **Elemento di apprendimento:** responsabile del miglioramento interno, utilizza le informazioni provenienti dall'elemento critico riguardo le prestazioni correnti dell'agente e determina se e come modificare l'elemento esecutivo affinché in futuro si comporti meglio. Può modificare uno qualsiasi dei componenti "di conoscenza" mostrati nei diagrammi precedenti.
- **Generatore di problemi:** suggerisce le azioni che portino a esperienze nuove e significative.



1.3. Ambienti

Gli ambienti sono i problemi di cui gli agenti razionali rappresentano le soluzioni.
Definire i PEAS (Performance, Environment, Actuators, Sensors).

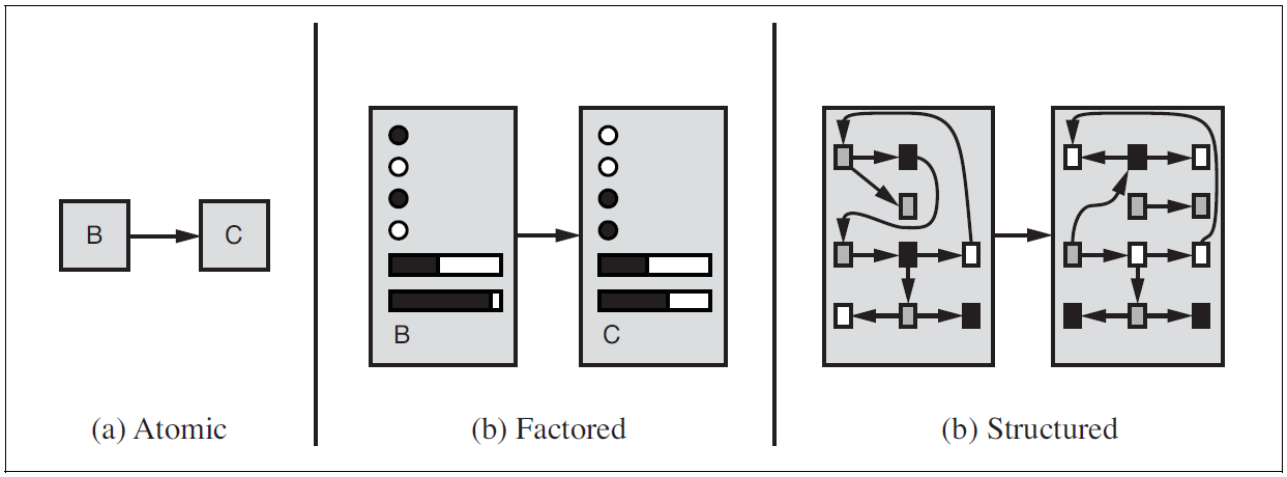
Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Proprietà degli ambienti:

- **Completamente osservabile vs. parzialmente osservabile:**
  Completamente quando i sensori danno accesso allo stato completo dell'ambiente in ogni momento, o almeno che misurino gli aspetti rilevanti.
  Parzialmente se i sensori sono inaccurati, presenza di rumore o alcuni dati rilevanti non vengono catturati da essi.
- **Agente singolo vs. multiagente:** agente singolo, multiagente competitivo o multiagente cooperativo.
- **Deterministico vs. stocastico:**
  Deterministico se lo stato successivo dell'ambiente è completamente determinato dallo stato corrente e dall'azione eseguita dall'agente. Altrimenti è stocastico.
- **Episodico vs. sequenziale:**
  Episodico quando l'esperienza dell'agente è divisa in episodi atomici, ogni episodio non dipende dalle azioni intraprese in quelli precedenti.
  Sequenziale quando ogni decisione può influenzare tutte quelle successive.
- **Statico vs. dinamico:**
  Dinamico se l'ambiente può cambiare mentre un agente sta pensando altrimenti è statico.
  Semidinamico se l'ambiente non cambia col passare del tempo, ma la valutazione della prestazione dell'agente sì.
- **Discreto vs. continuo:** ci si riferisce in modo in cui è gestito il tempo.
- **Noto vs. ignoto:** ci si riferisce alla conoscenza che ha l'agente circa le "leggi fisiche" dell'ambiente. In un ambiente noto sono conosciuti i risultati per tutte le azioni.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

1.4. Rappresentazione degli stati e transizioni



Rappresentazione atomica: ogni stato del mondo è indivisibile, non ha struttura interna.
Rappresentazione fattorizzata: suddivide ogni stato in un insieme fissato di variabili o attributi, che possono essere booleani, numeri reali, …. Si può rappresentare l'incertezza lasciando vuoto l'attributo corrispondente.
Rappresentazione strutturata: include oggetti, ognuno dei quali può avere attributi propri oltre a relazioni con altri oggetti.

2. Risolvere i problemi con la ricerca

Metodi utilizzabili da:

- Agente: **basato su obiettivi**
- Rappresentazione: **atomica**
- Ambiente:
  - **Completamente osservabile** (stato iniziale conosciuto)
  - **Deterministico** (non si possono gestire eventi inaspettati)
  - **Statico**
  - **Completamente noto**

Algoritmo di ricerca: prende un problema come input e restituisce una soluzione sotto forma di sequenza di azioni.

L'agente svolge ciclicamente i seguenti passi:

1. Formula un obiettivo
2. Formula un problema da risolvere
3. Invoca una procedura di ricerca
4. Esegue la sequenza di azioni restituita dalla procedura di ricerca, mentre la esegue vengono ignorate le percezioni, perché le conosce in anticipo.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

2.1. Definizione dei problemi

Un **problema** è definito da:

- **Stato iniziale:** in cui si trova l'agente.
- **Azioni possibili** dell'agente: dato uno stato *s*, Azioni(*s*) restituisce l'insieme di azioni che possono essere eseguite in *s*.
- **Modello di transizione:** descrizione di ciò che ogni azione fa. Risultato(*s*, *a*) restituisce lo stato risultante dall'esecuzione dell'azione a nello stato *s*.
- **Test obiettivo:** funzione che verifica se uno stato è l'obiettivo.
- **Costo di cammino:** assegna un costo numerico ad ogni cammino.
  Costo di passo c(*s*, *a* ,*s'*): costo che l'azione *a* fa passare lo stato *s* ad *s'*.

Soluzione: è una sequenza di azioni che porta dallo stato iniziale a uno stato obiettivo. La sua qualità è in funzione del costo di cammino.

2.2. Cercare soluzioni

Spazio degli stati: l'insieme di tutti gli stati raggiungibili a partire da quello iniziale mediante qualsiasi sequenza di azioni. Questo forma un grafo (albero di ricerca), i cui nodi rappresentano gli stati e gli archi le azioni.
Cammino: sequenza di stati collegati da una sequenza di azioni.
Frontiera: insieme dei nodi che sono stati generati e non ancora espansi. Ogni elemento è un nodo foglia (momentaneamente, potrebbe avere figli).
Stato ripetuto: causato da un cammino ciclico o ridondante, può causare il fallimento di alcuni algoritmi. Per evitare i cammini ridondanti spesso si usa una lista chiusa in cui si tiene traccia di dove si è passati.

2.2.1. Strutture dati per algoritmi di ricerca e prestazioni

Per ogni nodo dell'albero si hanno: Stato, Padre, Azione, Costo di cammino.
I nodi vengono memorizzati in una coda le cui operazioni su di essa sono: isEmpy, push, pop. Può essere FIFO, LIFO o con priorità.

Le prestazioni si misurano in base a:

- **Completezza**: l'algoritmo garantisce di trovare una soluzione.
- **Ottimalità**: l'algoritmo trova la soluzione ottima (quella con costo di cammino più piccolo).
- **Complessità temporale:** tempo necessario per trovare la soluzione.
- **Complessità spaziale:** quanta memoria necessità per effettuare la ricerca.

La complessità si esprime usando tre quantità:

- **Fattore di ramificazione *b*:** numero massimo di successori di un nodo (n° azioni max che si possono fare su quel nodo come prossima azione).
- **Profondità *d*:** del nodo obiettivo più vicino allo stato iniziale, ovvero il numero di passi lungo il camino a partire dalla radice.
- **Lunghezza massima *m*** dei cammini nello spazio degli stati.

2.3. Strategie di ricerca non informata

Sono strategie che non dispongono di informazioni aggiuntive sugli stati oltre a quella fornita nella definizione del problema.

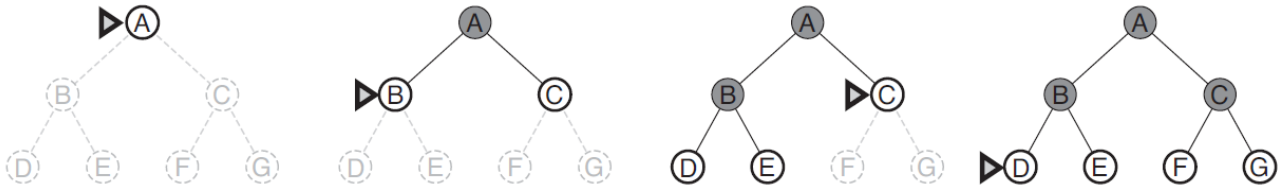
2.3.1. Ricerca in ampiezza (Breadth First)

Si espande prima il nodo radice, poi tutti i suoi successori, cioè **espande i nodi meno profondi**. Tutti i nodi ad un certo livello devono essere espansi prima che si possa espandere uno dei nodi al livello successivo.

Frontiera: coda **FIFO** (i successori sono inseriti in fondo)

Proprietà:

- Completezza: solo se b è finito.
- Ottimalità: solo se costo per ogni passo = 1.
- Complessità temporale: $\boldsymbol{O(b^{d+1})}$
- Complessità spaziale: $\boldsymbol{O(b^{d+1})}$
  Si mantiene ogni nodo in memoria e questo scaturisce il problema principale di questo algoritmo.



function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

```
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
frontier ← a FIFO queue with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
            frontier ← INSERT(child, frontier)
```

2.3.2. Ricerca in profondità (Depth First)

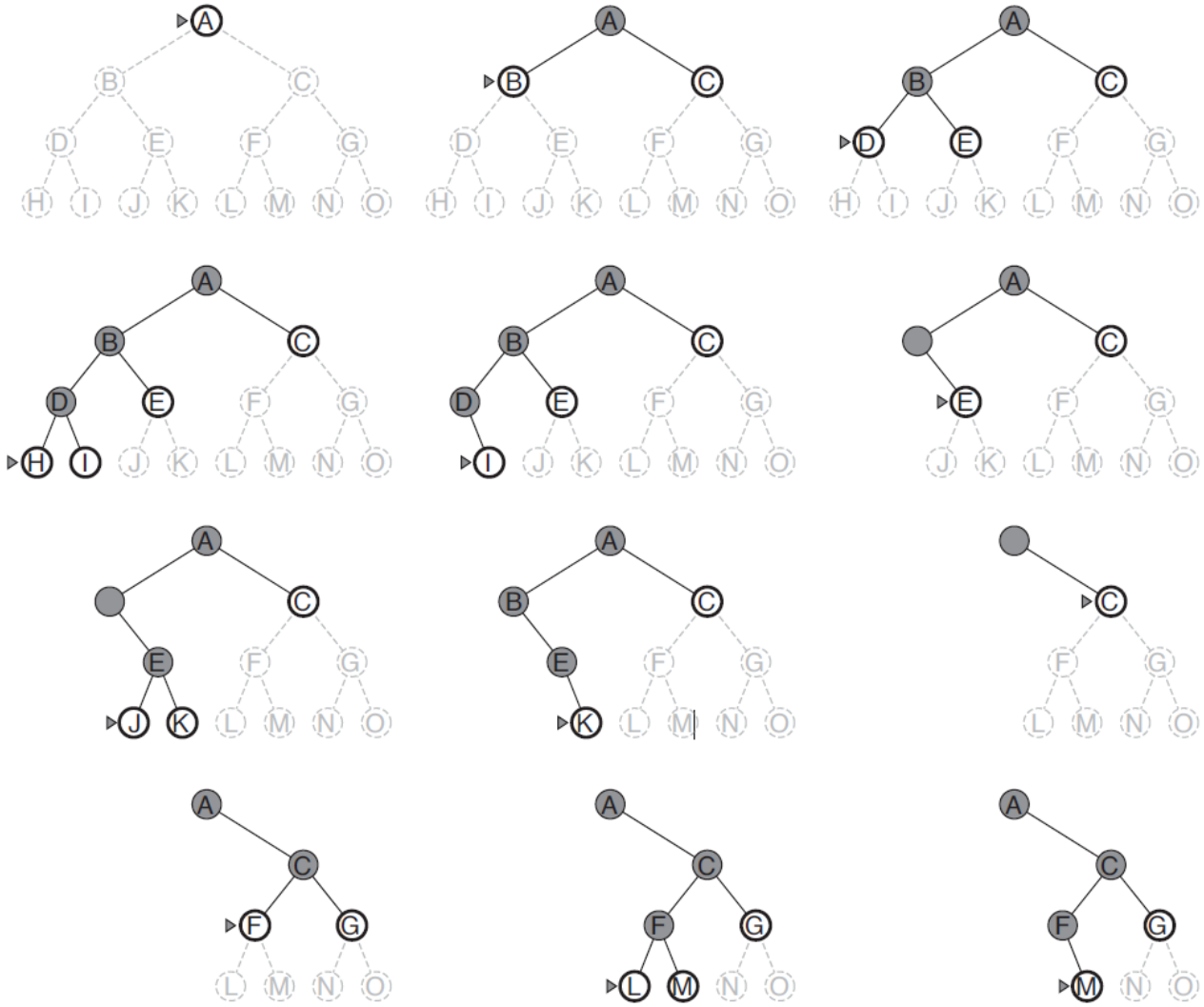
Espande sempre per primo il nodo più profondo nella frontiera dell'albero di ricerca.

È problematica solo se *m* è molto più grande di *d*, altrimenti è più veloce di breadth-first.

Frontiera: coda **LIFO** in cui viene scelto per l'espansione l'ultimo nodo generato.

Proprietà:

- Completezza: solo negli spazi di stati finiti.
- Ottimalità: No.
- Complessità temporale: $\boldsymbol{O(b^m)}$
- Complessità spaziale: $\boldsymbol{O(bm)}$
  I nodi espansi vengono cancellati dalla coda.



2.3.3. Ricerca a profondità limitata (Depth Limited)

Per evitare il problema degli alberi illimitati si imposta un limite alla profondità = L.

Proprietà:

- Completezza: solo se L>=d.
- Ottimalità: No.
- Complessità temporale: $O(b^l)$
- Complessità spaziale: $O(bl)$

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

2.3.4. Ricerca ad approfondimento iterativo (Iterative Deepening)

Esegue una serie di ricerche a profondità limitata, incrementando progressivamente il limite di profondità finché non trova una soluzione.
È il metodo preferito di ricerca non informata quando lo spazio di ricerca è grande e la profondità della soluzione non è nota

Proprietà:

- Completezza: Si.
- Ottimalità: solo se i passi hanno costo unitario.
- Complessità temporale: $O(b^d)$
- Complessità spaziale: $O(bd)$

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

2.3.5. Ricerca bidirezionale

Esegue due ricerche in parallelo, una in avanti dallo stato iniziale e l'altra all'indietro dall'obiettivo.
Proprietà:

- Completezza: solo se b è finito e si usa in entrambe le direzioni la ricerca in ampiezza.
- Ottimalità: solo se i passi hanno costo unitario e si usa in entrambe le direzioni la ricerca in ampiezza.
- Complessità temporale: $O(b^{d/2})$
- Complessità spaziale: $O(b^{d/2})$

2.4. Strategie di ricerca informata

Sfrutta conoscenza specifica del problema al di là della sua semplice definizione.
Euristica: regola per scegliere quei rami che presumibilmente portano a una soluzione accettabile del problema.
Si impiegano le euristiche in due casi tipici:

- Un problema **non** ha una soluzione esatta per una ambiguità inerente i dati
- Il problema ha una soluzione esatta ma il costo computazionale per trovarla è proibitivo (eg scacchi): l'euristica 'attacca' la complessità guidando la ricerca lungo il cammino più 'promettente'.

Poiché usano informazioni limitate, non possono predire l'esatto comportamento dello spazio degli stati e possono portare a una soluzione sub-ottima o fallire del tutto.

Ricerca best-first: sceglie il nodo da espandere in base ad una **funzione di valutazione f(n)** che è una stima di costo, quindi viene selezionato prima il nodo con la valutazione più bassa. La scelta di f determina la strategia di ricerca.
Una componente di f è una **funzione euristica h(n)=** costo stimato del cammino più conveniente dallo stato del nodo n a uno stato obiettivo.

2.4.1. Ricerca Best-First Greedy

Cerca sempre di espandere il nodo più vicino all'obiettivo. f(n)=h(n).

Proprietà:

- Completezza: No, può rimanere intrappolata in cicli.
- Ottimalità: No.
- Complessità temporale: $\boldsymbol{O(b^m)}$ con m profondità massima
- Complessità spaziale: $\boldsymbol{O(b^m)}$ tiene tutti i nodi in memoria

Ha i difetti della depth search.

2.4.2. Ricerca A*

La valutazione dei nodi viene eseguita combinando:

- g(n) costo per raggiungere n dal nodo iniziale
- h(n) costo per andare da n all'obiettivo

f(n) = g(n) + h(n): costo stimato della soluzione più conveniente che passa per n.
Euristica ottima se è ammissibile, ovvero non sbaglia mai per eccesso ila stima del costo per arrivare all'obiettivo.

Proprietà:

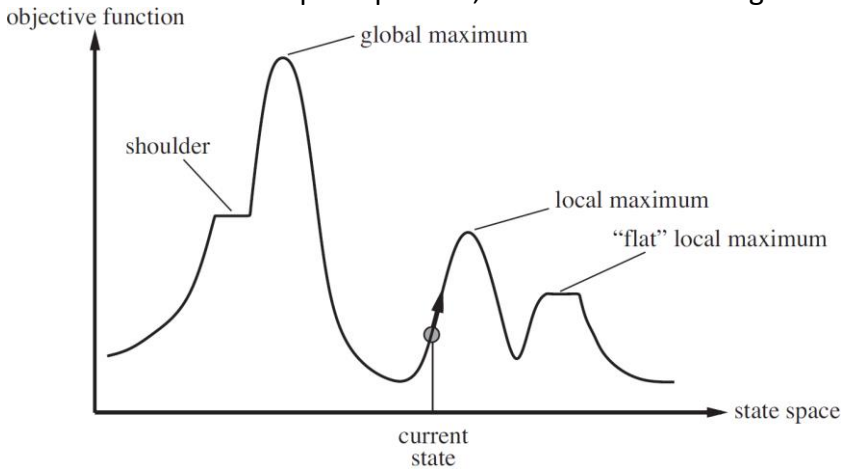
- Completezza: Si.
- Ottimalità: Si.
- Complessità temporale: $\boldsymbol{O(b^m)}$ con m profondità massima
- Complessità spaziale: $\boldsymbol{O(b^m)}$ tiene tutti i nodi in memoria

La complessità spaziale è ancora proibitiva, gli algoritmi RBFS e SMA* utilizzano quantità di memoria limitate.

2.5. Ricerca locale

Per molti problemi il cammino verso l'obiettivo è irrilevante e quello che conta è lo stato soluzione. Gli algoritmi di ricerca locale operano su un singolo nodo corrente invece che sui cammini, e si spostano nei nodi immediatamente adiacenti. I cammini non vengono memorizzati.
Usano poca memoria, possono trovare soluzioni ragionevoli in spazi degli stati grandi o infiniti.
Sono utili per risolvere problemi di ottimizzazione, in cui lo scopo è trovare lo stato migliore secondo una funzione obiettivo.

Panorama dello spazio degli stati: ha una posizione (definita dallo stato e un'altezza (definita dalla funzione di costo).
Lo scopo della ricerca locale è trovare il picco più alto, ovvero un massimo globale.



2.5.1. Ricerca Hill Climbing

Prevede di espandere lo stato corrente e valutare i discendenti: il migliore nodo generato viene scelto per una ulteriore espansione; ne' fratelli ne' genitori sono memorizzati.
Non si tiene traccia della storia: non può tornare indietro dai fallimenti.
Ad ogni passo il nodo corrente viene rimpiazzato dal vicino migliore.
Problema: hill climbing può rimanere bloccato in un massimo locale, se si raggiunge uno stato che ha una migliore valutazione di ogni altro figlio, l'algoritmo si ferma.

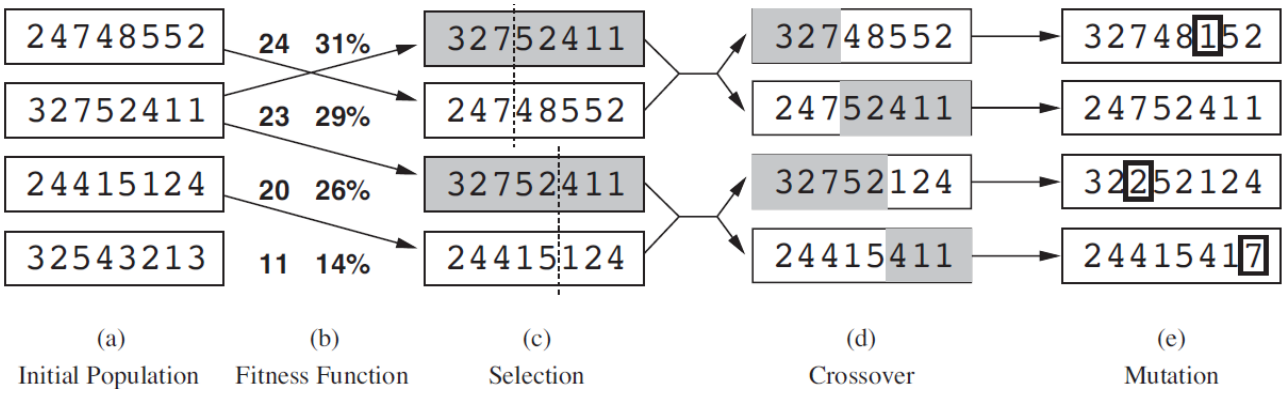
```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
      neighbor ← a highest-valued successor of current
      if neighbor.VALUE ≤ current.VALUE then return current.STATE
      current ← neighbor
```

Hill Climbing Stocastico: sceglie a caso tra tutte le mosse che vanno verso l'alto.
Hill Climbing con prima scelta: genera casualmente i successori.
Hill Climbing con riavvio casuale: conduce una serie di ricerche hill climbing partendo da stati iniziali casuali.
Simulated annealing: invece della mossa migliore sceglie una mossa casuale. Se migliora la situazione viene accettata, altrimenti la accetta con una probabilità inferiore a 1 che decresce con la "cattiva qualità" della mossa.
Local beam: tiene traccia di k stati anzi che di uno solo. Comincia con k stati generati casualmente, ad ogni passo genera i k successivi, se uno di quelli è l'obiettivo, termina; altrimenti scegli i k migliori.

2.5.2. Algoritmi genetici

Variante della beam search, basata su 5 step:

- **Popolazione iniziale:** Comincia con k stati generati casualmente, chiamati **popolazione**. Ogni **individuo** (stato) è rappresentato da una stringa composta da simboli di un alfabeto finito.
- **Funzione di fitness:** Ogni stato riceve una valutazione in base ad una funzione.
- **Selezione:** Viene fatta una scelta casuale di due coppie secondo la probabilità della funzione di fitness.
- **Crossover:** Per ogni coppia viene scelto casualmente un punto di crossover nelle stringhe. Vengono generati i nuovi stati incrociando le stringhe dei genitori nel punto di crossover.
- **Mutazione:** ogni posizione nelle stringhe è soggetta ad una mutazione casuale.



function GENETIC-ALGORITHM(*population*, FITNESS-FN) **returns** an individual
  inputs: *population*, a set of individuals
    FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 **to** SIZE(*population*) **do**
      x ← RANDOM-SELECTION(*population*, FITNESS-FN)
      y ← RANDOM-SELECTION(*population*, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *new_population*
    population ← *new_population*
  until some individual is fit enough, or enough time has elapsed
  return the best individual in *population*, according to FITNESS-FN

---

function REPRODUCE(x, y) **returns** an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, $c+1$, n))

2.6. Problemi di soddisfacimento di vincoli (CSP)

Utilizzano una rappresentazione fattorizzata: per ogni stato si ha una serie di variabili, ognuna delle quali ha un valore.
Un problema è risolto quando ogni variabile ha un valore che rispetta tutti i vincoli su di essa.
Utilizzano euristiche di uso generale per permettere la soluzione di problemi più complessi.
L'idea principale è quella di eliminare ampie porzioni dello spazio di ricerca tutte insieme, individuando combinazioni di variabili e valori che violano i vincoli.

2.6.1. Definizione dei problemi

- X è un insieme di variabili: $\{X_1, \dots, X_n\}$
- D è un insieme di domini, uno per ogni variabile: $\{D_1, \dots, D_n\}$
- C è un insieme di vincoli che specificano combinazioni di valori ammesse.

Ogni dominio D_i è costituito da un insieme di valori ammessi $\{v_1, \dots, v_k\}$.
Ogni vincolo C_i è costituito da una coppia <*ambito*, *rel*>, dove *ambito* è una tupla di variabili che partecipano al vincolo e *rel* è una relazione che definisce i valori che tali variabili possono assumere.

Ogni stato in un problema CSP è definito dall'**assegnamento** di valori ad alcune o tutte le variabili.
Assegnamento consistente: non viola alcun vincolo, detto anche legale.
Assegnamento completo: se a tutte le variabile è assegnato un valore.
Soluzione: è un assegnamento completo e consistente.

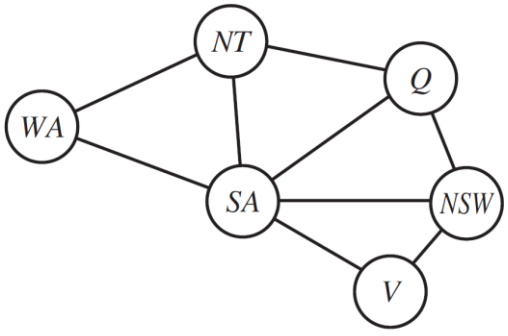
Tipi di variabili:

- Variabili discrete
  - Domini finiti: n variabili, dimensione max del dominio d di ogni variabile, numero di possibili assegnamenti O(dn).
    CSP risolvono problemi di dimensioni di varii ordini di grandezza superiori a quelli risolti dagli alg ricerca
  - Domini infiniti
- Variabili continue

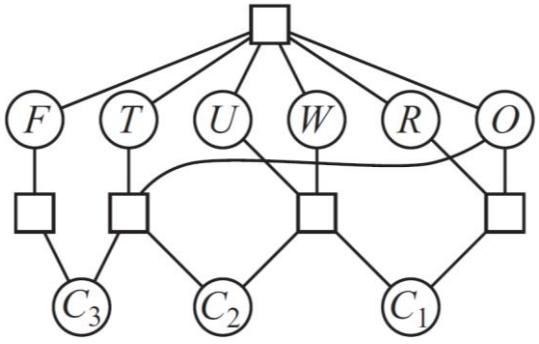
Tipi di vincoli:

- Unari: riguardano una variabile (e.g. SA ≠ green).
- Binari: riguardano coppie di variabili (e.g. SA ≠ WA).
- Globali: coinvolgono 3 o più variabili.

Grafo di vincoli binari: i nodi sono le variabili, gli archi connettono ogni coppia di variabili che partecipano a un vincolo.



Ipergrafo di vincoli: nodi normali sono le variabili, iper-nodi sono i vincoli n-ari.



Problema di ottimizzazione dei vincoli (COP): trovare la soluzione ottima.

In generale, Tasks posti nell'ambito del paradigma CSP sono computazionalmente intrattabili (NP-hard), le tecniche per risolvere computazionalmente trattabili sono:

- Tecniche di consistenza
- Ricerca
- Combinazione di entrambi

2.6.2. Tecniche di consistenza

Algoritmi che riducono il problema originale eliminando dai domini delle variabili i valori che non possono comparire in una soluzione finale.

Consistenza di nodo: una variabile è nodo-consistente se tutti i valori del suo dominio soddisfano i suoi vincoli unari.

Consistenza d'arco: una variabile è arco-consistente se ogni valore del suo dominio soddisfa i suoi vincoli binari.
Algoritmo AC-3: mantiene una coda di archi che inizialmente contiene tutti gli archi del CSP. Poi ne estrae uno arbitrario (X_i, X_j) rendendo X_i arco-consistente rispetto a X_j. Se questo lascia invariato D_i, l'algoritmo passa all'arco successivo della coda; se invece D_i cambia (riducendosi), aggiunge alla coda tutti gli archi (X_k, X_i), dove X_k è un vicino di X_j. Se D_i rimane vuoto allora il problema non ha soluzione, altrimenti l'algoritmo continua a controllare gli archi finché la coda non rimane vuota.

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

```
function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Consistenza di cammino: restringe i vincoli binari utilizzando vincoli impliciti che cono inferiti considerando triplette di variabili. Si implementa mediante l'algoritmo PC-2.

K-consistenza: un CSP è k-consistente se, per ogni insieme di k-1 variabili e ogni loro assegnamento consistente, è sempre possibile assegnare un valore consistente a ogni k-esima variabile.
La 1-consistenza è la consistenza di nodo. La 2-consistenza è la consistenza di arco. La 3-consistenza è la consistenza di cammino.
Un CSP è fortemente **k-consistente** se è anche (k-1)-consistente, (k-2)-consistente, …

Di solito si usano le 2-consistenze e più raramente le 3-consistenze per via nell'enorme utilizzo di memoria.

Vincoli globali: sono vincoli che coinvolgono molte variabili.

- Tuttediverse: si rimuove prima di tutto ogni variabile coinvolta nel vincolo che ha un dominio con un solo valore, cancellando tale valore dai domini di tutte le altre variabili. Si ripete finché ci sono variabili con domini ad un solo valore.
- Al massimo

2.6.3. Ricerca con backtracking

Molti CSP non possono essere risolti con la sola inferenza, gli algoritmi di backtracking operano su assegnamenti parziali.
È una ricerca in profondità che assegna valori ad una variabile per volta e torna indietro quando non ci sono più valori legali da assegnare.
L'algoritmo sceglie ripetutamente una variabile non assegnata e poi prova uno a uno i valori del suo dominio, cercando di trovare una soluzione. Se viene rilevata un'inconsistenza, restituisce un fallimento, e la chiamata precedente prova un altro valore.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment then
          add {var = value} to assignment
          inferences ← INFERENCE(csp, var, value)
          if inferences ≠ failure then
              add inferences to assignment
              result ← BACKTRACK(assignment, csp)
              if result ≠ failure then
                  return result
      remove {var = value} and inferences from assignment
  return failure
```

Select-Unasigned-Variable:

- **Euristica MRV:** sceglie la variabile per cui è maggiore la probabilità di arrivare presto a un fallimento.
- **Euristica di grado:** cerca di ridurre il fattore di ramificazione delle scelte future scegliendo la variabile coinvolta nel maggior numero di vincoli con le altre variabili non assegnate.

Una volta scelta la variabile, l'algoritmo deve decidere l'ordine con cui esaminare i suoi possibili valori con l'**euristica del valore meno vincolante**: predilige il valore che lascia più libertà alle variabili adiacenti sul grafo dei vincoli.

Verifica in avanti (forword checking): ogni volta che una variabile X è assegnata, il processo di verifica in avanti stabilisce la consistenza d'arco per essa: per ogni variabile non assegnata Y collegata a X da un vincolo, cancella dal dominio di Y ogni valore non consistente con quello scelto per X. Rileva molte inconsistenze ma non tutte, l'algoritmo MAC aiuta.

3. Conoscenza, ragionamento e pianificazione

3.1. Agenti basati sulla conoscenza

Formula: rappresenta un asserzione sul mondo.
Assioma: formula che è data per buona senza essere ricavata da altre formule.
Inferenza: derivazione di nuove formule a partire da quelle conosciute.
Base di conoscenza: è costituita da un insieme di formule, espresse mediante un linguaggio di rappresentazione della conoscenza. Possiede dei meccanismi per aggiungere nuove formule e per le interrogazioni: TELL() e ASK() che possono comportare un processo di inferenza.

Agente basato sulla conoscenza: può essere descritto solo a livello della conoscenza, ovvero basta specificare ciò che conosce e i suoi obiettivi.
Comunica alla base di conoscenza le sue percezioni tramite TELL() e chiede quali azioni eseguire tramite ASK(). Scelta l'azione da eseguire l'agente la registra sulla base di conoscenza tramite TELL() prima di eseguirla.
Make-Percept-Sentence(): prende una percezione e un dato istante temporale e restituisce la formula che asserisce che in quel momento l'agente ha ricevuto quella percezione.
Make-Action-Query(): prende in input un istante temporale e restituisce la formula che chiede quale azione intraprendere in quel momento.
Make-Action-Sentence(): costruisce una formula che asserisce che l'azione scelta è stata eseguita.

function KB-AGENT(*percept*) **returns** an *action*
  persistent: *KB*, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  action ← ASK(*KB*, MAKE-ACTION-QUERY(*t*))
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  t ← *t* + 1
  return *action*

Logica: fornisce strumenti indispensabili per controllare con sicurezza la rigorosità dei ragionamenti tra cui: dedurre conseguenze da certe premesse, studiare la verità o falsità di certe proposizioni data la verità o falsità di altre proposizioni, analizzare le inferenze in termini di operazioni su espressioni simboliche, costruire teorie, stabilire la consistenza e la validità di una data teoria.

Linguaggio	Scelta Ontologica	Scelta Epistemologica
Logica Proposizionale	fatti	vero/falso/sconosciuto
Logica del Primo Ordine	fatti, oggetti, relazioni	vero/falso/sconosciuto
Logica Temporale	fatti, oggetti, relazioni, tempi	vero/falso/sconosciuto
Teoria delle Probabilità	fatti	gradi di credenza $\in [0, 1]$
Logica Fuzzy	fatti con gradi di verità $\in [0, 1]$	valore conosciuto

3.2. Logica proposizionale

Sintassi e semantica

Le formule atomiche consistono di un singolo simbolo proposizionale, che rappresenta una proposizione che può essere vera o falsa.

Si possono costruire formule complesse grazie all'uso di parentesi e dei connettivi logici:

- ¬ Not
- ∧ And
- ∨ Or
- ⇒ Implicazione (if-then)
- ⇔ Bi-Implicazione (Se e solo se)

Precedenza degli operatori: ¬, ∧, ∨, ⇒, ⇔

F	G	$(\neg F)$	$(F \wedge G)$	$(F \vee G)$	$(F \rightarrow G)$	$(F \leftrightarrow G)$
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	true	false
false	false	true	false	false	true	true

Conseguenza logica: $\alpha \vDash \beta$

Tautologia: formula vera in tutte le interpretazioni. (Es. $((P \wedge (P \rightarrow Q)) \rightarrow Q)$)

Contraddizione: formula falsa in tutte le interpretazioni. (Es. $((P \rightarrow Q) \wedge P) \wedge (\neg Q)$)

Formule equivalenti $F \equiv G$: se e solo se esse hanno lo stesso valore di verità in tutte le interpretazioni.

Principali equivalenze:

F	$\equiv$	F	identità
$\neg(\neg F)$	$\equiv$	F	doppia negazione
$(G \wedge G)$	$\equiv$	G	idempotenza
$(G \vee G)$	$\equiv$	G	idempotenza
$(G \wedge TRUE)$	$\equiv$	G	legge dei neutri
$(G \wedge FALSE)$	$\equiv$	$FALSE$	legge dei neutri
$(G \vee TRUE)$	$\equiv$	$TRUE$	legge dei neutri
$(G \vee FALSE)$	$\equiv$	G	legge dei neutri
$(G \wedge \neg G)$	$\equiv$	$FALSE$	esclusione
$(G \vee \neg G)$	$\equiv$	$TRUE$	complementarietà
$((F \wedge G) \wedge H)$	$\equiv$	$((F \wedge (G \wedge H)$	associatività
$((F \vee G) \vee H)$	$\equiv$	$((F \vee (G \vee H)$	associatività
$(F \wedge G)$	$\equiv$	$(G \wedge F)$	commutatività
$(F \vee G)$	$\equiv$	$(G \vee F)$	commutatività
$(F \wedge (G \vee H))$	$\equiv$	$((F \wedge G) \vee (F \wedge H))$	distributività
$(F \vee (G \wedge H))$	$\equiv$	$((F \vee G) \wedge (F \vee H))$	distributività
$\neg(F \vee G)$	$\equiv$	$(\neg F \wedge \neg G)$	legge di De Morgan
$\neg(F \wedge G)$	$\equiv$	$(\neg F \vee \neg G)$	legge di De Morgan
$(F \vee (F \wedge G))$	$\equiv$	F	assorbimento
$(F \wedge (F \vee G))$	$\equiv$	F	assorbimento
$(F \vee (\neg F \wedge G))$	$\equiv$	$(F \vee G)$	assorbimento
$(F \wedge (\neg F \vee G))$	$\equiv$	$(F \wedge G)$	assorbimento
$(F \rightarrow G)$	$\equiv$	$(\neg F \vee G)$	eliminazione dell'implicazione
$(F \rightarrow (G \rightarrow H))$	$\equiv$	$(G \rightarrow (F \rightarrow H))$	
$(F \rightarrow (G \rightarrow H))$	$\equiv$	$((F \wedge G) \rightarrow H)$	
$(F \leftrightarrow G)$	$\equiv$	$((F \rightarrow G) \wedge (G \rightarrow F))$	doppia implicazione

G segue logicamente da un insieme di formule $F_1,\dots,F_n$: se G è vera in tutte le interpretazioni in cui $F_1,\dots,F_n$ sono vere.
Teorema di deduzione: Date le formule $F_1,\dots,F_n$ e G, G segue logicamente dall'insieme di formule $F_1,\dots,F_n$ se e so lo se la formula: $((F_1 \wedge F_2 \wedge \dots \wedge F_n) \rightarrow G)$ è una tautologia.
Oppure: … se $(F_1 \wedge F_2 \wedge \dots \wedge F_n \wedge (\neg G))$ è una contraddizione.

Teoria assiomatica: da degli assiomi si è in grado di produrre secondo determinati criteri di manipolazione sintattica stabiliti dalle regole di inferenza, nuove formule vere e sintatticamente corrette (teoremi).

Sistema assiomatico hilbertiano:

Connettivi utilizzati: ¬ e ⇒
Assiomi:
$(\mathbf{A1})\ A \rightarrow (B \rightarrow A)$
$(\mathbf{A2})\ (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
$(\mathbf{A3})\ (\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B)$
Regola di inferenza: Modus Ponens: dati A e A ⇒ B si deriva B.

Regole di inferenza per derivare una dimostrazione:

- Modus Ponens (Affermazione dell'Antecedente): dati A e A ⇒ B si deriva B: $\dfrac{\alpha \Rightarrow \beta,\ \alpha}{\beta}$
- Modus Tollens (Negazione del Conseguente): $\dfrac{\alpha \Rightarrow \beta,\ \neg\beta}{\neg\alpha}$
- Fallacia dell'Affermazione del Conseguente: $\dfrac{\alpha \Rightarrow \beta,\ \beta}{\alpha}$
- Fallacia della Negazione dell'Antecedente: $\dfrac{\alpha \Rightarrow \beta,\ \neg\alpha}{\neg\beta}$

Fallacia: è un errore logico, è un ragionamento che a prima vista può sembrare corretto ma se analizzato con attenzione presenta un errore.

Ragionamento induttivo: origine da premesse che nascono da una base osservativa, le conclusioni a cui perviene sono più ampie delle premesse e non sono certe, ma soltanto probabili.

3.3. Logica del primo ordine

È una logica più espressiva, la logica proposizionale assume che il mondo contenga solamente fatti, quella del primo ordine assume che il mondo contenga oggetti, relazioni e funzioni.

Sintassi

- Costanti: rappresentano oggetti.
- Predicati: rappresentano le relazioni. (Fratello, >,…)
- Funzioni: rappresentano funzioni. (Sqrt,…)
- Variabili (x, y, a, b,…)
- Connettivi: ¬, ∧, ∨, ⇒, ⇔
- Uguaglianza: =
- Quantificatori: ∀ ∃

Termine: *funzione(termine$_1$, …, termine$_n$)* oppure *costante* oppure *variabile*
Sentenza atomica: *predicato(termine$_1$, …, termine$_n$)* oppure *termine1=termine2*
Sentenze complesse: sentenze atomiche connesse dai connettivi.

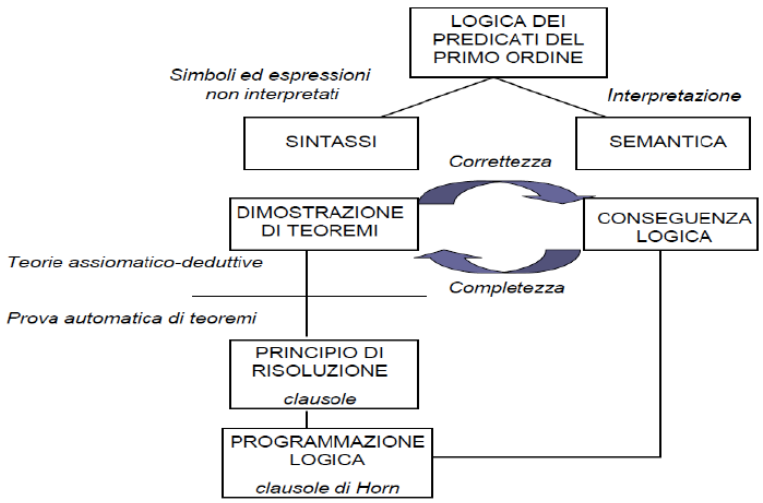
Una sentenza atomica è vera se gli oggetti riferiti sono nella relazione riferita dal predicato.

Equivalenza tra quantificatori (leggi di DeMorgan):

$$\forall x\ \neg P \equiv \neg\exists x\ P \qquad \neg(P \vee Q) \equiv \neg P \wedge \neg Q$$
$$\neg\forall x\ P \equiv \exists x\ \neg P \qquad \neg(P \wedge Q) \equiv \neg P \vee \neg Q$$
$$\forall x\ P \equiv \neg\exists x\ \neg P \qquad P \wedge Q \equiv \neg(\neg P \vee \neg Q)$$
$$\exists x\ P \equiv \neg\forall x\ \neg P \qquad P \vee Q \equiv \neg(\neg P \wedge \neg Q)$$

predicato(*termine1*)=*termine2* asserisce che l'oggetto a cui si riferisce predicato(termine1) e a quello a cui si riferisce *termine2* sono lo stesso.

Logica monotonica: se per ogni base di conoscenza arbitraria WB e un arbitraria formula Φ, l'insieme delle formule derivabili da WS è un sottoinsieme delle formule derivabili da WB∪Φ.



Le formule dei predicati del 1 ordine possono essere trasformate in clausole in logica proposizionale.
Passi per trasformare:

- **Trasformare in una FBF chiusa:** tutte le variabili che erano libere risultano quantificate universalmente. Esempio:

$$\forall x \left(p(y) \Rightarrow \neg \left(\forall y \ \left(q(x,y) \Rightarrow p(y)\right)\right)\right) \ diventa \ \forall x \ \forall y \left(p(y) \Rightarrow \neg \left(\forall y \ \left(q(x,y) \Rightarrow p(y)\right)\right)\right)$$

- **Applicazione delle equivalenze per i connettivi logici.** Es A→B è sostituito da ¬AVB

$$\forall x \ \forall y \left(\neg p(y) \vee \neg \left(\forall y \ \left(\neg q(x,y) \vee p(y)\right)\right)\right)$$

- **Applicazione della negazione ad atomi e non a formule composte** (Leggi di DeMorgan). Esempio:

$$\forall x \ \forall y \left(\neg p(y) \vee \left(\exists y \ \left(q(x,y) \vee \neg p(y)\right)\right)\right)$$

- **Cambiamento di nomi delle variabili, nel caso di conflitti**. Esempio:

$$\forall x \ \forall y \left(\neg p(y) \vee \left(\exists z \ \left(q(x,z) \vee \neg p(z)\right)\right)\right)$$

- **Spostamento dei quantificatori (∀ ∃) in testa alla formula**. Esempio:

$$\forall x \ \forall y \ \exists z \ \left(\neg p(y) \vee \left(q(x,z) \wedge \neg p(z)\right)\right)$$

- **Forma normale congiuntiva** cioè come congiunzione di disgiunzioni, con quantificatore in testa.

$$\forall x \ \forall y \ \exists z \ \left(\left(\neg p(y) \vee q(x,z)\right) \wedge \left(\neg p(y) \wedge \neg p(z)\right)\right)$$

- **Eliminazione dei quantificatori esistenziali tramite Skolemizzazione**: ogni variabile quantificata esistenzialmente viene sostituita da una funzione delle variabili quantificate universalmente che la precedono. Tale funzione è detta funzione di Skolem.
  …
- **Eliminazione dei quantificatori universali:** si ottiene una formula detta universale (tutte le sue variabili sono quantificate universalmente) in forma normale congiuntiva.
  ….

Risoluzione….

Unificazione: procedimento di manipolazione formale che stabilisce se due espressioni coincidono tramite opportune sostituzioni……..

Clausola: è una disgiunzione di letterali (cioè formule atomiche negate e non negate), in cui tutte le variabili sono quantificate universalmente in modo implicito (non compaiono quantificatori)
Clausole di Horn: hanno al più un letterale positivo.

3.4. Programmazione Logica

Linguaggio ad altissimo livello basato sulla logica dei predicati del Primo Ordine, manipola simboli e non numeri.
Programmazione dichiarativa in cui il programmatore descrive i fatti e le proprietà relative al problema da risolvere.
Un programma PROLOG è un insieme di clausole di Horn che rappresentano:

- FATTI riguardanti gli oggetti in esame e le relazioni che intercorrono A.
- REGOLE sugli oggetti e sulle relazioni (SE…ALLORA)
- GOAL (clausole senza testa), sulla base della conoscenza definita

sulle quali si vogliono fare delle interrogazioni.

Sintassi

PL1 / clause normal form	PROLOG	Description
$(\neg A_1 \vee \ldots \vee \neg A_m \vee B)$	B :- $A_1, \ldots, A_m$.	rule
$(A_1 \wedge \ldots \wedge A_m) \Rightarrow B$	B :- $A_1, \ldots, A_m$.	rule
A	A.	fact
$(\neg A_1 \vee \ldots \vee \neg A_m)$	?- $A_1, \ldots, A_m$.	query
$\neg(A_1 \wedge \ldots \wedge A_m)$	?- $A_1, \ldots, A_m$.	query

- Le clausole sono scritte "al contrario"
- ← Sostituita da **:-**
- virgole sono congiunzioni

Termini:

- **costante**: interi, floating point, atomi alfanumerici
- **variabile**: carattere maiuscolo o _ (es: Pippo, _pippo)
- **funzione** (struttura): f(t1, … tn)

Una volta inserita la base di conoscenza vengono inviate delle query al programma.
Un goal viene provato provando i singoli letterali da sinistra a destra.
Un goal atomico (ossia formato da un singolo letterale) viene provato confrontandolo e unificandolo con le teste delle clausole contenute nel programma.
Se esiste una sostituzione per cui il confronto ha successo
– se la clausola con cui unifica è un fatto, la prova termina;
– se la clausola con cui unifica è una regola, ne viene provato il Body
Se non esiste una sostituzione il goal fallisce.

3.5. Pianificazione

Problema: trovare una sequenza di azioni (piano) che raggiunge un dato goal quando eseguita a partire da un dato stato iniziale del mondo.
I goal sono usualmente specificati come una congiunzione di (sotto)goal da raggiungere.

Ci sono varie strategie per generare un piano:

- **Generative Planning:** utilizza dei principi primi legati alla conoscenza delle azioni per generare un piano, è necessario avere dei modelli formali delle azioni.
- **Case-based planning:** l'agente recupera un piano già prodotto per una soluzione simile e lo modifica per adattarlo al problema che sta affrontando.
- **Reinforcement Learning:** vengono eseguite delle azioni a caso, tenendo traccia dei risultati che si ottengono. Questi risultati vengono poi valutati per creare dei modelli di azioni da adottare in futuro.

Assunzioni tipiche:

- Il tempo è atomico, ogni azione può essere considerata indivisibile.
- Non sono ammesse azioni concorrenti anche se la azioni non hanno bisogno di essere eseguite in un determinato ordine. Viene sempre eseguita un'azione alla volta.
- Azioni deterministiche, il risultato delle azioni è completamente determinato e non c'è incertezza nel loro effetto (le azioni non possono fallire).
- L'agente è l'unica causa di cambiamento nel mondo.
- L'agente è omniscente, ha conoscenza completa dello stato del mondo (si assumene un mondo completamente osservabile, ovvero l'agente ha tutti i sensori che gli permettono di recuperare tutte le informazioni che gli interessano).
- Closed world assumption: tutte le informazioni a disposizione vengono assunte come vere, tutto quello che non si conosce è falso.

Differenze con il problem solving: nei problemi di ricerca non si entra mai nei dettagli del problema (stati, successori), mentre gli algoritmi di planning tengono in considerazione anche i dettagli del problema. Questo dovrebbe renderli più efficienti, anche se al momento non lo sono.
Tipicamente in un problema di planning, gli stati, il goal e le azioni vengono decomposte in insiemi di sentenze (usualmente espresse in FOL). Inoltre, il problem solving lavora "a stati" mentre con il planning si procede nello spazio dei piani, si parte da un piano vuoto e lo si va via via a popolare con le varie azioni.
In questo modo la ricerca procede lungo lo spazio dei piani e, ad ogni passaggio, si costruisce un piano parziale, permettendo così di creare dei sotto goal che possono essere pianificati indipendentemente, riducendo la complessità del problema di pianificazione.
Ad esempio, la funzione successore di un problema di ricerca andrebbe a generare tutti gli stati che possono essere raggiunti a partire da un dato stato mentre nella pianificazione vengono utilizzati i dettagli delle azioni per selezionare solo le azioni utili che permettono di avvicinarsi ad un goal.
Allo stesso modo, la descrizione a "black-box" del goal nasconde dei dettagli per il goal, i quali possono permettere di scomporre il goal in sotto-goal risolvibili a parte oppure possono essere utilizzati per valutare meglio la bontà di uno stato.
Quindi se il goal è composto da una serie di sotto goal tra loro indipendenti si può pianificare indipendentemente per ogni sotto-goal ottenendo così una pianificazione più semplice.

Rappresentazione: ogni stato viene rappresentato come una serie di congiunzione di fatti veri in quel determinato stato e conseguentemente un goal specifica quali fatti vengono richiesti che siano veri.
Tutte queste informazioni vengono rappresentate con sentenze logiche del primo ordine, pertanto il planning può essere visto come la combinazione della ricerca con la rappresentazione logica.

Calcolo delle situazioni: L'idea base è di rappresentare il problema di planning in FOL, inserendo nella base di conoscenza delle sentenze che modellano il mondo e che sono invariabili e delle altre sentenze che rappresentano lo stato corrente, le quali variano man mano che vengono attuate le azioni previste dal piano.
Con questa rappresentazione è possibile utilizzare l'inferenza per provare una sequenza di azioni. Così facendo un piano equivale ad una prova delle regole associate alle azioni a partire dalla situazione che caratterizza lo stato iniziale.
Questo approccio è teoricamente corretto, c'è però un problema pratico in quanto nel caso pessimo la complessità dell'inferenza è esponenziale. Inoltre, applicando la risoluzione viene trovata una prova (piano) che non è necessariamente il piano migliore. In più la risoluzione risponde con un "Si, c'è un piano ma non te lo dico".
Utilizzando la rappresentazione logica ci sono altri tre problemi tipici:

- Problema del frame: quando si passa da una situazione ad un'altra bisogna riuscire a determinare se cambia il valore di un predicato che non è stato coinvolto dalla trasformazione, cioè se la trasformazione ha causato un side-effect.
- Problema della qualifica: perché un'azione possa essere applicata devono essere soddisfatte delle precondizioni, ma come possono essere rappresentate? In alcuni casi risulta complesso riuscire ad esplicitare tutte le condizioni necessarie e sufficienti per poter eseguire un'azione. Per specificare in modo corretto tutto ciò può essere necessario dover andare ad inserire un elevato numero di clausole nella KB.
- Problema della ramificazione: versione duale della qualifica, descrivere in modo dettagliato gli esiti di un'azione può richiedere un elevato numero di clausole.

Quindi è meglio utilizzare un linguaggio ristretto con un algoritmo specializzato (planner) piuttosto che una soluzione generale, ottenendo così una soluzione più efficiente.

STRIPS: Approccio classico usato negli anni 70 nel quale gli stati vengono rappresentati come una congiunzione di letterali ground e i goal sono congiunzioni di letterali o variabili quantificate esistenzialmente.
Non c'è bisogno di specificare completamente lo stato, facendo ciò significa che quello non specificato viene assunto falso (closed world).
Vengono così rappresentati molti casi con poca memoria e spesso vengono rappresentati solo i cambiamenti dello stato piuttosto che l'intera situazione, limitando il problema del frame.
Al contrario di un dimostratore di teoremi, non viene provato se il goal è vero, ma viene cercata una sequenza di azioni che lo raggiunge.
In STRIPS le azioni vengono rappresentate come operatori, ognuno dei quali è composto da 3 componenti:

- Descrizione dell'azione
- Precondizioni: espresse come congiunzione di letterali positivi
- Effetti: congiunzione di letterali positivi o negativi che descrivono come la situazione cambia quando si applica l'operatore.

```
Op[
    Action: Go(there),
```

```
    Precondizioni: At(here) ∧ Path(here,there),
    Effetti: At(there) ∧ ¬At(here)
]
```

Tutte le variabili sono quantificate universalmente e tutte le variabili di situazione sono implicite: le precondizioni devono essere vere nello stato precedente dell'applicazione dell'operatore, gli effetti sono veri immediatamente dopo.

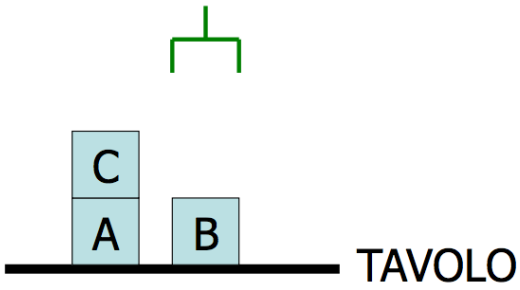
3.5.1. Mondo dei blocchi

Il mondo dei blocchi è un micro-mondo che consiste in un tavolo, un insieme di blocchi e un manipolatore robotico.
Alcuni vincoli del dominio:

- Un solo blocco può essere immediatamente sopra un altro
- Un qualsiasi numero di blocchi sul tavolo
- Il manipolatore può mantenere un solo blocco

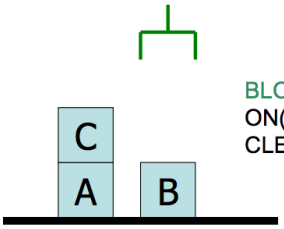
Una possibile rappresentazione è data da

```
on(a,tavolo)
on(c,tavolo)
on(b,a)
handempty
clear(b)
clear(c)
```



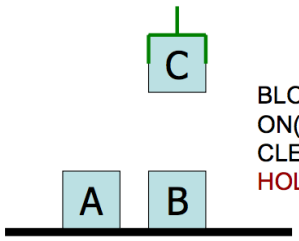
Un goal è composto da una serie di proposizioni, ogni stato che soddisfa le proposizioni goal può essere considerato come stato goal.

Esempio:



BLOCK(A), BLOCK(B), BLOCK(C),
ON(A,TAVOLO), ON(B,TAVOLO), ON(C,A),
CLEAR(B), CLEAR(C), HANDEMPTY

Unstack(C,A)
• P = HANDEMPTY, BLOCK(C), BLOCK(A),
CLEAR(C), ON(C,A)
• E = ¬HANDEMPTY, ¬CLEAR(C), HOLDING(C),
¬ ON(C,A), CLEAR(A)



BLOCK(A), BLOCK(B), BLOCK(C),
ON(A,TAVOLO), ON(B,TAVOLO), ~~ON(C,A)~~,
CLEAR(B), ~~CLEAR(C)~~, ~~HANDEMPTY~~,
HOLDING(C), CLEAR(A)

Unstack(C,A)
• P = HANDEMPTY, BLOCK(C), BLOCK(A),
CLEAR(C), ON(C,A)
• E = ¬HANDEMPTY, ¬CLEAR(C), HOLDING(C),
¬ ON(C,A), CLEAR(A)

Generazione di piani: La pianificazione può essere fatta a partire dallo stato iniziale o da quello goal.

Forward Planning: L'idea è quella di partire dallo stato iniziale e provare ad applicare tutte le possibili azioni fino a che non si trova una sequenza di azioni che porta ad uno stato goal.
Equivale quindi ad una ricerca nello spazio degli stati, dove per ogni stato vengono generati tanti successori quante sono le azioni applicabili, in un modo analogo alla ricerca breadth-first, senza tener conto della rilevanza o meno delle azioni. Un'azione è rilevante se un suo effetto combacia con la posizione del goal.
C'è il solito problema che se ci sono molte azioni applicabili ad uno stato si ottiene un fattore di branching enorme.

Backward Planning (Chaining): Si parte dalla specifica del goal (non dallo stato) e si cercano delle azioni rilevanti che soddisfano parte del goal, vengono così lasciate "aperte" le precondizioni delle azioni scelte e diventa necessario andare a soddisfare queste precondizioni con altre azioni. Si ripete questo procedimento finché non si ottengono delle precondizioni che vengono soddisfatte dallo stato iniziale.
Con questa strategia il fattore di branching è limitato e la ricerca avviene nello spazio dei goal, questo perché si considera la rappresentazione del goal e non degli stati.
Trattandosi di una ricerca in profondità c'è il rischio di ciclare, è quindi necessario inserire degli opportuni controlli. Infine, deve essere possibile effettuare il back-tracking delle scelte nel caso non si arrivi ad una soluzione.

Pianificazione in STRIPS
Vengono utilizzate due strutture dati:

- Lista di stati: contiene tutti i predicati che sono correntemente veri
- Pila di goal: una pila di goal da risolvere, con il goal corrente in testa alla pila.

Se il goal corrente non è soddisfatto dallo stato presente, STRIPS esamina gli effetti positivi dei vari operatori, e inserisce l'opertatore e la lista delle precondizioni dell'operatore sulla pila (sotto-goal). Quando il goal corrente è soddisfatto lo rimuove dalla pila.
Quando un operatore è in testa alla pila, registra l'applicazione dell'operatore sulla sequenza del piano e usa gli effetti per aggiornare lo stato corrente.
Importante: quando in cima alla pila c'è un goal composto, STRIPS lo scompone in tanti sotto-goal, li ordina e poi li inserisce nella pila. Il goal composto non viene tolto dalla pila, ma serve per valutare se effettivamente lo stato corrente lo soddisfa.

```
Strips(initial-state, goals)
    state = initial-state, plan = [], stack = []
    push goals on stack
    repeat until stack is empty
        if top of stack is goal and matcher state then
            pop stack
        else if top of stack is a conjunctive-goal g then
            select an ordering for the subgoal g
            push them on the stack
        else if on top of stack is a simple goal then
            choose an operator o whose add-list matcher goal
            replace goal sg whit operator o
            push the preconditions of o on the stack
        else if top of stack is an operator o then
            state = apply(o, state)
            plan = [plan:o]
```

3.5.2. Partial order planning

Un planner lineare costruisce un piano come una sequenza totalmente ordinata di passi, mentre un planner non lineare (POP), costruisce un piano come un insieme di passi con alcuni vincoli temporali. Vengono utilizzati dei vincoli del tipo S1 < S2 per specificare che S1 deve essere eseguito prima di S2.
Il raffinamento avviene quindi aggiungendo dei nuovi passi al piano oppure aggiungendo dei vincoli tra i passi già presenti nel piano.
La soluzione così ottenuta rappresenta un ordinamento parziale che può essere sempre convertito in un piano totalmente ordinando utilizzando un ordinamento topologico tra i vari passi.
I planner non lineari incorporano il principio del minimo impegno, cioè scelgono solo quelle azioni, ordinamenti e assegnamenti di variabili che sono assolutamente necessari, rimandando le altre scelte al futuro. Non vengono inoltre prese decisioni premature su aspetti non rilevanti per raggiungere il goal.
Confrontando i due approcci:

- Un planner lineare sceglie sempre di aggiungere un passo in un punto preciso della sequenza
- Un planner non-lineare sceglie di aggiungere un passo ed eventualmente qualche vincolo temporale fra passi

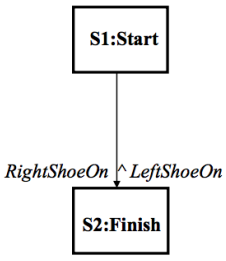
Struttura

- Insieme di passi {S1,S2, ...}, ognuno dei quali ha la descrizione di un operatore, precondizioni e post condizioni.
- Un insieme di link causali {... (Si,C,Sj) ...}, che specificano che uno dei propositi del passo Si è di raggiungere la condizione C del passo Sj.
- Un insieme di vincoli di ordinamento {... Si < Sj ...} che specificano che il passo Si deve venire eseguito prima di Sj.

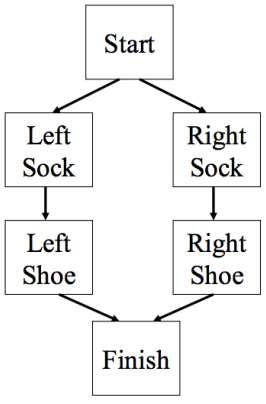
Un piano non lineare è completo se e solo se:

- Ogni passo (2) e (3) si trova nell'insieme (1).
- Se Sj ha un prerequisito C, allora esiste un link causale in (2) con la forma (Si, C, Sj).
- Se (Si, C, Sj) è in (2) e il passo Sk è in (1) e Sk "minaccia" (Si, C, Sj), cioè rende falso C, allora (3) contiene Sk<Si o Sk>Sj.

Ogni piano inizia allo stesso modo con un passo S1:Start e un S2:Finish.



Passi: {S1:[Op(Action:Start)],
S2:[Op(Action:Finish,
Pre: RightShoeOn^LeftShoeOn)]}
Link: {}
Ordinamenti: {S1<S2}



Vincoli e euristiche
Per popolare un piano non lineare si torna a fare una ricerca nello spazio dei piani.
Una strategia di ricerca è quella greedy che aggiunge solo passi che soddisfano una precondizione correntemente non soddisfatta. La filosofia del minimo impegno prevede poi di non ordinare i passi a meno che non sia strettamente necessario.
Bisogna poi tenere conto dei link causali (Si, C, Sj) che proteggono una condizione C, quindi non deve essere mai aggiunto un passo intermedio Sk tra Si e Sj che invalida C. Se un'azione parallela minaccia (threatens) C, cioè ha l'effetto di negare C (clobbering) è necessario risolvere la minaccia aggiungendo dei vincoli temporali. Si parla di demotion se viene posto Sk < Si oppure di promotion se Sk > Sj.

Proprietà di POP
L'algoritmo non è deterministico, se si verifica un fallimento si fa backtracking sui punti di scelta (choice point), questi possono essere la scelta di passo per raggiungere un sotto-obiettivo o la scelta di demotion o promotion in caso di minaccia.
POP è corretto, completo e sistematico (non ci sono ripetizioni), inoltre può essere esteso rappresentando le azioni nella logica del primo ordine.
L'algoritmo risulta efficiente se utilizza delle euristiche derivate dalla descrizione del problema, altrimenti c'è la solita esplosione combinatoria.
C'è quindi la necessità di avere delle buone euristiche e non è semplice derivare delle euristiche ammissibili.
Uno strumento utile per ottenere queste euristiche è il grado di planning, il quale raccoglie informazioni su quali piani sono impossibili non prendendo in considerazione le minacce e non considerano il "consumo" dei letterali che chiudono le precondizioni (il consumo di un letterale si ha quando l'esecuzione di un'azione cambia il valore del letterale).
Questi grafi permettono quindi di sapere se non c'è soluzione ad un problema di pianificazione.

POP con variabili non istanziate
Dal momento che le azioni vengono rappresentate con la logica del primo ordine, può capitare che la scelta di un'azione introduca delle variabili che non vengono assegnate.

Ad esempio per il mondo dei blocchi potrebbe essere definita l'azione MoveB(Z,X,Y) che sposta il blocco Z da sopra il blocco X a sopra il blocco Y.
Il pianificatore può scegliere di usare MoveB per soddisfare la precondizione on(a,b), che è uguale a on(Z,Y)θ con θ = {Z/a, Y/b}. Applicando la stessa sostituzione a MoveB si ottiene: MoveB(Z,X,Y)θ = MoveB(a,X,b), con X che rimane non istanziata.
Il fatto che alcune variabili possano rimanere non instanziate, rende necessari i vincoli del tipo var ≠ const e var ≠ var, perché le variabili non instanziate potrebbero andare a minacciare una condizione raggiunta da un'altra azione.
Ad esempio, se nel piano c'è l'azione MoveB(a,X,b) ed è necessaria per raggiungere il goal on(a,b), un'altra azione che ha come effetto ¬on(a,Q) può minacciare la condizione on(a,b) solo se viene utilizzata la sostituzione θ = {Q/b}. Deve essere quindi possibile porre dei vincoli dei tipo Q ≠ b.

3.5.3. Grafi di planning

Viene costruito a livelli:

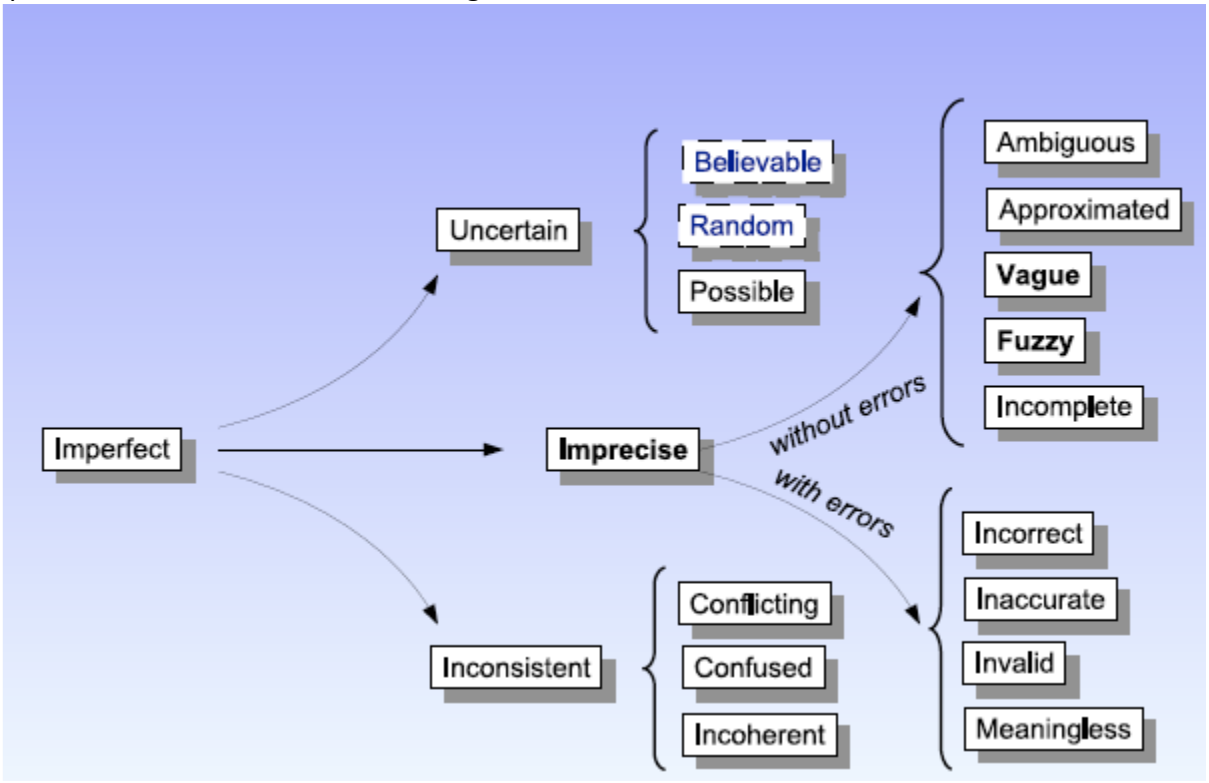
- Il primo livello contiene tutti i letterali dello stato iniziale
- I successivi livelli sono ottenuti dalla applicazione ripetuta delle azioni che hanno i prerequisiti soddisfatti
- I letterali di un livello sono riportati al livello successivo (persistance actions)

Il procedimento termina quando non è più possibile aggiungere nuovi letterali.

Per ottenere un risultato in modo efficiente non vengono considerate le variabili e non devono esserci troppi oggetti. Un'altra ipotesi semplificativa è che le azioni non consumano i letterali.

4. Ragionamento in presenza di incertezza

Imprecisione e inconsistenza: sono proprietà legate al contenuto dello statement: più di un mondo o nessun mondo è compatibile con l'informazione a disposizione.
Incertezza: carenza di informazione sul mondo, riguarda la relazione tra agente e mondo, è una proprietà dello stato di credenza dell'agente.



Informazione imprecisa: non sufficiente per rispondere a domande di interesse circa un determinato problema perché l'informazione è mancante.
Informazione incerta: Gli agenti non hanno quasi mai accesso a tutta l'informazione necessaria, devono essere in grado di agire in condizioni di incertezza
Ragionare non sul possibile ma sul probabile.

Grado di credenza:
Strumento principale per gestire i gradi di credenza è la Teoria della probabilità: degree compreso tra 0 e 1
assegnare un grado di belief a una sentenza (e.g. probability = 0) non significa che la sentenza è falsa.
Grado di verità è il soggetto della fuzzy logic: grado di verità [0,1]

Decidere nell'incertezza:
Teoria dell'utilità: usata per rappresentare e inferire preferenze: ogni stato ha un grado di utilità per l'agente che preferirà stati di utilità maggiore.
Teoria delle decisioni = teoria della probabilità + teoria dell'utilità

4.1. Sistemi e set fuzzy

Logica fuzzy: è un'estensione della logica booleana in cui si può attribuire a ciascuna proposizione un grado di verità diverso da 0 e 1 e compreso tra di loro.
Con grado di verità o valore di appartenenza si intende quanto è vera una proprietà: questa può essere, oltre che vera (= a valore 1) o falsa (= a valore 0) come nella logica classica, anche parzialmente vera e parzialmente falsa.

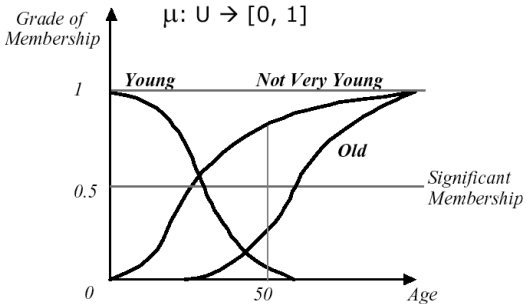
Applicazioni: Vengono usati nei sistemi di controllo, nei database, supporto alle decisioni, Sistemi di previsioni basati su serie storiche.
Rispetto ad altri "approssimatori universali" offrono il vantaggio esclusivo di offrire una forma naturale di rappresentazione della conoscenza empirica degli esperti.
Sono quindi "agevoli" da usare e decifrabili nei comportamenti, Semplificano l'interazione con l'utente, Basso costo, Robusti, Ampia gamma di applicabilità.

Proposizioni sono espresse come: **p: X is F**
dove:

- X è una variabile linguistica che prende il valore x nell'universo U
- F è un attributo che denota un fuzzy set definito su U

Esempio: *p: Temperature is HIGH*
$\mu_F(x)$ è interpretato come il valore di verità della proposizione p T(p)= $\mu_F(x)$ quindi, il valore di verità di p è il fuzzy set definito su [0,1]

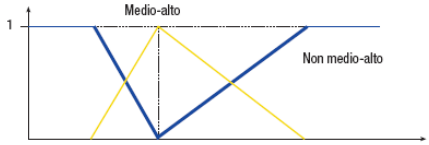
Crisp set C: $\mu_C: D \rightarrow \{0,1\}$ $\mu_C(x)$ è $booleana$
Fuzzy set F: $\mu_F: D \rightarrow [0, \dots, 1]$ $\mu_F(x)$ $ha\ valori\ reali$
$core(F) = \{x | \mu_F(x) = 1\}$
$support(F) = \{x | \mu_F(x) > 0\}$



Funzione di appartenenza: descrive per ciascun elemento dell'universo del discorso il suo grado di appartenenza ad un insieme.

Operazioni sui fuzzy set:

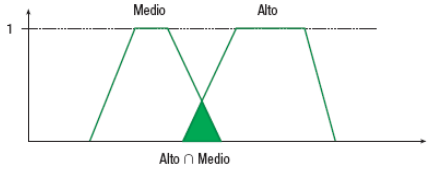
- AND = Intersezione: $\mu_{A \cap B}(x) = min(\mu_A(x), \mu_B(x))$
- OR = Unione: $\mu_{A \cup B}(x) = max(\mu_A(x), \mu_B(x))$
- NOT = Complemento: $\mu_{\tilde{A}}(x) = 1 - \mu_A(x)$

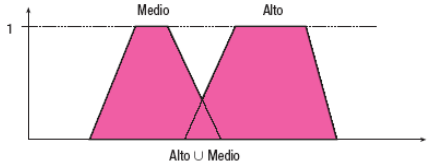


Relazioni sui fuzzy set:

- Equivalenza: $A \equiv B \iff \mu_A(x) = \mu_b(x)\ \forall x \in D$
- Inclusione: $A \subset B \iff \mu_A(x) < \mu_b(x)\ \forall x \in D$

Insieme vuoto: $\forall x\ \mu_{\emptyset}(x) = 0$
Appartenenza booleana all'universo: $\forall x\ \mu_D(x) = 1$
Le leggi di DeMorgan rimangono ma: $A \cup \tilde{A} \neq D\ e\ A \cap \tilde{A} \neq \emptyset$



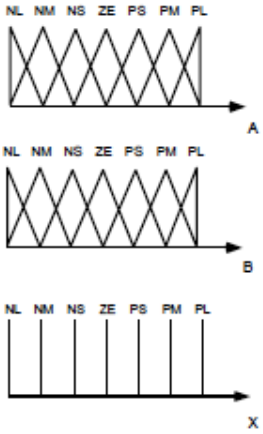


4.1.1. Regole Fuzzy

If x **is** A **then** y **is** B con x,y variabili linguistiche e A e B valori linguistici determinati dai fuzzy sets.

- **Fuzzificazione**: viene determinato il grado con cui ciascuna variabile di input appartiene ad ognuno degli insiemi Fuzzy definiti, per mezzo delle relative funzioni di membership
- **Applicazione degli operatori fuzzy:**
  - Se la premessa è composta dalla combinazione di più proposizioni si calcola il valore dell'antecedente (and)
  - Implicazione: si trasmette il valore al conseguente eventualmente tenendo conto del peso della regola
  - Composizione: gli output delle regole vengono aggregati (max)
- **Defuzzificazione**: si torna ai crisp values

Esempio:

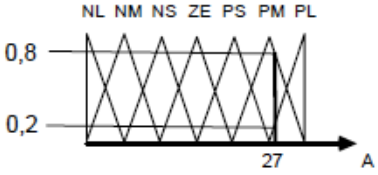


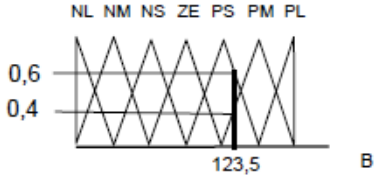
R1: if A is PL and B is PS Then X is PM

R2: if A is PM and B is PS Then X is PS

R3: if A is PL and B is PM Then X is PM

- **Fuzzyficazione**:





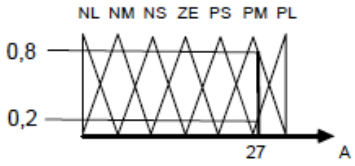
A = 27 Input facts B = 123,5

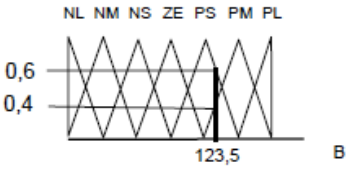
R1: IF (A is PL) (B is PS) THEN (X is PM)
0,2 0,6

R2: IF (A is PM) (B is PS) THEN (X is PS)
0,8 0,6

R3: IF (A is PM) (B is PM) THEN (X is PM)
0,8 0,4

Combinazione dei valori (And e Or) ed eventuale combinazione col peso della regola:





R1: if (A is PL) (B is PS) Then (X is PM)
0,2 0,6 1
0,2
0,2

We use here the **min** operator to combine the degree of matching of the antecedent with the rule weight (detachment)

R2: if (A is PM) (B is PS) Then (X is PS)
0,8 0,6 0,9
0,6
0,6

R3: if (A is PM) (B is PM) Then (X is PM)
0,8 0,4 1
0,4
0,4

Aggregazione dei conseguenti:

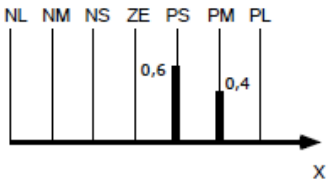
R1: if (A is PL) (B is PS) Then (X is PM)
0,2

R2: if (A is PM) (B is PS) Then (X is PS)
0,6

R3: if (A is PL) (B is PM) Then (X is PM)
0,4

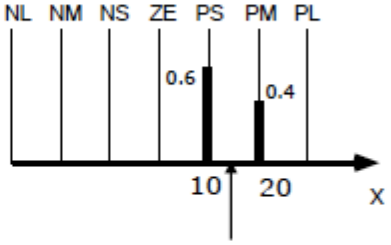
We use here the **max** operator to combine the weights of the same consequents (aggregation)

X is PM with weight 0,4 ← **max**
X is PS with weight 0,6



- **Defuzzificazione**:

X is PM with weight 0.4
X is PS with weight 0.6



(10*0.6 + 20*0.4)/(0.6 + 0.4) = **14**

4.2. Logica proposizionale probabilistica

Linguaggio formale per rappresentare e ragionare sulla conoscenza incerta. Versione di TP considerata usa una estensione della logica proposizionale per rappresentare le formule. Gradi di credenza sono applicati a delle proposizioni. Elemento base del linguaggio è la variabile aleatoria.

Variabili aleatorie: Booleane, discrete o continue.
Valori del dominio devono essere mutuamente esclusivi ed esaustivi.
Evento atomico: è una specifica completa dello stato del mondo di cui l'agente è incerto.
Probabilità a Priori o non condizionata: associata a una proposizione è il grado di credenza della proposizione in assenza di ogni altra informazione. Es. P(tempo=nuvoloso)=0.2
La prob incondizionata può essere usata solo quando non ci sono altre informazioni
Distribuzione di probabilità congiunta: per un insieme di variabili aleatorie fornisce la prob di ogni evento atomico. Rappresenta la descrizione dell'incertezza sul mondo. Se ci sono due variabili la descrizione è completa. Ogni domanda che concerne un dominio trova risposta nella distribuzione congiunta completa che fornisce la probabilità di ogni evento atomico. Esempio:

Intelligence =	low	high	
grade A	0.07	0.18	0.25
grade B	0.28	0.09	0.37
grade C	0.35	0.03	0.38
	0.7	0.3	1

Probabilità a posteriori o condizionale: rappresenta conoscenza a priori mentre la probabilità condizionale rappresenta una distribuzione + informata.

Regola di BAYES: $P(a \mid b) = P(a \wedge b) / P(b)$ if $P(b) \neq 0$
Regola Del prodotto: $P(a \wedge b) = P(a \mid b)\, P(b)$ affinché a e b siano veri è necessario che b sia vero e che lo sia anche a dato b
$P(b \mid a) = P(a \mid b)\, P(b) / P(a)$

La probabilità di ogni proposizione semplice o complessa può essere calcolata identificando gli eventi atomici in cui è vera e sommare la loro probabilità. Questo perché la probabilità di una proposizione è data dalla somma delle probabilità degli eventi atomici in cui è vera.

4.3. Reti Bayessiane

Sono una notazione grafica per asserzioni condizionalmente indipendenti per specifiche compatte di distribuzioni condizionali complete.

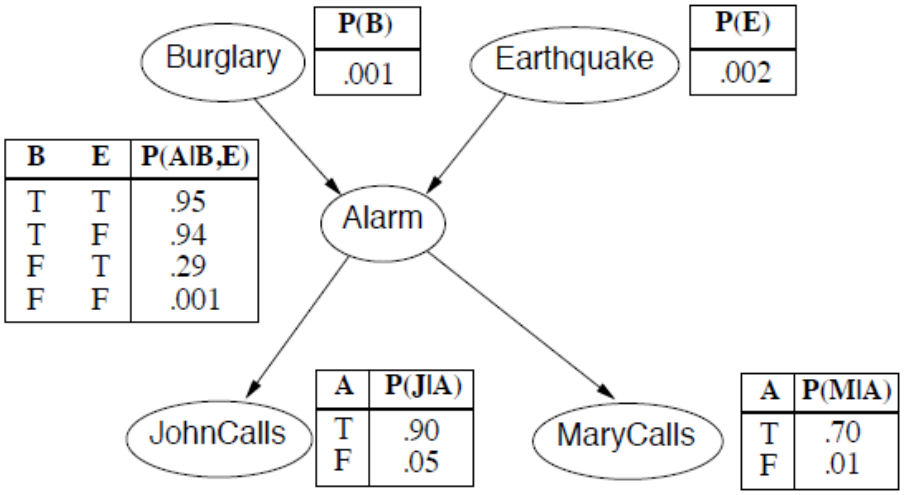
4.3.1. Sintassi

- Un insieme di nodi, uno per variabile
- Un insieme di archi diretti
- Un grafo diretto aciclico.
- Per ogni nodo si ha una distribuzione condizionale dati i suoi genitori: $P\left(X_i \middle| Parents(X_i)\right)$

Nel caso più semplice, la distribuzione condizionale è rappresentata come una **tabella della probabilità condizionale CPT** data la distribuzione su Xi per ogni combinazione di valori assunti dai genitori.

Esempio:

- Problema: sono al lavoro, il mio vicino di casa John mi chiama per dirmi che il mio allarme suona, ma invece il vicino Mary non mi chiama. Avvolte l'allarme scatta per piccoli terremoti. È un ladro?
- Variabili: Rapina, Terremoto, Allarme, JohnChiama, MaryChiama
- Topologia:



Compattezza: le reti bayessiane hanno una rappresentazione più compatta delle distribuzioni condizionali complete.
Una CPT per variabili booleane Xi con k genitori ha 2^k righe.
Ogni riga richiede un numero p per Xi=vero (Xi=falso: 1-p)
Se ogni variabile non ha più di k genitori, la rete completa richiede $O(n*2^k)$ numeri.
Distribuzioni condizionali complete hanno $O(2^n)$

4.3.2. Semantica

La distribuzione condizionale completa è definita come il prodotto delle distribuzioni condizionali locali: $P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P\left(X_i \middle| Parents(X_i)\right)$

Sull'esempio di prima: ***P****(j ∧ m ∧ a ∧ ¬b ∧ ¬e)* = ***P*** *(j | a)* ***P*** *(m | a)* ***P*** *(a | ¬b, ¬e)* ***P*** *(¬b)* ***P*** *(¬e)* = 0.90 × 0.70 × 0.001 × 0.999 × 0.998 = 0.000628

Costruzione di reti bayessiane:

1. Scegliere un ordinamento di variabili $X_1, \ldots, X_n$
2. For $i = 1$ to n
   aggiungi X_i alla rete
   seleziona genitori da $X_1, \ldots, X_{i-1}$ tali che
   $\mathbf{P}(X_i|Parents(X_i)) = \mathbf{P}(X_i|X_1, \ldots, X_{i-1})$

Questa scelta di genitori garantisce la semantica globale:

$$\begin{aligned}\mathbf{P}(X_1, \ldots, X_n) &= \prod_{i=1}^{n} \mathbf{P}(X_i|X_1, \ldots, X_{i-1}) \quad \text{(chain rule)}\\ &= \prod_{i=1}^{n} \mathbf{P}(X_i|Parents(X_i)) \quad \text{(per costruzione)}\end{aligned}$$

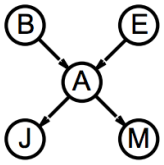
La scelta dell'ordinamento influenza notevolmente la dimensione del grafo. Ordinare le variabili mettendo prima le cause e poi l'effetto porta ad avere una rete più compatta rispetto all'ordinamento inverso.

4.3.3. Inferenza esatta tramite enumerazione

Query semplice sulla rete dell'allarme:

$$\begin{aligned}&\mathbf{P}(B|j,m)\\ &= \mathbf{P}(B,j,m)/P(j,m)\\ &= \alpha\mathbf{P}(B,j,m)\\ &= \alpha\Sigma_e\Sigma_a\mathbf{P}(B,e,a,j,m)\end{aligned}$$

inferenza tramite enumerazione



Riscrittura di entrate della distribuzione congiunta usando il prodotto di entrate di CPT:

$$\begin{aligned}&\mathbf{P}(B|j,m)\\ &= \alpha\Sigma_e\Sigma_a\mathbf{P}(B)P(e)\mathbf{P}(a|B,e)P(j|a)P(m|a)\\ &= \alpha\mathbf{P}(B)\Sigma_e P(e)\Sigma_a\mathbf{P}(a|B,e)P(j|a)P(m|a)\end{aligned}$$

Enumerazione ricorsiva depth-first: $O(n)$ in spazio, $O(d^n)$ in tempo

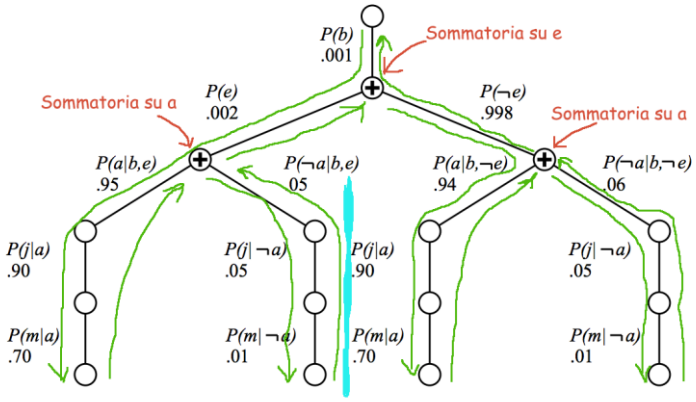
Algoritmo di enumerazione:

function ENUMERATION-ASK(X, **e**, bn) **returns** a distribution over X
  inputs: X, the query variable
    e, observed values for variables **E**
    bn, a Bayesian network with variables $\{X\} \cup \mathbf{E} \cup \mathbf{Y}$
  $\mathbf{Q}(X) \leftarrow$ a distribution over X, initially empty
  for each value x_i of X **do**
    extend **e** with value x_i for X
    $\mathbf{Q}(x_i) \leftarrow$ ENUMERATE-ALL(VARS[bn], **e**)
  return NORMALIZE($\mathbf{Q}(X)$)

---

function ENUMERATE-ALL($vars$, **e**) **returns** a real number
  if EMPTY?($vars$) **then return** 1.0
  $Y \leftarrow$ FIRST($vars$)
  if Y has value y in **e**
    then return $P(y \mid Pa(Y)) \times$ ENUMERATE-ALL(REST($vars$), **e**)
    else return $\Sigma_y\ P(y \mid Pa(Y)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}_y$)
      where $\mathbf{e}_y$ is **e** extended with $Y = y$

L'algoritmo esegue il calcolo delle varie sommatorie, ricorsivamente, in profondità e a partire da quella più a sinistra. Da notare che prima di ritornare la risposta alla query, questa deve essere normalizzata.



Nell'albero i pallini rappresentano le operazioni e gli archi rappresentano i valori

4.3.4. Inferenza esatta tramite eliminazione di variabile

L'algoritmo di inferenza tramite enumerazione può essere migliorato eliminando calcoli ripetitivi.

Eliminazione di variabile: effettuare le somme da destra a sinista,
memorizzare i risultati intermedi (fattori) per evitare di ricalcolarli

$$\begin{aligned}
&\mathbf{P}(B|j,m) \\
&= \alpha \underbrace{\mathbf{P}(B)}_{B} \Sigma_e \underbrace{P(e)}_{E} \Sigma_a \underbrace{\mathbf{P}(a|B,e)}_{A} \underbrace{P(j|a)}_{J} \underbrace{P(m|a)}_{M} \\
&= \alpha \mathbf{P}(B) \Sigma_e P(e) \Sigma_a \mathbf{P}(a|B,e) P(j|a) f_M(a) \\
&= \alpha \mathbf{P}(B) \Sigma_e P(e) \Sigma_a \mathbf{P}(a|B,e) f_J(a) f_M(a) \\
&= \alpha \mathbf{P}(B) \Sigma_e P(e) \Sigma_a f_A(a,b,e) f_J(a) f_M(a) \\
&= \alpha \mathbf{P}(B) \Sigma_e P(e) f_{\bar{A}JM}(b,e) \text{ (elimina } A) \\
&= \alpha \mathbf{P}(B) f_{\bar{E}\bar{A}JM}(b) \text{ (elimina } E) \\
&= \alpha f_B(b) \times f_{\bar{E}\bar{A}JM}(b)
\end{aligned}$$

Eliminare una variabile da un prodotto di fattori:

1. muovere i fattori costanti al di fuori della somma
2. aggiungere le sottomatrici al prodotto "pointwise" dei fattori rimanenti

$$\Sigma_x f_1 \times \cdots \times f_k = f_1 \times \cdots \times f_i \Sigma_x f_{i+1} \times \cdots \times f_k = f_1 \times \cdots \times f_i \times f_{\bar{X}}$$

assumendo che $f_1, \ldots, f_i$ non dipendano da X

Prodotto pointwise di fattori f_1 e f_2:

$$\begin{aligned}
&f_1(x_1, \ldots, x_j, y_1, \ldots, y_k) \times f_2(y_1, \ldots, y_k, z_1, \ldots, z_l) \\
&\quad = f(x_1, \ldots, x_j, y_1, \ldots, y_k, z_1, \ldots, z_l)
\end{aligned}$$

P.e., $f_1(a,b) \times f_2(b,c) = f(a,b,c)$

function ELIMINATION-ASK(X, **e**, bn) **returns** a distribution over X
  inputs: X, the query variable
    e, evidence specified as an event
    bn, a belief network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$
  $factors \leftarrow [\,]$; $vars \leftarrow$ REVERSE(VARS[bn])
  for each var **in** $vars$ **do**
    $factors \leftarrow [\text{MAKE-FACTOR}(var, \mathbf{e}) | factors]$
    if var is a hidden variable **then** $factors \leftarrow$ SUM-OUT($var, factors$)
  return NORMALIZE(POINTWISE-PRODUCT($factors$))

Da notare che l'algoritmo inizia rovesciando le variabili e con i fattori vuoti.
Quando viene trovata una variabile nascosta si esegue il Sum-Out della variabile, il quale per ogni valore possibile delle altre variabili, fissa un valore ed esegue la sommatoria dei valori al variare della variabile da sommare.
Complessità dell'inferenza esatta: nel caso pessimo, il tempo di esecuzione è esponenziale.
e una rete è singolarmente connessa, ovvero ogni coppia di nodi è connessa da al più un cammino (non diretto) il costo in tempo per l'algoritmo di eliminazione di variabile è $O(d^k n)$.

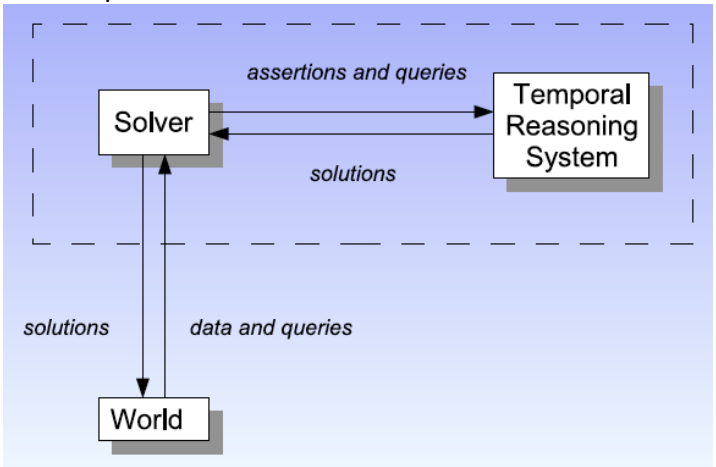
4.4. Ragionamento temporale

Applicazioni nel mondo reale:

- Programmazione della produzione: schedulazioni su catene di montaggio, tempi di produzione, sincronizzazione produzione di componenti
- Pianificazione: quale relazione temporale esiste tra le Azioni A e B?
- Gestione del personale: calendari, turni, appuntamenti, riunioni
- Logistica: arrivi e partenze dal magazzino, consegne, tempi di evasione degli ordini
- Gestione Orario dei treni, gestione binari
- Gestione Orario dei voli, tempi di assegnamento gates e piste degli aeroporti
- Data-base: quale è l'ordine cronologico di un insieme di oggetti, di informazioni?
- Settore Legale: Brevetti, analisi e verifica di alibi, calcolo dei danni
- quale malattia presenta questa sequenza temporali di sintomi?
- aspetto temporale della causalità
- Database di cartelle cliniche e informazioni di anamnesi

Il tempo permette di descrivere la causalità in quanto questa relazione traduce il fatto che gli eventi effetto si produrranno nel futuro degli eventi causa

Ragionamento temporale dal punto di vista AI



Un ragionatore temporale dovrebbe essere in grado di rispondere a queries di vario tipo, ad esempio:

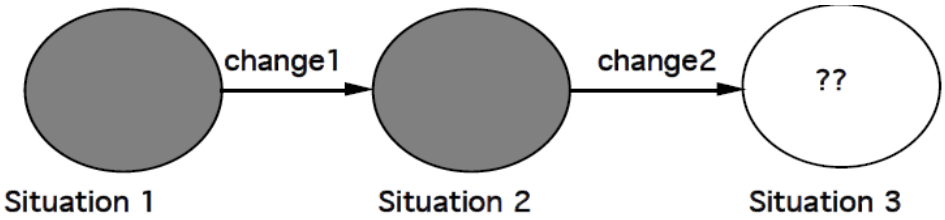
- È coerente l'informazione? Quale è uno scenario consistente?
- Può l'evento Xi accadere dopo Xj?
- Deve l'evento Xi accadere t istanti prima di Xj?
- In quali istanti t può verificarsi l'evento Xi?
- Se l'evento Xi accade in t1, in quali istanti t2 può accadereXj?

Logica temporale

3 approcci principali per introdurre il tempo in logica:

- il tempo come argomento in un predicato del primo ordine
- le logiche modali temporali
  FΦ ≡ Φ è vero a qualche tempo futuro
  PΦ ≡ Φ è vero a qualche tempo passato
- le logiche temporali "reified": associare una asserzione atemporale con una entità temporale (istanti, intervalli, etc). HOLDS(<forma atemporale>, <riferimento temporale>)

4.4.1. Calcolo situazionale



Ogni possibile stato dell'universo (snapshot) è rappresentato da una situazione
Le situazioni sono collegate e sono definite dalla sequenza di azioni che le determinano a partire da un certo stato iniziale.

Linguaggio logico progettato per rappresentare mondi che cambiano dinamicamente sotto l'effetto di azioni eseguite dall'agente.
È una formalizzazione con il calcolo dei predicati degli stati, delle azioni e degli effetti delle azioni sugli stati
Ipotesi: il mondo evolve in una sequenza discreta e lineare di "situazioni"
La situazione iniziale è rappresentata da una costante S0, le altre sono rappresentate da termini della forma `result(a, s)` oppure `do(a, s)`, dove `do` è un simbolo funzionale particolare, che si applica ad azioni e situazioni. Esempi: `do(forward, S0), do(turn(right), do(forward, S0))`

Fluenti: ogni predicato che può variare da una situazione a un'altra. hanno un argomento in più: la situazione. Un fluente è un predicato la cui verità, per dati argomenti, può variare nel tempo.

Rappresentazione: definire un insieme di assiomi che descriva il mondo e le regole della sua evoluzione.

- Descrizione dello stato iniziale
- Descrizione delle azioni, con le condizioni della loro applicabilità e i loroeffetti
- Descrizione dell'obiettivo

Pianificazione nel calcolo situazionale: un problema di pianificazione si risolve dimostrando che esiste una situazione obiettivo (raggiungibile dalla situazione iniziale). Dalla soluzione si estrae il piano.

Limiti del calcolo situazionale:

- cambiamenti dovuti all'agente
- non ci sono eventi esterni
- Numero elevato di frame axioms
- inadeguato per rappresentare azioni che hanno durata, concorrenza, ect

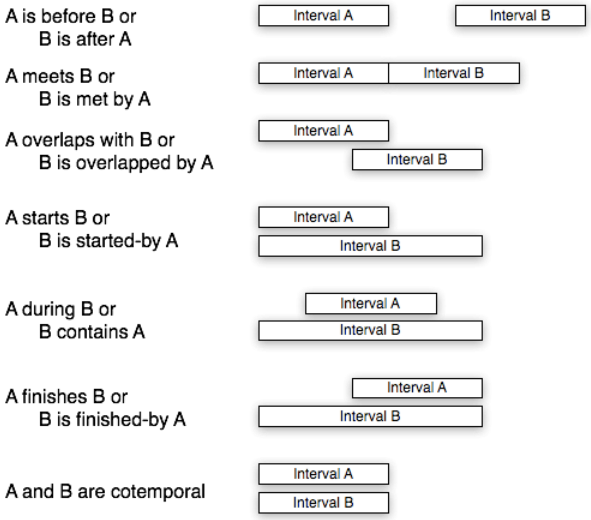
Ragionamento Temporale come CSP: tipi di vincoli temporali:

- Informazione qualitativa (relazioni). Es: L'evento A può avvenire prima o durante l'evento B
- Informazione Metrica (dati numerici). Es. 10:30 alle 11

4.4.2. Algebra degli Intervalli di Allen

Variabili: intervalli

Vincoli: sottoinsiemi dell'insieme delle 13 relazioni di base. 2^{13} vincoli diversi possibili.



Operazioni con i vincoli:

- inversione
- intersezione: l'intersezione è un vincolo su I1 e I2 con insiemi di relazione v ∩ v'
- composizione: Dati due vincoli: il primo definito sugli intervalli I1 e I2 insieme di relazioni v e il secondo definito su I2 e I3 con insieme di relazioni v'.
  La composizione è un vincolo su I1 e I3 con insieme v" rappresentante il più grande sottoinsieme di relazioni compatibili per transitività con v e v'.

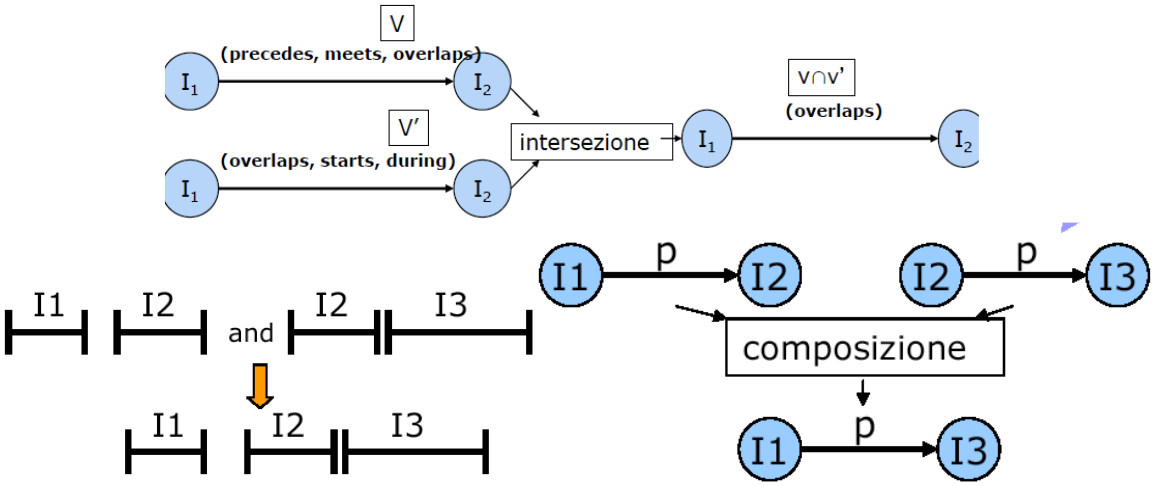


Tavola di composizione:

	before	after	meets	met-by	overlaps	overl.-by
before	before	*TEMP*	before	before meets overlaps starts during	before	before meets overlaps starts during
after	*TEMP*	after	during finishes after met-by overl.-by	after	during finishes after met-by overl.-by	after
meets	before	after met-by overl.-by started-by contains	before	finishes finished-by equals	before	overlaps starts during
met-by	before overlaps meets contains finished-by	after	starts started-by equals	after	during finishes overl.-by	after

5.Cenni di Apprendimento Automatico

Per risolvere alcuni problemi non è possibile applicare un approccio algoritmico tradizionale, per vari motivi, come l'impossibilità di formalizzare il problema, il rumore sui dati o l'alta complessità nel formulare una soluzione. Oppure ci sono dei casi in cui si riesce a formalizzare un problema ma non si ha idea di come risolverlo.

Per poter utilizzare questo approccio sono necessari:

- **Dati**: più ce ne sono meglio è, possono essere ottenuti in blocco, una volta per tutte o man mano interagendo con l'ambiente.
- **Conoscenza**: avere informazioni sul dominio applicativo permette di ottenere algoritmi più efficienti, anche in caso di informazioni incomplete o imprecise.

Si vogliono utilizzare i dati per ottenere una nuova conoscenza o raffinare quella di cui si dispone.

Ci sono 3 tipologie principali di apprendimento:

- Supervisionato
- Non supervisionato
- Con rinforzo

Apprendimento non supervisionato: L'agente impara a riconoscere pattern o schemi nell'input senza alcuna indicazione dei valori di output. (Clustering)

Apprendimento con Rinforzo: l'agente apprende esplorando l'ambiente e ricevendo ricompense nel caso di azioni positive. Lo scopo dell'agente è quello di massimizzare una funzione delle ricompense.

Apprendimento supervisionato: (reti neurali, SVM, decision threes, reti bayessiane)

- output discreto e limitato
- output continuo

Procedura di apprendimento:

- **Training**: al sistema viene fornito un insieme di coppie input-output, che adatta il proprio stato interno per classificare correttamente le coppie fornite.
- **Testing**: Al sistema viene fornito un diverso insieme di input (di cui si conosce l'output). Si valuta l'accuratezza del sistema, in termini di percentuale di risposte corrette.

Il dataset viene diviso in training set e testing set:

- Random sampling: i due dataset vengono presi a caso
- K-fold cross validation: Suddivido il dataset in K sottoinsiemi, alleno il sistema su K-1 sottoinsiemi e lo testo sul sottoinsieme restante. Itero K volte e prendo la mediana dei risultati.

Problema dell'overfitting: l'agente non generalizza il problema ma impara a memoria i dati.
Soluzione: Suddivido ulteriormente il training set, tenendo da parte un **validation set**. Durante l'allenamento, testo periodicamente l'accuratezza sul validation set: se l'errore aumenta, arresto l'allenamento.

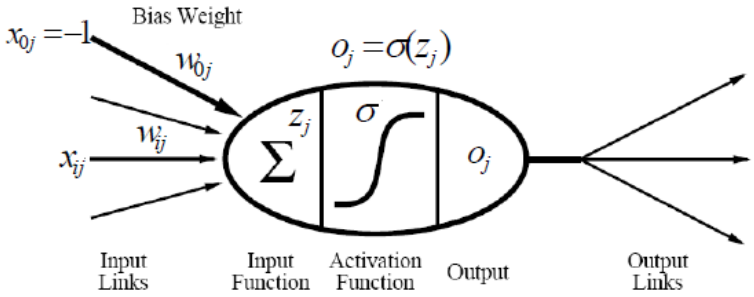
5.1. Reti neurali

Usate con:

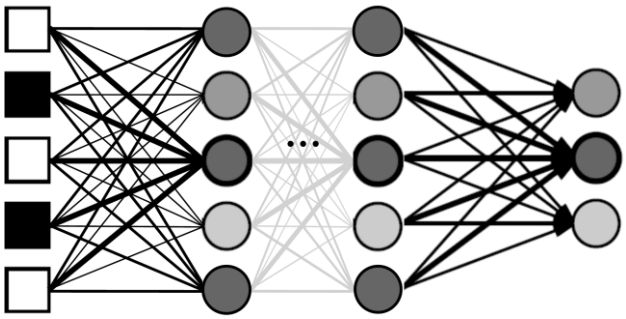
- Apprendimento supervisionato (classicazione, regressione, serie temporali, ...)
- Apprendimento non supervisionato (clustering, data mining, self-organized maps, memorie associative, ...)

Quando usare le reti neurali:

- Input: discreto e/o a valori reali, alta dimensionalità
- Output: vettore di valori discreti (classificazione) o reali (regressione)
- I dati (input e/o output) possono contenere rumore e la forma della funzione target totalmente sconosciuta
- Accettabile avere tempi lunghi di apprendimento e richiesta una veloce valutazione della funzione appresa
- La soluzione finale NON deve essere compresa da un esperto umano (“black box problem")



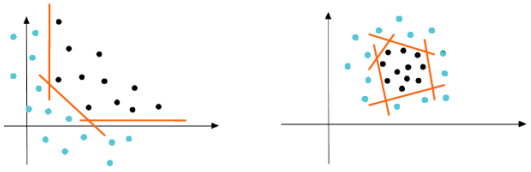
Reti neurali multistrato



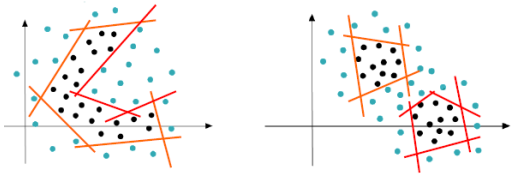
1 hidden layer è sufficiente per la stragrande maggioranza dei problemi (e l’allenamento è più rapido)

Una rete neurale dotata di due livelli nascosti (e di un numero sufficiente di nodi per livello) è in grado di rappresentare qualsiasi funzione, continua e discontinua, con precisione arbitraria

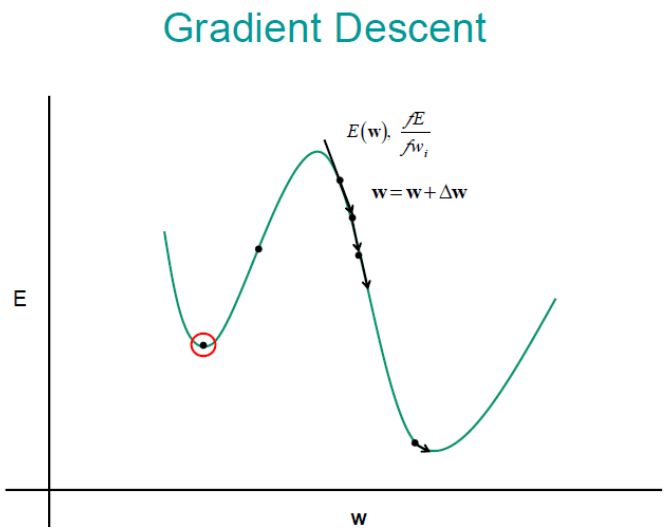
1 hidden layer: combinazioni lineari di semispazi



2 hidden layers: combinazioni lineari di combinazioni di semispazi



Backpropagation: Algoritmo di allenamento di una rete a partire dalle coppie (x,y) (training set). Sottopone più volte il training set alla rete, aggiustando i pesi per minimizzare l'errore quadratico. Algoritmo gradient descent, efficiente ma può arenarsi in un ottimo locale.
L'allenamento è, in generale, un problema NP-Completo.



Altre tipologie di reti neurali:

- Recurrent Neural Networks
- Associative Neural Networks
- Stochastic Neural Networks

Procedura di apprendimento

- 3 sottoinsiemi: Training, Validation e Test Set
- # nodi in ingresso = # features
- # nodi in uscita = # di classi
- # hidden layer e # nodi per livello: k-fold cross validation sul training set.
- Alleno la struttura scelta con tutto il training set,
- limitando l'overfitting col validation set.
- Valuto l'accuratezza finale sul test set.

Se devo scegliere il numero di nodi interni, parto con pochi e cresco (esponenzialmente) finché vedo un miglioramento