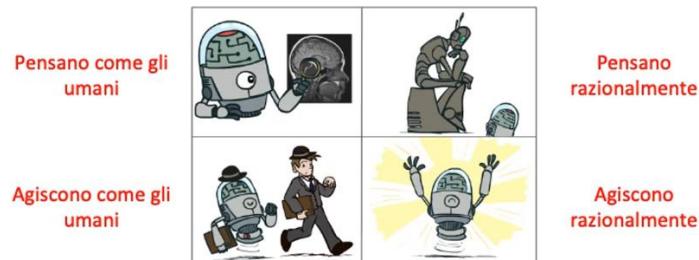


PER ALTRI APPUNTI CONSULTARE IL SITO:
https://luigi-v.github.io/Appunti_Universita/

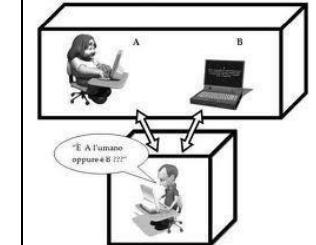
0. INTRO INTELLIGENZA ARTIFICIALE

Con il termine **Intelligenza Artificiale** si vuole costruire delle macchine intelligenti, ed abbiamo quattro aspetti che sono considerati di intelligenza:



Questo principio è stato pensato da Turing. Il **test di Turing** funziona nel seguente modo:
L'umano C interagisce attraverso una chat con un altro umano A e una macchina B, se non è in grado di distinguere che B è una macchina allora il test di Turing è superato.

Assumiamo che intelligente vuol dire umano, ovvero una macchina che interagisce come un essere umano.



Test di Turing:

- A uomo, B computer
- C persona (separata da A e B)
- C pone domande ad A e B per riconoscere chi è l'uomo e chi è il computer
- Comunicazione via testo
- B può cercare di ingannare C
- Idea alla base del test:
 - L'intelligenza può essere misurata guardando il comportamento esterno, non il ragionamento che ha portato a quel comportamento
 - Intelligente = umano
 - Ma ci sono comportamenti umani che non sono intelligenti, e viceversa

1. AGENTI INTELLIGENTI

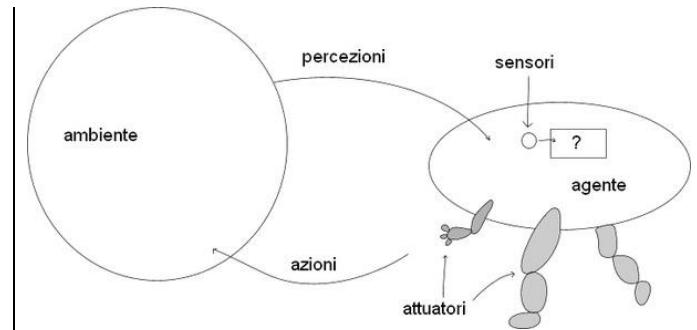
Un **agente** è un qualsiasi software che riceve delle informazioni, effettua delle elaborazioni e risponde a questa percezione facendo un'azione.

In altre parole, un agente è qualcosa che percepisce ed agisce in un ambiente, può essere sia solo hardware che solo software, anche un essere umano può essere considerato agente siccome percepiamo e poi agiamo.

Il termine **percezione** permette di riferirci al contenuto che i sensori di un agente stanno percependo. La sequenza percettiva di un agente è la storia completa di tutto ciò che l'agente ha percepito.

La scelta dell'**azione** di un agente in un dato istante può dipendere dalla sua conoscenza intrinseca e dall'intera sequenza percettiva osservata fino ad oggi, ma non da qualcosa che non ha percepito.

L'agente agisce sull'**ambiente** attraverso **attuatori**.



Quello di cui bisogna occuparsi è il punto interrogativo, ovvero decidere la migliore azione da effettuare in base a cosa ha percepito. Può essere modellato tramite una funzione, definita sulle percezioni ed ha come output un'azione.

La **funzione agente** mappa la storia delle percezioni in azioni:

$$f: P^* \rightarrow A$$

(l'asterisco indica che la funzione non lavora sull'ultima percezione ma potrebbe lavorare su N percezioni precedenti)

Il **programma agente** è in esecuzione sull'architettura per implementare f:

$$\text{agente} = \text{architettura} + \text{programma}$$

Esempio Aspirapolvere:

- **Percezioni:** posizione e contenuto, ad esempio [A, Sporco]
- **Azioni:** Sinistra, Destra, Aspira, NoOp

L'aspirapolvere presenta dei sensori che ci danno informazioni e può essere sia di posizione che di presenza di sporco.

Come andiamo a codificare la funzione f?

Lo si può fare in diversi modi, però una prima soluzione potrebbe essere quella di costruire una tabella:

Il dominio è rappresentato da tutte le sequenze percettive e per ognuna di esse diciamo l'azione da fare.

È una soluzione ottimale per sequenze limitate ma non per sequenze molto lunghe e potrebbero esserci aggiornamenti; quindi, è raro implementare la funzione tramite tabella.

Percept sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
:	:
[A, Clean], [A, Clean], [A, Clean]	Right
[A, Clean], [A, Clean], [A, Dirty]	Suck
:	:

L'alternativa è una funzione di questo tipo:

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

Bisogna però provare che questa sia la funzione giusta, detto in parole povere: è questo un agente intelligente (quindi razionale)?

1.1 RAZIONALITÀ

Viene fissata una **misura di prestazione** (questa misura deve essere minimizzata o massimizzata) che valuta la sequenza di stati dell'ambiente (percezioni):

- +1 per ogni spazio pulito in tempo T?
- +1 per ogni spazio pulito per istante di tempo, -1 per spostamento?
- Penalizzazione se ci sono >k riquadri sporchi?

Un **agente razionale** sceglie una qualunque azione che massimizza il valore atteso della misura di prestazione **data la sequenza di percezioni ottenuta fino all'istante corrente**. Effettua una scelta basata sulle percezioni che ha fino a quel momento, data la sua conoscenza fa la scelta migliore per massimizzare ma se mancano informazioni la scelta potrebbe non essere quella ottimale.

- Razionalità ≠ Onniscienza: le percezioni potrebbero non fornire tutte le informazioni rilevanti;
- Razionalità ≠ chiaroveggenza: l'effetto delle azioni potrebbe essere diverso da quello che mi aspetto;
- Razionalità ≠ successo;
- Razionalità → esplorazione, apprendimento, autonomia.

Un **agente intelligente** è un sistema che:

- Agisce intelligentemente/razionalmente;
- Fa ciò che è appropriato per la situazione e per i suoi obiettivi;
- È flessibile al variare dell'ambiente e degli obiettivi;
- Eventualmente impara dall'esperienza e fa scelte appropriate date le sue percezioni (limitate) ed i limiti della capacità computazionale.

Agire razionalmente significa agire:

- per raggiungere i propri obiettivi;
- date le proprie conoscenze;
- in funzione della sequenza di percezioni ricevute;
- date le azioni che si è in grado di compiere.

Le nuove percezioni non influenzano la descrizione (rappresentazione) del mondo dell'agente.

Un agente razionale fa la **cosa giusta**, ovvero quell'azione che procurerà all'agente il maggior successo massimizzando la misura di prestazione attesa (valutare, come e quando il successo atteso, considerato ciò che è stato percepito).

1.2 PEAS (Performance measure, Environment, Actuators, Sensors)

La specifica del **PEAS** è il primo passo nella progettazione di un *agente intelligente*, ad esempio, la progettazione di un pilota automatico per taxi:

- **Performance measure**: sicuro, veloce, ligio alla legge, profitti massimi (benzina e tempo);
- **Environment**: strada, altri veicoli nel traffico, pedoni, clienti;
- **Actuators**: accelerare, frenare, sterzare, frecce, clacson;
- **Sensors**: telecamere, sonar, tachimetro, GPS, contachilometri, accelerometro, sensori sullo stato del motore.

È importante capire, l'ambiente in cui si trova l'agente, e questi ambienti possono avere caratteristiche molto diverse tra di loro ed influenzano di molto l'agente da sviluppare. Pertanto, lo stato dell'agente dipende dalle sue percezioni.

OSSERVABILITÀ:

- **Completamente osservabile**: se i sensori dell'agente misurano tutti gli aspetti rilevanti per la scelta dell'azione (la rilevanza dipende dalla misura di prestazione). Gli ambienti di questo tipo, sono convenienti perché l'agente non ha bisogno di mantenere alcuno stato interno per tenere traccia del mondo;
- **Parzialmente osservabile**: sensori inaccurati, mancanza di alcuni sensori, rumore;
- **Non osservabile**: non ci sono sensori.

AGENTE SINGOLO O MULTIAGENTE:

- Agente che gioca a scacchi: ambiente multi-agente **competitivo**;
- Agente di traffico: ambiente multi-agente **parzialmente cooperativo** (evitare gli incidenti massimizza la misura di prestazione di tutti, trovare un parcheggio libero no).

Con più agenti bisogna capire che tipo di agenti ci sono (aiutano o vanno in competizione tra di loro dove la vittoria di uno significa sconfitta per un altro) quindi parliamo di un ambiente competitivo o parzialmente cooperativo.

AMBIENTE DETERMINISTICO/STOCASTICO:

- **Deterministico**: lo stato successivo dell'ambiente è determinato dallo stato corrente e dall'azione dell'agente, ad esempio completamente osservabile e deterministico (non ci sono incertezze);
- **Incerto**: se non completamente osservabile o non deterministico;
- **Stocastico**: incertezza descritta tramite probabilità;
- **Non deterministico**: azioni caratterizzate da risultati possibili, ma senza uso di probabilità.

AMBIENTE EPISODICO/SEQUENZIALE:

- **Episodico**: la scelta di un'azione non dipende dalle scelte fatte in "episodi" precedenti, ad esempio un agente che identifica i pezzi difettosi in una catena di montaggio.
- **Ambiente sequenziale**: una decisione può influenzare le successive (taxi, scacchi ecc.) in questo ambiente l'agente deve "pensare in avanti".

AMBIENTE DINAMICO STATICO:

- **Dinamico**: se l'ambiente può cambiare mentre l'agente pensa a che azione eseguire (guidare un taxi).
- **Semi-dinamico**: l'ambiente non cambia, ma la valutazione delle prestazioni dell'agente si (scacchi con orologio).

Altre caratteristiche di un ambiente:

- **Discreto/continuo**: stato dell'ambiente, gestione del tempo, percezioni e azioni;
- **Noto /ignoto**: stato di conoscenza dell'agente delle leggi dell'ambiente (può essere noto ma parzialmente osservabile, es solitario, oppure ignoto ma completamente osservabile es nuovo videogioco);
- **Caso più difficile**: parzialmente osservabile, multi-agente, stocastico, sequenziale, dinamico, continuo, ignoto (guidare un taxi).

TIPI DI AMBIENTE:

Il **tipo di ambiente** determina come progettare l'agente ed ovviamente il mondo reale è parzialmente osservabile, stocastico, sequenziale, dinamico, continuo, multi-agente.

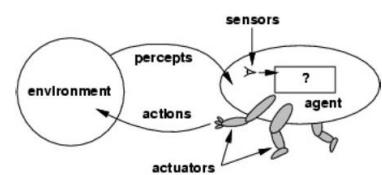
	Scacchi con tempo	Scacchi senza tempo	Guida Taxi	Internet shopping	Giocare a calcio	Fare un salto in alto
Osservabile	Si	Si	No	No	No	Si
Deterministico	Si	Si	No	Parzialmente	No	No
Episodico	No	No	No	No	No	No
Statico	Semi	Si	No	Semi	No	Si
Discrete	Si	Si	No	Si	No	No
Single agent	No	No	No	Si (eccetto aste)	No	Si

PROGETTAZIONE DI UN AGENTE:

Progettare il programma di un agente significa implementare la funzione corrispondenza delle percezioni alle azioni all'interno di un ambiente e con prestazioni predefinite.

La **funzione agente** è una descrizione astratta (formale), mentre il **programma agente** è una sua implementazione concreta in esecuzione sull'architettura dell'agente e l'**architettura** (meccanismo di calcolo) fornisce le percezioni al programma, esegue il programma e trasmette agli attuatori le azioni scelte dal programma.

Agente = architettura + programma



I **prerequisiti** è la conoscenza dettagliata di:

- tutte le **percezioni** e le possibili azioni conseguenti;
- **ambiente** (il problema di cui l'agente costituisce la soluzione) per scegliere tra tutte le possibili azioni quelle che saranno applicabili;
- **misura delle prestazioni** (definita a priori per non alterare la valutazione);
- **obiettivi** (per selezionare/valutare percezioni ed azioni).

Per implementare questi agenti si ha bisogno di uno **schema**:

Schema di agente semplice, il quale mantiene memoria della sequenza di percezioni (*incrementalità*) e si possono definire sistemi ad agenti cooperanti per applicazioni più complesse:

Agente basato su tabella, riceve la percezione e la aggiunge alla sequenza di quelle già ricevute e per decidere l'azione semplicemente va a vedere nella tabella a questa sequenza di percezione che azione corrisponde:

La tabella è totalmente specificata e l'agente tiene conto della sequenza delle percezioni consultando la tabella, la quale non viene aggiornata (in quanto già specificata).

Il vantaggio di questo tipo di agente è che abbiamo una progettazione semplice, completa ed efficace.

Mentre gli svantaggi sono che le dimensioni sempre crescenti (funzione della complessità dell'applicazione, ad esempio con gli scacchi 35¹⁰⁰ righe), il time-consuming per il progettista, nessuna autonomia è prevista per l'agente (non necessaria a causa della totale definizione delle corrispondenze) e di difficile aggiornamento, apprendimento complesso in ambiti limitati funzionano bene (ruolo dell'ambiente).

```

function SKELETON-AGENT(percept) returns action
static: memory, the agent's memory of the world
memory ← UPDATE-MEMORY(memory, percept)
action ← CHOOSE-BEST-ACTION(memory)
memory ← UPDATE-MEMORY(memory, action)
return action

function TABLE-DRIVEN-AGENT(percept) returns action
static: percepts, a sequence, initially empty



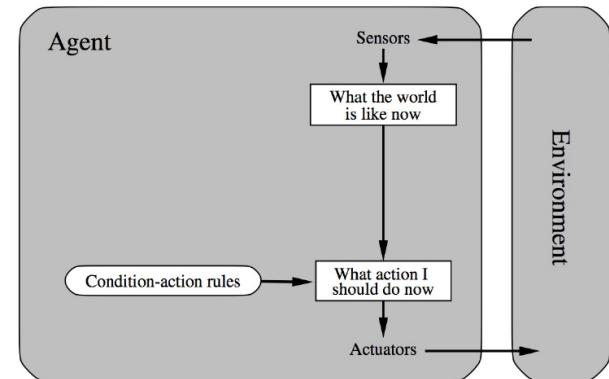


```

1.3 TIPOLOGIE DI PROGRAMMI DI AGENTI

AGENTE REATTIVO SEMPLICE (rispondono alle percezioni):

- Riceve dal sensore ciò che ha percepito, e sceglie l'azione in base alla percezione. Ci sono tanti ***if-then***. Trova la prima regola di produzione in accordo con la situazione corrente (definita da percezione in un ambiente osservabile e compie l'azione associata) ed ovviamente cambiando le regole cambia il tipo di agente.
- Si può applicare solo in alcuni contesti, ad esempio episodico, il problema in questo tipo di agenti è se l'***ambiente è osservabile o meno***, se non sono in grado di capire l'ambiente in che stato si trova, potrei eseguire un'azione sbagliata. Se l'ambiente è parzialmente osservabile può fallire. La soluzione è quella di tenere traccia delle percezioni passate (stato interno). Ad esempio, l'aspirapolvere.



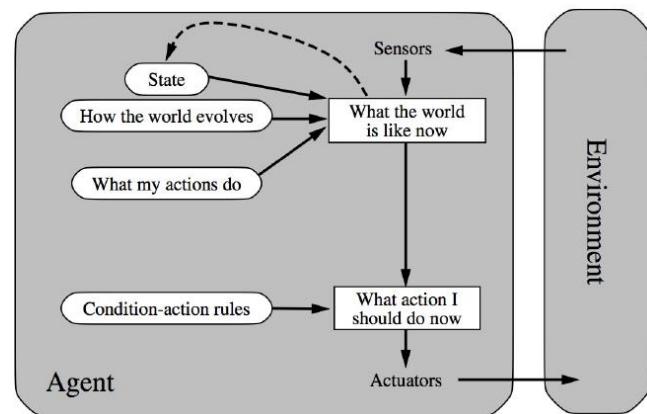
function SIMPLE-REFLEX-AGENT(*percept*) returns an action
persistent: *rules*, a set of condition-action rules

```

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

AGENTE BASATO SU MODELLO (considerano l'evolversi del mondo circostante):

- Aggiornare le informazioni sullo stato interno col passare del tempo richiede due tipi di conoscenza da codificare nel *programma agente* in qualche forma.
- In primis, la conoscenza che l'agente ha del mondo (*state*), di come esso evolve (*how*) e di cosa fanno le proprie azioni (*what*), si chiama ***modello del mondo***. Questi tipi di agenti hanno una conoscenza dello stato in cui si trova l'agente, l'ambiente in che stato si trova ed hanno la conoscenza di come il mondo evolve e cosa fanno le azioni. Avendo a disposizione queste informazioni, va a scegliere l'azione trova una regola in accordo alla situazione corrente (definita da percezioni e stato interno, compie l'azione associata e aggiorna lo stato del mondo).
- Il problema in questi tipi di agenti è che sono poco flessibili perché il comportamento è codificato direttamente nelle regole, la soluzione è l'introduzione di goal. Ad esempio, l'agente del taxi.



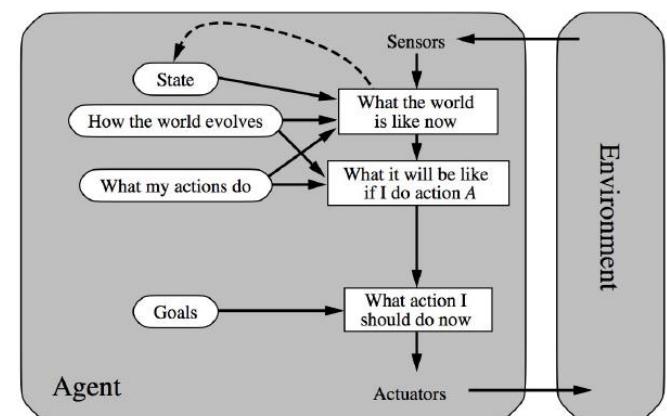
function MODEL-BASED-REFLEX-AGENT(*percept*) returns an action
persistent: *state*, the agent's current conception of the world state
model, a description of how the next state depends on current state and action
rules, a set of condition-action rules
action, the most recent action, initially none

```

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

AGENTI BASATI SU OBIETTIVI / PIANIFICATORI (agisce per raggiungere i propri obiettivi):

- La ricerca di una soluzione e la pianificazione permetteranno di identificare la sequenza di azioni che mettono in grado un agente di raggiungere i propri obiettivi. In questo caso la scelta dell'azione non è più pianificata ma dipende da cosa deve raggiungere l'agente.
- Anche questa tipologia di agenti presenta dei limiti, se ho più goal in conflitto fra loro o che non riesco a raggiungere pienamente, la soluzione è usare una ***misura di utilità***, codifichiamo i tanti obiettivi come se fossero parametri di una funzione così da dare più peso ad un obiettivo rispetto ad un altro. Viene fatta una valutazione dello stato e si sceglie lo stato con un'utilità maggiore.



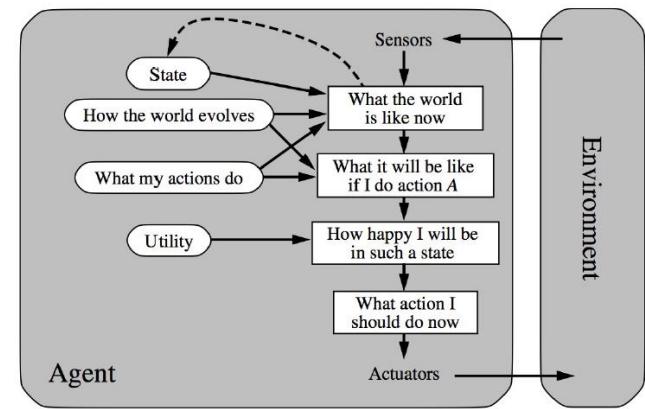
function GOAL-BASED-AGENT(*percept*) returns an action
persistent: *state*, the agent's current conception of the world state
model, a description of how the next state depends on current state and action
goal, a description of the desired goal state
plan, a sequence of actions to take, initially empty
action, the most recent action, initially none

```

state ← UPDATE-STATE(state, action, percept, model)
if GOAL-ACHIEVED(state, goal) then return a null action
if plan is empty then
    plan ← PLAN(state, goal, model)
    action ← FIRST(plan)
    plan ← REST(plan)
return action
    
```

AGENTI BASATI SU UTILITÀ (cercano di massimizzare le proprie utilità, *teoria delle decisioni*):

- L'**utilità** è una funzione predefinita che associa ad uno stato (o sequenza di stati se stiamo misurando l'utilità a lungo termine) dell'agente un numero reale che descrive il grado associato di felicità (quanto l'agente è felice di stare in quello stato). Uno stato del mondo preferibile ad un altro ha per l'agente una maggiore utilità. Si definisce quindi una **funzione di utilità**.



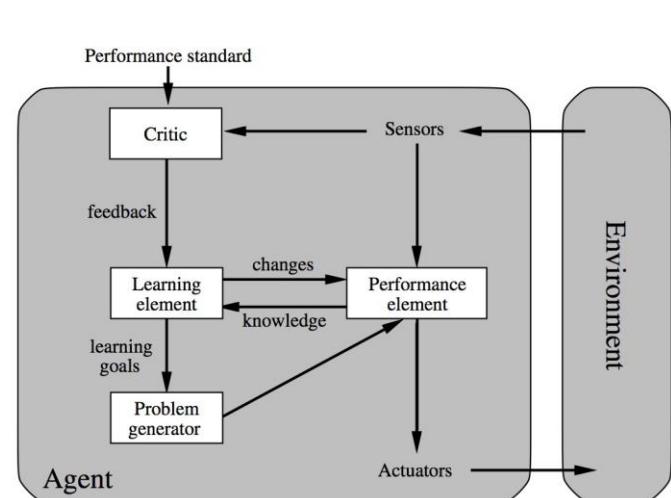
AGENTI BASATI SULL'APPRENDIMENTO:

Un'altra categoria di agenti è quella degli **agenti che variano nel tempo**, migliorano le loro prestazioni mediante l'**apprendimento** quindi in base all'esperienza acquisita durante la loro esecuzione.

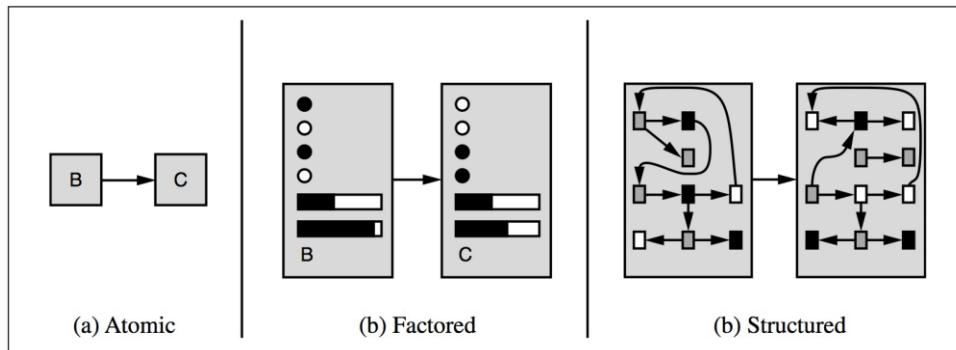
Per l'apprendimento di un agente intelligente si intende il processo che modifica ogni suo componente affinché si accordi meglio con l'informazione di feedback disponibile, migliorando così le prestazioni globali dell'agente.

Un agente di questo tipo può essere diviso in quattro componenti fondamentali:

- Il **performance element** seleziona le azioni sulla base delle percezioni, in questo box è racchiuso un **agente precedente**;
- Il **learning element** permette di migliorare le performance dell'agente. Per poter decidere quali azioni cambiare, ha bisogno di informazioni, del tipo: dove si trova l'agente è uno stato buono o è successo qualcosa di negativo?
- Tutto ciò viene detto dal modulo **critic** (detto *analizzatore*) che analizza le prestazioni correnti e decide se e come modificare l'elemento esecutivo per migliorarne. Dopo aver ottenuto queste informazioni, in base alla knowledge decide cosa cambiare.
- Il modulo **problem generator** suggerisce azioni che portino a esperienze nuove e significative, poiché altrimenti l'agente svolgerebbe sempre le stesse operazioni e quindi con questo modulo si può esplorare l'ambiente fornendo problemi di ottimizzazioni.



RAPPRESENTAZIONE DEGLI STATI:



- Nel caso (a) abbiamo solo i nomi degli stati e viene usato da algoritmi di ricerca HMM, l'agente deve cercare di trovare il percorso tra gli stati che mi porta all'obiettivo da raggiungere. Ogni stato del mondo è indivisibile e non ha una struttura interna.
- Nel caso (b) possiamo considerare degli stati in cui all'interno abbiamo degli attributi, usato da algoritmi di CSP, pianificazione e reti bayesiane.
- Nell'ultimo caso, prevede che gli attributi degli stati siano connessi tra di loro, usato da logica del primo ordine, modelli probabilistici, apprendimento basato sulla conoscenza NLP. Sarebbero dei grafi, con all'interno di ogni nodo degli ulteriori grafi, per rappresentare in maniera dettagliata la conoscenza.

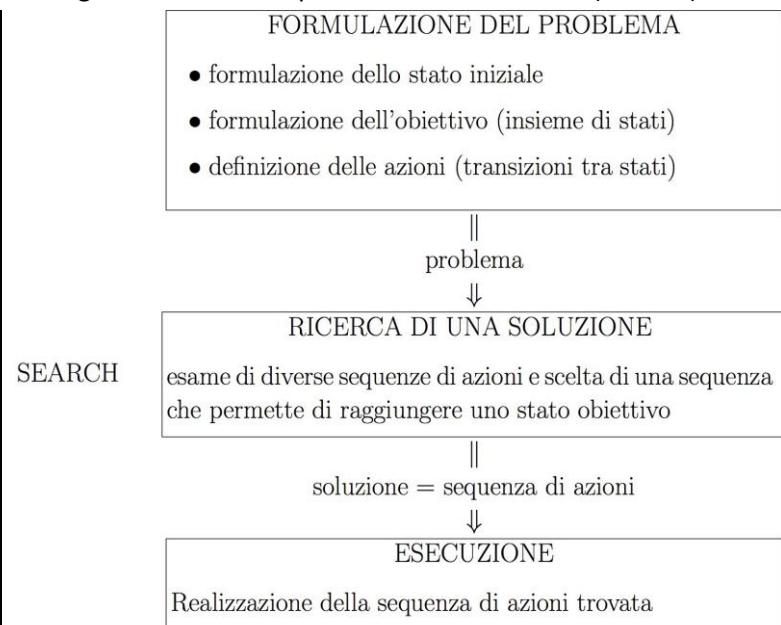
2. RISOLVERE I PROBLEMI CON LA RICERCA

AGENTI RISOLITORI DI PROBLEMI:

Adottano il paradigma della **risoluzione di problemi** come ricerca in uno spazio di stati. Sono particolari agenti con obiettivo, che pianificano l'intera sequenza di mosse prima di agire. Gli stati sono privi di struttura interna (atomici).

Passi da seguire:

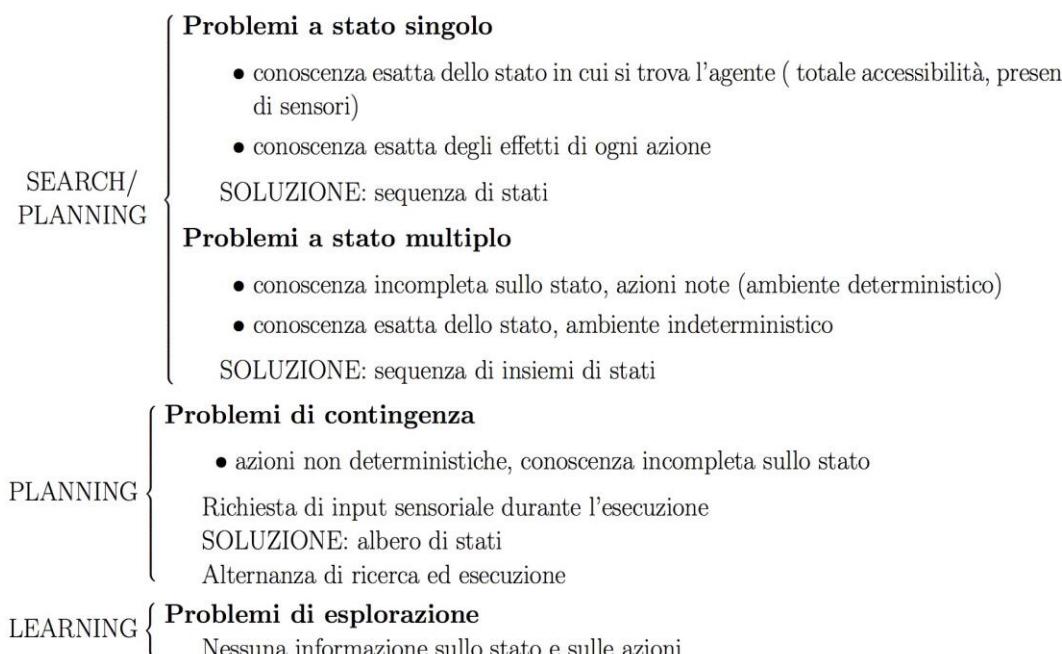
- **Determinazione obiettivo** (un insieme di stati);
- **Formulazione del problema**: l'agente escogita una descrizione degli stati e delle azioni necessarie per raggiungere l'obiettivo;
- **Determinazione della soluzione mediante ricerca**: Prima di intraprendere qualsiasi azione nel mondo reale, l'agente simula sequenze di azioni nel suo modello, cercando fino a trovare una sequenza di azioni che raggiunga l'obiettivo. Tale sequenza è detta soluzione;
- **Esecuzione del piano**.



Formalizzare significa definire una quadrupla o quintupla, capire quindi gli elementi del problema. Nella definizione del problema, l'ambiente viene rappresentato come un grafo degli stati (**spazio degli stati**) che descrive l'ambiente, quindi in quali stati l'ambiente si può trovare. L'algoritmo mi dovrà dire, qual è il percorso migliore per poter arrivare dallo stato iniziale ad uno dei tanti nodi obiettivo o l'unico nodo obiettivo. Il tipo di problema può portare a formulazioni differenti.

2.1 TIPI DI PROBLEMI

I problemi affrontabili dipendono dalla conoscenza che l'agente ha sullo stato in cui si trova e sulle azioni (e i suoi effetti).



- **Deterministico, pienamente osservabile** → problema a singolo stato dove l'agente conosce esattamente in quale stato si troverà, la soluzione è una sequenza fissata di azioni;
- **Non osservabile** → problema senza sensore dove l'agente potrebbe non sapere dov'è; la soluzione è una sequenza;
- **Non deterministico e/o parzialmente osservabile** → problema di contingenza: le percezioni forniscono nuove informazioni sullo stato corrente e spesso alternano ricerca ed esecuzione;
- **Spazio degli stati sconosciuto** → problema di esplorazione.

2.1 PROBLEMI SI SEARCH (a stato singolo o multiplo)

Componenti che definiscono un problema:

1. **Stato iniziale**: l'agente da che punto parte?
2. **Azioni o operatori**;
3. Test obiettivo (*goal test*);
4. **Funzione costo**: associa un costo ad ogni operatore. La **funzione** è importante perché l'agente deve minimizzare il costo.

	A stato singolo	A stato multiplo	SPAZIO DEGLI STATI
stato iniziale	$\{s_0\}$	$\{s_0, \dots, s_n\}$	
azioni o operatori	$op(s) = s'$	$op^*(\{s_1, \dots, s_k\}) = \{op(s_1)\} \cup \dots \cup \{op(s_k)\}$ (operatori su insiemi)	
goal test	$goal_state(s)$	$goal_state^*(\{s_1, \dots, s_k\}) = goal_state(s_1) \wedge \dots \wedge goal_state(s_k)$ (goal test per insiemi di stati)	

Dobbiamo definire una **funzione di transizione**, la quale quando l'agente si trova in uno stato ed esegue un'azione ci dice dove va a finire. In questo caso la funzione di transizione definisce lo spazio degli stati.

Un **cammino** è una sequenza di operatori. Il costo di un cammino è la somma dei costi degli operatori che lo compongono ed una soluzione è un cammino dallo stato iniziale (da uno qualsiasi degli stati iniziali) a uno stato che soddisfa il goal-test: è una **sequenza di operatori**.

Lo spazio degli stati può essere rappresentato come un grafo dove i vertici rappresentano gli stati mentre gli archi orientati tra di essi rappresentano le azioni.

ALGORITMI DI RICERCA:

Gli algoritmi di ricerca prendono in input una formalizzazione di un problema (tabella sopra) e restituiscono un **cammino soluzione**, cioè una sequenza di azioni che portano dallo stato iniziale allo stato goal. Ovviamente l'algoritmo mi deve fornire il miglior percorso, il quale viene definito come quello con funzione di costo minore.

Ogni **nodo** nell'albero di ricerca corrisponde a uno **stato** nello spazio degli stati e gli **archi** nell'albero di ricerca corrispondono alle azioni. La radice dell'albero corrisponde allo stato iniziale del problema.

È importante capire la differenza tra lo spazio degli stati e l'albero di ricerca:

- Lo **spazio degli stati** descrive l'insieme (forse infinito) di stati nel mondo, e le azioni che permettono transizioni da uno stato all'altro.
- L'**albero di ricerca** descrive i percorsi tra questi stati, raggiungendo l'obiettivo.

Viene definito quindi un **costo totale = costo della ricerca + costo del cammino soluzione**, perché molto spesso capita che cercare la migliore soluzione potrebbe richiedere molto tempo. Quindi in alcuni scenari (soprattutto quando lo spazio degli stati è molto ampio) non possiamo aspettare ore per avere l'ottimo, ma è opportuno avere una soluzione sub-ottimale con un tempo di risposta inferiore.

Struttura generale di un algoritmo di ricerca:

Inizialmente viene definito uno stato obiettivo (**goal**), viene formulato un problema con lo stato e l'obiettivo (**problem**) e rilascia la soluzione in **seq** che rappresenta la sequenza di azioni. Dopodiché prendo la prima azione e la restituisco (**action**) e il resto delle azioni saranno in seq; quindi, la seconda volta non entra in **seq is empty** e prosegue ad eseguire il piano fino a quando seq non si svuota.

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

  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then do
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
    action  $\leftarrow$  FIRST(seq)
    seq  $\leftarrow$  REST(seq)
  return action
```

Esempio Romania:

In vacanza in Romania, ora in Arad, l'aereo parte domani da Bucarest:

- Formulazione dell'obiettivo: essere a Bucarest
- Formulazione del problema:
 - stati: le varie città
 - azioni: guidare tra le città
- Trovare una soluzione:
 - sequenze di città ad esempio Arad, Sibiu, Fagaras, Bucarest

Rappresentiamo l'ambiente con le città, quindi ogni stato è una città.

Che tipo di assunzioni?

- L'ambiente è statico significa che non ci sono novità durante l'esecuzione del piano;
- Osservabile perché abbiamo tutte le informazioni (mappa);
- Discreto un insieme finito di azioni possibili;
- Deterministico si assume che l'agente possa eseguire il piano ad occhi chiusi ed arriva in questo caso a Bucarest.

VACUUM WORLD (il problema):

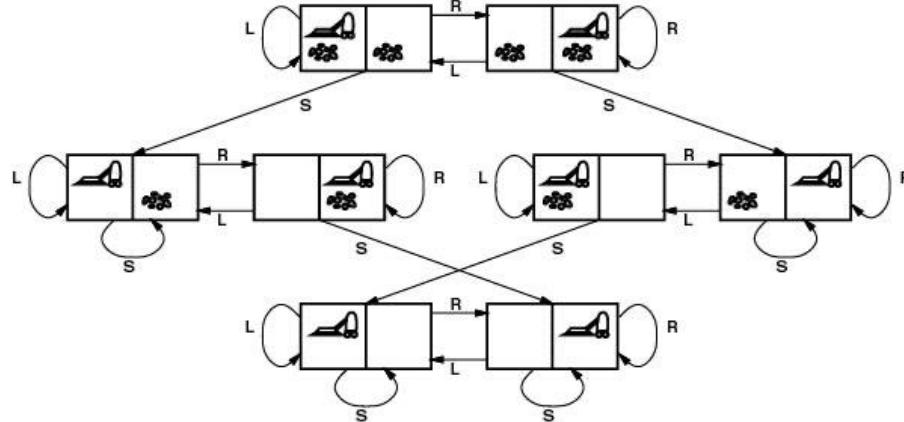
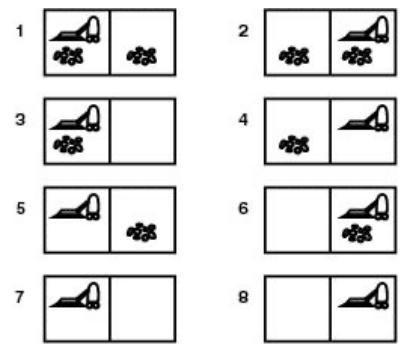
Versione semplice con solo due locazioni (sporche o pulite) e l'agente può essere in una delle due.

- *Percezioni*: Dirt, No_dirt
- *Azioni*: Left, Right, Suck

▪ *L'obiettivo* è di rimuovere lo sporco e trovarsi negli stati {7,8};

▪ *Funzione di costo*: ogni azione ha costo 1;

▪ *Spazio degli stati*:



Ad esempio:

- *Stato singolo*: se parto dallo stato 5, la soluzione è {Right, Suck};
- *Senza sensore*: inizia in {1,2,3,4,5,6,7,8} ad esempio, Right va in {2,4,6,8}, la soluzione è {Right, Suck, Left, Suck};

Contingenza:

- *Non deterministico*: Suck può sporcare una cella pulita;
- *Parzialmente osservabile*: posizione, sporco nella cella attuale;
- *Percepisce*: {L,Clean} cioè inizia in 5 o 7;

Soluzione? Right, if dirt then Suck

2.2 FORMULAZIONE PROBLEMA SINGOLO STATO

Un **problema** è definito da quattro elementi:

1. **Stato iniziale**, ad esempio 'Arad' (città);
2. **Azioni e funzioni successore** $S(x)$ = insieme di coppie azione-stato: ad esempio $S(\text{Arad})=\{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots\}$;
3. **Test obiettivo**, può essere:
 - a. **Esplicito**: elenco di stati, ad esempio una destinazione, $x=$ "In (Bucarest)";
 - b. **Implicito**: ad esempio, $\text{NoDirt}(X)$, scaccomatto.
4. **Costo di cammino** (somma):
 - a. Ad esempio, somma di distanze, numero di azioni eseguite etc;
 - b. $C(x,a,y)$ è il **costo di un passo**, che si assume essere ≥ 0 .

Questa quadrupla identificata viene fornita all'algoritmo per poter cercare la soluzione.

Una **soluzione** è una sequenza di azioni che portano da uno stato iniziale ad uno stato obiettivo.

SELEZIONARE UNO SPAZIO DI STATI:

Il mondo reale è molto complesso quindi bisogna effettuare un'**astrazione** dello spazio degli stati per risolvere problemi; quindi, ignoriamo i dettagli che non servono.

- Stato (astratto) = insieme di stati reali;
- Azione (astratta) = combinazione complessa di azioni reali, ad esempio, Arad \rightarrow Zerind rappresenta una serie complessa di possibili percorsi, deviazioni;
- Per garantire la realizzabilità, un qualsiasi stato vero "in Arad" deve raggiungere un qualsiasi stato vero in "Zerind";
- Soluzione (astratta) = insieme di percorsi reali che sono soluzioni nel mondo reale;
- Ogni azione astratta dovrebbe essere "più facile" rispetto al problema originale.

Come capire se l'astrazione è corretta? Se la soluzione che da l'algoritmo sia eseguibile. L'output dell'algoritmo, che sarebbe questa sequenza di azioni, sia effettivamente eseguibile dall'agente e porta dallo stato iniziale allo stato obiettivo.

Trovare la soluzione ottima per N-Puzzle è NP-HARD dovuto al numero di stati molto ampio.

Esempio puzzle dell'otto:

Stati: configurazioni della scacchiera

Stato iniziale: una certa configurazione

Obiettivo: una certa configurazione

Successori: mosse della casella bianca

in sù: ↑

in giù: ↓

a destra: →

a sinistra: ←

Goal-Test: Stato obiettivo?

Path-Cost: ogni passo costa 1

Lo spazio degli stati è un grafo con possibili cicli.

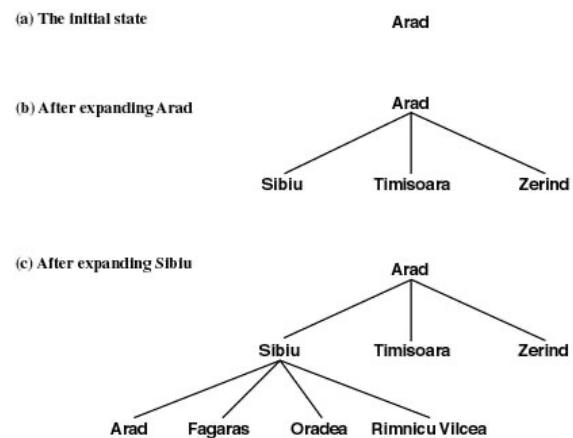
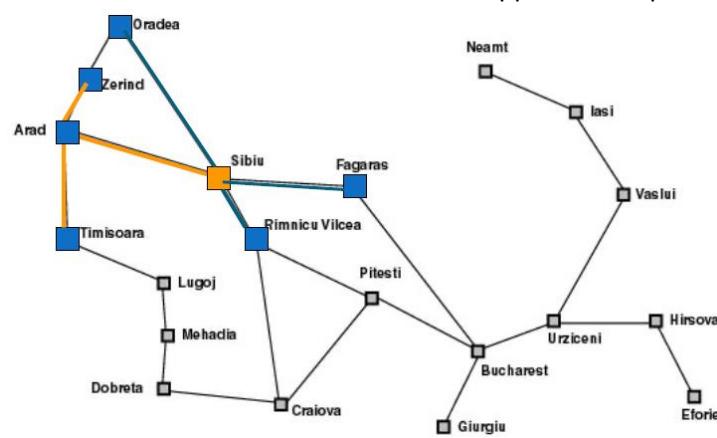
1	2	3
8		4
7	6	5

RICERCA DELLA SOLUZIONE:

Si parte dallo stato iniziale, espande quel nodo, aggiunge i suoi vicini sulla **frontiera** dopodiché sceglie uno dei nodi sulla frontiera per selezionarlo ed espanderlo.

Ogni nodo nell'albero di ricerca è un intero percorso nel grafo dello spazio degli stati.

Generazione di un **albero di ricerca** sovrapposto allo spazio degli stati:



STRATEGIA DI RICERCA:

1. Scegliere (tra le foglie dell'albero) un nodo da espandere, secondo una data strategia;
2. Controllare se il nodo scelto è un obiettivo;
3. Se non lo è, espandere il nodo: generare i suoi figli, ciascuno dei quali contiene uno stato risultante dall'applicazione di un operatore allo stato del nodo espanso.

Quando si espande un nodo si calcolano tutte le componenti dei nodi generati. La collezione di nodi in attesa di essere espansi (le foglie dell'albero) è chiamata **confine** o **frontiera**.

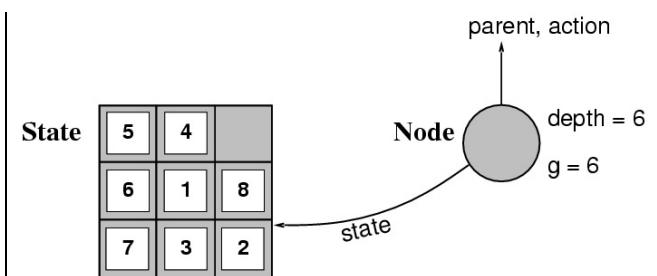
Se non ho nodi da espandere, vuol dire che non ho trovato il nodo obiettivo e quindi l'algoritmo non ha soluzione e fallisce.

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

I NODI DELL'ALBERO DI RICERCA:

Un nodo ha cinque componenti:

Uno **stato**, il **nodo genitore**, l'**azione** eseguita per generarlo, la **profondità** del nodo (distanza dalla radice), il **costo del cammino** dal nodo iniziale al nodo $g(x)$.



STRUTTURA DATI PER FRONTIERA:

Frontiera: lista dei nodi in attesa di essere espansi (le foglie dell'albero di ricerca), viene implementata come una coda con operazioni: Make-Queue, Empty?, First, Rest, RemoveFront, Insert, InsertAll.

I diversi tipi di coda hanno diverse Insert.

```

function TREE-SEARCH(problem,fringe) return a solution or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node  $\leftarrow$  REMOVE-FIRST(fringe)
    if GOAL-TEST[problem] applied to STATE[node] succeeds
      then return SOLUTION(node)
    fringe  $\leftarrow$  INSERT-ALL(EXPAND(node, problem), fringe)
  
```

```

function EXPAND(node, problem) return a set of nodes
  successors  $\leftarrow$  the empty set
  for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
    s  $\leftarrow$  a new NODE
    STATE[s]  $\leftarrow$  result
    PARENT-NODE[s]  $\leftarrow$  node
    ACTION[s]  $\leftarrow$  action
    PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s]  $\leftarrow$  DEPTH[node] + 1
    add s to successors
  return successors

```

Un aspetto importante degli algoritmi di ricerca è **evitare gli stati ripetuti**, perché su spazi di stati a grafo si generano più volte gli stessi nodi nella ricerca. Per evitarlo, si possono tenere in memoria stati già visitati, occupa spazio ma ci consente di evitare di visitarli di nuovo.

Quindi, si mantiene una lista dei nodi visitati (**closed**), prima di espandere un nodo si controlla se lo stato era stato già incontrato visitando un altro nodo. Se questo succede il nodo appena trovato non viene espanso.

Questo tipo di approccio funziona se i nodi che trovo dopo sono peggiori del primo, perché se ritrovassi lo stesso nodo dopo potrebbe far parte di un percorso migliore del primo e non possiamo ignorarlo, altrimenti potrei non arrivare all'ottimo.

In questo caso teniamo traccia degli stati già visitati in *closed*.

```

function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
  closed  $\leftarrow$  an empty set
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
  
```

VALUTAZIONE DI UNA STRATEGIA:

Una strategia di ricerca è definita scegliendo l'**ordine di espansione dei nodi**. Le strategie vengono valutate secondo le seguenti dimensioni:

- **Completezza**: se la soluzione esiste viene trovata;
- **Ottimalità (ammissibilità)**: trova la soluzione migliore, con costo minore;
- **Complessità nel tempo**: tempo richiesto per trovare la soluzione;
- **Complessità nello spazio**: memoria richiesta.

Le complessità di tempo e spazio si misurano in termini di:

- **b**: fattore di ramificazione massima dell'albero di ricerca (branching factor), indica quante azioni posso fare al massimo in un nodo;
- **d**: la profondità della soluzione a costo minimo;
- **m**: massima profondità dello spazio degli stati (potrebbe essere infinita).

2.3 STRATEGIE NON INFORMATE

Sono algoritmi che utilizzano solo ciò che abbiamo definito nel problema. Possono generare nuovi stati e possono distinguere uno stato obiettivo. Se uno stato non è obiettivo non sanno capire quanto è promettente. E sono:

Ricerca in ampiezza, Ricerca di costo uniforme, Ricerca in profondità, Ricerca in profondità limitata, Ricerca con approfondimento iterativo e Ricerca bidirezionale.

Criterio	BF	UC	DF	DL	ID	Bidir	
Tempo	b^{d+1}	$b^{1+\lfloor C^*/\varepsilon \rfloor}$	b^m	b^l	b^d	$b^{d/2}$	(*) se gli operatori hanno tutti lo stesso costo
Spazio	b^{d+1}	$b^{1+\lfloor C^*/\varepsilon \rfloor}$	bm	bl	bd	$b^{d/2}$	(**) per costi degli archi $\geq \varepsilon > 0$
Ottimale?	si(*)	si(**)	no	no	si(*)	si	(+) per problemi per cui si conosce un limite alla profondità della soluzione (se $l > d$)
Completa?	si	si(**)	no	si (+)	si	si	

3. ALGORITMI DI RICERCA INFORMATA

La ricerca esaustiva non è praticabile in problemi di complessità esponenziale, potremmo quindi usare una **conoscenza** del problema (positiva o negativa a seconda di come viene definita) ed esperienza per riconoscere i cammini più promettenti.

La **conoscenza euristica** dipende dal problema, ogni problema usa una serie di euristiche che possiamo definire e che aiutano a fare scelte oculate:

- Non evita la ricerca ma la **riduce**;
- Consente in genere di trovare una buona soluzione in tempi accettabili;
- Sotto certe condizioni garantisce **completezza** e **ottimalità**.

Queste ipotesi usate come conoscenza euristica, permettono di evitare di navigare tutto lo spazio degli stati.

La conoscenza viene data tramite una **funzione di valutazione** dello stato, detta funzione di valutazione euristica:

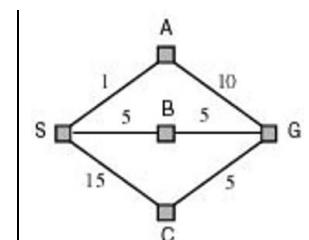
$$f: n \rightarrow R$$

Dipende solo dallo stato e fornisce una stima del costo di una soluzione che passa per il nodo n e arriva all'obiettivo.

Esempio di Euristică:

La città più vicina (o la città più vicina alla metà in linea d'aria) nel *route-finding*.

Per dire che una data città è vicina alla destinazione, si può vedere in linea d'aria quanto è vicina (*stima di conoscenza*). Potrebbe esserci una strada che li unisce o una strada con tante curve o non esserci proprio una strada. Tutto ciò è una conoscenza da poter dare all'algoritmo.



Quello che serve per poter definire una euristica è giudicare quanto un dato stato (o configurazione) è vicina ad uno stato obiettivo ed è un criterio da definire e dare all'algoritmo.

ALGORITMO BEST-FIRST:

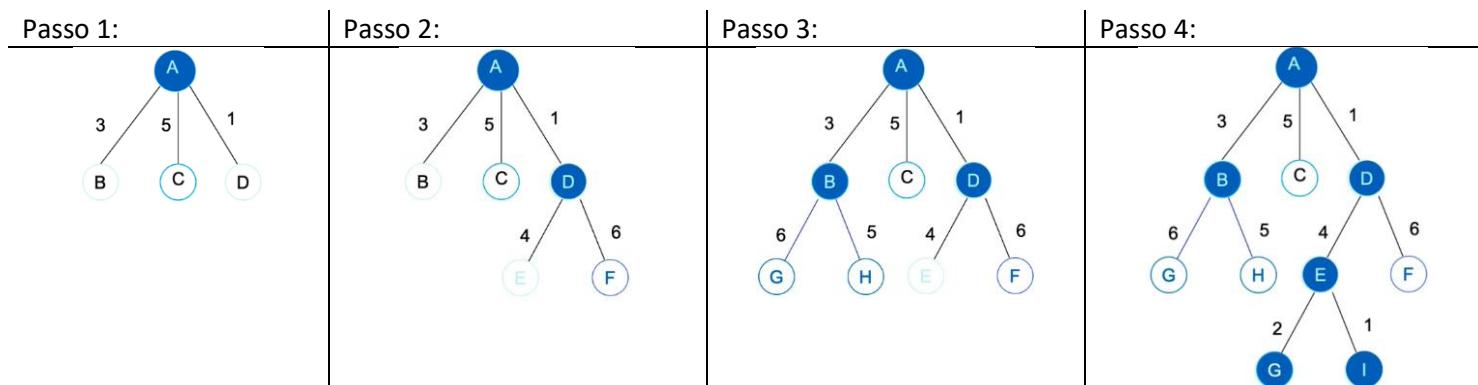
Bisogna decidiamo quale nodo della frontiera espandere come successivo. Un approccio generale è quello del **best-first search** il quale ad ogni passo sceglie il nodo n sulla frontiera per cui il valore della $f(f(n))$ è il migliore (nodo più promettente), dove migliore significa '**minore**' in caso di stima della distanza della soluzione, lo restituisce se è un **goal state** altrimenti espande i nodi figli.

Ogni nodo figlio viene aggiunto alla frontiera se non era stato raggiunto in precedenza oppure riaggiunto se ora è stato raggiunto con un cammino che ha il costo minore tra tutti quelli visti in precedenza.

L'algoritmo viene implementato da una coda con priorità che ordina in base al valore della funzione di valutazione euristica.

Diverse funzioni di valutazione determinano diverse versioni della ricerca Best-First, ad esempio la **ricerca guidata dal costo** è un caso particolare di ricerca best-first, in cui $f=g$, la funzione di valutazione è $g(n)$: costo del cammino dallo stato iniziale a n . Informazione euristica nulla.

Funzionamento:



RICERCA GREEDY BEST-FIRST:

Si usa come ricerca euristica una stima della distanza della soluzione, indicata con $h(n)[H \geq 0]$.

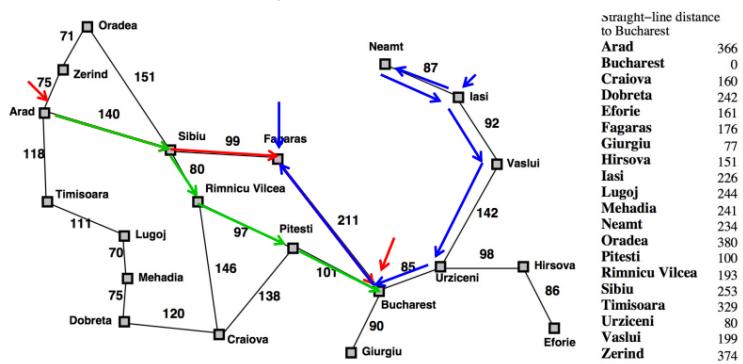
Espande prima il nodo con il valore $h(n)$ più basso, il nodo che sembra essere il più vicino all'obiettivo sulla base del fatto che questo potrebbe portare rapidamente ad una soluzione, quindi la funzione di valutazione $f(n)=h(n)$.

($f = \text{costo dell'intero percorso passando per un nodo } n$, mentre $h = \text{costo per arrivare all'obiettivo a partire da } n$)

Esempio:

Ricerca greedy per Route finding dove $h(n)$ = distanza in linea d'aria tra lo stato di n e la destinazione.

L'algoritmo è chiamato **greedy** perché come si può notare la soluzione viene trovata ma non è quella ottimale (perché costa 32 km in più rispetto a quella ottimale), in quanto ad ogni iterazione cerca di avvicinarsi il più possibile ad un obiettivo ma questo potrebbe portare a risultati peggiori.



Da Arad a Bucharest ...

Greedy: Arad, Sibiu, Fagaras, Bucharest (450)

Ottimo: Arad, Sibiu, Rimnicu, Pitesti, Bucharest (418)

L'algoritmo è:

- **Completo?** No, potrebbe andare in loop;
- **Tempo?** $O(b^m)$ ma una buona euristica può migliorare le prestazioni;
- **Spazio?** $O(b^m)$ mantiene tutti i nodi in memoria;
- **Ottimale?** No, non tiene conto dello sforzo per arrivare a quel nodo.

ALGORITMO A:

Un **algoritmo A** è un algoritmo BF con una funzione di valutazione dello stato del tipo:

$$f(n) = g(n) + h(n), \text{ con } h(n) \geq 0 \text{ e } h(goal) = 0$$

- $g(n)$ è il costo del cammino percorso per raggiungere n ;
- $h(n)$ una stima del costo per raggiungere da n un nodo goal.

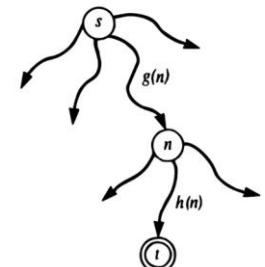
Casi particolari dell'algoritmo A:

- Se $h(n) = 0$ [$f(n)=g(n)$] si ha Ricerca a costo uniforme;
- Se $g(n) = 0$ [$f(n)=h(n)$] si ha Greedy BF.

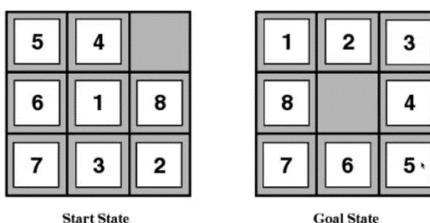
La funzione $f(n) = g(n) + h(n)$ nell'algoritmo A, dove $g(n)$ è noto mentre $h(n)$ rappresenta l'ipotesi euristica.

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

Noto Ipotesi dell'euristica



Esempio nel gioco dell'otto:



$$f(n) = \#mosse + \#\text{caselle-fuori-posto}$$

$$f(\text{Start}) = 0 + 7 \quad \text{Dopo } \leftarrow, \downarrow, \rightarrow, \uparrow \quad f = 4 + 7$$

LA STIMA IDEALE:

Funzione di valutazione ideale (*oracolo*) dove le informazioni sono esatte:

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

- $g^*(n)$ costo del cammino minimo dalla radice a n ;
- $h^*(n)$ costo del cammino minimo da n a goal;
- $f^*(n)$ costo del cammino minimo (ottimo) da radice a goal, attraverso n .

Normalmente $g(n) \geq g^*(n)$ e $h(n)$ è una stima di $h^*(n)$.

L'algoritmo A* è sia **completo** che **ottimale** quando la funzione euristica è **ammissibile**.

- **Definizione:**

Euristica ammissibile $\forall n \ h(n) \leq h^*(n)$ dove h è una sottostima (euristica che non sovrastima mai il costo per raggiungere un obiettivo ed è quindi **ottimistica**).

- **Definizione:**

Algoritmo A* è un algoritmo A in cui h è una funzione euristica ammissibile.

- **Teorema:**

Gli algoritmi A* sono ottimali:

$$f(n) = g(n) + h(n), \text{ con } h(n) \geq 0 \text{ e } h(goal) = 0, \text{ e inoltre } \forall n \ h(n) \leq h^*(n) \text{ dove } h \text{ è una sottostima.}$$

Corollario: BF e UC sono ottimali ($h(n)=0$) dove ipotesi nulla vuol dire sottostimare.

4. OLTRE LA RICERCA CLASSICA

Considereremo approcci differenti a problemi di ricerca, in particolare affronteremo lo stesso tipo di problemi (ricerca di soluzioni ottime) sempre in uno spazio degli stati.

- **Ricerca locale:** i quali si applicano solo a particolari tipi di problemi. Hanno il vantaggio di non richiedere troppa memoria: *Hill-climbing*, *Simulated annealing* e *Algoritmi genetici*;
- **Ricerca con azioni non deterministiche:** quando si eseguono azioni non si sa lo stato finale (azioni non deterministiche): *Alberi AND-OR*;
- **Online:** non si ha tutta la conoscenza dai sensori: *Online DFS* e *LRTA**.

4.1 RICERCA LOCALE

Algoritmi che non sono sempre utilizzabili poiché ci sono dei problemi in cui non è necessario conoscere la sequenza di azioni, poiché si vuole solo lo stato obiettivo. L'obiettivo è lo stato obiettivo e non il percorso, infatti la soluzione è costituita dallo stato goal stesso, esempio le otto regine vogliono sapere solo la configurazione finale con tutte le regine che non si attaccano e non i passi fatti. Quindi lo spazio degli stati è dato dall'insieme delle configurazioni "complete".

Trovare una configurazione **ottima**, oppure trovare una configurazione che soddisfi dei vincoli.

La ricerca locale è basata sull'esplorazione iterativa di **soluzioni vicine**, che possono migliorare la soluzione corrente mediante modifiche locali (ci si sposta solo tra vicini). Rappresentiamo ogni stato come una possibile soluzione del problema. L'algoritmo si basa sull'analisi del suo vicinato (**neighborhood**), dove una struttura dei "vicini" è una funzione **F** che assegna a ogni soluzione **s** dell'insieme di soluzioni **S** un insieme di soluzioni **N(s)** sottoinsieme di **S**.

La scelta di questa funzione è fondamentale per l'efficienza del sistema e definisce l'insieme delle soluzioni che possono essere raggiunte da **s** in un singolo passo di ricerca dell'algoritmo.

Può capitare in questo tipo di algoritmi, di trovarsi in situazioni di massimo (minimo) locale.

Un massimo locale è una soluzione **s** tale che data una funzione di valutazione **f**:

$$\forall s' \in N(s), f(s) \geq f(s')$$

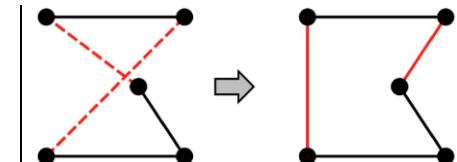
Quando risolviamo un problema di massimizzazione (minimizzazione) cerchiamo un max (min) globale s_{opt} , tale che:

$$\forall s, f(s_{opt}) \geq f(s)$$

Più è largo il vicinato più è probabile che un massimo locale sia anche un massimo globale. Quindi, più è largo un vicinato più aumenta la qualità della soluzione.

Esempio commesso viaggiatore:

Si parte con un percorso completo qualsiasi, e si eseguono scambi a coppie. Scambiamo i valori (archi), facendo sì che il percorso tocchi sempre tutti i nodi, ed avranno diversi valori di f. Dove la funzione f sarà la lunghezza del cammino.



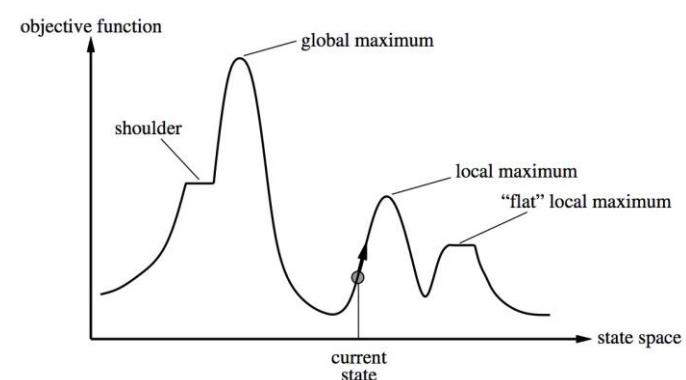
Gli algoritmi di ricerca locale sono efficienti dal punto di vista

dell'occupazione della memoria: tengono traccia solo dello stato corrente (non necessario un puntatore al padre) e si spostano su stati adiacenti.

Per problemi in cui:

- Il cammino che si segue non è importante: basta trovare la soluzione;
- Tutti gli elementi della soluzione sono nello stato ma alcuni vincoli sono violati.

Gli stati come punti su una superficie su cui l'algoritmo provoca movimento: i picchi sono massimi locali o soluzioni ottimali (se la funzione f è da ottimizzare).



HILL-CLIMBING:

Vengono generati i successori e valutati; viene scelto un nodo, che migliora la valutazione dello stato attuale:

- Il primo (salita semplice)
- Il migliore (salita rapida)
- Uno a caso (stocastico)

Se non c'è niente di meglio, l'algoritmo termina.

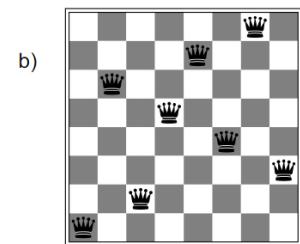
Esempio del problema delle 8-regine (formulazione a stato completo):

Funzione successore: muove una singola regina in un altro posto nella stessa colonna.

Funzione euristica $h(n)$: il numero di coppie di regine che si minacciano tra di loro.

*i numeri nelle caselle indicano il numero di vicini (regine che si possono minacciare)

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	13	16	13	16
14	14	17	15	14	16	16	16
17	14	16	18	15	15	15	16
18	14	15	15	14	14	16	16
14	14	13	17	12	14	12	18



- a) Mostra uno stato con $h=17$ ed il valore h per ogni possibile successore. Quanti successori sono?
- b) Un minimo locale nello spazio degli stati delle 8 regine ($h=1$), raggiunto dopo 5 passi

Problemi:

- **Massimi locali:** picco più alto degli stati vicini ma inferiore al massimo globale;
- **Cresta:** sequenza di massimi locali molto difficili da esplorare per gli algoritmi greedy (l'algoritmo va in loop, spostandosi tra i massimi sperando di salire);
- **Plateau:** un'area piatta del panorama dello spazio degli stati (funzione di valutazione piatta).

Un miglioramento per questo algoritmo è rappresentato dalle varianti.

MOSSA LATERALE:

Se si raggiunge un plateau (il successore ha lo stesso valore dello stato corrente) si potrebbe continuare la ricerca facendo una mossa laterale:

- Spostamento in uno stato con identico valore di h , sperando poi di trovare dei percorsi a salire, perché potrebbe non essere un massimo locale.

Bisogna evitare cicli, specialmente nel caso dei massimi (minimi) locali piatti. Una soluzione può essere quella di porre un limite massimo al numero consecutivo di mosse laterali.

VARIANTI DI HILL-CLIMBING:

- **Hill-climbing stocastico:** selezione casuale tra tutte le mosse che vanno verso l'alto. La probabilità della scelta può essere influenzata dalla "pendenza" delle mosse;
- **Hill-climbing con prima scelta:** come hill-climbing stocastico si generano casualmente i successori finché non se ne ottiene uno migliore, utile quando ci sono migliaia di successori;
- **Hill-climbing con riavvio casuale:** ripete le ricerche hill-climbing partendo da stati iniziali casuali fino a raggiungere un obiettivo. Se la probabilità di successo dell'algoritmo è p saranno necessari $1/p$ ripartenze per trovare la soluzione (nel caso delle 8 regine corrisponde a 7 iterazioni)

RICERCA LOCAL BEAM:

Si tiene traccia di k stati anziché di uno solo, ad ogni passo si generano i successori di tutti i k stati.

- Se si trova un goal ci si ferma;
- Altrimenti si prosegue con i k migliori, tra tutto l'insieme di tutti i vicini.

Vado ad analizzare i vicini di tutti i k stati, sto vedendo in parallelo su più stati. Ovviamente questo mi permette di arrivare prima all'obiettivo (insieme dei vicini più ampio). In una ricerca con riavvio casuale, ogni processo di ricerca viene eseguito indipendentemente dagli altri. In una ricerca local beam, vengono passate informazioni utili tra i thread di ricerca paralleli. In effetti, gli stati che generano i migliori successori dicono agli altri: "Vieni qui, l'erba è più verde!" L'algoritmo abbandona rapidamente le ricerche infruttuose e sposta le sue risorse dove si stanno facendo i maggiori progressi.

Nella variante **local beam stocastica**, si scelgono k successori a caso con probabilità maggiore per i migliori (**selezione naturale**). Terminologia: Organismo [stato] - Progenie [successori] - Fitness [il valore della f], idoneità.

ALGORITMI EVOLUTIVI:

Le configurazioni sono viste come individui di una popolazione, dove i meno adatti muoiono senza riprodursi mentre consente ai migliori di riprodursi più spesso. Ogni generazione dovrebbe essere nel complesso migliore rispetto alla precedente; quindi, aspettando abbastanza a lungo la popolazione dovrebbe evolvere verso individui migliori (vale a dire con valori massimi di f). Una prima variante è rappresentata dagli **algoritmi genetici**.

ALGORITMI GENETICI:

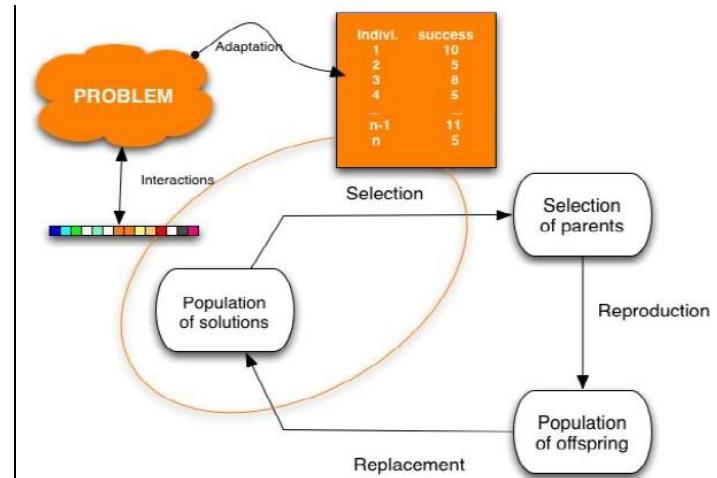
Si parte da una popolazione con k stati (lo stato è una soluzione al problema) generati casualmente, ogni individuo è rappresentato come stringa per poter applicare un algoritmo genetico (Es 24748552 stato delle 8 regine, prima regina - prima colonna/seconda riga, seconda regina - seconda colonna/quarta riga, ecc...).

Gli individui sono valutati da una **funzione di fitness** (numero di coppie di regine che non si attaccano). Si scelgono gli individui con probabilità proporzionale alla fitness.

- Variazione del local beam stocastico con ricombinazione:

È presente una fase di modellazione. Poi viene creata una popolazione casuale di soluzioni. Seleziono i migliori che verranno usati per la riproduzione (**Selection of parents**) producendo una nuova popolazione che verrà utilizzata nella successiva iterazione.

Popolazione iniziale: viene creata generando gli individui in maniera casuale. **Np** rappresenta il numero di individui generati e la dimensione della popolazione. Tale valore viene scelto in maniera euristica ed è dipendente dalla natura della funzione obiettivo e dalle dimensioni dello spazio di ricerca.



Negli algoritmi genetici standard **Np** rimane fisso durante l'evoluzione (ma in altre varianti può cambiare).

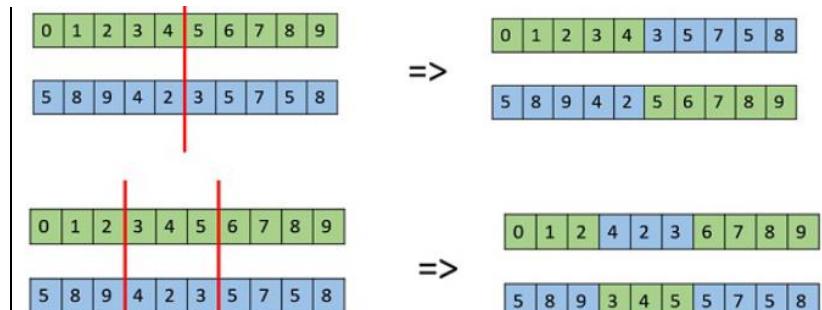
Fitness function: definisce la bontà dell'individuo (deve essere proporzionale alla bontà della soluzione). Nella scelta degli elementi genitori si tendono a favorire elementi con un elevato valore di fitness e utilizziamo come misura di fitness il valore della **funzione obiettivo f**.

Si potrebbero utilizzare altri valori di fitness (la **diversità** di un elemento rispetto agli altri membri della popolazione) che mi permetterebbero di esplorare zone dello spazio delle soluzioni differenti.

L'operatore di selezione: nel contesto degli algoritmi genetici, gli individui più forti sono quelli con fitness più alta, poiché risolvono meglio di altri il problema di ricerca dato; per questo essi devono essere privilegiati nella fase di selezione.

- **Roulette wheel:** la popolazione è rappresentata mediante una roulette con i settori proporzionali alla fitness degli elementi; i settori della roulette, sono più grandi per gli individui con valore di fitness più alto (quindi la pallina ha più probabilità di finire in queste caselle);
- **Tournament selection:** vengono scelti due individui a caso e quello tra i due che ha la fitness migliore viene selezionato per la nuova popolazione; l'operazione viene ripetuta N_p volte; prima della selezione gli individui vengono mescolati (shuffle).

Per poter generare nuovi individui, applico operatori generici. Il primo che usiamo è **crossover**, il quale per ogni coppia viene applicato con probabilità P_c . Si scelgono a caso due individui nel **mating pool (genitori)** e un punto di taglio (**punto di crossover**) su di essi. Le porzioni di genotipo alla destra del punto di crossover vengono scambiate generando due discendenti.



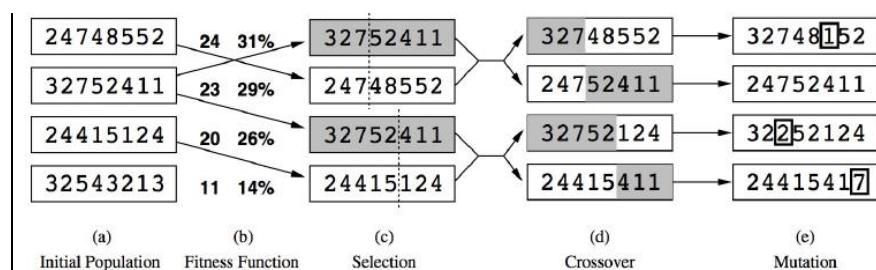
Per ogni figlio, la **mutation** viene applicata con probabilità P_m . L'operatore di crossover ricombina il materiale genetico esistente mentre l'operatore di mutation introduce nuovo materiale genetico. P_c e P_m si scelgono in maniera euristica e in genere $P_m < P_c$.

La **mutation** serve ad esplorare altri posti dello spazio di ricerca, aiutandoci nei problemi dei massimi locali.

Esempio:

Per ogni coppia viene scelto un punto di crossover e i due genitori producono due figli scambiandosi pezzi. Viene infine effettuata una mutazione casuale che dà luogo alla prossima generazione.

N.B. $24/(24+23+20+11) = 31\%$



CRITERIO DI ARRESTO:

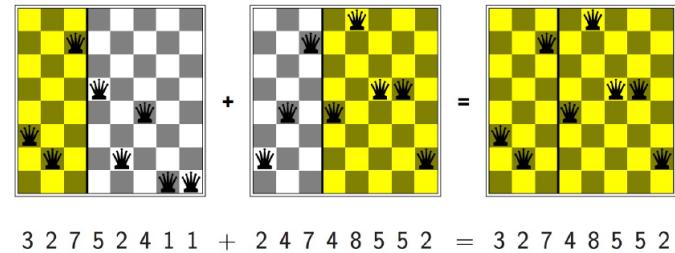
Il meccanismo di selezione, ricombinazione e calcolo della fitness viene iterato. L'evoluzione termina quando viene raggiunto l'ottimo (es. non ci sono regine in conflitto), se questo è noto.

Altrimenti l'evoluzione termina quando:

1. Viene raggiunto il numero massimo N_g di generazioni; il numero totale N_t di valutazioni della funzione obiettivo
 $N_t = N_p * N_g$;
2. Un indicatore di convergenza (uniformità della popolazione, mancanza di progressi nell'evoluzione) raggiunge un determinato valore.

Gli algoritmi genetici richiedono che gli stati siano codificati come stringhe.

Il crossover aiuta se e solo se le sottostinghe sono componenti significative.



Con il crossover possiamo balzare in un solo colpo in una zona molto lontana da entrambi gli elementi della coppia di punti combinati tra loro mentre nelle ricerche locali, negli algoritmi SA e nella mutazione stessa degli algoritmi genetici ci possiamo solo spostare in punti vicini.

In certi casi il processo di crossover fornisce dei grossi vantaggi. Questo succede in particolare quando il problema gode di una certa **separabilità** (punti della regione ammissibile decomponibili in parti indipendenti o quantomeno debolmente correlate tra loro).

4.2 RICERCA IN AMBITI PIÙ COMPLESSI

Azioni non deterministiche (esito dell'azione non certo) e **ambiente parzialmente osservabile**, possono essere risolti:

- Piani condizionali, ricerca AND-OR, stati credenza.

Ambienti sconosciuti e problemi di esplorazioni (percezioni forniscono nuove informazioni dopo l'azione, Ricerca online).

Negli agenti risolutori di problemi "classici" si assumono ambienti completamente osservabili e azioni/ambienti deterministicici. Il piano generato è una sequenza di azioni che può essere generato **offline** ed eseguito senza imprevisti (le percezioni non servono se non nello stato iniziale).

In un **ambiente parzialmente osservabile e non deterministico** le **percezioni** sono importanti poiché restringono gli stati possibili e informano sull'effetto dell'azione. Più che un piano, l'agente può elaborare una "strategia" che tiene conto delle diverse eventualità: **un piano con contingenza**.

Esempio aspirapolvere non deterministico:

Se aspira in una stanza sporca, la pulisce ma talvolta pulisce anche una stanza adiacente mentre se aspira in una stanza pulita, a volte rilascia sporco. Quindi sono necessarie delle variazioni al modello:

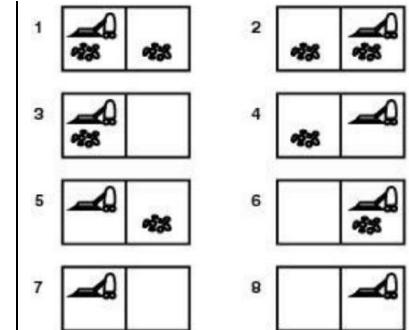
- Il modello di transizione restituisce un **insieme di stati**: l'agente non sa in quale si troverà;
- Il piano di **contingenza** sarà un piano condizionale e magari con cicli.

Esempio: Risultati (Aspira, 1) = {5, 7}

Piano possibile:

```
[Aspira,  
if stato=5  
    then [Destra, Aspira]  
    else []  
]
```

Da sequenza di azioni a piano (albero).



Fino ad ora abbiamo discusso strategie di ricerca per grafi OR nei quali vogliamo trovare un singolo cammino verso la soluzione. Il **grafo AND/OR** risulta appropriato quando si vogliono rappresentare problemi che si possono risolvere scomponendoli in un insieme di problemi più piccoli che andranno poi risolti.

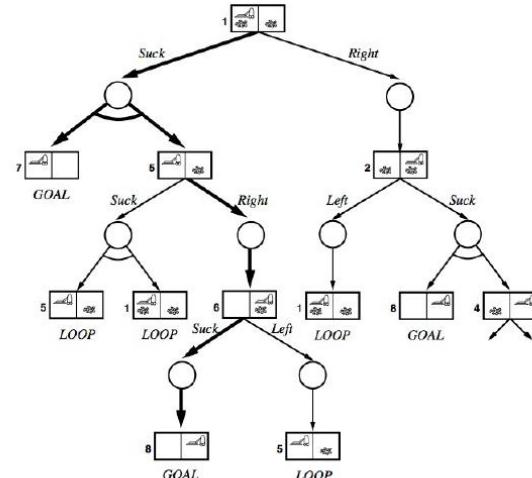
Nei problemi con azioni non deterministiche avremo un arco AND che deve puntare a un qualunque numero di nodi successori che si devono risolvere per risolvere il nodo AND stesso e dal nodo AND possono anche partire rami OR che indicano soluzioni alternative.

ALBERI DI RICERCA AND-OR:

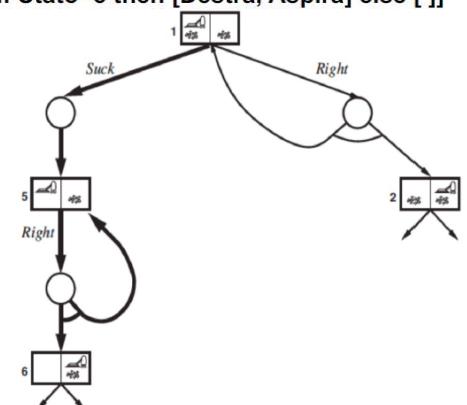
- Nel caso di **nodi OR**: l'agente sceglie 1 sola azione tra quelle possibili;
- Nel caso di **nodi AND**: l'agente deve tenere in considerazione tutti gli archi uscenti perché non si sa l'effetto dell'azione precedente (non determinismo). Le diverse contingenze (le scelte dell'ambiente) da considerare tutte;
- Una soluzione di ricerca a un problema di ricerca AND-OR è un albero che:
 - Ha un nodo obiettivo in ogni foglia;
 - Specifica un'unica azione nei nodi OR;
 - Include tutti gli archi uscenti dai nodi AND.

L'algoritmo di ricerca deve visitare l'albero e deve arrivare ad una soluzione in cui la foglia è un obiettivo e sul percorso tra la radice e la foglia deve tenere in considerazione tutti i nodi AND (i quali saranno degli IF).

I **nodi loop** sono dei nodi che ho già visto, poiché avendo azioni non deterministiche potrei fare più volte le stesse azioni sperando di arrivare all'obiettivo.



Piano: [Aspira, if Stato=5 then [Destra, Aspira] else []]



Piano: [Aspira, L₁: Destra, if Stato=5 then L₁ else Aspira]

Esempio aspirapolvere slittante:

Supponiamo che l'aspirapolvere quando si sposta può scivolare e rimanere nella stessa stanza ad esempio Risultati (Destra,1) = {1,2}. Sono necessarie delle variazioni, ovvero continuare a provare finché non va a destra. Il piano di contingenza potrà avere dei cicli.

Inoltre, bisogna distinguere tra:

- Osservabile e non deterministico (slittante)
- Non (o parzialmente) osservabile e deterministico (non so se la chiave aprirà la porta).

In questo secondo caso, se la chiave è sbagliata, si prova all'infinito ma niente cambia, il piano deve considerare anche questo aspetto.

RICERCA CON ASSENZA DI OSSERVAZIONI O CON OSSERVAZIONI PARZIALI:

Le **percezioni** non sono sufficienti a determinare lo stato esatto, anche se l'ambiente è deterministico.

- **Stato credenza**: un insieme di stati possibili in base alle conoscenze dell'agente;
- **Problemi senza sensori**;

Ma si possono trovare soluzioni anche senza affidarsi ai sensori ma utilizzando **stati-credenza**, l'algoritmo non lavora più sulla realtà ma sulle ipotesi dell'agente sull'ambiente.

AMBIENTE NON OSSERVABILE:

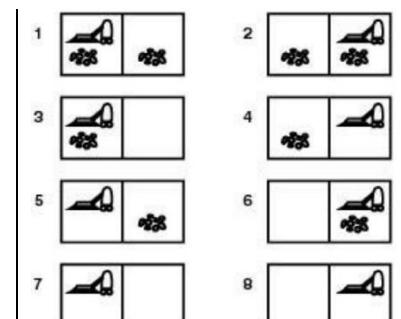
L'aspirapolvere non percepisce la sua locazione, né se la stanza è sporca o pulita e conosce la geografia del suo mondo e l'effetto delle azioni. Quindi l'agente deve tenere in considerazione tutte le possibili situazioni; quindi, all'inizio può trovarsi in uno degli 8 stati.

Stato iniziale = {1,2,3,4,5,6,7,8} ma l'esecuzione delle azioni riduce gli stati credenza. Quello che si fa è ricondurre un problema non osservabile ad un problema osservabile.

Esempio aspirapolvere senza sensori:

In questo esempio, se l'agente va a destra finisce in {2,4,6,8} ed ha guadagnato informazioni senza percepire nulla. Dopo aver effettuato [Right,Suck] finirà sempre in {4,8}. Finalmente, dopo [Right,Suck,Left,Suck] arriva nello stato obiettivo 7.

Nota: nello spazio degli stati credenza l'ambiente è **osservabile** (l'agente conosce le sue credenze). La soluzione (se c'è) sarà sempre una sequenza di azioni.



FORMULAZIONE DI PROBLEMI CON STATI-CREDENZA:

Se il problema originale ha N stati, il numero di possibili stati credenza è 2^N . Anche se molti potrebbero essere irraggiungibili dallo stato iniziale.

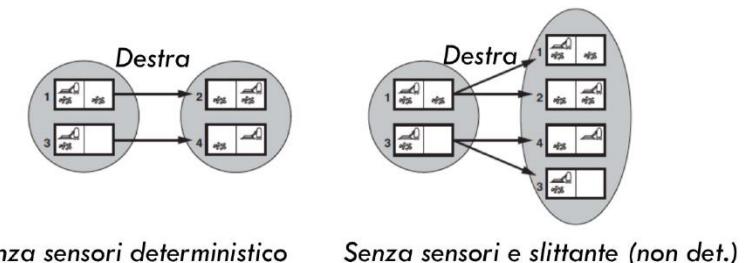
Stato-credenza iniziale $SC_0 \subseteq$ insieme di tutti gli N stati.

Azioni(b) = unione delle azioni lecite negli stati in b (ma se azioni illecite in uno stato hanno effetti dannosi meglio l'intersezione così da poter eseguire solo le azioni compatibili per tutti gli stati).

Modello di transizione rappresenta un modello in cui gli stati risultanti sono quelli ottenibili applicando le azioni a uno stato qualsiasi (l'unione degli stati ottenibili dai diversi stati possibili con le azioni eseguibili).

Test obiettivo: l'agente raggiunge l'obiettivo se tutti gli stati nello stato credenza soddisfano il test obiettivo.

Costo di cammino: il costo di eseguire un'azione potrebbe dipendere dallo stato, ma assumiamo di no.

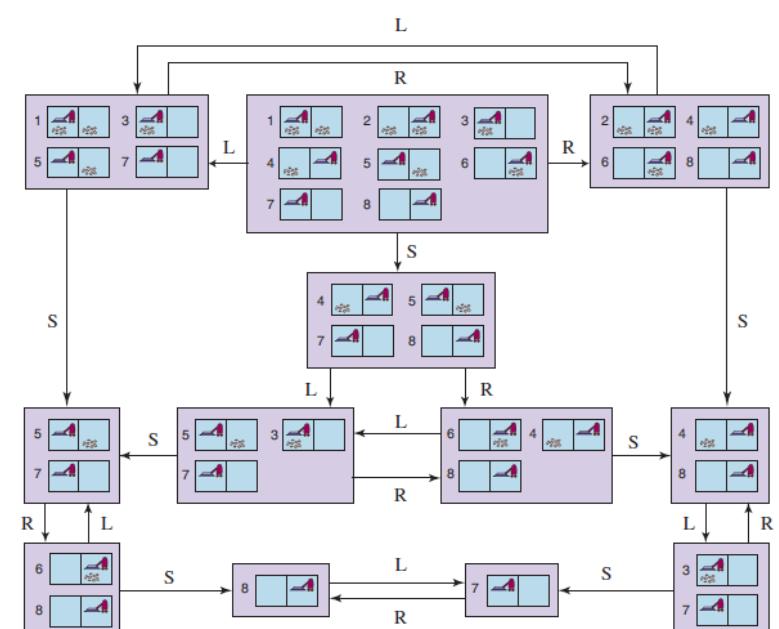


In questo tipo di problemi, il numero di stati credenza è molto alto, $2^8 = 256$ ma solo 12 raggiungibili, perché negli altri stati non è possibile arrivarci.

Ottimizzazioni che permettono di trascurare uno stato s se è stato già incontrato. Tramite una ricerca-grafo, generando s , e controllando se si è già incontrato uno stato credenza $s'=s$.

Inoltre, si può anche potare in modo più efficace in base al fatto che:

Se $s' \subseteq s$ allora ogni sequenza di azioni che è una soluzione per s lo è anche per s' . Quindi se $s' \subseteq s$, si può trascurare s , se $s \subseteq s'$ e da s' si è trovata una soluzione si può trascurare s .



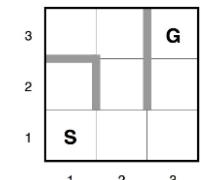
4.3 RICERCA ONLINE

Fino ad ora ci siamo focalizzati su tecniche di **Ricerca offline**, le quali computano una soluzione completa prima di effettuare la prima azione. Ora, analizzeremo le tecniche di **ricerca online**, dove l'agente alterna pianificazione e azione: effettua la prima azione, osserva l'ambiente e computa la successiva. Questo tipo di ricerca è utile in ambienti in cui non conosciamo nulla se non il funzionamento dell'agente.

1. Utile in ambienti dinamici o semi-dinamici (non c'è troppo tempo per pianificare);
2. Utile in ambienti non deterministici perché permette all'agente di concentrare i propri sforzi computazionali su contingenze che si presentano effettivamente piuttosto che su quelle che potrebbero non accadere mai;
3. Necessaria per **ambienti ignoti** tipici dei problemi di esplorazione.

Esempio labirinto con mappa e senza:

Con **mappa** (applicabili tutti gli algoritmi di ricerca visti) mentre **senza mappa** l'agente non può pianificare, può solo esplorare nel modo più razionale possibile e quindi usano una ricerca online.



ASSUNZIONI:

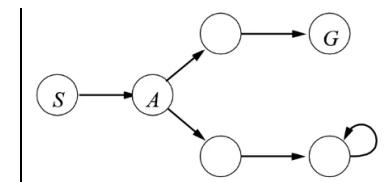
Un problema di ricerca online può essere risolto da un agente che esegue azioni, alternando computazione, sensoristica e azioni. Conoscenza:

- **AZIONI(s):** lista delle azioni permesse nello stato s ;
- **$c(s,a,s')$:** funzione di costo che può essere usata solo se s' è il risultato dell'azione (costo che si può usare dopo che l'azione ha prodotto il risultato);
- **Test-obiettivo(s):** funzione che verifica se lo stato raggiunto è lo stato obiettivo;
- **Risultato(s,a) non noto.**

Inoltre, l'agente potrebbe avere accesso ad una funzione euristica ammissibile $h(s)$ che stima la distanza dallo stato corrente s allo stato obiettivo.

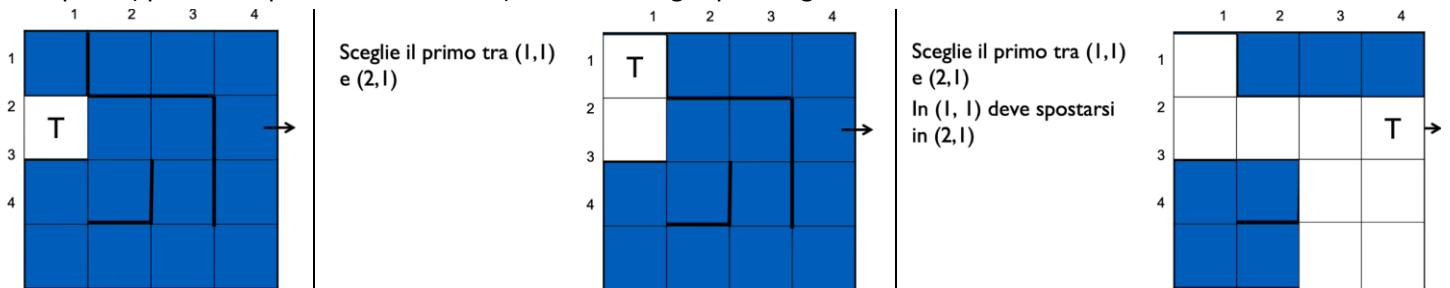
Per valutare un algoritmo online, possiamo confrontarlo solo col miglior algoritmo, quindi abbiamo il costo del cammino (quello effettivamente percorso) e il rapporto tra questo costo e quello ideale (conoscendo l'ambiente) è chiamato **rapporto di competitività**. Tale rapporto può essere infinito (identico se si comportano in maniera simile) e le prestazioni sono in funzione dello spazio degli stati.

Ambienti esplorabili in maniera sicura, dove è sempre possibile arrivare ad uno stato obiettivo, non esistono azioni irreversibili. Nessun algoritmo può evitare vicoli ciechi in tutti gli spazi degli stati.



RICERCA IN PROFONDITÀ ONLINE:

Gli **agenti online** ad ogni passo decidono l'azione da fare (non il piano) e la eseguono. Non possono espandere più stati e costruiscono una mappa dell'ambiente. Inoltre, è un metodo locale ed effettua **backtracking** se non ha più figli perché non può spostarsi da quella posizione. Ovviamente man mano che si effettua la visita salviamo ciò che abbiamo scoperto in una tabella, che elenca, gli stati predecessori nei quali l'agente ancora non è tornato indietro, se ha terminato gli stati la ricerca è completa (quello che producono le azioni). Funziona negli spazi degli stati dove le azioni sono reversibili.



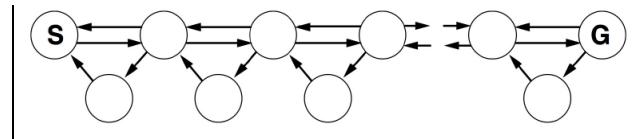
Bisogna tener traccia di alcune informazioni: devo tenere traccia di alcune informazioni, infatti quando arrivo in un nuovo stato devo salvare le azioni che ho fatto e quelle che devo ancora fare (**untried**), così nel caso in cui torno dal figlio al padre a causa di fallimenti possiamo provare l'azione successiva. Quando untried finisce vuol dire che ho provato tutte le azioni.

RICERCA LOCALE ONLINE:

Hill-Climbing già è un **algoritmo online** poiché mantiene solo lo stato corrente in memoria ma sfortunatamente si potrebbe bloccare in un massimo/minimo locale. Per evitare massimi/minimi locali, si può usare la tecnica della mossa laterale ma non è possibile sfuggire a questo problema.

Abbiamo due soluzioni:

- **Random-walk:** anziché scegliere in funzione delle euristiche, effettuiamo una camminata casuale (azione a caso) quindi alla fine esploro tutto l'ambiente e le performance dipendono da come è definito l'ambiente.
- **Apprendimento Real-Time (LRTA*):** esplorando si aggiustano i valori dell'euristica per renderli più realistici:
 - Se s è uno stato con f. euristica $h(s)$, i successori sono valutati $h(s)$ se inesplorati, altrimenti si prende la loro stima $H+$ costo azione per raggiungerli
 - **Costo LRTA*** $(s,a,s',H)=\begin{cases} h(s) & \text{se } s' \text{ non esplorato} \\ H(s') + \text{costo}(s,a,s') & \text{altrimenti} \end{cases}$. Aggiorno i costi in funzione dell'euristica se lo stato che sto esplorando non lo conosco.



5. RICERCA CON AVVERSARI

Ambienti **multi-agente**: dobbiamo considerare le azioni degli altri agenti e i loro effetti;

Ambienti **competitivi**: gli obiettivi degli agenti sono in conflitto.

Giochi più comuni in AI: **giochi a somma zero con informazione perfetta a turni e a due giocatori**:

- Ambienti deterministici, completamente osservabili;
- Valori di utilità uguali ma di segno opposti alla fine (la somma è sempre 1, 1 = vittoria - 0 = sconfitta - $\frac{1}{2}$ = pareggio).

GIOCHI CON AVVERSARIO:

Regole semplici e formalizzabili, abbiamo un ambiente multi-agente dove la presenza dell'avversario rende il problema **contingente** → più difficile rispetto ai problemi di ricerca visti fino ad ora. Non possiamo risolvere questi problemi con alberi AND-OR, perché in un certo istante possono esserci tante azioni e dovrei fare un **IF** per tutti i possibili esiti per ognuna di queste azioni.

Inoltre, presentano complessità e vincoli di tempo reale: si può solo cercare di fare la mossa migliore nel tempo disponibile.

TIPI DI GIOCHI:

1. **Giochi deterministici**, caso più semplice, bisogna scegliere tra un insieme di possibili azioni;
2. **Giochi basati su chance**, sono basati su probabilità (es. lancio del dado);
3. **Informazione perfetta**, abbiamo tutta la conoscenza del gioco (es. scacchi);

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information		bridge, poker, scrabble nuclear war

Quello che si vuole fare è andare a definire un algoritmo di ricerca che fornisce l'azione ottima che il giocatore può fare.

Inoltre, sono degli algoritmi onerosi poiché lo spazio degli stati è molto ampio quindi cercheremo di ridurre la complessità tramite **tecniche di riduzione (pruning)** dello spazio degli stati e viene fatta un'approssimazione, fornendo una soluzione approssimata.

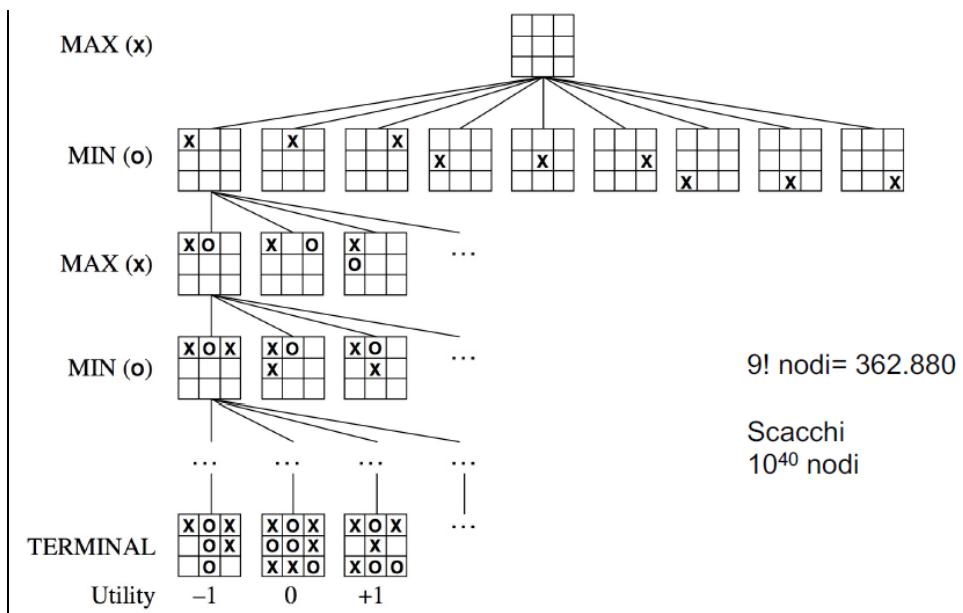
5.1 GIOCHI COME PROBLEMI DI RICERCA

Abbiamo due giocatori **MAX** e **MIN**, MAX muove per primo. La configurazione iniziale del gioco è lo **stato iniziale s**.

- **Giocatore(s)** rappresenta il giocatore a cui tocca muovere nello stato s .
- **Azioni (s)** indica l'insieme delle mosse lecite nello stato s .
- **Risultato(s,a)** indica il risultato della mossa(azione) a nello stato s .
- **Test-Terminazione(s)** è vero se la partita è finita (stato terminale).
- **Utilità (s,p)** rappresenta la funzione di utilità (o obiettivo) valore per giocatore p in stato terminale s .
- **Albero di gioco**: stato iniziale, azioni e risultato (nodi=Stati e archi=mosse) definiscono il **grafo dello spazio degli stati**. Un grafo dove i vertici sono gli stati, gli archi sono le mosse e lo stato può essere raggiunto con diversi cammini.

Esempio gioco del tris:

- Due giocatori: MAX e MIN;
- Nello stato iniziale, MAX ha nove possibili mosse;
- MAX mette una X e MIN mette una O;
- Finché non si raggiunge un nodo foglia (stato terminale): un giocatore ha fatto tris o tutte le caselle sono riempite;
- Per le foglie, valore di utilità (per MAX): buono per MAX e cattivo per MIN.



MINIMAX:

L'assunzione che fa questo algoritmo è che l'avversario giochi in maniera perfetta. Utile per giochi **deterministici e ad informazione perfetta** (scacchi).

L'idea è di scegliere la mossa che conduce alla posizione con valore **minimax** (calcolato per ogni nodo associato sia a nodi min che a nodi max) più alto = migliore vantaggio raggiungibile contro un avversario che gioca in modo ottimo.

Non dobbiamo trovare la strategia ottima per arrivare all'obiettivo e MAX deve prendere MIN in considerazione.

MINIMAX(n): utilità per MAX in nodo n, assumendo che i due agenti giochino in modo ottimo da lì alla fine. Max va verso valori alti mentre MIN verso valori bassi.

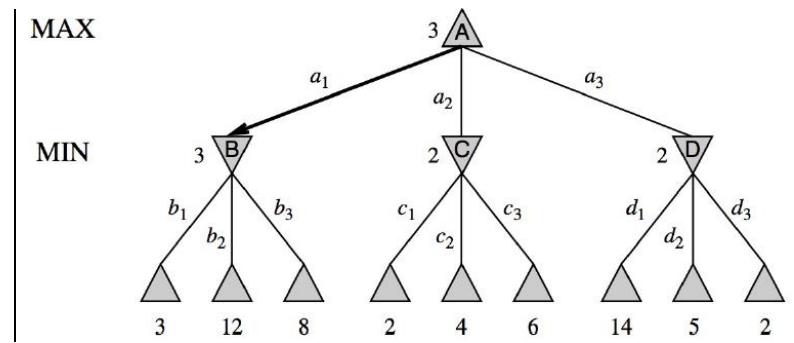
MINIMAX(n):

- **Utility(s,MAX) se TEST-TERMINALE(s);**
- Per MAX: $\max_{a \in \text{azioni}(s)} \text{VALORE-MINIMAX-RISULTATO}(s, a)$: prendo tra i vari nodi figli, devo scegliere l'azione che mi porta al nodo min con valore MINIMAX più alto;
- Per MIN: $\min_{a \in \text{azioni}(s)} \text{VALORE-MINIMAX-RISULTATO}(s, a)$ min fa il contrario, prende il valore più piccolo possibile.

Massimizza il risultato di MAX nel caso pessimo (e minimizza il risultato di MIN nel caso pessimo). Per calcolare questi valori bisogna fare l'esplorazione completa in profondità dell'albero di gioco, siccome i valori sono costruiti dalle foglie verso la radice.

Esempio minimax a due livelli:

Max va a considerare min come miglior giocatore possibile, il quale prenderà rispettivamente 3,2,2. Ora in questa situazione a MAX conviene arrivare in 3 così da garantirsi tale valore. Per calcolare i valori, prendo il minimo tra tutti i suoi figli e poi prendo il max tra tutti i figli. Risalendo dalle foglie.



È un algoritmo ricorsivo che procede fino alle foglie dell'albero e quindi esegue il backup dei valori minimax attraverso l'albero mentre la ricorsione si svolge.

L'algoritmo è **completo** se l'albero è finito, **ottimo** contro avversario ottimo altrimenti non ci dà la soluzione ottima ma si comporta comunque bene.

Se la profondità massima dell'albero è m e ci sono b mosse legali ad ogni punto, allora la complessità di tempo $O(b^m)$ mentre complessità di spazio $O(bm)$ per un algoritmo che genera azioni tutte in una volta.

Il problema è il tempo, ad esempio per il gioco degli scacchi $b \approx 35$, $m \approx 80 \rightarrow 10^{123}$

POTATURA ALFA-BETA:

Poiché il numero di stati è esponenziale nella profondità dell'albero, possiamo utilizzare la **potatura alfa-beta (alfa beta pruning)** che ci permette di ridurre il numero di nodi esaminati da MINIMAX, infatti possiamo dimezzare l'esponente.

Potiamo alcuni rami dell'albero, che non ci servono per calcolare i valori minimax.

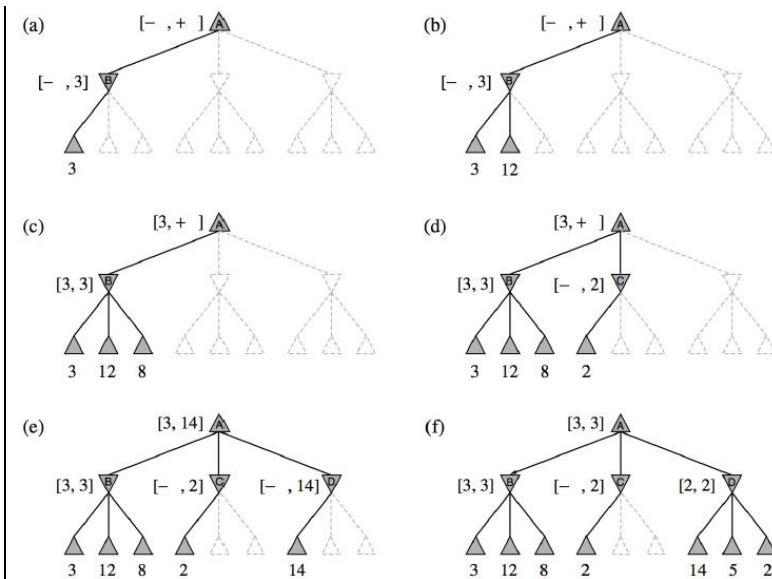
- α : valore della scelta migliore per MAX che abbiamo trovato fino a quel momento in un qualunque punto di scelta lungo il cammino, $\alpha = \text{almeno}$;
- β : lo stesso per MIN $\beta = \text{al massimo}$.

Aggiorniamo quindi tali valori potando i rami restanti che escono da un nodo non appena ci accorgiamo che il valore del nodo è peggio di α (per MAX) o di β (per MIN).

$$\begin{aligned}
 \text{MINIMAX}(\text{root}) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\
 &= \max(3, \min(2, x, y), 2) \\
 &= \max(3, z, 2) \quad \text{where } z = \min(2, x, y) \leq 2 \\
 &= 3.
 \end{aligned}$$

Tale tecnica non modifica il risultato finale, ed un buon ordinamento delle mosse (impatta sulle performance) migliora l'efficacia della potatura.

Con "ordine perfetto" (guardo prima i figli più promettenti), **complessità di tempo** = $O(b^{m/2})$ quindi si raddoppia la profondità di ricerca e si può facilmente raggiungere profondità (doppia) 8 e giocare bene a scacchi.



LIMITI ALLE RISORSE:

Supponiamo di avere a disposizione 100 secondi per mossa, e di poter esplorare 10^4 nodi al secondo, lui sarà in grado di visitare 10^6 nodi per mossa. Anche la potatura potrebbe risultare non sufficiente.

Approccio standard:

- **Test di taglio (cutoff)**: limite alla profondità;
- **Funzione di valutazione (Eval)**: stima del guadagno atteso in una certa posizione.

FUNZIONI DI VALUTAZIONE:

Proprietà:

1. Deve ordinare gli stati terminali come la vera funzione di utilità (+1, -1, ½);
2. Non deve richiedere troppo tempo;
3. Per stati non terminali, forte correlazione con la probabilità di vincere (incertezza perché ricerca interrotta prima di guardare tutto l'albero).

Molte funzioni di valutazioni sono spesso basate sulle **caratteristiche** di uno stato (es numero di pedoni bianchi, alfieri bianchi ecc.). Queste caratteristiche messe insieme ci forniscono le **classi di equivalenza di stati**, dove ogni classe può contenere stati che portano alla vittoria, al pareggio e alla sconfitta. La funzione di valutazione riflette la proporzione di stati che portano ad un certo risultato.

Esempio scacchi:

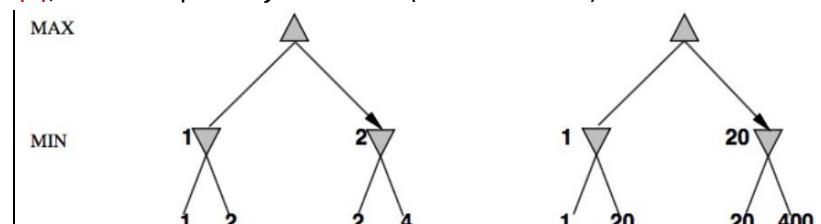
72% porta a vittoria (+1), 20% sconfitta (-1) 8% pareggio (1/2) → **valore atteso** = $(0.72 \cdot 1) + (0.20 \cdot 0) + (0.08 \cdot 1/2) = 0.76$.

Anche se non calcola esattamente il valore atteso, almeno non deve alterare l'ordinamento tra stati (questo rappresenta un vantaggio perché riesce a dare l'ordine giusto ovvero una mossa migliore rispetto ad un'altra), purtroppo potremmo avere troppe classi di equivalenza ma possiamo avere un valore per ogni caratteristica e poi combinarli (tramite combinazione lineare):

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s), \text{ dove } w = \text{peso} \text{ e } f = \text{features (caratteristiche).}$$

Il comportamento corretto è preservato per ogni trasformazione monotona di EVAL, quello che conta è solo l'ordine:

Il guadagno in giochi deterministici agisce come una funzione di **utilità ordinale**.



MINIMAX CON DECISIONE IMPERFETTA:

La strategia è quella di guardare avanti k mosse, *si espande l'albero di ricerca un certo numero di livelli k* (compatibile col tempo e lo spazio disponibili), *si valutano gli stati ottenuti e si propaga indietro il risultato con la regola del MAX e MIN*:

$$H\text{-MINIMAX}(s, d) =$$

$$\begin{cases} EVAL(s) & \text{if } CUTOFF\text{-TEST}(s, d) \\ \max_{a \in Actions(s)} H\text{-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} H\text{-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if } \text{PLAYER}(s) = \text{MIN}. \end{cases}$$

(dove d è la profondità)

Quando è vera la condizione applico la funzione di valutazione EVAL, successivamente i nodi interni saranno aggiornati come in precedenza max per il giocatore max e min per il giocatore min. Inoltre, il valore della profondità (d) viene dato in input dall'utente e dipende dalle performance che ha la macchina.

La funzione di valutazione EVAL è una stima della utilità attesa a partire da una certa posizione nel gioco. Le performance dell'algoritmo dipendono dal parametro k che diamo in input.

Supponiamo di considerare $b^m=10^6$, $b=35 \rightarrow m=4$, quindi il nodo MAX deciderà in base all'eval.

Per riuscire ad avere buone performance dovrebbe arrivare a valutare 8 strati. L'algoritmo, quindi, funziona bene se la funzione di valutazione è buona e se si riesce a scendere di molti livelli.

Altri miglioramenti possono essere:

- **Potatura in avanti (chiamate di tipo B)**, esplorare solo alcune mosse ritenute promettenti e tagliare le altre:
 - **Beam search (solo le prime k, con eval più alta) ma rischioso** perché se le soluzioni sono sulle parti tagliate perdo l'obiettivo, infatti sono tecniche di approssimazione;
 - **Tagli probabilistici (basati su esperienza)** che riducono il rischio che le mosse migliori vengano potate, miglioramenti notevoli ad alfa-beta in Logistello.
- **Database di mosse di apertura e chiusura**:
 - Nelle prime fasi ci sono poche mosse sensate e ben studiate, inutile esplorarle tutte;
 - Per le fasi finali il computer può esplorare off-line in maniera esaustiva e ricordarsi le migliori chiusure. Applicando direttamente queste mosse, invece di effettuare una ricerca sull'albero.

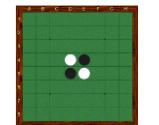
Tecniche per risparmiare tempo.

GIOCHI DETERMINISTICI IN PRATICA:

Dama: chinook ha terminato il dominio durato 40 anni del campione mondiale Marion Tinsley nel 1994. Ha utilizzato un database che definiva tutte le mosse ottime a partire da tutte le posizioni della scacchiera con 8 o meno pezzi: totale di 443.748.401.247 posizioni. Dal 2007 gioca in modo perfetto utilizzando la ricerca alpha-beta e un DB contenente 39 mila miliardi di posizioni finali.

Scacchi: Deep Blue ha battuto il campione mondiale Cary Kasparov nel 1997 in un incontro a sei partite. Deep Blue è in grado di esplorare 200 milioni di posizioni al secondo, usa una funzione di valutazione molto sofisticata e altri metodi segreti per estendere la ricerca fino a 40 strati.

Othello: campioni umani si rifiutano di giocare contro computer, perché questi ultimi sono troppo bravi.



Go: "campioni umani si rifiutano di giocare contro computer, perché questi ultimi sono dei veri novizi. In Go, $b > 300$, e quindi la maggior parte dei programmi usa basi di conoscenza su possibili situazioni per decidere le mosse da fare".

"oggi i migliori programmi giocano la maggior parte delle mosse a livello dei maestri; l'unico problema è che nel corso di una partita solitamente commettono almeno uno sbaglio grossolano che consente a un avversario forte di vincere".

RICERCA AD ALBERO DI MONTE CARLO:

La ricerca alfa-beta non è utilizzabile per il gioco del Go dove abbiamo un fattore di ramificazione >361 (visitata limitata a 4-5 strati con MINIMAX) oppure quando è difficile definire una buona funzione di valutazione.

Questo ha portato all'utilizzo della strategia di **ricerca ad albero di Monte Carlo (MTCS)** che stima il valore di uno stato s come utilità media su un certo numero di **simulazioni** di partite complete che iniziano da s.

Una simulazione sceglie prima le mosse per un giocatore poi per l'altro, fino ad arrivare ad uno stato terminale. Per i giochi con vittoria e sconfitta l'utilità media coincide con la percentuale di vittoria.

Le **mosse scelte durante la simulazione**, non avvengono in maniera casuale, ma viene usata una **politica di simulazione**, la quale deve essere in grado di far scegliere una mossa vincente per il giocatore, la scelta deve ricadere verso le migliori.

Le politiche di simulazione sono state apprese facendo giocare il programma contro sé stesso e con l'uso di **reti neurali** e si cerca di capire le mosse migliori che sono state effettuate.

Una volta ottenuta una politica di simulazione, bisogna decidere due cose:

1. *Da quali posizioni iniziare le simulazioni?*
2. *Quante esecuzioni effettuare per ogni posizione?*

Viene utilizzata la **Ricerca Monte Carlo pura**: N simulazioni dallo stato corrente e si determina quale delle mosse possibili nella posizione corrente ha la più alta percentuale di vittoria.

Convergenza al gioco ottimo al crescere di N, più simulazioni facciamo maggiore è la probabilità di vittoria, quindi si avvicina al valore reale. Però non è sufficiente per molti giochi quindi dobbiamo introdurre la **politica di selezione**, la quale concentra selettivamente le risorse computazionali sulle parti importanti dell'albero di gioco.

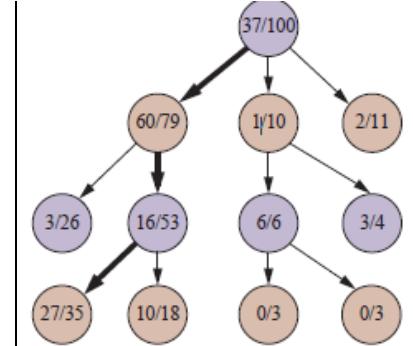
Abbiamo un numero ridotto di simulazioni, poiché non possiamo farne N dobbiamo effettuare un bilanciamento a due fattori:

- **Esplorazione** di stati per cui sono state fatte poche simulazioni. La simulazione dovrebbe coinvolgere i nodi che hanno poche simulazioni;
- **Sfruttamento** di stati in cui le precedenti simulazioni hanno prodotto buoni risultati (per avere stime più precise) coinvolgendo nodi che hanno le migliori performance.

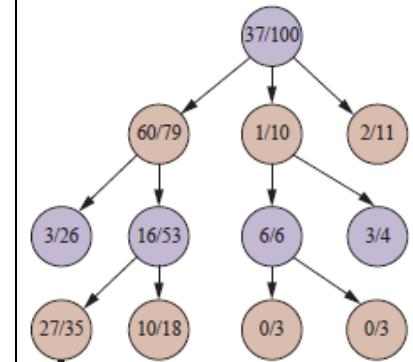
Quindi la politica di selezione deve considerare entrambi questi fattori in maniera tale da evitare sbagli ed evitare di perdere le mosse migliori.

L'algoritmo fa crescere l'albero di ricerca ad ogni iterazione con le operazioni fino a quando non arriviamo al limite di iterazioni o scade il tempo a disposizione:

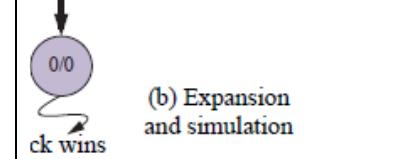
Selezione: scegliere il nodo che dobbiamo selezionare. In ogni nodo teniamo traccia del numero di vittorie e sconfitte (37 vittorie su 100 simulazioni). Quindi avremo uno sfruttamento se selezioniamo uno stato con ad esempio 27/35 vittorie mentre usiamo l'esplorazione se si sceglie uno stato con poche simulazioni. La selezione trova un percorso dalla radice alla foglia;



Espansione: facciamo crescere l'albero generando un nuovo figlio del nodo selezionato;



Simulazione: eseguiamo una simulazione dal nodo figlio generato scegliendo le mosse dei giocatori secondo la politica di simulazione;



Retro propagazione: aggiorniamo l'albero, quindi il risultato della simulazione viene usato per aggiornare tutti i nodi dell'albero di ricerca risalendo fino alla radice.

Tutto ciò è lineare nella profondità, e come abbiamo detto il processo si ripete N volte oppure fino alla fine del tempo a disposizione. Una volta terminato il ciclo, l'algoritmo fornisce in output la mossa con il più alto numero di simulazioni. Nell'esempio viene restituito 80, perché? Ad occhio la scelta dovrebbe essere quella con il rapporto maggiore (percentuale di vittoria) ma in realtà sceglie quelle col maggior numero di simulazioni.

Questo perché con le politiche definite, di solito il numero maggiore di simulazioni coincide con quelle con più probabilità di vittoria e inoltre se c'è qualcuno con il miglior rapporto, un numero inferiore di simulazioni può rappresentare più incertezza. Un aspetto importante è la **politica di selezione**.

CONFRONTO:

Il tempo per il calcolo di una simulazione è lineare nella profondità dell'albero di gioco. Se un gioco ha $b=32$ e $m=100$, con un miliardo di stati:

- Minmax cerca fino a profondità 6;
- Alfa-beta con ordinamento delle mosse fino a 12;
- Montecarlo potrebbe fare 10 milioni di simulazioni.

Le performance dipendono dall'accuratezza della funzione euristica rispetto alle politiche di selezione e simulazione.

Se la funzione di valutazione è imprecisa, la ricerca alfa-beta sarà imprecisa, l'errore in un nodo può portare ad una scelta errata.

La ricerca Montecarlo, al contrario, si basa sull'aggregato di simulazioni, quindi, ***non è vulnerabile al singolo errore***, può essere usata anche per nuovi giochi in cui non si ha esperienza per definire una funzione di valutazione. Le politiche di selezione e simulazione possono essere apprese usando reti neurali addestrate facendo giocare il programma contro sé stesso.

Però potrebbero esserci dei ***rischi*** con la ***ricerca Montecarlo***:

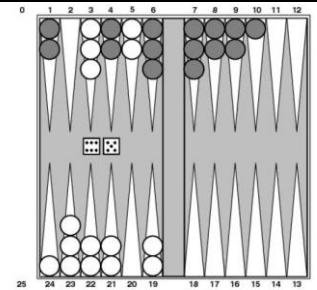
- Una linea di gioco importante non venga mai esplorata;
- In una posizione critica contro un giocatore esperto, può esserci un solo ramo che porta a una sconfitta. Poiché questo non è facilmente identificabile a caso, la ricerca potrebbe non vederlo e non ne terrà conto.
- In sostanza, la ricerca tenta di tagliare le sequenze meno rilevanti

5.2 GIOCHI CON CASUALITÀ (STOCASTICI)

I giochi ***stocastici*** ci avvicinano un po' all'imprevedibilità della vita reale includendo un elemento casuale, ad esempio in giochi in cui è previsto un lancio di dadi, il problema diventa più complesso se consideriamo ad esempio il gioco del backgammon. In questo gioco, il giocatore deve lanciare i dadi e in base ai risultati si hanno le mosse possibili.

L'idea è cercare di risolvere questo tipo di gioco con l'algoritmo ***minmax***.

Lancio dadi 6-5,
4 mosse legali
per il bianco:
(5-10, 5-11)
(5-11, 19-24)
(5-10, 10-16)
(5-11, 11-16)



Per modellare l'albero di gioco, dobbiamo estendere l'albero perché tra max e min non abbiamo un arco azione come prima ma il lancio del dado e le possibili azioni.

Infatti, introduciamo accanto ai nodi scelta, i nodi ***possibilità (chance, i quali indicano le possibili azioni)*** e nel calcolare il valore dei nodi MAX e MIN adesso dobbiamo tenere conto delle probabilità dell'esperimento casuale. Inoltre, si devono calcolare il valore massimo e minimo ***attesi*** di una posizione (il valore medio atteso complessivo dei possibili esiti dei nodi chance).

REMINDER SULLA PROBABILITÀ:

Una ***variabile casuale*** rappresenta un evento il cui risultato è sconosciuto mentre una distribuzione di probabilità è un'assegnazione dei pesi agli esiti.

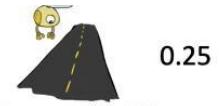
Il ***valore atteso*** di una variabile casuale è la media ponderata della distribuzione di probabilità sui risultati.

Quanto tempo per arrivare all'aeroporto?

Tempo:	20 min	+	30 min	+	60 min	→	35 min
Probabilità:	x 0.25		x 0.50		x 0.25		

Esempio: Traffico su autostrada

- ▶ Variabile casuale: $T = \text{se c'è traffico}$
- ▶ Risultati: $T \in \{\text{nessuno}, \text{leggero}, \text{pesante}\}$
- ▶ Distribuzione: $P(T=\text{nessuno}) = 0.25, P(T=\text{leggero}) = 0.50, P(T=\text{pesante}) = 0.25$



Alcune leggi sulla probabilità:

- ▶ Le probabilità sono sempre non negative
- ▶ La somma delle probabilità di tutti i possibili risultati è 1



Le probabilità possono cambiare:

- ▶ $P(T=\text{pesante}) = 0.25, P(T=\text{pesante} | \text{ora}=8\text{am}) = 0.60$



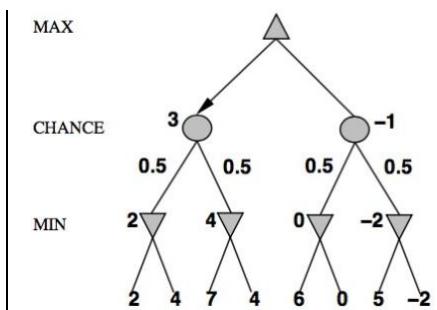
MINIMAX CON NODI POSSIBILITÀ:

Esempio con lancio moneta, infatti MAX calcolerà il suo valore con $(2*0.5+4*0.5)=3$.

Questo rappresenta il modo in cui minmax viene adatto a problemi in cui c'è l'introduzione dell'incertezza dovuta ad un evento probabilistico.

Non abbiamo più valori Minimax, ma solo valori attesi (media su tutti i possibili risultati dei nodi di casualità).

Con il lancio di un dado a 6 facce ci saranno molti più archi uscenti.



EXPECTIMINMAX – LA REGOLA:

I nodi terminali e i nodi MAX e MIN lavorano come prima con la differenza che le mosse legali per MAX e MIN dipenderanno dall'esito del lancio dei dati nel nodo chance precedente.

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \\ \sum_r P(r) \text{EXPECTIMINIMAX}(\text{RESULT}(s, r)) & \text{if } \text{PLAYER}(s) = \text{CHANCE} \end{cases}$$

Backgammon $b \approx 20$ (fino a 4000 con risultato uguale, e.g. 1-1, poiché il gioco consente di muovere 4 volte invece di 2) profondità $d=4$, quindi $20^*(21*20)^3 \approx 1.2*10^9$. Con l'aumentare della profondità, la probabilità di raggiungere un dato nodo si riduce (*lookahead si riduce*). In questo caso la potatura Alfa-beta è molto meno efficace.

TDGammon (minmax per backgammon) usa una ricerca di profondità 2+ una funzione Eval molto buona (basata su reti neurali) con un livello simile ad un campione mondiale.

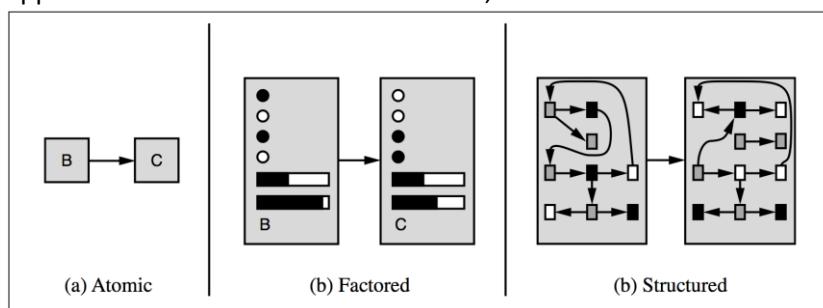
5.3 INFORMAZIONI IMPERFETTE

Oltre ad avere eventi probabilistici, abbiamo anche **informazione imperfetta** (non sappiamo tutto) in tutti i giochi a carte coperte (poker, bridge). Parliamo di giochi parzialmente osservabili poiché conosciamo le nostre carte ma non quelle degli avversari.

Tipicamente si può calcolare una probabilità per ogni possibile ordine delle carte, come se avessimo un dado con tantissime facce all'inizio del gioco. L'idea è quella di calcolare il valore minimax di ogni azione in ogni ordine delle carte, quindi scegliere l'azione con il valore aspettato più alto.

6. PROBLEMI DI SODDISFACIMENTO VINCOLI (CSP)

- **Approccio atomic:** rappresentano lo stato senza nessun'altra informazione;
- **Approccio factored:** stati più complessi che presentano informazioni ed aiutano gli algoritmi a trovare una soluzione in maniera molto più rapida. Vengono considerati come attributi con i possibili valori (algoritmi CSP);
- **Approccio structured:** gli attributi all'interno dello stato sono in relazione tra di loro, usato da logica del primo ordine, modelli probabilistici e apprendimento basato sulla conoscenza,



I problemi di soddisfacimento dei vincoli sono problemi con una struttura particolare, per cui conviene pensare ad algoritmi specializzati. La classe di problemi formulabili, in questo modo, è piuttosto ampia: layout di circuiti, scheduling, ecc... Esistono euristiche generali che si applicano e che consentono la risoluzione di problemi significativi per questa classe.

FORMULAZIONE DI PROBLEMI CSP:

Per poter andare a definire un **problema CSP** dobbiamo avere a disposizione:

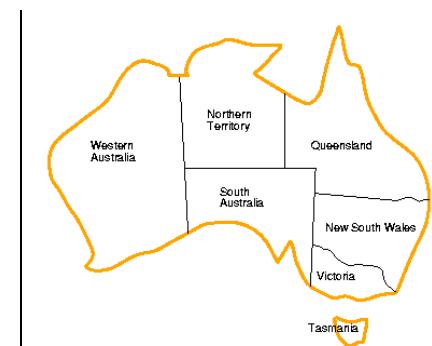
- **Insieme di variabili** X_1, X_2, \dots, X_n (sono le incognite del problema) alle quali assegniamo dei valori per risolvere il problema, ognuna di esse ha un dominio associato;
- **Dominio** D_1, D_2, \dots, D_n . Un dominio D_i consiste di un insieme di valori ammissibili per le variabili X_i . Una volta fatto ciò dobbiamo determinare i valori da assegnare a queste variabili affinché i vincoli siano soddisfatti;
- Un **insieme di vincoli** sui valori di queste variabili possa essere soddisfatto C_1, C_2, \dots, C_m .

Un modello di un CSP è un assegnamento di valori a variabili che soddisfa tutti i vincoli (es $X_1 < X_2$):

- **Stato:** assegnamento parziale di valori a variabili $\{X_i = v_i, X_j = v_j\}$;
- **Stato iniziale:** {} si parte dallo stato iniziale e ci spostiamo con gli archi tra gli stati fino ad una soluzione;
- **Azioni:** assegnamento di un valore a una variabile;
- **Soluzione (goal test):** assegnamento completo (le variabili hanno tutte un valore) e **consistente** (vincoli soddisfatti).

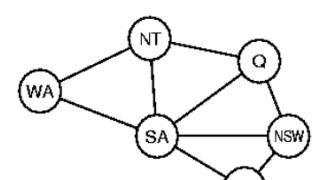
Esempio colorazione di una mappa:

- Variabili: WA, NT, Q, NSW, V, SA, T
- Domini $D_i = \{\text{red, green, blue}\}$
- Vincoli: regioni adiacenti devono avere colori differenti cioè:
 - $WA \neq NT$ (se il linguaggio lo permette), oppure
 - $(WA, NT) \in \{\text{(red, green), (red, blue), (green, red), (green, blue), ...}\}$



Possiamo rappresentare questo problema con un grafo dei vincoli:

Disegniamo un nodo per ogni variabile e un arco per ogni vincolo tra le due variabili.



Le soluzioni sono assegnazioni che soddisfano tutti i vincoli, cioè:

{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}



TIPI DI PROBLEMI CSP:

Il tipo di problema può essere di diversa natura (ci sono varie classificazioni di problemi).

Le variabili nei problemi possono essere:

- **Variabili discrete** con domini finiti o infiniti (CSP booleani con vero o falso);
- **Variabili con domini continui** (programmazione lineare) vincoli espressi come uguaglianze o disuguaglianze lineari.

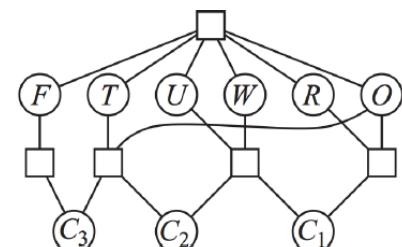
I possibili vincoli possono essere:

- Unari (x pari);
- Binari ($x > y$);
- Di grado superiore ($x+y=z$);
- Assoluti, devono essere soddisfatti tutti i vincoli;
- Preferenza, si preferiscono alcuni vincoli rispetto ad altri.

Esempio giochi enigmistici:

- Ogni lettera = cifra diversa
- Tuttediverse(F, T, U, W, R, O)
- $O + O = R + 10 C_1$
- $C_1 + W + W = U + 10 C_2$
- $C_2 + T + T = O + 10 C_3$
- $C_3 = F$
- Rappresentato in un ipergrafo di vincoli - nodi e ipernodi (quadrati) che rappresentano vincoli n-ari.

$$\begin{array}{r} T \quad W \quad O \\ + \quad T \quad W \quad O \\ \hline F \quad O \quad U \quad R \end{array}$$



SOLUZIONE DI UN CSP:

Soluzione di un CSP è un insieme di valori presi dai domini che assegnati alle rispettive variabili soddisfano tutti i vincoli. L'assegnazione dei valori alle variabili, ci porta a ridurre lo spazio di ricerca, quindi le possibili combinazioni da cercare. Un CSP può avere una soluzione (sudoku ben posto), zero soluzioni (sudoku sbagliato) o più soluzioni (sudoku non ben posto non dovrebbe succedere).

RISOLVERE I PROBLEMI DI CSP:

Bisogna come definire algoritmi efficienti per andare a risolvere questi problemi:

- **Algoritmi di propagazione di vincoli (inferenza)**: utilizzano i vincoli per ridurre il numero di valori legali per una variabile, che a sua volta può ridurre i valori legali per un'altra variabile e così via. L'idea è che questo lascerà meno scelte da considerare quando faremo la prossima scelta di un'assegnazione per una variabile.
La propagazione del vincolo può essere intrecciata con la ricerca o può essere eseguita come fase di preelaborazione, prima dell'inizio della ricerca. A volte questa preelaborazione può risolvere l'intero problema, quindi non è necessaria alcuna ricerca. Algoritmo AC-3. Questi non per forza portano ad una soluzione;
- **Algoritmi di ricerca** operano su assegnamenti di valori alle variabili e backtracking;
- **Alternanza di ricerca ed inferenza o backtracking** con forward checking o o con MAC;
- **Algoritmi di ricerca locale**.

INFERENZA NEI CSP - CONSISTENZA DI UN NODO:

Questi tipi di algoritmi si basano sul concetto di consistenza, ovvero verificare se il dominio delle variabili sono consistenti coi vincoli. Una singola variabile è nodo-consistente se tutti i valori del suo dominio soddisfano i suoi vincoli unari.

Ad esempio, se gli australiani del sud non amano il verde, possiamo rendere il nodo SA consistente eliminando il verde dal suo dominio. Diremo che un grafo è nodo-consistente se ogni variabile del grafo è nodo consistente.

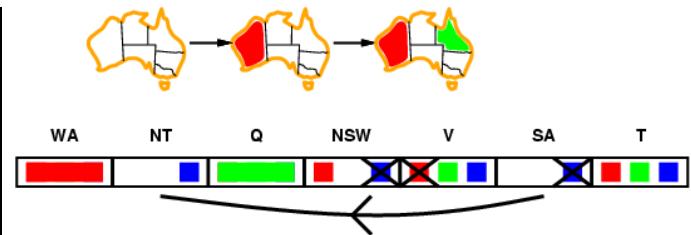
Inoltre, è facile ridurre tutti i vincoli unari in un CSP riducendo il dominio delle variabili con vincoli unari all'inizio del processo di risoluzione.

INFERENZA NEI CSP – PROPAGAZIONE DEI VINCOLI:

È utile la consistenza di arco, cioè se tra X e Y è presente un arco sul grafo dei vincoli, l'**arco consistente**, vuol dire che i valori possibili per X non violino i vincoli dei possibili valori di Y.

L'arco $X \rightarrow Y$ è consistente se e solo se per ogni valore x di X c'è qualche y consentito. Quindi per i quali non esiste un valore in y , li possiamo rimuovere.

Nel momento in cui vengono aggiornati i valori, anche i vicini devono aggiornarsi. Se X perde un valore, i vicini di X devono essere ricontrrollati (nell'esempio dobbiamo rimuovere il rosso in Victoria). Questo può essere eseguito dopo ogni assegnazione.



ARC CONSISTENCY ALGORITHM (AC-3):

L'algoritmo fa uso di una coda dove vengono inseriti gli archi (vincoli tra le variabili). Inizialmente la coda contiene tutti gli archi del CSP.

Otteniamo l'arco dalla coda e rimuoviamo l'inconsistenza. La funzione prende ogni valore di X_i , se non esistono valori in Y che soddisfano questo vincolo cancelliamo x dal dominio di X_i , nel caso in cui è vero bisogna aggiornare i vicini che hanno vincoli con X_i , e quindi aggiungo l'arco in coda per rianalizzarlo, questo significa aggiungere ogni vicino X_k di X_i .

```

function AC-3(csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$ 
    if RM-INCONSISTENT-VALUES( $X_i, X_j$ ) then
      for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
        add  $(X_k, X_i)$  to queue

  function RM-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff remove a value
    removed  $\leftarrow \text{false}$ 
    for each  $x$  in DOMAIN[ $X_i$ ] do
      if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy constraint( $X_i, X_j$ )
        then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow \text{true}$ 
    return removed

```

La complessità di tempo è $O(cd^3)$ dove c è il **numero di vincoli** mentre d è la **cardinalità del dominio**.

Il controllo di consistenza dell'arco è $O(d^2)$ mentre il numero max di volte che un arco entra in queue è d.

Ciò che possiamo notare di questo algoritmo è che non dà una soluzione per il problema CSP ma permette di ridurre il dominio delle variabili.

CERCARE UNA SOLUZIONE (RICERCA):

La tecnica di soluzione più semplice è detta **Generate & Test**

(G&T). Si assegna un valore ad ogni variabile e se si verifica che tutti i vincoli sono soddisfatti è stata trovata una soluzione altrimenti si prova con valori diversi.

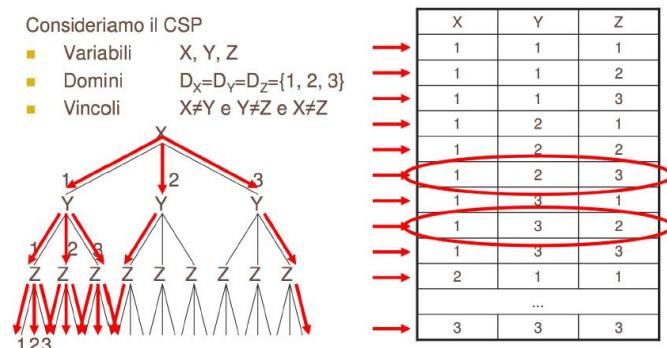
Il procedimento continua finché non ci sono più assegnamenti nuovi da provare (nel frattempo si è passati per (tutte) le soluzioni).

Una soluzione l'algoritmo la trova in tempo esponenziale nel numero di variabili con base gli elementi del dominio (d^n).

Applicazione al sudoku:

1. Si mette un numero in ogni cella vuota;
2. Si verifica che le regole siano rispettate;
3. Se qualche regola è violata, si prova con altri numeri.

Se abbiamo n variabili ed un dominio con d valori le foglie dell'albero sono d^n . Il G&T non è realistico per problemi complessi, infatti, nel caso del Sudoku, con 31 celle già riempite, abbiamo $9^{50} \approx 5 \cdot 10^{47}$ possibilità di esplorare.



STRATEGIE DI RICERCA:

La forza bruta non si può usare nei CSP, viene quindi applicata una ricerca in profondità.

Ricerca con backtracking (BT): ad ogni passo si assegna una variabile e si torna indietro in caso di fallimento.

Controllo anticipato della violazione dei vincoli: è inutile andare avanti fino alla fine e poi controllare; si può fare backtracking non appena si scopre un vincolo violato. La ricerca è limitata in profondità dal numero di variabili.

EURISTICHE PER PROBLEMI CSP (INDIPENDENTI DAL DOMINIO):

Sono diverse dalle euristiche precedenti, le quali erano definite appositamente per quei tipi di problemi (distanza linea d'aria ad esempio). In questo caso le regole sono generali per tutti i tipi di CSP:

- Quale variabile scegliere?
- Quali valori scegliere?
- Qual è l'influenza di una scelta sulle altre variabili? (Propagazione di vincoli)
- Come evitare di ripetere fallimenti? (Backtracking intelligente)

SCELTA DELLE VARIABILI:

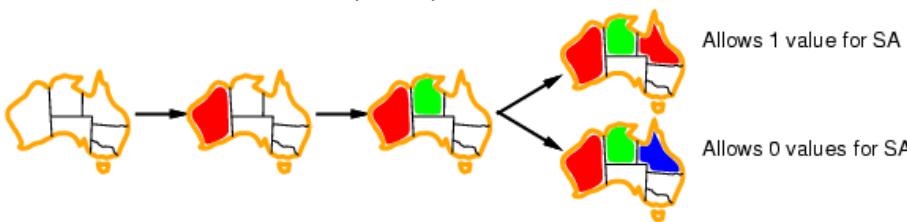
1. **MRV (Minimum Remaining Values)**: scegliere la variabile che ha meno valori possibili, la variabile più vincolata, in questo modo è possibile scoprire prima i fallimenti. Se alcune variabili X non hanno più valori legali, l'euristica MRV selezionerà X e l'errore verrà rilevato immediatamente, evitando ricerche inutili attraverso altre variabili;
2. In base al grado (**DH- Degree Heuristic**): scegliere la variabile coinvolta in più vincoli con le altre variabili (la variabile più vincolante) da usare a parità di MRV.



SCELTA DEI VALORI:

Una volta scelta la variabile, bisogna scegliere il valore da assegnare, ovvero:

- Il **valore meno vincolante (LCV - Least constraining value)**: quello che esclude meno valori per le altre variabili direttamente collegate con la variabile scelta. Cerca sempre di lasciare la massima flessibilità ai successivi assegnamenti di variabili. Cerca di trovare la soluzione, se c'è, il prima possibile.



PROPAGAZIONE DI VINCOLI:

- Verifica in avanti (**forward checking**) assegnato un valore ad una variabile si possono eliminare i valori incompatibili per le altre variabili collegate da vincoli (un passo solo). Se ho aggiornato X, vado a vedere le variabili vincolate a x e aggiorno i possibili valori, quindi l'algoritmo prevede di portare dietro i domini aggiornati;
- propagazione di vincoli**: si itera il forward checking; se una variabile ha il suo dominio ristretto per effetto dell'FC si vanno a controllare le variabili collegate.

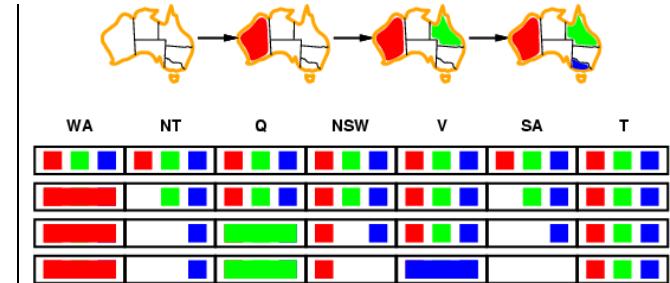
In generale una volta assegnato un valore ad una variabile, i vincoli eliminano dei valori possibili dai domini delle altre variabili. Nell'esempio, una volta scelto $X=1$, vado a vedere le variabili associate a X, ovvero Y e Z e rimuovo i valori incompatibili in questo caso 1 e 1.

Il **forward checking** è applicabile all'interno dell'algoritmo di backtracking come tecnica per stabilire quando tornare indietro. Ogni volta che si raggiunge uno stato non consistente (con almeno una variabile priva di valori rimasti \rightarrow node consistency) si effettua il backtracking.

Esempio forward checking map:

L'idea è di tenere traccia dei valori legali possibili per le variabili non assegnate, termina la ricerca quando una qualsiasi variabile non ha valori legali.

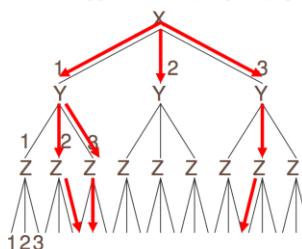
FC propaga le informazioni da variabili assegnate a variabili non assegnate, ma non fornisce una rilevazione precoce dei fallimenti. In questo caso NT e SA non possono essere blu. La propagazione dei vincoli ripetutamente impone dei vincoli locali



CONSISTENZA DEGLI ARCHI (MAC):

Un metodo veloce per propagare i vincoli. Nel grafo di vincoli, un arco orientato da A a B è consistente se per ogni valore x di A c'è almeno un valore y di B consistente con x . Quello che si fa è controllare la consistenza degli archi all'inizio e dopo ogni assegnamento (**MAC - Maintaining Arc Consistency**). Dopo un assegnamento di una variabile, si sfrutta il grafo dei vincoli per togliere i valori non più ammissibili delle altre variabili, quindi ad ogni assegnamento i valori dei domini delle variabili vengono ridotti e si taglano dei rami dell'albero.

- Consideriamo il CSP
- Variabili X, Y, Z
 - Domini $D_X=D_Y=D_Z=\{1, 2, 3\}$
 - Vincoli $X \neq Y \wedge Y \neq Z \wedge X \neq Z$



CSP CON MIGLIORAMENTO ITERATIVO:

Si parte con tutte le variabili assegnate (tutte le regine sulla scacchiera) e ad ogni passo si modifica l'assegnamento ad una variabile per cui un vincolo è violato. **Algoritmo di riparazione euristica**, quindi l'euristica nello scegliere un nuovo valore potrebbe essere quella dei **conflicti minimi**: si sceglie il valore che crea meno conflitti.

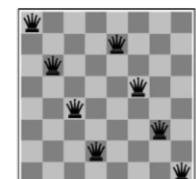
Esempio 8 regine:

Vengono elencate in maniera esaustiva tutte le combinazioni ammissibili andando a specificare le coppie che possiamo ammettere. L'algoritmo da un possibile assegnamento di valori V_1 a V_8 andrà a spostarsi in una soluzione in cui una di queste V cambia valore e il numero di conflitti si riduce al minimo.

Questo algoritmo è molto efficace, con 1 milione di regine soluzione in 50 passi, viene utilizzato in problemi reali di progettazione e scheduling.

Formulazione come CSP:

- V_i : posizione della regina nella colonna i -esima
- $D_i: \{1 \dots 8\}$
- Vincoli di "non-attacco" tra V_1 e V_2 : $\{(1,3), (1,4), (1,5), \dots, (1,8), (2,4), (2,5), \dots, (2,8), \dots\}$



METODI RIPARATIVI (RICERCA LOCALE):

Di solito ci sono una serie di **plateau** dove gli algoritmi restano bloccati. Si possono utilizzare le **mosse laterali** per uscire dai plateau, oppure **tabù search**, dove tiene conto di stati visitati di recente così si evita di ritornarci, oppure il simulated annealing. La ricerca locale può essere usata in **ambienti online** quando il problema può cambiare, ad esempio scheduling voli aerei in caso di imprevisti.

METODI RIPARATIVI VS COTRUTTIVI:

Gli algoritmi costruttivi funzionano bene soprattutto su CSP-binari (o con pochi vincoli e pochi valori): infatti, la complessità di AC-3 è $O(nk^3)$ dipende da k =valori e n =archi. Diventano poco gestibili con Constraint Network a cardinalità maggiore e con molti valori (Es N-regine con $N>106$).

Gli algoritmi riparativi, locali, forniscono **meno garanzie teoriche** ma nel caso di problemi molto complessi risultano efficienti nella pratica.

SOTTO PROBLEMI INDIPENDENTI:

La struttura del problema, rappresentata dal grafo dei vincoli, può essere utilizzata per trovare rapidamente delle soluzioni. Esamineremo problemi con una struttura specifica e strategie per migliorare il processo di ricerca di una soluzione.

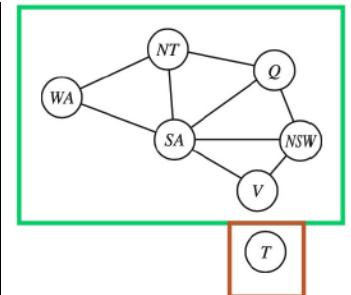
Il primo caso ovvio è quello dei **sotto problemi indipendenti**, ad esempio nella colorazione della mappa, colorare la Tasmania (era indipendente dagli altri) e colorare la terraferma sono sotto-problemi indipendenti, dove qualsiasi soluzione per la terraferma combinata con qualsiasi soluzione per la Tasmania produce una soluzione per l'intera mappa.

Ciascun **componente collegato** del grafo del vincolo corrisponde a un sotto-problema CSP_i .

Se l'assegnazione S_i è una soluzione di CSP_i , allora $U_i S_i$ è una soluzione di $U_i CSP_i$. Questo è importante, perché supponiamo che ogni CSP_i presenti c variabili dal numero totale di n variabile, dove c è una costante, allora abbiamo n/c sottoproblemi, ognuno dei quali richiede al più d^c lavoro per essere risolto, dove d è la dimensione del dominio. Quindi il lavoro totale è $O(d^c n/c)$ che è lineare in n , senza la scomposizione il lavoro totale sarebbe $O(d^n)$ che è esponenziale in n .

Il risparmio nel tempo di calcolo è rilevante:

- n : numero variabili
- c : variabili per sotto-problemi
- d : dimensione del dominio
- n/c problemi indipendenti
- $O(d^c)$: complessità per risolvere uno
- $O(d^c n/c)$: lineare sul numero di variabili n piuttosto che $O(d^n)$ esponenziale

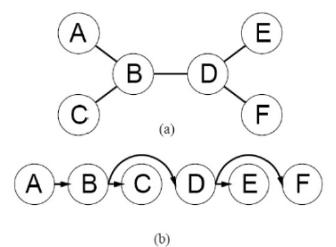


Dividere un CSP booleano con 80 variabili in 4 sotto-problemi riduce il tempo di soluzione nel caso peggiore della vita dell'universo a meno di un secondo.

Il problema è che il 99% dei casi il grafo non si può dividere nel caso in cui i vincoli impattano su tutte le variabili.

STRUTTURA AD ALBERO:

Un grafo di vincoli è un **albero** quando due nodi sono collegati da un solo percorso; possiamo scegliere qualsiasi variabile alla radice di un albero. Scelta una variabile come radice, l'albero induce un **ordinamento topologico** sulle variabili. I figli di un nodo sono elencati dopo il loro genitore. Mostreremo che qualsiasi albero strutturato può essere risolto in tempo lineare nel numero di variabili (DAC).



DIRECTED ARC CONSISTENCY (DAC):

Un CSP è definito come **direct arc-consistent** secondo un ordinamento delle variabili X_1, X_2, \dots, X_n se e solo se ogni X_i è arco consistente con ogni X_j per $j > i$. Possiamo fare in modo che un grafo a forma di albero sia **direct arc-consistent** in un passaggio sulle n variabili; ogni passaggio deve confrontare fino a d possibili valori di dominio per due variabili, per un tempo totale di $O(nd^2)$.

Tree-CSP-solver:

1. Procedendo da X_n a X_2 , gli archi $PARENT(X_i) \rightarrow X_i$ sono consistenti riducendo il dominio di X_i , se necessario. Può essere fatto in un solo passaggio
2. Procedendo da X_1 a X_n assegnare valori alle variabili; non c'è bisogno di **backtracking** poiché ogni valore per un padre ha almeno un valore legale per il figlio.

Effettua un ordinamento topologico, parte dall'ultimo nodo e arriva al primo nodo (radice) e applica la funzione per rendere consistente il padre con X_j .

Una volta resa consistente la sequenza di direct arc consistent. Parte dall'inizio e qualsiasi assegnamento di variabili porta ad una soluzione.

Se il nostro grafo può essere trasformato in albero, applichiamo l'algoritmo e subito troviamo una soluzione.

RIDURRE GRAFI IN ALBERI:

Per applicare tutto ciò bisogna prendere un grafo e trasformarlo in un albero.

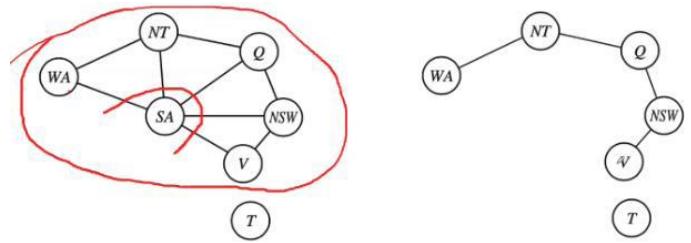
Se potessimo eliminare l'Australia Meridionale, il grafico diventerebbe un albero (nell'esempio di colorazione). Ciò può essere fatto fissando un valore per SA (che in questo caso non ha importanza) e rimuovendo valori incoerenti dalle altre variabili, aggiornando anche il dominio delle variabili.

Ad esempio, possiamo fissare a SA = rosso, l'algoritmo restituirà blu, verde, blu, verde, ecc...

```

function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components X, D, C
  n  $\leftarrow$  number of variables in X
  assignment  $\leftarrow$  an empty assignment
  root  $\leftarrow$  any variable in X
  X  $\leftarrow$  TOPOLOGICALSORT(X, root)
  for j = n down to 2 do
    MAKE-ARC-CONSISTENT(PARENT(Xj), Xj)
    if it cannot be made consistent then return failure
  for i = 1 to n do
    assignment[Xi]  $\leftarrow$  any consistent value from Di
    if there is no consistent value then return failure
  return assignment

```



INSIEME DI TAGLIO DEI CICLI:

In generale, dobbiamo applicare una strategia di suddivisione del dominio, provando con diversi assegnamenti:

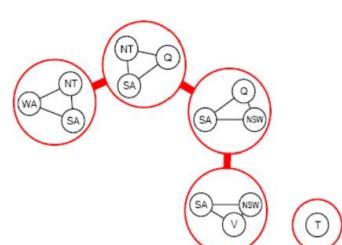
1. Scegli un sottoinsieme *S* delle variabili del CSP in modo tale che il grafo dei vincoli diventi un albero dopo la rimozione di *S*. *S* è chiamato **cycle cutset**;
2. Per ogni possibile assegnazione coerente alle variabili in *S*:
 - a. Rimuovere dai domini delle variabili rimanenti tutti i valori che sono incompatibili con l'assegnamento per *S*;
 - b. Se il rimanente CSP ha una soluzione, restituirla insieme all'assegnazione per *S*.

La complessità temporale è di $O(d^c(n-c)d^2)$, dove *c* è la dimensione del cycle cutset e *d* la dimensione del dominio.

Dobbiamo provare ciascuna delle combinazioni d^c di valori per le variabili in *S*, e per ogni combinazione dobbiamo risolvere un **problema ad albero** di dimensioni (*n-c*).

DECOMPOSIZIONE AD ALBERO:

L'approccio consiste in una decomposizione ad albero del grafo dei vincoli in una serie di sotto-problemi connessi. Ogni sotto-problema viene risolto in modo indipendente e le soluzioni risultanti vengono quindi combinate in modo intelligente.



PROPRIETÀ DI UNA DECOMPOSIZIONE:

La decomposizione deve soddisfare i tre requisiti:

1. Ogni variabile nel problema originale appare in almeno uno dei problemi secondari;
2. Se due variabili sono collegate da un vincolo nel problema originale, devono apparire insieme (insieme al vincolo) in almeno uno dei problemi secondari;
3. Se una variabile appare in due sotto-problemi nella struttura, deve apparire in ogni sotto-problema lungo il percorso che collega quei sotto-problemi.

Le condizioni 1-2 assicurano che tutte le variabili e i vincoli siano rappresentati nella decomposizione mentre la condizione 3 riflette il vincolo secondo il quale ogni data variabile deve avere lo stesso valore in ogni sotto-problema in cui appare; i collegamenti che uniscono i sotto-problemi nella struttura impongono questo vincolo.

RISOLVERE UN PROBLEMA DECOMPOSTO:

Risolviamo ogni sotto-problema in modo indipendente. Se un problema non ha soluzione, il problema originale non ha soluzione. Mettere insieme le soluzioni, risolviamo quindi un meta-problema definito come segue:

- Ogni sotto-problema è una "mega-variabile" il cui dominio è l'insieme di tutte le soluzioni per il sotto-problema.
Es. Dom(X1)={<WA=r,SA=b,NT=g>...} le 6 soluzioni al primo sotto problema.
- I vincoli assicurano che le soluzioni dei sotto problemi assegnino gli stessi valori alle variabili che condividono.

Idealmente dovremmo trovare, tra i molti possibili, una decomposizione dell'albero con **larghezza minima** (la dimensione del sotto problema più grande -1). Questo è NP-hard. Se w è la larghezza minima dell'albero delle possibili decomposizioni dell'albero, la complessità è $O(nd^{w+1})$.

7. AGENTI LOGICI

Il funzionamento è completamente differente da quelli visti fino ad ora, mentre prima andavamo a definire algoritmi per trovare la soluzione, al contrario qui andiamo a far ragionare le macchine.

Viene utilizzato un **approccio dichiarativo**, gli agenti hanno una conoscenza che dipende dal problema mentre la parte computazionale (ragionamento) è definita in maniera indipendente dal problema e ciò che cambia è la **base di conoscenza (KB - Knowledge Base)** la quale rappresenta la parte fondamentale.

La KB è tutto ciò che l'agente conosce, ed essa viene definita tramite la logica matematica; quindi, la rappresentiamo tramite un insieme di **formule**, ed ogni formula rappresenta un'asserzione sul mondo, inoltre, una formula è detta **assioma** quando non deve essere ricavata da altre formule, ed è possibile ricavare nuova conoscenza applicando le regole dell'ambiente a partire dagli assiomi.

La KB deve prevedere meccanismi per aggiungere nuove formule e per le interrogazioni.

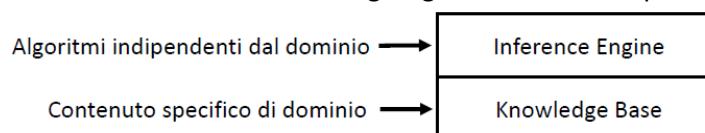
Si usa un approccio dichiarativo attraverso due azioni standard:

- **TELL** (Asserisci): l'agente acquisisce qualche informazione e la inserisce nella KB;
- **ASK** (Chiedi): utilizziamo la KB per sapere qualcosa, viene lanciato sull'inference engine.

La risposta di ogni richiesta (ASK) posta alla base di conoscenza deve essere una conseguenza di quello che è stato detto (TELL) in precedenza.

Un agente può essere descritto sulla base di due livelli di astrazione:

- **Livello della conoscenza**: cosa si conosce, a prescindere da come è implementato;
- **Livello dell'implementazione**: le strutture di dati nella KB e gli algoritmi che li manipolano.



Un concetto importante è quello dell'**espressività**, ovvero quanto permette di andare a specificare. Più è espressivo più il processo di inferenza è complesso.

Esempio un semplice agente basato sulla conoscenza:

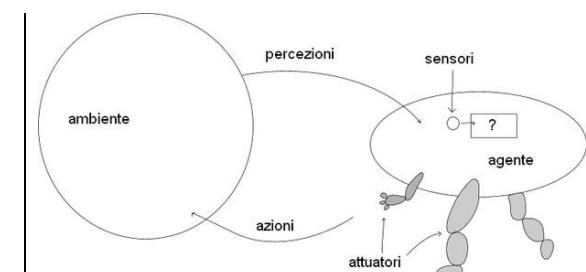
L'algoritmo nella prima riga prende in input una percezione e restituisce un'azione.

Nella seconda riga mantiene in memoria una KB che può contenere conoscenza iniziale (background knowledge).

Per decidere quale azione fare dipende dal KB, infatti, registra la percezione nella KB tramite TELL, successivamente effettua l'ASK, e chiede quale azione bisogna effettuare all'istante t. Registra l'azione e la va ad eseguire.

Il programma agente fa tre cose:

1. Comunica le sue percezioni alla KB (TELL);
2. Chiede alla KB quale azione eseguire (ASK);
3. Registra l'azione scelta nella KB prima di esegirla (TELL).



```
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
```

7.1 IL MONDO DEL WUMPUS

Descriviamo un ambiente in cui gli agenti basati sulla conoscenza possono dimostrare il loro valore:

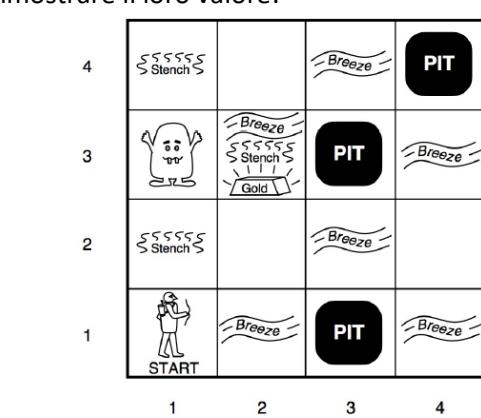
- Una caverna formata da stanze collegate da passaggi.
- L'agente vuole scovare l'oro.
- **ATTENZIONE**: wumpus e pozzi nascosti nelle stanze uccidono l'agente.
- L'agente ha solo una freccia per uccidere il wumpus.

Misura di prestazione:

- +1000 quando l'agente trova l'oro;
- -1000 quando l'agente muore;
- -1 per ogni azione eseguita;
- -10 per l'uso della freccia.

Descrizione dell'ambiente:

- Nelle celle adiacenti al wumpus c'è stench;
- Nelle celle adiacenti ai pozzi c'è Breeze;
- Nella cella che contiene l'oro c'è scintillio (glitter);
- L'agente ha solo una freccia per uccidere il wumpus;
- La freccia colpisce il wumpus solo se si trova nella direzione corretta;
- L'agente può prendere l'oro solo se si trova nella stessa cella.



Attuatori:

- Girare a destra - Girare a sinistra - Andare avanti – Afferrare - Tirare – Esci.

Sensori:

- Fetore – Brezza – Scintillio – Urto – Urlo.

CARATTERIZZAZIONE DEL MONDO DEL WUMPUS:

Abbiamo un mondo **parzialmente osservabile** perché alcuni aspetti dello stato non sono percepibili, presenta un agente **singolo** solo lui si può muovere, in un ambiente **deterministico** (gli esiti vengono specificati esattamente), **sequenziale** siccome si può raggiungere un premio soltanto dopo diverse azioni, **statico** dato che il wumpus e i pozzi non si muovono, ed infine **discreto** dato che ha un numero finito di stati distinti.

Esempio esplorare il mondo del Wumpus:

A=Agente - B=Brezza - G=Scintillio(Glitter) - OK=Cella sicura - P=Pozzo - S=Fetore(Stench) - V=Cella visitata - W=Wumpus

Cella [1,1]:

- La KB iniziale contiene le regole dell'ambiente
- La prima percezione è: [none,none,none,none,none]
- Ci possiamo muovere verso una cella sicura, ad esempio la cella [1,2]

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK			
1,1	A	2,1	3,1
OK	OK		4,1

Cella [1,2]:

- La brezza indica che c'è un pozzo in [1,3] o in [2,2]
- La seconda percezione è [none,brezza,none,none,none]
- L'unica cella sicura è la [1,1]
- Dobbiamo tornare indietro e andare a destra (cella [2,1])

4			
3			
2	B	OK	
1	A		
	OK	OK	
1			
2			
3			
4			

Cella [2,1]:

- Il fetore indica che il wumpus è vicino
- La seconda percezione è [fetore,none,none,none,none]
- Il wumpus non può essere nella cella [1,1]
- Il wumpus non può essere nella cella [2,2], altrimenti l'agente avrebbe percepito puzza quando era in [1,2]
- L'agente può dedurre che il wumpus si trova nella cella [3,1]
- La mancanza di brezza in [2,1] implica che non ci sono pozzi in [2,2]

4			
3	P?		
2	B	OK	S
1	A	OK	OK
	V	S	OK
1	A	>A	W
2			
3			
4			

Cella [2,2]:

- Non si percepisce nulla le celle vicine sono sicure
- La terza percezione è [none,none,none,none,none]
- Le celle vicine possono essere contrassegnate con OK
- Ci possiamo muovere verso una cella sicura, ad esempio la cella [3,2]

4			
3	P?	OK	
2	B	OK	S
1	A	OK	OK
	V	S	OK
1	A	>A	W
2			
3			
4			

Cella [3,2]:

- Lo scintillio indica che abbiamo trovato l'oro
- La terza percezione è [fetore,brezza,scintillio,none,none]
- L'agente può afferrare l'oro

4			
3	P?	OK	
2	B	OK	S
1	A	OK	OK
	V	S	OK
1	A	>A	W
2			
3			
4			

7.2 LOGICA IN GENERALE

Le logiche sono linguaggi formali per rappresentare le informazioni, in modo tale che possano essere tratte delle conclusioni. La **sintassi** specifica come devono essere formattate le sentenze mentre la **semantica** specifica il significato delle sentenze quindi la verità delle formule rispetto a ogni mondo possibile.

La **conseguenza logica** è una formula che segue logicamente da un'altra:

$$KB \models \alpha, \text{ ad esempio, } x+y=4 \text{ implica } 4=x+y$$

Dove KB implica la sentenza α sé e soltanto se α è vera in tutti i **mondi** in cui KB è vera. La conseguenza logica è una relazione tra sentenza (sintassi) che si basa sulla semantica.

Consideriamo la seguente situazione:

- Nessuna percezione in [1,1];
- Brezza in [2,1].

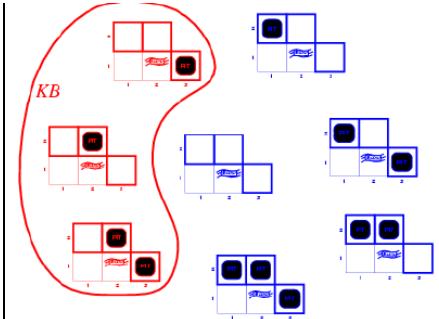
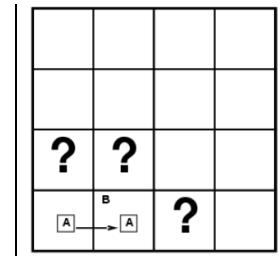
L'agente vuole sapere se le stanze adiacenti contengono pozzi:

- [1,2], [2,2], [3,1];
- Il pozzo può esserci o meno (V/F);
- $2^3 = 8$ possibili modelli (mondi).

La KB è falsa nei modelli che contraddicono le conoscenze dell'agente, nell'esempio, quando la cella 1,2 contiene il pozzo.

I modelli in cui nella [1,2] c'è un pozzo sono falsi, siccome l'agente è stato nella cella [1,1] e non ha percepito brezza.

- $KB \models \alpha_1$: Non c'è pozzo in [1,2]
 α_1 = non c'è pozzo in 1,2 tutti quelli in cui non c'è un pozzo in 1,2 vanno bene
 (quelle 4 vanno bene);
- $KB \models \alpha_2$: L'agente non può concludere che non ci sia pozzo in [2,2]
 α_2 = non c'è pozzo in 2,2 è falsa perché abbiamo 3 da fuori e non vale l'inclusione.
 In alcuni modelli in cui KB è vera, α_2 non lo è.



Questa tecnica è chiamata **model checking (processo inferenziale)** che permette di verificare se una certa formula è conseguenza logica di un'altra formula o di una KB.

La differenza tra la conseguenza logica e l'inferenza sta nel fatto che la **conseguenza logica** è dire l'ago si trova in un pagliaio mentre l'**inferenza** equivale a trovarlo.

INFERENZA:

α può essere derivato da KB attraverso un algoritmo (procedura) di inferenza i :

$$KB \vdash_i \alpha$$

Le proprietà che ci servono sono:

- **Correttezza (soundness)**: l'algoritmo i è corretto se ogni volta che vale $KB \vdash_i a$, è anche vero che $KB \models a$ (un algoritmo che deriva solo formule che sono conseguenze logiche);
- **Completezza (completeness)**: i è completo se ogni volta che vale $KB \models a$ è anche vero che $KB \vdash_i a$ (un algoritmo che deriva ogni formula che è conseguenza logica).

LOGICA PROPOZIONALE (BOOLEANA):

È una logica semplice, la quale permette di illustrare le idee che sono alla base dei linguaggi logici.

La **sintassi** della logica proposizionale definisce le formule accettabili (sentenze). I simboli proposizionali (S_1, S_2, \dots) sono **formule atomiche** che rappresentano una **proposizione** che può essere vera o falsa.

Ed è possibile costruire **formule complesse** grazie all'uso delle **parentesi** e dei **connettivi logici** ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$).

La **sintassi** la si va a definire con una grammatica, la quale indica come si scrivono le formule della logica proposizionale.

La **semantica** della logica proposizionale definisce le regole usate per determinare il valore di verità di una formula nei confronti di un particolare modello, dove ogni **modello** specifica per ogni simbolo se è vero o falso, deve specificare, dato un modello, come si fa a calcolare il valore di verità di qualsiasi formula.

Le regole per valutare il valore di verità rispetto ad un modello m sono le seguenti ed inoltre è possibile esprimere tramite una **tavola di verità**.

$\neg S$	è vero sse	S è falso
$S_1 \wedge S_2$	è vero sse	S_1 è vero and S_2 è vero
$S_1 \vee S_2$	è vero sse	S_1 è vero or S_2 è vero
$S_1 \Rightarrow S_2$	è vero sse	S_1 è falso or S_2 è vero
i.e., è falso sse	S_1 è vero	and S_2 è falso
$S_1 \Leftrightarrow S_2$ è vero sse	$S_1 \Rightarrow S_2$ è vero and $S_2 \Rightarrow S_1$ è vero	

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

SENTENZE PER IL MONDO DEL WUMPUS:

Sulla base della logica proposizionale possiamo costruire una base di conoscenza per il mondo del wumpus. Concentriamoci sugli aspetti "immutabili" di tale mondo.

Per ora, possiamo usare i seguenti simboli per ogni posizione [x,y]

- $P_{x,y}$ è vero se esiste un pozzo in [xy]
- $W_{x,y}$ è vero se esiste un wumpus in [xy], morto o vivo
- $B_{x,y}$ è vero se l'agente percepisce una brezza in [xy]
- $S_{x,y}$ è vero se l'agente percepisce un fetore in [xy]

Esempio del model checking:

In [1,1] non ci sono pozzi ne brezza:

- $R_1: \neg P_{1,1}$
- $R_4: \neg B_{1,1}$

In [2,1] c'è brezza:

- $R_5: B_{2,1}$

In una cella si percepisce brezza sse c'è un pozzo in una cella adiacente:

- $R_2: B_{1,1} \leftrightarrow (P_{1,2} \vee P_{2,1})$
- $R_3: B_{2,1} \leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$

?	?		
A	B	A	?
1	2	3	4

INFERENZA PER IL MONDO DEL WUMPUS:

Per costruire un primo algoritmo di inferenza utilizziamo l'approccio **model checking**, ovvero si costruiscono tutti i modelli del KB dove il risultato delle variabili è TRUE. Un'implementazione diretta della definizione di conseguenza logica.

Data una sentenza α (ad esempio $\neg P_{1,2}$), enumeriamo esplicitamente i modelli e verifichiamo se α è vera in ogni modello in cui KB lo è:

- 7 simboli proposizionali;
- $2^7 = 128$ modelli.

Data una sentenza α , enumeriamo esplicitamente i modelli e verifichiamo se α è vera in ogni modello in cui KB lo è:

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	true	true	true	true	false	false						
false	false	false	false	false	false	true	true	true	false	true	false	false
:	:	:	:	:	:	:	:	:	:	:	:	:
false	true	false	false	false	false	true	true	true	true	true	true	false
false	true	false	false	false	true	false	true	true	true	true	true	true
false	true	false	false	true	true	true	true	true	true	true	true	true
false	true	false	false	true	false	false	true	false	false	true	true	false
:	:	:	:	:	:	:	:	:	:	:	:	:
true	false	true	true	false	true	false						

Da questi risultati possiamo dire $\neg B_{1,1}$ e $B_{2,1}$, a noi interessava sapere $\neg P_{1,2}$.

ALGORITMO PER CALCOLARE CONSEGUENZE LOGICHE:

L'algoritmo va a costruire dei modelli, inizialmente è vuoto e man mano per ogni simbolo assegna una volta il valore vero e una volta falso, ricorsivamente su tutte le variabili.

Una volta arrivato alla fine va a vedere se la KB è vera o falsa in quella sequenza di valori.

Per n simboli la complessità temporale è $O(2^n)$.

fuction TV-CONSEGUE?(KB, α) returns true oppure false

inputs: KB, la base di conoscenza, una formula logica proposizionale α , la query, una formula logica proposizionale

simboli \leftarrow una lista di simboli proposizionali contenuti in KB e α
return TV-VERIFICA-TUTTO(KB, α , simboli, {})

fuction TV-VERIFICA-TUTTO(KB, α , simboli, modello) returns true oppure false

if VUOTO(simboli) **then**
if CP-VERO?(KB, modello) **then return** CP-VERO?(α , modello)
else return true //quando KB è false, restituisce sempre true

else do
 $P \leftarrow$ PRIMO(simboli); resto \leftarrow RESTO(simboli);
return TV-VERIFICA-TUTTO(KB, α , resto, modello $\cup \{P = \text{true}\}$) **and**
TV-VERIFICA-TUTTO(KB, α , resto, modello $\cup \{P = \text{false}\}$)

EQUIVALENZA LOGICA:

Un altro modo per poter implementare un algoritmo di inferenza è quello di effettuare delle dimostrazioni. Anziché provare in maniera esaustiva tutti i valori si va a dimostrare il tutto. Per far ciò abbiamo bisogno dell'equivalenza logica.

Due formule α e β sono **logicamente equivalenti** se sono vere nello stesso insieme di modelli ($\alpha \equiv \beta$).

Una definizione alternativa afferma che: due formule α e β sono equivalenti soltanto se ognuna di esse è conseguenza logica dell'altra, cioè:

$$\alpha \equiv \beta \text{ se e solo se } \alpha \models \beta \text{ e } \beta \models \alpha$$

VALIDITÀ:

Un altro strumento utile è la validità. Una formula è **valida** se è vera in tutti i modelli (anche dette tautologie), $A \vee \neg A$ è sempre valida. Tale proprietà è connessa all'inferenza attraverso il **Teorema di Deduzione**:

$$KB \models a \text{ se e solo se } (KB \Rightarrow a) \text{ è valida}$$

Tale teorema afferma che ogni formula di implicazione valida descrive un'inferenza legittima.

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutatività di \wedge
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutatività di \vee
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associatività di \wedge
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associatività di \vee
$\neg(\neg \alpha) \equiv \alpha$	eliminazione della doppia negazione
$(\alpha \Rightarrow \beta) \equiv (\neg \alpha \Rightarrow \beta)$	contrapposizione
$(\alpha \Rightarrow \beta) \equiv (\neg \alpha \vee \beta)$	eliminazione dell'implicazione
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	eliminazione del bicondizionale
$\neg(\alpha \wedge \beta) \equiv (\neg \alpha \vee \neg \beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg \alpha \wedge \neg \beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributività di \wedge su \vee
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributività di \vee su \wedge

SODDISFACIBILITÀ:

Una formula è **soddisfacibile** se è vera in, o soddisfatta da, *qualche* modello: $R_1 \vee R_2 \vee R_3 \vee R_4 \vee R_5$ vera in 3 modelli

Al contrario, una formula è **insoddisfacibile** se è vera in nessun modello: $A \wedge \neg A$

La **soddisfaccibilità** è connessa all'inferenza attraverso:

KB $\models a$ se e soltanto se (**KB** $\wedge \neg a$) è insoddisfacibile

Dimostriamo che $\neg a$ è vera in tutti i modelli, parliamo di dimostrazione per assurdo.

7.3 REGOLE DI INFERENZA

Le **regole di inferenza** possono essere usate per derivare una dimostrazione o prova; quindi, applichiamo una catena di conclusioni che portano all'obiettivo desiderato. Esistono diversi tipi di regole di inferenza e noi useremo il **Modus Ponens**

$$\frac{a \Rightarrow \beta, a}{\beta}$$

questo ci dice che se a è vero, e a implica β è vero allora vuol dire che β è vero.

Ogni volta che sono date le formule $\alpha \Rightarrow \beta$ e α , allora si può inferire la formula β .

Modus Ponens

$$\frac{(WumpusDavanti \wedge WumpusVivo) \Rightarrow ScagliaFreccia, \quad (WumpusDavanti \wedge WumpusVivo)}{ScagliaFreccia}$$

Se è data $(WumpusDavanti \wedge WumpusVivo) \rightarrow ScagliaFreccia$ e vale $(WumpusDavanti \wedge WumpusVivo)$, allora si può inferire $ScagliaFreccia$.

Un'altra regola è l'**eliminazione degli AND**, se $\frac{a \wedge \beta}{a}$ se nella KB ho a AND β allora posso aggiungere a.

Eliminazione degli AND

$$\frac{(WumpusDavanti \wedge WumpusVivo)}{WumpusVivo}$$

Se è data $(WumpusDavanti \wedge WumpusVivo)$, allora si può inferire $WumpusVivo$.

Si aggiungono anche tutte le equivalenze della tabella precedente dove sono elencate le equivalenze.

Esempio di trasformazione (con inferenza):

Consideriamo la seguente situazione:

- Nessuna percezione in [1,1];
- Brezza in [2,1].

L'agente vuole sapere se le stanze adiacenti contengono pozzi:

- [1,2], [2,2], [3,1];
- Il pozzo può esserci o meno (V/F);
- $2^3 = 8$ possibili modelli (mondi).

Vogliamo dimostrare $\neg P_{1,2}$:

1. $R_2: B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
2. $R_6: (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
3. $R_7: ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
4. $R_8: (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1}))$
5. $R_9: \neg(P_{1,2} \vee P_{2,1})$
6. $R_{10}: \neg P_{1,2} \wedge \neg P_{2,1}$

Applichiamo la regola di De Morgan su R_9 .

?	?		
	B		?
A	\rightarrow	A	

ALGORITMI PER DIMOSTRAZIONI:

Possiamo applicare uno degli algoritmi di ricerca non informata per trovare una sequenza di passi che costituisca la dimostrazione:

- **Stato iniziale:** la base di conoscenza;
- **Azioni:** l'insieme delle azioni consiste di tutte le regole di inferenza applicate a tutte le formule che corrispondono alla metà superiore della regola di inferenza;
- **Risultato:** il risultato di un'azione è l'aggiunta della formula nella metà inferiore della regola di inferenza;
- **Obiettivo:** arrivare ad uno stato che contiene la formula che stiamo cercando di dimostrare.

7.4 DIMOSTRAZIONE DEI TEOREMI

Le metodologie per effettuare la dimostrazione dei teoremi si dividono in due categorie principali:

- **Regole di inferenza:** Generazione di nuove sentenze a partire dalle precedenti;
Dimostrazione = un'applicazione sequenziale di regole di inferenza può usare queste ultime come operatori di un algoritmo di ricerca standard e di solito è richiesta la trasformazione delle sentenze in qualche **forma normale**.
- **Model checking:** Enumerazione della tavola di verità (sempre esponenziale in n). Backtracking migliorato, attraverso l'**algoritmo DPLL**. Ricerca euristica nello spazio dei modelli (corretto ma non completo), attraverso gli algoritmi **min-conflicts-like-hill-climbing**.

DIMOSTRAZIONE PER RISOLUZIONE:

Le **regole di inferenza** viste finora sono corrette, ma non abbiamo discusso della completezza degli algoritmi di inferenza. Un'ulteriore regola d'inferenza è la **risoluzione**; questa regola unita a qualsiasi algoritmo di ricerca completo dà luogo ad un algoritmo di inferenza completo.

Versione semplice (risoluzione unitaria):

$$\frac{l_1 \vee \dots \vee l_k, \quad m}{l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_k} \text{ con } m \text{ complemento di } l_i$$

dove ogni l è un letterale e l_i ed m sono letterali complementari (cioè uno è la negazione dell'altro).

La regola di inferenza unitaria prende una clausola (disgiunzione di letterali) e un letterale e produce una nuova clausola. Con questa regola posso aggiungere alla KB la sequenza che c'era prima ma cancellando l_i , per poter applicare dobbiamo **avere una disgiunzione di letterali**.

Versione complessa (risoluzione completa):

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

dove l_i e m_j sono letterali complementari (cioè uno è la negazione dell'altro)

La risoluzione prende due clausole e ne produce una nuova che contiene tutti i letterali delle due clausole originali tranne i due complementari.

Esempio di risoluzione (col Wumpus):

Supponiamo che l'agente torni da [2,1] a [1,1] e da li passi in [1,2] dove percepisce una puzza, ma nessuna brezza d'aria:

- $R_{11}: \neg B_{1,2}$
- $R_{12}: B_{1,2} \leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3})$

Applichiamo lo stesso processo che ci ha portato a R_{10} ed otteniamo:

- $R_{13}: \neg P_{2,2}$
- $R_{14}: \neg P_{1,3}$

Applichiamo l'eliminazione del bicondizionale a R_3 , seguita dal modus ponens con R_5 :

- $R_{15}: P_{1,1} \vee P_{2,2} \vee P_{3,1}$

Applichiamo la regola di risoluzione tra R_{15} e R_{13} :

$$\frac{P_{1,1} \vee P_{2,2} \vee P_{3,1}, \quad \neg P_{2,2}}{P_{1,1} \vee P_{3,1}}$$

Applichiamo la regola di risoluzione tra R_{16} e R_1 :

$$\frac{P_{1,1} \vee P_{3,1}, \quad \neg P_{1,1}}{P_{3,1}}$$

- $R_{17}: P_{3,1}$

Una dimostrazione di teoremi basato sulla risoluzione può, per ogni coppia di formule α e β della logica proposizionale, decidere se $\alpha \models \beta$.

- Usa la **forma normale congiuntiva (CNF)**, congiunzione di disgiunzioni letterali (clausole), cioè (OR) AND (OR) AND ...;
- La regola di risoluzione è **corretta**, si riescono a derivare soltanto sentenze implicate;
- La risoluzione è **completa** nel senso che può essere sempre usata o per confermare o per rifiutare una sentenza (non può essere usata per enumerare le sentenze vere).

FORMA NORMALE CONGIUNTIVA (CNF):

La regola di risoluzione si applica soltanto a clausole (disgiunzione di letterali).

Ogni formula della logica proposizionale è logicamente equivalente ad una **congiunzione di clausole**; quindi, ogni formula in logica proposizionale (anche una KB) può essere trasformata in CNF.

Esempio CNF:

Convertiamo la seguente formula in CNF:

$$\begin{aligned} & B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) \\ 1. & (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) \\ 2. & (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) \\ 3. & (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) \\ 4. & (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) \end{aligned}$$

1. Eliminiamo \Leftrightarrow , attraverso $(\alpha \Leftrightarrow \beta) \equiv ((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha))$.
2. Eliminiamo \rightarrow , attraverso $(\alpha \rightarrow \beta) \equiv (\neg \alpha \vee \beta)$.
3. Muoviamo \neg all'interno, attraverso le regole di De Morgan e di doppia negazione.
4. Applichiamo la legge di distributività, distribuendo \wedge su \vee .

La formula originale è ora in CNF, come congiunzione di tre clausole.

Dimostrazione:

La dimostrazione di ciò viene fatta per assurdo, ovvero mostra che $KB \wedge \neg a$ non è soddisfacibile.

Converte la formula $KB \wedge \neg a$ in CNF e si applica la regola di risoluzione alle clausole riguardanti, ogni coppia che contiene letterali complementari è risolta per produrre una nuova clausola che viene aggiunta all'insieme (se non è già presente).

Il processo continua finché non si verifica una delle seguenti due possibilità:

1. Non è possibile aggiungere alcuna clausola (a non è conseguenza logica di KB)
2. La risoluzione applicata a due clausole dà come risultato la clausola vuota (KB segue logicamente a) siamo arrivati ad una **contraddizione**.

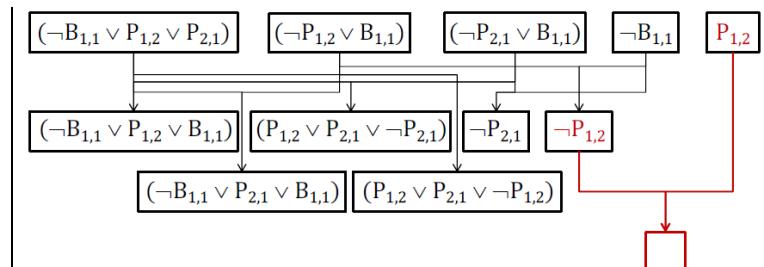
Esempio di esecuzione:

Quando l'agente si trova in [1,1] non percepisce brezza

- $R_2: B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
- $R_4: \neg B_{1,1}$

Vogliamo dimostrare che $\alpha = \neg P_{2,1}$ è conseguenza di KB .

- $KB: R_2 \wedge R_4$
 - Ricordiamo che R_2 può essere trasformata in CNF ottenendo $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$
- $\alpha = \neg P_{2,1}$
 - Dobbiamo negare α e congiungerla a KB



- È stato dimostrato l'assurdo, quindi si può dire che $\neg P_{1,2}$ è una conseguenza delle prime quattro clausole, ovvero KB .

Clausole:

- $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) \wedge \neg B_{1,1} \wedge P_{1,2}$

CLAUSOLE SPECIALI:

Alcune basi di conoscenza nel mondo reale soddisfano certe restrizioni sulla forma delle formule che contengono (clausole).

- **Clausola definita:** Una disgiunzione di letterali in cui esattamente uno è positivo:

$$(\neg L_{1,1} \vee \neg Brezza \vee B_{1,1}) \quad \text{SI} \qquad (\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \quad \text{NO}$$

- **Clausola obiettivo:** Una disgiunzione di letterali in cui nessuno è positivo:

$$(\neg L_{1,1} \vee \neg Brezza \vee \neg B_{1,1}) \quad \text{SI}$$

- **Clausola di Horn:** Una disgiunzione di letterali in cui al **massimo uno è positivo**. Una clausola di Horn è una clausola definita o una clausola obiettivo. Sono chiuse rispetto alla risoluzione, cioè se si risolvono due clausole di Horn, si ottiene un'altra clausola di Horn.

PROPRIETÀ DELLE CLAUSOLE DEFINITE/HORN:

1. Le clausole definite possono essere scritte come un'implicazione:

- a. $(\neg L_{1,1} \vee \neg Brezza \vee B_{1,1}) \rightarrow (L_{1,1} \wedge Brezza) \Rightarrow B_{1,1}$

- b. La premessa è il **corpo** mentre la conclusione è la **testa**, nelle clausole di Horn. Una sentenza che contiene un singolo letterale vero, come $L_{1,1}$ è chiamata **fatto**.

2. L'inferenza delle clausole di Horn può essere svolta mediante gli algoritmi di **forward/backward chaining**;

3. Con le clausole di Horn è possibile determinare la conseguenza logica in **tempo lineare** rispetto alla dimensione della KB.

FORWARD CHAINING:

L'algoritmo di concatenazione in avanti (**forward chaining**) determina se un singolo simbolo proposizionale q (query) è conseguenza logica di una KB composta da clausole definite.

IDEA: per ogni clausola le cui premesse sono soddisfatte, aggiungi la sua conclusione all'insieme di fatti noti, finché non viene aggiunta q o non è più possibile effettuare alcuna inferenza.

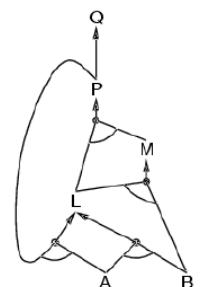
Esempio FC:

Voglio provare $Q = \text{true}$. L'algoritmo parte da ciò che è vero (ovvero A e B), ora cerca di vedere se un lato sinistro delle implicazioni è vero. Nell'esempio diventano vere le implicazioni in questo ordine:

5° implicazione \rightarrow 3° implicazione \rightarrow 2° implicazione \rightarrow 1° implicazione \rightarrow 4° implicazione.

Si tiene traccia per ogni formula quanti elementi della premessa diventano veri. Quando il conto è 0 (i lati sinistri) si aggiunge la conclusione all'agenda.

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B



BACKWARD CHAINING:

Determina se un singolo simbolo proposizionale q, è conseguenza logica di una KB composta da clausole definite. Se è già noto che q è vera non fa nulla, altrimenti verifica se tutte le premesse di qualche implicazione che ha q come conclusione sono vere attraverso **backward chaining**.

- Evita cicli poiché verifica se un nuovo subgoal è stato già inserito nello stack degli obiettivi;
- Evita la ripetizione del calcolo, verifica se un nuovo subgoal è stato già dimostrato vero o è già fallito;
- Anche in questo caso, un'implementazione efficiente verrà eseguita in tempo lineare.

In altre parole, si parte dal lato destro (query), e si va a ritroso (ai fatti).

Anche in questo caso, un'implementazione efficiente verrà eseguita in **tempo lineare**.

Esempio BC:

Nell'esempio precedente, suppongo $Q = \text{true}$ e vedo se c'è una implicazione con Q come lato destro, ovvero la 1° implicazione così da rendere vera P e faccio lo stesso con le altre implicazioni.

FORWARD vs BACKWARD:

Il **forward** segue un ragionamento guidato dai dati, l'attenzione parte dai fatti conosciuti, elaborazione automatica ed inconscia. Il **backward** segue un ragionamento basato sugli obiettivi, appropriato per il problem-solving.

Esempio basato sull'inferenza nel mondo Wumpus:

Un agente del mondo Wumpus che usa la logica proposizionale:

- $\neg P_{1,1}$ (nessun pozzo in [1,1] punto iniziale)
- $\neg W_{1,1}$ (nessun Wumpus in [1,1] punto iniziale)
- $B_{x,y} \leftrightarrow (P_{x,y+1} \vee P_{x,y-1} \vee P_{x+1,y} \vee P_{x-1,y})$ (Brezza -> pozzo in una cella vicina)
- $S_{x,y} \leftrightarrow (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y})$ (Fetore -> Wumpus in una cella vicina)
- $W_{1,1} \vee W_{1,2} \vee \dots \vee W_{4,4}$ (almeno 1 Wumpus)
- $\neg W_{1,1} \vee \neg W_{1,2}$ (al piu 1 Wumpus)
- $\neg W_{1,1} \vee \neg W_{8,9}$
- ... (tutte le combinazioni di coppie delle celle)

\Rightarrow 64 diversi simboli proposizionali, 155 sentenze

LIMITI DI ESPRESSIVITÀ DELLA LOGICA PROPOZIONALE:

La logica proposizionale è un linguaggio semplice che consiste di simboli proposizionali e connettivi logici. Può gestire proposizioni che sono note come vere, false o indefinite. Tuttavia, essa non è adatta ad ambienti di dimensione illimitata perché manca della potenza espressiva per affrontare in modo conciso tempo, spazio e schemi universali di relazioni tra oggetti. **Pertanto, la logica proposizionale non viene usata nella pratica perché il potere espressivo è bassissimo.**

La KB contiene sentenze fisiche per ogni cella ed è presente una proliferazione rapida di clausole.

8. LOGICA DEL PRIMO ORDINE

I vantaggi della logica proposizionale sono:

- **Linguaggio dichiarativo**, si gestiscono anche informazioni parziali (grazie all'uso della disgiunzione e della negazione);
- **Composizionale**, dove il significato di una formula è una funzione del significato delle sue parti;
- La semantica della logica proposizionale è **indipendente dal contesto**;

Lo svantaggio della logica proposizionale **non ha la potenza espressiva** per descrivere un ambiente con molti oggetti in modo conciso. Ad esempio, non è possibile dire (formalizzare) in modo semplice l'espressione "Nelle stanze adiacenti a quelle che contengono un pozzo si percepisce uno spostamento d'aria", è necessario quindi scrivere una sentenza per ogni cella $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$, mentre invece noi vorremmo un formalismo che in maniera sintetica rappresenti questo tipo di informazione.

LOGICA DEL PRIMO ORDINE:

Partendo dai concetti presenti nella logica proposizionale (semantica dichiarativa e compositiva, indipendenza dal contesto e non ambiguità). Costruire una logica **più espressiva** prendendo in prestito metodi di rappresentazione del linguaggio naturale (evitando le sue limitazioni).

La logica proposizionale assume che il mondo contiene **fatti** mentre nella logica del primo ordine, come nel linguaggio naturale, assume che il mondo contiene:

- **Oggetti**: persone, case, numeri, colori, guerre, partite a baseball
- **Relazioni**: legano due oggetti unarie, come rosso, tondo, falso, primo; o n-arie come fratello di, più grande di, all'interno di, parte di, avvenuto dopo
- **Funzioni**: padre di, migliore amico, uno più di, all'inizio di, ...

"Uno più due uguale a tre"

- ▶ Oggetti: uno, due, tre, uno più due
- ▶ Relazioni: uguale
- ▶ Proprietà: ----
- ▶ Funzioni: più

"Le stanze adiacenti a quella che contiene il Wumpus sono puzzolenti"

- ▶ Oggetti: stanze, Wumpus
- ▶ Relazioni: adiacenti
- ▶ Proprietà: puzzolente
- ▶ Funzioni: ----

SINTASSI E SEMANTICA:

I modelli della logica proposizionale collegano simboli a valori di verità predefiniti. I modelli della logica del primo ordine sono più interessanti (contengono oggetti) ed il dominio di un modello è l'insieme di oggetti che contiene (gli elementi del dominio, **oggetti** come persone, case, numeri, colori, guerre, partite a baseball).

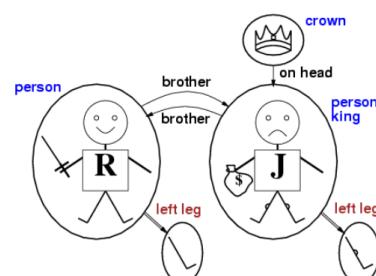
Gli oggetti del modello possono essere messi in relazione in molti modi attraverso una tupla, dove una relazione è un insieme di **tuple** di oggetti collegati:

- Relazioni unarie (**proprietà**): si applicano ad un oggetto: Persona (Maria);
- Relazioni binarie: collegano coppie di oggetti: FratelloDi (Maria, Giuseppe);
- Relazioni n-arie: collegano n oggetti.

Alcuni tipi di relazioni si possono considerare meglio come **funzioni** (prendono in input un oggetto e restituiscono un altro oggetto).

Quando ogni dato oggetto può essere collegato attraverso di esse ad esattamente un altro oggetto:

- GambaSinistra (Maria)
- GambaSinistra(Giuseppe)



Oggetti: Riccardo, Giovanni, Corona

Relazioni: FratelloDi(Riccardo,Giovanni), SullaTesta(Corona,Giovanni)

▶ Proprietà (Relazioni unarie): Persona(Riccardo), Re(Giovanni),...

Funzioni: GambaSinistra(Riccardo)

ELEMENTI SINTATTICI DI BASE:

Linguaggio: il vocabolario

- Costanti: Giovanni, A, Riccardo
- Predicati: Fratello, >, ...
- Funzioni: Radice quadrata, Gamba Sinistra
- Variabili: x,y,a,b
- Connettivi: \rightarrow , AND, OR, SSE, not
- Uguaglianza: =
- Quantificatori: \forall, \exists

$Sentence \rightarrow AtomicSentence \mid ComplexSentence$	$Quantifier \rightarrow \forall \mid \exists$
$AtomicSentence \rightarrow Predicate \mid Predicate(Term, \dots) \mid Term = Term$	$Constant \rightarrow A \mid X_1 \mid John \mid \dots$
$ComplexSentence \rightarrow (Sentence) \mid [Sentence]$	$Variable \rightarrow a \mid x \mid s \mid \dots$
$\mid \neg Sentence$	$Predicate \rightarrow True \mid False \mid After \mid Loves$
$\mid Sentence \wedge Sentence$	$Function \rightarrow Mother \mid LeftLeg \mid \dots$
$\mid Sentence \vee Sentence$	
$\mid Sentence \Rightarrow Sentence$	
$\mid Sentence \Leftrightarrow Sentence$	
$\mid Quantifier Variable, \dots Sentence$	
$Term \rightarrow Function(Term, \dots)$	$OPERATOR PRECEDENCE : \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow$
$\mid Constant$	
$\mid Variable$	

TERMINI:

Un **termine** è un'espressione logica che si riferisce ad un oggetto. La sintassi dei termini ben formati:

- **Termine** → Costante | Variabile | Funzione (Termine,...) un numero di termini pari alla arità della funzione.

Esempi di termini ben formati:

- x,A,B,2
- f(x,y)
- +(2,3)
- Padre-di (Giovanni)

FORMULA ATOMICA:

Una **formula atomica** è composta da un simbolo di predicato seguito da una lista di termini tra parentesi.

La sintassi delle formule ben formate:

- **Formula-atomica** → True | False | Termine = Termine | Predicato(Termine, ...) un numero di termini pari alla arità del predicato.

Esempi di formule atomiche ben formate:

- Ama(Giorgio, Lucia)
- +(2, 3) = 5
- Amico(Padre-di(Giorgio), Padre-di(Elena))

Una formula atomica è **vera** in un dato modello se la relazione a cui fa riferimento il simbolo predicato è verificata tra gli oggetti.

FORMULA COMPLESSA:

Una formula **complessa** è composta da formule atomiche che possono essere composte mediante i connettivi logici.

La sintassi delle formule ben formate:

- **Formula** → Formula-atomica | Formula Connuttivo | Formula | Quantificatore Variabile Formula | \neg Formula | (Formula)

Esempi di formule ben formate:

- Studia(Paolo) → Promosso(Paolo)
- Fratello(Riccardo, Giovanni) \wedge Fratello(Giovanni, Riccardo)

QUANTIFICATORI:

I **quantificatori** permettono di esprimere proprietà di intere collezioni di oggetti invece di enumerare gli oggetti per nome.

- **Universale - Per ogni (\forall)**: questo quantificatore ha la seguente forma $\forall < \text{Variabile} > < \text{Formula} >$.
Ad esempio:

Per ogni x , se x è un Re, allora x è una persona: $\forall x \text{Re}(x) \Rightarrow \text{Persona}(x)$

Dal punto di vista della semantica, $\forall x A(x)$ è vera in un dato modello m se A è vera in **tutte** le possibili interpretazioni estese costruite a partire dall'interpretazione fornita da m .

Inoltre, se il dominio è finito equivale ad un grosso AND, $\forall x \text{Mortale}(x)$ diventa $\text{Mortale(Gino)} \wedge \text{Mortale(Pippo)} \wedge \dots$

Tipicamente, siccome difficilmente una proprietà è universale, \forall si usa insieme a \Rightarrow $\forall x \text{Persona}(x) \Rightarrow \text{Mortale}(x)$.

Un errore comune da evitare: usare \wedge come connettivo principale di \forall :

$\forall x \text{Persona}(x) \wedge \text{Mortale}(x)$ significa ognuno è persona e ognuno è mortale.

- **Esistenziale - Esiste (\exists)**: formalmente il quantificatore esistenziale ha la seguente forma: $\exists < \text{Variabile} > < \text{Formula} >$.
Ad esempio:

Esiste una x corona tale che x è SullaTesta di Giovanni: $\exists x \text{Corona}(x) \wedge \text{SullaTesta}(x, \text{Giovanni})$

Dal punto di vista della semantica $\exists x A(x)$ è vera in un dato modello m se A è vera in **almeno** una interpretazione estesa che assegna x ad un elemento del dominio.

Inoltre, se il dominio è finito equivale a un grosso OR: $\exists x \text{Persona}(x)$ diventa $\text{Persona(Gino)} \vee \text{Persona(Pippo)} \vee \dots$

Un errore comune da evitare: usare \Rightarrow come connettivo principale di \exists :

$\exists x \text{Persona}(x) \Rightarrow \text{Mortale}(x)$ rappresenta una formula troppo debole.

QUANTIFICATORI ANNIDATI:

Quando i quantificatori sono dello stesso tipo si può scrivere il quantificatore una volta seguito da più variabili.

$\forall x, y \text{Fratello}(x, y) \Rightarrow \text{Consanguineo}(x, y)$

Quando i quantificatori sono diversi il loro ordine è importante:

- $\forall x (\exists y \underline{\text{Ama}(x, y)})$ Tutti amano qualcuno: la parte sottolineata indica l'ambito di y , la parte dentro la parentesi ambito di x ;
- $\exists y (\forall x \text{Ama}(x, y))$ Esiste qualcuno amato da tutti.

IL MONDO DEL WUMPUS IN LOGICA DEL PRIMO ORDINE:

Le costanti (fetore, brezza, scintillio, urto e urlo) sono raccolte in una lista.

Una tipica formula riguardante le percezioni sarà (predicato binario):

- $\text{Percezione}([\text{Fetore}, \text{Brezza}, \text{Scintillio}, \text{None}, \text{None}], 5)$

Quindi quando l'agente riceve le percezioni, alla KB si aggiunge il predicato binario composta dalla lista delle 5 costanti e l'istante temporale (all'istante 5 ha percepito fetore, brezza e scintillio).

Per determinare l'azione migliore si esegue la query:

- ASKVAR($\exists a \text{ BestAction}(a, 5)$), chiede la migliore azione all'istante 5;
- Restituisce una lista di legami come $\{a/\text{Afferra}\}$, dato che c'è scintillio la migliore azione è fare afferra.

Le percezioni implicano alcuni fatti sullo stato corrente:

- $\forall t, s, g, m, c \text{ Percezione}([s, \text{Brezza}, g, m, c], t) \Rightarrow \text{Brezza}(t)$
- $\forall t, s, b, m, c \text{ Percezione}([s, b, \text{Scintillio}, m, c], t) \Rightarrow \text{Scintillio}(t)$

Si prendono le percezioni e si costruiscono dei prediciati, ad esempio nella percezione brezza è vero può essere estratto in un predicato. Quando l'agente riceve le percezioni, aggiungiamo alla KB il predicato binario, composta dalla percezione e l'istante temporale.

Una volta costruiti i prediciati si possono definire degli assiomi che dicono cosa fare in base alla percezione che ha ricevuto l'agente. Semplici comportamenti reattivi possono essere implementati da formule di implicazione quantificata:

- $\forall t \text{ Scintilla}(t) \rightarrow \text{BestAction}(\text{Afferra}, t)$

Invece di usare nomi distinti per ogni locazione, usiamo un termine complesso in cui la riga e la colonna della locazione compaiono come indici interni (una lista). Le stanze adiacenti sono così definite:

$$\forall x, y, a, b \text{ Adiacente}([x, y], [a, b]) \leftrightarrow ((x=a) \text{ and } (y=b-1 \vee y=b+1)) \vee ((y=b) \text{ and } (x=a-1 \vee x=a+1))$$

dove $x=a$ indica stessa riga, $y=b$ indica stessa colonna.

Usiamo un **predicato unario** per rappresentare il pozzo e una **costante** per il Wumpus.

Proprietà associate alle locazioni, la posizione dell'agente cambia nel tempo:

- Pos (Agente, s, t), agente si trova all'istante t in posizione s.

Se l'agente in un determinato istante temporale sente fetore si trova in una cella puzzolente:

- $\forall s, t \text{ Pos}(\text{Agente}, s, t) \wedge \text{Fetore}(t) \Rightarrow \text{Puzzolente}(s)$

Conoscere questa proprietà ci permette di definire delle regole opportune.

Le locazioni ventose sono vicine ad un pozzo quindi la **regola diagnostica**, inferire la causa dall'effetto:

- $\forall s \text{ Ventosa}(s) \rightarrow \exists r \text{ Adiacente}(r, s) \text{ and } \text{Pozzo}(r)$

Regola causale, inferire l'effetto dalla causa (ragionamento basato sul modello):

- $\forall r \text{ Pozzo}(r) \rightarrow [\forall s \text{ Adiacente}(r, s) \rightarrow \text{Ventosa}(s)]$

Assioma per la freccia:

- $\forall t \text{ HaFreccia}(t+1) \rightarrow (\text{HaFreccia}(t) \wedge \neg \text{Azione}(\text{Scocca}, t))$

9. INFERNZA NELLA LOGICA DEL PRIMO ORDINE

Abbiamo già visto come sia possibile effettuare inferenze corrette e complete nella logica proposizionale. Vogliamo estendere tali risultati ed ottenere algoritmi corretti e completi per KB in logica del primo ordine. Ottenendo una risposta (se questa esiste) a qualsiasi domanda espressa in logica del primo ordine. Conversione in logica proposizionale, sfruttando semplici regole per trasformare formule quantificate in formule prive di quantificatori. Quindi abbiamo una formulazione di metodi di inferenza in grado di manipolare direttamente formule nel primo ordine.

REGOLA DI INFERNZA PER \forall :

Istanziazione universale (\forall eliminazione)

- Sostituendo un termine ground (privo di variabili) alla variabile

$$\frac{\forall v \ \alpha}{\text{SUBST}(\{v/g\}, \alpha)} \quad \text{SUBST}(\theta, \alpha) \text{ applicazione della sostituzione } \theta \text{ alla formula } \alpha$$

per ogni variabile v e termine ground g

Ad esempio vediamo come si possono ottenere nuove formule sostituendo: *tutti i re avidi sono malvagi*

- $\forall x \text{Re}(x) \wedge \text{Avido}(x) \Rightarrow \text{Malvagio}(x)$
- $\text{Re}(\text{Giovanni}) \wedge \text{Avido}(\text{Giovanni}) \Rightarrow \text{Malvagio}(\text{Giovanni})$
 $\text{Re}(\text{Riccardo}) \wedge \text{Avido}(\text{Riccardo}) \Rightarrow \text{Malvagio}(\text{Riccardo})$
- $\text{Re}(\text{Fratello}(\text{Giovanni})) \wedge \text{Avido}(\text{Fratello}(\text{Giovanni})) \Rightarrow \text{Malvagio}(\text{Fratello}(\text{Giovanni}))$
- ...

REGOLA DI INFERNZA PER \exists :

Istanziazione esistenziale (\exists eliminazione)

- Sostituendo un nuovo simbolo costante (unico, che non appare in nessun'altra parte della KB) alla variabile

$$\frac{\exists v \ \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

per ogni sentenza α , variabile v e costante k

Ad esempio vediamo come si può ottenere una nuova formula sostituendo: *Giovanni ha una corona sulla testa*

- $\exists x \text{Corona}(x) \wedge \text{SullaTesta}(x, \text{Giovanni})$
- $\text{Corona}(\text{C}_1) \wedge \text{SullaTesta}(\text{C}_1, \text{Giovanni})$

Il nuovo simbolo costante introdotto nella sostituzione è detto **costante di Skolem**

EQUIVALENZA:

L'istanziazione universale può essere applicata più volte per aggiungere nuove sentenze, la nuova KB è **logicamente equivalente** a quella originale.

L'istanziazione esistenziale può essere applicata una sola volta per sostituire la sentenza esistenziale.

La nuova KB non è logicamente equivalente alla vecchia, ma risulta essere soddisfacibile (vera in almeno un modello) se e soltanto se la KB originale lo era (inferenzialmente equivalente).

L'algoritmo di Skolem non mantiene l'equivalenza semantica.

RIDUZIONE ALL'INFERNZA PROPOSIZIONALE:

Utilizzando le regole viste prima, diventa possibile ridurre l'inferenza del primo ordine a quella proposizionale.

Supponiamo che la KB contenga:

$$\forall x \text{Re}(x) \wedge \text{Avido}(x) \Rightarrow \text{Malvagio}(x)$$

$$\text{Re}(\text{Giovanni})$$

$$\text{Avido}(\text{Giovanni})$$

$$\text{Fratello}(\text{Riccardo}, \text{Giovanni})$$

Istanziando la formula universale **in tutti i modi possibili** otteniamo:

$$\text{Re}(\text{Giovanni}) \wedge \text{Avido}(\text{Giovanni}) \Rightarrow \text{Malvagio}(\text{Giovanni})$$

$$\text{Re}(\text{Riccardo}) \wedge \text{Avido}(\text{Riccardo}) \Rightarrow \text{Malvagio}(\text{Riccardo})$$

$$\text{Re}(\text{Giovanni})$$

$$\text{Avido}(\text{Giovanni})$$

$$\text{Fratello}(\text{Riccardo}, \text{Giovanni})$$

questa è una logica del primo ordine FOL (first order logic) e facendo l'istanziamento per togliere il \forall si vanno a cambiare i predicati con tutti gli oggetti.

PROPOZIONALIZZAZIONE:

La tecnica appena applicata è detta **proposizionalizzazione**. Può essere applicata in modo assolutamente.

Ad esempio, i simboli proposizionali ottenuti nell'esempio precedente sono:

$$\text{Re}(\text{Giovanni}), \text{Avido}(\text{Giovanni}), \text{Malvagio}(\text{Giovanni}), \text{Re}(\text{Riccardo}), \dots$$

A questo punto possiamo trattare la KB come proposizionale e applicare gli algoritmi visti. Il problema è che le costanti sono in numero finito, ma nel caso delle funzioni il processo non funziona bene, perché potrei ottenere infiniti oggetti.

Se la KB contiene il simbolo funzione *Padre*, il numero di istanze da creare è infinito

$$\text{Giovanni}, \text{Padre}(\text{Giovanni}), \text{Padre}(\text{Padre}(\text{Giovanni})) \dots$$

TEOREMA DI HERBRAND:

Se una formula α segue logicamente dalla KB originale (quella in logica FOL), allora ci sarà una dimostrazione che coinvolge solo un sottoinsieme finito della KB proposizionalizzata.

Si può procedere incrementalmente:

- Creare le istanze con le costanti;
- Creare poi tutti i termini di profondità 1;
- Poi profondità 2;
- Finché la formula α non è conseguenza logica della KB costruita fino a quel momento.

L'algoritmo è **completo** poiché ogni formula che segue logicamente può essere dimostrata. Il problema si pone quando α non è conseguenza logica, poiché l'algoritmo andrà all'infinito introducendo il **problema di semidecidibilità**.

SEMIDECIDIBILITÀ IN FOL:

Il problema della conseguenza logica, per la logica del primo ordine è **semidecidibile**: questo significa che esistono algoritmi che rispondono affermativamente per ogni formula che è conseguenza logica, ma nessun algoritmo potrà rispondere negativamente per ogni formula che non è conseguenza logica.

PROBLEMI CON LA PROPOZITIONALIZZAZIONE:

Sembra che la proposizionalizzazione generi una grossa quantità di sentenze irrilevanti.

Ad esempio considerando la seguente KB:

$$\forall x \text{Re}(x) \wedge \text{Avido}(x) \Rightarrow \text{Malvagio}(x)$$

Re(Giovanni)

∀y Avido(y)

Fratello(Riccardo, Giovanni)

Sembra ovvio che si riesca ad ottenere *Malvagio(Giovanni)*, tuttavia vengono prodotti una serie di **fatti irrilevanti**, come *Avido(Riccardo)*. Con p predicati di aritità k ed n costanti, ci saranno $p \cdot n^k$ istanziazioni.

9.1 UNIFICAZIONE

Bisogna cercare un altro modo migliore per andare a provare delle query e non generando istanze irrilevanti, quindi bisogna lavorare non in termini proposizionali ma in FOL. Ma il problema della FOL è rendere uguali parti delle formule.

Il processo di **unificazione** determina se due espressioni possono essere rese identiche mediante una sostituzione di termini alle variabili. È un componente chiave di tutti gli algoritmi di inferenza del primo ordine.

L'algoritmo **UNIFY** prende due formule e, se esiste, restituisce un loro unificatore:

- **UNIFY** (p, q) = θ se **SUBST**(θ, p) = **SUBST**(θ, q)

Possiamo inferire che esiste una sostituzione θ per la quale *Re(x)* e *Avido(x)* corrispondono a *Re(Giovanni)* e *Avido(y)*.

- $\theta = \{x/Giovanni, y/Giovanni\}$ funziona

L'algoritmo prende due formule, e mi restituisce se esiste, un theta (θ), la quale rappresenta una sostituzione di variabili che fa sì che p e q diventino uguali.

Esempio:

QUERY = Chi conosce Giovanni? = **Conosce (Giovanni,x)**

p	q	θ
<i>Conosce(Giovanni,x)</i>	<i>Conosce(Giovanni, Giacomina)</i>	$\{x/Giacomina\}$
<i>Conosce(Giovanni,x)</i>	<i>Conosce(y, Guglielmo)</i>	$\{x/Guglielmo, y/Giovanni\}$
<i>Conosce(Giovanni,x)</i>	<i>Conosce(y, Madre(y))</i>	$\{y/Giovanni, y/Madre(Giovanni)\}$
<i>Conosce(Giovanni,x)</i>	<i>Conosce(x, Elisabetta)</i>	<i>fallimento</i>

L'ultima unificazione fallisce perché x non può assumere contemporaneamente i valori Giovanni ed Elisabetta.

STANDARDIZZAZIONE SEPARATA:

Tutto ciò può essere evitato rinominando le variabili di una delle formule per evitare collisioni. Questa operazione prende il nome di **standardizzazione separata**.

Nell'esempio, si può rinominare la x di *Conosce(x, Elisabetta)* in z_{17} senza modificarne il significato:

$$\text{UNIFY}(\text{Conosce (Giovanni,x)}, \text{Conosce}(z_{17}, \text{Elisabetta})) = \{x / \text{Elisabetta}, z_{17} / \text{Giovanni}\}$$

9.2 MODUS PONENS GENERALIZZATO(GMP)

Per tutte le formule atomiche p_i , p'_i e q quando esiste una sostituzione θ tale che $\text{SUBST}(\theta, p'_i) = \text{SUBST}(\theta, p_i)$ per tutti gli i , si può applicare la regola di inferenza chiamata **modus ponens generalizzato (gmp)**.

$$\frac{p_1', p_2', \dots, p_n', (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

Sappiamo che tutti i p'_i sono veri (presenti in KB) e supponiamo di avere l'implicazione nella formula sopra, nella logica proposizionale si andava ad aggiungere q nella KB. In questo caso, poiché la formula può avere delle variabili si applica l'unificazione, trovando la sostituzione che rende identiche $p'_i = p_i$ tramite θ .

Quindi in KB non viene aggiunto q ma $\text{SUBST}(\theta, q)$.

Applicando tale regola al nostro esempio:

p_1' è $\text{Re}(Giovanni)$	p_1 è $\text{Re}(x)$
p_2' è $\text{Avido}(y)$	p_2 è $\text{Avido}(x)$
θ è $\{x/Giovanni, y/Giovanni\}$	q è $\text{Malvagio}(x)$
$\text{SUBST}(\theta, q)$ è $\text{Malvagio}(Giovanni)$	

FORWARD CHAINING:

Abbiamo già visto un algoritmo di questo tipo per clausole definite proposizionali.

1. Si parte con le formule atomiche nella KB,
2. Si applica **Modus Ponens** in avanti;
3. Si aggiungono nuove formule atomiche, finché non è più possibile compiere altre inferenze.

Vedremo come si può applicare l'algoritmo a clausole definite del primo ordine, le quali sono definite come **Situazione → Risposta** le quali sono particolarmente utili per sistemi che effettuano inferenze non appena ricevono nuove informazioni.

CLAUSOLE DEFINITE DEL PRIMO ORDINE:

Le **clausole definite** del primo ordine sono molto simili a quelle proposizionali, ovvero una disgiunzione di letterali dove esattamente uno dei quali è positivo. Una clausola definita può essere:

- Una **formula atomica**: $\text{Re}(Giovanni)$;
- Un'**implicazione** in cui l'antecedente è una congiunzione di letterali positivi e la conseguenza è un singolo letterale positivo: $\text{Re}(x) \wedge \text{Avido}(x) \Rightarrow \text{Malvagio}(x)$;

Le variabili presenti in questa forma sono quantificate con il quantificatore universale; quindi, non possiamo usare i quantificatori esistenziali; quindi, con le clausole definite non possiamo definire tutto ciò che è definibile con la logica del primo ordine; quindi, non è possibile trasformare qualcosa della logica del primo ordine in una forma tale da poter applicare l'algoritmo di forward chaining.

A differenza delle clausole definite proposizionali, quelle del primo ordine possono includere variabili, che si considerano sempre **quantificate universalmente**. Non tutte le KB possono essere convertite in un insieme di clausole definite a causa della restrizione sulla presenza di un solo letterale positivo.

Esempio di KB usando il forwardchaining:

Consideriamo il seguente problema:

La legge americana afferma che per un cittadino è un crimine vendere armi ad una nazione ostile. Lo stato di Nono, un nemico dell'America, possiede dei missili e gli sono stati venduti tutti dal Colonnello West, un americano.

Dimostreremo che West è un criminale:

Per un americano è un crimine vendere armi ad una nazione ostile:

$$\text{Americano}(x) \wedge \text{Arma}(y) \wedge \text{Vende}(x,y,z) \wedge \text{Ostile}(z) \Rightarrow \text{Criminale}(x)$$

Nono possiede dei missili ovvero $\exists x \text{ Possiede}(Nono, x) \wedge \text{Missile}(x)$: $\text{Possiede}(Nono, M_1) \wedge \text{Missile}(M_1)$

Tutti i missili sono stati venduti dal Colonnello West: $\text{Missile}(x) \wedge \text{Possiede}(Nono, x) \Rightarrow \text{Vende}(West, x, Nono)$

I missili sono armi:

$$\text{Missile}(x) \Rightarrow \text{Arma}(x)$$

Un nemico dell'America viene considerato ostile:

$$\text{Nemico}(x, America) \Rightarrow \text{Ostile}(x)$$

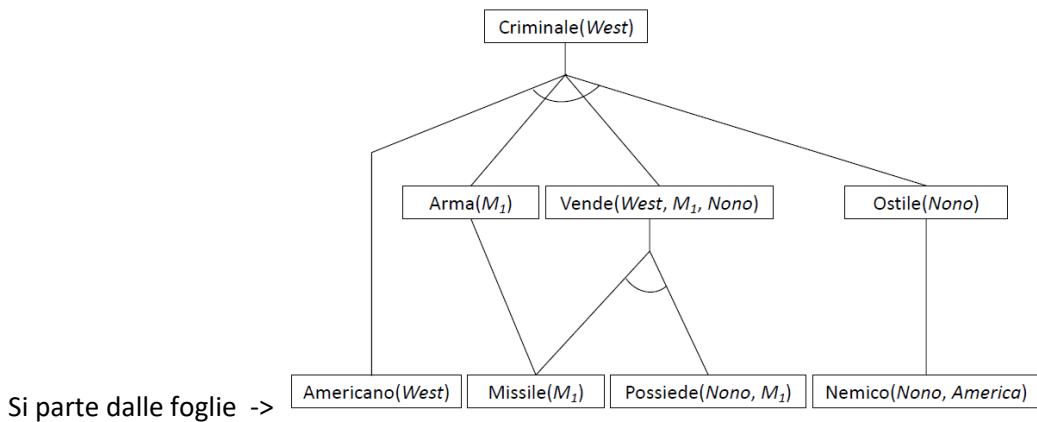
West è americano:

$$\text{Americano}(West)$$

Lo stato di Nono è un nemico dell'America:

$$\text{Nemico}(Nono, America)$$

Questa KB non contiene simboli di funzione ed è un esempio di KB che prende il nome di **Datalog**, rende l'inferenza molto più semplice.



L'algoritmo è **corretto**: ogni inferenza è un'applicazione del **Modus Ponens** ed è anche **completo**.

Nel caso di KB Datalog (non contengono simboli di funzione) l'algoritmo termina in tempo polinomiale. Sia **k** la massima **arità** tra tutti i predici, **p** il numero di predici ed **n** quello dei simboli costante, non possono esserci più di **pn^k** fatti distinti, quindi dopo questo numero di iterazioni l'algoritmo deve aver raggiunto il punto fisso.

BACKWARD CHAINING:

In questo tipo di algoritmi si seguono le regole a ritroso per trovare fatti noti che supportino la dimostrazione. Questo tipo di algoritmo è molto utilizzato nella programmazione logica (la forma più diffusa di ragionamento automatico).

L'algoritmo costruisce un albero and-or, perché possiamo avere tante clausole che hanno la stessa formula e cerchiamo più implicazioni per vedere quale dimostra quella proposizione.

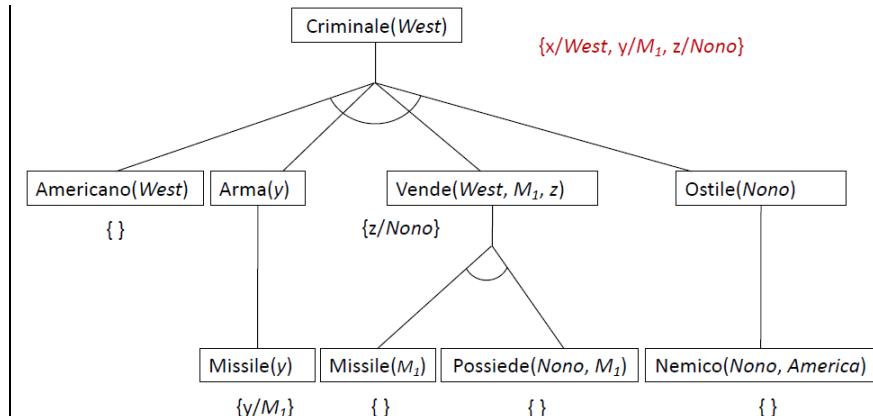
La parte **OR** si deve al fatto che la query obiettivo può essere dimostrata da qualsiasi regola della KB, opera cercando tutte le clausole che potrebbero unificare con l'obiettivo. Standardizzando le variabili nelle clausole in modo che risultino variabili del tutto nuove e verificando se la **rhs** (right hand side) della clausola unifica con l'obiettivo. La parte **AND** si deve al fatto che tutti i congiunti della lista **Ihs** (left hand side) di una clausola devono essere dimostrati.

Il **vantaggio** di questo algoritmo è dato dal fatto che evitiamo conoscenza inutile ma soprattutto, questo tipo di visita in profondità ha il vantaggio del risparmio dello spazio di memoria.

Esempio backward chaining:

Si parte dalla root ->

- Al primo sottoalbero scopriamo x=West;
- Al secondo sottoalbero invece y=M₁;
- Mentre all'ultimo sottoalbero sarà z=Nono.



PROPRIETÀ DEL BACKWARD CHAINING:

La soluzione vista per il backward è chiaramente un algoritmo di ricerca in profondità, i suoi requisiti spaziali sono lineari nella dimensione della dimostrazione. Potremmo avere incompletezza dovuta a cicli infiniti (è possibile risolvere questo problema effettuando un controllo sul goal corrente versus gli altri goal memorizzati in uno stack).

Inefficienza dovuta alla ripetizione dei subgoal (sia in caso di successo che di fallimento) ma è possibile risolvere questo problema effettuando caching dei risultati precedenti (spazio extra).

9.3 ALGORITMO DI RISOLUZIONE

Abbiamo visto che la risoluzione è una procedura di inferenza completa per refutazione nella logica proposizionale.

- **Teorema di Deduzione:** KB |= α sé e soltanto se (KB → α) è valida [cioè KB ∧ ¬α è insoddisfacibile];
- **Estendiamo** la sua applicazione alla logica del primo ordine.

La regola di risoluzione per le clausole del primo ordine è una versione “sollevata” di quella proposizionale:

$$\frac{l_1 \vee l_2 \vee \dots \vee l_k, \quad m_1 \vee m_2 \vee \dots \vee m_n}{\text{SUBST}(\theta, l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n)}$$

dove UNIFY(l_i, ¬m_j) = θ

Due clausole, che grazie alla standardizzazione separata presupponiamo sempre non avere alcuna variabile in comune, possono essere risolte se contengono letterali complementari.

I letterali proposizionali sono complementari se uno è la negazione dell'altro; quelli di primo ordine lo sono se uno unifica con la negazione dell'altro.

Esempio:

$$\begin{aligned} & [\text{Animale}(F(x)) \vee \text{Ama}(G(x), x)] \text{ e } [\neg \text{Ama}(u, v) \vee \neg \text{Uccide}(u, v)] \\ & [\text{Animale}(F(x)) \vee \neg \text{Uccide}(G(x), x)] \text{ con unificatore } \theta = \{u/G(x), \\ & v/x\} \end{aligned}$$

CONVERSIONE IN CNF:

È possibile applicare passi di risoluzione solo a formato CNF ($KB \wedge \neg \alpha$), questa metodologia di risoluzione risulta completa per KB in logica del primo ordine.

Vediamo, allora, come convertire le formule in **forma normale congiuntiva**, una congiunzione di clausole, ognuna delle quali è formata da una disgiunzione di letterali. I letterali possono contenere variabili, le quali sono sempre considerate universalmente quantificate.

Esempio:

La formula

$$\forall x \text{ Americano}(x) \wedge \text{Arma}(y) \wedge \text{Vende}(x, y, z) \wedge \text{Ostile}(z) \Rightarrow \text{Criminale}(x)$$

Diventa in CNF

$$\neg \text{Americano}(x) \vee \neg \text{Arma}(y) \vee \neg \text{Vende}(x, y, z) \vee \neg \text{Ostile}(z) \vee \text{Criminale}(x)$$

La procedura per la conversione in CNF è molto simile a quella che abbiamo visto per il caso proposizionale, la differenza principale sorge dalla necessità di eliminare i quantificatori esistenziali.

Illustriamo per passi la procedura generale, traducendo la seguente formula:

$$\begin{aligned} & \text{Chiunque ami tutti gli animali è amato da qualcuno} \Rightarrow \\ & \forall x [\forall y \text{ Animale}(y) \Rightarrow \text{Ama}(x, y)] \Rightarrow [\exists z \text{ Ama}(z, x)] \end{aligned}$$

I passi sono i seguenti:

- Eliminazione delle implicazioni:

$$\forall x [\neg \forall y \neg \text{Animale}(y) \vee \text{Ama}(x, y)] \vee [\exists z \text{ Ama}(z, x)]$$

- Spostamento all'interno delle negazioni: $\neg \forall x p \equiv \exists x \neg p$ e $\neg \exists x p \equiv \forall x \neg p$:

$$\forall x [\exists y \neg (\neg \text{Animale}(y) \vee \text{Ama}(x, y))] \vee [\exists x \text{ Ama}(y, x)]$$

$$\forall x [\exists y \neg \neg \text{Animale}(y) \wedge \neg \text{Ama}(x, y)] \vee [\exists x \text{ Ama}(y, x)]$$

$$\forall x [\exists y \text{ Animale}(y) \wedge \neg \text{Ama}(x, y)] \vee [\exists x \text{ Ama}(y, x)]$$

- Standardizzazione delle variabili: ogni quantificatore deve usare una variabile diversa:

$$\forall x [\exists y \text{ Animale}(y) \wedge \neg \text{Ama}(x, y)] \vee [\exists z \text{ Ama}(z, x)]$$

- Skolemizzazione: una forma più generale di istanziazione esistenziale. Ogni variabile esistenziale viene sostituita da una funzione di Skolem che include variabili universalmente quantificate:

$$\forall x [\text{Animale}(F(x)) \wedge \neg \text{Ama}(x, F(x))] \vee [\text{Ama}(G(x), x)]$$

- Omissione di quantificatori universali:

$$[\text{Animale}(F(x)) \wedge \neg \text{Ama}(x, F(x))] \vee [\text{Ama}(G(x), x)]$$

- Distribuzione di \vee su \wedge :

$$[\text{Animale}(F(x)) \vee \text{Ama}(G(x), x)] \wedge [\neg \text{Ama}(x, F(x))] \vee \text{Ama}(G(x), x)$$

Ora la formula è in CNF ed è composta da due clausole:

Ogni formula della logica del primo ordine può essere convertita in una formula CNF inferenzialmente equivalente.

Esempio di dimostrazione:

