

Intelligenza Artificiale

UniVR - Dipartimento di Informatica

Fabio Irimie

1° Semestre 2025/2026

Indice

1	Introduzione	3
1.1	Tipi di intelligenza artificiale	3
1.1.1	Autonomous agents	3
1.1.2	Data analysis	3
1.1.3	Machine Learning	3
1.1.4	Time series analysis	4
1.1.5	Intelligent Agents	4
1.2	Markov Decision Process (MDP)	4
1.3	Generative AI	5
2	Agenti e ambiente	5
2.1	Razionalità	6
2.2	PEAS	7
2.3	Tipi di ambienti	8
2.4	Agenti di problem solving	8
3	Ricerca nello spazio degli stati	10
3.1	Ricerca generale	10
3.1.1	Tree search	10
3.1.2	Stato e nodo	10
3.1.3	Tree search generale	10
3.1.4	Stati ripetuti	11
3.2	Ricerca non informata	11
3.2.1	Breadth-first search	12
3.2.2	Uniform-cost search	12
3.2.3	Depth-first search	13
3.2.4	Iterative deepening search	13
3.3	Ricerca informata	15
3.3.1	Best-first search	15
3.3.2	Greedy best-first search	15
3.3.3	A* search	16
3.3.4	Consistenza e ammissibilità	16
3.3.5	Euristiche	17
3.4	Ricerca locale	18
3.4.1	Hill climbing	19
3.4.2	Simulated annealing	20
3.4.3	Local beam search	21
3.4.4	Algoritmi genetici	21
3.5	Ricerca locale in uno spazio continuo	22
3.5.1	Gradient ascent/descent	22
3.5.2	Algoritmo di Newton-Raphson	23
3.5.3	Calcolo degli zeri del gradiente	24
3.5.4	Gradiente empirico	24
3.6	Constrained satisfaction problem	25
3.6.1	Grafo dei vincoli	26
3.6.2	Problemi combinatori	28
3.6.3	Backtracking search	29
3.6.4	Inferenza	30

3.6.5	Look Ahead	31
3.6.6	Forward checking look ahead	31
3.6.7	Arc consistency look ahead	32
3.6.8	Forzare l'arc consistency	33
3.6.9	Tree decomposition	34
4	Logical Agents	34
4.1	Knowledge based agents	34
4.2	Logica in generale	36
4.2.1	Entailment (Derivazione logica)	37
4.2.2	Inferenza	37
4.2.3	Inferenza mediante enumerazione	38
4.2.4	Metodi di dimostrazione	38
4.2.5	Equivalenza logica	39
4.2.6	Validità e soddisfacibilità	39
4.3	Sistema di inferenza	40
4.3.1	Proprietà di un sistema di inferenza	40
4.4	Problema di deduzione	41
4.4.1	Resolution	41
4.4.2	Conversione in CNF	41
4.5	Forward e backward chaining	42
4.5.1	Forward chaining	43
4.5.2	Backward chaining	43
4.5.3	Differenze tra forward e backward chaining	44
5	Rappresentare l'incertezza	44
5.1	Probabilità	44
5.1.1	Decidere con incertezza	45
5.1.2	Basi di probabilità	45
5.1.3	Variabili casuali	46
5.1.4	Eventi atomici	46
5.1.5	Probabilità a priori	46
5.1.6	Probabilità congiunta	46
5.1.7	Probabilità continua	47
5.1.8	Probabilità condizionata	47
5.1.9	Inferenza per enumerazione	48
5.1.10	Normalizzazione	49
5.2	Indipendenza	50
5.2.1	Indipendenza condizionata	50

1 Introduzione

Nel 1950 Alan Turing pubblica un articolo intitolato "Computing Machinery and Intelligence" in cui propone un esperimento per determinare se una macchina può essere considerata intelligente. L'esperimento, noto come "test di Turing", coinvolge un interrogatore umano che comunica con due entità nascoste: una macchina e un essere umano. L'interrogatore deve fare domande a entrambe le entità e, basandosi sulle risposte, deve determinare quale delle due è la macchina. Se l'interrogatore non riesce a distinguere tra le risposte della macchina e quelle dell'essere umano, la macchina è considerata intelligente.

In futuro l'attenzione si è spostata sulla ricerca di metodi per risolvere problemi che richiedono intelligenza umana, utilizzando algoritmi e modelli matematici fino ad arrivare alle reti neurali e intelligenza artificiale.

Definizione 1.1. L'intelligenza artificiale è una disciplina che studia come **simulare** l'intelligenza umana in scenari complessi

1.1 Tipi di intelligenza artificiale

1.1.1 Autonomous agents

Sono sistemi che percepiscono l'ambiente e agiscono in modo autonomo per raggiungere obiettivi specifici.

1.1.2 Data analysis

Utilizzo di algoritmi per analizzare grandi quantità di dati e estrarre informazioni utili e correlazioni complesse.

1.1.3 Machine Learning

È lo sviluppo di algoritmi che permettono a dei modelli di apprendere dai dati di esempio e migliorare le loro prestazioni nel tempo senza essere esplicitamente programmati. Ad esempio riconoscimento di immagini.

L'apprendimento automatico è diviso in tre categorie principali:

- **Unsupervised learning:** il modello viene addestrato su un insieme di dati non etichettati, dove l'obiettivo è scoprire strutture nascoste o pattern nei dati senza avere risposte corrette predefinite.
- **Supervised learning:** il modello viene addestrato su un insieme di dati etichettati, dove ogni esempio di input è associato a una risposta corretta. L'obiettivo è che il modello impari a mappare gli input alle risposte corrette.
- **Reinforced learning:** il modello impara attraverso interazioni con l'ambiente, ricevendo ricompense o penalità in base alle azioni intraprese. L'obiettivo è massimizzare la ricompensa totale nel tempo.

1.1.4 Time series analysis

L'analisi delle serie temporali è un'area dell'apprendimento automatico che si concentra sull'analisi di dati collezionati nel tempo. Le serie temporali sono sequenze di dati misurati a intervalli regolari, come temperatura giornaliera, prezzi delle azioni o dati di vendita mensili. L'obiettivo dell'analisi delle serie temporali è identificare pattern, tendenze e stagionalità nei dati per fare previsioni future.

Gli approcci comuni per l'analisi delle serie temporali includono:

- **Riconoscimento di anomalie e cause:** è un processo di identificazione di dati o eventi che si discostano significativamente dal comportamento normale o atteso. Queste anomalie possono indicare problemi, errori o situazioni insolite che richiedono attenzione.
- **Generative transformers:** sono una classe di modelli che permettono di predire il prossimo elemento in una sequenza di dati partendo dagli elementi precedenti, come ad esempio la parola successiva in una frase o il pixel successivo in un'immagine. Si sfrutta il concetto di **attenzione** per pesare l'importanza relativa delle diverse parti della sequenza di input durante la generazione dell'output.

1.1.5 Intelligent Agents

Un agente intelligente è un sistema che percepisce l'ambiente circostante attraverso sensori e agisce su l'ambiente per raggiungere un obiettivo specifico. Gli elementi chiave di un agente intelligente includono:

- **Performance measure:** misura il successo dell'agente nel raggiungere i suoi obiettivi
- **Rationality:** l'agente deve agire in modo da massimizzare la sua performance measure attesa

1.2 Markov Decision Process (MDP)

Un MDP è un modello matematico utilizzato per rappresentare problemi di decisione sequenziali. Gli elementi principali sono:

- **State:** rappresenta l'ambiente in un dato momento
- **Actions:** insieme delle azioni che l'agente può intraprendere
- **Transition model:** effetto che le azioni hanno sull'ambiente (potrebbero essere parzialmente incognite)

$$T : (\text{state}, \text{action}) \rightarrow \text{next_state}$$

- **Reward:** valore **immediato** dell'esecuzione di un'azione

$$R : (\text{state}, \text{action}, \text{next_state}) \rightarrow \text{real_number}$$

- **Policy:** strategia che l'agente utilizza per decidere quale azione intraprendere in ogni stato con l'obiettivo di massimizzare la ricompensa totale attesa nel tempo

$$\pi : (\text{state}) \rightarrow \text{action}$$

1.3 Generative AI

L'intelligenza artificiale generativa si riferisce a una classe di modelli di intelligenza artificiale che sono in grado di generare nuovi contenuti, come testo, immagini, musica o video, a partire da dati di addestramento. Questi modelli hanno miliardi di parametri e sono **preaddestrati** su grandi quantità di dati. In sostanza questi modelli "predicono il futuro" basandosi sui dati su cui sono stati addestrati e un **prompt** (input dell'utente).

2 Agenti e ambiente

Gli agenti includono umani, robot, softbot, termostati ecc... La funzione dell'agente mappa lo storico di percezioni in azioni:

$$f : \mathcal{P}^* \mapsto \mathcal{A}$$

Il programma dell'agente è eseguito su architettura fisica per produrre la funzione f .

Esempio 2.1. Un esempio potrebbe essere un insieme di stanze $\{A, B\}$ e un robot aspirapolvere che può percepire la sua posizione e il contenuto della stanza. L'agente potrebbe quindi percepire $[A, \text{Sporco}]$ se ci fosse dello sporco nella stanza A. Le azioni potrebbero essere di movimento o pulizia. Tutto questo dipende dalla sequenza di percezioni, ad esempio in una tabella:

Percezione	Azione
$[A, \text{Pulito}]$	Vai a B
$[A, \text{Sporco}]$	Pulisci
$[B, \text{Pulito}]$	Vai ad A
$[B, \text{Sporco}]$	Pulisci
$[A, \text{Pulito}], [A, \text{Pulito}]$	Vai a B
$[A, \text{Pulito}], [A, \text{Sporco}]$	Pulisci

Tabella 1: Esempio di tabella di percezioni e azioni

Non possiamo dire se questa è una funzione corretta perchè non abbiamo una **performance measure** che ci dica se l'agente sta facendo un buon lavoro.

Definizione 2.1. Se un agente ha $|\mathcal{P}|$ possibili percezioni, allora al tempo T avrà:

$$\sum_{t=1}^T |\mathcal{P}|^t$$

Se lo storico di percezioni è irrilevante, cioè se ad ogni percezione è associata un'azione la funzione viene chiamata **Reflex**.

2.1 Razionalità

Per definire l'intelligenza di un agente si utilizza una misura di performance che valuta la sequenza di percezioni.

Esempio 2.2. Tornando all'esempio del robot aspirapolvere si potrebbero assegnare i seguenti punteggi:

- Un punto per ogni stanza pulita per ogni unità di tempo
- Meno un punto per ogni mossa
- Penalizzazione per ogni stanza sporca

Esempio 2.3. Un altro esempio è il seguente ambiente:

- Ci sono 3 stanze (A, B, C) e due robot (r_1, r_2)
- r_1 può sorvegliare solo A e B e r_2 solo B e C
- r_1 inizia dalla stanza A e r_2 dalla C
- Il tempo di percorrenza tra le stanze è 0
- Performance measure: minimizza il tempo in cui una stanza non è sorvegliata, cioè il tempo totale in cui una stanza non è visitata da nessun robot

Un possibile comportamento razionale potrebbe essere il seguente (alternata):

Stato	A	B	C	Tempo
[A, C]	0	1	0	1
[B, C]	1	0	0	2
[A, C]	0	1	0	3
[A, B]	0	0	1	4
Average idleness	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{4}$	Tot: $\frac{1}{3}$

Un altro comportamento potrebbe essere (fissata):

Stato	A	B	C	Tempo
[A, C]	0	1	0	1
[B, C]	1	0	0	2
[A, C]	0	1	0	3
[B, C]	1	0	0	4
Average idleness	$\frac{1}{2}$	$\frac{1}{2}$	0	Tot: $\frac{1}{3}$

Entrambi i comportamenti hanno la stessa performance measure, ma il primo è migliore del secondo perchè penalizza meno una singola stanza rispetto alle

altre. Per capirlo bisogna non solo minimizzare la performance measure, ma anche minimizzare la varianza.

2.2 PEAS

Per progettare un agente intelligente bisogna definire l'ambiente in cui opera:

- **Performance measure:** come viene valutato il successo dell'agente
- **Environment:** il contesto in cui l'agente opera
- **Actuators:** i mezzi attraverso cui l'agente agisce sull'ambiente
- **Sensors:** i mezzi attraverso cui l'agente percepisce l'ambiente

Esempio 2.4. Prendiamo ad esempio un taxi automatico, il PEAS potrebbe essere:

- Performance measure:
 - Soddisfazione del cliente
 - Sicurezza
 - Efficienza del carburante
 - Rispetto delle leggi stradali
- Environment:
 - Traffico stradale
 - Condizioni meteorologiche
 - Segnali stradali
 - Pedoni e altri veicoli
- Actuators:
 - Volante
 - Acceleratore
 - Freni
 - Indicatori di direzione
- Sensors:
 - Telecamere
 - Lidar
 - Radar
 - Sensori di velocità
 - GPS

2.3 Tipi di ambienti

Gli ambienti possono essere classificati in base a diverse caratteristiche:

- **Osservabile:** se l'agente può percepire completamente lo stato dell'ambiente in ogni momento
- **Deterministico:** se l'azione dell'agente determina in modo univoco il prossimo stato dell'ambiente
- **Episodico:** se l'esperienza dell'agente è divisa in episodi indipendenti, cioè l'azione in un episodio non influisce sugli episodi successivi
- **Statico:** se l'ambiente non cambia mentre l'agente sta prendendo una decisione
- **Discreto:** se l'insieme di stati, azioni e percezioni è finito o numerabile
- **Singolo agente:** se l'agente opera da solo nell'ambiente senza la presenza di altri agenti

Esempio 2.5. Prendiamo ad esempio i seguenti ambienti provando a classificarli:

	Crossword	Robo-selector	Poker	Taxi
Osservabile	Sì	Parziale	Parziale	Parziale
Deterministico	Sì	No	No	No
Episodico	No	Sì	No	No
Statico	Sì	No	Sì	No
Discreto	Sì	No	Sì	No
Singolo agente	Sì	Sì	No	No

Il tipo di ambiente cambia radicalmente la soluzione del problema:

- **Deterministico, completamente osservabile:** Single-state problem
- **Completamente non osservabile:** Conformant problem, l'agente non sa in che stato si trova, ma potrebbe trovare una soluzione
- **Non deterministico e/o parzialmente osservabile:** Contingency problem, l'agente deve prevedere le possibili situazioni future e agire di conseguenza
- **Spazio degli stati sconosciuto:** Exploration problem, l'agente deve esplorare l'ambiente per scoprire gli stati e le azioni disponibili

2.4 Agenti di problem solving

È una forma ristretta di agenti che formulato un problema e un obiettivo partendo da uno stato cerca una soluzione ignorando le percezioni, siccome ci si trova in un single-state problem. Questo si chiama Offline problem solving perchè l'agente ha completa conoscenza dell'ambiente. Online problem solving è quando l'agente non ha completa conoscenza dell'ambiente.

Esempio 2.6. Il seguente è un esempio di problem solving agent:

```
1 function Simple-Problem-Solving-Agent(percept) returns action
2   static: seq, an action sequence, initially empty
3           state, some description of the current world state
4           goal, a goal, initially null
5           problem, a problem formulation
6
7   state <- Update-State(state, percept)
8
9   if seq is empty then
10    goal <- Formulate-Goal(state)
11    problem <- Formulate-Problem(state, goal)
12    seq <- Search( problem)
13
14   action <- First(seq)
15   seq <- Rest(seq)
16   return action
17
```

Esempio 2.7. Consideriamo il problema "Vacanze in Romania". Bisogna formulare un viaggio da Arad a Bucarest sapendo che l'aereo parte domani.

- **Goal:** Arrivare a Bucarest
- **Formulazione del problema:**
 - Stati: città della Romania
 - Azioni: volare tra le città
- **Soluzione:** Sequenza di città

Si potrebbe usare una mappa per trovare il percorso più breve (visione completa del mondo) e trovare una soluzione ottimale. Questo problema è definito da 4 componenti:

- **Stato iniziale:** ad esempio "ad Arad"
- **Funzione di transizione:** insieme di coppie (stato, azione) che mappano uno stato in un altro, ad esempio:

$$S(A) = \{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots\}$$

- **Test dell'obiettivo:** una funzione che verifica se lo stato corrente soddisfa l'obiettivo, ad esempio:

$$\text{Goal-Test}(s) = \begin{cases} \text{true} & \text{se } s = \text{Bucarest} \\ \text{false} & \text{altrimenti} \end{cases}$$

- **Path cost:** è una funzione che assegna un costo (additivo) a ogni azione, ad esempio la somma di distanze o il numero di azioni:

$$c(x, a, y) \geq 0$$

- **Soluzione:** Una sequenza di azioni che portano dallo stato iniziale allo stato obiettivo.

3 Ricerca nello spazio degli stati

3.1 Ricerca generale

3.1.1 Tree search

Un algoritmo di ricerca ad albero esplora lo spazio degli stati partendo dallo stato iniziale e generando nuovi stati (successori) applicando le azioni disponibili, cioè **espandendo** gli stati:

```

1 function Tree-Search(problem, strategy) function Tree-Search(
  problem, strategy) returns a solution, or failure
2 initialize the search tree using the initial state of problem
3 loop do
4   if no candidates for expansion then return failure
5   choose a leaf node for expansion according to strategy
6   if node contains a goal state then return the solution
7   else add successor nodes to the search tree (expansion)
8   endreturns a solution, or failure
9   initialize the search tree using the initial state of problem
10  loop do
11    if no candidates for expansion then return failure
12    choose a leaf node for expansion according to strategy
13    if node contains a goal state then return the solution
14    else add successor nodes to the search tree (expansion)
15  end

```

3.1.2 Stato e nodo

Stato e nodo non sono la stessa cosa, infatti:

- **Stato:** rappresenta una configurazione dell'ambiente
- **Nodo:** è una struttura dati che costituisce una parte dell'albero di ricerca e include informazioni aggiuntive come il genitore, l'azione che ha portato a quello stato, il costo del percorso o la profondità nell'albero, ecc...

3.1.3 Tree search generale

Espandere un nodo significa generare i suoi figli, cioè i nodi successori e tutti i nodi non esplorati sono chiamati **frontiera**.

```

1 function Tree-Search( problem, frontier) returns a solution, or
  failure
2   frontier <- Insert(Make-Node(problem.Initial-State))
3   while not IsEmpty(frontier) do
4     node <- Pop(frontier)
5     if problem.Goal-Test(node.State) then return node

```

```

6     frontier <- InsertAll(Expand(node, problem))
7   end loop
8   return failure

```

La strategia è quella di scegliere l'ordine in cui i nodi vengono espansi, cioè come viene gestita la frontiera. Le strategie sono valutate in base a:

- **Completezza:** se garantisce di trovare una soluzione quando esiste
- **Complessità di tempo:** numero di nodi generati o espansi
- **Complessità di spazio:** numero massimo di nodi memorizzati in memoria
- **Ottimalità:** se garantisce di trovare la soluzione migliore

Le complessità di spazio e di tempo sono misurate in termini di:

- *b*: maximum branching factor, numero massimo di figli per nodo
- *d*: profondità della soluzione meno costosa
- *m*: profondità massima dell'albero di ricerca (potrebbe essere infinita)

3.1.4 Stati ripetuti

Fallire nel riconoscere stati ripetuti può trasformare un problema lineare in un problema esponenziale. Bisogna quindi mantenere una lista di stati già visitati e non espandere nodi che portano a stati già visitati:

```

1 function Graph-Search( problem, frontier) returns a solution, or
   failure
2   explored <- an empty set
3   frontier <- Insert(Make-Node(problem.Initial-State))
4   while not IsEmpty(frontier) do
5     node <- Pop(frontier)
6     if problem.Goal-Test(node.State) then return node
7     if node.State is not in explored then
8       add node.State to explored
9       frontier <- InsertAll(Expand(node, problem))
10    end if
11  end loop
12  return failure

```

3.2 Ricerca non informata

Gli algoritmi di ricerca non informata utilizzano soltanto i dati disponibili nella definizione del problema e i principali sono:

- Breadth-first search
- Uniform-cost search (Dijkstra)
- Depth-first search
- Depth-limited search
- Iterative deepening search

3.2.1 Breadth-first search

Questo algoritmo espande il nodo non esplorato più superficiale, cioè il nodo più vicino alla radice. Utilizza una coda FIFO per la frontiera e i nuovi successori vengono aggiunti alla fine della coda.

```
1 function BFS( problem) returns a solution, or failure
2   node <- node with State=problem.Initial-State, Path-Cost=0
3   if problem.Goal-Test(node.State) then return node
4   explored <- empty set frontier <- FIFO queue with node as the
      only element
5   loop do
6     if frontier is empty then return failure
7     node <- Pop(frontier)
8     add node.State to explored
9     for each action in problem.Actions(node.State) do
10      child <- Child-Node(problem,node,action)
11      if child.State is not in (explored or frontier) then
12        if problem.Goal-Test(child.State) then return child
13        frontier <- Insert(child)
14      end if
15    end for
16  end loop
```

Questo tipo di ricerca è:

- **Completa:** Sì, soltanto se b è finito, cioè se il branching factor è limitato
- **Complessità di tempo:** $b + b^2 + b^3 + \dots + b^d = O(b^d)$
- **Complessità di spazio:** $O(b^d)$, perchè bisogna memorizzare tutti i nodi generati
- **Ottimale:** Sì, soltanto se il costo delle azioni è uniforme

3.2.2 Uniform-cost search

Questo algoritmo espande il nodo non esplorato con il **costo del percorso più basso**. La frontiera è una coda di priorità ordinata in base al costo del percorso. Questo tipo di ricerca è:

- **Completa:** Sì, se il costo minimo delle azioni $\geq \epsilon$ (con piccola ma $\epsilon > 0$)
- **Complessità di tempo:** Numero di nodi $g \leq$ del costo del percorso ottimale C^* . $O(b^{1+\lceil C^*/\epsilon \rceil})$
- **Complessità di spazio:** $O(b^{1+\lceil C^*/\epsilon \rceil})$
- **Ottimale:** Sì perchè i nodi vengono espansi in ordine di costo del percorso

Ci sono due modifiche principali rispetto alla BFS che garantiscono l'ottimalità:

1. Il goal test viene fatto quando il nodo viene estratto dalla frontiera, non quando viene generato. (Questo elemento spiega il +1 nella complessità)
2. Controllare se un nodo generato è già presente nella frontiera con un costo più alto e in tal caso sostituirlo con il nuovo nodo a costo più basso

3.2.3 Depth-first search

Questo algoritmo espande il nodo non esplorato più profondo, cioè il nodo più lontano dalla radice. Utilizza una pila LIFO per la frontiera e i nuovi successori vengono aggiunti all'inizio. Questo tipo di ricerca è:

- **Completa:** No, perchè può rimanere bloccata in un ramo infinito, a meno che l'albero di ricerca non abbia una profondità limitata. Si potrebbero evitare loop modificando l'algoritmo per evitare stati ripetuti sul percorso corrente
- **Complessità di tempo:** $O(b^m)$, dove m è la profondità massima dell'albero di ricerca
- **Complessità di spazio:** $O(bm)$, bisogna memorizzare soltanto il percorso corrente e i nodi fratelli
- **Ottimale:** No, perchè non garantisce di trovare la soluzione migliore

3.2.4 Iterative deepening search

Questo algoritmo combina i vantaggi della BFS e della DFS. Esegue una serie di ricerche in profondità limitata, aumentando progressivamente il limite di profondità fino a trovare una soluzione.

```
1 # Depth-Limited Search
2 function DLS(problem, limit) returns soln/fail/cutoff
3   R-DLS(Make-Node(problem.Initial-State), problem, limit)
4
5
6 function R-DLS(node, problem, limit) returns soln/fail/cutoff
7   if problem.Goal-Test(node.State) then return node
8   else if limit = 0 then return cutoff # raggiunta la profondità
9     massima
10  else
11    # flag: c'è stato un cutoff in uno dei sottoalberi?
12    cutoff-occurred? <- false
13    for each action in problem.Actions(node.State) do
14      child <- Child-Node(problem, node, action)
15      result <- R-DLS(child, problem, limit-1)
16      if result = cutoff then cutoff-occurred? <- true
17      else if result = failure then return result
18    end for
19    if cutoff-occurred? then return cutoff else return failure
20  end else
21
22 # Iterative Deepening Search
23 function IDS(problem) returns a solution
24   inputs: problem, a problem
25   for depth <- 0 to infinity do
26     result <- DLS(problem, depth)
27     if result = cutoff then return result
28   end
```

Questo tipo di ricerca è:

- **Completa:** Sì
- **Complessità di tempo:** $db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- **Complessità di spazio:** $O(bd)$

- **Ottimale:** Sì, se il costo delle azioni è uniforme

Esercizio 3.1. Assumi:

1. Un albero di ricerca ben bilanciato, tutti i nodi hanno lo stesso numero di figli
2. Il goal state è l'ultimo che viene espanso nel suo livello (il più a destra)
3. Se il branching factor è 3, la soluzione più superficiale è a profondità 3 (la radice è a profondità 0) e si utilizza la ricerca in ampiezza quanti nodi vengono generati?
4. Se il branching factor è 3, la soluzione più superficiale è a profondità 3 (la radice è a profondità 0) e si utilizza la iterative deepening quanti nodi vengono generati?

Esercizio 3.2. Un uomo ha un lupo, una pecora e un cavolo. L'uomo è sulla riva di un fiume con una barca che può trasportare solo lui e un altro oggetto. Il lupo mangia la pecora e la pecora mangia il cavolo, quindi non può lasciarli insieme da soli.

1. Formalizza il problema come un problema di ricerca
2. Usa BFS per risolvere il problema

Soluzione:

Formalizziamo gli stati come una tupla:

$$\langle W, S, C, M, B \rangle$$

dove:

- W: posizione del lupo
- S: posizione della pecora
- C: posizione del cavolo
- M: posizione dell'uomo
- B: stato della barca

La posizione può essere 0 (left) o 1 (right).

Lo stato iniziale è:

$$\langle 0, 0, 0, 0, 0 \rangle$$

Lo stato obiettivo è:

$$\langle 1, 1, 1, 1, 1 \rangle$$

Le azioni possibili sono:

- Porta il lupo (CW)
- Porta la pecora (CS)
- Porta il cavolo (CC)
- Porta niente (CN)

Operatore	Precondizione	Funzione
CW	$M = B, M = W, S \neq C$	$\langle W, S, C, M, B \rangle \mapsto \langle \bar{W}, S, C, \bar{M}, \bar{B} \rangle$
CS	$M = B, M = S$	$\langle W, S, C, M, B \rangle \mapsto \langle W, \bar{S}, C, \bar{M}, \bar{B} \rangle$
CC	$M = B, M = C, W \neq S$	$\langle W, S, C, M, B \rangle \mapsto \langle W, S, \bar{C}, \bar{M}, \bar{B} \rangle$
CN	$M = B$	$\langle W, S, C, M, B \rangle \mapsto \langle W, S, C, \bar{M}, \bar{B} \rangle$

Notiamo che in tutte le precondizioni c'è $M = B$ perchè l'uomo deve essere sempre con la barca, quindi si possono unire i due stati in uno solo M .

3.3 Ricerca informata

Gli algoritmi di ricerca informata utilizzano informazioni aggiuntive (euristiche) per guidare la ricerca verso la soluzione in modo più efficiente.

3.3.1 Best-first search

Questo algoritmo usa una **funzione di valutazione** per ogni nodo che stima la "desiderabilità". La frontiera è una coda ordinata in ordine decrescente di desiderabilità. A seconda di come viene definita la desiderabilità si ottengono diversi algoritmi:

- Greedy best-first search
- A*

3.3.2 Greedy best-first search

Questo algoritmo espande il nodo che sembra essere il più vicino alla soluzione secondo una funzione di valutazione euristica $h(n)$ che stima il costo rimanente per raggiungere l'obiettivo da un nodo n .

Esempio 3.1. In una mappa di una città, la funzione di valutazione potrebbe essere la distanza in linea d'aria dal nodo corrente alla destinazione. In questo modo, l'algoritmo esplora prima i nodi che sembrano più vicini alla destinazione, riducendo il numero di nodi esplorati rispetto a una ricerca non informata.

Questo tipo di ricerca è:

- **Completa:** No, perchè può rimanere bloccata in un ciclo infinito. È completo se lo spazio di ricerca è finito e ci sono controlli per evitare stati ripetuti
- **Complessità di tempo:** $O(b^m)$ nel peggiore dei casi, ma può essere molto più veloce con una buona euristica

- **Complessità di spazio:** $O(b^m)$, bisogna memorizzare tutti i nodi generati
- **Ottimale:** No

3.3.3 A* search

Questo algoritmo evita di espandere cammini che sono già molto costosi e ha come funzione di valutazione:

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

dove:

- $g(n)$: costo del percorso dal nodo iniziale a n
- $h(n)$: stima del costo rimanente per raggiungere l'obiettivo da n
- $f(n)$: stima del costo totale del percorso passando per n

L'euristica, per poter garantire l'ottimalità, deve essere **ammissibile**, cioè per ogni nodo la stima di quel nodo deve essere minore o uguale del vero costo per arrivare all'obiettivo, quindi non deve **sovrastimare** il costo rimanente:

$$h(n) \leq h^*(n) \quad h(n) \geq 0 \rightarrow h(G) = 0$$

dove $h^*(n)$ è il costo effettivo del percorso da n .

Teorema 3.1. Per A* l'euristica ammissibile implica l'ottimalità

Questo tipo di ricerca è:

- **Completa:** Sì, tranne se ci sono infiniti nodi con $f \leq f(G)$
- **Complessità di tempo:** Esponenziale in errore relativo in $h \times$ lunghezza del numero di passi della soluzione ottimale. (Se l'euristica è buona, la complessità sarà molto più bassa)
- **Complessità di spazio:** $O(b^d)$, bisogna memorizzare tutti i nodi generati
- **Ottimale:** Sì, ma richiede assunzioni sull'euristica (ammissibilità, consistenza) e una strategia di ricerca (ricerca ad albero o grafo)

3.3.4 Consistenza e ammissibilità

Definizione 3.1. Un euristica è **consistente** se:

$$h(n) \leq c(n, a, n') + h(n')$$

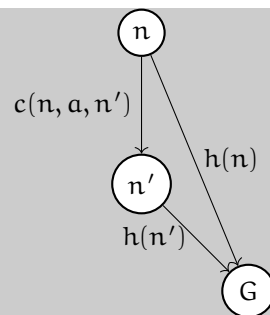


Figura 1: Esempio di euristica consistente

- Si può dimostrare che se h è consistente, allora $f(n)$ non decresce lungo qualsiasi cammino
- A* espande i nodi in ordine crescente di f , quindi trova sempre la soluzione ottimale

Quindi si espande sempre prima un cammino ottimo rispetto a un cammino non ottimo.

La consistenza implica l'ammissibilità e può essere dimostrato per induzione sul cammino verso il goal. L'ammissibilità però non implica la consistenza.

Consistenza \rightarrow Ammissibilità

Ammissibilità \nrightarrow Consistenza

- Tree-Search + euristica ammissibile \rightarrow A* ottimale
- Graph-Search + euristica ammissibile \nrightarrow A* ottimale (può scartare il cammino ottimale per un nodo ripetuto)
- Graph-Search + euristica consistente \rightarrow A* ottimale

3.3.5 Euristiche

Le euristiche possono essere create in diversi modi, prendiamo ad esempio l'8-puzzle:

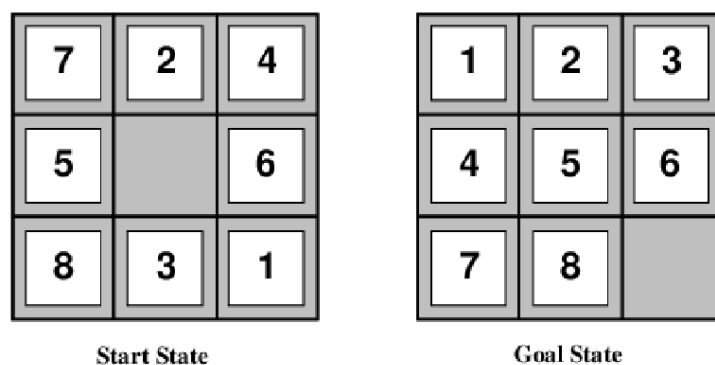


Figura 2: Esempio di 8-puzzle

Per questo problema si potrebbe utilizzare come euristica:

- $h_1(n)$ = numero di pezzi fuori posto
- $h_2(n)$ = somma delle distanze di Manhattan (numero di mosse orizzontali e verticali necessarie per portare ogni pezzo alla sua posizione obiettivo)

Entrambe le euristiche sono ammissibili, ma h_2 è più precisa di h_1 perchè fornisce una stima più vicina al costo reale per raggiungere l'obiettivo.

In questo caso si dice che h_2 **domina** h_1 se sono entrambe ammissibili e $h_2(n)$ è sempre maggiore o uguale a h_1 :

$$h_2(n) \geq h_1(n) \quad \forall n$$

Teorema 3.2. Date due qualsiasi euristiche **ammissibili** h_a e h_b , allora l'euristica definita come:

$$h(n) = \max(h_a(n), h_b(n))$$

è anch'essa ammissibile e domina sia h_a che h_b

Le euristiche ammissibili possono essere derivate dall'esatto costo della soluzione di un problema **rilassato**, cioè un problema simile a quello originale ma con restrizioni rimosse. Ad esempio, per l'8-puzzle si potrebbe rilassare il problema permettendo di muovere una casella ovunque (in questo caso $h_1(n)$ da la soluzione migliore), oppure permettendo di muovere una casella in qualsiasi casella adiacente (in questo caso $h_2(n)$ da la soluzione migliore)

Definizione utile 3.1. Il costo della soluzione ottimale di un problema rilassato non è maggiore del costo della soluzione ottimale del problema reale.

3.4 Ricerca locale

La ricerca locale è una tecnica di ricerca che si concentra su una soluzione cercando di migliorarla iterativamente. Gli algoritmi più comuni sono:

- Hill climbing
- Simulated annealing
- Algoritmi genetici

Questo tipo di ricerca è utile quando il percorso per arrivare alla soluzione non è importante, ma solo la soluzione finale. Ci sono due approcci principali:

- Trovare la configurazione ottimale (ad esempio (TSP - Traveling Salesman Problem))
- Trovare una configurazione che soddisfa dei vincoli (ad esempio il problema delle n regine)

Si possono anche usare algoritmi di **iterative improvement** che partono da una configurazione iniziale e cercano di migliorarla iterativamente fino a raggiungere un punto di ottimo locale.

Esempio 3.2. Un esempio di ricerca locale è il Traveling Salesman Problem (TSP), dove l'obiettivo è trovare il percorso più breve che visita un insieme di città esattamente una volta e ritorna alla città di partenza.

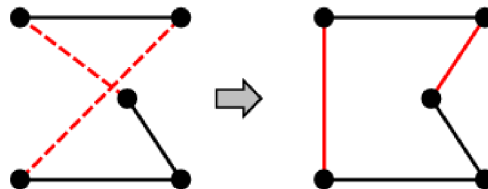


Figura 3: Esempio di TSP

Varianti di questo approccio arrivano fino a 1% della soluzione ottimale in tempi ragionevoli per migliaia di città.

Esempio 3.3. Un altro esempio è il problema delle n regine, dove l'obiettivo è posizionare n regine su una scacchiera $n \times n$ in modo che nessuna regina minacci un'altra.

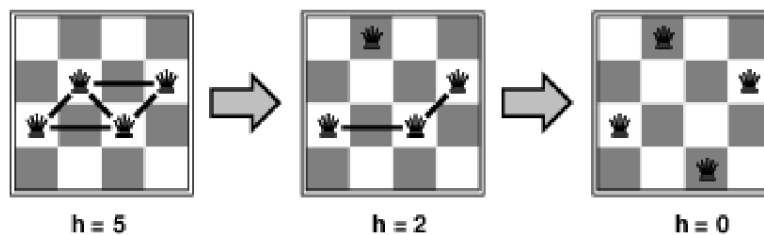


Figura 4: Esempio di n-regine

Risolve quasi sempre il problema quasi istantaneamente per n molto grande, ad esempio $n = 1.000.000$.

3.4.1 Hill climbing

Hill climbing è un algoritmo di ricerca locale che parte da una soluzione iniziale e cerca di migliorarla iterativamente spostandosi verso la soluzione migliore nella sua vicinanza. L'algoritmo continua a muoversi finché non trova un punto dove nessuna delle soluzioni vicine è migliore della soluzione corrente. Ritorna un massimo locale, che potrebbe non essere il massimo globale.

```

1 function Hill-Climbing(problem) returns a state that is a local
  maximum
2   inputs: problem, a problem
3   local variables: current, a node
4                   neighbor, a node
5   current <- Make-Node(problem.Initial-State)

```

```

6 loop do
7   neighbor <- a highest-valued successor of current
8   if neighbour.Value <= current.Value then return
9     current.State
10  end if
11  current <- neighbor
12 end

```

La rappresentazione del problema assumendo di avere tutti gli stati sull'asse x e il valore della funzione obiettivo sull'asse y è chiamata **state space landscape**:

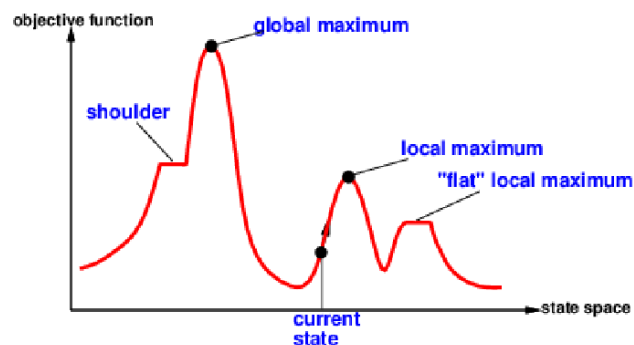


Figura 5: Esempio di state space landscape per hill climbing

- **Random-restart hill climbing**: esegue hill climbing più volte da stati iniziali casuali per aumentare le probabilità di trovare il massimo globale
- **Random sideways moves**: permette di fare mosse che non migliorano la soluzione corrente per evitare di rimanere bloccati in una sezione piatta

3.4.2 Simulated annealing

Simulated annealing è un algoritmo di ricerca locale ispirato al processo di raffreddamento dei metalli. L'algoritmo parte da una soluzione iniziale e cerca di migliorarla iterativamente, ma a differenza di hill climbing, permette di **accettare soluzioni peggiori** con una certa probabilità che **diminuisce nel tempo**. Questo aiuta a evitare di rimanere bloccati in massimi locali.

```

1 function Simulated-Annealing( problem, schedule) returns a solution
   state
2   inputs: problem, a problem
3           schedule, a mapping from time to 'temperature'
4   local variables: current, a node
5                   next, a node
6                   T, a 'temperature' controlling prob. of downward
   steps
7   current <- Make-Node(problem.Initial-State)
8   for t <- 1 to infinity do
9     T <- schedule(t) // temperature at time t
10    if T = 0 then return current
11    next <- a randomly selected successor of current
12    deltaE <- next.Value - current.Value
13    if deltaE > 0 then current <- next
14    else current <- next only with probability  $e^{-(\text{deltaE}/T)}$ 

```

Se la "temperatura" T diminuisce lentamente abbastanza, allora l'algoritmo converge **sempre** alla soluzione ottimale x^* . Questo perchè:

$$e^{\frac{E(x^*)}{kT}} / e^{\frac{E(x)}{kT}} = e^{\frac{E(x^*) - E(x)}{kT}} \gg 1 \quad \text{per } T \rightarrow 0$$

3.4.3 Local beam search

Local beam search è un algoritmo di ricerca locale che mantiene **un insieme di soluzioni candidate** e cerca di migliorarle iterativamente. In ogni iterazione, l'algoritmo **seleziona casualmente i successori di k , ma con un bias verso i migliori** per formare il nuovo insieme di soluzioni candidate.

3.4.4 Algoritmi genetici

Gli algoritmi genetici sono una classe di algoritmi di ricerca ispirati ai processi di evoluzione biologica. Questi algoritmi utilizzano meccanismi simili alla selezione naturale, alla mutazione e al crossover per evolvere una popolazione di soluzioni candidate verso soluzioni migliori. Gli algoritmi genetici hanno bisogno di stati encodati come stringhe di caratteri. Il crossover combina due stringhe per creare una nuova stringa e ha senso solo se le sottostringhe hanno un significato indipendente.

Esempio 3.4. Prendiamo ad esempio il problema delle n regine in cui si ha un encoding della scacchiera come un numero in cui la cifra in posizione i rappresenta la riga in cui si trova la regina nella colonna i .

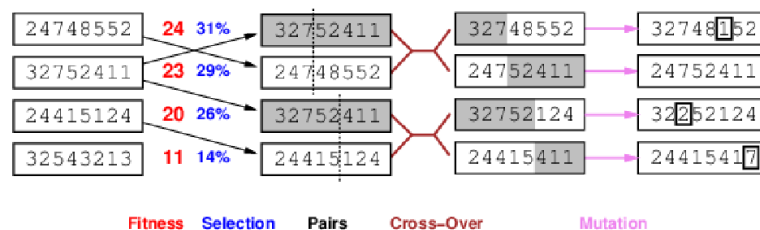


Figura 6: Esempio di algoritmo genetico per il problema delle n -regine

Questo genera una configurazione nuova a partire da coppie di configurazioni esistenti, selezionate in base alla loro "fitness" (un valore che misura quanto una configurazione si avvicina alla soluzione ottimale).

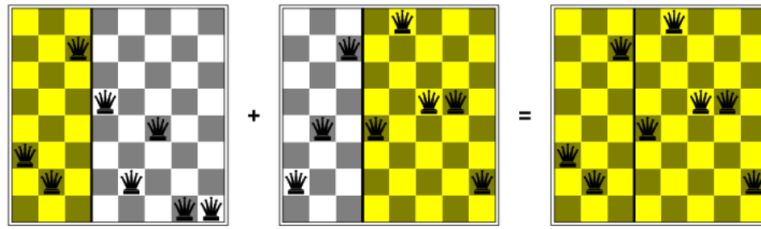


Figura 7: Rappresentazione della scacchiera

In questo caso la fitness è calcolata come il numero di regine che si minacciano a vicenda. Il caso peggiore è quello in cui tutte le regine si minacciano a vicenda, quindi la fitness è 28 (per $n = 8$) perchè:

$$\text{num_minacce} = \frac{n(n-1)}{2} = \frac{8 \cdot 7}{2} = 28$$

quindi:

$$\text{fitness} = 28 - \text{num_minacce}$$

3.5 Ricerca locale in uno spazio continuo

La ricerca locale può essere estesa a spazi di stato continui. Per risolvere questi problemi si possono utilizzare tecniche come:

- **Discretizzazione:** suddividere lo spazio continuo in una griglia di punti discreti con una risoluzione δ e utilizzare algoritmi di ricerca locale
- **Random perturbations:** si prende una soluzione e si applicano piccole perturbazioni casuali per esplorare lo spazio delle soluzioni
- **Gradient:** si calcola analiticamente il gradiente della funzione obiettivo $f(x)$

3.5.1 Gradient ascent/descent

Il gradiente di una funzione $f(x)$ è il seguente:

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right)$$

Per trovare la direzione di massima crescita della funzione obiettivo si pone il gradiente uguale a zero:

$$\nabla f(x) = 0$$

Spesso non si può porre il gradiente a 0 globalmente, ma si può migliorare localmente:

- Aggiornare la soluzione nella direzione massima del gradiente per ogni coordinata
- Più la funzione è "ripida" più si fanno passi grandi

Aggiornare una coordinata viene effettuato tramite una funzione generale $g(x_1, x_2)$:

$$x_1 \leftarrow x_1 + \alpha \frac{\partial g(x_1, x_2)}{\partial x_1} \quad x_2 \leftarrow x_2 + \alpha \frac{\partial g(x_1, x_2)}{\partial x_2}$$

Oppure in forma vettoriale:

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \nabla g(X) = \begin{bmatrix} \frac{\partial g(X)}{\partial x_1} \\ \frac{\partial g(X)}{\partial x_2} \end{bmatrix}$$

$$X \leftarrow X + \alpha \nabla g(X)$$

Dove α è lo "step size", cioè la dimensione del passo da fare:

- Se α è troppo grande si rischia di saltare soluzioni
- Se α è troppo piccolo i passi richiesti possono essere troppi

3.5.2 Algoritmo di Newton-Raphson

È una tecnica generale per trovare le radici di una funzione, cioè risolvere un'equazione del tipo $g(x) = 0$. Per farlo si trova un'approssimazione iniziale \bar{x}_0 della soluzione e iterativamente si aggiorna l'approssimazione usando la formula:

$$\bar{x}_{n+1} = \bar{x}_n - \frac{g(\bar{x}_n)}{g'(\bar{x}_n)}$$

dove g' è la derivata di g :

$$g'(x) = \frac{dg(x)}{dx}$$

Esempio 3.5. Consideriamo la funzione $g(x) = x^2 - a$.

1. Mostra che il metodo di Newton-Raphson porta alla formula:

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n} \right)$$

Soluzione:

La formula di Newton-Raphson è:

$$\begin{aligned} \bar{x}_{n+1} &= \bar{x}_n - \frac{g(\bar{x}_n)}{g'(\bar{x}_n)} \\ &= \bar{x}_n - \frac{\bar{x}_n^2 - a}{2\bar{x}_n} \\ &= \frac{2\bar{x}_n^2 - (\bar{x}_n^2 - a)}{2\bar{x}_n} \\ &= \frac{\bar{x}_n^2 + a}{2\bar{x}_n} \\ &= \frac{1}{2} \left(\bar{x}_n + \frac{a}{\bar{x}_n} \right) \end{aligned}$$

2. Fissato $\alpha = 4$ e $x_0 = 1$, calcola x_i , $i \in \{1, 2, 3\}$

Soluzione:

Sostituendo i valori nella formula ottenuta al passo precedente si ottiene:

$$x_1 = \frac{1}{2} \left(1 + \frac{4}{1} \right) = \frac{5}{2} = 2.5$$

$$x_2 = \frac{1}{2} \left(\frac{5}{2} + \frac{4}{\frac{5}{2}} \right) = \frac{1}{2} \left(\frac{25}{10} + \frac{8}{5} \right) = \frac{1}{2} \cdot \frac{41}{10} = \frac{41}{20} = 2.05$$

$$x_3 = \frac{1}{2} \left(\frac{41}{20} + \frac{4}{\frac{41}{20}} \right) = \frac{1}{2} \left(\frac{41}{20} + \frac{80}{41} \right) = \frac{1}{2} \cdot \frac{1681 + 1600}{820} \approx 2.0006$$

Graficamente si può vedere che la funzione converge rapidamente a 2:

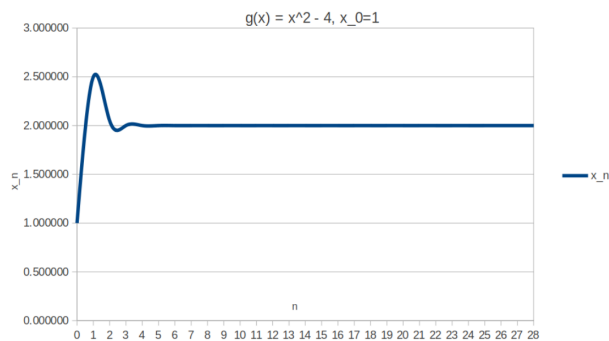


Figura 8: Esempio grafico del metodo di Newton-Raphson

3.5.3 Calcolo degli zeri del gradiente

Utilizzando il metodo di Newton-Raphson si possono trovare gli zeri del gradiente, cioè dove la funzione generica $g(x)$ è $\nabla f(X)$. In questo caso le funzioni di aggiornamento in forma vettoriale diventano:

$$x \leftarrow x - H_f^{-1}(x) \nabla f(x)$$

dove $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$ è la matrice Hessiana. Per problemi in più dimensioni calcolare tutte le entrate della matrice Hessiana può essere computazionalmente costoso, quindi spesso si usano metodi approssimati.

Questo è ancora un metodo **locale**, quindi soffre degli stessi problemi della ricerca locale, come massimi locali e punti sella. Random restart e simulated annealing possono essere utili anche per spazi continui.

3.5.4 Gradiente empirico

A volte si può calcolare $f(X)$ per un certo input, ma non si può calcolare $\nabla f(X) = 0$ neanche localmente. L'**empirical gradient** è la risposta di $f(X)$ a piccoli incrementi o decrementi di X .

3.6 Constrained satisfaction problem

Assumiamo di avere un singolo agente, azioni deterministiche e un ambiente completamente osservabile (discreto). In questo tipo di problemi:

- Lo stato è definito da un insieme di variabili $X = X_i$ che può assumere valori in un insieme di domini $D = D_i$.
- Il goal test è un insieme di vincoli che specificano le combinazioni ammissibili per sottoinsiemi di variabili
- Si possono usare algoritmi che sfruttano queste proprietà che sono più efficienti degli algoritmi di ricerca generici

Esempio 3.6. Un esempio di problema CSP è il problema del map coloring, in cui si deve colorare una mappa in modo che nessuna regione confinante abbia lo stesso colore.

- **Variabili:** WA, NT, Q, NSW, V, SA, T
- **Domini:** $D_i = \{\text{red, green, blue}\}$
- **Vincoli:** Regioni adiacenti devono avere colori diversi. Si rappresenta con: $WA \neq NT$



Figura 9: Esempio di map coloring

Una soluzione è ad esempio:

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

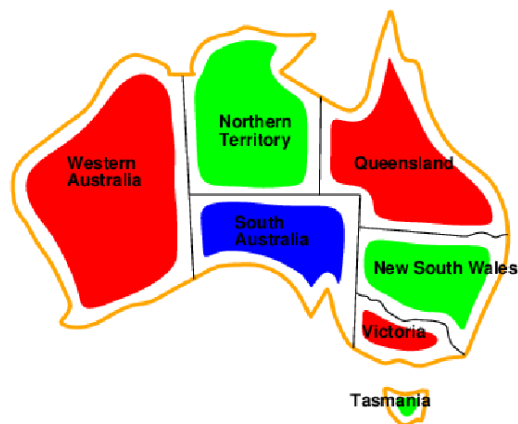


Figura 10: Esempio di soluzione del map coloring

3.6.1 Grafo dei vincoli

Il grafo dei vincoli, detto anche primal graph, è una rappresentazione grafica di un problema CSP in cui è presente un nodo per ogni variabile e un arco per ogni vincolo tra due variabili:

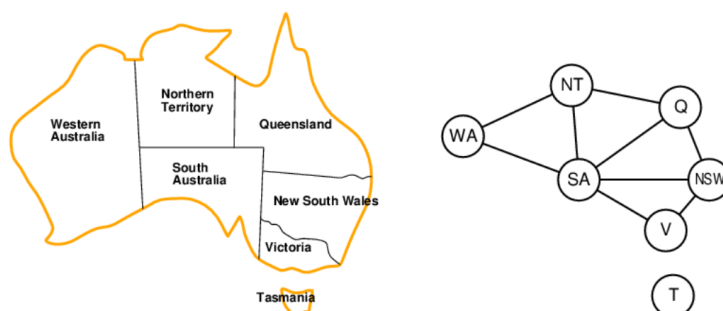


Figura 11: Esempio di grafo dei vincoli per il problema del map coloring

Definizione 3.2. Il grafo dei vincoli è definito come una tupla di 3 elementi:

$$CN = \langle X, D, C \rangle$$

dove:

- $X = \{x_1, \dots, x_n\}$: insieme di variabili
- $D = \{D_1, \dots, D_n\}$: insieme di domini
- $C = \{(S_1, R_1), \dots, (S_m, R_m)\}$: insieme di vincoli, dove ogni vincolo (S_i, R_i) è composto da:

- $S_i \subseteq X$: sottoinsieme di variabili coinvolte nel vincolo (scope)
- R_i : sottoinsieme del prodotto cartesiano delle variabili in S_i , cioè l'insieme delle combinazioni ammissibili delle variabili in S_i
- Soluzione: un'assegnazione di **tutte** le variabili che soddisfa **tutti** i vincoli. Esistono anche soluzioni parziali consistenti, cioè una soluzione parziale che soddisfa tutti i vincoli in cui lo scope contiene solo variabili assegnate. La soluzione parziale consistente non è necessariamente parte di una soluzione completa.
- Tasks: è una funzione di ottimizzazione, ad esempio controllo di consistenza, trovare una o tutte le soluzioni

Esempio 3.7. Consideriamo seguente crossword:

X1	X2	X3
	X4	
	X5	

Figura 12: Esempio di crossword

Bisogna assegnare le lettere delle parole disponibili alle caselle vuote in modo che le parole risultanti siano valide.

- **Variabili:** parole possibili: MAP, ARC
- **Domini:** $D_i =$ lettere dell'alfabeto
- **Vincoli:** lettere condivise devono essere uguali:

$$C_1 [\{x_1, x_2, x_3\}, (\text{MAP}), (\text{ARC})]$$

$$C_2 [\{x_2, x_4, x_5\}, (\text{MAP}), (\text{ARC})]$$

Il grafo dei vincoli è il seguente:

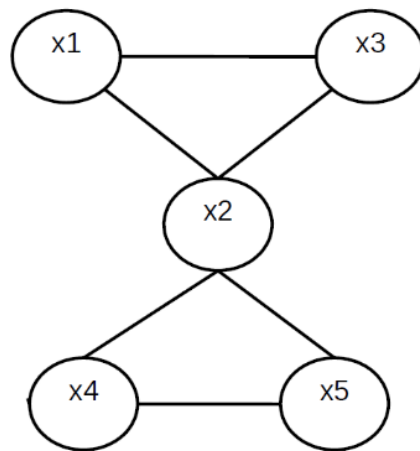


Figura 13: Esempio di grafo dei vincoli per il problema del crossword

Esiste un altro grafo chiamato **grafo duale** in cui i nodi rappresentano i vincoli e gli archi rappresentano le variabili condivise tra i vincoli (vincolo di uguaglianza):

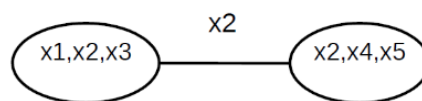


Figura 14: Esempio di grafo duale per il problema del crossword

3.6.2 Problemi combinatori

I problemi combinatori sono tutti quei problemi in cui dato un insieme di soluzioni si vuole trovare la soluzione migliore. Il CSP è un sottoinsieme di problemi combinatori. I principali tipi di problemi combinatori sono:

- **Decisioni:** dato un insieme di soluzioni, decidere se esiste una soluzione che soddisfa certi criteri. Ad esempio colora un grafo con k colori, bisogna dire se è possibile o no fissato un k .
- **Ottimizzazione:** bisogna ottimizzare un obiettivo. Ad esempio colorare un grafo con k colori minimizzando i conflitti.
- **Ottimizzazione multiobiettivo:** bisogna ottimizzare più obiettivi contemporaneamente. Ad esempio minimizzare il rischio e massimizzare il profitto in un portafoglio di investimenti.
- **Graphical models:** sono problemi definiti da:
 - Insieme di variabili
 - Domini delle variabili
 - Funzioni **locali** che definiscono i vincoli

- Funzione **globale** che rappresenta un'aggregazione delle funzioni locali
- Soluzioni, ovvero assegnazioni delle variabili, che ottimizzano la funzione globale

3.6.3 Backtracking search

Il backtracking search è un algoritmo di ricerca per i problemi CSP. Questo algoritmo è utile quando le assegnazioni delle variabili sono commutative, cioè l'ordine in cui le variabili vengono assegnate non cambia il risultato finale:

WA = red, NT = green è equivalente a NT = green, WA = red

Questo algoritmo è semplicemente una ricerca in profondità per CSP con assegnamenti alle variabili singoli. L'ordine delle variabili può impattare la performance dell'algoritmo.

```

1 function Backtracking-Search(csp) returns solution or failure
2   return Backtrack({ }, csp)
3
4 function Backtrack(assignment, csp) returns solution or failure
5   if assignment is complete then return assignment
6   var <- Select-Unassigned-Variable(csp)
7   for each value in Order-Domain-Values(var, assignment, csp) do
8     if value is consistent with assignment then
9       add {var = value} to assignment
10      inferences <- Inferences(csp, var, value)
11      if inferences ≠ failure then
12        add inferences to assignment
13        result <- Backtrack(assignment, csp)
14        if result ≠ failure then
15          return result
16        endif
17      endif
18    endif
19    remove {var = value} and inferences from assignment
20  endfor
21  return failure

```

Le principali decisioni che impattano l'algoritmo sono:

- Come selezionare la variabile
- Come selezionare il valore
- Come fare inferenze

Per migliorare l'algoritmo si possono usare le seguenti tecniche:

- **Ordinare le variabili:**
 - Minimum Remaining Values: Seleziona la variabile con il minor numero di valori legali rimasti nel dominio, quindi prima si fallisce meglio è

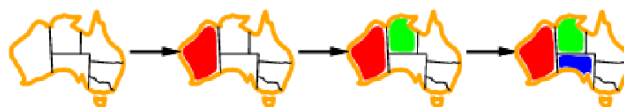


Figura 15: Esempio di Minimum Remaining Values

- Degree Heuristic: Seleziona la variabile che è coinvolta nel maggior numero di vincoli con altre variabili non assegnate

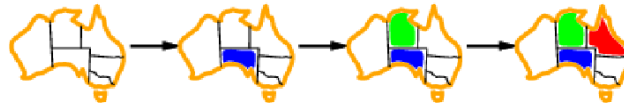


Figura 16: Esempio di Degree Heuristic

- **Ordinare i valori delle variabili:**

- Least Constraining Value: Seleziona il valore che lascia il maggior numero di opzioni aperte per le altre variabili non assegnate

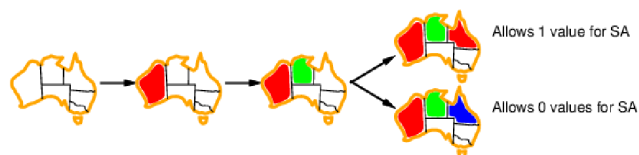


Figura 17: Esempio di Least Constraining Value

- **Consistenza locale:** Arc o Path consistency per ridurre i domini delle variabili e individuare fallimenti precoci
- **Look-ahead:** Predirre i conflitti futuri per evitare di esplorare rami che porteranno a fallimenti
- **Look-back:** Analizzare verso dove fare backtracking
- **Tree decomposition:** Sfrutta la struttura del problema per dividerlo in sotto-problemi più piccoli e risolverli separatamente

3.6.4 Inferenza

Si possono fare ragionamenti riguardo ai vincoli per fare inferenze per nuovi vincoli. Se consideriamo l'esempio del graph coloring:

- Dati $\{x_1, x_2, x_3\}$, $\{D_1 = D_2 = D_3\}$, $D_i = \{R, B\}$ e $C = \{C_1 : (x_1 \neq x_2), C_2 : (x_2 \neq x_3)\}$
- Si può inferire che $C_3 : (x_1 \neq x_3)$ perchè se $x_1 = x_3$ allora x_2 non può assumere nessun valore legale

I vantaggi e svantaggi dell'aggiunta di vincoli sono:

- Vincoli più stringenti portano ad uno spazio di ricerca più piccolo
- Aggiungere vincoli richiede più computazione
- Ogni volta che una nuova variabile viene assegnata bisogna controllare più vincoli

- Se il problema consiste soltanto in vincoli binari allora non si hanno mai più di $O(n)$ controlli
- Se il problema consiste in vincoli di ordine superiore r allora si hanno $O(n^{r-1})$ controlli

Questa inferenza rende il grafo backtrack free.

Un grafo si dice **backtrack free** se ogni foglia è un goal state. Una DFS su un grafo backtrack-free garantisce un assegnamento completo e consistente

3.6.5 Look Ahead

Look ahead è una tecnica che permette di fare inferenze sui vincoli futuri per evitare di esplorare rami che porteranno a fallimenti. Quindi data un'inferenza approssimata si vuole predire l'impatto della prossima assegnazione di una variabile e vedere come impatta i futuri assegnamenti. Ci sono due strategie principali:

- **Forward checking:** Controlla le variabili assegnate separatamente da quelle non assegnate.
- **Arc consistency look ahead:** Propaga la consistenza di arco, cioè quella che assicura che per ogni valore di una variabile esista un valore legale nella variabile connessa tramite un vincolo binario, in tutta la rete

3.6.6 Forward checking look ahead

È la forma più limitata di propagazione dei vincoli. Propaga l'effetto di un valore selezionato su tutte le variabili future **separatamente**, cioè una ad una. Se il dominio di una variabile futura diventa vuoto, allora si tenta il valore successivo per la variabile corrente.

Esempio 3.8. Prendiamo ad esempio il problema del map coloring, l'idea principale è quella di tenere traccia dei valori legali rimanenti per le variabili non assegnate. Viene terminata la ricerca quando qualsiasi variabile non ha più valori legali.

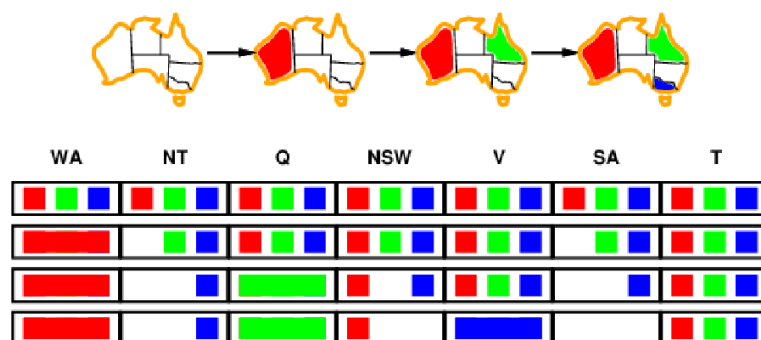


Figura 18: Esempio di forward checking per il problema del map coloring

La complessità del forward checking è: $O(ek^2)$, dove:

- e il numero di vincoli
- k valore per ogni variabile futura
- k valore per la variabile corrente

3.6.7 Arc consistency look ahead

L'arc consistency look ahead forza la consistenza di arco su tutte le variabili rimanenti.

- Un arco $x_k \rightarrow x_j$ è **consistente** se e solo se per ogni assegnamento di x_k c'è almeno un assegnamento di x_j che è consistente con il vincolo (x_k, x_j) .
- Forzare la consistenza di arco: se nessun valore di x_j è consistente con un dato valore di $x_k = c$ allora si rimuove c dal dominio di x_k
- Forward checking: si impone la consistenza da ogni variabile non assegnata a un nuovo assegnamento
- Nota: nel forward checking non si controlla mai la consistenza tra variabili non assegnate, nell'arc consistency look-ahead per ogni nuovo assegnamento si controlla la consistenza di arco tra ogni coppia di variabili

L'arc consistency viene rappresentata da una funzione che modifica il dominio di x_k se non è consistente con x_j :

$$\text{rev}(x_k, x_j)$$

Esempio 3.9. Consideriamo il problema del map coloring. I passi dell'algoritmo sono:

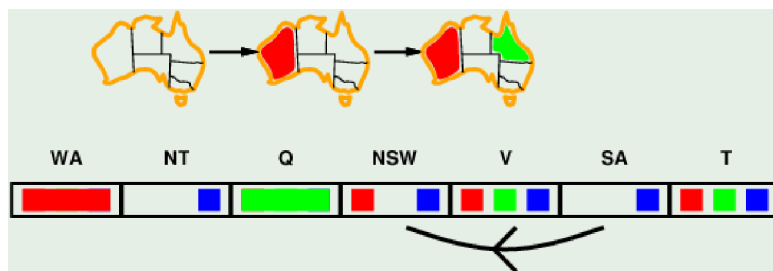


Figura 19: Passo 1

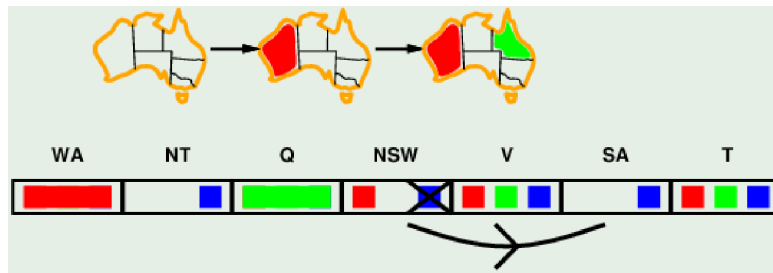


Figura 20: Passo 2

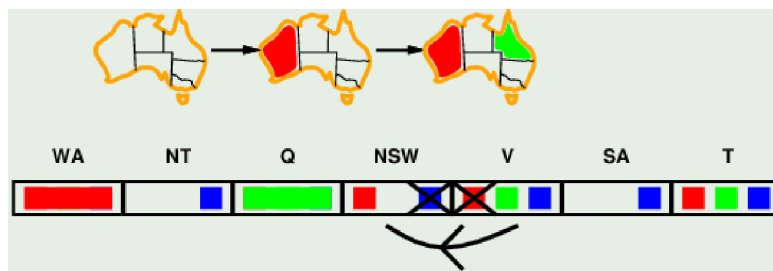


Figura 21: Passo 3

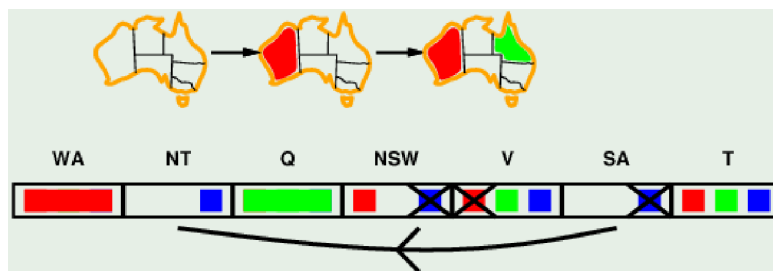


Figura 22: Passo 4

La complessità dell'algoritmo arc consistency migliore (AC-4) è $O(ek^3)$.

3.6.8 Forzare l'arc consistency

Definizione 3.3. • Un arco x_i, x_j è arc consistent se e solo se x_i è arc consistent rispetto a x_j e x_j è arc consistent rispetto a x_i .

- Un grafo è arc consistent se e solo se tutti i suoi vincoli sono arc consistent.
- Si può facilmente assicurare che una coppia di variabili siano arc consistent (procedura Revise), Ma rivedere la consistenza di arco su una

variabile potrebbe rendere un'altra variabile non arc consistent.

- Sono necessarie procedure sistematiche per garantire l'arc consistency sulle reti.

- L'arc consistency con un dominio vuoto implica che il problema non ha soluzione.
- L'arc consistency con tutti i domini non vuoti **non** implica che ci sia una soluzione. Quindi l'arc consistency non è completa.
- L'unica cosa che si può dire a riguardo è che se l'arc consistency con tutti i domini non vuoti non ha cicli nel grafo dei vincoli e ha solo vincoli binari, allora il grafo è backtrack free ed esiste una soluzione.

3.6.9 Tree decomposition

Consiste nel decomporre il problema in una struttura ad albero che permette di sfruttare la proprietà degli alberi di essere backtrack free. L'idea più semplice è quella di togliere le variabili assegnate fino a che il grafo non diventa un albero.

Definizione 3.4. Dato un grafo non orientato, il sottoinsieme di nodi del grafo è definito **cycle cutset** se la rimozione di questi nodi rende il grafo aciclico.

Per trovare una soluzione al problema si devono provare tutte le possibili assegnazioni delle variabili nel cycle cutset e per ogni assegnazione si risolve il problema tramite arc propagation. La complessità è esponenziale, ma dipende solo dal numero di nodi nel cycle cutset.

4 Logical Agents

Un agente logico è un agente che utilizza la logica per rappresentare la conoscenza e ragionare su di essa. La logica fornisce un linguaggio formale per esprimere fatti e regole sul mondo, permettendo all'agente di dedurre nuove informazioni e prendere decisioni basate sulla conoscenza acquisita.

4.1 Knowledge based agents

Gli agenti knowledge based sono divisi in due componenti principali:

- **Inference engine:** si occupa di dedurre nuove informazioni dalla knowledge base utilizzando regole logiche
- **Knowledge base:** contiene la conoscenza dell'agente rappresentata in forma logica, cioè è un insieme di frasi logiche appartenenti ad un linguaggio formale

L'approccio dichiarativo per costruire un agente consiste nel **dirgli** (Tell) cosa deve sapere e poi l'agente può **chiedere** (Ask) alla sua knowledge base per sapere cosa fare.

Un esempio di agente knowledge based è il seguente:

```

1 function KB-Agent( percept) returns an action
2   static: KB, a knowledge base
3         t, a counter, initially 0, indicating time
4   Tell(KB, Make-Percept-Sentence( percept, t))
5   action <- Ask(KB, Make-Action-Query(t))
6   Tell(KB, Make-Action-Sentence(action, t))
7   t <- t + 1
8   return action

```

L'agente deve essere in grado di:

- Rappresentare stati, azioni ecc...
- Incorporare nuove percezioni nella knowledge base
- Aggiornare le rappresentazioni interne del mondo
- **Dedurre** proprietà nascoste del mondo
- **Dedurre** azioni appropriate

Esempio 4.1. Consideriamo il problema del Wumpus World:

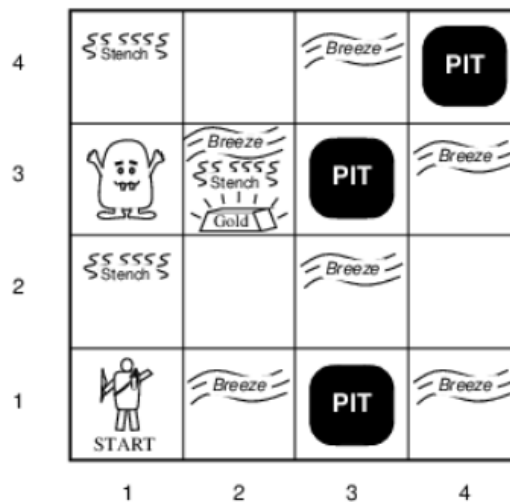


Figura 23: Esempio di Wumpus World

- **Performance measure:**
 - +1000 per uscire con l'oro
 - -1 per ogni azione
 - -1000 per morire
 - -10 per ogni freccia usata
- **Environment:**
 - 4x4 griglia

- Wumpus in una casella
- Pozzi in alcune caselle
- Oro in una casella
- Caselle adiacenti al Wumpus hanno puzza
- Caselle adiacenti ai pozzi hanno brezza
- Glitter nella casella con l'oro
- Prendi per prendere l'oro dalla casella
- Rilascia per lasciare l'oro nella casella
- Sparare uccide il Wumpus se è nella stessa direzione della freccia
- Sparare usa l'unica freccia disponibile

- **Actuators:**

- Gira a sinistra
- Gira a destra
- Avanti
- Prendi
- Rilascia

- **Sensors:**

- Puzza
- Brezza
- Glitter

Questo problema è:

- **Non osservabile** perchè si ha solo percezione locale
- **Deterministico** perchè le azioni hanno effetti certi
- **Non episodico** perchè le azioni sono sequenziali
- **Statico** perchè l'ambiente non cambia
- **Discreto** perchè ci sono un numero finito di stati e azioni
- **Singolo agente** perchè c'è solo un agente che agisce nell'ambiente (il wumpus è parte dell'ambiente)

4.2 Logica in generale

La logica è un sistema formale per rappresentare informazioni e ragionare su di esse.

- **Sintassi:** definisce le regole per costruire frasi valide nel linguaggio
- **Semantica:** definisce il significato delle frasi nel linguaggio, ad esempio definisce quando una frase è vera o falsa in un certo mondo

4.2.1 Entailment (Derivazione logica)

La derivazione o entailment indica che da una certa cosa ne segue un'altra:

$$KB \models \alpha$$

La knowledge base implica una frase α se e solo se α è vera in tutti i mondi in cui è vera la knowledge base. La derivazione è una relazione tra frasi logiche (sintassi) che si basa sulla semantica. Un modello è un mondo formalmente strutturato rispetto a quale verità o falsità di frasi logiche può essere valutata. m è un modello di una frase α , se α è vera in m , allora $M(\alpha)$ è l'insieme di tutti i modelli di α . Di conseguenza

$$KB \models \alpha \iff M(KB) \subseteq M(\alpha)$$

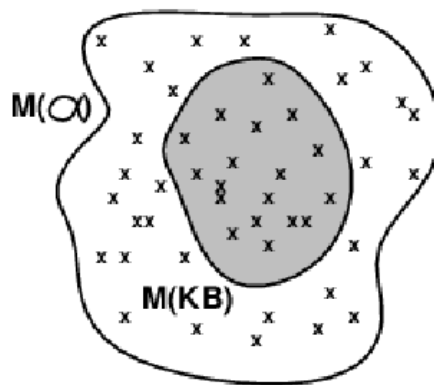


Figura 24: Esempio di entailment

Esempio 4.2. Un esempio potrebbe essere:

- $KB =$ Juventus ha vinto e la Roma ha vinto
- $\alpha =$ Juventus ha vinto

4.2.2 Inferenza

L'inferenza è il processo di derivare nuove frasi logiche da frasi esistenti nella knowledge base utilizzando regole logiche. Una frase α può essere derivata dalla knowledge base KB con una procedura i :

$$KB \vdash_i \alpha$$

Un sistema di inferenza ha due proprietà importanti:

- **Soundness:** se $KB \vdash_i \alpha$ allora $KB \models \alpha$ cioè tutto ciò che viene derivato è vero
- **Completeness:** se $KB \models \alpha$ allora $KB \vdash_i \alpha$ cioè tutto ciò che è vero può essere derivato

Esempio 4.3. Rappresentiamo il problema del Wumpus World in logica proposizionale:

- $P_{i,j}$: è vero se c'è un pozzo nella casella (i,j)
- $B_{i,j}$: è vero se c'è brezza nella casella (i,j)

Consideriamo i seguenti fatti:

$$R_1 : \neg P_{1,1}$$

$$R_2 : \neg B_{1,1}$$

$$R_3 : B_{2,1}$$

Rappresentiamo in forma logica la frase "i pozzi causano brezza nelle caselle adiacenti":

- Considerando caselle specifiche:

$$R_4 : B_{1,1} \iff (P_{1,2} \vee P_{2,1})$$

$$R_5 : B_{2,1} \iff (P_{1,1} \vee P_{2,2} \vee P_{3,1})$$

- R rappresenta le regole del mondo
- P rappresenta le variabili da assegnare
- KB è la knowledge base, cioè l'and logico di tutte le regole

4.2.3 Inferenza mediante enumerazione

Esiste un algoritmo che verifica per ogni possibile assegnamento se $KB \models \alpha$:

```

1 function TT-Entails?(KB, alpha) returns true or false
2   inputs: KB, the knowledge base, a sentence in prop. logic
3           alpha, the query, a sentence in prop. logic
4   symbols <- a list of the proposition symbols in KB and alpha
5   return TT-Check-All(KB, alpha, symbols, [ ])
6
7 function TT-Check-All(KB, alpha, symbols, model) returns true or
8   false
9   if Empty?(symbols) then
10     if PL-True?(KB, model) then return PL-True?(alpha, model)
11     else return true
12   else do
13     P <- First(symbols); rest <- Rest(symbols)
14     return TT-Check-All(KB, alpha, rest, Extend(P, true, model))
15           and
16           TT-Check-All(KB, alpha, rest, Extend(P, false, model))

```

Siccome si devono provare tutte le possibili combinazioni di verità per n variabili, la complessità è $O(2^n)$, quindi il problema è co-NP-completo.

4.2.4 Metodi di dimostrazione

I metodi di dimostrazione si dividono in due categorie principali:

- **Model checking:**

- Truth table enumeration: verifica tutti i modelli possibili
- Improved backtracking: usa backtracking per evitare di esplorare modelli non validi. Ad esempio DPLL (Davis-Putnam-Logemann-Loveland)
- Heuristic search in model space: cerca modelli validi usando euristiche per guidare la ricerca (corretto ma non completo). Ad esempio algoritmo come hill-climbing

- **Applicazione delle regole di inferenza**

- Generazione di nuove frasi logiche a partire da quelle esistenti
- Dimostrazione: Una sequenza di applicazioni di regole di inferenza
- Richiedono la traduzione di frasi logiche in **forma normale**

4.2.5 Equivalenza logica

Due frasi logiche sono logicamente equivalenti se e solo se sono vere negli stessi modelli:

$$\alpha \equiv \beta \iff \alpha \models \beta \wedge \beta \models \alpha$$

Esempio 4.4. Delle tautologie importanti sono:

$$\begin{aligned}
 (\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) \\
 (\alpha \vee \beta) &\equiv (\beta \vee \alpha) \\
 ((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) \\
 ((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) \\
 \neg(\neg\alpha) &\equiv \alpha \\
 (\alpha \implies \beta) &\equiv (\neg\alpha \implies \neg\beta) \\
 (\alpha \implies \beta) &\equiv (\neg\alpha \vee \beta) \\
 (\alpha \iff \beta) &\equiv ((\alpha \implies \beta) \wedge (\beta \implies \alpha)) \\
 \neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) \\
 \neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) \\
 (\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \\
 (\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))
 \end{aligned} \tag{1}$$

4.2.6 Validità e soddisfacibilità

Una frase logica è **valida** se è vera in tutti i modelli, ad esempio:

$$\text{True}, \quad \alpha \vee \neg\alpha, \quad \alpha \implies \alpha$$

La validità è collegata alla derivazione dal teorema della deduzione:

$$KB \models \alpha \iff (KB \implies \alpha) \text{ è valida}$$

Una frase logica è **soddisfacibile** se è vera in almeno un modello, ad esempio:

$$A \vee B, \quad C$$

Invece è **insoddisfacibile** se non è vera in nessun modello, ad esempio:

$$\text{False}, \quad A \wedge \neg A$$

La soddisfacibilità è collegata alla derivazione dal seguente teorema:

$$KB \models \alpha \iff (KB \wedge \neg \alpha) \text{ è insoddisfacibile}$$

4.3 Sistema di inferenza

Un sistema di inferenza è un insieme di regole che permettono di derivare nuove frasi logiche da frasi esistenti. Le regole sono scritte nella forma:

$$\frac{A_1 \dots A_k}{A} \quad \frac{\text{Premesse}}{\text{Conclusioni}}$$

Definizione 4.1. Una derivazione A è derivata da un insieme di formule Γ con un sistema di inferenza \mathcal{R} ($\Gamma \vdash_{\mathcal{R}} A$) se esiste una sequenza A_1, \dots, A_n di formule tale che:

- $A_n = A$
- $\forall i \in \{1, \dots, n\}$ è vera una delle seguenti:
 1. $A_i \in \Gamma$
 2. A_i è una derivazione diretta delle formule nella sequenza precedente

La sequenza A_1, \dots, A_n è una **dimostrazione** di A . Γ sono le **premesse** (assunzioni o ipotesi) per A .

4.3.1 Proprietà di un sistema di inferenza

- **Correttezza delle regole di inferenza:** Le conclusioni devono essere delle conseguenze logiche delle premesse
- **Completezza:** Se una formula è una conseguenza logica delle premesse, allora deve essere possibile derivarla usando le regole di inferenza
- **Completezza refutazionale:** Esiste una derivazione di \Box se le ipotesi unite alla negazione della conclusione sono insoddisfacibili:

$$\Box \text{ è derivabile da } H \cup \{\neg \psi\} \iff H \text{ è insoddisfacente}$$

4.4 Problema di deduzione

Un problema di deduzione consiste nel determinare se una certa formula logica può essere derivata da un insieme di formule esistenti:

$$\Gamma \models \alpha$$

Per risolverlo si possono usare due approcci principali:

- **Dimostrazione per assurdo** (reductio ad absurdum): Si dimostra che

$$\Gamma \wedge \neg \alpha \text{ è insoddisfacibile}$$

I principali metodi sono la **resolution**

- **Forward/backward reasoning**: È un algoritmo polinomiale corretto e completo per un insieme limitato di formule logiche (Horn clauses)

4.4.1 Resolution

Per usare la resolution si deve prima convertire ogni formula logica in **forma normale congiuntiva** (CNF). La CNF è una congiunzione di clausole (letterali), ad esempio:

$$(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$$

La regola di inferenza della resolution è:

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

Dove l_i e m_j sono letterali complementari, cioè uno è la negazione dell'altro. La risoluzione è corretta e completa per la logica proposizionale.

Esempio 4.5. Alcune applicazioni della risoluzione sono le seguenti:

$$\frac{A \vee B \quad \neg B}{A}$$

4.4.2 Conversione in CNF

Consideriamo la formula logica:

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

Per convertire una formula logica in CNF si seguono i seguenti passi:

1. Elimina \iff , sostituendo $\alpha \iff \beta$ con $(\alpha \implies \beta) \wedge (\beta \implies \alpha)$

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

2. Elimina \implies , rimpiazzando $\alpha \implies \beta$ con $\neg \alpha \vee \beta$

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

3. Muovi \neg dentro usando le leggi di De Morgan e doppia negazione:

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

4. Applica la regola della distribuzione:

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

L'algoritmo della risoluzione è il seguente:

```

1 function PL-Resolution(KB, alpha) returns true or false
2   inputs: KB, the knowledge base, a sentence in propositional logic
3           alpha, the query, a sentence in propositional logic
4   clauses <- the set of clauses in the CNF representation of KB and
5               not alpha
6   new <- { }
7   loop do
8     for each Ci, Cj in clauses do
9       resolvents <- PL-Resolve(Ci, Cj)
10      if resolvents contains the empty clause then return true
11      new <- new union resolvents
12  if new is included in clauses then return false
13  clauses <- clauses union new

```

La risoluzione è da applicare solo a due singoli letterali alla volta.

Esempio 4.6. Consideriamo la knowledge base:

$$KB = (B_{1,1} \iff (P_{1,2} \vee P_{2,1}) \wedge \neg B_{1,1}) \quad \alpha = \neg P_{1,2}$$

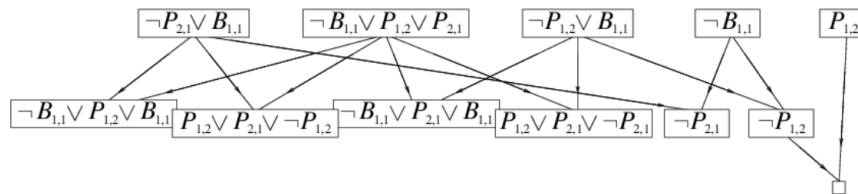


Figura 25: Esempio di risoluzione

4.5 Forward e backward chaining

Per usare il forward e backward chaining si deve prima convertire ogni formula logica in **Horn clauses**. Ciascuna clausola deve rispettare una delle seguenti regole:

- È un singolo simbolo proposizionale, oppure
- È una congiunzione di simboli proposizionali che implica un singolo simbolo proposizionale, ad esempio:

$$(A \wedge B) \implies C$$

Questa forma permette di avere come regola il **modus ponens**, cioè:

$$\frac{\alpha_1, \dots, \alpha_n \quad \alpha_1 \wedge \dots \wedge \alpha_n \implies \beta}{\beta}$$

Questi algoritmi hanno entrambi complessità lineare.

4.5.1 Forward chaining

L'idea di questo algoritmo è quella di trovare qualsiasi regola le cui premesse siano soddisfatte dalla knowledge base e aggiungere la conclusione alla knowledge base finchè non si trova la formula desiderata.

```

1 function PL-FC-Entails?(KB, q) returns true or false
2   inputs: KB # knowledge base, a set of propositional Horn clauses
3           q # query, a prop. symbol
4   local variables: count # a table, indexed by clause, initially
                     the number of premises
5                   inferred # a table, indexed by symbol, each
                     entry initially false
6                   agenda # a list of symbols, initially the
                     symbols known in KB
7   while agenda is not empty do
8     p <- Pop(agenda)
9     unless inferred[p] do
10      inferred[p] <- true
11      for each Horn Clause c in whose premise p appears do
12        decrement count[c]
13        if count[c] = 0 then do
14          if Head[c] = q then return true
15          Push(Head[c], agenda)
16   return false

```

Definizione 4.2 (Dimostrazione di completezz). Forward checking deriva tutte le formule atomiche che possono essere derivate dalla knowledge base.

1. Raggiunge un **punto fisso** dove non si possono più derivare nuove formule atomiche
2. Si considera lo stato finale come un modello m assegnando true o false ai simboli
3. Ogni clausola nella knowledge base originale è vera in m
4. Quindi m è un modello di KB
5. Per ogni α (formula atomica), se la knowledge base deriva α :

$$KB \models \alpha$$

allora α è vera in **ogni** modello di KB, incluso m

4.5.2 Backward chaining

L'idea di questo algoritmo è quella di iniziare dalla formula desiderata e cercare di dimostrare che è vera cercando regole che la concludono e poi cercando di dimostrare che le premesse di queste regole sono vere. Bisogna fare attenzione a:

- Evitare cicli: controllare se il nuovo subgoal è già nella lista dei subgoal
- Evitare ripetizioni: evitare di dimostrare lo stesso subgoal più volte

4.5.3 Differenze tra forward e backward chaining

- **Forward chaining:** È **data-driven**, cioè un processo automatico che parte dai fatti noti
- **Backward chaining:** È **goal-driven**, cioè parte da un obiettivo specifico da raggiungere. La complessità può essere molto meglio di $O(n)$ nella dimensione della knowledge base

5 Rappresentare l'incertezza

Esempio 5.1. Consideriamo l'azione A_t = lasciare l'aeroporto t minuti prima del volo.

Ci chiediamo se A_t ci porterà al volo in orario. I problemi sono:

- **Osservabilità parziale:** non sappiamo se ci saranno ingorghi stradali, condizioni meteo avverse, ecc...
- **Sensori rumorosi:** le previsioni del tempo e le condizioni del traffico non sono sempre accurate
- **Incertezza nel risultato delle azioni:** si potrebbero avere imprevisti come incidenti stradali, guasti dell'auto, ecc...
- **Complessità immensa:** non è possibile modellare e predire ogni possibile evento

Ci sono talmente tante precondizioni che non possiamo tenerle tutte in considerazione.

Ci sono tre principali metodi per gestire l'incertezza:

- **Default o nonmonotonic logic:** si fanno assunzioni che sono vere a meno che non si dimostri il contrario
- **Fuzzy logic:** cerca di rendere la logica non binaria, permettendo di esprimere **gradi di verità**, ad esempio valori tra 0 e 1
- **Probabilità:** gli eventi sono binari (veri o falsi), ma con una certa probabilità.

5.1 Probabilità

Le asserzioni probabilistiche sommano gli effetti di:

- **Lazyness:** mancanza di rappresentare informazioni complete
- **Ignorance:** mancanza di conoscenza su fatti specifici

Esiste anche la probabilità **soggettiva** o **Bayesiana**, che rappresenta il grado di fiducia in una certa asserzione, basata sulle conoscenze attuali, ad esempio:

$$P(A_{25} \mid \text{nessun incidente riportato}) = 0.06$$

Questi non sono gradi di verità, ma gradi di **conoscenza** (belief). Le probabilità cambiano in base alle nuove evidenze.

5.1.1 Decidere con incertezza

Consideriamo di avere le seguenti probabilità:

$$\begin{aligned} P(A_{25} \text{ mi fa arrivare in tempo} \mid \dots) &= 0.04 \\ P(A_{90} \text{ mi fa arrivare in tempo} \mid \dots) &= 0.70 \\ P(A_{120} \text{ mi fa arrivare in tempo} \mid \dots) &= 0.95 \\ P(A_{1440} \text{ mi fa arrivare in tempo} \mid \dots) &= 0.9999 \end{aligned}$$

Non è immediato quale azione scegliere, questo dipende sulle **preferenze** di chi sceglie. Le preferenze sono rappresentate dalla **Utility theory**. Per scegliere l'azione che massimizza l'utilità attesa si usa la **Decision theory** che combina l'utility theory con la **Probability theory Maximum Expected Utility (MEU)**.

5.1.2 Basi di probabilità

- Il **sample space** Ω è l'insieme di tutti i possibili risultati di un esperimento casuale
- $\omega \in \Omega$ è un **sample point** (o mondo possibile, o evento atomico)
- Un **modello probabilistico** è un **sample space** con un assegnamento di probabilità $P(\omega)$ per ogni $\omega \in \Omega$
- Un evento A è un sottoinsieme di Ω . La probabilità di un evento è la somma delle probabilità dei sample point che lo compongono:

$$P(A) = \sum_{\omega \in A} P(\omega)$$

- Le variabili casuali possono avere vari domini e sono soggette al cambiamento, cioè non si possono decidere a priori i loro valori. P induce una **distribuzione di probabilità** su ogni variabile casuale, cioè una funzione che assegna una probabilità ad ogni possibile valore della variabile:

$$P(X = x_i) = \sum_{\omega \in \Omega \mid X(\omega) = x_i} P(\omega)$$

- Una **proposizione** è la probabilità che una variabile casuale assuma un certo valore:

$$P(X = x_i) \text{ è la proposizione "X assume il valore } x_i \text{"}$$

Le proposizioni possono essere usate anche per creare modelli di logica proposizionale, ad esempio date due variabili casuali A e B :

- evento a = insieme di sample points in cui $A(\omega) = \text{true}$
- evento $\neg a$ = insieme di sample points in cui $A(\omega) = \text{false}$
- evento $a \wedge b$ = punti dove $A(\omega) = \text{true}$ e $B(\omega) = \text{true}$

La probabilità implica che due eventi in relazione tra loro hanno probabilità collegate. Quando si calcolano le probabilità quindi bisogna considerare l'intersezione degli eventi, sottraendo le sovrapposizioni dalla somma delle probabilità.

5.1.3 Variabili casuali

Le variabili casuali sono proposizioni atomiche, mentre invece le proposizioni composte sono create combinando variabili casuali. I tipi di variabili casuali sono:

- **Proporzionali o booleane:** assumono valori vero o falso, ad esempio:

Carie = ho una carie?

Carie = true è una proposizione atomica, scritta anche come carie

- **Discrete** (finite o infinite): assumono un insieme numerabile di valori

Previsioni del tempo = {sole, pioggia, neve, nuvoloso}

Previsioni del tempo = pioggia è una proposizione. I valori devono essere esaustivi e mutuamente esclusivi

- **Continue** (limitate o illimitate): assumono un insieme non numerabile di valori

Temp = 21.6

5.1.4 Eventi atomici

Gli eventi atomici sono assegnazioni di tutte le variabili casuali. Gli eventi atomici hanno le seguenti proprietà:

1. Mutuamente esclusivi, cioè non possono essere veri contemporaneamente
2. Esaustivi, cioè uno di essi deve essere vero
3. Conseguono una verità per ogni proposizione
4. Qualsiasi proposizione è logicamente equivalente ad una disgiunzione di eventi atomici rilevanti

Gli eventi atomici sono come i modelli per la logica proposizionale.

5.1.5 Probabilità a priori

La probabilità a priori è la probabilità di un evento senza alcuna conoscenza aggiuntiva. È analoga ai fatti nella knowledge base logica.

5.1.6 Probabilità congiunta

La probabilità congiunta rappresenta la probabilità di ogni evento atomico di un insieme di variabili casuali.

Esempio 5.2. Consideriamo la probabilità:

$$P(\text{Weather}, \text{Cavity})$$

Questa probabilità condizionata è una matrice di valori:

Weather =	Sunny	Rain	Cloudy	Snow
Cavity = True	0.144	0.02	0.016	0.02
Cavity = False	0.576	0.08	0.064	0.08

Tabella 2: Esempio di probabilità condizionata

Se si sommano tutte queste probabilità si ottiene 1, mentre se si sommano le probabilità per una riga o colonna si ottiene la probabilità marginale, cioè la probabilità incondizionata di una variabile casuale.

5.1.7 Probabilità continua

La probabilità continua è definita tramite una funzione parametrica di un valore chiamata **funzione di densità di probabilità** (PDF).

5.1.8 Probabilità condizionata

La probabilità condizionata rappresenta la probabilità di un evento dato che un altro evento sia vero. Ad esempio:

$$P(\text{carie} \mid \text{dolore}) = 0.6$$

rappresenta che la probabilità di avere una carie, dato che l'unica informazione disponibile è quella in cui si ha dolore, è del 60%.

La notazione per distribuzioni condizionate è:

$$P(C \mid T)$$

che è un vettore da due vettori di due elementi:

$$P(C \mid T) = [[P(c \mid t), P(\neg c \mid t)], [P(c \mid \neg t), P(\neg c \mid \neg t)]]$$

Alcune informazioni possono essere irrilevanti e quindi possono essere ignorate. La probabilità condizionata può essere calcolata come:

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)} \text{ se } P(b) > 0$$

dove $P(a \wedge b)$ è la probabilità congiunta di a e b e può essere anche scritta come (regola del prodotto):

$$P(a \wedge b) = P(a \mid b) \cdot P(b) = P(b \mid a) \cdot P(a)$$

Esiste anche la regola della catena che permette di rappresentare una probabilità generica utilizzando i condizionali:

$$\begin{aligned} P(X_1, \dots, X_n) &= P(X_1, \dots, X_{n-1}) \cdot P(X_n | X_1, \dots, X_{n-1}) \\ &= P(X_1, \dots, X_{n-2})P(X_{n-1} | X_1, \dots, X_{n-2})P(X_n | X_1, \dots, X_{n-1}) \\ &= \dots \\ &= \prod_{i=1}^n P(X_i | X_1, \dots, X_{i-1}) \end{aligned}$$

5.1.9 Inferenza per enumerazione

Per ogni proposizione φ , si sommano gli eventi atomici dove è vero:

$$P(\varphi) = \sum_{\omega: \omega \models \varphi} P(\omega)$$

Ricordando che:

- Ogni proposizione φ è equivalente alla disgiunzione degli eventi atomici in cui è vera
- Gli eventi atomici sono mutuamente esclusivi

Esempio 5.3. Consideriamo la seguente tabella di probabilità congiunta:

	<i>toothache</i>		\neg <i>toothache</i>	
	<i>catch</i>	\neg <i>catch</i>	<i>catch</i>	\neg <i>catch</i>
<i>cavity</i>	.108	.012	.072	.008
\neg <i>cavity</i>	.016	.064	.144	.576

Figura 26: Tabella di esempio

La probabilità di avere *toothache* è data dalla somma delle probabilità degli eventi atomici in cui *toothache* è vero:

	<i>toothache</i>		\neg <i>toothache</i>	
	<i>catch</i>	\neg <i>catch</i>	<i>catch</i>	\neg <i>catch</i>
<i>cavity</i>	.108	.012	.072	.008
\neg <i>cavity</i>	.016	.064	.144	.576

Figura 27: Calcolo della probabilità di *toothache*

$$P(\text{toothache}) = 0.108 + 0.012 + 0.016 + 0.064 = 0.2$$

Si può anche calcolare la probabilità congiunta di due eventi:

	<i>toothache</i>		\neg <i>toothache</i>	
	<i>catch</i>	\neg <i>catch</i>	<i>catch</i>	\neg <i>catch</i>
<i>cavity</i>	.108	.012	.072	.008
\neg <i>cavity</i>	.016	.064	.144	.576

Figura 28: Calcolo della probabilità congiunta di toothache e cavity

$$P(\text{cavity} \wedge \text{toothache}) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28$$

Oppure anche probabilità condizionate:

	<i>toothache</i>		\neg <i>toothache</i>	
	<i>catch</i>	\neg <i>catch</i>	<i>catch</i>	\neg <i>catch</i>
<i>cavity</i>	.108	.012	.072	.008
\neg <i>cavity</i>	.016	.064	.144	.576

Figura 29: Calcolo della probabilità condizionata di not cavity dato toothache

$$\begin{aligned}
 P(\neg \text{cavity} \mid \text{toothache}) &= \frac{P(\neg \text{cavity} \wedge \text{toothache})}{P(\text{toothache})} \\
 &= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} \\
 &= 0.4
 \end{aligned}$$

5.1.10 Normalizzazione

L'idea è quella di calcolare la distribuzione di probabilità su una variabile di interesse fissando delle **variabili di evidenza** e sommando su tutte le **variabili nascoste**, cioè quelle non di interesse.

Esempio 5.4. Consideriamo la tabella dell'esempio precedente, si può calcolare la probabilità condizionata e considerare il denominatore come una costante di normalizzazione:

	<i>toothache</i>		\neg <i>toothache</i>	
	<i>catch</i>	\neg <i>catch</i>	<i>catch</i>	\neg <i>catch</i>
<i>cavity</i>	.108	.012	.072	.008
\neg <i>cavity</i>	.016	.064	.144	.576

Figura 30: Calcolo della probabilità condizionata di cavity dato toothache

$$\begin{aligned}
 P(\text{Cavity} \mid \text{Toothache}) &= \alpha P(\text{Cavity}, \text{Toothache}) \\
 &= \alpha \begin{bmatrix} P(\text{Cavity} = \text{true}, \text{Toothache} = \text{true}) \\ P(\text{Cavity} = \text{false}, \text{Toothache} = \text{true}) \end{bmatrix} \\
 &= \alpha \begin{bmatrix} \langle 0.108, 0.016 \rangle \\ \langle 0.012, 0.064 \rangle \end{bmatrix} \\
 &= \alpha \langle 0.12, 0.08 \rangle \\
 &= \langle 0.6, 0.4 \rangle
 \end{aligned}$$

Il risultato finale è un vettore la cui somma dei valori è 1.

5.2 Indipendenza

Due variabili casuali A e B sono indipendenti se e solo se:

$$P(A \mid B) = P(A) \quad \text{oppure} \quad P(B \mid A) = P(B) \quad \text{oppure} \quad P(A, B) = P(A) \cdot P(B)$$

5.2.1 Indipendenza condizionata

Non sempre si possono considerare due variabili casuali come indipendenti, ma possono esistere delle terze variabili che non influenzano la relazione tra le prime due.

Esempio 5.5. Consideriamo l'esempio del dentista, se sappiamo che una persona ha una carie, la probabilità che la sonda del dentista si incastri tra i denti non dipende dal fatto che la persona abbia dolore ai denti o meno. Quindi:

$$P(\text{catch} \mid \text{toothache}, \text{cavity}) = P(\text{catch} \mid \text{cavity})$$

e lo stesso vale se la persona non ha una carie:

$$P(\text{catch} \mid \text{toothache}, \neg \text{cavity}) = P(\text{catch} \mid \neg \text{cavity})$$

Quindi Catch è **condizionatamente indipendente** da Toothache dato Cavity:

$$P(\text{Catch} \mid \text{Toothache}, \text{Cavity}) = P(\text{Catch} \mid \text{Cavity})$$

Si può quindi ignorare una variabile quando si calcola la probabilità condizionata.