Agent Based Artificial Intelligence

Stefano Di Lena

2025

Indice

Capitolo 1

Intelligent Agents

Un **agent** è tutto ciò che può essere visto come un sistema che percepisce il suo *ambiente* tramite i *sensori* ed agisce su di esso tramite *attuatori*.

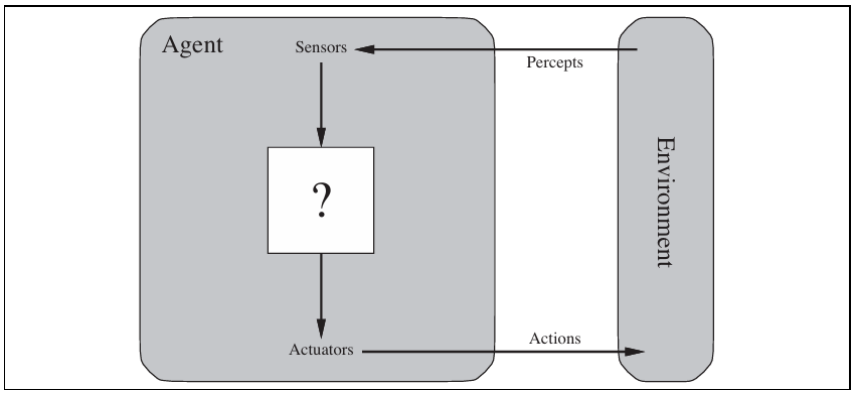

Figura 1.1: Gli agenti interagiscono con l'ambiente attraverso sensori ed attuatori.

Il comportamento di un agente è descritto, matematicamente, dal **agent function** che descrive la corrispondenza di una qualsiasi sequenza percettiva ed una specifica azione. Questa funzione è implementata, *internamente* (in maniera concreta, in esecuzione in un sistema fisico), da un **agent program**.

Consideriamo l'agente aspirapolvere che si muove nel seguente ambiente.

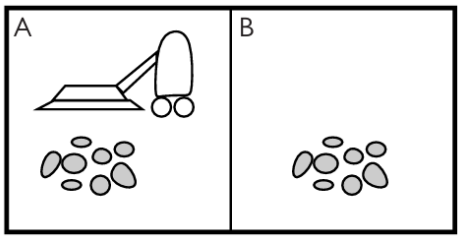

Figura 1.2: Un mondo per l'aspirapolvere con due sole posizioni.

Ogni posizione può essere pulita o sporca e l'agente può decidere di aspirare, andare a sinistra, a destra oppure non fare niente.

Un semplice esempio è: "se il riquadro corrente è sporco allora aspira, altrimenti muoviti sull'altro blocco". Altri esempi sono descritti nella tabella successiva.

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

Figura 1.3: Tabulazione parziale di una semplice funzione agente per il mondo dell'aspirapolvere mostrato sopra.

1.1 Rationality

Un **rational agent** sceglie un'azione che massimizza il valore atteso della sua misura di prestazione, per ogni possibile sequenza di percezioni.

Una **performance measure** valuta una sequenza di stai dell'ambiente.

Razionale non vuol dire onnisciente, né chiaroveggente (non è detto che i risultati delle azioni siano quelli attesi). Per razionale intendiamo che esplorando l'ambiente riesce ad apprendere in autonomia.

1.2 Specificare un ambiente

Nell'esempio dell'aspirapolvere abbiamo specificato la performance measure, l'ambiente esterno, gli attuatori ed i sensori dell'agente. Tutto questo può essere raggruppato nel termine **task environment**. Viene anche nominiamo considerando gli acronimi, come descrizione **PEAS** (Performance, Environment, Actuators, Sensors). Quanto meglio si riescono a descrivere questi elementi, tanto migliori saranno le performance dell'agente.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Figura 1.4: Descrizione PEAS dell'ambiente operativo di un taxi automatizzato.

1.2.1 Proprieties of thask environments

Un ambiente operativo si dice **completamente osservabile** quando i sensori dell'agente misurano tutti gli aspetti *rilevanti* per la scelta dell'azione. A causa della presenza di rumore o della scarsa precisione di alcuni sensori un ambiente potrebbe essere **parzialmente osservabile**. Se l'agente non dispone di sensori è **inosservabile**.

Un ambiente si dice **deterministico** se lo stato successivo dell'ambiente è completamente determinato dallo stato corrente e dall'azione eseguita dall'agente. In caso contrario si dice **stocastico**, in questo caso il modello è esplicitamente associato a probabilità.

In un ambiente operativo **episodico** l'esperienza dell'agente è divisa in episodi atomici. In ogni episodio l'agente riceve una percezione e poi esegue una singola azione. Un episodio non dipende dalle azioni intraprese in quelle precedenti. Negli ambienti **sequenziali**, invece, ogni decisione può influenzare le successive.

Un ambiente si dice **dinamico** per un agente se può cambiare mentre questo sta decidendo come agire. In caso contrario diciamo che è **statico**.

La distinzione tra **discreto** e **continuo** si applica allo *stato* dell'ambiente, al modo in cui è gestito il *tempo*, alle *percezioni* ed *azioni* dell'agente. [Esempi: la scacchiera ha un numero finito di stati distinti (discreto), la guida di un taxi è un problema continuo nel tempo (la sua velocità e posizione cambia continuamente nel tempo)].

Un agente che risolve un puzzle da solo è chiaramente in un ambiente **single-agent**. Se abbiamo più agenti che comunicano tra di loro, siamo in un ambiente **multi-agent**, il quale può essere:

- *competitivo* (ad esempio una partita a scacchi tra due agenti);
- *cooperativo* (ad esempio nella guida autonoma dove ogni agente deve collaborare per evitare gli incidenti).

Conoscere il tipo di ambiente è fondamentale per la corretta progettazione di un agente. Il mondo reale è parzialmente osservabile, multi-agent, non deterministico, sequenziale, dinamico, continuo e ignoto.

1.3 The Structure of Agents

Definiamo quattro tipi base di programma agente che contengono i principi alla base di quasi tutti i sistemi intelligenti. Vedremo anche come queste classi possono essere migliorate inserendo una forma di apprendimento.

1.3.1 Simple reflex agents

L'agente reattivo semplice sceglie le azioni sulla base della percezione corrente, ignorando la storia percettiva precedente.

Utilizza le regole *condizione-azione* (anche nota come *if-then*). Nell'esempio del taxi a guida autonoma, se l'agente nota che una macchina davanti sta frenando allora deve iniziare anche lui a frenare.

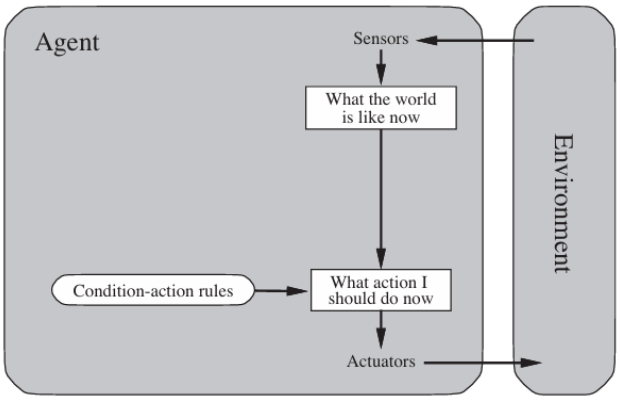

Figura 1.5: Diagramma schematico di un agente reattivo semplice.

Sono la tipologia più semplice di agente, ma la loro intelligenza è molto limitata. Funzionerà solo nel caso in cui l'ambiente è completamente osservabile (anche una minima parte di non-osservabilità può causare grandi problemi).

```
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
```

Figura 1.6: Programma per l'agente reattivo semplice nell'ambiente in Figura 1.2 che implementa la funzione agente rappresentata dalla Figura 2.3.

Nello schema di Figura 1.5 usiamo i *rettangoli* per indicare lo stato interno attuale del processo di decisione dell'agente e gli *ovali* per rappresentare le azioni di fondo usate nel processo.

1.3.2 Model-based reflex agents

L'agente che si basa di un *modello* del mondo, ovvero uno che mantiene uno stato interno che dipende dalla storia delle percezioni e che riflette alcuni aspetti dello stato corrente non-osservabili, viene detto model-based.

Notiamo dalla Figura 1.7 come lo stato percepito è combinato ad uno stato interno per generare una versione aggiornata dello stato corrente, basandosi sul modello dell'agente di "come il funziona il mondo".

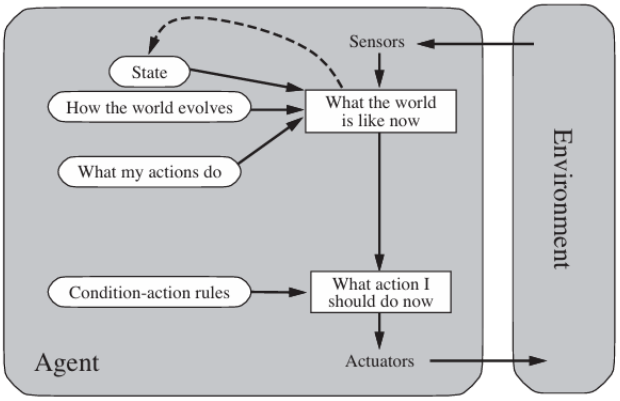

Figura 1.7: Un agente reattivo basato su modello.

1.3.3 Goal-based agents

L'agente che si muove per raggiungere un obiettivo. Considerando sempre l'esempio del taxi, questo ad un incrocio potrebbe girare a sinistra, a destra o proseguire dritto. La decisione giusta dipende dalla destinazione finale impostata. Quindi l'agente oltre allo stato corrente necessita anche di informazioni sull'obiettivo, per descrivere le situazioni *desiderabili*.

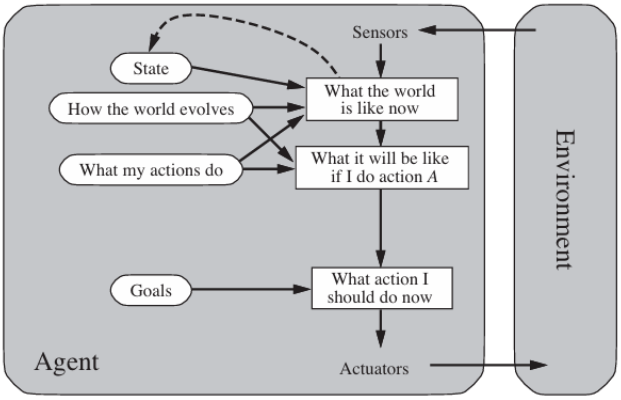

Figura 1.8: Un agente basato su modello che raggiunge degli obiettivi.

Questo tipo di agente sembra meno efficiente, ma risulta molto più flessibile (la conoscenza che guida le sue decisioni è rappresentata esplicitamente e può essere modificata).

1.3.4 Utility-based agents

Gli obiettivi forniscono una distinzione binaria tra stati "contenti" e "scontenti". Una misura delle performance potrebbe confrontare più stati e capire quanto renderebbero felice l'agente una volta raggiunti. Per usare un temine più scientifico diremo *utilità* al posto di felicità o contentezza.

L'agente basato sull'utilità sceglierà, quindi, un azione considerando quanto uno stato è desiderabile o meno, in aggiunta all'importanza dell'obiettivo.

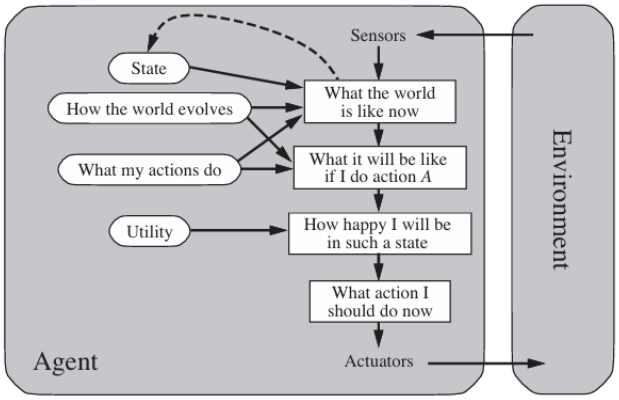

Figura 1.9: Un agente basato su modello che massimizza l'utilità.

1.3.5 Learning agents

Tramite l'apprendimento un agente riesce ad operare in ambienti inizialmente sconosciuti, diventando col tempo più competenti di quanto già fossero all'inizio (basandosi sulla conoscenza iniziale).

Un agente di apprendimento può essere scomposto in quattro componenti dal punto di vista concettuale, come mostrato in Figura 1.10:

- *learning element* - responsabile delle migliorie;
- *performance element* - responsabile nella scelta delle azioni esterne;
- *critic* - da un feedback al learning element sul come l'agente sta operando, il quale viene usato per valutare le modifiche da effettuare per migliorare il performance element nelle operazioni future;
- *problem generator* - suggerisce delle azioni che porteranno ad una nuova esperienza informativa.

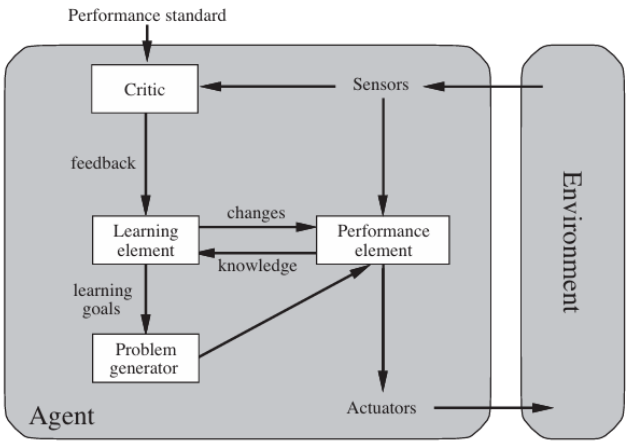

Figura 1.10: Un agente capace di apprendere.

Capitolo 2

Solving Problems By Searching

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

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

Figura 2.1: Un semplice problem-solving agent. Viene formulato prima un obiettivo in un problema, l'agente cerca una sequenza di azioni che possano risolvere il problema e le esegue una alla volta. Quando finisce formula un ulteriore obiettivo e ricomincia.

Immaginiamo un agente che si trovi in vacanza in Romania e vuole visitare le zone più belle, migliorare la conoscenza della lingua e fare un po' di vita notturna, evitando di ubriacarsi. Supponiamo ora che l'agente si trovi nella città di Arad ed abbia un biglietto di ritorno non rimborsabile con partenza da Bucarest. L'agente osserva la segnaletica stradale e nota che ci sono tre strade che partono da Arad e procedono verso Sibiu, Timisoara e Zerind. Nessuna di queste è l'obiettivo, quindi no sa quale strada scegliere.

Se l'agente non ha altre informazioni dovrà compiere la scelta casualmente. Assumiamo, invece, che l'agente abbia a disposizione la mappa (Figura 2.2) e potrà risolvere il problema in quattro fasi.

1. **Goal formulation**: l'agente adotta l'*obiettivo* di raggiungere Bucarest;
2. **Problem formulation**: l'agente elabora una descrizione degli stati e delle azioni necessarie per aggiungere quest'ultimo;

3. **Ricerca**: l'agente simula nel suo modello delle sequenze di azioni, continuando a cercare finché non trova una sequenza che raggiunge l'obiettivo, questa viene detta *soluzione*. Potrebbero capitare dei casi in cui non esiste una soluzione;

4. **Esecuzione**: esegue le azioni specificate nella soluzione, una per volta.

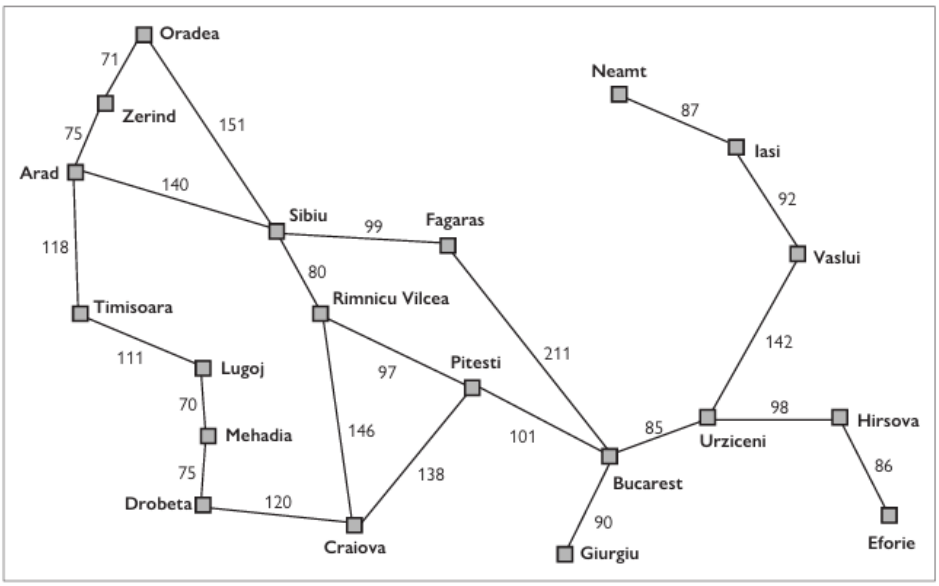

Figura 2.2: Mappa semplificata della Romania.

Problem types

- *single-state problem*: l'agente sa esattamente in quale stato sarà al prossimo passo, la soluzione è una sequenza (deterministico, completamente osservabile).

- *conformant problem*: l'agente potrebbe non avere idea di dov'è, se c'è una soluzione questa è una sequenza (non-osservabile).

- *contingency problem*: percepisce nuove informazioni sullo stato corrente, la soluzione sarà una policy contingente (non-deterministico e/o parzialmente osservabile).

- *exploration problem*: "online" (state space sconosciuto).

2.1 Problemi di ricerca e soluzioni

Un problema può essere definito formalmente da cinque componenti:

- **initial state**, lo stato iniziale, da cui parte l'agente. (Per l'esempio della Romania è Arad);

- **successor function**, le *azioni* possibili dell'agente, partendo da uno stato s (Nel nostro caso da Arad sono possibili {Sibiu, Timisoara, Zerind});

- **goal test**, un insieme di uno o più stati obiettivo che possono essere *impliciti* (Niente Sporcizia, nell'esempio dell'aspirapolvere) oppure *espliciti* (arrivare a Bucarest, nell'esempio della Romania);

- **path cost**, una funzione che assegna un costo numerico a ciascun percorso. Assumiamo che il path cost sia descritto dalla somma dei costi delle azioni fatte individualmente lungo il tragitto. Il **costo del passo** scegliendo l'azione a partendo dallo stato s per raggiungere s' è denotato da $c(s, a, s')$. [Nell'esempio della Romania gli step cost sono rappresentati dalla distanza in Km].

Una **soluzione** al problema è una sequenza di azioni che porta dallo stato iniziale ad uno stato obiettivo.

2.1.1 Formulazione dei problemi

La formulazione del problema (come arrivare a Bucarest) è un **modello**, quindi una descrizione matematica astratta e non qualcosa di reale.

Il processo di rimozione dei dettagli da una rappresentazione prende il nome di **astrazione**. Al fine di arrivare a Bucarest non è necessario conoscere le condizioni della strada, quelle atmosferiche, il traffico, il paesaggio, i compagni di viaggio o la presenza di posti di blocco nelle vicinanze.

Per una buona formulazione del problema serve il giusto livello di dettaglio.

2.2 Example Problems

La risoluzione di problemi è applicata a una vasta gamma di ambienti operativi.

2.2.1 Problemi standardizzati

Un primo esempio di **toy problem** è quello dell'*aspirapolvere*.

- **States:** lo stato è determinato dalla posizione dell'agente e quella della polvere. Ci sono quindi $2 \times 2^2 = 8$ possibili stati. [Un ambiente largo n blocchi avrebbe $n \times 2^n$ stati].

- **Initial state:** ogni stato potrebbe essere iniziale.

- **Azioni:** Sinistra, Destra ed Aspira. [Ambienti più grandi potrebbero includere anche Sopra e Sotto].

- **Goal test:** verificare che ogni quadrato sia pulito.

- **Path cost:** ogni step ha costo 1.

Comfrontato con il mondo reale, questo problema standardizzato ha locations discrete, polvere discreta e pulizia affidabile, inoltre non diventa più sporco.

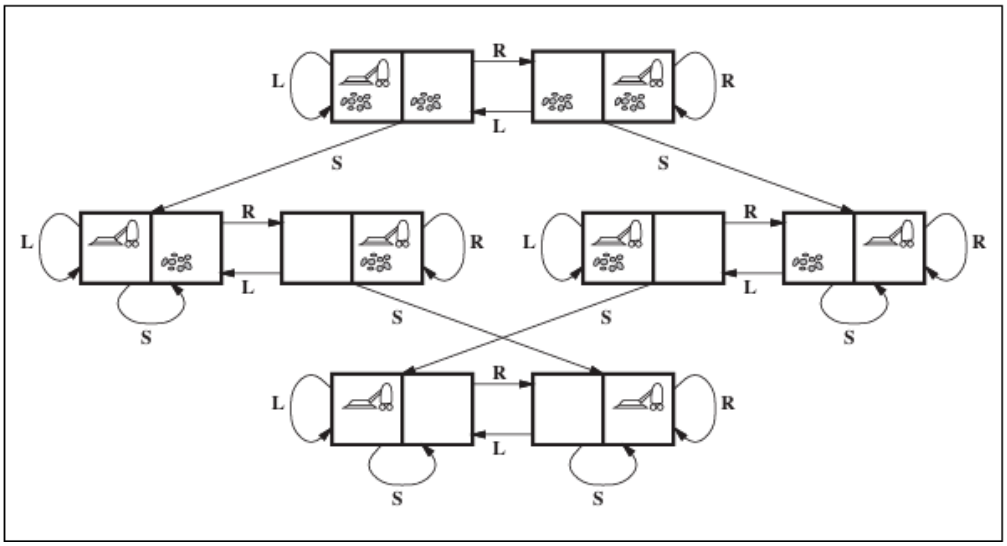

Figura 2.3: Lo state space per il mondo aspirapolvere. Sono possibili le seguenti azioni: L = *Left*, R = *Right*, S = *Suck*.

Un altro esempio è l'*8-puzzle*, il quale consiste in una tavola 3x3 contenente 8 caselle 1x1 (numerate da 1 ad 8) ed uno spazio lasciato bianco. Una casella adiacente ad uno spazio bianco può scorrere nello spazio. L'obiettivo è quello di raggiungere un Goal state che potrebbe essere ad esempio quello in figura.

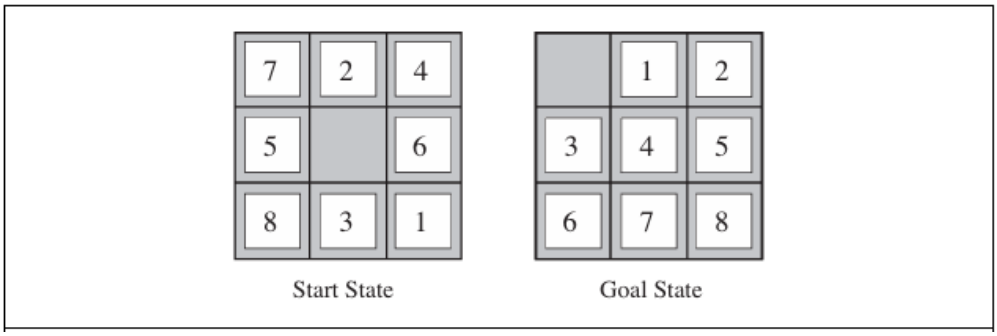

Figura 2.4: Istanza tipica per l'8-puzzle.

- **States:** una descrizione di stato specifica la posizione di ognuno degli 8 tasselli.
- **Initial state:** ogni stato può essere designato come stato iniziale.
- **Azioni:** a differenza nel mondo fisico, in cui sono i tasselli a scorrere, il modo più semplice per descrivere un'azione è quello di pensarla come se fosse lo spazio vuoto a muoversi a Sinistra, Destra, Su o Giù (se lo spazio vuoto si trova su un bordo o su un angolo, non tutte le azioni sono applicabili).

- **Goal state:** ogni stato potrebbe essere quello obiettivo, solitamente però consiste nell'ordinare i numeri (come in Figura 2.4).
- **Path cost:** ogni azione costa 1.

2.2.2 Problemi del mondo reale

Un esempio di **real-world problems** è quello di una *Sequenza di montaggio automatica*.

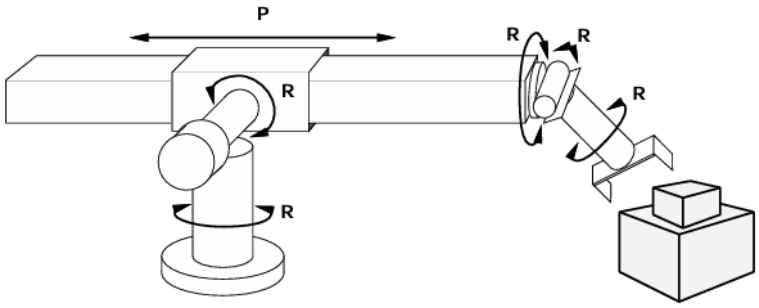

Figura 2.5: Robotic assembly.

In questo problema lo scopo è trovare l'ordine in cui assemblare le varie parti dell'oggetto: se questo è sbagliato, a un certo punto non sarà possibile aggiungere un pezzo senza smontare una parte del lavoro già svolto. Controllare che una determinata azione nella sequenza sia ammissibile è un difficile problema di ricerca geometrico, molto simile alla navigazione dei robot. Di conseguenza, la generazione di azioni legali è la parte più costosa della costruzione di una sequenza di montaggio. Ogni algoritmo, se vuole essere applicabile nella pratica, deve evitare di esplorare lo spazio degli stati nella sua interezza e considerarne solo una frazione molto piccola.

2.3 Searching for Solution

Un *algoritmo di ricerca* riceve in input un problema d ricerca e restituisce una soluzione o un'indicazione di fallimento. Consideriamo degli algoritmi che sovrappongono un **albero di ricerca** al grafo dello spazio degli stati, formando vari cammini a partire dallo stato iniziale e cercando di trovarne uno che raggiunga uno stato obiettivo. Ciascun **nodo** nell'albero corrisponde ad uno stato nello spazio degli stati, mentre i **rami** dell'albero corrispondono ad azioni. La radice dell'albero corrisponde allo stato iniziale del problema.

Lo spazio degli stati descrive l'insieme (eventualmente infinito) degli stati nel mondo e le azioni che consentono le transizioni da uno stato all'altro. L'albero di ricerca descrive i cammini tra questi stati per raggiungere l'obiettivo (potrebbero esserci più cammini per raggiungere qualsiasi stato, ma per ogni nodo c'è un cammino univoco per tornare alla radice).

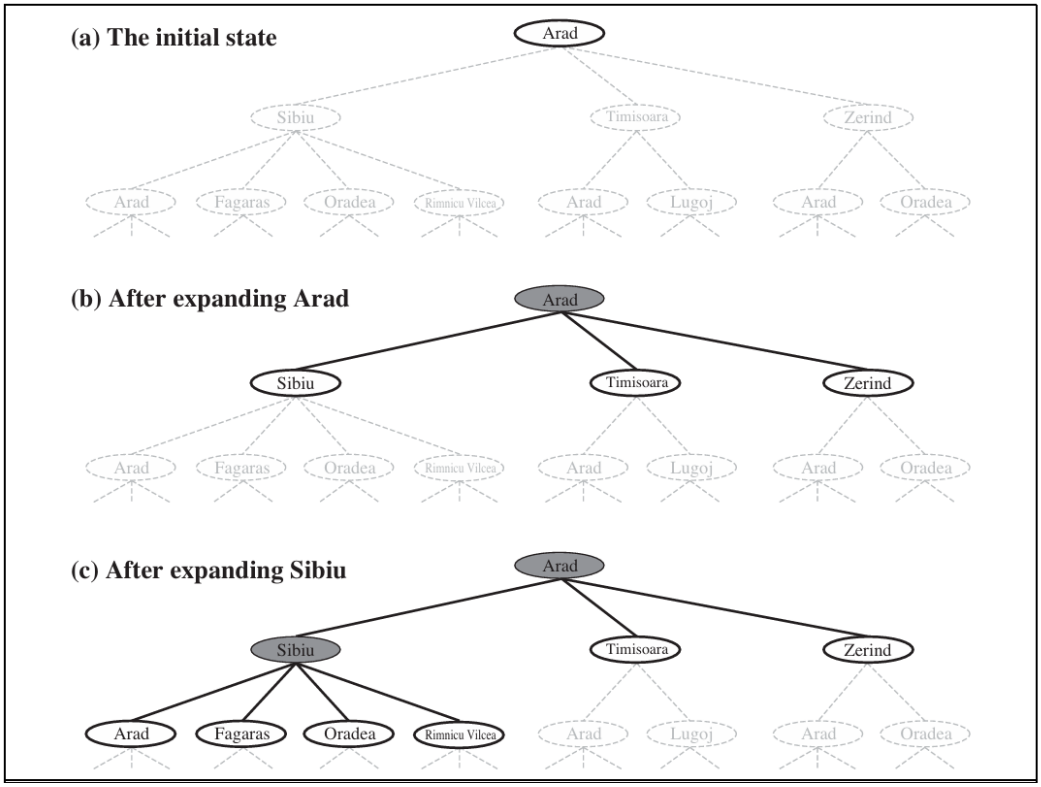

Figura 2.6: Un albero decisionale parziale per cercare una strada che porti a Bucarest da Arad. I nodi che sono stati espansi sono rappresentati in grigio; i nodi che sono stati generati ma non ancora espansi sono cerchiati in grassetto; i nodi che non sono stati ancora generati sono mostrati da linee leggere tratteggiate.

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE(node)) then return node
      fringe ← INSERTALL(EXPAND(node, problem), fringe)

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

build a successor node

Figura 2.7: Una descrizione informale della tree-search generale e la funzione per espandere un nodo.

Partendo da uno stato iniziale, possiamo espandere il nodo (considerando le azioni disponibili per quello stato) e generare un nuovo nodo (chiamato *nodo figlio*) per ognuno degli stati risultati. Ogni nodo contiene anche un puntatore al *nodo padre* (quello che lo ha generato). La collezione di nodi che sono in attesa di essere espansi è chiamata *frontiera* (*fringe*).

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

Figura 2.8: Graph search.

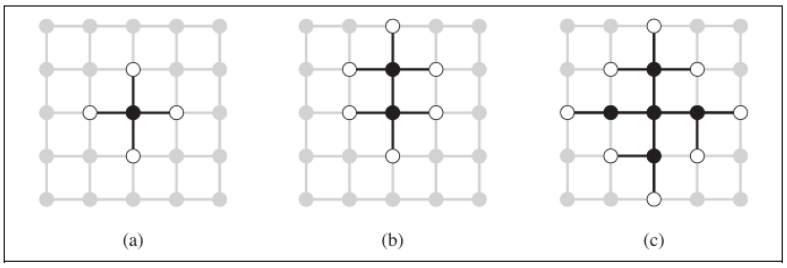

Figura 2.9: La proprietà di separazione della graph-search, illustrata con un problema su griglia rettangolare. La frontiera (nodi bianchi) separano le regioni esplorate dello spazio degli stati (nodi neri) da quelle non esplorate (nodi grigi).

2.3.1 Infrastructure for search algorithms

Ci sono molti modi per rappresentare i nodi, assumiamo che un nodo `n` sia una struttura dati contenente cinque componenti:

- `n.State`: lo stato nello spazio degli stati a cui corrisponde il nodo;
- `n.Parent`: il nodo nell'albero di ricerca che ha generato il nodo corrente;
- `n.Action`: l'operatore che è stato applicato per generare il nodo;
- `n.Path-Cost`: il numero di nodi del cammino dalla radice a questo nodo;
- `n.depth`.

Una struttura dati per memorizzare la frontiera potrebbe essere una **coda**. Le operazioni svolta da una frontiera sono:

- EMPTY?(queue) restituisce *true* se non ci sono più nodi nella coda;
- POP(queue) rimuove il primo elemento della coda e lo restituisce;
- INSERT(element, queue) inserisce un elemento al posto appropriato nella coda.

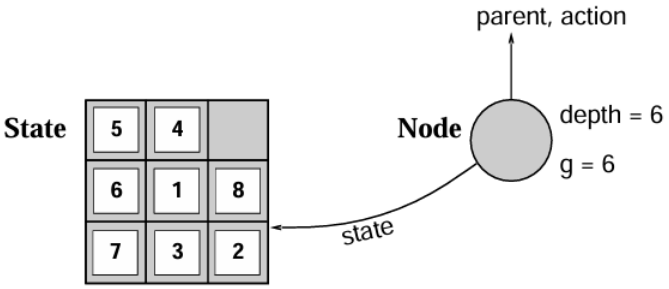

Figura 2.10: I nodi sono strutture dati dalle quali è costruito l'albero decisionale. Ognuno ha un parente, uno stato ed altri campi. Le frecce puntano da un nodo figlio ad uno genitore.

Negli algoritmi di ricerca sono utilizzati tre tipi di code. First-In First-Out (**FIFO**) in cui viene estratto prima il nodo che è stato aggiunto alla coda per primo (utilizzato nella ricerca in ampiezza). Last-In First-Out (**LIFO**), conosciuta anche come *stack*, in cui viene estratto il nodo che è stato aggiunto alla coda per ultimo (utilizzata nella ricerca in profondità). La **priority queue** in cui viene estratto prima il nodo di costo minimo in base ad una funzione di valutazione f (usata nella ricerca best-first).

2.3.2 Cammini ridondanti

Potrebbe capitare che ci siano stati ripetuti nell'albero di ricerca, generati da un ciclo. Quindi, nonostante lo spazio degli stati ha un insieme finito di stati, l'albero di ricerca completo è *infinito*. Si dice che "gli algoritmi che non sanno ricordare il passato sono condannati a ripeterlo".

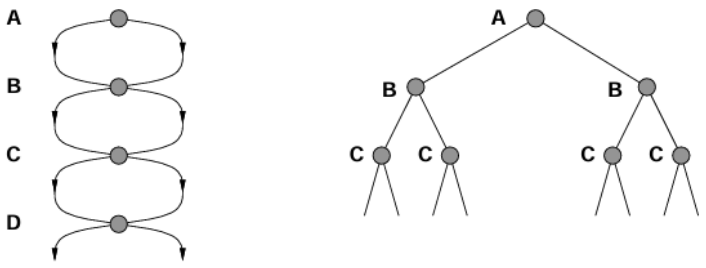

Figura 2.11: Repeated states.

Un primo approccio per evitare tutti i cammini ridondanti è quello di ricordare tutti gli stati precedentemente raggiunti. Un secondo approccio consiste nel non preoccuparsi di ripetere il passato. Un ultimo approccio è quello di controllare la presenza di cammini ciclici senza dover risalire la catena dei padri e verificare se lo stato al termine del cammino è già apparso precedentemente (andando così facendo a risparmiare memoria).

2.3.3 Measuring problem-solving performance

Per individuare una *strategia di ricerca*, ovvero un criterio in base al quale selezionare quale nodo espandere sulla frontiera, ci basiamo su:

- **completezza:** l'algoritmo garantisce di trovare una soluzione (se esiste) e di riportare correttamente il fallimento (se la soluzione non esiste)?
- **complessità temporale:** quanto tempo impiega a trovare la soluzione (questo può essere misurato in secondi o in modo astratto: in base al numero di stati ed azioni)?
- **complessità spaziale:** di quanta memoria necessità per effettuare la ricerca?
- **ottimalità:** la soluzione che trova ha il costo minimo tra tutte?

I parametri utilizzati per valutare la complessità spaziale e temporale dell'albero sono:

- **b** (*branching factor*), numero di successori di un nodo che devono essere considerati. [Consideriamo il valore massimo per il fattore di ramificazione perché vogliamo una stima del caso peggiore];
- **d** (*depth*), rappresenta la profondità del nodo obiettivo nella soluzione migliore;
- **m** (*maximum depth of the state space*), equivale alla massima profondità dell'albero e potrebbe essere infinita.

2.4 Uninformed Search Strategies

Nelle strategie di ricerca non informata non si hanno informazioni su quanto uno stato sia vicino all'obiettivo. É possibile solamente distinguere uno stato obiettivo ed uno non-obiettivo.

2.4.1 Breadth-first search

É una semplice strategia nella quale il root node si espande per primo, poi tutti i suoi successori, poi i successori di quest'ultimi e così via. L'espansione dei nodi sulla frontiera segue una politica FIFO (First-In First-Out).

La ricerca in ampiezza trova sempre una soluzione con un numero minimo di azioni, perché quando genera nodi di profondità d ha già generato tutti i nodi a profondità $d-1$, perciò se uno di essi fosse una soluzione, sarebbe stato trovato. É **completa** in ogni caso.

Supponendo di effettuare la ricerca in un albero uniforme dove ogni stato ha b successori; la radice dell'albero di ricerca genera b nodi, ognuno dei quali genera altri b nodi, per un totale di b^3 al terzo livello, e così via. Supponendo che la soluzione sia a profondità d. Il numero di nodi generati sarà quindi:

$$1 + b + b^2 + \cdots + b^d = O(b^d)$$

Tutti i nodi rimangono in memoria, quindi la **complessità temporale** e **spaziale** sono entrambe $O(b^d)$.

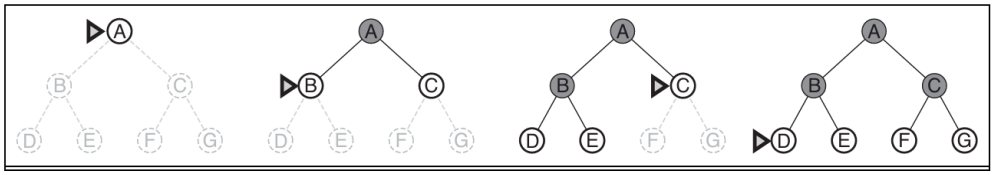

Figura 2.12: Ricerca in ampiezza su un semplice albero binario. Ad ogni passo viene espanso il nodo segnato con il triangolo.

Questi problemi con una complessità esponenziale possono essere risolti con metodi di ricerca non informata solo per istanze piccole.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figura 2.13: Requisiti di tempo e memoria per la breadth-first-search. I numeri mostrati assumono un branching factor $b = 10$; 1 milione di nodi/secondo; 1000 bytes/nodo.

Il problema principale, se ad uno va bene aspettare 13 giorni (per la soluzione di un problema avente profondità 12), è la memoria (nessun computer ha 1 petabyte di memoria).

2.4.2 Uniform-cost search

Nell'algoritmo di *Dijkstra*, invece di espandere il nodo superficiale, si espande il nodo n avente il path cost minore $g(n)$.

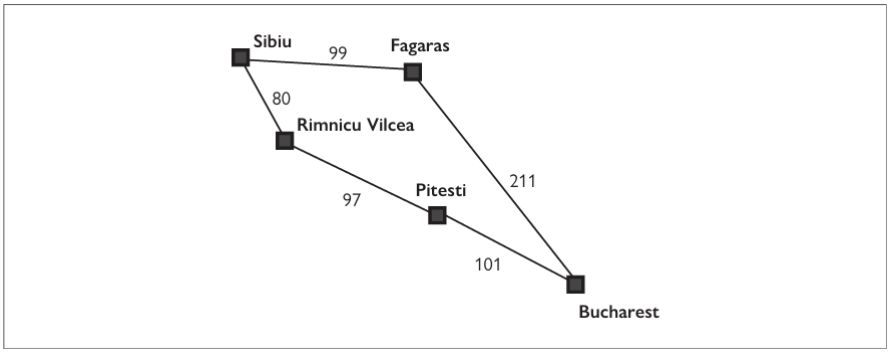

Figura 2.14: Una parte dello spazio degli stati per la Romania.

Se considerialmo il problema di adanre da Sibiu a Bucarest. I nodi successori di Sibiu sono Rimnicu Vilcea e Fagras (che hanno rispettivamente costo 80 e 99). Il nodo di costo minimo è Rimnicu, per cui viene espanso questo, aggiungendo Pitesti il costo diventa $80 + 97 = 177$. Il nodo con costo minimo ora è Fagras e quindi viene espanso, aggiungendo Bucarest con costo $99+211 = 310$. Questo è l'obiettivo ma l'algoritmo controlla se è stato raggiunto l'obiettivo solo quando si espande un nodo, non quando si genera, perciò non ha ancora scoperto che questo cammino porta all'obiettivo e continua. Sceglierà Pitesti come prossima espansione aggiungendo un secondo cammino a Bucarest con costo $80 + 97 + 101 = 278$. Ha un costo inferiore perciò lo considera, determina che è un obiettivo. [N.B. se il controllo fosse avvenuto al momento di generazione di un nodo anziché al momento di espansione del nodo di costo minimo, avremmo restituito un cammino di costo più alto].

La **completezza** è garantita solo se il costo di ogni step è maggiore rispetto ad una costante positiva ϵ. Perché altrimenti potremmo rimanere bloccati in loop eseguendo azioni a costo zero (ad esempio, una sequanza di azioni NoOp[1]).

La complessità della ricerca a costo uniforme è caratterizzata in termini di C^* (costo della soluzione ottima) ed ϵ (limite inferiore imposto al costo di ogni azione, $\epsilon > 0$)).

Nel caso peggiore la **complessità temporale** e **spaziale** dell'algoritmo è $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, che può essere molto maggiore di b^d. [Quando tutte le azioni hanno lo stesso costo la ricerca in ampiezza e quella a costo uniforme sono simili].

Questa ricerca è **ottima** rispetto al costo (la prima soluzione che trova avrà un osto basso almeno quanto quello di ogni altro nodo sulla frontiera). I cammini sono esaminati sistematicamente in ordine di costo crescente.

2.4.3 Depth-first search

La ricerca in profondità espande sempre per primo il nodo a profondità maggiore nella frontiera (viene gestita come una LIFO).

La ricerca procede immediatamente al livello più profondo dell'albero (dove i nodi non hanno successori), una volta che un ramo non può essere più

[1]No Operation.

espanso viene escluso dalla frontiera e la ricerca ricomincia al successivo nodo più profondo che ha successori non esplorati.

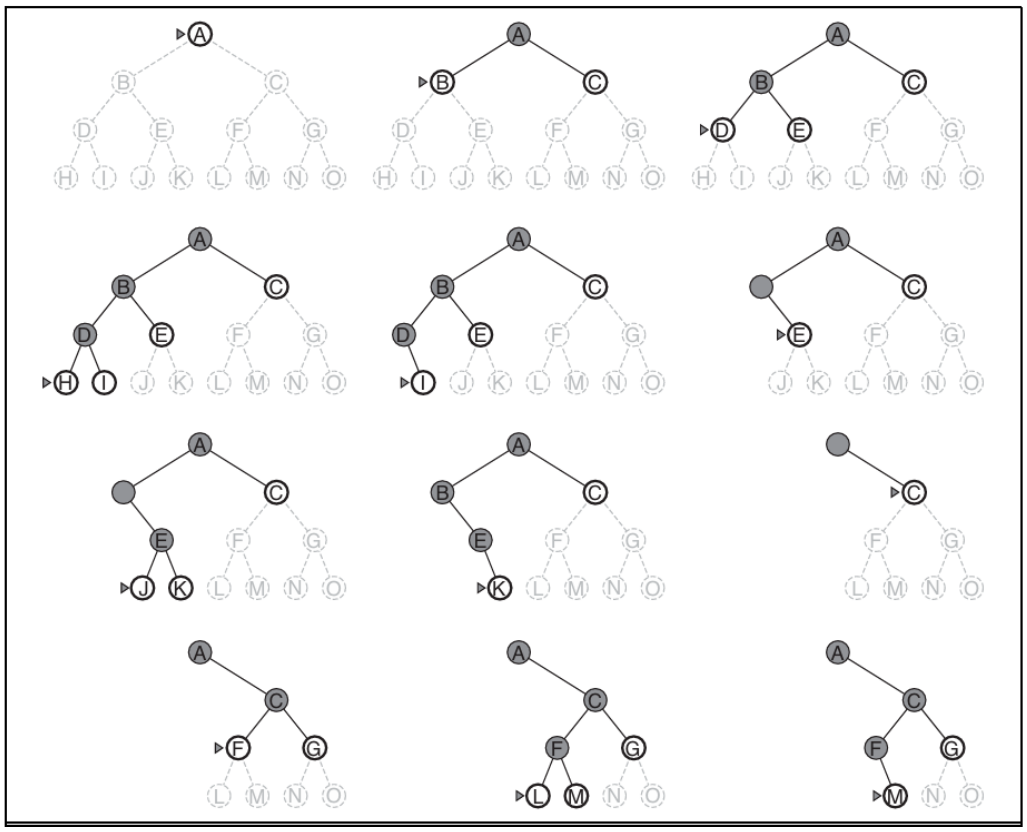

Figura 2.15: Depth-first search di n albero binario. Il nodo obiettivo è M. I nodi esplorati che non hanno figli vengono eliminati dalla memoria.

La ricerca in profondità è **incompleta** in spazi degli stati infiniti, può entrare in cammini infiniti (anche se non ci sono cicli). Mentre per spazi degli stati finiti che sono alberi, la ricerca in profondità è *efficiente* e **completa**.

Questa ricerca non è **ottima** rispetto al costo, infatti restituisce sempre la prima soluzione che trova (anche se non è la meno costosa).

Una ricerca in profondità ad albero richiede un tempo proporzionale al numero degli stati e ha una **complessità di memoria** $O(bm)$, dove b è il fattore di ramificazione ed m la massima profondità dell'albero.

La **complessità temporale** dipende dalla dimensione dello spazio degli stati. Un albero di ricerca depth-first può generare tutti gli $O(b^m)$ nodi. [risulta terribile se $m >> d$, in caso contrario potrebbe essere più veloce della breadth-first].

2.4.4 Depth-limited search

Imponiamo un limite di profondità ℓ alla ricerca depth-limited. Questo risolve il problema dei cicli infiniti, però potrebbe comunque non raggiungere la **completezza** se $\ell < d$ (è probabile se d è sconosciuto).

La **complessità temporale** è $O(b^\ell)$ e quella **spaziale** $O(b\ell)$.

Se conosciamo la profondità, ad esempio $d = 20$ per le città nella mappa della Romania, allora la soluzione (se c'è) deve essere al massimo di lunghezza $\ell = 19$.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Figura 2.16: Un implementazione ricorsiva della depth-limited tree search.

2.4.5 Iterative deepening search

Applichiamo in maniera iterativa la ricerca in profondità limitata, incementando di volta in volta il limite (prima 0, poi 1, poi 2, ect...) fino a quando non si trova una soluzione, oppure restituisce valore *fallimento*.

La ricerca ad approfondimento operativo è **ottima** per problemi in cui tutte le azioni hanno lo stesso costo. É **completa** se il fattore di diramazione è finito.

I **requisiti** di **memoria** sono modesti quando esiste una soluzione: $O(bd)$; mentre quella **temporale** $O(b^d)$.

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

Figura 2.17: L'algoritmo di ricerca ad approfondimento iterativo.

La ricerca ad approfondimento iterativo potrebbe sembrare uno spreco, dato che gli stati vicini all'inizio della ricerca vengono rigenerati più volte. Tuttavia, in molti spazi degli stati la maggior parte dei nodi si trova al livello più basso, perciò la ripetizione dei livelli superiori non ha un grande impatto.

É possibile utilizzare un approccio ibrido che esegue la ricerca in ampiezza finché non si è esaurita quasi tutta la memoria disponibile, e poi un approfondimento iterativo da tutti i nodi della frontiera.

In generale, la ricerca ad approfondimento iterativo è il metodo preferito di ricerca non informata quando lo spazio degli stati in cui cercare è troppo

grande per essere mantenuto in memoria e la profondità della soluzione non è nota.

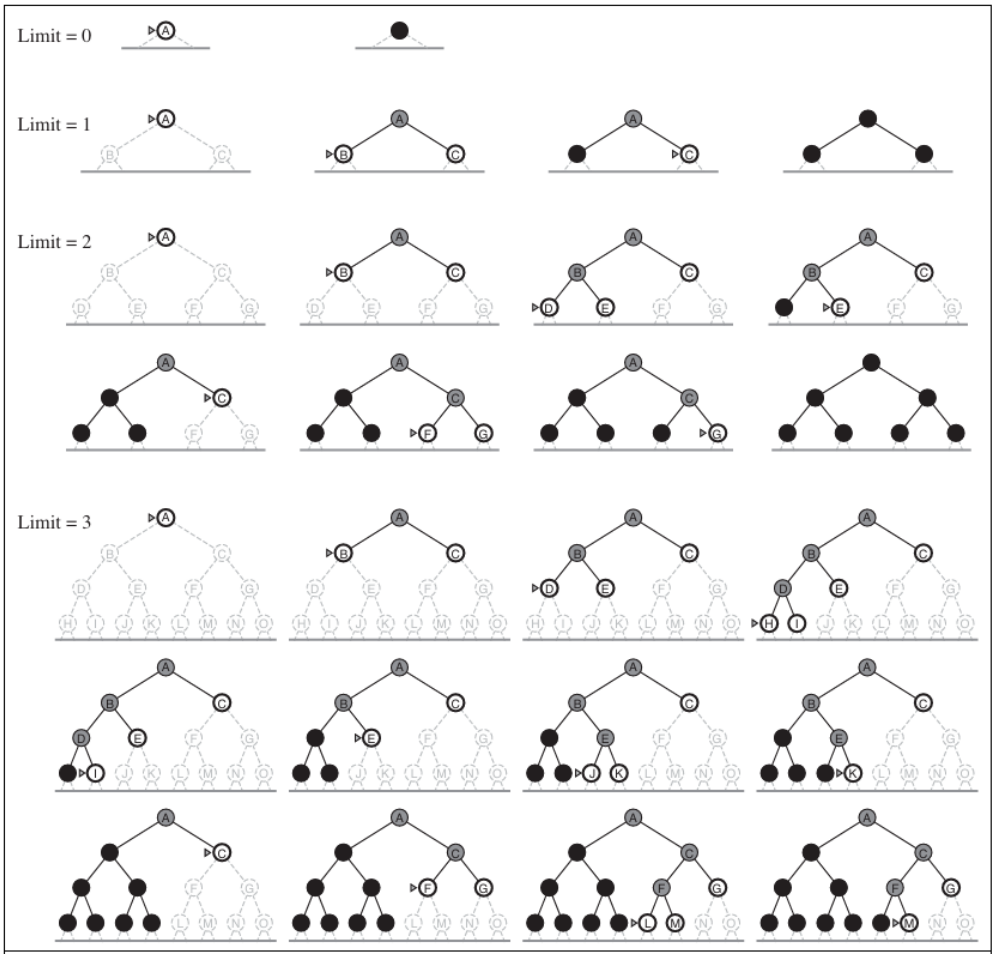

Figura 2.18: Quattro iterazioni dell'algoritmo di ricerca ad approfondimento iterativo su un albero binario.

2.4.6 Bidirectional search

Vengono fatte simultaneamente due ricerche (una dallo stato iniziale al goal e l'altra dall'obiettivo all'initial state), sperando che queste si incontrino nel mezzo.

La **complessità temporale** e uguale a quella **spaziale** e pari a $O(b^{d/2})$, se si usa la breadth-first search in entrambe le direzioni.

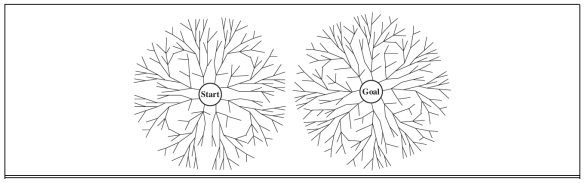

Figura 2.19: Una rappresentazione schematica di una ricerca bidirezionale che avrà successo quando un ramo del nodo START incontra un ramo del nodo GOAL.

2.4.7 Confronto

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figura 2.20: Valutazioni delle strategie tree-search. b è il branching factor; d è la profondità della soluzione più superficiale; m è la profondità massima dell'albero di ricerca; l è il depth-limit. Gli apici rappresentano: [a] completo se b è finito; [b] completo se lo *step cost* $\geq \epsilon$; [c] ottimale se gli step cost sono tutti identici; [d] se entrambe le direzioni utilizzano la breadth-first search.

2.5 Informed Search Strategies (Heuristic)

Questo tipo di ricerca sfrutta la conoscenza specifica del dominio applicativo per fornire suggerimenti su dove si potrebbe trovare l'obiettivo. I suggerimenti hanno la forma di una *funzione euristica*, denotata con $h(n)$: *costo del cammino più economico dallo stato del nodo n ad uno stato obiettivo.*

Se n è il nodo obiettivo, allora $h(n) = 0$.

2.5.1 Best-first search

É un'istanza della tree-search o della graph-search nella quale un nodo è selezionato per espandersi basandoci su una *funzione di valutazione* $f(n)$. Viene espanso il nodo avente il valore minore di questa funzione.

2.5.2 Greedy best-first search

La ricerca "golosa" considera $f(n) = h(n)$ per espandere i nodi, espande prima il nodo che appare più vicino all'obiettivo (sulla base del fatto che questo con maggiore probabilità porti prima ad una soluzione).

Arad	366	**Mehadia**	241
Bucarest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figura 2.21: Valori di h_{DLA}, Distanze in Linea d'Aria verso Bucarest.

Consideriamo il problema dell'itinerario in Romania con l'utilizzo della *staight-line distance* (distanza in linea d'aria). Il primo nodo espanso da Arad sarà Sibiu, perché l'euristica afferma che è più vicino a Bucarest di Zerind e Timisoara. Il nodo successivo sarà Fagaras, perché ora è il più vicino secondo l'euristica. Fagaras a sua volta genererà Bucarest, che è l'obiettivo.

La soluzione trovata in questo caso però ha un cammino di 32 chilometri più lungo rispetto quello ottimo. Questo algoritmo "avido" ad ogni passo cerca di arrivare il più vicino possibile ad un obiettivo ma questa golosità lo porta a risultati peggiori di quelli ottenibili con maggiore cautela.

La ricerca best-first greedy è **incompleta** negli spazi degli stati infiniti ed in quelli finiti senza il controllo degli stati già percorsi, nella versione tree. É invece **completa** negli spazi degli stati finiti per la versione graph.

La **complessità spaziale** e quella **temporale** (nella tree search) sono $O(b^m)$ nel caso peggiore, con m la massima profondità dello spazio di ricerca. Con una buona funzione euristica questa complessità può essere ridotta notevolmente.

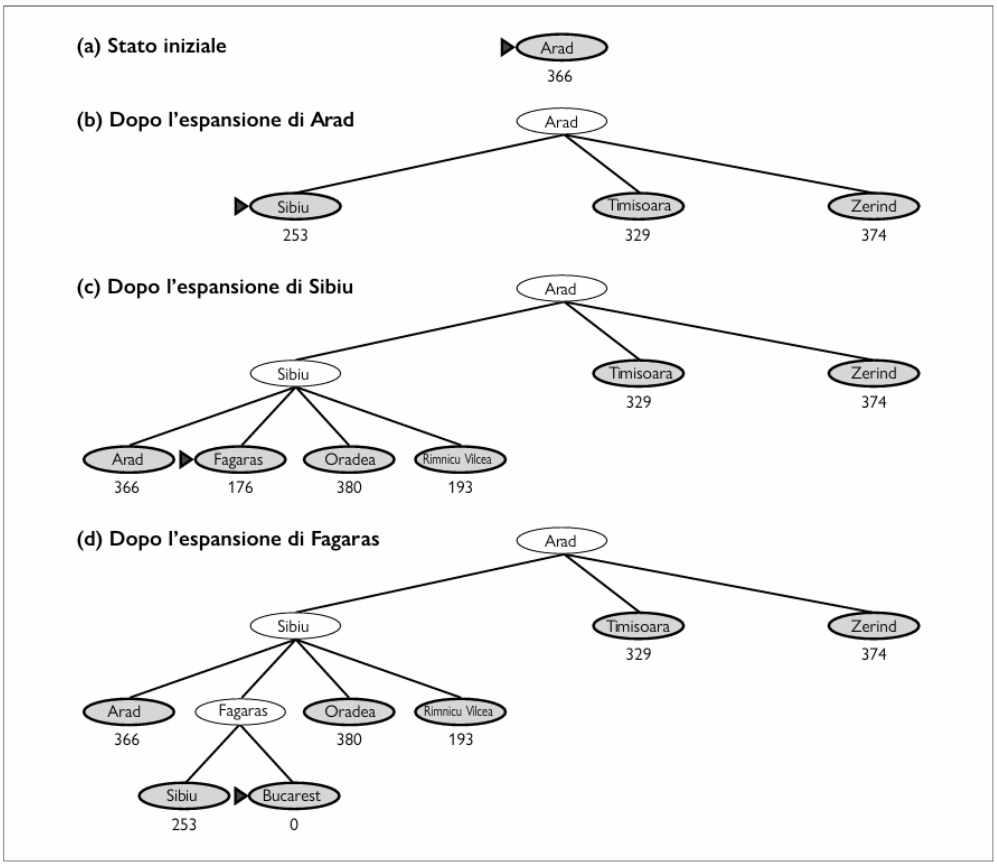

Figura 2.22: Passi di una ricerca (ad albero) best-first greedy di un itineriario verso Bucarest che usa l'euristica della distanza in linea d'aria. I nodi sono etichettati con i loro valori h.

L'algoritmo **non** è **ottimale** perché tiene conto solo dell'euristica e non del costo del cammino.

2.5.3 A* search

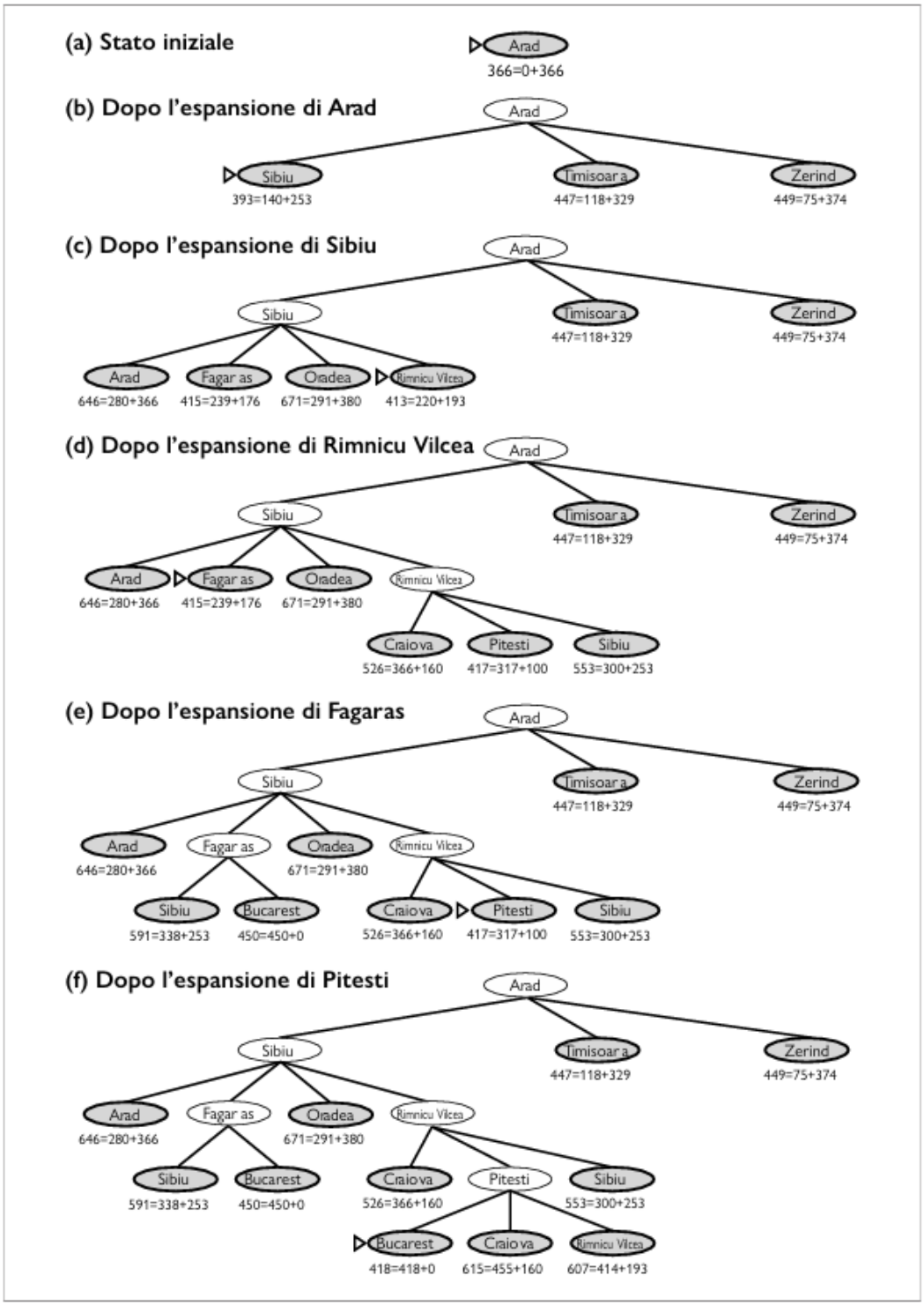

Figura 2.23: Passi di una ricerca A* di un itinerario verso Bucarest. I nodi sono etichettati con i valori $f = g + h$. I valori h sono distanze in linea d'aria verso Bucarest.

Questa ricerca, pronunciata *A-star* search, valuta i nodi combinando il costo per raggiungere il nodo, $g(n)$, ed il costo per andare dal nodo all'obiettivo, $h(n)$: $f(n) = g(n) + h(n)$; che rappresenta il costo stimato del cammino migliore che porta alla soluzione attraverso n.

La ricerca A* è completa. Una proprietà fondamentale è l'**ammissibilità**. Un'euristica è ammissibile se non *sovrastima mai* il costo per raggiungere un obiettivo, è quindi *ottima* rispetto al costo. Possiamo dimostrare ciò per assurdo, supponendo che il cammino ottimo abbia costo C^* ma l'algoritmo restituisca un cammino di costo maggiore ($C > C^*$); allora deve esistere un nodo n che si trova sul cammino ottimo che non è espanso (perché se tutti i nodi fossero stati espansi sarebbe stata restituita la soluzione ottima).

Per l'ammissibilità deve accadere che:

$$h(n) \leq h^*(n)$$

Dove $h^*(n)$ è il costo del cammino ottimo (vero) da n, fino al più vicino obiettivo.

Un'ulteriore proprietà è la **consistenza**. Un'euristica è consistente se, per ogni nodo n ed ogni suo successore n' (generato da un'azione a), abbiamo:

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

Questa è una forma generale della **triangle inequality**, per la quale ogni lato di un triangolo non può essere più lungo della somma degli altri due.

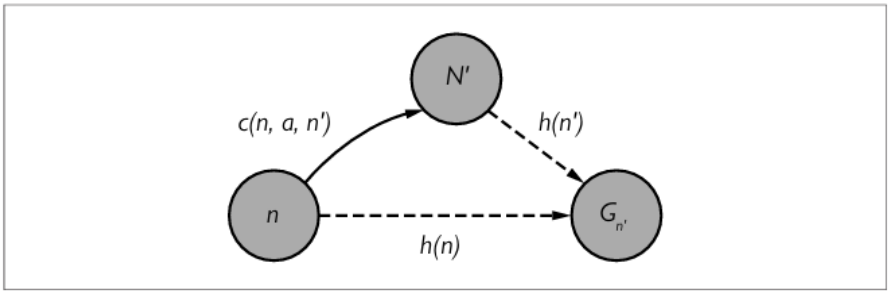

Figura 2.24: *Disuguaglianza triangolare*: se l'euristica h è *consistente*, allora il singolo numero h(n) sarà minore della somma del costo c(n,an') dell'azione per andare da n a n' e della stima euristica h(n').

Se h è consistente, quindi abbiamo:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

Ogni euristica consistente è ammissibile (ma non vale il vice versa).

Properties of A*

Riassumento, l'algoritmo è **completo** in spazi di stato finiti, quindi se non c'è un numero infinito di nodi con $f \leq f(G)$ (dove G è un generico Goal).

Se C* è il costo del cammino della soluzione ottima, possiamo affermare che:

- A* espande tutti i nodi che possono essere raggiunti dallo stato iniziale su un cammino in cui per ogni nodo si ha $f(n) < C^*$ (chiamiamo questi nodi: *certamente espansi*);
- A* potrebbe espandere alcuni dei nodi sul confine obiettivo ($f(n) = C^*$) prima di selezionare un nodo obiettivo;
- A* non espande nessun nodo avente $f(n) > C^*$.

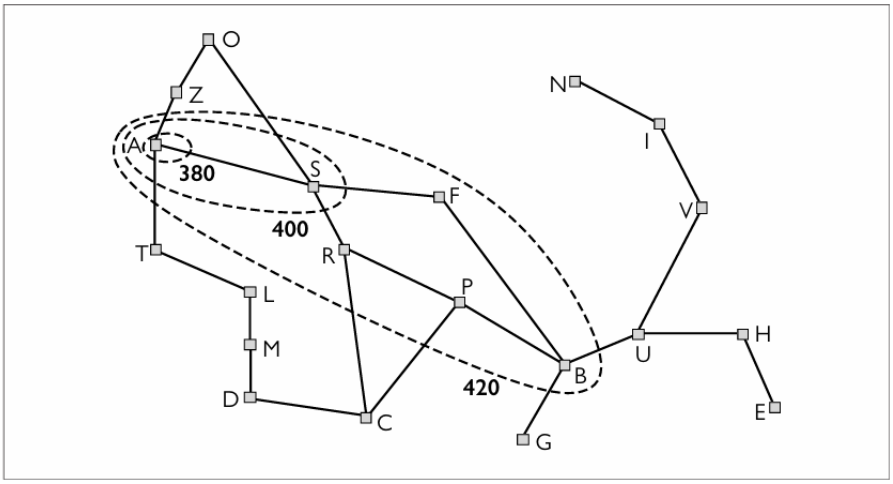

Figura 2.25: Mappa della Romania che indica i confini per $f = 380, f = 400, f = 420$ con Arad come stato iniziale. I nodi all'interno di un dato confine hanno tutti un costo $f = g + h$ minore o uguale al valore del confine. [Le città sono indicate con la lettere iniziale del loro nome per motivi di spazio].

2.5.4 Admissible Heuristic Functions

Per il rompicapo ad 8 tasselli le possibili euristiche sono:

- h_1 = il numero di tasselli fuori posto (spazi vuoti non inclusi);
- h_2 = la somma delle distanze di tutti i tasselli dalla loro posizione corrente a quella nella configurazione obiettivo. Dato che i tasselli non possono muoversi in diagonale, la distanza è la somma delle distanze in orizzontale e verticale (**distanza Manhattan** o anche detta *distanza tra isolati*).

Nell'esempio in Figura 2.26 $h_1 = 8$ ed $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$.

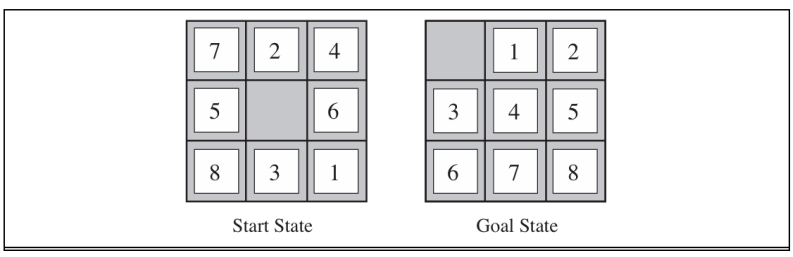

Figura 2.26: Una tipica istanza del 8-puzzle. La soluzione è lunga 26 steps.

Come speravamo, nessuna di queste euristiche ha sovrastimato il vero costo della soluzione (che è 26).

Quando un'euristica ha in un nodo valore maggiore di un'altra, diciamo che essa **domina** nel nodo l'altra. Quando questo accade per tutti i nodi, l'euristica che domina l'altra è migliore ai fini della ricerca.

Si punta a scegliere l'euristica dominante. Se dovesse accadere che per alcuni nodi è dominante un'euristica e per altri un'altra, verrà scelta una funzione a tratti composta dalle euristiche migliori in ogni nodo:
$h(n) = \max(h_a(n), h_b(n))$.

2.5.5 Relaxed Problem

Un problema con meno restrizioni sulle azioni possibili è detto *problema rilassato*. Qualsiasi soluzione ottima del problema originale è anche soluzione del problema rilassato. Quindi, *il costo di una soluzione ottima di un problema rilassato è un'euristica ammissibile per il problema originale.*

Bell'esempio precedente quindi: h_1 è la soluzione migliore se un tassello può muoversi ovunque, mentre h_2 è scelta se è possibile muovere un tassello solo negli spazi adiacenti.

2.5.6 Ricerca con memoria limitata

Algoritmo **IDA*** (Iterative-Deepening A*) è il modo più semplice per ridurre i requisiti di memoria. La principale differenza tra questo algoritmo è l'iterative deepening standard è il valore utilizzato come livello di arresto. Nell'IDA* viene usato il valore di $f = g + h$ e non la profondità del nodo.

Il **RBFS** (Recursive Best-First Search) è un semplice algoritmo ricorsivo che emula la best-first search usando però solamente uno spazio lineare.

Ogni volta che viene selezionato un nodo, in esso salviamo anche le alternativa migliore successiva. Procediamo ad espandere tenendo sotto controllo il valore salvato; quando questo è inferiore a quelli raggiunti sulla frontiera bisogna fare backtracking (cancellando tutta la parte espansa e salvando in memoria il nuovo valore di f raggiunto).

Nella figura:

- a) Seguendo il cammino che passa da Rimnicu Vilcea finché la foglia migliore corrente (Pitesti) non ha un valore peggiore del migliore cammino alternativo (Fagaras).

- b) Si "torna su" (ricorsione) ed il valore della foglia migliore del sottoalbero dimenticato (417) è salvato in Rimnicu Vilcea; quindi viene espanso Fagras, rivelando che il valore della foglia migliore è in realtà 450.

- c) Torniamo ancora una volta indietro ed il valore della foglie migliore del sottoalbero dimenticato (450) è salvato in Fagaras; quindi si ri-espande Rimnicu Vilcea e l'espansione continua fino a Bucarest, dato che il miglior cammino alternativo (quello passante per Timisoara) cosa almeno 447.

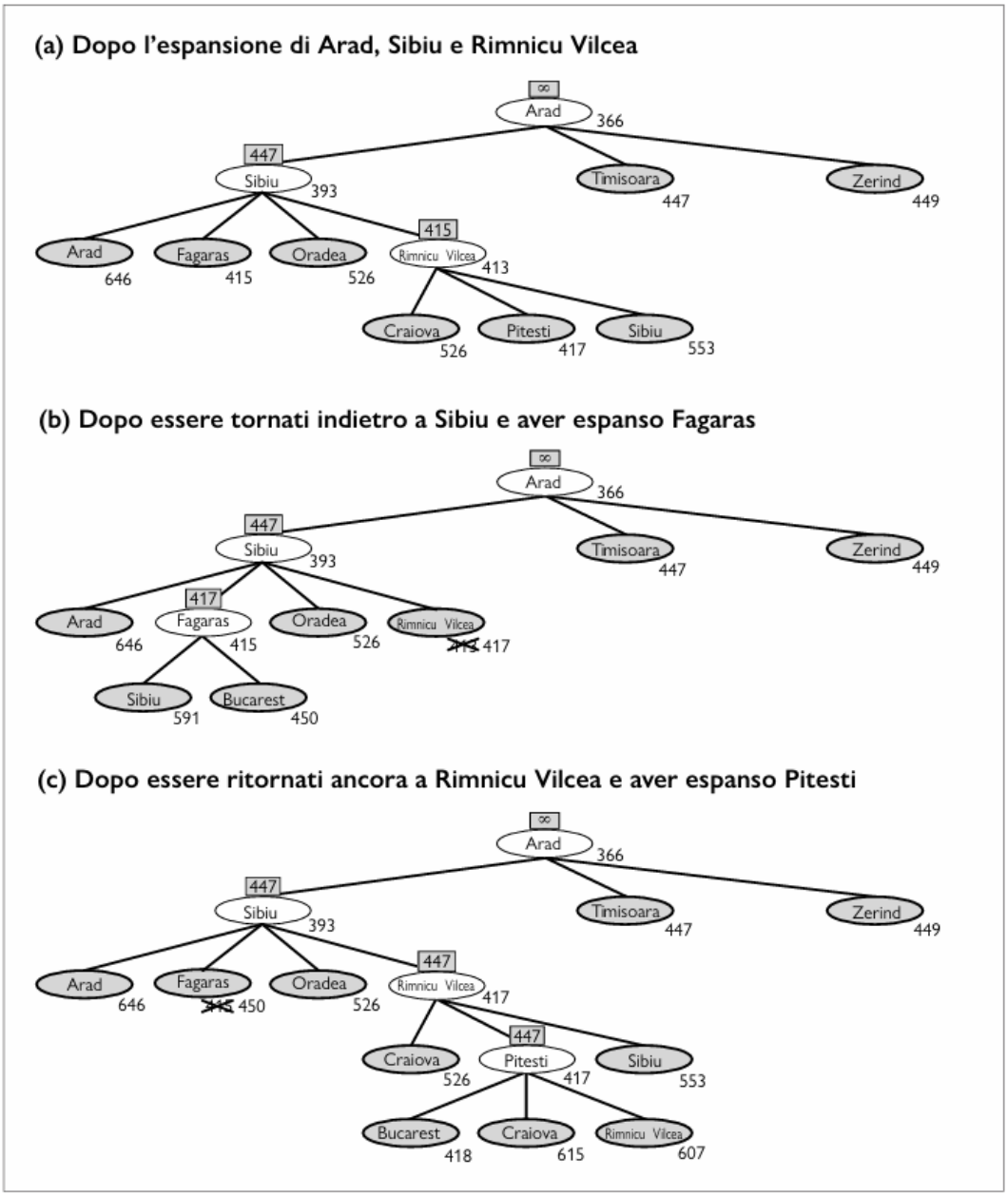

Figura 2.27: Fasi di una ricerca RBFS per trovare l'itinerario più breve verso Bucarest. Il limite ai valori di ogni chiamata ricorsiva è indicato in un box sopra al nodo corrente, ed ogni nodo è etichetato con il suo costo.

Il **SMA*** (Simplified Memory-bounded A*), invece, consiste nel procedere come per l'A* ad espandere la foglia migliore fino a quando la memoria non è esaurita. Dopo di che viene scartato dalla frontiera il nodo avente il peggior valore di f, poiché si suppone che esso non appartenga al cammino ottimo. Viene, comunque, salvato il valore del nodo scartato nel suo relativo nodo padre, nel caso in cui bisogna fare backtracking. Se due o più nodi hanno lo stesso valore di f verrà cancellato il nodo più vecchio.

Capitolo 3

Beyond Classical Search

3.1 Local Search Algorithms

Gli algoritmi di *ricerca locale* operano cercando a partire da uno stato iniziale e procedendo verso gli stati adiacenti senza tener traccia dei cammini, né degli stati già raggiunti. Questo significa che non sono sistematici (potrebbero non esplorare mai una porzione dello spazio degli stati, dove effettivamente risiede una soluzione). I principali vantaggi di questa ricerca sono: la poca memoria utilizzata ed il fatto che riescono a trovare soluzioni ragionevoli anche in spazi degli stati infiniti.

In questo tipo di problemi non siamo interessati alla sequenza di stati che attraversiamo per arrivare all'obiettivo, ma siamo solo interessati alla configurazione finale; quindi è possibile usare come metodo di risoluzione sia la ricerca informata che quella non informata.

3.1.1 Hill-climbing search

L'algoritmo tiene traccia solo dello stato corrente e ad ogni iterazione passa allo stato vicino con valore più alto, cioè punta nella direzione che presenta l'ascesa più ripida (*steppest ascent*), senza guardare oltre gli stati immediatamente vicini a quello corrente.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                   neighbor, a node
  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
  end
```

Figura 3.1: Algoritmo di ricerca hill climbing. Rappresenta la tecnica più semplice di ricerca locale; ad ogni passo il nodo corrente viene rimpiazzato dal miglior vicino.

Questo algoritmo viene talvolta chiamato *greedy local search* perché sceglie uno stato vicino "buono" senza pensare a come andrà avanti. Può capitare che l'hill climbng si blocchi a causa dei seguenti motivi:

- **Local maxima**: un massimo locale è un picco più alto degli stati vicini, ma inferiore al massimo globale.
- **Ridges**: una cresta dà origine ad una sequenza di massimi locali, molto difficili da esplorare da parte degli algoritmi greedy.
- **Plateaux**: è un'area piatta del panorama dello spazio degli stati. Può essere un massimo locale piatto, da cui non è possibile fare ulteriori progressi, oppure una *spalla* (**shoulder**), da cui si potrà salire ulteriormente.

In ognuno di questi casi l'algoritmo raggiunge un punto dal quale non riesce a compiere ulteriori progressi.

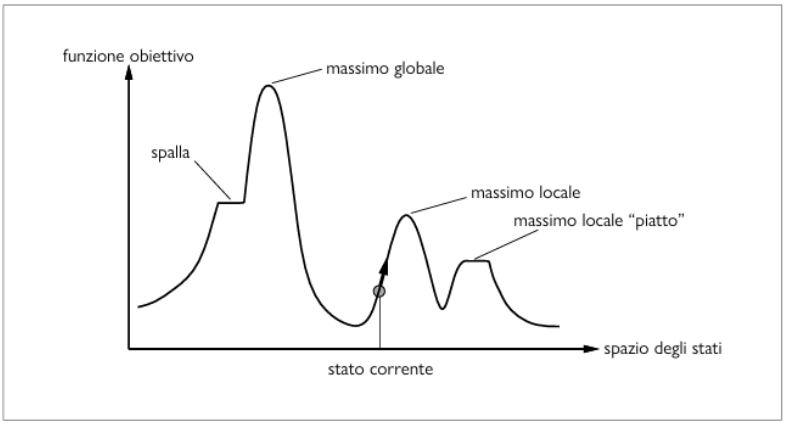

Figura 3.2: Panorama dello spazio degli stati mono-dimensionale, in cui l'altezza corrisponde alla funzione obiettivo. Lo scopo è trovare il massimo globale.

3.1.2 Simulated annealing

Un algoritmo che combina l'hill climbing con un'esplorazione casuale in modo tale da ottenere sia efficienza che completezza.

L'idea alla base consiste nel permettere alcune mosse "errate" ma decrementandone gradualmente la dimensione e la frequenza.

La probabilità concessa per compiere mosse errate è:

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

Il comportamento dell'algoritmo è simile alla *tempra* in metallurgia (il processo usato per indurire i metalli riscaldandoli ad altissime temperature e poi raffreddandoli gradualmente).

Per la distribuzione di Boltzman, se T decresce con sufficiente lentezza viene sempre raggiunto lo stato ottimo:

$$\frac{e^{\frac{E(x^*)}{kT}}}{e^{\frac{E(x)}{kT}}} = e^{\frac{E(x^*)-E(x)}{kT}} >> 1 \text{ (per T piccole)}$$

Questo algoritmo è ampiamente usato per risolvere problemi di configurazione VLSI, a partire dagli anni 1980.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps
  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```

Figura 3.3: Una versione stocastica dell'hill climbing dove sono permesse anche mosse verso il basso. L'input *schedule* determina il valore della temperatura T in funzione del tempo.

3.1.3 Local beam search

Questo algoritmo tiene traccia di k stati anziché uno solo; scegliendo i k miglior successori.

L'informazione viene passata da un thread di ricerca parallelo all'altro.

Lo stato che genera i migliori successori attira gli altri, quindi vengono abbandonate le ricerche aventi meno progressi.

Potrebbe capitare che un thread conduca tutta la ricerca, per risolvere questo problema si scelgono tra i successori: $k - n$ migliori con n casuali non migliori. In questo modo si possono esplorare situazioni che localmente appaiono non buone ma potrebbero condurre all'ottimo globale,

3.1.4 Genetic algorithms

Questi algoritmi sono una variante della beam search stocastica in cui ogni stato successore è generato combinando *due* stati genitori invece che modificare un singolo stato.

L'algoritmo genetico inizia con un set di k stati generati casualmente, chiamati *popolazione*. Ogni stato (o *individuo*) è rappresentato da una stringa. Nell'esempio delle 8 regine sono necessari $8 \times log_2 8 = 24bit$. Altrimenti lo stato potrebbe essere rappresentato da 8 digits (ognuno nel range da 1 ad 8).

Ogni stato è valutato da una *fitness function* (una funzione oggettiva che assegna valore maggiore ai migliori stati). Per l'8-queens problem usiamo il numero di coppie non attaccate.

Due coppie vengono scelte casualmente per la riproduzione, in base alle probabilità. Un punto di *crossover* è scelto casualmente nella stringa.

I successori sono creati ricombinando le parti separate dai punti di incrocio: la prima parte appartiene al genitore 1 e la seconda al genitore 2 e così via.

Ogni prole potrebbe essere soggetta a *mutazioni* casuali. Nella Figura 3.4 viene mutato un digit nel primo, terzo e quarto successore.

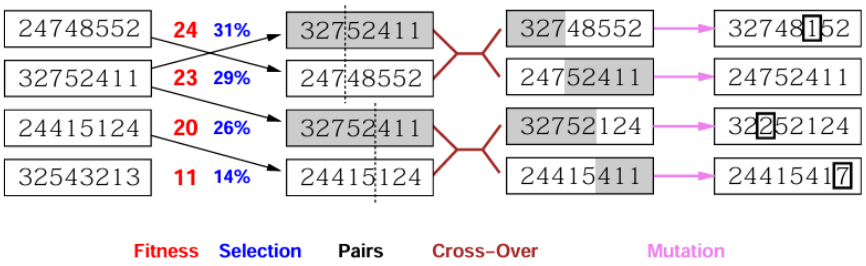

Figura 3.4: Un algoritmo genetico, illustrato con stringhe di digit rappresentanti gli stati dell'8-queens problem. La popolazione iniziale è ordinata in base alla funzione di finess, dando come risultato le coppie. Queste poi danno origine a dei discendenti che saranno soggetti a mutazione.

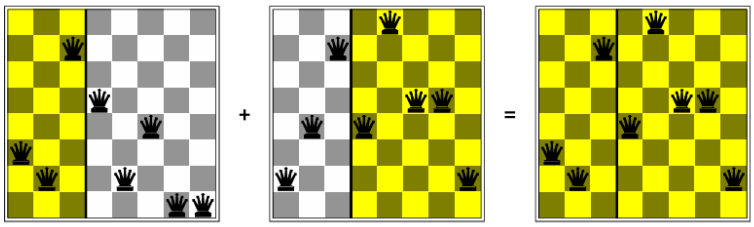

Figura 3.5: Gli stati del problema delle 8 regine corrispondenti ai primi due genitori (dei Paiirs) ed al primo stato della prole (dopo il Cross-Over). Le colonne evidenziate in giallo verranno mantenute, quelle normali saranno perse nel passo di crossover.

3.2 Local Search in Continuous Spaces

Supponiamo d voler costruire tre nuovi aeroporti in Romania, in posizioni tali da minimizzare la somma dei quadrati delle distanze in linea d'aria da ogni città sulla mappa all'aeroporto più vicino. Lo spazio degli stati sarà quindi definito dalle coordinate degli aeroporti: $(x_1, y_1), (x_2, y_2), (x_3, y_3)$. Questo spazio ha sei dimensioni, quindi gli stati sono definiti da 6 *variabili*.

Muoversi in questo spazio significa spostare uno o più aeroporti sulla mappa. La funzione obiettivo si calcola come:

$$f(x) = f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

Dove C_i rappresenta l'insieme delle città il cui aeroporto più vicino (nello stato x) è i. Un modo per affrontare uno spazio degli stati continuo consiste nella *discretizzazione* di esso. Anziché permettere alle posizioni (x, y) di essere

un qualsiasi punto in uno spazio bidimensionale continuo, potremmo limitarle punti fissi su una grigli rettangolare con spaziatura di lunghezza δ. Quindi, al posto di avere un numero infinito di successori, ogni stato ne avrebbe soltanto 12 (corrispondenti all'incremento di una delle sei variabili di un valore $\pm\delta$).

I metodi che misurano il progresso compiuto in base alla variazione della funzione obiettivo tra due punti vicini si chiamano metodi del *gradiente empirico*.

Molti metodi cercano di usare il *gradiente* del panorama per trovare un massimo.

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right)$$

In alcuni casi, è possibile trovare un massimo risolvendo: $\nabla f = 0$. Questo è il caso in cui abbiamo un solo aeroporto. Con tre aeroporti, l'espressione del gradiente dipende dalle città che sono più vicine ad ogni aeroporto nello stato corrente, quindi possiamo calcolare il gradiente localmente (e non globalmente).

Per molti problemi, l'algoritmo più efficace rimane il metodo di Newton-Raphson:

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

Dove $H_f(x)$ è la matrice *Hessiana* delle derivate seconde, i cui elementi sono dati da: $H_{ij} = \partial^2 f / \partial x_i \partial x_j$

Capitolo 4

Constraint Satisfaction Problems (CSPs)

Un problema di soddisfacimento di vincoli è costituito da tre componenti:

- $\mathcal{X}$ insieme di variabili;
- $\mathcal{D}$ insieme di domini;
- $\mathcal{C}$ insieme di vincoli che specificano le combinazioni di valori ammesse.

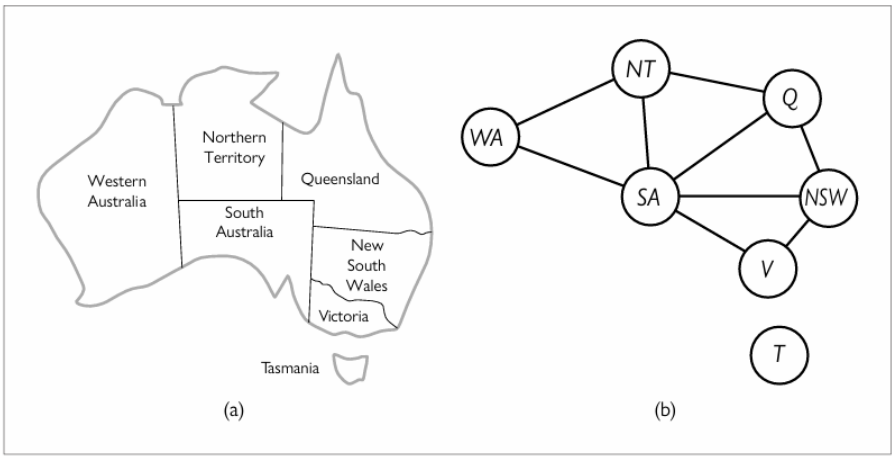

Figura 4.1: I principali stati e territori dell'Australia. La colorazione di questa mappa può essere vista come un CSP. L'obiettivo è assegnare un colore ad ogni regione in modo che non esistano due regioni adiacenti con lo stesso colore. A destra notiamo una rappresentazione del problema sotto forma di constraint graph.

Per formulare il problema in Figura 4.1, definiamo una variabile per ogni regione:

$$\mathcal{X} = \{WA, NT, Q, NSW, V, SA, T\}$$

Il dominio di ogni variabile è l'insieme:

$$D_i = \{rosso, verde, blu\}$$

I vincoli richiedono che le regioni adiacenti abbiano colori distinti, quindi avremmo nove vincoli:

$$\mathcal{C} = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V,$$

$$WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$$

Che può essere anche enumerato come:

$$\{(rosso, verde), (rosso, blu), (verde, rosso), (verde, blu), (blu, rosso), (blu, verde)\}$$

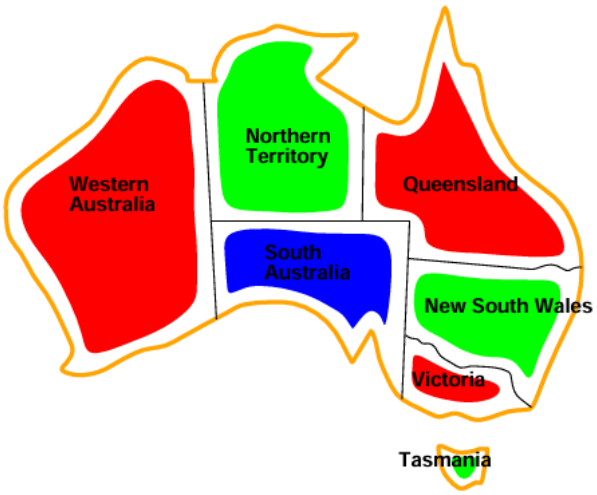

Figura 4.2: Una possibile soluzione.

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

Viene visualizzato un CSP come *grafo di vincoli* per velocizzare la soluzione.

Se un compito T_1 deve essere svolto prima del compito T_2 è necessario aggiungere la durata necessaria affinché il primo compito sia terminato. I vincoli di *precedenza* quindi sono rappresentabili così: $T_1 + d_1 \leq T_2$.

4.1 Varianti del formalismo CSP

Una soluzione esiste solo per i *linear constraints*; non esistono algoritmi in grado di risolvere *nonlinear constraints* generali su variabili intere.

L'**unary constraint** in cui il vincolo include una sola variabile. Ad esempio nel map-coloring problem potrebbe essere il caso in cui $SA \neq green$.

Il **binary constraint** mette in relazione due variabili. Ad esempio $SA \neq NSW$.

Un vincolo che interessa un numero arbitrario di variabili si dice **global constraint**. Uno dei più comuni vincoli globali è *alldiff* che afferma che tutte le variabili interessate dal vincolo devono avere valori diversi. Un altro

esempio è la *criptoaritmetica* (ogni lettera corrisponde ad una cifra diversa), si potrebbe scrivere un vincolo a sei variabili $Alldiff(F, T, U, W, R, O)$. I vincoli imposti dall'aritmetica dell'addizione sulle quattro colonne del riompicapo possono essere scritti come:

$$O + O = R + 10 \cdot C_1$$

$$C_1 + W + W = U + 10 \cdot C_2$$

$$C_2 + T + T = O + 10 \cdot C_3$$

$$C_3 = F$$

Ogni vincolo a dominio finito può essere ridotto ad un insieme di valori binari se si introduce un numero sufficiente di variabili ausiliarie. Quindi qualsiasi CSP può essere trasformato in un problema avente solo vincoli binari.

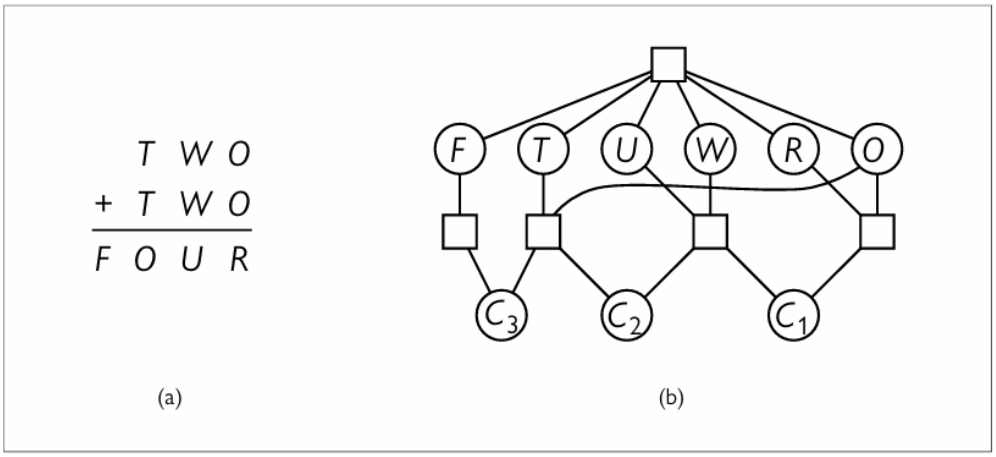

Figura 4.3: Un problema di cripto-aritmetica. Ogni lettera indica una singola cifra distinta; lo scopo è trovare una corrispondenza tra lettere e cifre tale che la somma risulti aritmeticamente corretta, con il vincolo aggiuntivo che non sono permessi zeri alla sinistra dei numeri. L'ipergrafo dei vincoli per il problema in questione è mostrato a destra. Le variabili C_1, C_2, C_3 rappresentano le cifre di riporto per le tre colonne.

4.2 Propagazione dei vincoli: Inference in CSP

Un algoritmo di ricerca procede in un solo modo: espande un nodo per visitare i successori. Un algoritmo CSP, invece, ha delle scelte: può generare successori scegliendo un nuovo assegnamento di variabile, oppure può effettuare un tipo specifico di *inferenza* denominato **constraint propagation** (utilizzando i vincoli per ridurre il numero di valori legali per una variabile, che potrebbe ridurre i valori legali per un'altra variabile e così via).

4.2.1 Node consistency

Una singola variabile (corrispondente ad un nodo nel grafo CSP) è *nodo-consistente* se tutti i valri del suo dominio soddisfano i suoi vincoli unari.

Per esempio, nel problema di colorazione della mappa dell'Australia, dove gli australiani del sud non amano il verde: la variabile SA inizia con dominio $[rosso, verde, blu]$ e possiamo renderla nodo-consistente eliminando il *verde*; lasciando quindi il dominio ridotto $[rosso, blu]$.

4.2.2 Arc consistency

Una varibile in un CSP si dice *arco-consistente* se ogni valore del suo dominio soddisfa i suoi vincoli binari. Quindi, X_i è arco-consistente rispetto a X_j se per ogni valore nel dominio corrente D_i c'è un valore nel dominio D_j che soddisfa il vincolo binario sull'arco (X_i, X_j).

Un grafo è arco-consistente se ogni variabile è arco-consistente con ogni altra. Nel problema della colorazione della mappa dell'Australia non serve a nulla la consistenza d'arco.

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
      for each Xk in NEIGHBORS[Xi] do
        add (Xk, Xi) to queue

---------------------------------------------------------------------

function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[Xi] do
    if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
      then delete x from DOMAIN[Xi]; removed ← true
  return removed
```

Figura 4.4: L'algoritmo AC-3 per il controllo della consistenza d'arco (chiamato così dal suo inventore perché era la terza versione presentata nell'articolo). Dopo l'applicazione di esso ogni arco è consistente, oppure qualche variabile avrà un dominio vuoto (il che significa che il CSP non può essere risolto).

Inizialmente abbiamo una coda che contiene tutti gli archi del CSP. AC-3 estrae un arco arbitrario (X_i, X_j) dalla coda, rendendo X_i arco-consistente rispetto a X_j. Se questo lascia invariato D_i allora l'algoritmo passa all'arco successivo; se invece D_i si riduce, aggiungiamo alla coda tutti gli archi (X_k, X_i) dove X_k è adiacente a X_i. Il cambiamento in D_i potrebbe consentire ulteriori riduzioni in D_k. Se D_i è ridotto all'insieme vuoto, sappiamo che l'intero CSP non ha soluzione consistente e AC-3 restituisce subito il fallimento. Altrimenti continuiamo a controllare, cercando di rimuovere valori dai domini delle variabili finché non vi sono più archi nella coda. A questo punti rimane un CSP

che è equivalente all'originale (ha le stesse soluzioni), ma avrà una ricerca più rapida, perché le variabili hanno domini più piccoli.

Supponendo di avere un CSP con n variabili, ognuna con dimensione del dominio non superiore a d e con c vincoli binari (archi). Ogni arco (X_k, X_i) può essere inerito nella coda soltanto d volte perché X_i ha al più d valori che si possono eliminare. Il controllo della consistenza di un arco può essere svolto in un tempo $O(d^2)$, perciò otteniamo un tempo totale del caso peggiore $O(cd^3)$.

4.2.3 Path consistency

Se volessimo colorare la mappa dell'Australia utilizzando solo due colori: rosso e blu; la consistenza d'arco non servirebbe a nulla, perché ogni variabile è già arco-consistente.

La *consistenza di cammino* restringe i vincoli binari utilizzando vincoli impliciti che sono inferiti considerando triplette di variabili.

Un insieme di due variabili $\{X_i, X_j\}$ è cammino-consistente rispetto ad una terza variabile X_m se, per ogni assegnamento $\{X_i = a, X_j = b\}$ esiste un assegnamento di X_m che soddisfi i vincoli su $\{X_i, X_m\}$ e $\{X_m, X_j\}$.

Il problema della colorazione della mappa dell'Australia utilizzando solo due colori è comunque impossibile.

4.2.4 K-consistency

Un CSP è *k-consistente* se, per ogni insieme di $k - 1$ variabili e ogni loro assegnamento consistente, è sempre possibile assegnare un valore consistente ad ogni k-esima variabile. La 1-consistenza significa che, dato l'insieme vuoto, possiamo rendere consistente ogni insieme di una sola variabile (è il caso della consistenza di nodo). La 2-consistenza corrisponde alla consistenza d'arco. Per grafi con vincoli binari, la 3-consistenza corrisponde alla consistenza di cammino.

Un CSP è *fortemente* k-consistente, se è k-consistente ed anche (k-1)-consistente, (k-2)-consistente,..., fino a 1-consistente. Il tempo di esecuzione totale è $O(n^2 d)$, con n numero di nodi e d valori del dominio.

4.3 Backtracking search

La ricerca con backtraking è una depth-first search che sceglie i valori una variabile alla volta e tiene conto di quando una variabile non ha più valori legali da assegnare. L'algoritmo sceglie ripetutamente le variabili non assegnate e cerca, uno ad uno, tutti i valori nel suo dominio, cercando una soluzione.

La ricerca con backtracking può essere migliorata usando euristiche *indipendenti dal dominio* che traggono vantaggio dalla rappresentazione fattorizzata dei CSP, a differenza degli algoritmi di ricerca non informata che potevano essere migliorati soltanto fornendo funzioni euristiche specifiche del dominio.

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment given CONSTRAINTS[csp] then
          add {var = value} to assignment
          result ← RECURSIVE-BACKTRACKING(assignment, csp)
          if result ≠ failure then return result
          remove {var = value} from assignment
  return failure
```

Figura 4.5: Algoritmpo di backtracking per CSP. L'algoritmo è modellato sulla ricerca in profondità ricorsiva.

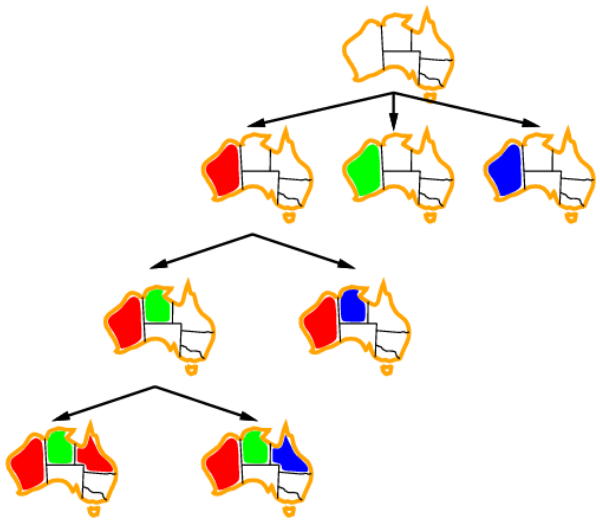

Figura 4.6: Una parte dell'albero di ricerca per il problema di colorazione della mappa.

4.3.1 Ordinamento di variabili e valori

La strategia più semplice per scegliere le variabili non assegnate è l'ordinamento statico (scegliere le variabili in ordine). La seconda strategia più semplice è quella di scegliere le variabili a caso. Nessuna delle due strategie è ottima.

Figura 4.7: Minimum Remaining Values (MRV).

L'idea di scegliere la variabile con il minor numero di valori legali è chiamata **euristica MRV**. Questa sceglie la variabile per cui si ha maggiore probabilità di arrivare presto ad un fallimento, potando così l'albero di ricerca. Se una variabile X non ha più alcun valore possibile, l'MRV sceglierà X ed il fallimento sarà rilevato immediatamente, evitando così ricerche inutili su altre variabili.

Solitamente ha prestazioni migliori di un ordinamento casuale o statico (anche se i risultati variano notevolmente a seconda del problema). Inoltre l'MRV non è di aiuto nella scelta della prima regione da colorare, perché all'inizio ognuna di esse ha esattamente tre colori legali.

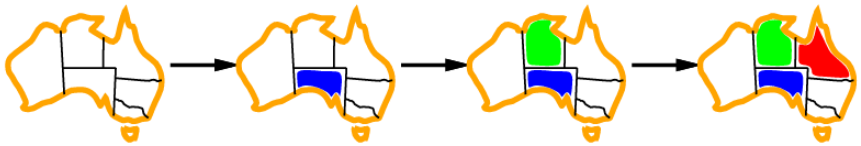

Figura 4.8: Degree heuristic.

L'**euristica di grado** cerca di ridurre il fattore di ramificazione delle scelte future scegliendo la variabile coinvolta nel maggior numero di vincoli con le altre variabili non assegnate. Nel problema della colorazione della mappa dell'Australia, SA è la variabile di grado più alto (5), le altre variabili hanno grado più piccolo (2 o 3, eccetto T che ha grado 0).

Se assegniamo prima SA, potremmo arrivare ad una soluzione senza passi falsi (backtracking).

L'MRV è solitamente più potente, quella di grado torna utile in caso di "pareggi" nell'ordinamento.

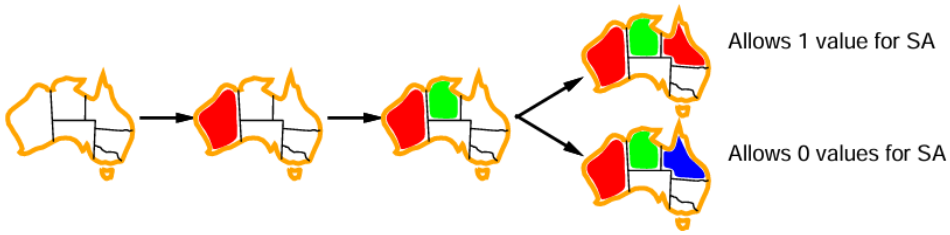

Figura 4.9: Least constraining value.

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

4.3.2 Interleaving search and Inference

Una delle più semplici forme di inferenza è la *verifica in avanti* (**forward checking**). Ogni volta che la variabile X è assegnata, il processo di verifica

in avanti stabilisce la consistenza d'arco per essa: per ogni va non assegnata Y, collegata ad X da un vincolo, cancella dal dominio di Y ogni valore non consistente con quello scelto per X.

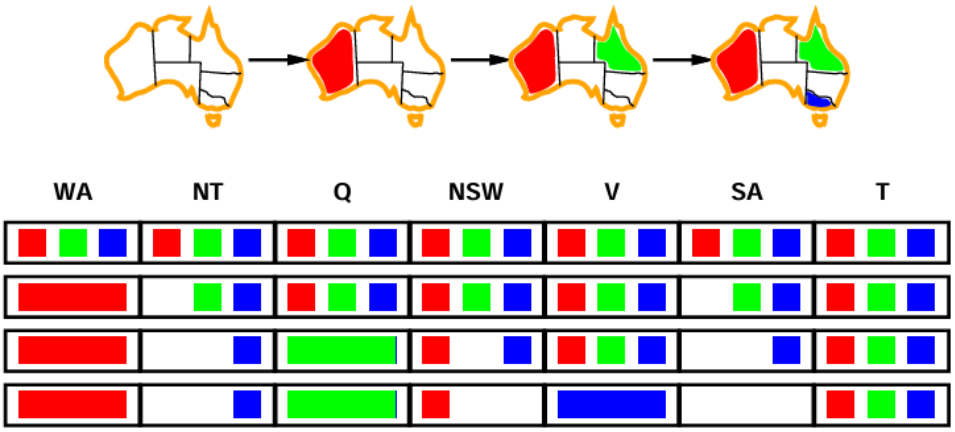

Figura 4.10: Il progresso di una ricerca per la colorazione di una moppa che sfruta la verifica in avanti. Prima di tutto si assegna WA = rosso; fatto questo, la verifica in avanti cancella il colore rosso dai domini delle variabili adiacenti NT ed SA. Dopo Q = verde, tale colore è rimosso dai domini di NT, SA e NSW. Dopo V = blu, anche blu è cancellato dai domini di NSW ed SA, dopodiché SA non ha più valori legali.

4.4 La struttura dei Problemi

Un grafo dei vincoli è un **albero** quando due variabili qualsiasi sono collegate da un solo cammino. Ogni CSP strutturato come albero può essere risolto in un tempo che cresce linearmente con il numero delle variabili.

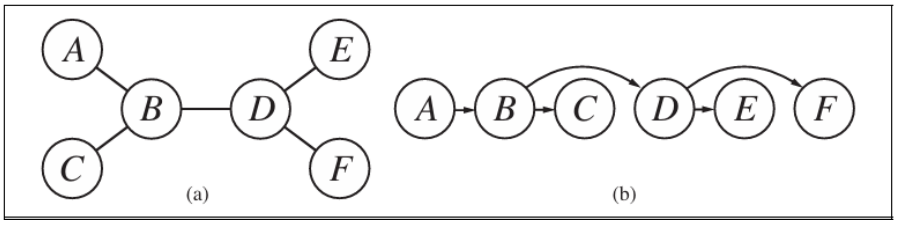

Figura 4.11: Grafo dei vincoli CSP con una struttura ad albero (a). Ordinamento lineare delle variabili consistente con l'albero, con A come radice (b).

Per risolvere un CSP con struttura ad albero si sceglie una variabile come radice dell'albero e si sceglie un ordinamento della variabili tale che ogni variabile appaia dopo il proprio genitore nell'albero. Un ordinamento di questo tipo prende il nome di *ordinamento topologico*.

Ogni albero con n nodi ha $n-1$ archi, perciò possiamo rendere questo grafo arco-direzionale-orientato in $O(nd^2)$.

Poiché ogni collegamento da un genitore al figlio è arco-consistente, sappiamo che per ogni valore scelto come genitore, ci sarà un valore valido da

scegliere come figlio. Quindi, non dovremmo ricorrere al backtracking, ma ci spostiamo linearmente tra le variabili.

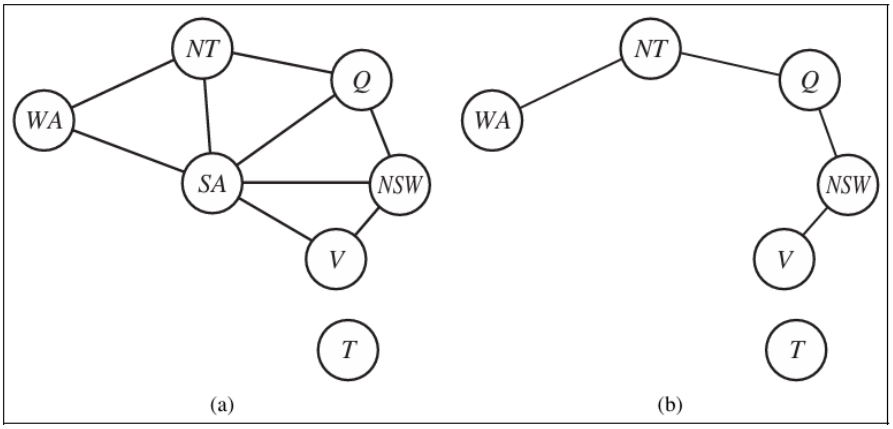

Figura 4.12: Grafo dei vincoli originario (a). Grafo dei vincoli dopo la rimozione di SA (b).

La mappa dell'Australia, senza l'Australia del Sud (SA) ha un grafo ad albero. Possiamo rimuoverla (sul grafo, non nel mondo reale) fissando un valore ad SA e cancellando dai domini delle altre variabili tutti i valori che sono inconsistenti con quello.

L'algoritmo generale risultante è:

1. scegliere un sottoinsieme S delle variabili del CSP tale che il grafo dei vincoli diventi un albero dopo la rimozione di S. questo prende il ome di *insieme di taglio dei cicli* (cycle cutset);

2. per ogni possibile assegnamento delle variabili in S che soddisfa tutti i vincoli su S:

   (a) rimuovere dal dominio delle variabili rimanenti tutti i valori non consistenti con gli assegnamenti in S,

   (b) se il CSP risultante ha una soluzione restituirla insieme all'assegnamento per S.

Se il cycle cutset ha dimensione c, il tempo di esecuzione totale è $O(d^c(n-c)d^2)$. Si prova ciascuna delle d^c combinazioni di valori delle variabili in S e per ognuna dobbiamo risolvere un problema su un albero di dimensione $n - c$. Se il grafo è quasi un albero c sarà piccolo.

4.5 Local Search for CSPs

Si procede generando uno stato qualsiasi (cioè un'assegnazione qualsiasi per tutte le variabili). Valutiamo quanto è "buona" questa assegnazione (attraverso la valutazione del numero di vincoli che l'assegnazione viola). Ci si muove

su gli altri stati possibili, andando alla ricerca di assegnazioni per cui i vincoli violati siano di meno (*min-conflicts*).

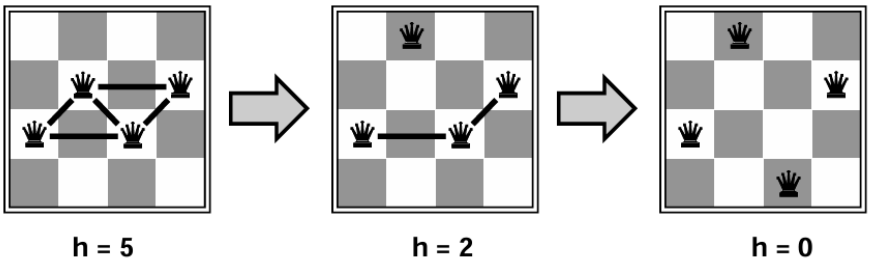

Figura 4.13: 4-Queens problem.

Nel problema delle 4-regine posso partire dal primo stato con 5 violazioni, cambio la posizione di una regina e riduco il numero dei conflitti. Procedo così fino ad azzerare il numero di violazioni. Questo funziona molto bene in problema di grandi dimensioni.

Con 4 regine in 4 colonne gli stati disponibili sono $4^4 = 256$. L'obiettivo è quello di non avere regine sulla stessa colonna o riga e confinanti diagonalmente. Le mosse possibili sono spostare le regine sulle colonne. La valutazione viene fatta tramite $h(n) =$ *numero di regine in conflitto.*

Capitolo 5

Adversarial Search

Gli ambienti *competitivi* in cui vi sono due o più agenti con obiettivi in conflitto danno origine alla **ricerca con avversari**.

5.1 Games

Nella teoria dei giochi avremo una soluzione sotto forma di strategia (una sequenza di azioni che ci porterà a vincere). Fatta una mossa, il giocatore avversario risponde e noi dobbiamo saper controbattere.

In questo ambito è importante cogliere i *limiti di tempo*. Distinguiamo i giochi in categorie.

	Deterministic	**Chance**
Perfect Information	chess, checkers, go, othello	backgammon, monopoly
Imperfect Information	battleships, blind tic-tac-toe	bridge, poker, scrabble, nuclear war

Tabella 5.1: Types of games.

Ipotesi operative:

- l'agente contro cui giochiamo è *razionale* (quindi in grado di sapere qual è la mossa migliore per lui);
- gioco a somma zero (se un giocatore vince (+1) l'altro perde (-1).

Con albero di gioco intendiamo un albero di ricerca che segue ogni sequenza di mosse fino a raggiungere uno stato terminale. Questo potrebbe essere infinito se lo spazio degli stati è illimitato o se le regole del gioco consentono di ripetere le posizioni all'infinito.

Per il gioco del tris l'albero è relativamente piccolo $9! = 362,880$ nodi terminali ($5,478$ stati distinti).

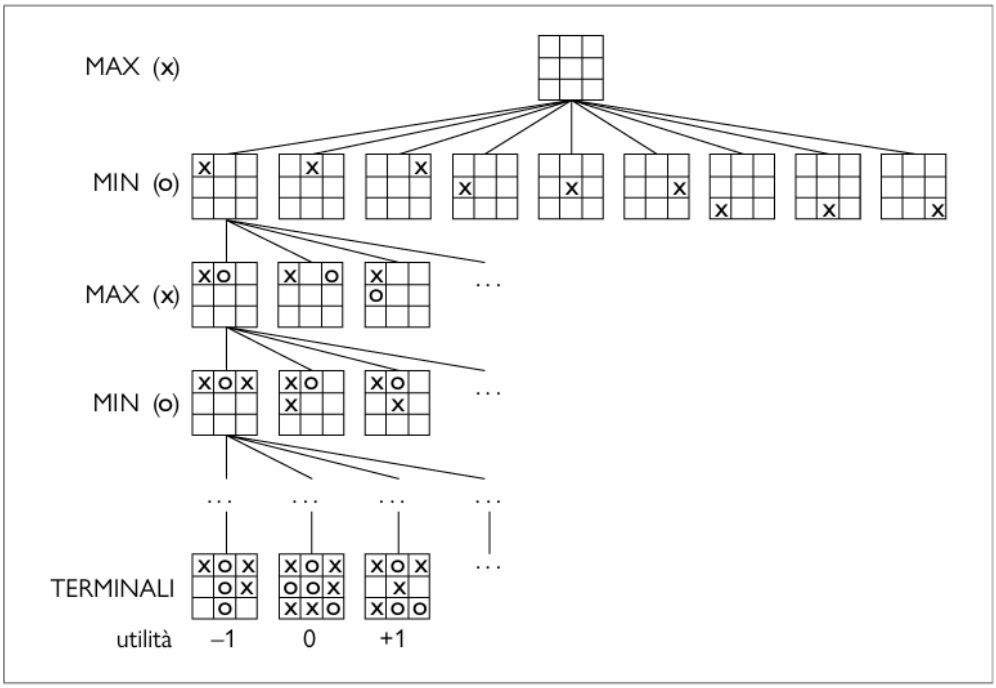

Figura 5.1: Un albero di gioco (parziale) per il gioco del tris (tic-tac-toe). Il nodo più in alto è lo stato iniziale, e MAX muove per primo, ponendo una X in un riquadro vuoto. É riportata una parte dell'albero, indicando le mosse alternate di MIN (che usa il simbolo O) e MAX, fino al raggiungimento degli stati terminali, a cui possono essere associati dei valori di utilità (in base alle regole del gioco). [I valori di utilità in figura sono associati a MAX].

5.2 Optimal Decision

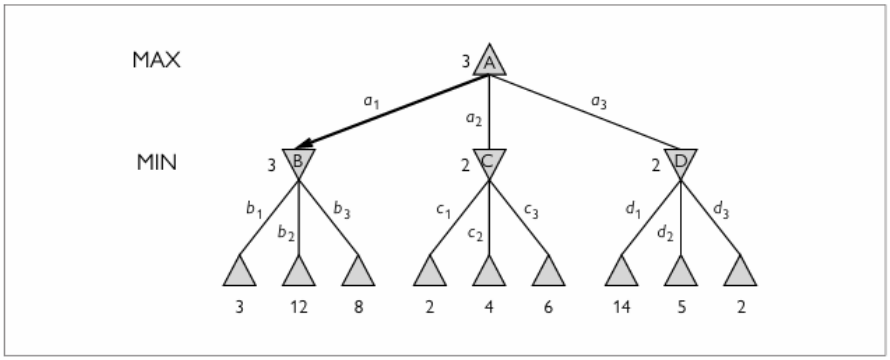

Figura 5.2: Albero di gioco a due strati. I nodi Δ sono "nodi MAX" (in cui è il turno di MAX a muovere), quelli ∇ sono "nodi MIN". I nodi terminali indicano i valori di utilità per MAX; gli altri sono etichettati con loro valori minimax. Alla radice la mossa migliore di MAX è a_1, perché porta al successore con il più alto valore, mentre la risposta migliore per MIN è b_1, perché porta al successore con valore più basso.

MAX vuole trovare una sequenza di azioni che lo porti alla vittoria, ma MIN non vuole permetterglielo.

La ricerca **minimax** viene effettuata quando abbiamo più risultati possibili. É possibile determinare la strategia ottima calcolando il valore minimax di ogni stato, ovvero l'utilità (per MAX) di trovarsi nello stato corrispondente, assumendo che entrambi gli agenti giochino in modo ottimo da lì al termine della partita.

5.2.1 Algoritmo minimax

Se la profondità massima dell'albero è m ed in ogni punti ci sono b mosse legali, la complessità temporale dell'algoritmo sarà $O(b^m)$; quella spaziale $O(bm)$, se l'algoritmo genera i successori tutti insieme, oppure $O(m)$ se li genera uno per volta.

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
  return v
```

Figura 5.3: Algoritmo per calcolare la mossa ottima mediante minimax. Le funzioni MAX-VALUE e MIN-VALUE attraversano l'intero albero di gioco fino alle foglie per determinare il valore di uno stato e la mossa per raggiungerlo.

5.2.2 Potatura alfa-beta

Considerando un nodo n da qualche parte nell'albero, tale che il Giocatore abbia la facoltà di raggiungerlo. Se il Giocatore ha una scelta migliore (ad esempio m') allora non raggiungerà mai n. Possiamo quindi potare n non appena abbiamo raccolto abbastanza informazioni (esaminando alcuni dei suoi discendenti).

La ricerca minimax è in profondità, per cui in ogni momento dobbiamo considerare solo i nodi lungo un singolo cammino dell'albero. L'alpha-beta pruning prende il suo nome dai parametri addizionali in VALORE-MAX($stato, \alpha, \beta$), che descrivono i limiti sui valori "portati su" in un qualsiasi punto del cammino:

- α = valore della scelta migliore per MAX (valore più alto trovato).
- β = il valore della scelta migliore per MIN (valore più basso trovato).

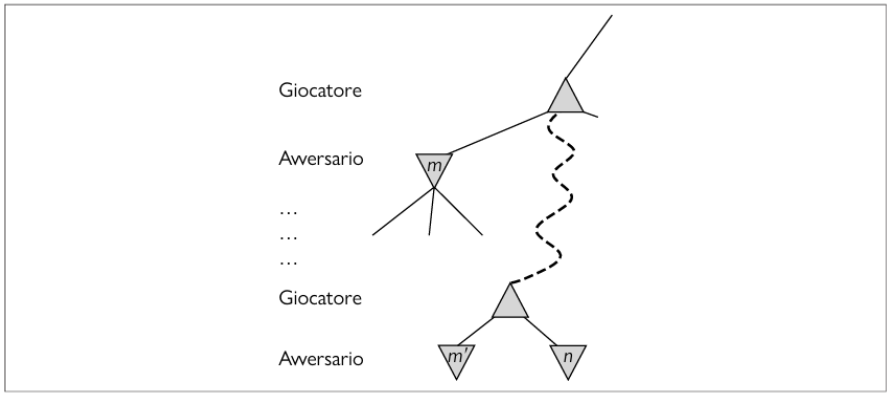

Figura 5.4: Se per il Giocatore m o m' sono preferibili, durante il gioco non arriveremo mai ad n.

L'efficacia della potatura alfa-beta dipende fortemente dall'ordine in cui sono esaminati gli stati. Sarebbe opportuno quindi esaminare per primi i successori migliori. Se fosse possibile ciò per scegliere la mossa migliore si dovrebbero esaminare solo $O(b^{m/2})$ nodi, invece degli $O(b^m)$ richiesti dalla ricerca minimax. Quindi il fattore di ramificazione diventa $\sqrt{b}$ (nel caso degli scacchi quindi avremo 6 invece di 35).

```
function ALPHA-BETA-DECISION(state) returns an action
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  same as MAX-VALUE but with roles of α, β reversed
```

Figura 5.5: Algoritmo di ricerca alfa-beta. Le funzioni sono le stesse della RICERCA-MINIMAX ma manteniamo dei limiti che usiamo per tagliare la ricerca quando un valore è al di fuori di essi.

Per sfruttare al meglio il tempo di calcolo, che è limitato, possiamo interrompere la ricerca in anticipo ed applicare una *funzione di valutazione* euristica agli stati, considerando i nodi non terminali come se lo fossero. In altre parole sostituiamo la funzione UTILITY con EAL (che stima l'utilità di uno stato). Inoltre sostituiamo il test di terminazione con un *cutoff test* che è libero di decidere quando interrompere la ricerca in base alla profondità della stessa e a qualsiasi proprietà dello stato che sceglie di considerare.

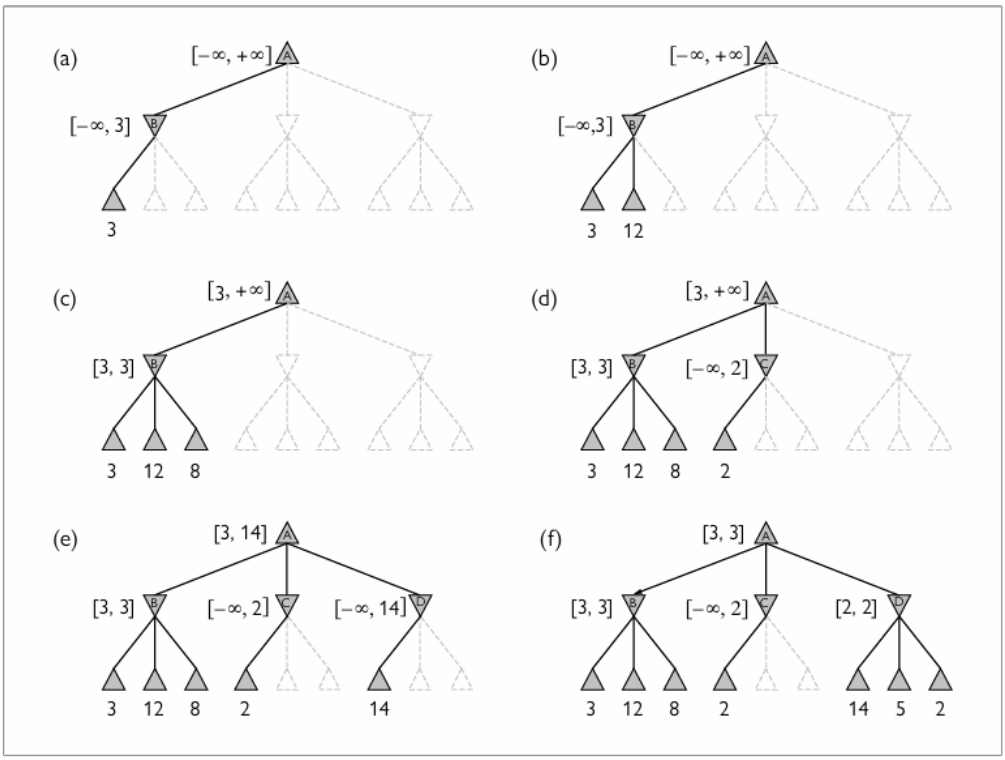

Figura 5.6: Fasi del calcolo della decisione ottima. In ogni punto viene indicato l'intervallo dei possibili valori di ogni nodo. (a) La prima foglia sotto B ha valore 3; ne consegue che B, che è un nodo MIN, ha valore di *al più* 3. (b) La seconda foglia sotto B ha valore 12; MIN la eviterebbe, per cui il valore B è ancora al più 3. (c) la terza foglia sotto B ha valore 8; ora abbia considerato tutti gli stati successori di B, per cui possiamo dire che il valore di B è esattamente 3. (d) La prima foglia sotto C ha valore 2; ne consegue che C, che è un nodo MIN, ha un valore di al più 2. Ma sappiamo che B vale 3, per cui MAX non sceglierebbe C in nessun caso, non c'è alcuna ragione gli altri stati successori di C. (e) La prima foglia sotto D ha valore 14, cosicché D vale al più 14; questo valore è ancora superiore alla migliore alternativa per MAX, quindi continuiamo ad esplorare. (f) Il secondo successore di D vale 5, quindi dobbiamo ancora esplorare. Il terzo stato successore ha valore 2, per cui D vale esattamente 2 e la decisione di MAX alla radice è muoversi nello stato B che offre un valore 3.

5.2.3 Optimal decisions in multiplayer games

Sostituiamo il valore singolo di ogni nodo con un vettore di valori, il quale fornirà l'utilità dello stato dal punto di vista di ogni giocatore.

Solitamente si creano *alleanze* temporanee tra i giocatori. Ad esempio se A e B sono più deboli rispetto a C, non si attaccheranno a vicenda ma penseranno ad indebolire C. Non appena hanno raggiunto l'obiettivo comune, l'alleanza perde valore ed A e B potrebbero violare gli accordi.

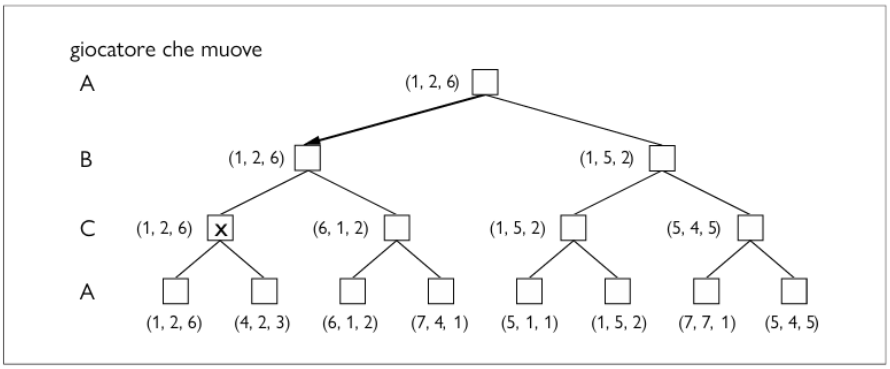

Figura 5.7: I primi tre strati (ply) di un albero di gioco con tre giocatori (A, B, C). Ogni nodo è etichettato con i valori dal punto di vista di ogni giocatore. La mossa migliore partendo dalla radice è evidenziata con la freccia.

5.2.4 Evaluation functions

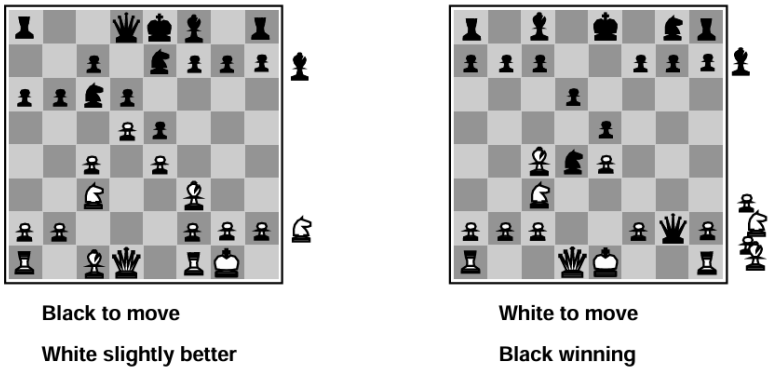

Figura 5.8: Due posizioni sulla scacchiera durante una partita a scacchi.

Matematicamente sommiamo le caratteristiche (features) di uno stato per ottenere la valutazione di una posizione.

$$EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

Dove ogni f_i è una feature della posizione (dipende dal numero di pedoni, regine, torri, alfieri e cavalli del Bianco rispetto al Nero) ed ogni w_i è un peso (weight) che indica l'importanza di tale caratteristica.

I pesi dovrebbero essere normalizzati in modo tale che la somma pesata sia compresa nell'intervallo 0 (sconfitta) ed 1 (vittoria).

5.3 Giochi Stocastici

Introducono un elemento imprevedibile come il lancio di un dato.

Il backgammon è un tipico gioco che combina fortuna ed abilità. Oltre ai nodi MAX e MIN, dobbiamo aggiungere dei **nodi di casualità**. I rami uscenti da ogni nodo di casualità rappresentano i diversi esiti del lancio dei dati.

Ci sono 36 modi di tirare due dadi; dato che però, ai fini del gioco, 6-5 è lo stesso di 5-6, i possibili tiri distinti diventano 21 (ogni tiro doppio: 1-1, 2-2, 3-3, 4-4, 5-5, 6-6 ha probabilità di verificarsi 1/36; gli altri 15 tiri hanno ognuno probabilità 1/18).

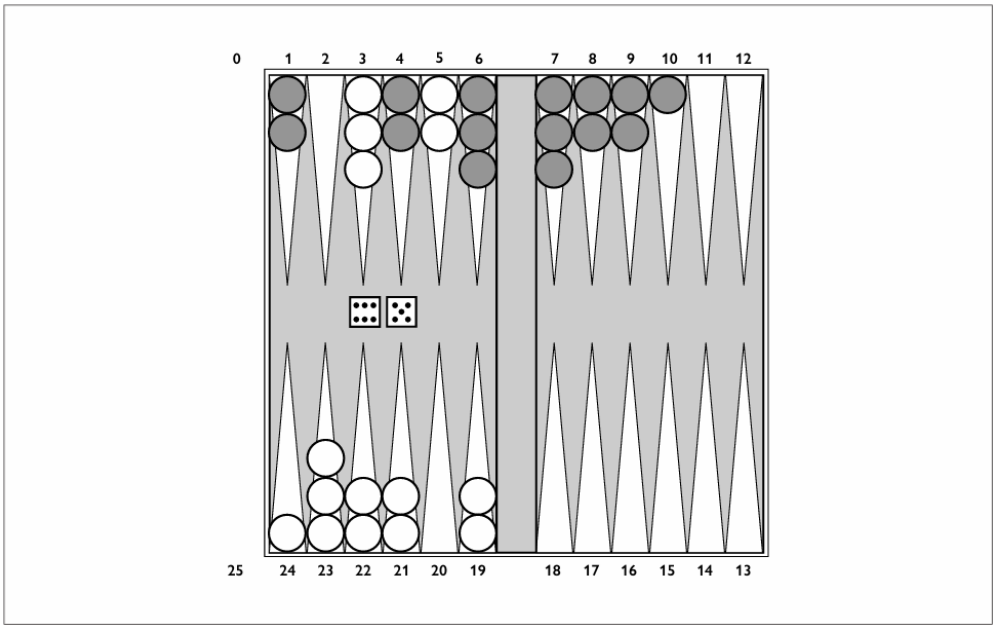

Figura 5.9: Tipica posizione nel backgammon: lo scopo del gioco è far uscire tutti i propri pezzi dal tavoliere. Il Bianco muove in senso orario verso la posizione indicata dal numero 25 ed il Nero in senso antiorario verso lo 0. Un pezzo può muoversi in qualsiasi posizione, a meno che questa non contenga due o più pezzi avversari; se ce n'è uno solo viene catturato e deve ricominciare dall'inizio. Nella posizione indicata, il Bianco ha tirato 6-5 e deve scegliere tra quattro mosse legali: (5-11, 5-10), (5-11, 19-24), (5-10, 10-16), (5-11, 11-16). La notazione, nell'ultimo caso ad esempio, significa muovere un pezzo dalla posizione 5 alla 1 e poi muovere un pezzo dalla 11 alla 16.

Per i giochi stocastici abbiamo il valore **expectiminimax**, una generalizzazione del valore minimax per i giochi deterministici. I nodi MAX e MIN funzionano esattamente come prima, i nodi di casualità calcoliamo il *valore atteso* (la somma del valore di tutti i risultati pesati in base alla probabilità di ciascuna azione).

Notiamo dalla Figura 5.10 come cambiando alcuni valori della valutazione, il programma si comporta in modo totalmente diverso.

Per evitare questo problema la funzione di valutazione deve restituire valori che siano una trasformazione lineare positiva della *probabilità di vincita*.

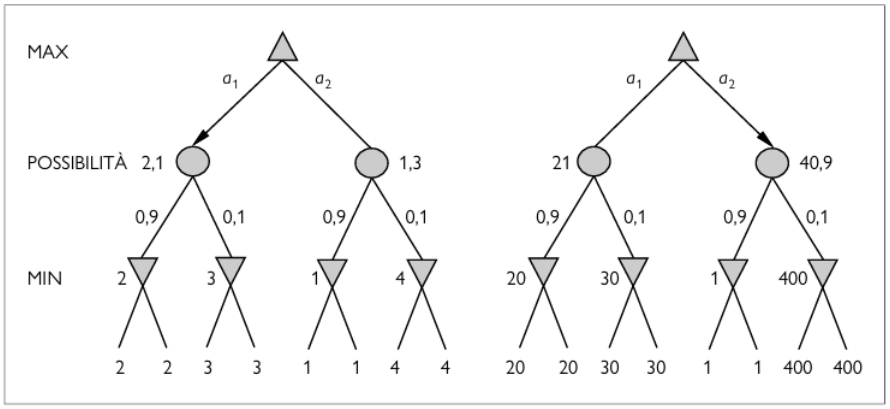

Figura 5.10: Trasformazione dei valori delle foglie che preserva l'ordinamento (modifica tuttavia la scelta della mossa migliore).

5.4 Partially Observable Games

Nei giochi parzialmente osservabili *deterministici*, l'incertezza sullo stato del gioco nasce interamente dall'impossibilità di accedere alle scelte fatte dall'avversario. Un esempio di questi sono: Battaglia Navale, Stratego, tic-tac-toe.

5.4.1 Giochi di carte

I giochi come bridge, whist, hearts e poker sono caratterizzati da osservabilità parziale *stocastica*, in cui le informazioni mancanti sono generate da una distribuzione casuale delle carte. Consideriamo il seguente racconto:

> Giorno 1: la strada A porta ad una pentola d'oro; la strada B porta ad un bivio. La diramazione sinistra porta a due pentole d'oro, mentre in quella a destra veniamo investiti da un autobus.
>
> Giorno 2: la strada A porta ad una pentola d'oro; la strada B ad un bivio. La diramazione destra porta a due pentole d'oro, mentre nella sinistra veniamo investiti da un autobus.
>
> Giorno 3: la strada A porta una pentola d'oro; la strada B porta ad un bivio. Questa volta non sappiamo qual è la via giusta ma ci viene detto che una diramazione porta alle pentole d'oro e l'altra alla morte.

La media sulla chiaroveggenza porta al seguente ragionamento: giorno 1 scelgo B, giorno 2 scelgo B ed al giorno 3 scelgo ancora B perché la situazione sarà identica o al giorno 1 o al giorno 2. Questa strategia quindi fallisce, perché potrebbe portarti a morte certa.

Nel bridge, ad esempio, ogni giocatore vede solo due delle quattro mani (ci sono due ani nascoste di 13 carte ciascuna). Risolvere anche una sola distribuzione è difficile, in questo caso dovremmo risolverne circa dieci milioni. Quindi si ricorre all'**astrazione**, considerando identiche le mani simili.

Capitolo 6

Logical Agents

6.1 Knowledge-Based Agents

La base di conoscenza è costituita da un insieme di formule (set of *sentences*). Ognuna rappresenta un'asserzione sul mondo. Una formula data per buona, senza essere ricavata da altre formule, è detta *assioma*.

Un processo di *inferenza*, ovvero la derivazione di nuove formule a partire da quelle conosciute, è implementato tramite due azioni TELL e ASK.

```
function KB-Agent(percept) returns an action
  static: 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
```

Figura 6.1: Un generico agente basato sulla conoscenza. Data una percezione, l'agente la aggiunge alla propria base di conoscenza (chiede alla base di conoscenza quale sia l'azione migliore e dice alla base di conoscenza che ha in effetti intrapreso tale azione).

Ogni volta che viene invocato, il programma agente fa tre cose: comunica le sue percezioni (attraverso TELL) alla base di conoscenza; chiede (ASK) quale azione eseguire; l'azione scelta viene registrata nella base di conoscenza (con TELL) e successivamente viene restituita in modo da poter essere eseguita.

Questo procedimento prende il nome di *approccio dichiarativo* alla realizzazione di sistemi. In contrapposizione a questo c'è l'approccio procedurale.

Un agente in grado di apprendere può essere completamente autonomo.

6.2 Wumpus World

Una caverna, formata da stanze collegate da passaggi. Da qualche parte c'è il terribile wumpus. Il wumpus può essere colpito da lontano da un agente che però ha solo una freccia. Alcune stanze contengono dei pozzi sena fondo (che intrappolano chiunque entri, tranne il wumpus che non caccia essendo più grande del buco). La caratteristica positiva di quest'ambiente è la possibilità di trovare dell'oro.

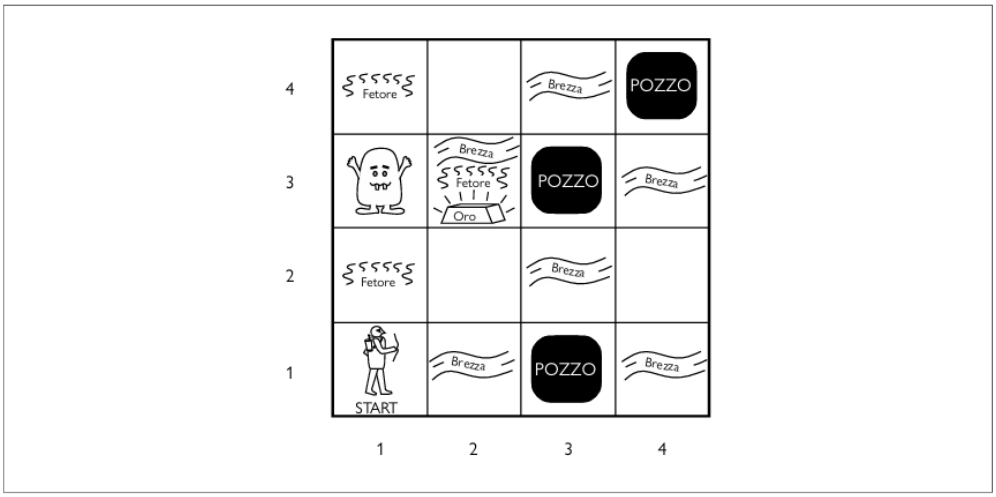

Figura 6.2: Mondo wumpus. L'agente sta in basso a sinistra rivolto ad est.

Descrizione PEAS dell'ambiente:

- **Misura di prestazione:** + 1000 se si riesce ad uscire dalla caverna portando l'oro, -1000 se si cade in un pozzo o si viene mangiati dal wumpus, -1 per ogni azione eseguita, -10 per l'uso della freccia. Il gioco termina se l'agente muore o esce dalla caverna.

- **Ambiente:** una griglia 4x4 di stanze, circondata da pareti di delimitazione. L'agente comincia sempre in posizione [1, 1], rivolto verso est. Le posizioni dell'oro e del wumpus invece sono scelte casualmente, con una distribuzione uniforme, tra tutti i riquadri eccetto quello iniziale. Inoltre, ogni casella (tranne quella iniziale) ha una probabilità 0.2 di contenere un pozzo senza fondo.

- **Attuatori:** l'agente può muoversi **Avanti**, *GiraSinistra di 90°*, *GiraDestra di 90°*. L'agente muore se entra in un riquadro contenente il pozzo o il wumpus vivo. Se l'agente tenta di muoversi in avanti e si scontra con un muro, non si muove. L'azione *Afferra* può essere usata per prendere l'oro che si trova nella stessa stanza. L'azione *Scocca* scaglia una freccia in linea retta nella direzione verso cui l'agente è rivolto (la freccia continua la sua corsa finché non colpisce il wumpus uccidendolo o sbatte contro un muro). L'azione *Esci* può essere utilizzata per arrampicarsi fuori dalla caverna, ma soltanto dalla stanza [1, 1].

- **Sensori:** l'agente ha cinque sensori, ognuno dei quali fornisce un singolo bit di informazione:
  - nei riquadri adiacenti (non diagonali) a quello che contiene il wumpus, l'agente percepirà un *Fetore*;
  - nei riquadri direttamente adiacenti a un pozzo l'agente percepirà una *Brezza*;
  - nel riquadro contenente l'oro l'agente percepirà uno *Scintillio*;
  - qualora l'agente dovesse sbattere contro un muro, percepirà un *Urto*.
  - quando il wumpus viene ucciso emette un *Ululato*, percepibile nell'intera caverna.

Le percezioni sono fornite all'agente sotto forma di un lista di 5 simboli. Per esempio se c'è puzza, movimento d'aria, nessuno scintillio, urto o ululato, riceverà la percezione $[Fetore, Brezza, None, None, None]$.

L'ambiente è deterministico, discreto, statico e monoagente (il wumpus non si può muovere). É sequenziale, perché si può raggiungere un premio soltanto dopo aver intrapreso molte azioni. É parzialmente osservabile, perché alcuni aspetti dello stato non sono direttamente percepibili (la posizione dell'agente, lo stato di salute del wumpus, la disponibilità di una freccia). Le parti inosservate dello stato sono la posizione del wumpus e dei pozzi.

L'agente sa di essere nella posizione [1, 1] e che la stanza è sicura; indichiamo ciò rispettivamente con "OK".

La prima percezione è [None, None, None, None, None], da cui l'agente può dedurre che le stanze adiacenti, [1, 2] e [2, 1], sono sicure, quindi OK.

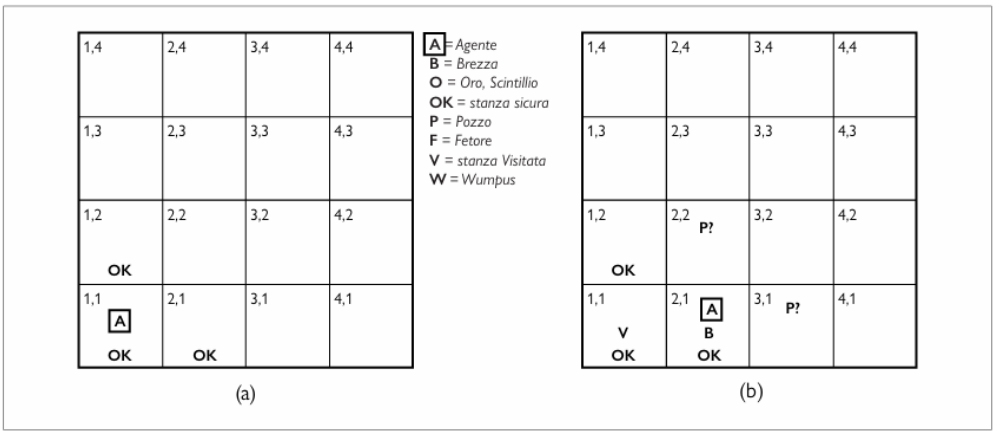

Figura 6.3: Il primo passo dell'agente nel mondo del wumpus. (a) La situazione iniziale, dopo la percezione [None, None, None, None, None]. (b) Dopo la mossa [2, 1], otteniamo percezione [None, Brezza, None, None, None].

Un agente cauto entrerà solo nelle stanze segnate con OK. Supponiamo che l'agente decida di avanzare in [2,1]. L'agente percepisce un movimento d'aria o brezza (denotata da "B") in [2, 1], perciò in una stanza adiacente dev'esserci

un pozzo. Questo non può essere in [1, 1], per cui deve trovarsi in [2, 2] o [3, 1], o anche in entrambe le posizioni. La notazione "P?" indica questa possibilità. A questo punto, c'è un solo riquadro marcato con OK e non ancora visitato; il nostro agente prudente tornerà quindi indietro in [1, 1] e da lì procederà in [1, 2]. L'agente percepisce un fetore in [1, 2], significa che il wumpus è vicino. Esso non può trovarsi in [1, 1] per definizione, e neppure in [2, 2] altrimenti l'agente avrebbe percepito la puzza anche prima (nella stanza [2, 1]). L'agente quindi deduce che il wumpus si trova in [1, 3] e lo segna con la notazione "W!".

L'agente allora si è convinto che in [2, 2] non ci sono né pizzi né mostri.

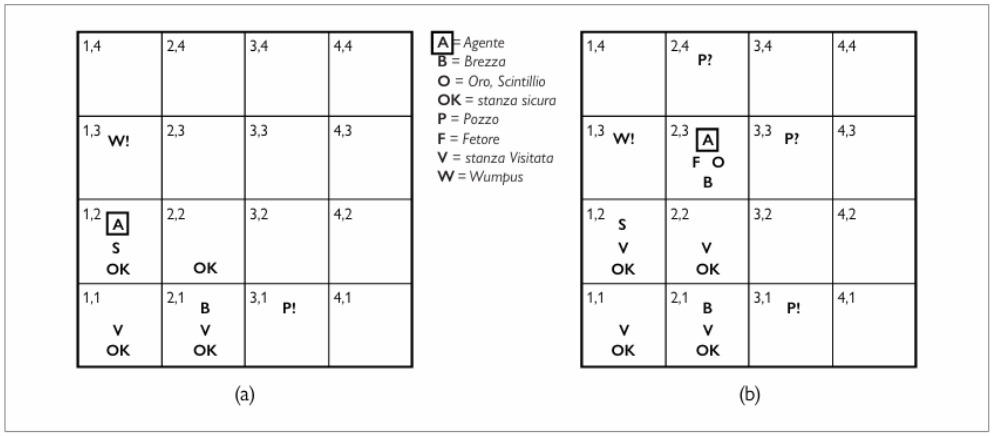

Figura 6.4: Fasi successive dell'esplorazione dell'agente. (a) Dopo le mosse [1, 1] e [1, 2], ottenendo percezione [Fetore, None, None, None, None]. (b) Dopo le mosse [2, 2] e [2, 3], ottenendo percezione [Fetore, Brezza, Scintillio, None, None].

6.3 Logic

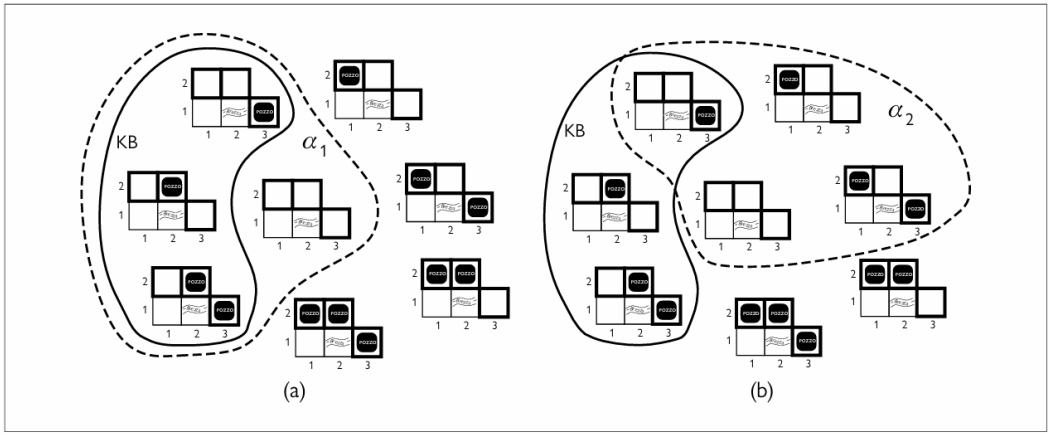

Figura 6.5: I possibili modelli per la presenza di pozzi nelle stanze [1, 2], [2, 2] e [3, 1]. La KB (knowledge Base) corrispondente al non aver rilevato nulla in [1, 1] ed una brezza in [2, 1] è mostrato dalla linea continua. (a) la linea tratteggiata mostra modelli di α_1 (nessun pozzo in [1, 2]). (b) La linea tratteggiata mostra modelli di α_2 (nessun pozzo in [2, 2]).

Le basi della conoscenza sono costituite da formule, queste sono espresse secondo le regole della *sintassi* del linguaggio di rappresentazione, che specifica quali di esse sono "ben formate". Una logica deve anche definire la *semantica* (il significato delle formule). Nelle logiche standard, ogni formula dev'essere o vera o falsa in ogni mondo possibile (non esistono possibilità intermedie).

Quando bisogna essere precisi, si usa il termine **modello** al posto di "mondo possibile".

Se una formula α è vera in un modello m, diciamo che m soddisfa α. Per indicare l'insieme di tutti i modelli di α scriviamo $M(\alpha)$.

L'*entailment* tra due frasi indica che una segue logicamente l'altra. In notazione matematica si scrive cosi: $\alpha \vDash \beta$. In questo caso alfa consegue logicamente beta. Questo vale se e solo se in ogni modello in cui α è vera, lo è anche β. In notazione: $M(\alpha) \subseteq M(\beta)$. Quindi, alfa è un'asserzione più forte di beta.

L'algoritmo di inferenza riportato in Figura 6.5 è chiamato *model checking*, perché enumera tutti i possibili modelli per verificare che α sia vero in tutti quelli in cui è vera la KB, ovvero $M(KB) \subseteq M(\alpha)$.

Possiamo pensare l'insieme di tutte le conseguenze della KB come un pagliaio ed α come un ago. La conseguenza logica dice che l'ago è nel pagliaio; l'inferenza equivale a trovarlo.

Se un algoritmo di inferenza i può derivare α da KB, scriviamo: $KB \vdash_i \alpha$.

Un algoritmo di inferenza che deriva solo formule che sono conseguenze logiche si dice *corretto* (**sound**), esso preserva la verità.

Un'altra proprietà è la **completezza**, un algoritmo è *completo* quando può derivare ogni formula che è conseguenza logica.

6.4 Propositional Logic

La logica proposizionale è la logica più semplice, è molto simile alla logica booleana, essa ha una sintassi di tipo compositivo.

6.4.1 Sintassi

Definisce le formule accettabili. Le formule *atomiche* consistono in un singolo simbolo proposizionale. Ogni simbolo rappresenta una proposizione che può essere vera o falsa. Ci sono cinque connettivi logici di uso comune (in ordine di precedenza):

- *not* ($\neg$), negazione;
- *and* ($\wedge$), congiunzione;
- *or* ($\vee$), disgiunzione;
- *implicazione* ($\Rightarrow$);
- *bicondizionale* ($\Longleftrightarrow$).

Dire che una formula è valida corrisponde a dire che è sintatticamente corretta.

6.4.2 Semantica

Le regole usate per determinare il valore di verità (true, false) di una formula nei confronti di un particolare modello.

Se le formule nella base di conoscenza fanno uso dei simboli $P_{1,2}, P_{2,2}, P_{3,1}$; un modello possibile è:

$$m_1 = \{P_{1,2} = true, P_{2,2} = true, P_{3,1} = false\}$$

$P_{1,2}$ è semplicemente un simbolo, potrebbe significare "c'è un pozzo in [1, 2]".

Per le formule atomiche: *True* è vero in ogni modello e $False$ è falso in ogni modello.

Per le formule complesse abbiamo cinque regole ("iff", significa "if and only if"):

- $\neg P$ è vero iff P è falso in m.
- $P \wedge Q$ è vero iff P e Q sono entrambi veri in m.
- $P \vee Q$ è vero iff o P o Q è vero in m.
- $P \Rightarrow Q$ è vero a meno che P sia vero e Q falso in m.
- $P \iff Q$ è vero se P e Q sono entrambi vero o entrambi falsi in m.

P	Q	¬P	P ∧ Q	P ∨ Q	P ⇒ Q	P ⇔ 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

Figura 6.6: Tavole di verità dei cinque connettivi logici.

Esempio: Knowledge base ed inference procedure

Costruiamo una base di conosceza per il mondo del wumpus. Concentriamoci prima sugli aspetti *immutabili* di tale mondo:

- $P_{x,y}$ è vero se esiste un pozzo in [x,y].
- $W_{x,y}$ è vero se esiste un wumpus in [x,y]; morto o vivo.
- $B_{x,y}$ è vero se esiste una brezza in [x,y].
- $S_{x,y}$ è vero se esiste un fetore in [x,y].
- $L_{x,y}$ è vero se l'agente si trova nella locazione [x,y].

Etichettiamo ogni formula con R_i.

In [1,1] non ci sono pozzi: $R_1 : \neg P_{1,1}$.

In una stanza si percepisce brezza se e solo se c'è un pozzo in una stanza adiacente: $R_2 : B_{1,1} \iff (P_{1,2} \vee P_{2,1})$; $R_3 : B_{2,1} \iff (P_{1,2} \vee P_{2,2} \vee P_{3,1})$.

Le formule sopra sono vere in tutti i mondi del wumpus. Le percezioni relative alla brezza raccolte nelle due prime stanze visitate dall'agente sono: $R_4 : \neg B_{1,1}$; $R_5 : B_{2,1}$.

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

Figura 6.7: Una tavola di verità per la base di conoscenza descritta sopra. KB è vera se sono vere tutte le formule da R_1 ad R_5, il che si verifica solo in 3 delle 128 righe. In tutte e tre $P_{1,2}$ è falsa, quindi possiamo dedurre che non c'è alcun pozzo in [1,2]. In [2,2] invece potrebbe esserci un pozzo o meno.

Data una formula $\alpha, KB \models \alpha$, dobbiamo capire se (ad esempio) $\neg P_{1,2}$ è conseguenza logica della nostra KB.

Nel *model checking*, enumeriamo esplicitamente i modelli e verifichiamo che α è vera in ogni modello in cui la KB lo è.

I modelli sono rappresentati da un assegnamento dei valori *true* o $false$ ad ogni simbolo proposizionale.

```
function TT-ENTAILS?(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic
  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, [])

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
      if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
      else return true
  else do
      P ← FIRST(symbols); rest ← REST(symbols)
      return TT-CHECK-ALL(KB, α, rest, EXTEND(P, true, model)) and
             TT-CHECK-ALL(KB, α, rest, EXTEND(P, false, model))
```

Figura 6.8: Algoritmo che enumera una tavola di verità per determinare la conseguenza logica proporzionale ("TV" sta appunto per "Tavola di Verità").

L'algoritmo è **corretto**, dato che implementa direttamente la definizione di conseguenza logica ed è **completo** perché funziona con ogni base di conoscenza KB, formula α e termina sempre l'esecuzione.

Se il numero totale di simboli contenuti in KB e α è: n, i modelli saranno 2^n. La complessità temporale dell'algoritmo allora è $O(2^n)$; quella spaziale solamente $O(n)$, perché l'enumerazione viene svolta in profondità.

6.5 Propositional Theorem Proving

$(\alpha \wedge \beta)$	$\equiv$	$(\beta \wedge \alpha)$ 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 $\vee$
$\neg(\neg\alpha)$	$\equiv$	α eliminazione della doppia negazione
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\beta \Rightarrow \neg\alpha)$ 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 $\wedge$

Figura 6.9: Equivalenze logiche standard. I simboli α, β, γ rappresentano formule arbitrarie della logica proporzionale.

La conseguenza logica può essere ottenuta tramite la *dimostrazione di teoremi*, applicando regole di inferenza direttamente alle formule della nostra base di conoscenza per costruire una dimostrazione della formula desiderata senza consultare modelli.

Equivalenza logica: due formule α e β sono logicamente equivalenti se sono vere nello stesso insieme di modelli. Scriviamo $\alpha \equiv \beta$. Questo accade *se e solo se* $\alpha \models \beta$ e $\beta \models \alpha$; quindi, se ognuna di esse è conseguenza logica dell'altra.

Validità: una formula è valida se è vera in tutti i modelli. Ad esempio $P \vee \neg P$ è valida. Queste formule sono note anche come *tautologie*. **Teorema di deduzione:**

date due formule qualsiasi α *e* β : $\alpha \models \beta$ *se e solo se la formula* $\alpha \Rightarrow \beta$ *è valida.*

Viceversa, il teorema afferma che ogni formula di implicazione valida descrive un'inferenza legittima.

Soddisfacibilità: una formula è soddisfacibile se è vera in, o soddisfatta da, qualche modello. La soddisfacibilità può essere verificata enumerando i possibili modelli finché non se ne trova uno che soddisfa la formula.

Una formula è **insoddisfacibile** se è sempre falsa $A \wedge \neg A$.

Validità e soddisfacibilità sono connesse: α è valida se e solo se $\neg\alpha$ è insoddisfacibile; di contro α è soddisfacibile se e solo se $\neg\alpha$ non è valida.

$$\alpha \models \beta \textit{ se e solo se } (\alpha \wedge \neg\beta) \textit{ è insoddisfacibile.}$$

Dimostrare β partendo da α verificando l'insoddifacibilità di $(\alpha \wedge \beta)$ corrisponde alla dimostrazione per *assurdo*, detta anche per *refutazione* o *contraddizione*.

6.5.1 Regole di inferenza

Mondus Ponens:

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

Per esempio, se sono date le formule $(WumpusDavanti \wedge WumpusVivo) \Rightarrow ScagliaFreccia$ e vale $(WumpusDavanti \wedge WumpusVivo)$, allora si può inferire $ScagliaFreccia$.

Eliminazione degli and, data una congiunione, si può inferire uno qualsiasi dei congiunti:

$$\frac{\alpha \wedge \beta}{\alpha}$$

Per esempio, da $(WumpusDavanti \wedge WumpusVivo)$ si può inferire $WumpusVivo$.

Utilizziamo queste regole nel mondo del wumpus. La base di conosceza iniziale contiene le formule da R_1 ad R_5 e vogliamo dimostrare $\neg P_{1,2}$; ovvero che non c'è un pozzo in [1,2].

1. Applichiamo l'eliminazione del bicondizionale ad R_2 per ottenere

$$R_6 : (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

2. Applichiamo l'eliminazione degli and a R_6 per ottenere

$$R_7 : ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

3. L'equivalenza logica della contrapposizione fornisce

$$R_8 : (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1}))$$

4. Applichiamo Mondus Ponens con R_8 e la percezione R_4 (cioè $\neg B_{1,1}$), ottendo

$$R_9 : \neg(P_{1,2} \vee P_{2,1})$$

5. Applichiamo la regola di De Morgan, arrivando alla conclusione

$$R_{10} : \neg P_{1,2} \wedge \neg P_{2,1}$$

Ovvero, non ci sono pozzi né in [1,2] né in [2,1].

Monotonicità: afferma che un insieme di formule che sono conseguenze logiche può solo *aumentare* man mano che si aggiunge informazione alla base di conoscenza. Per qualsiasi coppia di formule α, β:

$$se \quad KB \models \alpha \quad allora \quad KB \wedge \beta \models \alpha$$

La *risoluzione* è una regola di inferenza, che unita a qualsiasi algoritmo di ricerca completo dà luogo ad un algoritmo di inferenza completo.

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \quad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

Dove ℓ_i, m_j sono letterali complementari.

La regola di inferenza di *risoluzione unitaria:*

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \quad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k}$$

dove ogni ℓ è un letterale, ed ℓ_i, m sono letterali complementare (uno la negazione dell'altro). Questa regola quindi prende una *clausola* ed un letterale e produce una nuova clausola.

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

La clausola risultante dovrebbe contenere solo una copia di ogni letterale. La rimozione delle eventuali copie è chiamata *fattorizzazione.*

6.5.2 CNF (Conjuntive Normal Form)

Ogni formula della logica proposizionale è logicamente equivalente a una congiunzione di clausole. Procedura di conversione:

1. eliminiamo $\iff$, rimpiazzando $\alpha \iff \beta$ con $(\alpha \Rightarrow \beta) \wedge (\alpha \Rightarrow \beta)$:

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

2. eliminiamo $\Rightarrow$, rimpiazzando $\alpha \Rightarrow \beta$ 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. Il $\neg$ si deve applicare solo ai letterali, per cui lo "spostiamo all'interno", usando le seguenti equivalenze:

   $$\neg(\neg\alpha) \equiv \alpha \text{ (eliminazione della doppia negazione)}$$

   $$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \wedge \neg\beta) \text{ (De Morgan)}$$
   $$\neg(\alpha \vee \beta) \equiv (\neg\alpha \vee \neg\beta) \text{ (De Morgan)}$$

   Nel nostro caso:

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

4. Abbiamo troppi operatori nidificati, sfruttando la legge di distributività:

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

La formula trasformata in CNF ha una lettura più ardua, ma può essere usata come input per una procedura di risoluzione.

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic
  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
      for each C_i, C_j in clauses do
          resolvents ← PL-RESOLVE(C_i, C_j)
          if resolvents contains the empty clause then return true
          new ← new ∪ resolvents
      if new ⊆ clauses then return false
      clauses ← clauses ∪ new
```

Figura 6.10: Algoritmo di risoluzione per la logica proposizionale. La funzione restituisce l'insieme di tutte le possibili clausole ottenute risolvendo i due input.

Per dimostrare che $KB \models \alpha$, dimostriamo che $KB \wedge \neg\alpha$ è insoddisfacibile. Dobbiamo, quindi, giungere ad una contraddizione.

Convertiamo per prima cosa $(KB \wedge \neg\alpha)$ in CNF. 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 possibilità:

- non è più possibile aggiungere alcuna clausola, nel qual caso α non è conseguenza logica di KB;
- la risoluzione applicata a due clausole dà come risultato la clausola vuota, nel qual caso KB consegue logicamente α.

La clausola vuota (disgiunzione senza alcun disgiunto) è equivalente a *false*.

Nel mondo del wumpus: quando l'agente si trova in [1,1] non percepisce alcun spostamento d'aria, per cui non ci possono essere pozzi nelle stanze adiacenti. La base di conoscenza relativa è:

$$KB = R_2 \wedge R_4 = (B_{1,1} \iff (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

Vogliamo dimostrare che alfa corrisponde a $\neg P_{1,2}$. PL-RESOLUTION è completo. Teorema di risoluzione ground: se n insieme di clausole è insoddisfacibile, la sua chiusura della risoluzione contiene la clausola vuota.

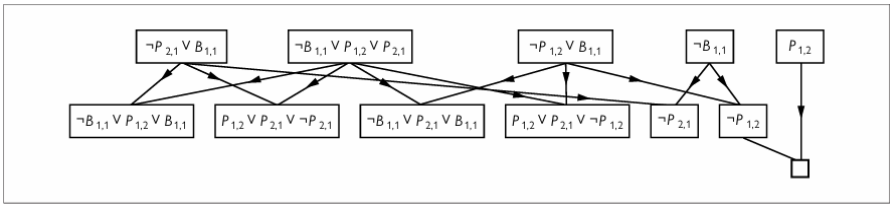

Figura 6.11: Applicazione parziale di PL-RESOLUTION ad una semplice inferenza nel mondo del wumpus per dimostrare la query $\neg P_{1,2}$. Ognuna delle quattro clausole più a sinistra nella riga superiore è accoppiata con ognuna delle altre tre e si applica la regola di risoluzione per ottenere le clausole nella riga inferiore. La terza e quarta clausola nella riga superiore si combinano per dare come risultato la clausola $\neg P_{1,2}$ che viene poi risolta con $P_{1,2}$ ottenendo la clausola vuota, ad indicare che la query è dimostrata.

6.5.3 Clausole di Horn

Una disgiunzione di letterali in cui al massimo uno dei letterali è positivo. Le clausole *definite* (una disgiunzione di letterali di cui esattamente uno è positivo) e le clausole *obiettivo* (senza alcun letterale positivo), sono clausole di Horn.

1. Ogni clausola definita può essere scritta come un'implicazione la cui premessa è una congiunzione di letterali positivi e la cui conclusione è un singolo letterale positivo.

2. L'inferenza sulla clausole di Horn può essere svolta mediante algoritmi di concatenazione in avanti e all'indietro.

3. Con le clausole di Horn è possibile determinare la conseguenza logica in un tempo che cresce linearmente con la dimensione della base di conoscenza.

6.5.4 Forward chaining and Backward chaining

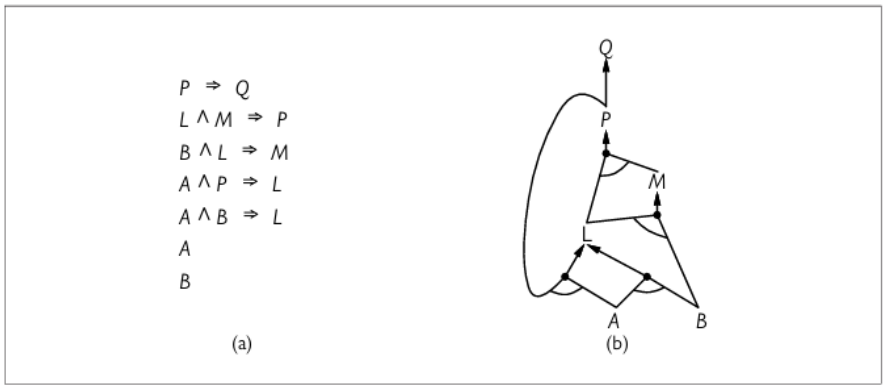

Figura 6.12: Un insieme di clausole definite (a). Il corrispondente grafo AND-OR (b).

Nei grafi AND-OR, più collegamenti uniti da un arco indicano una congiunzione (ogni collegamento dev'essere dimostrato), mentre collegamenti multipli senza arco indicano una disgiunzione (basta dimostrare uno dei qualsiasi collegamenti). Partendo dalle foglio conosciute (A e B) l'inferenza si propaga nel grafo il più lontano possibile. Ogni volta che si incontra una congiunzione, la propagazione attende finché non sono noti tutti i congiunti.

É facile notare che la concatenazione in avanti è **corretta** (ogni inferenza sostanzialmente è un'applicazione del Modus Pones) e l'algoritmo è **completo** (ogni formula atomica conseguenza logica sarà derivata).

La concatenazione in avanti è un esempio del concetto generale di *ragionamento guidato dai dati*.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional Horn clauses
          q, the query, a proposition symbol
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known in KB

  while agenda is not empty do
      p ← POP(agenda)
      unless inferred[p] do
          inferred[p] ← true
          for each Horn clause c in whose premise p appears do
              decrement count[c]
              if count[c] = 0 then do
                  if HEAD[c] = q then return true
                  PUSH(HEAD[c], agenda)
  return false
```

Figura 6.13: Algoritmo di concatenazione in avanti per la logica proposizionale. La coda tiene traccia dei simboli che sono noti come veri ma non ancora "considerati". La tabella conto tiene traccia di quante premesse di ogni implicazione non sono ancora dimostrate. Ogni volta che un nuovo simbolo p della coda è considerato, il conto viene ridotto di uno per ogni implicazione in cui appare la premessa p (facilmente individuata in tempo costante, con un'indicizzazione appropriata). Se un conto arriva a zero, significa che tutte le premesse di un'implicazione sono note, in modo tale che la sua conclusione può essere aggiunta alla coda. Infine, dobbiamo tener traccia dei simboli considerati; un simbolo non dev'essere aggiunto nuovamente alla code se è già compreso nell'insieme dei simboli inferiti. Questo permette di evitare di svolgere elaborazioni inutili e previene cicli infiniti nel caso esistano coppie di implicazioni come $P \Rightarrow Q$ e $Q \Rightarrow P$.

L'algoritmo di concatenazione all'indietro, invece, parte dalla query e lavora a ritroso. Se è già noto che la query è vera, non occorre fare nulla. In caso contrario, l'algoritmo trova tutte le implicazioni nella base di conoscenza che hanno q come conclusione; se tutte le premesse di una di quelle implicazioni possono essere dimostrate vere mediante la concatenazione all'indietro, allora q è vera. La concatenazione all'indietro è una forma di *ragionamento basato sugli obiettivi*.

Capitolo 7

First Order Logic

La logica proposizionale è un linguaggio *dichiarativo* perché la sua semantica è basata su una relazione di verità che collega le formule e i mondi possibili. L'uso della negazione e della disgiunzione lo rende sufficientemente espressivo da gestire informazione parziale. Un'altra caratteristica è la *composizionalità*, il significato di una formula è una funzione del significato delle sua parti. La logica proposizionale però è anche *limitata* nelle capacità di rappresentazione.

Quando prendiamo in esame la sintassi di un linguaggio naturale, gli elementi che si distinguono più facilmente sono i sostantivi e i sintagmi nominali che si riferiscono ad **oggetti** (stanze, pozzi, wumpus) e i verbi e i sintagmi verbali, con aggettivi e avverbi, che si riferiscono a **relazioni** tra gli oggetti (ventosa, adiacente, scocca). Alcune relazioni, dette **funzioni**, hanno un solo "valore" per ogni "input". Altri esempi di:

- *oggetti* (persone, cavalli, numeri, teoria, colori, partite, guerre, secoli...);
- *relazioni*, possono essere unarie: proprietà (essere rossi, alti, primi, falsi), oppure n-arie (fratello di, più grande di, avvenuto dopo, di colore, possiede...);
- *funzioni* (padre di, miglior amico, uno più di, all'inizio di...).

Il linguaggio della *logica del primo ordine* è costruito intorno agli oggetti e alle relazioni. La differenza fondamentale tra la logica proposizionale e quella del primo ordine sta nell'impegno *ontologico* assunto da ognuno dei linguaggi, ovvero le sue ipotesi circa la natura della realtà. Matematicamente, questo impegno è espresso attraverso la natura dei modelli formali rispetto ai quali è definita la verità delle formule.

linguaggio	impegno ontologico (ciò che esiste nel mondo)	impegno epistemologico (le credenze di un agente circa un fatto)
logica proposizionale	fatti	vero/falso/sconosciuto
logica del primo ordine	fatti, oggetti, relazioni	vero/falso/sconosciuto
logica temporale	fatti, oggetti, relazioni, tempi	vero/falso/sconosciuto
teoria della probabilità	fatti	grado di credenza $\in [0, 1]$
logica fuzzy	fatti con gradi di verità $\in [0, 1]$	intervallo di valori conosciuto

Figura 7.1: Linguaggi formali con i rispettivi impegni ontologici ed epistemologici.

La logica proposizionale dà per scontato che i fatti nel mondo sono veri oppure no; la logica del primo ordine fa ulteriori ipotesi, in particolare che il mondo consista di oggetti legati da relazioni che possono o meno essere verificate.

7.1 Sintassi e Semantica della FOL

Il **dominio** di un modello è l'insieme degli oggetti che contiene. Il dominio non deve essere vuoto (ogni mondo possibile deve contenere almeno un oggetto).

La Figura 7.2 mostra un modello con cinque oggetti (Riccardo Cuor di Leone, Re Giovanni il malvagio, le gambe sinistre di entrambi e una corona). Questi oggetti possono essere messi in relazione in molti modi.

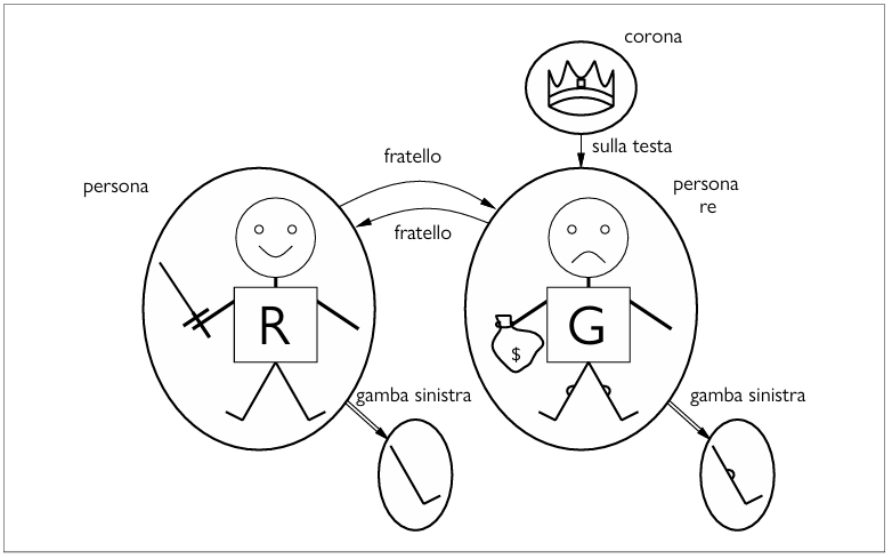

Figura 7.2: Modello che contiene cinque oggetti, due relazioni binarie (fratello, sulla testa), tre relazioni unarie (persona, re, corona) ed una funzione unaria (gamba sinistra).

Gli elementi sintattici base sono: *costanti:* (rappresentano gli oggetti, introduciamo un simbolo che servirà a dare un nome a degli oggetti); *predicati:* (rappresentano le relazioni); *funzioni.*

Un **termine** è un'espressione logica che si riferisce ad un oggetto.

Invece di scrivere "la gamba sinistra di Re Giovanni", possiamo usare *GambaSinistra(Giovanni).*

Possiamo mettere insieme i termini e i simboli di predicato per dare origine a **formule atoiche** capaci di asserire fatti. Queste sono composte quindi da un simbolo di predicato seguito da una lista di termini tra parentesi: *Fratello(Riccardo, Giovanni).*

Le formule atomiche possono avere termini complessi come argomenti, ad esempio *Sposato(Padre(Riccardo), Madre(Giovanni)).*

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

Possiamo usare i *connettivi logici* per costruire delle **formule complesse**.

I **quantificatori** ci permettono di esprimere caratteristiche di intere collezioni di oggetti senza doverli enumerare uno per uno.

Il **quantificatore universale** ($\forall$).

$$\forall x \quad Re(x) \Rightarrow Persona(x)$$

Per ogni x, se x è un re, allora x è una persona. Il simbolo x prende il nome di **variabile**.

É possibile, invece, formulare enunciati circa alcuni oggetti senza citarli per nome, usando un **quantificatore esistenziale** ($\exists$).

$$\exists x \quad Corona(x) \wedge SullaTesta(x, Giovanni)$$

Re Giovanni ha una corona sulla testa.

Possiamo esprime formule più complesse usando più quantificatori.

$$\forall x \, \forall y \quad Fratello(x, y) \Rightarrow Consanguineo(x, y)$$

I fratelli sono consanguinei.

$$\forall x, \; y \quad Consanguineo(x, y) \iff Consanguineo(y, x)$$

La consanguineità è una relazione simmetrica.

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

Ognuno ama qualcuno.

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

C'è qualcuno che è amato da tutti.

L'ordine con cui sono scritti i quantificatori, quindi, è molto importante.

Dire che tutti odiano i broccoli è come dire che non esiste nessuno a cui piacciono:

$$\forall x \quad \neg Gradisce(x, Broccoli)$$

$$\neg \exists x \quad Gradisce(x, Broccoli)$$

A tutti piace il gelato equivale a dire che non c'è nessuno a cui non piace:

$$\forall x \quad \neg Gradisce(x, Gelato)$$

$$\neg \exists x \quad \neg Gradisce(x, Gelato)$$

$$\begin{array}{lcl} \neg \exists x \; \neg P & \equiv & \forall x \; P \\ \neg \forall x \; P & \equiv & \exists x \; \neg P \\ \forall x \; P & \equiv & \neg \exists x \; \neg P \\ \exists x \; P & \equiv & \neg \forall x \; \neg P \end{array} \qquad \begin{array}{lcl} \neg P \wedge \neg Q & \equiv & \neg P \vee Q \\ \neg (P \wedge Q) & \equiv & \neg P \vee \neg Q \\ P \wedge Q & \equiv & \neg (\neg P \vee \neg Q) \\ P \vee Q & \equiv & \neg (\neg P \wedge \neg Q) \end{array}$$

Figura 7.3: Leggi di De Morgan per le formule quantificate e non.

Possiamo usare il simbolo di **uguaglianza** per affermare che due termini fanno riferimento allo stesso oggetto.

$$Padre(Giovanni) = Enrico$$

Determinare la verità di una formula di uguaglianza consiste nel verificare se i due termini si riferiscono allo stesso oggetto.

Possiamo usare l'uguaglianza per specificare che due termini non soo lo stesso oggetto.

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

7.2 Tell & Ask in FOL

Per aggiungere formule alla base di conoscenza usiamo TELL. Questo tipo di formule sono chiamate *asserzioni*. Per interrogare la base di conoscenza si usa ASK. Le domande poste così prendono il nome di *query*.

La risposta viene detta *sostituzione* e si indica con σ. Data una formula S su cui si è realizzata una sostituzione, diciamo che S sostituito è uguale a qualcosa. Ad esempio:

$$S = Smarter(x, y)$$

$$\sigma = \{x/Hilary, y/Bill\}$$

$$S\sigma = Smarter(Hilary, Bill)$$

Scrivere poi $ASK(KB, S)$ ritorna alcune o tutte le σ tali che $KB \models S\sigma$. Altre risposte potrebbero essere semplicemente *vero* o *falso*.

Capitolo 8

Inference in First-Order Logic

Un modo per fare inferenza del primo ordine è quello di convertire la base di conoscenza in logica proposizionale e usare l'inferenza proposizionale. Un primo passo è quello di eliminare i quantificatori universali.

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

Da cui possiamo inferire:

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

e

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

La regola di **Universal Instantiation (UI)** afferma che possiamo inferire tutte le formule ottenute sostituendo un *termine ground* (cioè privo di variabili) ad una variabile quantificata universalmente.

$$\frac{\forall v \ \alpha}{SUBST(\{v/g\}, \alpha)}$$

per ogni variabile v e termine ground g.

La regola di **Existential Instantiation (EI)** sostituisce una variabile quantificata esistenzialmente con un unico nuovo simbolo *costante* k; per ogni formula α, variabile v.

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

Per esempio, dalla formula:

$$\exists x \quad Corona(x) \wedge SullaTesta(x, Giovanni)$$

possiamo inferire:

$$Corona(C_1) \wedge SullaTesta(C_1, Giovanni)$$

A patto che C_1 non appaia in alcun altro punto della KB; questo prende il nome di *costante di* ***Skolem***.

Come convertire qualsiasi base di conoscenza del primo ordine in una base di conoscenza proposizionale: innanzitutto una formula quantificata universalmente può essere sostituita dall'insieme di tutte le possibili istanze. Per esempio, se la base di conoscenza contiene le seguenti formule:

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

$$Re(Giovanni)$$

$$Avido(Giovanni)$$

$$Fratello(Riccardo, Giovanni)$$

e che gli unici oggetti siamo Giovanni e Riccardo. Applichiamo la UI alla prima formula usando tutte le possibili soluzioni ($\{x/Giovanni\}, \{x/Riccardo\}$). Otteniamo:

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

$$Re(Riccardo) \wedge Avido(Riccardo) \Rightarrow Malvagio(Riccardo)$$

Considerando le formule atomiche ground otteniamo la conclusione:
$Malvagio(Giovanni)$.

Questa tecnica di **proposizionalizzazione** può essere applicata in modo generale. Tuttavia, c'è un problema: quando la base di conoscenza include un simbolo di funzione, l'insieme delle possibili soluzioni di termini ground diventa infinito. Per esempio, se la base di conoscenza contiene il sibolo $Padre$, si possono costruire infiniti termini annidati: $Padre(Padre(Padre(Giovanni)))$.

Teorema di Herbrand: se una query α è implicata da una KB in logica del primo ordine, essa è anche inplicata da un sottoinsieme finito dell KB proposizionalizzata ottenuta dalla KB in FOL.

Dato un qualsiasi sottoinsieme finito, la profondità di annidamento dei termini deve essere limitata. Possiamo generare prima tutte le istanziazioni con simboli di costante, poi tutti i termini di profondità 1, quindi tutti quelli di profondità 2, e così via finché non saremo in grado di costruire la dimostrazione proposizionale della formula che è conseguenza logica.

Teorema di Tuning e Church: il problema della conseguanza logica, per la logica del primo ordine è **semidecidibile**; ovvero 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.

8.1 Unification

Una regola che, se esiste una sostituzione θ, rende ognuno dei congiunti della premessa dell'implicazione identico a una formula già presente nella base di conoscenza.

Supponiamo di avere le seguenti formule e di volerle unificare:
$Knows(John, x); Knows(John, Jane)$.

Per poterle unificare dobbiamo trovare una sostituzione per x che rende le due formule uguali, in questo caso è banalmente $Jane$ la soluzione. Indichiamo questa sostituzione con $\theta\{x/Jane\}$.

Avessimo avuto $Knows(John, x)$ e $Knows(x, OJ)$ a primo impatto diremmo che l'unificazione non è possibile, perché $x = John; x = OJ$. Ma essendo due formule diverse (una p e l'altra q), la variabile è semplicemente un segnaposto, possiamo quindi *standardizzare* la seconda formula, ottenendo: $Knows(y, OJ)$. Quindi la soluzione diventa: $\theta\{y/John, x/OJ\}$.

Mondus Pones generalizzato (GMP):

$$\frac{p'_1, p'_2, \cdots, p'_n, (p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q)}{SUBST(\theta, q)}$$

per le formule atomiche p_i, p'_i, q; ove ci sia una sostituzione θ tale che $SUBST(\theta, p'_i) = SUBST(\theta, p_i)$ per ogni i.

Questa regola ha $n + 1$ premesse: le n formule atomiche p'_i e la singola implicazione. La conclusione è il risultato dell'applicazione della sostituzione θ al conseguente q.

É facile dimostrare che il GMP è una regola di inferenza corretta. Per ogni formula p e per ogni sostituzione θ:

$$p \models SUBST(\theta, p)$$

è vero per istanziazione universale.

8.2 Concatenazione in Avanti

Le clausole del primo ordine sono disgiunzioni di letterali esattamente uno dei quali è positivo. Problema di esempio:

> La legge americana afferma che per un americano 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 dal Colonnello West, un americano.

Dobbiamo rappresentare i fatti sotto forma di clausole definita della FOL.

"Per un americano è un crimine vendere armi a nazioni ostili":

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

"Nono possiede dei missili":

$$Possiede(Nono, M_1)$$

$$Missile(M_1)$$

"Tutti i missili gli sono stati venduti dal Colonnello West":

$$Missile(x) \wedge Posside(Nono, x) \Rightarrow Vende(West, x, Nono)$$

Dobbiamo sapere che i missili sono delle armi:

$$Missile(x) \Rightarrow Arma(x)$$

Un nemico dell'America viene considerato ostile:

$$Nemico(x, America) \Rightarrow Ostile(x)$$

"West è americano":

$$Americano(West)$$

"Lo stato di Nono è un nemico dell'America"

$$Nemico(Nono, America)$$

Questa base di conoscenza è di tipo *Datalog* (linguaggio costituito da clausole definite del primo ordine senza simboli di funzione).

```
function FOL-FC-Ask(KB, α) returns a substitution or false
  repeat until new is empty
    new ← { }
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← Standardize-Apart(r)
      for each θ such that (p1 ∧ ... ∧ pn)θ = (p'1 ∧ ... ∧ p'n)θ
                  for some p'1, ..., p'n in KB
        q' ← Subst(θ, q)
        if q' is not a renaming of a sentence already in KB or new then do
          add q' to new
          φ ← Unify(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

Figura 8.1: Algoritmo di concatenazione in avanti. Ad ogni iterazione l'algoritmo aggiunge alla KB tutte le formule atomiche che possono essere inferite in un passo dalle formule di implicazione e dalle formule atomiche già presenti nella KB.

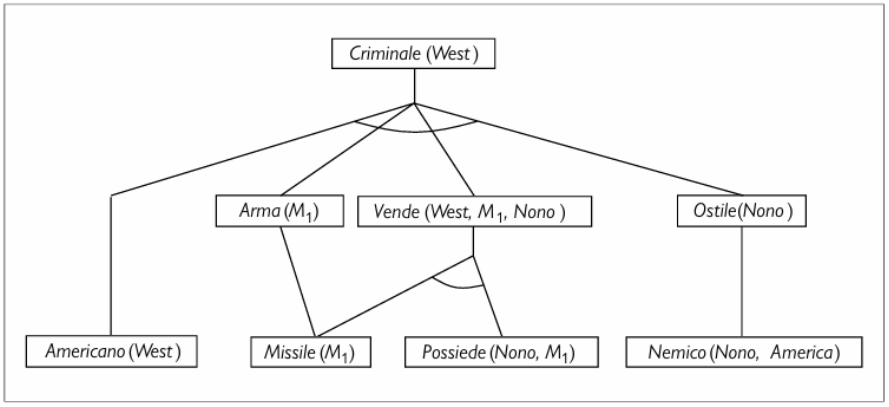

Figura 8.2: Albero di dimostrazione generato dalla concatenazione in avanti sull'esempio del colonnello criminale. I fatti iniziali si trovano al livello più basso, quelli inferiti nella prima iterazione al livello intermedio e quelli inferiti nella seconda iterazione al livello più alto.

Attraverso il forward chaining partendo dai fatti, deduciamo che il colonnello West è un criminale.

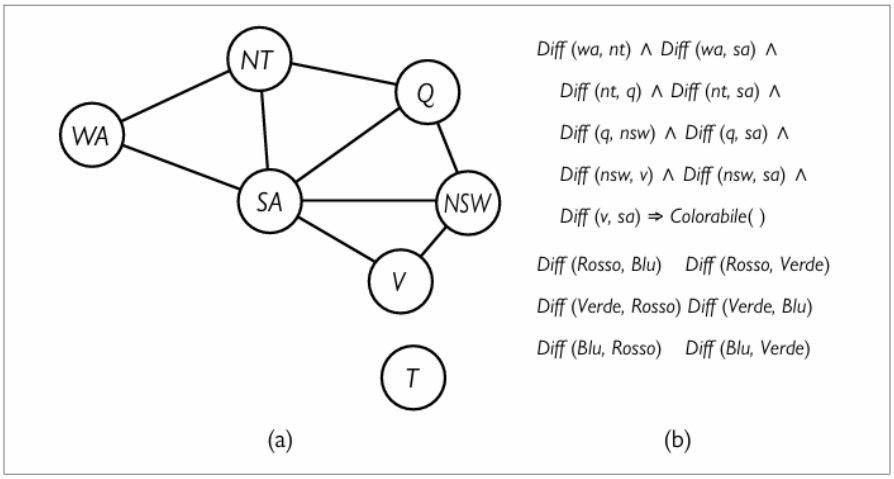

Figura 8.3: Il grafo dei vincoli per la colorazione della mappa dell'Australia (a). Il CSP della colorazione della mappa espresso con un'unica clausola definita (b). Ogni area della mappa è rappresentata come variabile il cui valore può essere una delle costanti (Rosso, Verde, Blu). Le variabili devono avere colori diversi (Diff).

Possiamo esprimere ogni CSP con domini finiti con una singola clausola defiita e alcuni fatti ground ad essa associati.

Dato che i CSP includono come casi speciali i problemi 3SAT, possiamo concludere ce cercare il *matching* tra una clausola definita e un insieme di fatti è un problema NP-Hard. La conclusione $Colorabile()$ potrà essere inferita solo se il CSP ha soluzione.

8.3 Concatenazione all'Indietro

```
function FOL-BC-Ask(KB, goals, θ) returns a set of substitutions
  inputs: KB, a knowledge base
          goals, a list of conjuncts forming a query (θ already applied)
          θ, the current substitution, initially the empty substitution { }
  local variables: answers, a set of substitutions, initially empty

  if goals is empty then return {θ}
  q' ← Subst(θ, First(goals))
  for each sentence r in KB
          where Standardize-Apart(r) = ( p1 ∧ ... ∧ pn ⇒ q)
          and θ' ← Unify(q, q') succeeds
      new_goals ← [p1, ..., pn | Rest(goals)]
      answers ← FOL-BC-Ask(KB, new_goals, Compose(θ', θ)) ∪ answers
  return answers
```

Figura 8.4: Algoritmo di concatenazione all'indietro, per le basi di conoscenza del primo ordine.

La concatenazione all'indietro è un tipo particolare di ricerca AND/OR, dove la parte OR si deve al fatto che la query obiettivo può essere dimostrata da qualsiasi regola dela basi di conoscenza e la parte AND si deve al fatto che tutti i congiunti della lista *lhs* (left-hand side) di una clausola devono essere dimostrati. La concatenazione all'indietro soffre, però, del problema degli stati ripetuti e dell'incompletezza.

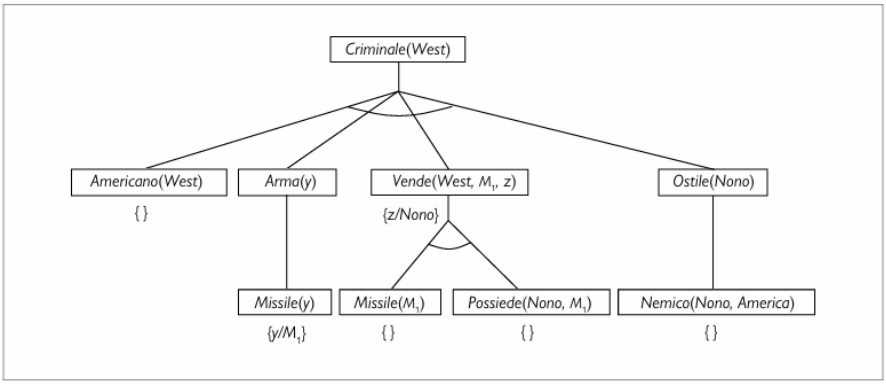

Figura 8.5: Albero costruito con la concatenazione all'indietro per dimostrare che il Colonnello West è un Criminale. L'albero va letto in profondità e da sinistra a destra.

8.3.1 Prolog

Il linguaggio di programmazione logica più utilizzato.

I programmi Prolog consistono in un insieme di clausole definite espresse con una notazione un po' differente da quella standard della logica del primo ordine. Le variabili sono scritte in maiuscolo e le costanti in minuscolo (opposto alla convenzione logica usata fin'ora). Per separare i congiunti nelle clausole si utilizzano le virgole e la clausola è scritta "all'indietro" rispetto a come siamo abituati (invece di $A \wedge B \Rightarrow C$ abbiamo `C :- A, B`). Esempio:

```
criminale(X) :- americano(X), arma(Y), vende(X,Y,Z), ostile (Z).
```

La notazione `[E|L]` denota una lista il cui primo elemento è E ed il resto è L.

Ecco un programma prolog che ha successo se la lista Z è il risultato della concatenazione delle liste X e Y (`concatena(X,Y,Z)`):

```
concatena([],Y,Y).
concatena([A|X],Y,[A|Z]) :- concatena(X,Y,Z).
```

Queste clausole possono essere lette così: "concatena la lista vuota e la lista Y, produce la lista Y; [A—Z] è il risultato della concatenazione di [A—X] con Y a patto che Z sia il risultato della concatenazione di X ed Y".

8.4 Risoluzione

Conversione in **CNF** della formula "chiunque ami tutti gli animali è amato da qualcuno":

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

1. Eliminazione delle implicazioni:
$$\forall x\ \neg[\forall y\ Animale(y) \Rightarrow Ama(x,y)] \vee [\exists y\ Ama(y,x)]$$
$$\forall x\ \neg[\forall y\ \neg Animale(y) \vee Ama(x,y)] \vee [\exists y\ Ama(y,x)]$$

2. Spostamento all'interno delle negazioni:
$$\forall x\ [\exists y\ \neg(\neg Animale(y) \vee Ama(x,y))] \vee [\exists y\ Ama(y,x)]$$
$$\forall x\ [\exists y\ Animale(y) \wedge \neg Ama(x,y)] \vee [\exists y\ Ama(y,x)]$$

3. Standardizzazione delle variabili:
$$\forall x\ [\exists y\ Animale(y) \wedge \neg Ama(x,y)] \vee [\exists z\ Ama(z,x)]$$

4. *Skolemizzazione* (processo di rimozione dei quantificatori esistenziali per eliminazione):
$$\forall x\ [Animale(A) \wedge \neg Ama(x,A)] \vee Ama(B,x)$$
In questo caso si andrebbe a perdere il significato originario, le entità di Skolem devono dipendere da x, otteniamo quindi delle funzioni di Skolem:
$$\forall x\ [Animale(F(x)) \wedge \neg Ama(x,F(x))] \vee Ama(G(x),x)$$
Per regola, gli argomenti di una funzione di Skolem sono tutte le variabili universalmente quantificate all'interno del cui campo di azione appare un quantificatore esistenziale.

5. Omissione dei quantificatori universali:
$$[Animale(F(x)) \wedge \neg Ama(x,F(x))] \vee Ama(G(x),x)$$

6. Distribuzione di $\vee$ si $\wedge$:
$$[Animale(F(x)) \vee Ama(G(x),x)] \wedge [\neg Ama(x,F(x)) \vee Ama(G(x),x)]$$

La risoluzione dimostra che $KB \models \alpha$, provando che $KB \wedge \neg\alpha$ non è soddisfacibile. Per fare questo dobbiamo ottenere la clausola vuota.

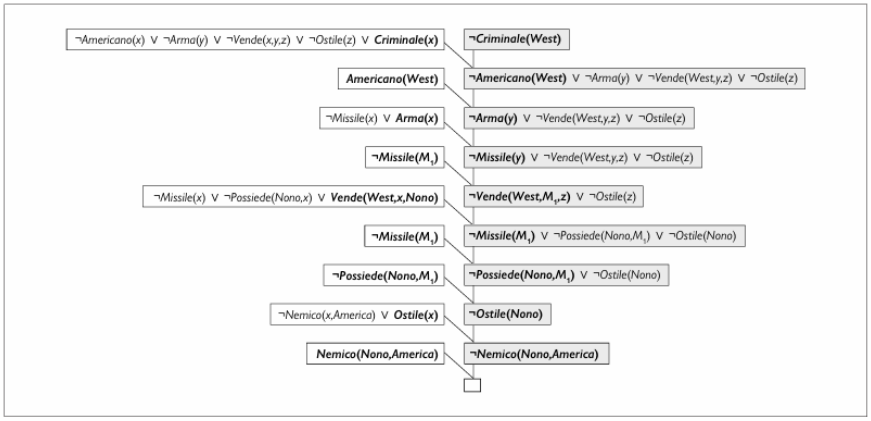

Figura 8.6: Dimostrazione per risoluzione che il Colonnello West è un Criminale. Ad ogni passaggio, i letterali che unificano sono evidenziati in grassetto e la clausola con il letterale positivo ha uno sfondo bianco.