Introduzione

Cos’è l’intelligenza artificiale

Agire umanamente: l’approccio del test di Turing

Il **test di Turing**, proposto da Alan Turing nel 1950, è stato concepito come un esperimento mentale in grado rispondere alla domanda: “Una macchina è in grado di pensare?”. Un computer supererà il test se un esaminatore umano, dopo aver posto alcune domande in forma scritta, non sarà in grado di capire se le risposte provengono da una persona o no. Una macchina in grado di superare il test applicato in modo rigoroso avrebbe bisogno delle seguenti capacità:

- **interpretazione del linguaggio naturale** per comunicare con successo nel linguaggio umano;
- **rappresentazione della conoscenza** per memorizzare quello che conosce o sente;
- **ragionamento automatico** per rispondere alle domande e trarre nuove conclusioni;
- **apprendimento automatico** (*machine learning*) per adattarsi a nuove circostanze, individuare ed estrapolare schemi.

Il **test di Turing totale** richiede l’interazione con oggetti e persone nel mondo reale. Per superare il test di Turing totale, un robot necessiterà anche di:

- **visione artificiale** e riconoscimento vocale per percepire il mondo;
- **robotica** per manipolare gli oggetti e spostarsi fisicamente.

Pensare umanamente: l’approccio della modellazione cognitiva

Ci sono tre modi per pensare nella vita reale:

- l’**introspezione**, ovvero il tentativo di catturare “al volo” i nostri pensieri mentre scorrono;
- la **sperimentazione psicologica**, ovvero l’osservazione di una persona in azione;
- l’**imaging cerebrale**, ovvero l’osservazione del cervello in azione.

Il campo interdisciplinare delle **scienze cognitive** unisce modelli computazionali sviluppati dall’IA e tecniche di sperimentazione psicologica nel tentativo di costruire teorie precise e verificabili sul funzionamento della mente umana.

Agire razionalmente: l’approccio degli agenti razionali

Un **agente** è semplicemente qualcosa che agisce, che fa qualcosa. Un **agente razionale** agisce in modo da ottenere il miglior risultato o, in condizioni di incertezza, il miglior risultato

atteso.

L'*IA si è concentrata sullo studio e la costruzione di agenti che* ***fanno la cosa giusta****.* Che cosa sia la cosa giusta dipende dall'obiettivo fornito all'agente. Questo paradigma generale è talmente pervasivo che potremmo chiamarlo **modello standard**.

Il problema di raggiungere un accordo tra le nostre reali preferenze e l'obiettivo posto nella macchina è detto **problema di allineamento dei valori**: i valori o obiettivi affidati alla macchina devono essere allineati a quelli dell'uomo.

Agenti Intelligenti

Agenti e ambienti

Un agente è qualsiasi cosa possa essere vista come un sistema che percepisce il suo ambiente attraverso dei sensori e agisce su di esso mediante attuatori.
L'ambiente può essere qualsiasi cosa, virtualmente l'universo intero! Ma nella pratica è soltanto la parte dell'universo di cui ci interessa lo stato quando progettiamo l'agente, la parte che influenza ciò che l'agente percepisce e sulla quale influiscono le azioni dell'agente.

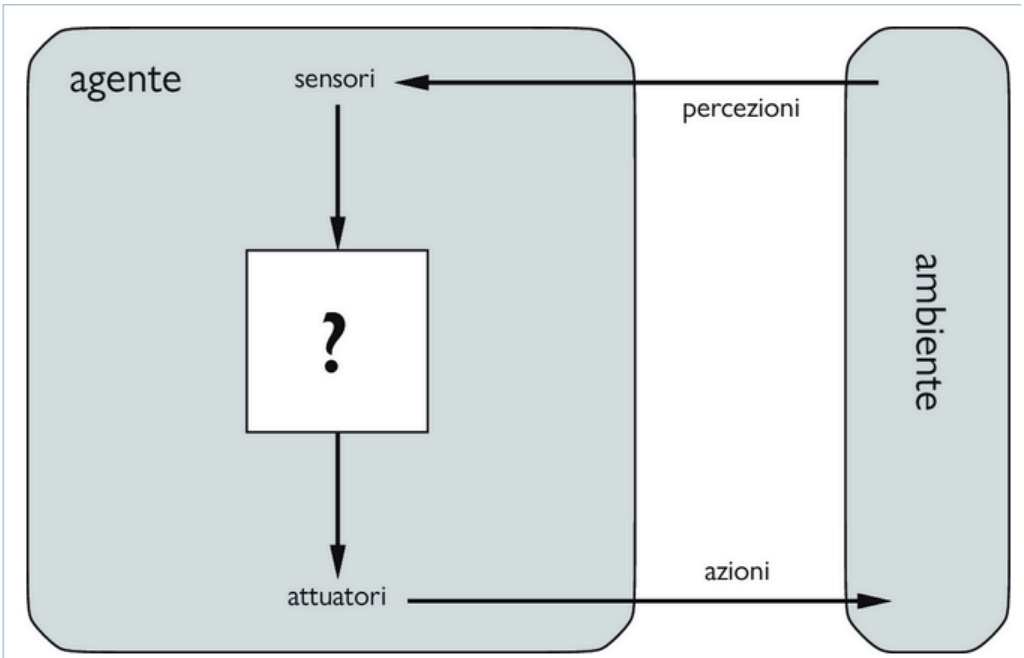


Figura 2.1 Gli agenti interagiscono con l'ambiente attraverso sensori e attuatori.

Useremo il termine percezione (percept) per indicare i dati che i sensori di un agente percepiscono. La sequenza percettiva (percept sequence) di un agente è la storia completa di tutto ciò che esso ha percepito nella sua esistenza.
In generale, la scelta dell'azione di un agente in un qualsiasi istante può dipendere dalla

conoscenza integrata in esso e dall'intera sequenza percettiva osservata fino a quel momento, ma non da qualcosa che l'agente non abbia percepito.
Specificando l'azione prescelta dall'agente per ogni possibile sequenza percettiva, abbiamo descritto l'agente in modo completo. In termini matematici diciamo quindi che il comportamento di un agente è descritto dalla funzione agente, che descrive la corrispondenza tra una qualsiasi sequenza percettiva e una specifica azione.

Possiamo immaginare di rappresentare *in forma di* tabella la funzione agente che descrive un certo agente. Volendo analizzare un agente, in linea di principio è possibile ricostruire la sua tabella provando tutte le possibili sequenze percettive e registrando l'azione prescelta dall'agente come risposta. La tabella, naturalmente, è una descrizione *esterna* dell'agente. *Internamente*, la funzione agente di un agente artificiale sarà implementata da un **programma agente**. La funzione agente è una descrizione matematica astratta; il programma agente è una sua implementazione concreta.

Comportarsi correttamente: il concetto di razionalità

Un **agente razionale** è un agente che fa la cosa giusta.

Misure di prestazione

La cosa giusta fa riferimento alla nozione di consequenzialismo: valutiamo il comportamento di un agente considerandone le *conseguenze.* Quando un agente viene inserito in un ambiente, genera una sequenza di azioni in base alle percezioni che riceve. Questa sequenza di azioni porta l'ambiente ad attraversare una sequenza di stati: se tale sequenza è desiderabile, significa che l'agente si è comportato bene. Questa nozione di desiderabilità è catturata da una **misura di prestazione** che valuta una sequenza di stati dell'ambiente.
La nozione di razionalità per gli umani varia in base al loro punto di vista mentre nelle macchine la nozione di razionalità varia in base al progettista. È difficile definire la misura di prestazione ma come regola generale, è meglio progettare le misure di prestazione in base all'effetto che si desidera ottenere sull'ambiente piuttosto che su come si pensa che debba comportarsi l'agente.

Razionalità

In un dato momento, ciò che è razionale dipende da quattro fattori:

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

Questo porta alla **vera definizione di agente razionale**:

> per ogni possibile sequenza di percezioni, un agente razionale dovrebbe scegliere un'azione che massimizzi il valore atteso della sua misura di prestazione, date le

informazioni fornite dalla sequenza percettiva e da ogni ulteriore conoscenza dell'agente.

Onniscienza, apprendimento e autonomia

Dobbiamo distinguere accuratamente il concetto di razionalità e quello di **onniscienza**. Un agente onnisciente conosce il risultato *effettivo* delle sue azioni e può agire di conseguenza. La razionalità massimizza il risultato *atteso*, mentre la perfezione massimizza quello *reale*. La nostra definizione di razionalità non richiede quindi l'onniscienza, perché la scelta razionale dipende solo dalla sequenza percettiva *fino al momento corrente*.
Intraprendere azioni *mirate a modificare le percezioni future* – ciò che viene talvolta chiamato, con un termine inglese, **information gathering** (raccolta di informazioni) – è una parte importante della razionalità. Un secondo esempio di information gathering è rappresentato dall'**esplorazione** che dev'essere intrapresa da un agente aspirapolvere posto in un ambiente inizialmente sconosciuto.

La nostra definizione richiede che un agente razionale non si limiti a raccogliere informazioni, ma sia anche in grado di **apprendere** il più possibile sulla base delle proprie percezioni. La sua configurazione iniziale potrebbe riflettere una conoscenza pregressa sull'ambiente, che però può modificarsi e aumentare a mano a mano che l'agente accumula esperienza.

Quando un agente si appoggia alla conoscenza pregressa fornita dal progettista invece che alle proprie percezioni e ai suoi processi di apprendimento, diciamo che manca di **autonomia**. Un agente razionale dovrebbe essere autonomo e apprendere il più possibile per compensare la presenza di conoscenza parziale o erronea. Dopo aver accumulato una sufficiente esperienza in un dato ambiente, il comportamento dell'agente razionale può diventare a tutti gli effetti *indipendente* dalla conoscenza pregressa. Incorporare l'apprendimento nel progetto di un agente razionale lo rende capace di operare in una grande varietà di ambienti differenti.

La natura degli ambienti

Specificare un ambiente

Consideriamo ora gli **ambienti operativi**, cioè i "problemi" di cui gli agenti razionali rappresentano le "soluzioni", infatti la misura di prestazione, l'ambiente esterno e gli attuatori e i sensori dell'agente possono essere raggruppati nel termine **ambiente operativo** (*task environment*), o anche **PEAS** (*Performance*, *Environment*, *Actuators*, *Sensors*).

tipo di agente	misura di prestazione	ambiente	attuatori	sensori
guidatore di taxi	sicuro, veloce, ligio alla legge, viaggio confortevole, massimizza i profitti, minimizza l'impatto su altri utenti della strada	strada, altri veicoli nel traffico, polizia, pedoni, clienti, tempo atmosferico	sterzo, acceleratore, freni, frecce, clacson, schermo, voce	telecamere, radar, tachimetro, GPS, sensori del motore, accelerometro, microfoni, touchscreen

Figura 2.4 Descrizione PEAS dell'ambiente operativo di un autista di taxi automatizzato.

Proprietà degli ambienti operativi

Definizioni informali delle Proprietà degli ambienti operativi:

- **Completamente osservabile/parzialmente osservabile**: se i sensori di un agente gli danno accesso allo stato completo dell'ambiente in ogni momento, allora diciamo che l'ambiente operativo è completamente osservabile. In effetti, perché un ambiente operativo sia completamente osservabile basta che i sensori dell'agente misurino tutti gli aspetti che sono *rilevanti* per la scelta dell'azione. Un ambiente potrebbe essere parzialmente osservabile a causa di sensori inaccurati o per la presenza di rumore, o semplicemente perché una parte dei dati non viene rilevata dai sensori. Se l'agente non dispone di sensori, l'ambiente è **inosservabile**.
- **Agente singolo/multiagente**: In un sistema con un agente singolo, c'è un'unica entità che agisce e prende decisioni. In un sistema multiagente, ci sono molteplici agenti che operano simultaneamente. Questi agenti possono collaborare, competere o semplicemente coesistere nell'ambiente.
- **Deterministico/non deterministico/stocastico**: se lo stato successivo dell'ambiente è completamente determinato dallo stato corrente e dall'azione eseguita dall'agente (o dagli agenti), allora si può dire che l'ambiente è deterministico; in caso contrario si dice che è non deterministico. Diciamo che un modello dell'ambiente è stocastico se è esplicitamente associato a probabilità.
- **Episodico/sequenziale**: 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. L'aspetto fondamentale è che un episodio non dipende dalle azioni intraprese in quelli precedenti. Negli ambienti sequenziali, al contrario, ogni decisione può influenzare tutte quelle successive.
- **Statico/dinamico**: se l'ambiente può cambiare mentre un agente sta decidendo come agire, allora diciamo che è dinamico per quell'agente; in caso contrario diciamo che è

statico. Gli ambienti statici sono più facili da trattare perché l'agente non deve continuamente osservare il mondo mentre decide l'azione successiva e non si deve preoccupare del passaggio del tempo. Gli ambienti dinamici, invece, chiedono continuamente all'agente quello che vuole fare. Se l'ambiente stesso non cambia al passare del tempo, ma la valutazione della prestazione dell'agente sì, allora diciamo che l'ambiente è **semidinamico**.

- **Discreto/continuo**: la distinzione tra discreto e continuo si applica allo *stato* dell'ambiente, al modo in cui è gestito il *tempo*, alle *percezioni* e *azioni* dell'agente. Per esempio, la scacchiera ha un numero finito di stati distinti (se si esclude il tempo. Perché con il tempo diventa continuo come il taxi).
- **Noto/ignoto**: in termini rigorosi, questa distinzione non si riferisce all'ambiente in sé, ma allo stato di conoscenza dell'agente (o del progettista) circa le "leggi fisiche" dell'ambiente stesso. In un ambiente noto, sono conosciuti i risultati (o le corrispondenti probabilità, se l'ambiente è stocastico) per tutte le azioni. Ovviamente, se l'ambiente è ignoto, l'agente dovrà apprendere come funziona per poter prendere buone decisioni.

Il caso più difficile è quello *parzialmente osservabile*, *multiagente*, *non deterministico*, *sequenziale*, *dinamico*, *continuo* e *ignoto*.

ambiente operativo	osservabile	agenti	deterministico	episodico	statico	discreto
cruciverba	completamente	singolo	deterministico	sequenziale	statico	discreto
scacchi con orologio	completamente	multi	deterministico	sequenziale	semi	discreto
poker	parzialmente	multi	stocastico	sequenziale	statico	discreto
backgammon	completamente	multi	stocastico	sequenziale	statico	discreto
autista di taxi	parzialmente	multi	stocastico	sequenziale	dinamico	continuo
diagnosi medica	parzialmente	singolo	stocastico	sequenziale	dinamico	continuo

Ambiente: automazione

L'ambiente richiede la simulazione attraverso uno strumento software che si occupa di:
- generare gli stimoli per gli agenti.
- raccogliere le azioni in risposta.
- aggiornare il proprio stato [attivare altri processi implicati da tale cambiamento che influenzano a loro volta l'ambiente].
- valutare le prestazioni degli agenti.

Simulatore

```
function Run-Eval-Environment(state,
                              Update-Fn, agents,
                                          Performance-Fn)
returns scores
local variables: scores  %(vector of size = #agents, all 0)
 repeat
    for each agent in agents do
       Percept[agent] ← Get-Percept(agent, state)
    end
    for each agent in agents do
       Action[agent] ← Program[agent](Percept[agent])
    end
    state ← Update-Fn(actions, agents, state)
    scores ← Performance-Fn(scores, agents, state)
until termination(state)
return scores
```

Simulatore - Prolog

```
start :-      init(InitState),    %inizializzazione unico agente e ambiente, …
              run_env(InitState, 0, Score).

run_env(State, Score, FinScore) :-
  get_percept(Perception), %acquisizione percezioni ag.

  selectAct(State, Perception, Action),   %selezione azione, …
  update_env(State, Action, NewState), %attuazione

  evaluatePerf(NewState, Score, CurrScore),

  check_term(NewState, CurrScore, FinScore).
                                          %check GOAL, …


check_term(State, CS, CS) :- satgoal(State).
check_term(State, CS,0) :-  unsecure(State),
                            writeln('LOST!\n Done!!'),
check_term(State, CS, FS) :-
    run_env(State, CS, FS). %next step
```

La struttura degli agenti

Il compito dell'IA è progettare il **programma agente** che implementa la funzione agente – che fa corrispondere le percezioni alle azioni. Diamo per scontato che questo programma sarà eseguito da un dispositivo computazionale dotato di sensori e attuatori fisici; questa prende il nome di **architettura agente**:

agente = architettura + programma.

Programmi agente

I programmi agente che progetteremo nel corso del libro hanno tutti la stessa struttura: prendono come input la percezione corrente dei sensori e restituiscono un'azione agli attuatori. La differenza tra il programma agente, che prende come input solamente la percezione corrente, e la funzione agente, che potrebbe dipendere dall'intera storia delle percezioni. Il programma agente è costretto a basarsi sulla sola percezione corrente perché l'ambiente non può fornirgli nulla di più; se le sue azioni devono dipendere dalla sequenza percettiva precedente, l'agente stesso dovrà preoccuparsi di memorizzarla.
La tabella rappresenta in modo esplicito la funzione agente implementata dal programma agente. Per costruire un agente razionale con questa tecnica, i progettisti devono costruire una tabella che contenga l'azione appropriata per ogni possibile sequenza percettiva.

L'approccio basato su tabelle esplicite è destinato al fallimento in quanto:
Sia P l'insieme di possibili percezioni e T la durata della vita dell'agente (ovvero il numero totale di percezioni che riceverà). La tabella dovrà contenere $\sum_{t=1}^{T} |P|^t$ righe.
L'enorme dimensione di queste tabelle significa che (a) nessun agente fisico nell'universo avrà mai lo spazio necessario per memorizzarle, (b) il progettista non avrà mai il tempo di crearle e (c) nessun agente avrà mai il tempo di apprendere le righe corrette in base all'esperienza.

> *La sfida principale dell'IA sta nel trovare il modo di scrivere programmi che, nella massima misura possibile, producano un comportamento razionale con una piccola quantità di codice anziché con un'enorme tabella.*

delineeremo quattro tipi base di programma agente che contengono i principî alla base di quasi tutti i sistemi intelligenti:

- agenti reattivi semplici;
- agenti reattivi basati su modello;
- agenti basati su obiettivi;
- agenti basati sull'utilità.

Agenti reattivi semplici

Il tipo più semplice è l'agente reattivo semplice. Questi agenti scelgono le azioni sulla base della percezione *corrente*, ignorando tutta la storia percettiva precedente. La riduzione più importante deriva dall'aver ignorato la storia delle percezioni, cosa che riduce il numero di sequenze percettive rilevanti da P^T a solo P. Possiamo quindi implementarlo con una serie di if-then-else.

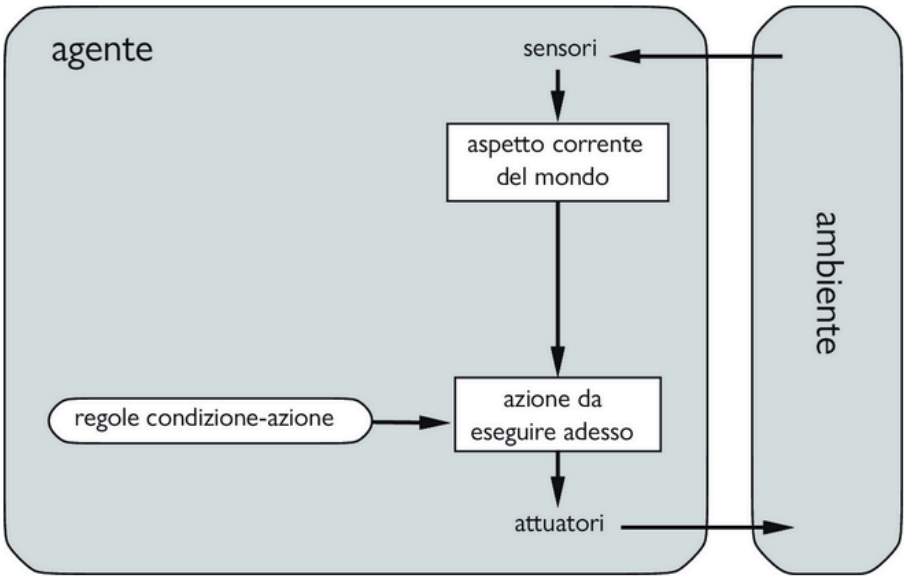


Figura 2.9 Diagramma schematico di un agente reattivo semplice. I rettangoli denotano lo stato interno corrente del processo decisionale dell'agente, mentre gli ovali rappresentano le informazioni di base utilizzate nel processo.

Agenti reattivi - programma

function AGENTE-REATTIVO-SEMPLICE (*percezione*)
returns *azione*
persistent: *regole* %un insieme di regole condizione-azione
stato ← INTERPRETA-INPUT(*percezione*)
regola ← REGOLA-CORRISPONDENTE(*stato, regole*)
azione ← *regola*.AZIONE()
return *azione*

L'agente semplice funzionerà *solo se si può selezionare la decisione corretta in base alla sola percezione corrente, ovvero solo nel caso in cui l'ambiente sia completamente osservabile.* Anche una minima parte di non-osservabilità può causare grandi problemi. Spesso gli agenti reattivi semplici non sono in grado di evitare cicli infiniti quando operano in ambienti parzialmente osservabili. Evitare i cicli infiniti è possibile quando l'agente è in grado di **randomizzare** le sue azioni, scegliendone una in modo casuale.

Agenti reattivi basati su modello

Il modo più efficace di gestire l'osservabilità parziale, per un agente, è *tener traccia della parte del mondo che non può vedere nell'istante corrente*. Questo significa che l'agente deve mantenere una sorta di stato interno che dipende dalla storia delle percezioni e che quindi riflette almeno una parte degli aspetti non osservabili dello stato corrente. Aggiornare l'informazione dello stato interno al passaggio del tempo richiede che il programma agente possieda due tipi di conoscenza. Prima di tutto, deve avere informazioni sull'evoluzione del mondo nel tempo, suddivisibili approssimativamente in due parti: gli effetti delle azioni dell'agente e le modalità di evoluzione del mondo indipendentemente dall'agente. Questa conoscenza sul "funzionamento del mondo", implementata mediante semplici circuiti logici o sviluppata in una teoria scientifica completa, viene chiamata **modello di transizione** del mondo. In secondo luogo, ci servono in formazioni su come lo stato del mondo si rifletta nelle percezioni dell'agente. Questo tipo di conoscenza è chiamato **modello sensoriale**.

Il modello di transizione e il modello sensoriale, insieme, consentono a un agente di tenere traccia dello stato del mondo, per quanto possibile date le limitazioni dei sensori. Un agente che utilizza tali modelli prende il nome di **agente basato su modello**.

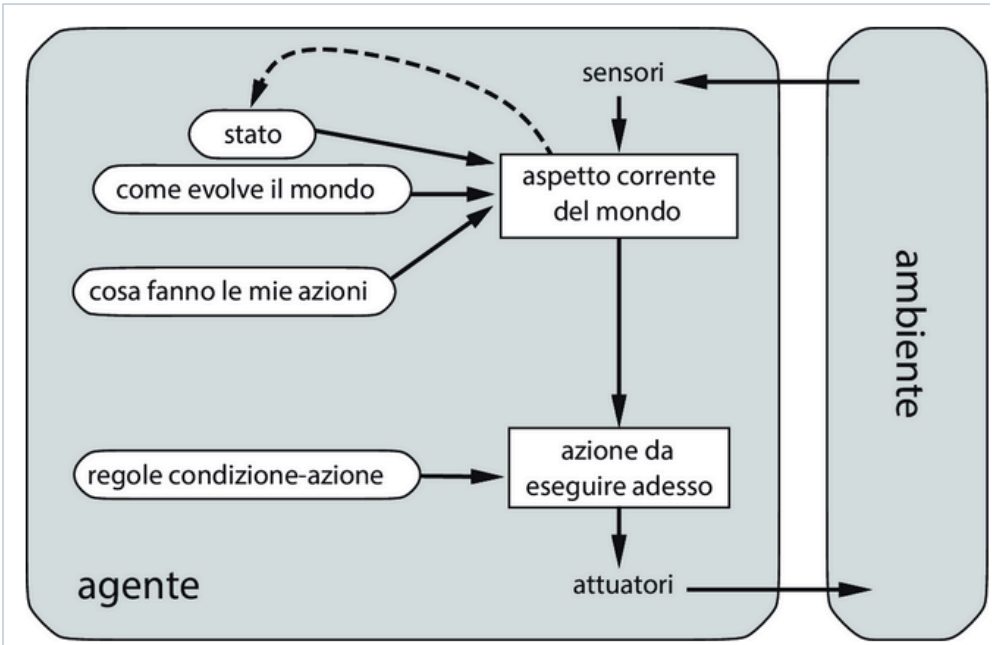


Figura 2.11 Un agente reattivo basato su modello.

Agenti basati su modello

function AGENTE-BASATO-SU-MODELLO (*percezione*)
returns *azione*

persistent: *stato*, %una descrizione dello stato corrente
modello, %conoscenza del mondo
regole, %un insieme di regole condizione-azione
azione, %l'azione più recente

stato ← AGGIORNA-STATO(*stato, azione, percezione, modello*)
regola ← REGOLA-CORRISPONDENTE(*stato, regole*)
azione ← *regola*.AZIONE()

return *azione*

L'incertezza riguardo lo stato corrente potrebbe essere inevitabile, ma l'agente deve comunque prendere una decisione.

Agenti basati su obiettivi

Conoscere lo stato corrente dell'ambiente non sempre basta a decidere che cosa fare. Oltre che della descrizione dello stato corrente l'agente ha bisogno di qualche tipo di informazione riguardante il suo **obiettivo** (*goal*), che descriva situazioni desiderabili.

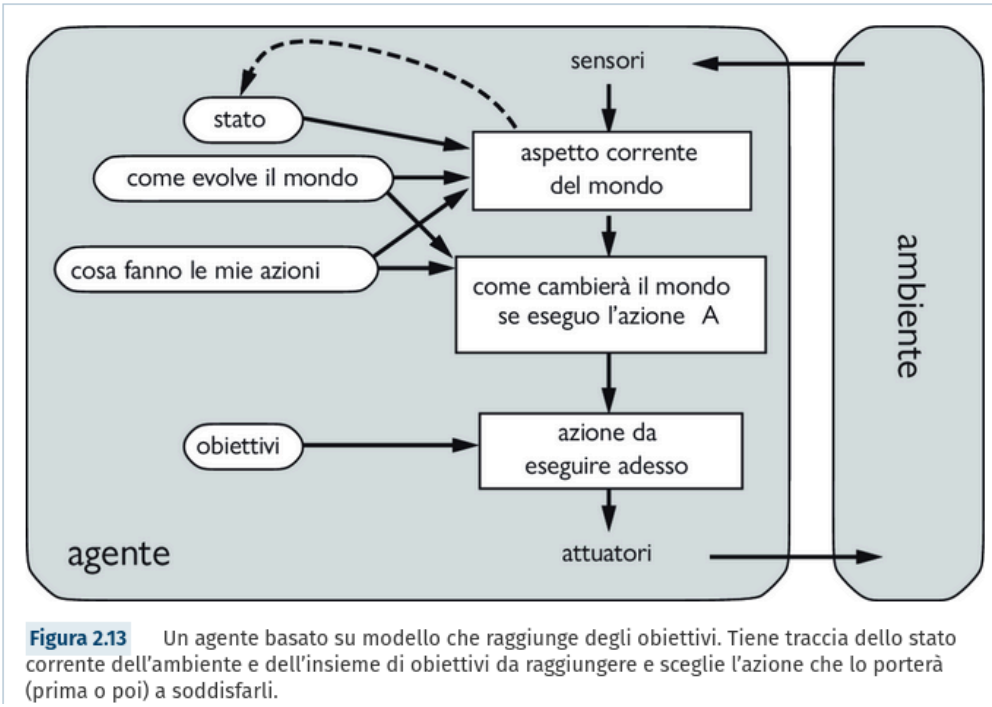


Figura 2.13 Un agente basato su modello che raggiunge degli obiettivi. Tiene traccia dello stato corrente dell'ambiente e dell'insieme di obiettivi da raggiungere e sceglie l'azione che lo porterà (prima o poi) a soddisfarli.

Talvolta scegliere un'azione in base a un obiettivo è molto semplice, quando questo può essere

raggiunto in un solo passo. Altre volte è più difficile, per esempio quando l'agente deve considerare lunghe sequenze di azioni per trovare il "cammino" che porta al risultato agognato. La **ricerca** e la **pianificazione** sono sottocampi dell'IA dedicati proprio a identificare le sequenze di azioni che permettono a un agente di raggiungere i propri obiettivi. Notate che questo tipo di decisioni non ha nulla a che vedere con le regole condizione–azione che abbiamo descritto prima, perché ora dobbiamo prendere in considerazione il futuro sotto due aspetti: "cosa accadrà se faccio così e cosà?" e anche "se faccio questo sarò soddisfatto?" Nella progettazione degli agenti reattivi, questa informazione non viene rappresentata esplicitamente, perché le regole fornite internamente mettono direttamente in corrispondenza percezioni e azioni. Benché un agente basato su obiettivi sembri meno efficiente, d'altra parte è più flessibile, perché la conoscenza che guida le sue decisioni è rappresentata esplicitamente e può essere modificata.

Agenti basati sull'utilità

Nella maggior parte degli ambienti gli obiettivi, da soli, non bastano a generare un comportamento di alta qualità. Gli obiettivi forniscono solamente una distinzione binaria tra stati utili e non utili, laddove una misura di prestazione più generale dovrebbe permettere di confrontare stati del mondo differenti e misurare precisamente la contentezza che potrebbero portare all'agente. Abbiamo già visto che una misura di prestazione assegna un punteggio a qualsiasi sequenza di stati dell'ambiente, così da poter facilmente distinguere tra modi più o meno desiderabili di raggiungere l'obiettivo. Una **funzione di utilità** di un agente è, in sostanza, un'internalizzazione della misura di prestazione. Purché la funzione di utilità interna e la misura di prestazione esterna concordino, un agente che sceglie le azioni per massimizzare l'utilità sarà razionale in base alla misura di prestazione esterna.

Un agente basato sull'utilità presenta molti vantaggi in termini di flessibilità e apprendimento. Inoltre per un agente basato su obiettivi, gli obiettivi possono essere inadeguati mentre un agente basato sull'utilità è in grado di prendere decisioni razionali. Ad esempio, quando ci sono più obiettivi in conflitto che non si possono soddisfare tutti insieme (per esempio, velocità e sicurezza), la funzione di utilità specifica come bilanciarli. In secondo luogo, quando ci sono più obiettivi raggiungibili ma nessuno può essere ottenuto con certezza, il concetto di utilità fornisce un mezzo per confrontare le probabilità di successo e l'importanza degli obiettivi.

Situazioni di parziale osservabilità e non determinismo sono molto diffuse nel mondo reale, e di conseguenza anche i casi in cui occorre prendere decisioni in condizioni di incertezza sono comuni. In termini tecnici, un agente razionale basato sull'utilità sceglie l'azione che massimizza l'**utilità attesa** dei risultati, ovvero l'utilità che l'agente si attende di ottenere, in media, date le probabilità e le utilità di ciascun risultato. Un agente che possiede una funzione di utilità *esplicita* può prendere decisioni razionali, seguendo un algoritmo generale che non dipende dalla specifica funzione di utilità che si desidera massimizzare.

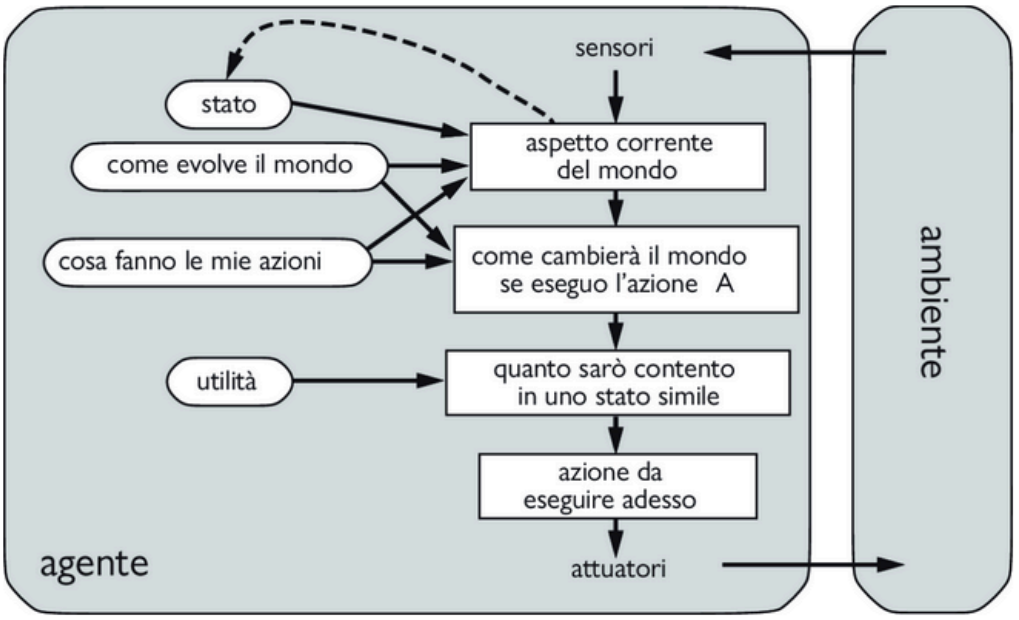


Figura 2.14 Un agente basato su modello che massimizza l'utilità. Oltre a tener traccia dello stato dell'ambiente, l'agente impiega una funzione utilità che misura le sue preferenze tra i vari stati del mondo. L'azione prescelta è quella che massimizza l'utilità attesa, calcolata come media dei valori di utilità degli stati possibili pesati con la rispettiva probabilità di verificarsi.

Agenti capaci di apprendere

Fin qui abbiamo descritto programmi agente che usano vari metodi per scegliere le azioni. Non abbiamo ancora spiegato in che modo *nascono* i programmi agente. Qualsiasi tipo di agente (basato su modello, su obiettivi, su utilità e così via) può essere costruito come agente capace di apprendere (o meno). L'apprendimento presenta un altro vantaggio: permette agli agenti di operare in ambienti inizialmente sconosciuti, diventando col tempo più competenti di quanto fossero all'inizio, allorché si basavano sulla sola conoscenza iniziale.

Un agente capace di apprendere può essere diviso in quattro componenti astratti:
- **elemento di apprendimento** (*learning element*): responsabile del miglioramento interno.
- elemento esecutivo (*performance element*): che si occupa della selezione delle azioni esterne. Il progetto dell'elemento di apprendimento dipende molto da quello dell'elemento esecutivo.
- **elemento critico:** riguarda le prestazioni correnti dell'agente e determina se e come modificare l'elemento esecutivo affinché in futuro si comporti meglio (l'elemento di apprendimento utilizza informazioni provenienti da questo elemento).
- **generatore di problemi**: il cui scopo è suggerire azioni che portino a esperienze nuove e significative. Se si lasciasse mano libera all'elemento esecutivo, esso continuerebbe a ripetere le azioni che ritiene migliori date le conoscenze attuali. Ma se l'agente è disposto a esplorare qualche altra possibilità, e magari eseguire nel breve termine qualche azione subottima, potrebbe scoprire l'esistenza di azioni migliori a lungo termine. Scopo del generatore di problemi è suggerire tali azioni esplorative. L'osservazione di coppie di stati successivi

dell'ambiente permette all'agente di imparare "cosa fanno le mie azioni" e "come evolve il mondo" a seguito delle sue azioni.

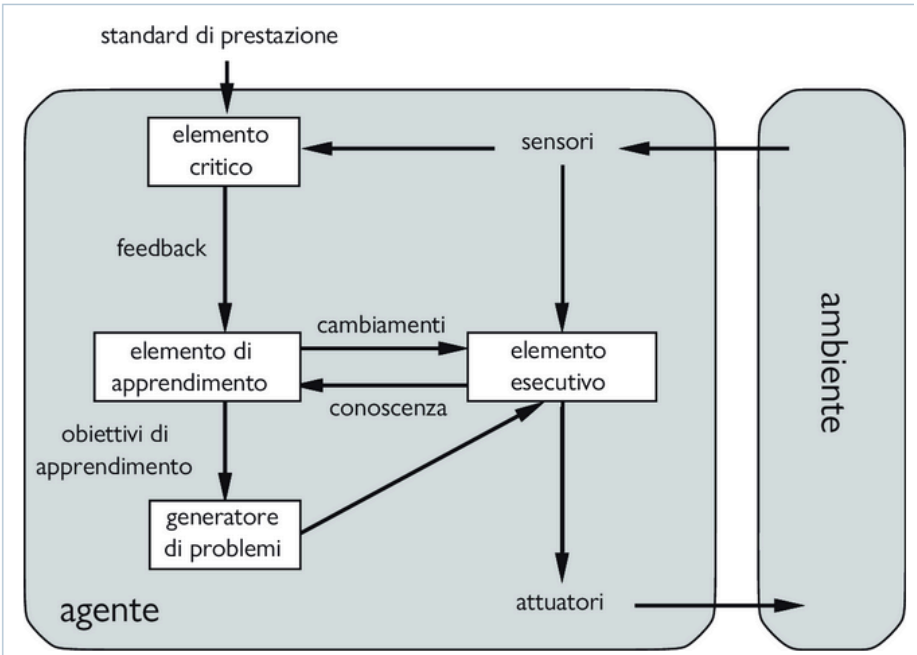
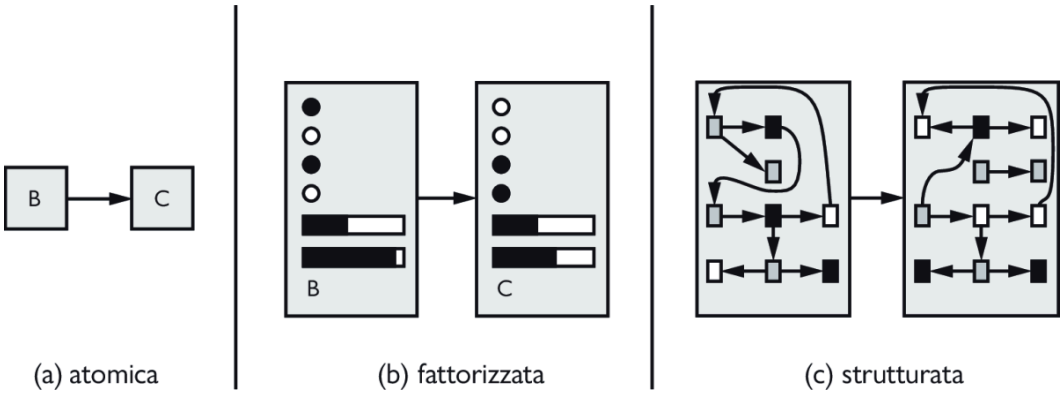


Figura 2.15 Un modello generale di agente capace di apprendere. Il riquadro "elemento esecutivo" rappresenta ciò che in precedenza abbiamo considerato come l'intero programma agente. Ora, il riquadro "elemento di apprendimento" può modificare il programma per migliorarne la prestazione.

Funzionamento dei componenti dei programmi agente

In una rappresentazione atomica ogni stato del mondo è indivisibile, non ha struttura interna.
Una rappresentazione fattorizzata suddivide ogni stato in un insieme fissato di variabili o attributi, ognuno dei quali può avere un valore.
Una rappresentazione strutturata permette di descrivere in modo esplicito oggetti come mucche e camion insieme alle loro relazioni.



Risolvere i problemi con la ricerca

Quando l'azione giusta da compiere non è subito evidente, un agente può avere la necessità di *guardare in avanti*, cioè considerare una *sequenza* di azioni che formano un cammino che porterà a uno stato obiettivo. Questo tipo di agente è chiamato **agente risolutore di problemi** e il processo computazionale che effettua è una **ricerca**. Gli agenti risolutori di problemi utilizzano rappresentazioni **atomiche** in cui gli stati del mondo sono considerati come entità prive di una struttura interna visibile agli algoritmi per la risoluzione dei problemi. Gli agenti che utilizzano rappresentazioni di stati **fattorizzate** o **strutturate** sono solitamente chiamati **agenti pianificatori**.

Consideriamo soltanto gli ambienti più semplici: episodici, a singolo agente, completamente osservabili, deterministici, statici, discreti e noti. Distingueremo tra algoritmi **informati**, in cui l'agente è in grado di stimare la distanza dall'obiettivo, e **non informati**, in cui non c'è la disponibilità di tale stima.

Agenti risolutori di problemi

Immaginiamo un agente che si trova in vacanza in Romania e vuole visitare le città. Il problema decisionale è complesso, come fa un agente a capire dove andare da una città ad un altra. Se l'agente non ha altre informazioni, cioè se l'ambiente è **ignoto**, non può fare altro che eseguire una delle azioni scelte a caso. L'agente può eseguire un processo di risoluzione del problema in quattro fasi.

- **Formulazione dell'obiettivo**: l'agente adotta l'**obiettivo** di raggiungere Bucarest. Gli obiettivi aiutano a organizzare il comportamento limitando gli scopi e quindi le azioni da considerare.
- **Formulazione del problema**: l'agente elabora una descrizione degli stati e delle azioni necessarie per raggiungere l'obiettivo, ovvero un modello astratto della parte del mondo interessata. Per il nostro agente, un buon modello consiste nel considerare le azioni di viaggiare da una città a un'altra città adiacente.
- **Ricerca**: prima di effettuare qualsiasi azione nel mondo reale, l'agente simula nel suo modello sequenze di azioni, continuando a cercare finché trova una sequenza che raggiunge l'obiettivo: tale sequenza si chiama **soluzione**. L'agente potrebbe simulare più sequenze che non raggiungono l'obiettivo, ma alla fine troverà una soluzione oppure troverà che non è possibile alcuna soluzione.
- **Esecuzione**: l'agente ora può eseguire le azioni specificate nella soluzione, una per volta.

È importante osservare che in un ambiente completamente osservabile, deterministico e noto, *la soluzione di qualsiasi problema è una sequenza fissata di azioni*. Se il modello è corretto, una volta che l'agente ha trovato una soluzione, può ignorare le sue percezioni mentre esegue le azioni dato che ha la garanzia che la soluzione lo condurrà all'obiettivo. Nella teoria del controllo

si parla in questo caso di sistema **ad anello aperto**, perché ignorando le percezioni si rompe il ciclo tra agente e ambiente. Se vi è la possibilità che il modello sia errato, o che l'ambiente non sia deterministico, l'agente sarebbe più sicuro usando un approccio **ad anello chiuso** che tiene monitorate le percezioni.

In ambienti parzialmente osservabili o non deterministici, una soluzione sarebbe una strategia ramificata che consigliasse diverse azioni future in base alle percezioni raccolte.

Problemi di ricerca e soluzioni

Un **problema** di ricerca può essere definito formalmente come segue:

- Un insieme di possibili **stati** in cui può trovarsi l'ambiente. Lo chiamiamo **spazio degli stati**.
- Lo **stato iniziale** in cui si trova l'agente inizialmente.
- Un insieme di uno o più **stati obiettivo**.
- Le **azioni** possibili dell'agente. Dato uno stato *s*, Azioni(*s*) restituisce un insieme finito di azioni che possono essere eseguite in *s*. Diciamo che ognuna di queste azioni è **applicabile** in *s*.
  Es: Azioni(*Arad*) = {*VersoSibiu*, *VersoTimisoara*, *VersoZerind*}.
- Un **modello di transizione** che descrive ciò che fa ogni azione. Risultato(*s*, *a*) restituisce lo stato risultante dall'esecuzione dell'azione *a* nello stato *s*.
  Es: Risultato(*Arad*, *VersoZerind*) = *Zerind*
- Una **funzione di costo dell'azione**, denotata da Costo-Azione(*s*, *a*, *s*′) nei programmi o *c*(*s*, *a*, *s*′) nei calcoli matematici, che restituisce il costo numerico di applicare l'azione *a* nello stato *s* per raggiungere lo stato *s*′.

Una sequenza di azioni forma un **cammino**; una **soluzione** è un cammino che porta dallo stato iniziale a uno stato obiettivo. Una **soluzione ottima** è, tra tutte le possibili soluzioni, quella che ha il costo minimo. Lo spazio degli stati può essere rappresentato come un **grafo** in cui i vertici (o nodi) rappresentano gli stati e i collegamenti orientati (archi) tra di essi rappresentano le azioni.

La formulazione dei problemi

La nostra formulazione del problema di arrivare a una determinata città è un **modello**, cioè una descrizione matematica astratta, e non qualcosa di reale. Il processo di rimozione dei dettagli da una rappresentazione prende il nome di **astrazione**. Per una buona formulazione del problema serve il giusto livello di dettaglio. L'astrazione è *valida* se possiamo espandere ogni soluzione astratta in una soluzione nel mondo più dettagliato; L'astrazione è *utile* se eseguire ogni azione nella soluzione è più facile che nel problema originale;

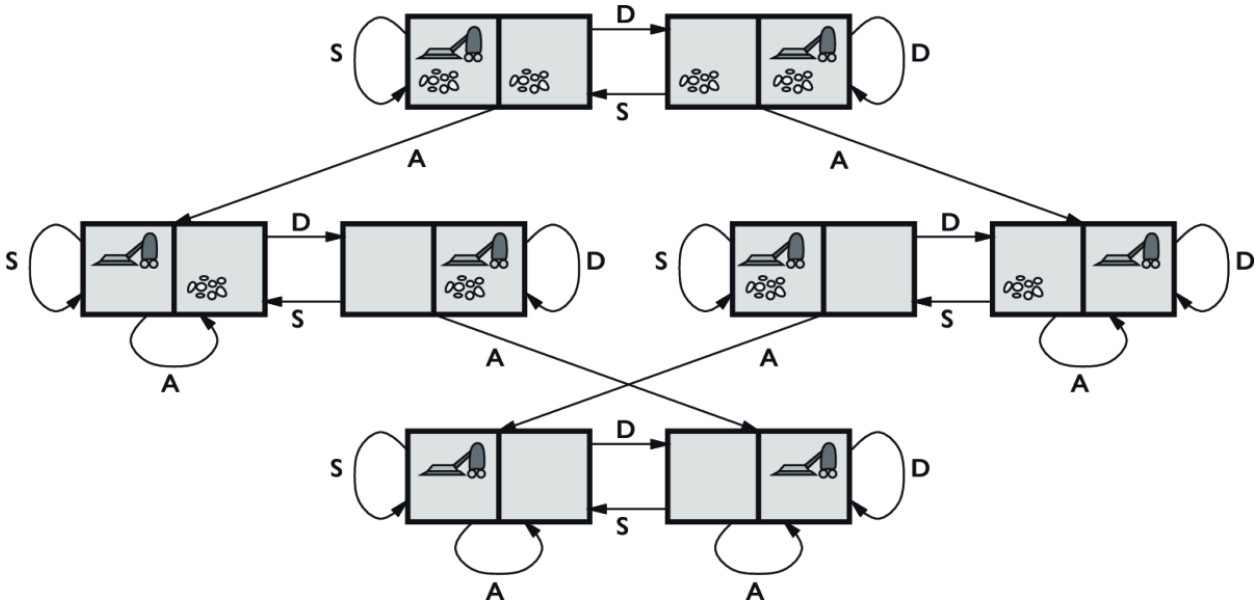
Problemi esemplificativi

Lo scopo di un **problema standardizzato** è illustrare o mettere alla prova diversi metodi di risoluzione di problemi. Un **problema del mondo reale**, è un problema le cui soluzioni sono effettivamente utili alle persone e la cui formulazione è specifica, non standardizzata.

Problemi standardizzati

Un esempio del problema standardizzato (su griglia) è quello dell'aspirapolvere:

- **Stati**: uno stato del mondo indica quali oggetti sono in quali celle. Nel mondo dell'aspirapolvere gli oggetti sono l'agente e lo sporco. In generale, un mondo dell'aspirapolvere con n celle ha $n \times 2^n$ stati.
- **Stato iniziale**: ogni stato può essere designato come stato iniziale.
- **Azioni**: nel mondo a due celle abbiamo definito tre azioni: *Sinistra*, *Destra* e *Aspira*.
- **Modello di transizione**: l'azione *Aspira* rimuove lo sporco dalla cella dove si trova l'agente; *Avanti* muove l'agente di una cella nella direzione verso cui è orientato, a meno che non abbia davanti un muro, nel qual caso l'azione non ha effetto.
- **Stati obiettivo**: gli stati in cui ogni cella è pulita.
- **Costo di azione**: ogni azione costa 1.



Il grafo che rappresenta lo spazio degli stati per il mondo dell'aspirapolvere a due celle. Ci sono 8 stati e 3 azioni per ciascuno stato: S = *Sinistra*, D = *Destra*, A = *Aspira*.

Un esempio di problema reale è il **problema di ricerca dell'itinerario**:

- **Stati**: ognuno ovviamente comprende una posizione (per esempio un aeroporto) e l'ora corrente. Inoltre, poiché il costo di un'azione (un tratto di volo) può dipendere da

eventuali tratte precedenti, dalle loro tariffe e dal loro stato di tratte nazionali o internazionali, lo stato deve registrare altre informazioni su questi aspetti “storici”.
- **Stato iniziale**: l’aeroporto da dove parte l’utente.
- **Azioni**: prendere un volo dalla posizione corrente, in una classe qualsiasi, partendo dopo l’ora corrente, lasciando tempo sufficiente per il trasferimento all’interno dell’aeroporto, se necessario.
- **Modello di transizione**: lo stato risultante dal prendere un volo avrà come nuova posizione la destinazione del volo e come ora corrente quella di arrivo del volo.
- **Stato obiettivo**: una città di destinazione.
- **Costo di azione**: una combinazione di costo monetario, tempi di attesa, durata dei voli, procedure di dogana, qualità dei posti a sedere, ora del giorno, tipo di aereo, punti per programmi frequent flyer e così via.

Problemi reali

Altri problemi reali sono:
I **problemi di viaggio** descrivono una serie di località che devono essere visitate, anziché una singola destinazione obiettivo.
Il **problema del commesso viaggiatore** (TSP, *traveling salesperson problem*) è un problema di viaggio in cui ogni città presente su una mappa va visitata.
Un problema di **configurazione VLSI** richiede che vengano posizionati al meglio i milioni di componenti e di connessioni che formano un chip in modo da minimizzare l’area del chip stesso.
La **navigazione dei robot**.
Sequenze di montaggio automatico di oggetti complessi (come i motori elettrici) da parte di un robot.

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.

Algoritmi di ricerca

Un **algoritmo di ricerca** riceve in input un problema di ricerca e restituisce una soluzione o un’indicazione di fallimento. In questo capitolo consideriamo 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 di ricerca corrisponde a uno stato nello spazio degli stati e i rami dell’albero di ricerca corrispondono ad azioni. La radice dell’albero corrisponde allo stato iniziale del problema.

È importante comprendere la distinzione tra spazio degli stati e albero di ricerca. Lo spazio degli stati descrive l’insieme (eventualmente infinito) degli stati nel mondo e le azioni che consentono le transizioni da uno stato a un altro. L’albero di ricerca descrive cammini tra questi stati per raggiungere l’obiettivo.

Possiamo **espandere** il nodo, considerando le Azioni disponibili per quello stato, usando la

funzione Risultato per vedere dove portano tali azioni e **generare** un nuovo nodo (chiamato **nodo figlio** o **nodo successore**) per ognuno degli stati risultanti.

Ora dobbiamo scegliere quale dei nodi figli considerare. Questa è l'essenza della ricerca: seguire un'opzione e mettere per il momento da parte le altre, pronti a riprenderle in seguito chiamiamo l'insieme di questi nodi **frontiera** dell'albero di ricerca, e diciamo che ogni stato per cui vi è un nodo generato è stato **raggiunto** (indipendentemente dal fatto che tale nodo sia stato espanso o meno). Notate che la frontiera **separa** due regioni del grafo dello spazio degli stati: una regione interna in cui ogni stato è stato espanso e una esterna di stati che non sono ancora stati raggiunti.

Ricerca best-first (*)

Come decidiamo quale nodo della frontiera espandere? Un approccio molto generale è quello della **ricerca best-first** (letteralmente "prima il migliore"), in cui scegliamo un nodo *n* che corrisponde al valore minimo di una **funzione di valutazione** *f*(*n*). A ogni iterazione scegliamo un nodo sulla frontiera in cui *f*(*n*) ha valore minimo e lo restituiamo se il suo stato è uno stato obiettivo, altrimenti applichiamo Espandi per generare nodi figli. Ogni nodo figlio viene aggiunto alla frontiera se non è stato raggiunto in precedenza, o viene aggiunto di nuovo se ora è stato raggiunto con un cammino di costo inferiore ai precedenti. L'algoritmo restituisce un'indicazione di fallimento oppure un nodo che rappresenta un cammino che porta a un obiettivo.

Strutture dati per la ricerca

Gli algoritmi di ricerca richiedono una struttura dati per tenere traccia dell'albero di ricerca. Un **nodo** nell'albero di ricerca è rappresentato da una struttura dati con quattro componenti:

- *n*.Stato: lo stato a cui corrisponde il nodo;
- *n*.Padre: il nodo dell'albero di ricerca che ha generato il nodo corrente;
- *n*.Azione: l'azione applicata allo stato del padre per generare il nodo corrente;
- *n*.Costo-Cammino: il costo totale del cammino che va dallo stato iniziale al nodo corrente. Nelle formule matematiche utilizziamo *g*(*nodo*) come sinonimo di Costo-Cammino.

Seguendo i puntatori Padre a partire da un nodo possiamo risalire agli stati e alle azioni lungo il cammino fino a tale nodo. Procedendo in questo modo a partire da un nodo obiettivo troviamo la soluzione.

Ci serve una struttura dati per memorizzare la **frontiera**. La scelta più appropriata è una **coda**. Negli algoritmi di ricerca si utilizzano tre tipi di code:

- la **coda con priorità**, in cui viene estratto prima il nodo di costo minimo in base a una funzione di valutazione *f*. Questo tipo di coda è usato nella ricerca best-first;
- la **coda FIFO** o *First-In, First-Out* (il primo che entra è il primo a uscire), in cui viene estratto prima il nodo che è stato aggiunto alla coda per primo. (usato nella ricerca in

ampiezza);
- la **coda LIFO** o *Last-In, First-Out* (l'ultimo che entra è il primo a uscire), detta anche **stack**, in cui viene estratto il nodo che è stato aggiunto alla coda per ultimo. (usato nella ricerca in profondità..

Gli stati raggiunti possono essere memorizzati come tabella di lookup (per esempio una tabella hash) in cui ogni chiave è uno stato e ogni valore è il nodo per tale stato.

Cammini ridondanti

L'albero di ricerca completo è *infinito* perché non vi è limite al numero di volte per cui è possibile percorrere un ciclo. Un ciclo è un caso particolare di **cammino ridondante** (cammini con stesso padre e stessa foglia).
Come fa un algoritmo a ricordarsi del passato? Questo problema può essere affrontato con tre approcci:

- Il primo approccio consiste nel ricordare tutti gli stati precedentemente raggiunti (come avviene nella ricerca best-first), il che ci consente di individuare tutti i cammini ridondanti e mantenere soltanto il cammino migliore per raggiungere ogni stato. Questo approccio è adatto per spazi degli stati in cui vi sono molti cammini ridondanti, ed è la scelta preferita quando c'è spazio di memoria sufficiente per contenere la tabella degli stati raggiunti.
- Il secondo approccio consiste nel non preoccuparsi di ripetere il passato. Esistono alcune formulazioni di problemi in cui è raro o impossibile che due cammini conducano allo stesso stato. Per questi problemi possiamo risparmiare spazio in memoria se *non* tracciamo gli stati raggiunti e non controlliamo la presenza di cammini ridondanti. Un algoritmo di ricerca è detto di **ricerca su grafo** se controlla la presenza di cammini ridondanti e **ricerca ad albero** se non esegue tale controllo.
- Il terzo approccio è un compromesso: controlliamo la presenza di cammini ciclici, ma non di cammini ridondanti. Poiché ogni nodo ha una catena di puntatori padre, possiamo controllare la presenza di cammini ciclici senza la necessità di usare memoria aggiuntiva risalendo la catena dei padri in modo da verificare se lo stato al termine del cammino è già apparso precedentemente.

Misurare le prestazioni nella risoluzione di problemi

Prima di esaminare la progettazione di vari algoritmi di ricerca, consideriamo i criteri usati per sceglierli. Possiamo valutare le prestazioni di un algoritmo secondo quattro parametri.

- **Completezza**: l'algoritmo garantisce di trovare una soluzione, se questa esiste, e di riportare correttamente il fallimento, se la soluzione non esiste. Per essere completo, un algoritmo di ricerca deve procedere in modo **sistematico** nell'esplorazione di uno spazio degli stati infinito, assicurandosi di poter raggiungere qualsiasi stato collegato a quello iniziale.
- **Ottimalità rispetto al costo**: trova la soluzione con il costo di cammino minimo fra tutte.

- **Complessità temporale**: quanto tempo impiega a trovare una soluzione? Il tempo può essere misurato in secondi o, in modo più astratto, in base al numero di stati e azioni considerato.
- **Complessità spaziale**: di quanta memoria necessita per effettuare la ricerca.

Strategie di ricerca non informata

Un algoritmo di ricerca non informata non riceve alcuna informazione su quanto uno stato sia vicino all'obiettivo.

Ricerca in ampiezza

Quando tutte le azioni hanno lo stesso costo, una strategia appropriata è la ricerca in ampiezza, si tratta di una ricerca sistematica, quindi completa anche su spazi degli stati infiniti.

Potremmo implementare la ricerca in ampiezza come chiamata di Ricerca-Best-First con la funzione di valutazione $f(n)$ è uguale alla profondità del nodo, cioè al numero di azioni necessario per raggiungerlo. Una coda FIFO (*first-in-first-out*) risulterà più veloce di una coda con priorità: i nuovi nodi vanno in fondo alla coda e quelli vecchi vengono espansi per primi. Inoltre, *raggiunti* può essere un insieme di stati, anziché una corrispondenza da stati a nodi, perché una volta raggiunto uno stato, non possiamo più trovare un cammino migliore per raggiungerlo. . Questo significa anche che possiamo effettuare un test obiettivo anticipato, controllando se un nodo è una soluzione non appena viene *generato*, invece di effettuare il test obiettivo a posteriori come nella ricerca best-first, aspettando che un nodo sia estratto dalla coda.

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 di profondità $d - 1$, perciò se uno di essi fosse una soluzione, sarebbe stato trovato. Ciò significa che è ottimale rispetto al costo per problemi in cui tutte le azioni hanno lo stesso costo, ma non per problemi che non hanno tale proprietà.

Supponiamo che la soluzione sia a profondità d, poiché ogni nodo ha al più b figli allora il numero totale di nodi generati è:

$$1 + b + b_2 + b_3 + \cdots + b^d = O(b^d).$$

Tutti i nodi rimangono in memoria, perciò le complessità temporale e spaziale sono entrambe $O(b_d)$, quindi esponenziali, cosa non ottimale. In generale, *i problemi di ricerca con complessità esponenziale non possono essere risolti mediante ricerche non informate, se non nelle istanze più piccole.*

Algoritmo di Dijkstra o ricerca a costo uniforme

Quando le azioni hanno costi diversi, la scelta ovvia è quella di usare una ricerca best-first in cui la funzione di valutazione è il costo del cammino dalla radice al nodo corrente. Questa ricerca (Algoritmo di Dijkstra) si chiama **ricerca a costo uniforme**. L'idea è che mentre la ricerca in ampiezza si diffonde a ondate di profondità uniforme, la ricerca a costo uniforme si diffonde a ondate di costo uniforme. L'algoritmo può essere implementato come chiamata di Ricerca-Best-First con Costo-Cammino come funzione di valutazione.

La complessità della ricerca a costo uniforme è caratterizzata in termini di C^*, il costo della soluzione ottima, ed ε, un limite inferiore imposto al costo di ogni azione, con $\varepsilon > 0$. Quindi, nel caso peggiore la complessità temporale e spaziale dell'algoritmo è $O(b^{1+C^*/\varepsilon})$, che può essere molto maggiore di b^d. Ciò avviene perché la ricerca a costo uniforme può esplorare grandi alberi di azioni con costi bassi prima di esplorare cammini che comportano un costo elevato e forse azioni utili. Quando tutte le azioni hanno lo stesso costo, $b^{1+C^*/\varepsilon}$ è b^{d+1} e la ricerca a costo uniforme è simile alla ricerca in ampiezza.

La ricerca a costo uniforme è completa e ottima rispetto al costo, perché la prima soluzione che trova avrà un costo basso almeno quanto quello di ogni altro nodo sulla frontiera. Questa ricerca esamina sistematicamente tutti i cammini in ordine di costo crescente, quindi non entra mai in un cammino infinito (assumendo che tutti i costi delle azioni siano $> \varepsilon > 0$).

Ricerca in profondità e problema della memoria

La ricerca in profondità espande sempre per primo il nodo *a profondità maggiore* nella frontiera, si implementa tramite una pila (LIFO). Potrebbe essere implementata con una chiamata di Ricerca-Best-First in cui la funzione di valutazione *f* è l'opposto (negativo) della profondità. La ricerca in profondità non è ottima rispetto al costo, infatti restituisce sempre la prima soluzione che trova anche se non è la meno costosa.

Per spazi degli stati finiti che sono alberi, la ricerca in profondità è efficiente e completa; per spazi degli stati aciclici potrebbe arrivare a espandere lo stesso stato più volte attraverso cammini diversi, ma (alla fine) esplorerà sistematicamente l'intero spazio.

In spazi degli stati ciclici, la ricerca in profondità può bloccarsi in un ciclo infinito, perciò alcune implementazioni controllano ciascun nodo nuovo per verificare la presenza di cicli. Infine, in spazi degli stati infiniti la ricerca in profondità non è sistematica: può entrare in un cammino infinito, anche se non vi sono cicli. Quindi, la ricerca in profondità è incompleta.

Per i problemi in cui è praticabile una ricerca ad albero, la ricerca in profondità ha esigenze di memoria molto più ridotte. Non viene mantenuta una tabella dei nodi raggiunti, e la frontiera è molto piccola. Per uno spazio degli stati finito e a forma di albero, una ricerca ad albero in profondità richiede un tempo proporzionale al numero degli stati e ha una complessità di memoria solo $O(bm)$, dove b è il fattore di ramificazione e m è la massima profondità dell'albero.

Alcuni problemi troppo costosi con la ricerca in ampiezza possono essere affrontati usando la ricerca in profondità, grazie al fatto che richiede poca memoria.

(*) Esiste una variante della ricerca in profondità, la **ricerca con backtracking**. Nel backtracking, quando si espande un nodo non vengono generati tutti i successori, bensì uno solo; ogni nodo parzialmente espanso ricorda quale successore deve essere generato in seguito. In più, i successori sono generati *modificando* la descrizione dello stato corrente anziché allocando memoria per uno stato nuovo, e questo riduce i requisiti di memoria a una descrizione di stato e un cammino di $O(m)$ azioni, con un notevole risparmio rispetto agli $O(bm)$ stati da memorizzare nella ricerca in profondità. Con il backtracking, abbiamo anche la possibilità di mantenere una struttura dati efficiente per gli stati sul cammino corrente, che ci consente di controllare la presenza di un cammino ciclico in tempo $O(1)$ anziché $O(m)$. Affinché il backtracking funzioni, dobbiamo avere la possibilità di *annullare* ogni modifica quando "torniamo sui nostri passi". (*)

Ricerca a profondità limitata e ad approfondimento iterativo

Per evitare che la ricerca in profondità si perda in un cammino infinito possiamo usare la ricerca a profondità limitata, in cui forniamo un limite di profondità, ℓ, e consideriamo tutti i nodi a profondità ℓ come se non avessero successori. La complessità temporale è $O(b^\ell)$ e quella spaziale è $O(b^\ell)$. Se si sceglie male ℓ l'algoritmo risulta ancora incompleto. Poiché è una ricerca in profondità per migliorare possiamo eliminare i cammini ciclici al costo di un po' di tempo di calcolo in più. La scelta migliore sarebbe scegliere ℓ come diametro del grafo

La ricerca ad approfondimento iterativo risolve il problema di trovare un buon valore di ℓ provando tutti i valori: prima 0, poi 1, poi 2, e così via fino a quando si trova una soluzione, oppure la ricerca a profondità limitata restituisce il valore *fallimento* anziché il valore *soglia*. L'approfondimento iterativo riunisce vantaggi offerti dalla ricerca in profondità e da quella in ampiezza:

- i requisiti di memoria sono: $O(bd)$ quando esiste una soluzione, $O(bm)$ su spazi degli stati finiti senza soluzione. (profondità)
- la ricerca ad approfondimento operativo è ottima per problemi in cui tutte le azioni hanno lo stesso costo, ed è completa su spazi degli stati aciclici finiti, o su qualsiasi spazio degli stati finito quando controlliamo i nodi per la presenza di cicli risalendo l'intero cammino. (ampiezza)

La complessità temporale è $O(b^d)$ quando esiste una soluzione, $O(b^m)$ quando non esiste. Ogni iterazione della ricerca ad approfondimento iterativo genera un nuovo livello, come la ricerca in ampiezza, ma mentre quest'ultima lo fa memorizzando tutti i nodi in memoria, la ricerca ad approfondimento iterativo lo fa ricostruendo tutti i livelli precedenti, risparmiando memoria in cambio di un tempo maggiore. In una ricerca ad approfondimento iterativo, i nodi al livello più basso (a profondità d) sono generati una sola volta, quelli al livello immediatamente superiore due volte, e così via fino ai figli diretti del nodo radice, che sono generati d volte. Quindi, nel caso peggiore il numero totale dei nodi generati è:

$$N(\text{RAI}) = (d)b^1 + (d-1)b^2 + \cdots + (1)b^d ,$$

il che ci dà una complessità temporale $O(b^d)$, asintoticamente equivalente alla ricerca in ampiezza. *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.*

Ricerca bidirezionale (*)

Gli algoritmi che abbiamo trattato finora partono da uno stato iniziale e possono raggiungere ognuno dei possibili stati obiettivo. Un approccio alternativo è quello della ricerca bidirezionale, che ricerca procedendo simultaneamente in avanti a partire dallo stato iniziale e all'indietro a partire dallo stato obiettivo (o dagli stati obiettivi, se sono più di uno) nella speranza che le due ricerche si incontrino. L'idea di base è che $b^{d/2} + b^{d/2}$ è molto minore di b^d.

Affinché questa strategia funzioni, è necessario tenere traccia di due frontiere e due tabelle di stati raggiunti, e occorre essere in grado di ragionare all'indietro. Descriviamo la ricerca bidirezionale best-first. Anche se ci sono due frontiere separate, il prossimo nodo da espandere è sempre quello con un valore minimo della funzione di valutazione. Quando la funzione di valutazione è il costo di cammino otteniamo una ricerca bidirezionale a costo uniforme, e se il costo del cammino ottimale è C^*, nessun nodo con costo > $C^*/2$ verrà espanso, cosa che può velocizzare notevolmente l'esecuzione.

Passiamo all'algoritmo due versioni del problema e la funzione di valutazione, una nella direzione in avanti (indice F) e una all'indietro (indice B). Quando la funzione di valutazione è il costo di cammino sappiamo che la prima soluzione trovata sarà una soluzione ottima, ma con funzioni di valutazione diverse non è necessariamente così, quindi teniamo traccia della migliore soluzione trovata finora, che potrebbe essere aggiornata più volte prima che il test.

Ricerca sui grafi

Mantiene una lista dei nodi visitati (lista chiusa), prima di espandere un nodo si controlla se lo stato era stato già incontrato prima, se questo succede, il nodo appena trovato non viene espanso. Ottimale solo se abbiamo la garanzia che il costo del nuovo cammino non è maggiore.

Confronto tra le strategie di ricerca non informata

criterio	in ampiezza	a costo uniforme	in profondità	a profondità limitata	ad approfondimento iterativo	bidirezionale (se applicabile)
completa?	Sì[1]	Sì[1,2]	No	No	Sì[1]	Sì[1,4]
ottima rispetto al costo?	Sì[3]	Sì	No	No	Sì[3]	Sì[3,4]
tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
spazio	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Strategie di ricerca informata o euristica

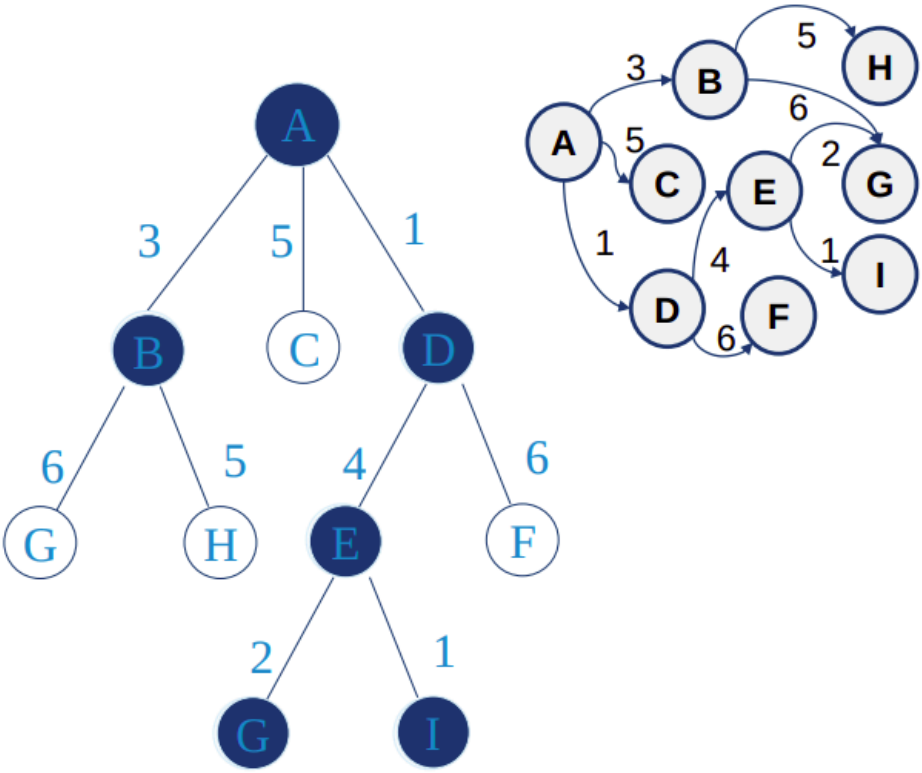
La ricerca esaustiva non è praticabile in problemi di complessità esponenziale. Questo paragrafo mostra come una strategia di ricerca informata, che sfrutta conoscenza specifica del dominio applicativo per fornire suggerimenti su dove si potrebbe trovare l'obiettivo, possa trovare soluzioni in modo più efficiente di una strategia non informata. Noi usiamo conoscenza del problema ed esperienza per riconoscere i cammini più promettenti, non evita la ricerca ma la riduce, consente in genere di trovare una buona soluzione in tempi accettabili e sotto certe condizioni garantisce completezza e ottimalità.

I suggerimenti hanno la forma di una funzione euristica denotata con $h(n)$

$h(n)$ = costo stimato del cammino meno costoso dallo stato del nodo n a uno stato obiettivo.

Ricerca best-first greedy o "golosa"

La ricerca best-first greedy è una forma di ricerca best-first che espande prima il nodo con il valore più basso di $h(n)$, cioè quello che appare più vicino all'obiettivo, sulla base del fatto che è probabile che questo porti rapidamente a una soluzione. Perciò la funzione di valutazione è $f(n) = h(n)$.

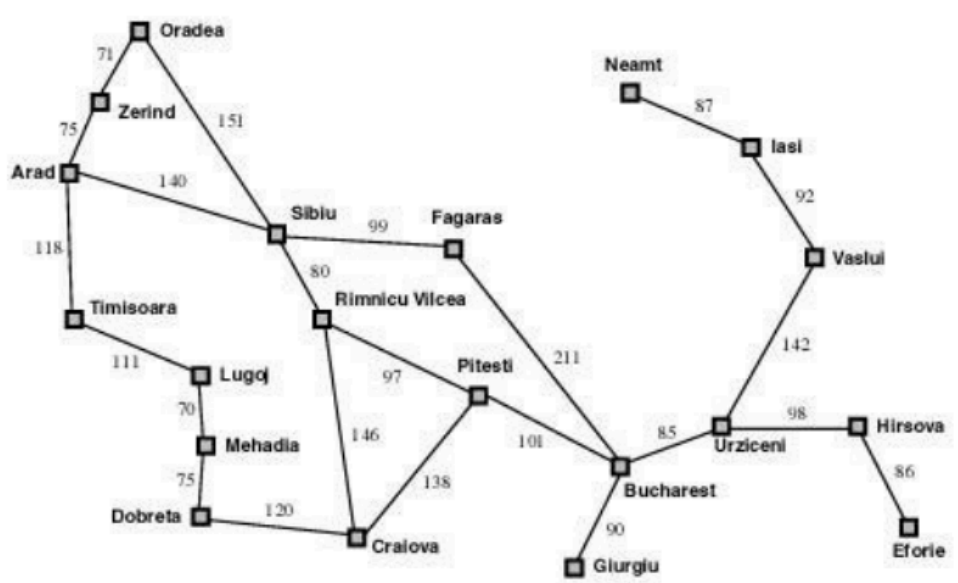


La ricerca best-first greedy su grafo è completa negli spazi degli stati finiti, ma non in quelli infiniti. Nel caso peggiore la complessità temporale e spaziale è $O(|V|)$, ma con una buona funzione euristica è possibile ridurla considerevolmente, arrivando in certi problemi a $O(bm)$. La Best First non è in generale completa, né ottimale.

Ricerca A*

La forma più diffusa di algoritmo di ricerca informata è la ricerca A*, una ricerca best-first che utilizza la funzione di valutazione: $f(n) = g(n) + h(n)$.
dove $g(n)$ è il costo del cammino dal nodo iniziale al no il costo del cammino dal nodo iniziale al nodo n e $h(n)$ rappresenta il costo *stimato* del cammino più breve da n a uno stato obiettivo, per cui abbiamo:
$f(n)$ = costo stimato del cammino migliore che continua da n fino a un obiettivo.

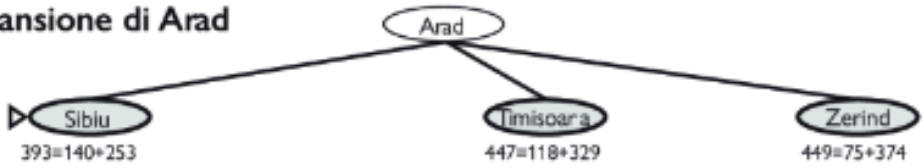
Casi particolari dell'algoritmo A:
Se $h(n) = 0$ $[f(n) = g(n)]$ si ha Ricerca Uniform Cost.
Se $g(n) = 0$ $[f(n) = h(n)]$ si ha Greedy Best First.



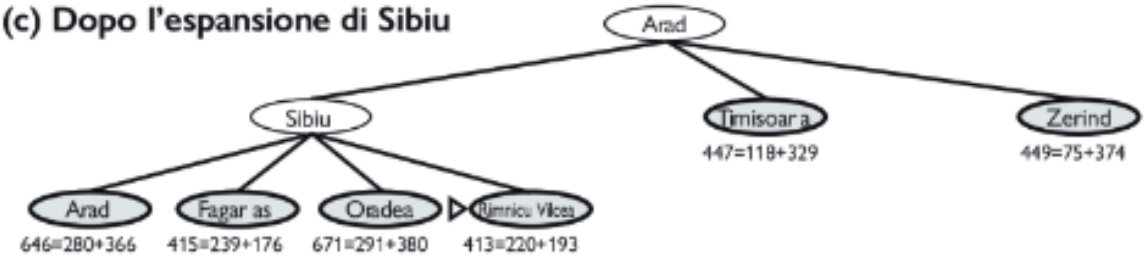
Straightline distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

(a) Stato iniziale

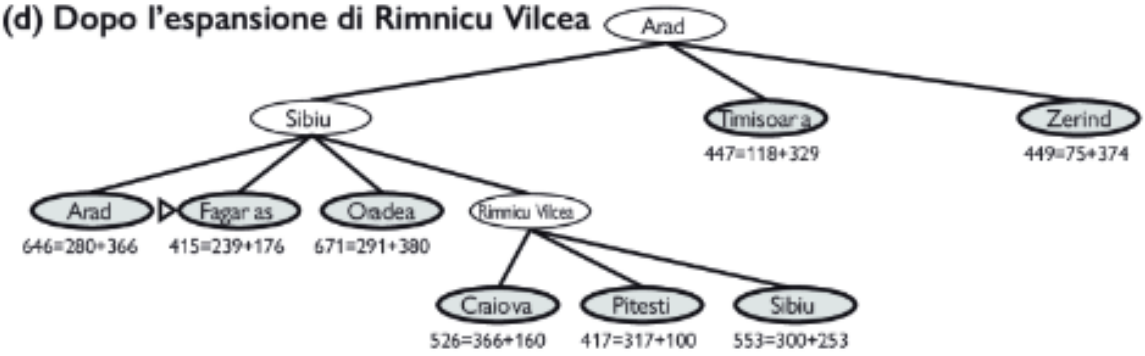
(b) Dopo l'espansione di Arad

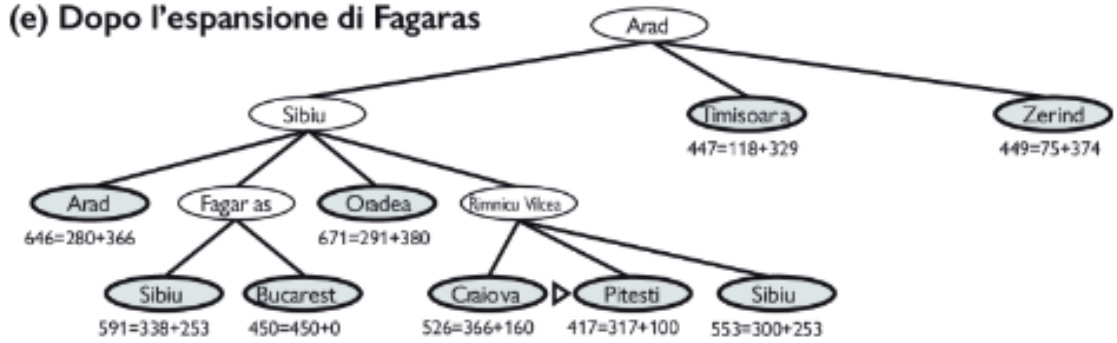


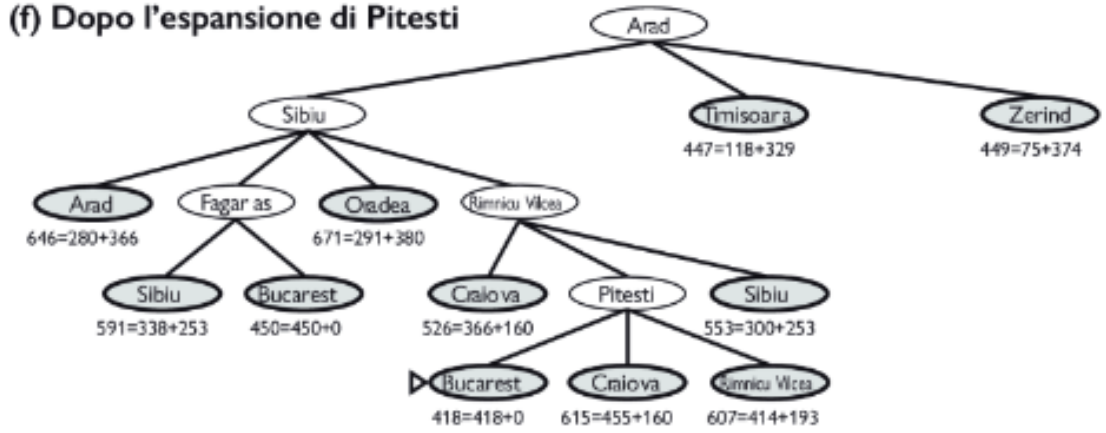
(c) Dopo l'espansione di Sibiu



(d) Dopo l'espansione di Rimnicu Vilcea





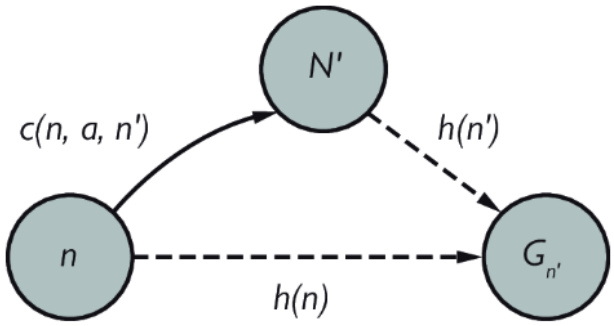


Proprietà di A*:
Completezza? Si (a meno che non vi sia un numero infinito di nodi con f ≤ f(G)).
Tempo? Esponenziale.
Spazio? Mantiene tutti i nodi in memoria.
Ottimalità? Si.
Osservazioni:
Una sottostima può farci compiere del lavoro inutile, però non ci fa perdere il cammino migliore. La componente g fa sì che si abbandonino cammini che vanno troppo in profondità. Una funzione che qualche volta sovrastima può farci perdere la soluzione ottimale. Nel caso di ricerca su albero l'uso di un'euristica ammissibile è sufficiente a garantire l'ammissibilità. Nel caso di ricerca su grafo serve una proprietà più forte: la consistenza (detta anche monotonicità).

Il fatto che sia o meno ottimale rispetto al costo dipende da alcune proprietà dell'euristica. Una proprietà fondamentale è l'ammissibilità: **un'euristica ammissibile** è tale se *non sovrastima mai* il costo per raggiungere un obiettivo.

Una proprietà leggermente più forte è la consistenza. Un'euristica $h(n)$ è consistente se, per ogni nodo n e ogni successore n' di n generato da un'azione a, abbiamo:

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



Questa è una forma della disuguaglianza triangolare, per cui ogni lato di un triangolo non può essere più lungo della somma degli altri due.

Ogni euristica consistente è ammissibile (ma non vale il viceversa), perciò A* con un'euristica consistente è ottima rispetto al costo. Inoltre, con un'euristica consistente, la prima volta che raggiungiamo uno stato sarà su un cammino ottimo, perciò non dovremo mai aggiungere di nuovo uno stato alla frontiera, né dovremo mai modificare un elemento in *raggiunti*. Con un'euristica inammissibile, A* può essere ottima rispetto al costo oppure no.

Confini di ricerca

A* è completo, con euristica monotona è ottimale, è ottimamente efficiente: a parità di euristica nessun altro algoritmo espande meno nodi (senza rinunciare a ottimalità) ma il problema sta nell'occupazione di memoria ($O(b^{d+1})$).

Ricerca con memoria limitata

La ricerca A* ha molte buone qualità, ma espande una grande quantità di nodi. È possibile esplorare un numero minore di nodi (con risparmio di tempo e di spazio) se si è disponibili ad accettare soluzioni subottime ma "sufficientemente buone", che chiamiamo soluzioni soddisfacenti (*satisficing*). Se consentiamo alla ricerca A* di usare un'euristica inammissibile – che potrebbe sovrastimare – rischiamo di non trovare la soluzione ottima, ma l'euristica potrebbe essere più accurata, riducendo così il numero di nodi espansi.

La memoria è suddivisa tra la *frontiera* e gli stati *raggiunti*. Nella nostra implementazione della ricerca best-first, uno stato che si trova sulla frontiera è memorizzato in due posizioni: come nodo nella frontiera (così possiamo decidere il prossimo nodo da espandere) e come elemento nella tabella degli stati raggiunti (così sappiamo se lo abbiamo già visitato). Per molti problemi questa ridondanza non desta preoccupazioni dato che la dimensione della *frontiera* è molto più piccola della dimensione di *raggiunti*, perciò la duplicazione degli stati nella frontiera richiede una quantità di memoria praticamente trascurabile. Alcune implementazioni, tuttavia, mantengono gli stati in una sola delle due posizioni, risparmiando un po' di spazio in cambio di una maggiore complessità dell'algoritmo. Un'altra possibilità è quella di rimuovere gli stati da

raggiunti quando si è in grado di dimostrare che non servono più. Ora consideriamo alcuni nuovi algoritmi progettati per risparmiare spazio in memoria.

La ricerca **beam** (letteralmente "ricerca a fascio", inteso come fascio di luce) limita la dimensione della frontiera. L'approccio più semplice consiste nel mantenere soltanto i *k* nodi con i migliori costi *f*, scartando ogni altro nodo espanso; questo naturalmente rende la ricerca incompleta e subottima, ma possiamo scegliere *k* per fare buon uso della memoria disponibile, e l'algoritmo viene eseguito velocemente perché espande meno nodi. Mentre la ricerca a costo uniforme o la ricerca A* si espandono in confini concentrici, la ricerca beam esplora soltanto un settore di tali confini, il "fascio" che contiene i *k* migliori candidati. Una versione alternativa della ricerca beam non impone un limite stretto sulla dimensione della frontiera, ma mantiene ogni nodo il cui costo *f* rientra in un fattore δ del migliore costo *f*. La Beam Search non è completa.

La ricerca A* ad approfondimento iterativo (**IDA***, da *Iterative-Deepening A**) sta alla ricerca A* come la ricerca ad approfondimento iterativo sta alla ricerca in profondità: la ricerca IDA* fornisce i vantaggi della ricerca A* senza la necessità di mantenere in memoria tutti gli stati raggiunti, al costo di visitare alcuni stati più volte. Nell'approfondimento iterativo standard la soglia è la profondità, che aumenta di uno a ogni iterazione. Nella ricerca IDA* la soglia è il costo *f* (*g* + *h*); a ogni iterazione il valore soglia è il più piccolo costo *f* di qualsiasi nodo che abbia superato la soglia nella precedente iterazione. In altre parole, ogni iterazione esegue una ricerca esaustiva di un confine *f*, trova un nodo appena al di là di tale confine e utilizza il costo *f* di tale nodo come confine successivo. IDA* è completo e ottimale, occupazione di memoria O(bd).

La ricerca best-first ricorsiva (**RBFS**, da *Recursive Best-First Search*), tenta di imitare il funzionamento di una ricerca best-first standard usando solamente uno spazio lineare. L'algoritmo assomiglia a una ricerca in profondità ricorsiva, ma invece di continuare a seguire indefinitamente il cammino corrente, utilizza la variabile *f_limite* per tenere traccia il valore *f* del miglior cammino *alternativo* che parte da uno qualsiasi degli antenati del nodo corrente. Se il nodo corrente supera questo limite, la ricorsione torna indietro al cammino alternativo. Durante il ritorno, RBFS sostituisce il valore *f* di ogni nodo lungo il cammino con un valore di backup, il miglior il valore *f* dei suoi nodi figli. In questo modo RBFS ricorda il valore *f* della foglia migliore nel sottoalbero abbandonato e può quindi decidere in seguito di ri-espanderlo. La ricerca RBFS è più efficiente della ricerca IDA*, ma soffre ancora per un'eccessiva ri-generazione di nodi. RBFS è un algoritmo ottimo se la funzione euristica *h*(*n*) è ammissibile. La sua complessità spaziale è lineare nella profondità della più profonda soluzione ottimale, ma quella temporale è abbastanza difficile da definire, perché dipende sia dall'accuratezza della funzione euristica sia dalla frequenza dei cambiamenti del cammino ottimale durante l'espansione dei nodi. L'algoritmo RBFS espande i nodi in ordine crescente di costo *f* anche se *f* non è monotona. Il problema degli algoritmi IDA* e RBFS è che usano *troppo poca* memoria.

Due algoritmi migliori sono **MA*** (*memory-bounded A**) e **SMA*** (*simplified MA**). SMA* procede proprio come A*, espandendo la foglia migliore finché la memoria è piena. A questo punto non

può aggiungere un nuovo nodo all'albero di ricerca senza cancellarne uno vecchio. SMA* scarta sempre il nodo foglia *peggiore*, quello con costo *f* più alto. Come RBFS, memorizza nel nodo padre il valore del nodo dimenticato. In questo modo la radice di un sottoalbero dimenticato conosce la qualità del cammino migliore in quel sottoalbero. Con quest'informazione SMA* ri-genera il sottoalbero dimenticato solo quando *tutti gli altri cammini* promettono di comportarsi peggio di quello. Potremmo anche dire che, se tutti i discendenti di un nodo *n* sono dimenticati, allora non sappiamo da che parte andare partendo da *n*, ma abbiamo ancora un'idea chiara di quanto sia conveniente passare per *n*. L'algoritmo SMA* è completo se c'è una soluzione raggiungibile, ovvero se *d*, la profondità del nodo obiettivo più vicino alla radice, è inferiore alla dimensione della memoria espressa in nodi. La strategia è ottima se c'è una soluzione ottima raggiungibile; altrimenti viene restituita la soluzione migliore tra quelle raggiungibili. In algoritmi a memoria limitata (IDA* e SMA*) le limitazioni della memoria possono portare a compiere molto lavoro inutile • Difficile stimare la complessità temporale effettiva • Le limitazioni di memoria possono rendere un problema intrattabile dal punto di vista computazionale.

Funzioni euristiche

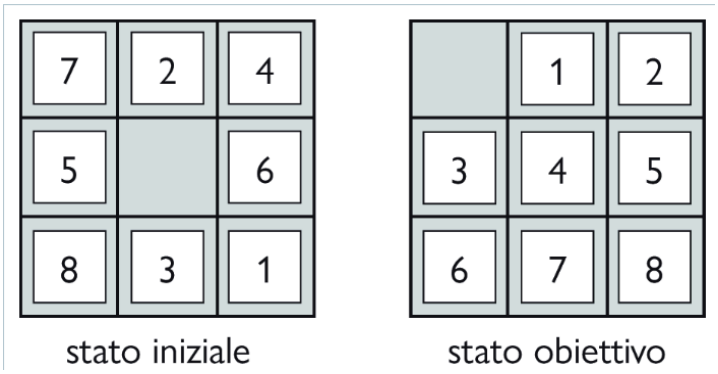
A parità di ammissibilità, una euristica può essere più efficiente di un'altra nel trovare il cammino soluzione migliore (visitare meno nodi): dipende da quanto informata è (o dal grado di informazione posseduto):

- *h(n)=0* minimo di informazione (BF o UF)
- h*(n) massimo di informazione (oracolo)

0 <= h(n) <= h*(n)

Teorema: Se h1 <= h2 , i nodi espansi da A* con h2 sono un sottoinsieme di quelli espansi da A* con h1 . Se h1 <= h2 , A* con h2 è almeno efficiente quanto A* con h1. Un'euristica più informata riduce lo spazio di ricerca (è più efficiente), ma è tipicamente più costosa da calcolare

Prendiamo come esempio il rompicapo degli 8 tasselli: lo scopo del rompicapo è di far scivolare i tasselli orizzontalmente o verticalmente nello spazio vuoto, finché la configurazione del gioco corrisponde a quella obiettivo.



Con l'aumentare dei tasselli la memoria necessaria aumenterà esponenzialmente, si descrivono di seguito le euristiche più comuni:

- h_1 = il numero di tasselli fuori posto (spazi vuoti non inclusi). h_1 è un'euristica ammissibile, perché ogni tassello fuori posto richiederà almeno uno spostamento per farlo arrivare al posto giusto;
- h_2 = la somma delle distanze di tutti i tasselli dalla loro posizione corrente a quella nella configurazione obiettivo. Dato che i tasselli non si possono muovere in diagonale, la distanza è la somma delle distanze in orizzontale e verticale, talvolta chiamata distanza tra isolati o distanza Manhattan. Anche h_2 è ammissibile, perché una mossa può al massimo spostare un tassello un passo più vicino al suo obiettivo.

Effetto dell'accuratezza dell'euristica sulle prestazioni

Un modo di caratterizzare la qualità di un'euristica è il fattore di ramificazione effettivo, indicato con b^*. Se il numero totale di nodi generati da A* per un particolare problema è N, e la profondità è d, allora b^* è il fattore di ramificazione che un albero uniforme di profondità d dovrebbe avere per contenere $N + 1$ nodi. Quindi,

$$N + 1 = 1 + b^* + (b^*)_2 + \cdots + (b^*)_d.$$

Per esempio, se A* trova una soluzione a profondità 5 usando 52 nodi, il fattore di ramificazione effettivo è 1,92. Questo fattore può cambiare da un'istanza del problema all'altra, ma solitamente per un dominio del problema specifico (come quello dei rompicapi a 8 tasselli) ha un valore abbastanza costante su tutte le istanze di problemi non banali. Di conseguenza, misurazioni sperimentali di b^* su un numero abbastanza piccolo di problemi può già fornire una buona idea dell'utilità generale dell'euristica. Un'euristica ben progettata dovrebbe avere un valore di b^* vicino a 1, permettendo così la soluzione di problemi abbastanza grandi a un costo computazionale ragionevole.

L'euristica può migliorare di molto la capacità di esplorazione dello spazio degli stati rispetto alla ricerca cieca. Migliorando anche di poco l'euristica si riesce ad esplorare uno spazio molto più grande.

Alcune strategie per ottenere euristiche ammissibili:

- Rilassamento del problema
- Massimizzazione di euristiche
- Database di pattern disgiunti
- Combinazione lineare
- Apprendere dall'esperienza

Rilassamento del problema:
Un problema con meno restrizioni sulle azioni possibili è detto problema rilassato. Il grafo dello

spazio degli stati per il problema rilassato è un *supergrafo* di quello dello spazio degli stati originale, perché la rimozione dei vincoli crea nuovi archi nel grafo.

Poiché il problema rilassato aggiunge archi al grafo dello spazio degli stati, qualsiasi soluzione ottima del problema originale è, per definizione, soluzione anche del problema rilassato, che però può avere soluzioni *migliori* se gli archi aggiunti forniscono delle scorciatoie. Quindi, *il costo di una soluzione ottima di un problema rilassato è un'euristica ammissibile per il problema originale.* Inoltre, poiché l'euristica derivata è un costo esatto per il problema rilassato, deve rispettare la disuguaglianza triangolare ed è quindi consistente.

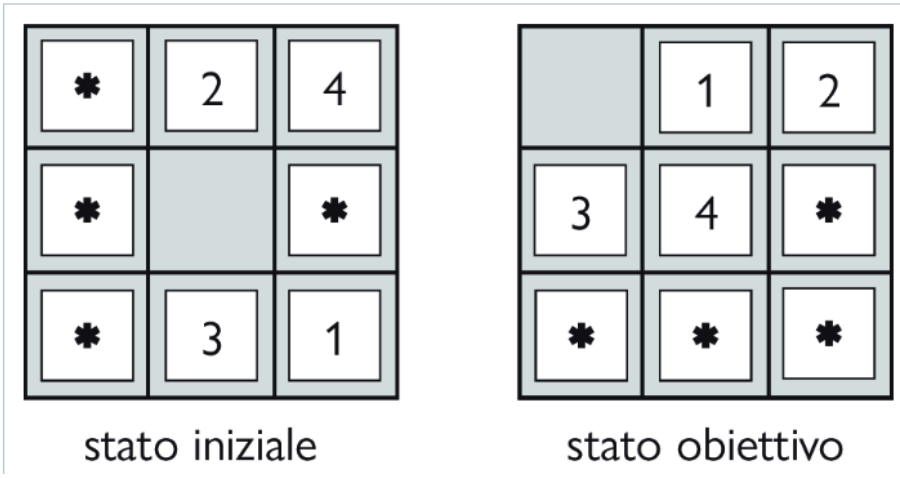
Se per un problema abbiamo una collezione di euristiche ammissibili $h_1 \ldots h_m$ e nessuna domina le altre, quale dovremmo scegliere? Possiamo prendere il meglio di tutte definendo:

$$h(n) = \max\{h_1(n), \ldots, h_m(n)\}.$$

Questa euristica composita sceglie la funzione più accurata per ogni nodo. Dato che tutte le euristiche componenti sono ammissibili, h è ammissibile (e se le h_i sono tutte consistenti, h è consistente). L'unico svantaggio è che $h(n)$ richiede più tempo di calcolo. Se questo costituisce un problema, un'alternativa è quella di selezionare a caso una delle euristiche a ogni valutazione, o usare un algoritmo di apprendimento automatico per predire quale sarà l'euristica migliore. In questo modo si può ottenere un'euristica inconsistente (perfino se ogni h_i è consistente), ma nella pratica si ottiene solitamente una risoluzione del problema più rapida.

Generare euristiche da sottoproblemi: database di pattern

È anche possibile derivare euristiche ammissibili dal costo della soluzione di un sottoproblema del problema dato. Per esempio, nel problema del rompicapo dei 8 tasselli il sottoproblema considera solo i tasselli 1, 2, 3 e 4 e lo spazio vuoto nelle loro posizioni corrette. Palesemente, il costo della soluzione ottimale di questo sottoproblema è un limite inferiore del costo del problema completo. Quest'euristica, in alcuni casi, si è rivelata molto più accurata della distanza Manhattan.



L'idea alla base dei database di pattern è di memorizzare i costi esatti delle soluzioni di ogni possibile istanza di sottoproblema.

Ora possiamo calcolare un'euristica ammissibile h_{DB} per ogni stato incontrato durante una ricerca, semplicemente estraendo dal database la corrispondente configurazione del sottoproblema. Il database stesso è costruito eseguendo una ricerca all'indietro dallo stato obiettivo e memorizzando il costo di ogni nuova configurazione incontrata; la spesa sostenuta per questa ricerca è ammortizzata dal fatto che il database può essere usato per molte istanze del problema.

La scelta di usare i primi quattro tasselli e lo spazio vuoto è arbitraria; avremmo potuto costruire database per i tasselli 5-6-7-8, oppure 2-4-6-8, e così via. Ognuno di essi fornisce un'euristica ammissibile, e queste possono essere tutte combinate, prendendo sempre il loro valore massimo.

Ma potremmo sommarle e ottenere un'euristica ancora più accurata? In generale no, perchè le soluzioni ai sottoproblemi interferiscono e la somma delle euristiche in generale non è ammissibile. Si deve eliminare il costo delle mosse che contribuiscono all'altro sottoproblema così da avere dei pattern disgiunti. Database di pattern disgiunti consentono di sommare i costi (euristiche additive).

Generare euristiche con punti di riferimento (*landmark*)

Abbiamo visto che un modo per trovare un'euristica è quello di immaginare un problema rilassato per cui sia facile trovare una soluzione ottima. Un'alternativa è quella di imparare dall'esperienza. Ogni soluzione ottima per un'istanza del rompicapo fornisce una coppia di esempio (obiettivo, cammino), e da questi esempi è possibile usare un algoritmo di apprendimento per costruire una funzione h che possa, con un po' di fortuna, approssimare il costo di cammino reale per altri stati che dovessero presentarsi durante la ricerca. La maggior parte di questi approcci prevedono l'apprendimento di un'approssimazione imperfetta della funzione euristica, e quindi rischiano l'inammissibilità.

Alcune tecniche di apprendimento automatico funzionano meglio quando si forniscono loro le **caratteristiche** (*feature*) di uno stato rilevanti per predire il valore dell'euristica di quello stato, invece della descrizione completa dello stato.

Quando diverse caratteristiche influenzano la bontà di uno stato, si può usare una combinazione lineare: $h(n) = c_1h_1(n) + c_2h_2(n)+...+c_kh_k(n)$.

Notate che questa euristica soddisfa la condizione che $h(n) = 0$ per stati obiettivo, ma non è necessariamente ammissibile o consistente.

Risolvere i problemi con la ricerca

In un mondo non deterministico, l'agente avrà bisogno di un piano condizionale e di eseguire azioni diverse a seconda di ciò che osserva. In caso di osservabilità parziale, l'agente dovrà anche tenere traccia degli stati in cui potrebbe trovarsi. Per guidare l'agente attraverso uno spazio sconosciuto che deve apprendere mentre procede si usa la **ricerca online**.

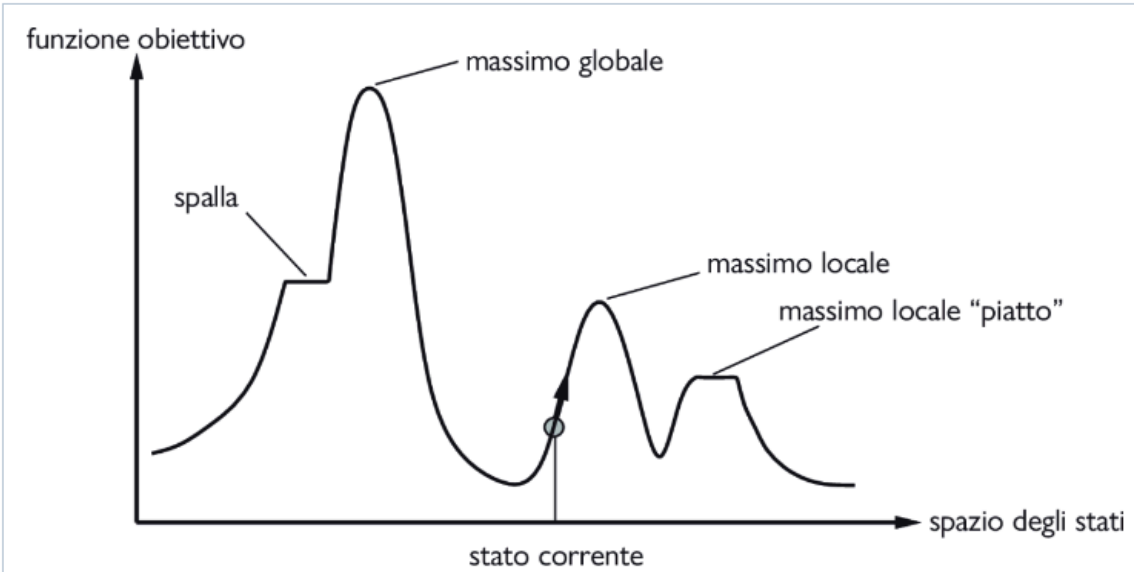
Ricerca locale e problemi di ottimizzazione

A volte invece siamo interessati soltanto allo stato finale e non al cammino percorso per raggiungerlo. Per esempio, nel problema delle 8 regine ci interessa soltanto trovare una configurazione valida delle 8 regine (infatti, conoscendo la configurazione, è facile ricostruire i passi per crearla).

Gli algoritmi di **ricerca locale** operano cercando a partire da uno stato iniziale e procedendo verso gli stati adiacenti senza tenere traccia dei cammini, né degli stati già raggiunti. Questo significa che non sono sistematici – potrebbero anche non esplorare mai una porzione dello spazio degli stati dove in effetti risiede una soluzione. Hanno però due vantaggi: (1) usano pochissima memoria e (2) spesso riescono a trovare soluzioni ragionevoli in spazi degli stati grandi o anche infiniti per cui usare algoritmi sistematici non sarebbe praticabile.

Gli algoritmi di ricerca locale possono anche risolvere **problemi di ottimizzazione**, in cui lo scopo è trovare lo stato migliore secondo una **funzione obiettivo**.

Per comprendere la ricerca locale, potete considerare gli stati di un problema disposti in un **panorama dello spazio degli stati**. Ogni punto (stato) in nel panorama ha una "altezza" definita dal valore della funzione obiettivo; se l'altezza corrisponde a una funzione obiettivo, lo scopo è quello di trovare il picco più alto, un **massimo globale** e chiamiamo questo processo **hill climbing** (traducibile letteralmente in "scalare la collina"). Se l'altezza corrisponde al costo, lo scopo è quello di trovare l'avvallamento più profondo un **minimo globale** e parliamo in questo caso di **discesa del gradiente** (*gradient descent*).



Ricerca hill climbing

L'algoritmo di ricerca hill climbing tiene traccia di un solo stato corrente e a ogni iterazione passa allo stato vicino con valore più alto, cioè punta nella direzione che presenta l'ascesa più ripida, senza guardare oltre gli stati immediatamente vicini a quello corrente.

```
function Hill-Climbing(problema) returns uno stato che è un massimo locale

    corrente ← problema.StatoIniziale

    while true do

        vicino ← lo stato successore di corrente di valore più alto

        if Valore(vicino) ≤ Valore(corrente) then return corrente

        corrente ← vicino
```

L'algoritmo hill climbing viene talvolta chiamato **ricerca locale greedy**, perché sceglie uno stato vicino "buono" senza pensare a come andrà avanti. L'hill climbing può procedere molto rapidamente verso la soluzione, perché di solito è abbastanza facile migliorare uno stato sfavorevole.Sfortunatamente, spesso l'hill climbing rimane bloccato per le seguenti ragioni:

- **massimi locali**: un massimo locale è un picco più alto degli stati vicini, ma inferiore al massimo globale. Gli algoritmi hill climbing che raggiungono la vicinanza di un massimo locale saranno attirati verso il picco, ma rimarranno bloccati lì senza poter andare altrove.
- **creste**: una cresta (*ridge*) da origine a una sequenza di massimi locali molto difficili da esplorare da parte degli algoritmi greedy;
- **plateau**: un plateau 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. Una ricerca hill climbing potrebbe perdersi sul plateau.

I possibili miglioramenti che si possono fare sono:

- Proseguire nella ricerca una volta raggiunto un plateau, consentendo una mossa laterale nella speranza che il plateau sia in realtà una spalla. Ma se fossimo realmente su un massimo locale piatto, con questo approccio finiremmo per vagare sul plateau per sempre. Possiamo quindi porre un limite al numero di mosse laterali consecutive e fermarci dopo quelle.
- **L'hill climbing stocastico** sceglie a caso tra tutte le mosse che vanno verso l'alto: la probabilità della scelta può essere influenzata dalla "pendenza" delle mosse. Normalmente questo algoritmo converge più lentamente di quello che sceglie sempre la mossa più conveniente, ma in alcuni panorami di stati è capace di trovare soluzioni migliori.
- **L'hill climbing con prima scelta** implementa la precedente versione stocastica generando casualmente i successori fino a ottenerne uno preferibile allo stato corrente. Questa strategia è molto buona quando uno stato ha molti successori.
- **L'hill climbing con riavvio casuale**, l'algoritmo conduce una serie di ricerche hill climbing partendo da stati iniziali generati casualmente, fino a quando raggiunge un obiettivo. È completo con probabilità 1, perché prima o poi dovrà generare, come stato iniziale, proprio un obiettivo. Se ogni ricerca hill climbing ha una probabilità p di successo, il numero atteso di riavvii richiesti è $1/p$.

Simulated annealing

Un algoritmo hill climbing che non scende mai "a valle" verso stati con valore più basso (o costo più alto) è sempre vulnerabile alla possibilità di rimanere bloccato in corrispondenza di un massimo locale. Di contro, un'esplorazione del tutto casuale che si muove verso uno stato successore senza preoccuparsi del valore finirà per incontrare il massimo globale, ma sarà estremamente inefficiente. Sembra ragionevole, quindi, cercare di combinare in qualche modo l'hill climbing con un'esplorazione casuale in modo da ottenere sia l'efficienza che la completezza.

Un algoritmo che fa proprio questo è il **simulated annealing**. Per spiegare il simulated annealing, cambiamo il punto di vista dalla scalata di colline alla **discesa del gradiente** (cioè la minimizzazione del costo) e immaginiamoci alle prese con il problema di far entrare una pallina da ping-pong nella fessura più profonda. Se lasciamo semplicemente rotolare la pallina, questa si fermerà in corrispondenza di un minimo locale. Se scuotiamo la superficie, possiamo far uscire la pallina da questo avvallamento, rischiando però di farla entrare in un altro minimo più profondo, dove starà ancora per più tempo. La soluzione proposta dal simulated annealing è di cominciare scuotendo molto e poi ridurre gradualmente l'intensità dello scuotimento. La struttura complessiva dell'algoritmo simulated annealing è simile a quella dell'hill climbing: stavolta però, invece della mossa *migliore*, viene scelta una mossa *casuale*. Se la mossa migliora la situazione, viene sempre accettata; in caso contrario l'algoritmo la accetta con una probabilità inferiore a 1. La probabilità decresce esponenzialmente con la "cattiva qualità" della mossa, misurata dal peggioramento ΔE della valutazione. La probabilità decresce anche con la "temperatura" T che scende costantemente: le mosse "cattive" saranno accettate più facilmente all'inizio, in condizioni di T alta, e diventeranno sempre meno probabili a mano a mano che T si abbassa. Se la *velocità_raffreddamento* fa decrescere la temperatura da T a 0 abbastanza lentamente, per una proprietà della distribuzione di Boltzmann, $e^{\Delta E/T}$, tutta la probabilità è concentrata sui massimi globali, che l'algoritmo troverà con probabilità tendente a 1.

```
function Simulated-Annealing(problema, velocità_raffreddamento) returns uno stato soluzione

    corrente ← problema.StatoIniziale

    for t ← 1 to ∞ do

        T ← velocità_raffreddamento[t]

        if T = 0 then return corrente

        successivo ← un successore scelto a caso di corrente

        ΔE ← Valore(corrente) - Valore(successivo)

        if ΔE > 0 then corrente ← successivo

        else corrente ← successivo solo con probabilità e^(-ΔE/T)
```

Ricerca local beam

Memorizzare un solo nodo può sembrare una reazione estrema ai problemi legati alle limitazioni di memoria. L'algoritmo di ricerca local beam tiene traccia di *k* stati anziché uno solo. All'inizio comincia con *k* stati generati casualmente: a ogni passo, sono generati i successori di tutti i *k* stati. Se uno qualsiasi di essi è un obiettivo, l'algoritmo termina; altrimenti sceglie i *k* successori migliori dalla lista di tutti i successori e ricomincia.

A prima vista, una ricerca local beam con *k* stati potrebbe sembrare nulla più di *k* ricerche a riavvio casuale eseguite in parallelo anziché una dopo l'altra. In realtà, i due algoritmi sono piuttosto differenti. In una ricerca a riavvio casuale, ogni processo di ricerca è del tutto indipendente dagli altri. *In una ricerca local beam, informazione utile viene passata dall'uno all'altro thread di ricerca paralleli.* L'algoritmo abbandona rapidamente le ricerche infruttuose e sposta la sue risorse dove si stanno verificando i progressi maggiori.

La ricerca local beam può soffrire di una carenza di diversificazione tra i *k* stati, che possono concentrarsi in cluster in una piccola regione dello spazio degli stati, rendendo così questa ricerca poco più di una versione *k* volte più lenta dell'hill climbing. Una variante chiamata **ricerca beam stocastica**, analoga all'hill climbing stocastico, aiuta ad alleviare questo problema: invece di scegliere i migliori *k* successori, in questo caso si scelgono i successori con probabilità proporzionale al loro valore, aumentando così la diversificazione.

Ricerca con azioni non deterministiche

Gli agenti risolutori di problemi "classici" assumono: Ambienti completamente osservabili e deterministici, il piano generato è una sequenza di azioni che può essere generata offline ed eseguita senza imprevisti, le percezioni non servono se non nello stato iniziale. In un ambiente parzialmente osservabile e non deterministico le percezioni sono importanti: restringono gli stati possibili (vincoli provenienti dall'ambiente) e informano sull'effetto dell'azione (esperienza).

Quando l'ambiente è parzialmente osservabile, invece, l'agente non sa con sicurezza in quale stato si trova; e quando l'ambiente è non deterministico, l'agente non sa in quale stato arriverà dopo l'esecuzione di un'azione. Chiamiamo **stato-credenza** un insieme di stati fisici che l'agente ritiene siano possibili.

In ambienti parzialmente osservabili e non deterministici, la soluzione di un problema non è una sequenza ma un **piano condizionale** (piano di contingenza o strategia) che specifica che cosa fare in base alle percezioni ricevute dall'agente durante l'esecuzione del piano.

Il mondo dell'aspirapolvere erratico

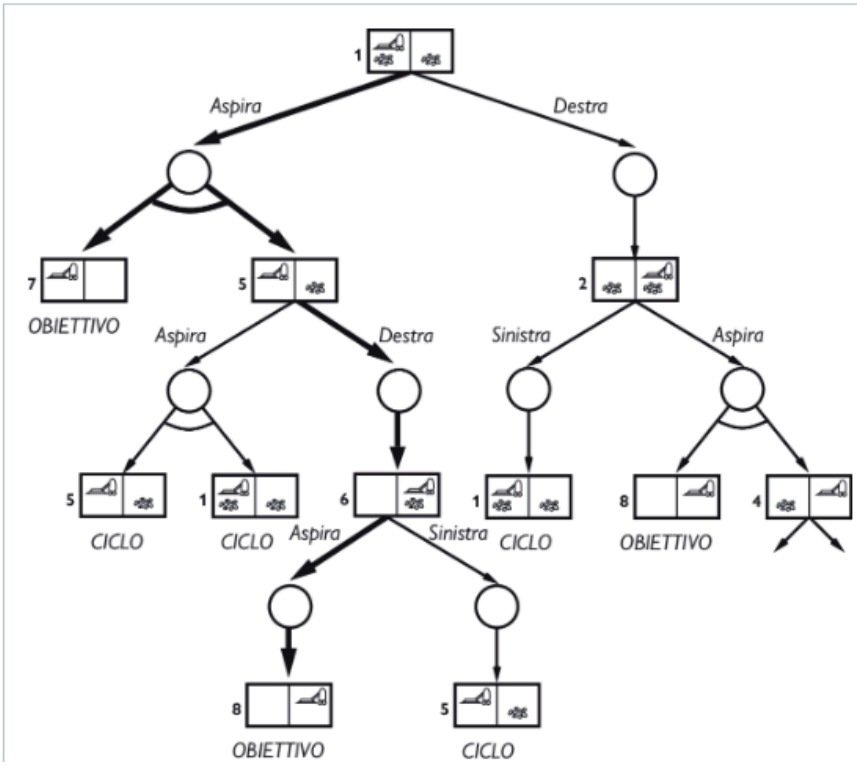
Nel mondo dell'aspirapolvere erratico, l'azione *Aspira* opera come segue:

- quando è applicata a un riquadro sporco, l'azione lo pulisce e talvolta pulisce anche un riquadro adiacente;
- quando è applicata a un riquadro pulito, l'azione talvolta deposita dello sporco sul tappeto.

Per fornire una formulazione precisa di questo problema, dobbiamo generalizzare la nozione di **modello di transizione**. Invece di definire il modello di transizione con una funzione Risultato che restituisce un singolo stato, utilizziamo una funzione Risultati che restituisce un insieme di possibili stati risultato.

Alberi di ricerca and-or

Iniziamo costruendo alberi di ricerca, ma in questo caso gli alberi hanno un carattere diverso. In un ambiente deterministico, l'unica ramificazione è introdotta dalle scelte dell'agente in ogni stato: "Posso fare questa azione o quella"; parliamo in questo caso di **nodi or**. In un ambiente non deterministico, la ramificazione è anche legata alla scelta del risultato per ogni azione, effettuata dall'*ambiente*. In questo caso parliamo di **nodi and**. Questi due tipi di nodi si alternano, generando un **albero and–or**.



Una soluzione per un problema di ricerca and–or è un sottoalbero dell'albero di ricerca completo che (1) ha un nodo obiettivo in ogni foglia, (2) specifica una sola azione in ognuno dei suoi nodi or e (3) include ogni ramo uscente da ognuno dei suoi nodi and.

```
function Ricerca-And-Or(problema) returns un piano condizionale, o fallimento
    return Ricerca-Or(problema, problema.StatoIniziale, [ ])
function Ricerca-Or(problema, stato, cammino) returns un piano condizionale, o fallimento
    if problema.È-Obiettivo(stato) then return il piano vuoto
    if È-Ciclo(cammino) then return fallimento
    for each azione in problema.Azioni(stato) do
        piano ← Ricerca-And(problema, Risultati(stato, azione), [stato + cammino])
        if piano ≠ fallimento then return [azione + piano]
    return fallimento
function Ricerca-And(problema, stati, cammino) returns un piano condizionale, o fallimento
    for each s_i in stati do
        piano_i ← Ricerca-Or(problema, s_i, cammino)
        if piano_i = fallimento then return fallimento
    return [if s_1 then piano_1 else if s_2 then piano_2 else . . . if s_{n-1} then piano_{n-1} else piano_n]
```

Prova, prova ancora

Consideriamo un mondo dell'aspirapolvere *scivoloso*, identico a quello dell'aspirapolvere normale (non erratico) fatta eccezione per il fatto che talvolta le azioni di movimento falliscono, lasciando l'agente nella medesima posizione. É chiaro che non ci sono soluzioni acicliche dallo stato 1, e Ricerca-And-Or terminerebbe fallendo. Esiste però una soluzione ciclica, quella di continuare a provare *Destra* finché funziona. Possiamo esprimere questa soluzione con un nuovo costrutto **while**:

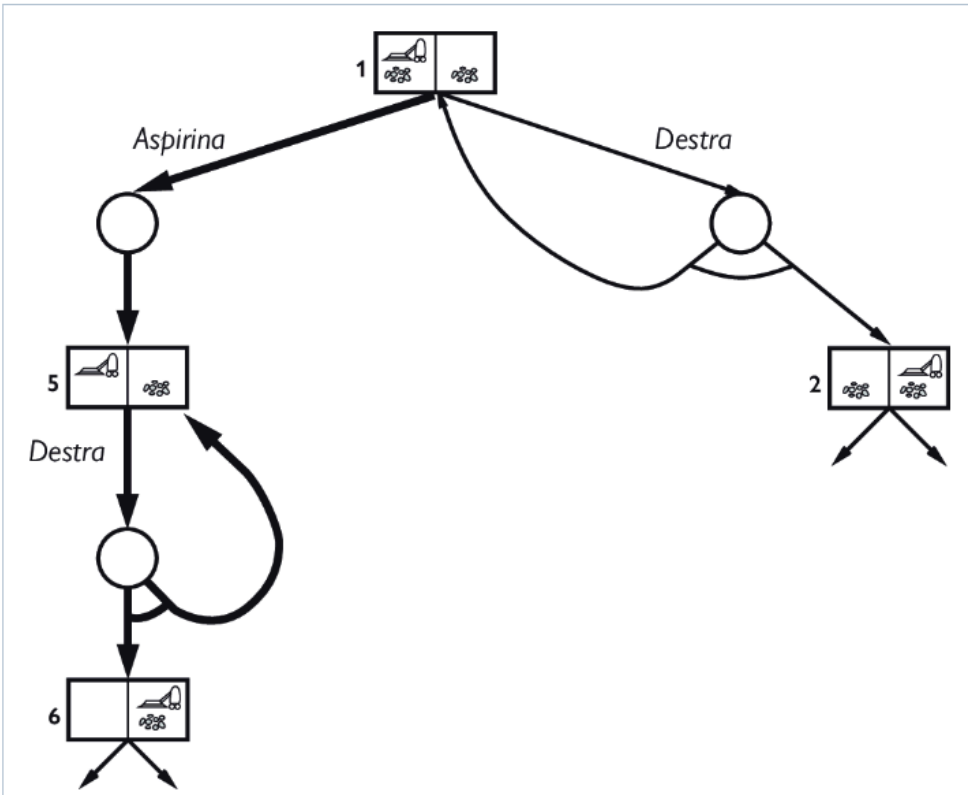


Figura 4.12 Parte del grafo di ricerca per il mondo dell'aspirapolvere scivoloso, in cui abbiamo evidenziato (alcuni) cicli. Tutte le soluzioni per questo problema sono piani ciclici perché non vi è modo per spostarsi in maniera affidabile.

Una condizione minima è che ogni foglia sia uno stato obiettivo e che una foglia sia raggiungibile da qualsiasi punto nel piano. Oltre a questo, dobbiamo considerare il non determinismo. Se il meccanismo di guida dell'aspirapolvere robotizzato funziona per un po' di tempo, ma in altri momenti slitta in modo casuale e indipendente, allora l'agente può essere fiducioso che, se l'azione viene ripetuta per un numero di volte sufficiente, alla fine funzionerà e il piano avrà successo. Ma se il non determinismo è dovuto a un fatto non osservato circa il robot o l'ambiente, allora la ripetizione dell'azione non è di alcun aiuto.

Ricerca con osservazioni parziali

Passiamo ora al problema dell'osservabilità parziale, in cui le percezioni dell'agente non sono sufficienti per determinare lo stato esatto. Questo significa che alcune delle azioni dell'agente punteranno a ridurre l'incertezza riguardo lo stato corrente.

Ricerca in assenza di osservazioni

Quando le osservazioni dell'agente *non forniscono alcuna informazione*, abbiamo un

problema senza sensori, talvolta detto **problema conformante**. Si possono trovare soluzioni anche senza affidarsi ai sensori utilizzando stati-credenza.

Consideriamo ora una versione senza sensori del mondo dell'aspirapolvere (deterministico). Supponiamo che l'agente conosca la geografia del suo mondo, ma non la propria posizione o la distribuzione della sporcizia. In tal caso, il suo stato iniziale potrebbe essere qualsiasi elemento dell'insieme {1,2,3,4,5,6,7,8}. La soluzione di un problema senza sensori è una sequenza di azioni, non un piano condizionale (perché non vi è percezione). Ma la ricerca viene effettuata nello spazio degli stati-credenza e non degli stati fisici. Nello spazio degli stati-credenza il problema è *completamente osservabile* perché l'agente conosce sempre il proprio stato-credenza.

Il problema originale *P* ha come componenti Azioni, Risultato e così via, e il problema di stati-credenza ha i componenti seguenti.

- **Stati**: lo spazio degli stati-credenza contiene ogni possibile sottoinsieme degli stati fisici. Se *P* ha *N* stati, allora il problema di stati-credenza ha 2*N* stati-credenza, anche se molti potrebbero essere irraggiungibili dallo stato iniziale.
- **Stato iniziale**: è tipicamente lo stato-credenza costituito da tutti gli stati in *P*, anche se in alcuni casi l'agente avrà una conoscenza maggiore di questa.
- **Azioni**: unione delle azioni lecite negli stati in b (ma se azioni illecite in uno stato hanno effetti dannosi meglio intersezione).
- **Modello di transizione**: gli stati risultanti sono quelli ottenibili applicando le azioni a uno stato qualsiasi (l' unione degli stati ottenibili dai diversi stati con le azioni eseguibili).

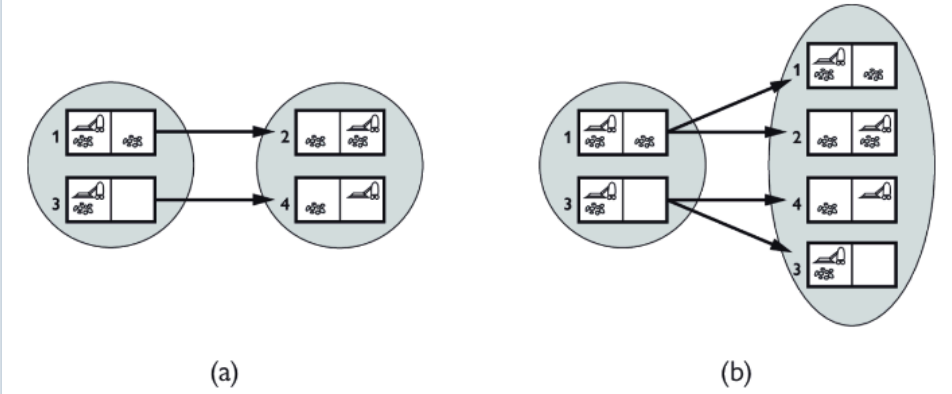


Figura 4.13 (a) Predizione del successivo stato-credenza per il mondo dell'aspirapolvere senza sensori con un'azione deterministica, *Destra*. (b) Predizione per lo stesso stato-credenza e la stessa azione nella versione scivolosa del mondo dell'aspirapolvere senza sensori.

- **Test obiettivo**: l'agente *può* raggiungere l'obiettivo se *qualche* stato *s* nello stato-credenza soddisfa il test obiettivo del problema sottostante.

- **Costo di azione**: Se la stessa azione può avere costi diversi in stati differenti, allora il costo di eseguire un'azione in un dato stato-credenza potrebbe essere uno tra diversi valori.

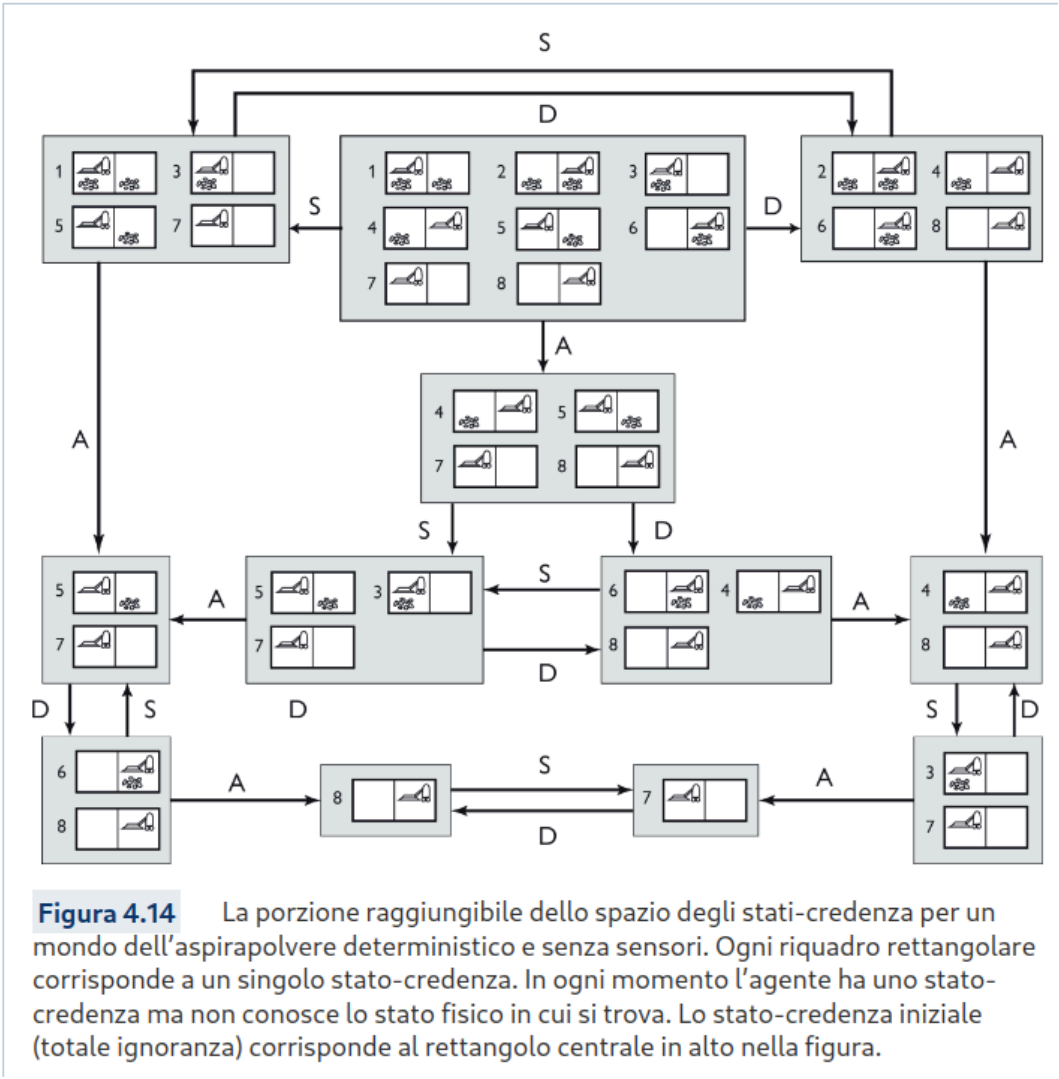


Figura 4.14 La porzione raggiungibile dello spazio degli stati-credenza per un mondo dell'aspirapolvere deterministico e senza sensori. Ogni riquadro rettangolare corrisponde a un singolo stato-credenza. In ogni momento l'agente ha uno stato-credenza ma non conosce lo stato fisico in cui si trova. Lo stato-credenza iniziale (totale ignoranza) corrisponde al rettangolo centrale in alto nella figura.

Gli stati credenza possibili sono 2 8=256 ma solo 12 sono raggiungibili. Si può effettuare un Ricerca-Grafo e controllare, generando s, se si è già incontrato uno stato credenza s'=s e trascurare s.
Si può anche "potare" in modo più efficace … 1. Se s' s (con s' stato credenza già incontrato) si può trascurare s 2. Se s s' e da s' si è trovata una soluzione si può trascurare s.
Lo spazio degli stati può essere molto più grande e la rappresentazione atomica obbliga a elencare tutti gli stati. Non è molto "compatta". Un altro approccio è quello di usare una **soluzione incrementale**:
Dovendo trovare una soluzione per {1, 2, 3 …} si cerca una soluzione per stato 1 e poi si controlla

che funzioni per 2 e i successivi; se no se ne cerca un'altra per 1. Scopre presto i fallimenti ma cerca un'unica soluzione che va bene per tutti gli stati.

Ricerca in ambienti parzialmente osservabili

Molti problemi non si possono risolvere senza sensori. Per esempio, il rompicapo a 8 tasselli senza sensori è impossibile da risolvere. Nel caso di un problema parzialmente osservabile, la specifica del problema includerà una funzione Percezione(*s*) che restituisce la percezione ricevuta dall'agente in un dato stato. Se i sensori sono non deterministici, possiamo utilizzare una funzione Percezioni che restituisce un insieme di possibili percezioni. Per problemi completamente osservabili, Percezione(*s*) = *s* per ogni stato *s*, mentre per problemi senza sensori Percezione(*s*) = *null*.
La transizione avviene in tre fasi:

1. La fase di predizione calcola lo stato-credenza risultante dall'azione, Risultato(*b*,*a*), esattamente come nei problemi senza sensori.
2. La fase delle percezioni possibili calcola l'insieme delle percezioni che potrebbero essere osservate nello stato-credenza predetto.
3. La fase di aggiornamento calcola, per ogni possibile percezione, lo stato-credenza che risulterebbe da essa.

L'agente deve gestire percezioni *possibili* durante la pianificazione, perché non può conoscere le percezioni *effettive* fino al momento dell'esecuzione del piano. Mettendo insieme tutte e tre le fasi otteniamo i possibili stati-credenza risultanti da una data azione e le conseguenti percezioni possibili.

Agenti per ricerca online e ambienti sconosciuti

Fin qui ci siamo concentrati su agenti che utilizzano algoritmi di ricerca offline, calcolando una soluzione completa prima di effettuare la prima azione. Al contrario, nella ricerca online un agente opera alternando computazione e azione: prima esegue un'azione, poi osserva l'ambiente e determina l'azione successiva. La ricerca online si presta bene agli ambienti dinamici o semidinamici, nei quali non c'è tempo per stare immobili a calcolare l'azione successiva. Inoltre, la ricerca online è utile anche in domini non deterministici, perché consente all'agente di concentrare le attività di calcolo sulle contingenze che si verificano effettivamente, anziché su quelle che *potrebbero* verificarsi ma probabilmente non lo faranno.
Naturalmente c'è un compromesso: più un agente pianifica in anticipo, meno spesso gli capiterà di trovarsi nei guai. In ambienti ignoti, l'agente non conosce gli stati e gli effetti delle azioni e deve sfruttare le sue azioni come esperimenti per poter apprendere le caratteristiche dell'ambiente. Un esempio classico di ricerca online è il problema di costruzione di mappe: I metodi per uscire dai labirinti rappresentano anch'essi esempi di algoritmi di ricerca online.

Problemi di ricerca online

Un problema di ricerca online viene risolto con attività di elaborazione, percezione e azione.

Osservate in particolare che l'agente *non può* determinare Risultato(*s*,*a*) se non trovandosi effettivamente in *s* ed eseguendo *a*. Infine, l'agente potrebbe avere accesso a una funzione euristica ammissibile *h*(*s*) capace di stimare la distanza dallo stato corrente a uno stato obiettivo.

Assunzioni per un problema di esplorazione:
• Solo lo stato corrente è osservabile, l'ambiente è ignoto.
• Non si conosce l'effetto delle azioni e il loro costo.
• Gli stati futuri e le azioni che saranno possibili non sono conosciute a priori.
• Si devono compiere azioni esplorative come parte della risoluzione del problema.

Conoscenze di un agente online in s:
• Le azioni legali nello stato attuale s.
• Risultato(*s*,*a*), ma dopo aver eseguito a.
• Il costo della mossa c(s, a, s'), solo dopo aver eseguito a.
• Goal-Test(s).
• La stima della distanza: dal goal: h(s).

Generalmente lo scopo dell'agente è raggiungere uno stato obiettivo minimizzando il costo (un altro possibile scopo può essere semplicemente di esplorare l'intero ambiente). Il costo è rappresentato da quello totale del cammino effettivamente percorso dall'agente. È prassi comune confrontare questo costo con quello del cammino che l'agente potrebbe seguire *se conoscesse in anticipo lo spazio di ricerca*, cioè il cammino ottimo nell'ambiente noto. Nel linguaggio degli algoritmi online, questo confronto determina il cosiddetto **rapporto di competitività** (*competitive ratio*), che auspichiamo sia il più piccolo possibile.

Gli esploratori online sono vulnerabili ai **vicoli ciechi**, stati da cui non è raggiungibile alcuno stato obiettivo. Se l'agente non conosce il significato di ogni azione, potrebbe eseguire azioni che non permetteranno di raggiungere mai l'obiettivo. In generale, *nessun algoritmo può evitare i vicoli ciechi in tutti gli spazi degli stati*. Gli ambienti sono esplorabili in maniera sicura solo se non esistono azioni irreversibili e lo stato obiettivo può sempre essere raggiunto, diversamente non si può garantire una soluzione. Anche negli ambienti esplorabili in modo sicuro non può essere garantito un rapporto di competitività limitato.

Agenti per ricerca online

Gli agenti online ad ogni passo decidono l'azione da fare (non il piano) e la eseguono.
La ricerca in profondità online consiste nell'esplorazione sistematica delle alternative, è

necessario ricordarsi ciò che si è scoperto. Il backtracking significa appunto tornare sui propri passi.

Ricerca locale online

Nella ricerca online si conosce il valore della funzione euristica solo quando è stato esplorato lo stato. Come quella in profondità, anche la ricerca hill climbing gode della proprietà di località nelle espansioni dei suoi nodi. In effetti, dato che tiene in memoria solamente lo stato corrente, la ricerca **hill climbing** è *già* un algoritmo online! Sfortunatamente, l'algoritmo di base non è molto utile per l'esplorazione, perché l'agente si può bloccare in corrispondenza di un massimo locale. Per di più, non si può neppure usare il meccanismo del riavvio casuale, perché l'agente non può teletrasportarsi in un nuovo stato iniziale.

Invece dei riavvi casuali si potrebbe considerare:
- L'uso di un **random walk** che sceglie in modo casuale una delle azioni possibili nello stato corrente.
- L'uso di apprendimento **Real-Time**: in questo modo si arricchisce la ricerca hill climbing con memoria anziché casualità, rendendolo più efficace. Esplorando si aggiustano i valori dell'euristica per renderli più realistici, utilizza l'algoritmo LRTA* (Learning Real Time A*). L'algoritmo aggiorna la stima del costo dello stato che ha appena lasciato e poi sceglie la mossa "apparentemente migliore" in base alle stime correnti dei costi. LRTA* cerca di simulare A* con un metodo locale: tiene conto del costo delle mosse come può aggiornando al volo la H. Completo in spazi esplorabili in maniera sicura, nel caso pessimo visita tutti gli stati due volte ma è mediamente più efficiente della profondità online, non ottimale, a meno di usare una euristica perfetta.

Agenti logici

Gli agenti basati sulla conoscenza decidono quali azioni intraprendere utilizzando un processo di ragionamento che opera su una rappresentazione interna di conoscenza.
Il mondo è tipicamente complesso: ci serve una rappresentazione parziale e incompleta di una astrazione del mondo utile agli scopi dell'agente. Maggiore conoscenza del mondo è cruciale per risolvere problemi in cui la osservazione è parziale ed il mondo evolve dinamicamente. Quindi, per ambienti parzialmente osservabili ad esempio, ci servono linguaggi di rappresentazione della conoscenza più espressivi e capacità inferenziali.

Agenti basati sulla conoscenza

Il componente più importante degli agenti basati sulla conoscenza è appunto la **base di conoscenza**, o KB (dall'inglese *knowledge base*). Informalmente, la base di conoscenza è costituita da un insieme di **formule**, espresse mediante un linguaggio di rappresentazione della conoscenza. Ogni formula rappresenta un'asserzione sul mondo. Quando una formula è data

per buona senza essere ricavata da altre formule, la chiamiamo **assioma**. La base di conoscenza deve prevedere meccanismi per aggiungere nuove formule e per le interrogazioni (o *query*). I nomi standard per queste due azioni sono rispettivamente Tell (asserisci) e Ask (chiedi): entrambe possono comportare un processo di **inferenza**, ovvero la derivazione di nuove formule a partire da quelle conosciute. L'inferenza deve soddisfare il requisito fondamentale che la risposta a ogni richiesta (Ask) posta alla base di conoscenza sia una conseguenza di quello che le è stato detto (attraverso Tell) in precedenza.

Un agente basato sulla conoscenza può essere costruito semplicemente dicendogli (Tell) ciò che deve sapere. Iniziando con una base di conoscenza vuota, il progettista dell'agente può utilizzare Tell su varie formule, passandole all'agente una per una, finché l'agente sa come operare nel proprio ambiente. Questo procedimento prende il nome di approccio **dichiarativo** alla realizzazione di sistemi. In contrapposizione a questo, l'approccio **procedurale** codifica direttamente in un programma i comportamenti desiderati sotto forma di codice.

Agenti basati sulla logica proposizionale

I sistemi basati su conoscenza (o Knowledge-based) sono sistemi per i quali le posizioni intenzionali sono giustificate/espresse/radicate nella rappresentazione in simboli (sin dal loro design). L'espressione simbolica della conoscenza è detta Base di Conoscenza.
La logica consente un'implementazione possibile:
- Rappresentazione: come l'insieme delle frasi in logica proposizionale o logica di primo ordine.
- Ragionamento: la capacità di dedurre le conseguenze logiche dei fatti/assiomi.

Un formalismo per la rappresentazione della conoscenza ha tre componenti:

1. Una sintassi: un linguaggio composto da un vocabolario e regole per la formazione delle frasi (enunciati).
2. Una semantica: che stabilisce una corrispondenza tra gli enunciati e fatti del mondo; se un agente ha un enunciato nella sua KB, crede che il fatto corrispondente sia vero nel mondo.
3. Un meccanismo inferenziale (codificato o meno tramite regole di inferenza come nella logica) che ci consente di inferire nuovi fatti (conseguenza logica).

La KB può essere vista come un insieme di formule o come una singola formula che asserisce tutte le formule individuali.
Un algoritmo (o procedura) di inferenza che deriva solo formule che sono conseguenze logiche si dice **corretto**; si dice anche che preserva la verità. Una procedura di inferenza non corretta, essenzialmente, genera formule a suo piacimento. Un'altra proprietà desiderabile è la **completezza**: un algoritmo di inferenza è completo se può derivare ogni formula che è conseguenza logica
L'ultimo aspetto da considerare è quello del **grounding**: il legame tra i processi di ragionamento logico e l'ambiente reale in cui si trova l'agente. Una risposta semplice è che il legame è creato dai sensori dell'agente. Per esempio, il nostro agente nel mondo del wumpus ha un rilevatore di

odori. Il programma agente crea una formula adeguata ogni volta che viene percepito un odore. Di conseguenza, ogni volta che quella formula è nella base di conoscenza, è vera nel mondo reale. Il significato e la verità delle formule percettive sono così definite dai processi di percezione e di costruzione sintattica che le producono. Regole generali sono prodotte da un processo di costruzione di formule chiamato **apprendimento**. L'apprendimento è però soggetto a errori.

Logica proposizionale: una logica molto semplice

Sintassi

La **sintassi** della logica proposizionale definisce le formule accettabili. Le **formule atomiche** consistono di un singolo **simbolo proposizionale**. Ogni simbolo rappresenta una proposizione che può essere vera o falsa. (P, Q, $W_{1,3}$, ...). Due simboli proposizionali hanno un valore prefissato: True o False. Si possono costruire **formule complesse** partendo da formule più semplici grazie all'uso delle parentesi e di operatori chiamati **connettivi logici**: ¬, ∧, ∨, ⇒, ⇔. (not, and, or, implicazione, sse). Possiamo omettere le parentesi assumendo questa precedenza tra gli operatori: ¬ > ∧ > ∨ > ⇒, ⇔.

Semantica

Con questo termine si intendono le regole usate per determinare il valore di verità di una formula nei confronti di un particolare modello. Nella logica proposizionale, un modello stabilisce semplicemente il valore di verità – *true* o *false* – di ogni simbolo proposizionale. Con tre simboli ci sono 2^3 = 8 possibili modelli. La semantica della logica proposizionale deve specificare come si fa, dato un modello, a calcolare il valore di verità di *qualsiasi* formula. Questo viene fatto ricorsivamente. Tutte le formule sono costruite partendo da formule atomiche e applicando i cinque connettivi. Per le formule atomiche è facile:

- *True* è vero in ogni modello e *False* è falso in ogni modello;
- il valore di verità di ogni altro simbolo proposizionale dev'essere specificato direttamente nel modello.

Per le formule complesse abbiamo cinque regole che valgono per qualsiasi sottoformula *P* e *Q* in ogni modello *m:*

- *¬P è vero sse P è falso in m.*
- *P ∧ Q è vero sse P e Q sono entrambi veri in m.*
- *P ∨ Q è vero sse P o Q è vero in m.*
- *P ⇒ Q è vero a meno che P sia vero e Q falso in m.*
- *P ⇔ Q è vero sse P e Q sono entrambi veri o entrambi falsi in m.*

Queste regole possono anche essere espresse con tavole di verità che specificano i valori di verità di una formula complessa per ogni possibile configurazione dei suoi componenti.

Logica dei predicati di prim'ordine (FOL)

- Il linguaggio: vocabolario
  - *Connettivo* → ∧ | ∨ |¬ | ⇒| ⇔| ⇐
  - *Quantificatore* → ∀ | ∃
  - *Variabile* → x|y| … a|s … (lettere minuscole)
  - *Costante* → Es. A | B | … Mario | Pippo| 2 …
  - *Funzione* → Es. *Hat* |*Padre-di*|+|-|…
    (con *arità* ≥ 1) 1 1 2 2
  - *Predicato* → Es. *On* | *Clear* | ≥ |< …
    (con *arità* ≥ 0) 2 1 2 2

Una **interpretazione** *I* stabilisce una corrispondenza precisa tra elementi atomici del linguaggio ed elementi della concettualizzazione.

Una formula A è conseguenza logica di un insieme di formule KB sse in ogni modello di KB, anche A è vera (KB |= A). Indicando con M(α) l'insieme delle interpretazioni che rendono α vera, cioè i modelli di α, e con M(KB) i modelli di un insieme di formule KB abbiamo che:
KB |= α sse M(KB) ⊆ M(α).

Dimostrazione di teoremi nella logica proposizionale

Il primo concetto è quello di **equivalenza logica**: due formule α e β sono logicamente equivalenti se sono vere nello stesso insieme di modelli. Questo si scrive $\alpha \equiv \beta$.

$$A \wedge B \equiv B \wedge A$$
$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

Il secondo concetto è quello di **validità**. Una formula è valida se è vera in *tutti* i modelli. Per esempio, la formula $P \vee \neg P$ è valida. Le formule valide sono note anche come tautologie e sono *necessariamente* vere. Dato che la formula *True* è vera in tutti i modelli, ogni formula valida è logicamente equivalente a *True*.

L'ultimo concetto è la **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. Il problema di determinare la soddisfacibilità delle formule della logica proposizionale, o problema **SAT**, è stato il primo problema di cui è stata dimostrata la NP-completezza. Nell'informatica molti problemi sono in realtà problemi di soddisfacibilità.

Naturalmente, validità e soddisfacibilità sono strettamente connesse: α è valida sse $\neg\alpha$ è insoddisfacibile; di contro, α è soddisfacibile sse $\neg\alpha$ non è valida.

Inferenza e dimostrazioni

Questo paragrafo tratta **regole di inferenza** che possono essere applicate per derivare una **dimostrazione** o prova, ovvero una catena di conclusioni che portano all'obiettivo desiderato. La regola più nota è il **Modus Ponens**, che si scrive come segue:

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

Un'altra utile regola di inferenza è l'eliminazione degli and in base alla quale, data una congiunzione, si può inferire uno qualsiasi dei congiunti:

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

Un'ultima proprietà dei sistemi logici è la **monotonicità**, che afferma che un insieme di formule che sono conseguenze logiche può solo *aumentare* man mano che si aggiunge informazione alla base di conoscenza.

Dimostrazione per risoluzione

La regola di inferenza di **risoluzione unitaria** prende una **clausola** - disgiunzione di letterali - dove l è un letterale e m è il letterale complementare.
La rimozione delle eventuali copie di letterali è chiamata **fattorizzazione**. Per esempio, se risolviamo ($A \vee B$) con ($A \vee \neg B$) otteniamo ($A \vee A$), che viene ridotto ad A mediante fattorizzazione.

Forma normale congiuntiva

La regola di risoluzione si applica solo a clausole (disgiunzioni di letterali), ragion per cui sembrerebbe utilizzabile solo su basi di conoscenza e query composte da clausole. *Ogni formula della logica proposizionale è logicamente equivalente a una congiunzione di clausole.*
Una formula così espressa viene detta in **forma normale congiuntiva**, CNF.

Un algoritmo di risoluzione

Per prima cosa si converte ($KB \wedge \neg\alpha$) in CNF; quindi si applica la regola di risoluzione alle clausole risultanti. Ogni coppia che contiene letterali complementari è risolta per produrre una nuova clausola che viene aggiunta all'insieme (se non vi è già presente). Il processo continua finché non si verifica una delle due 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 da *KB* consegue logicamente α.

(lez 7 qualche slide e libro cap 7.3)