

Introduzione all’Intelligenza Artificiale

Federico Matteoni

A.A. 2019/20

Indice

1 Agenti Intelligenti	7
1.1 Intelligenza	7
1.2 Agenti	7
1.2.1 Caratteristiche	8
1.2.2 Percezioni e Azioni	8
1.2.3 Agente e ambiente	8
1.2.4 Agenti Razionali	9
1.2.5 Agenti Autonomi	9
1.3 Ambienti	10
1.3.1 PEAS	10
1.3.2 Simulatore di Ambienti	10
1.3.3 Proprietà dell'Ambiente-Problema	11
1.4 Struttura di un Agente	12
1.4.1 Strutture di Agenti Caratteristici	12
1.4.2 Tipi di rappresentazione	15
2 Problem Solving	17
2.1 Agenti Risolutori di Problemi	17
2.1.1 Processo di risoluzione	17
2.2 Algoritmi di Ricerca	18
2.2.1 Ricerca ad Albero	18
2.2.2 Breadth-First	19
2.2.3 Depth-First	21
2.2.4 Depth-First ricorsiva	21
2.2.5 Depth-Limited	21
2.2.6 Iterative-Deepening	22
2.3 Direzione della Ricerca	22
2.4 Problematiche	23
2.4.1 Tre soluzioni	24
2.5 Uniform-Cost	24
2.6 Confronto delle Strategie (albero)	25
2.7 Conclusioni	25
3 Ricerca Euristica	27
3.1 Funzione di Valutazione Euristica	27
3.2 Best-First	27
3.2.1 Algoritmo A	28
3.2.2 Algoritmo A*: La Stima Ideale	29
3.2.3 Perché A* è vantaggioso?	30
3.2.4 Conclusioni su A*	31
3.2.5 Casi speciali di A	31
3.3 Costruire le Euristiche di A*	32
3.3.1 Valutazione di funzioni euristiche	32
3.3.2 Confronto di euristiche ammissibili	32
3.3.3 Misura del potere euristico	32
3.3.4 Capacità di esplorazione	33
3.4 Come si inventa un'euristica?	33

3.4.1	Rilassamento del problema	33
3.4.2	Massimizzazione di euristiche	33
3.4.3	Pattern Disgiunti	34
3.4.4	Apprendere dall'esperienza	34
4	Algoritmi Evolutivi Basati su A*	35
4.1	Beam Search	35
4.2	IDA*	35
4.3	RBFS	35
4.4	A* con memoria limitata	35
4.5	Conclusioni	36
5	Oltre la Ricerca Classica	37
5.1	Verso ambienti più realistici	37
5.2	Ricerca Locale	37
5.2.1	Algoritmi di ricerca locale	37
5.2.2	Panorama dello spazio degli stati	38
5.2.3	Algoritmo Hill Climbing	38
5.2.4	Algoritmo di Tempra Simulata	40
5.2.5	Algoritmo Local Beam	40
5.2.6	Algoritmo Beam Search Stocastico	41
5.3	Algoritmi Genetici	41
5.3.1	Gradient	42
5.4	Ambienti più realistici	43
5.4.1	Azioni non deterministiche	43
5.4.2	Come si pianifica	43
6	I Giochi con Avversario	45
6.1	Giochi con Avversario	45
6.1.1	Ciclo <i>pianifica-agisci-percepisci</i>	45
6.2	Giochi come problemi di ricerca	46
6.2.1	Algoritmo MinMax	46
6.2.2	Algoritmo Min-Max Euristico (con orizzonte)	48
6.2.3	Potatura Alfa-Beta	49
7	Problemi di Soddisfacimento di Vincoli	51
7.1	Formulazione	51
7.2	Strategie per problemi CSP	51
7.2.1	Ricerca in problemi CSP	52
7.2.2	Backtracking	52
8	Agenti Basati su Conoscenza	55
8.1	Agenti Knowledge-Based	55
8.1.1	Il mondo del Wumpus	56
8.1.2	Knowledge-Base	56
8.1.3	Algoritmo TT-entails	57
8.2	Algoritmi per la soddisfacibilità (SAT)	58
8.2.1	Algoritmo DPLL	59
8.2.2	Metodi locali per SAT	60
8.2.3	Algoritmo WalkSAT	60
8.3	Inferenza come Deduzione	61
8.3.1	Regola di risoluzione per PROP	62
8.3.2	Logica del Primo Ordine	63
8.3.3	Inferenza nella Logica del Prim'Ordine	63
8.3.4	Teorema di Herbrand	63
8.4	Definizione e Confronto di Euristiche Ammissibili	64

9 Strategie di risoluzione	65
9.1 Strategia di risoluzione	65
9.1.1 Strategie di Cancellazione	65
9.1.2 Strategie di Restrizione	66
9.1.3 Sottoinsieme a regole del FOL	66
9.1.4 Sistemi a regole logici	66
9.1.5 Programmazione Logica	66
9.1.6 Risoluzione SLD	66

Introduzione

Alessio Micheli, Maria Simi

elearning.di.unipi.it/course/view.php?id=174

Intelligenza Artificiale si occupa della **comprendione** e della **riproduzione** del comportamento *intelligente*.

Psicologia cognitiva: obiettivo comprendione intelligenza umana, costruendo modelli computazionali e verifica sperimentale.

Approccio costruttivo: costruire entità dotate di intelligenze e **razionalità**. Questo tramite codifica del pensiero razionale per risolvere problemi che richiedono intelligenza non necessariamente facendolo come lo fa l'uomo.

Definizioni di IA: pensiero-azione, umanamente-razionalmente.

Costruire macchine intelligenti sia che operino come l'uomo che diversamente.

formalizzaz conoscenze e meccanizzazione ragionemtno in tutti i settori dell'uomo

comprendione tramite modelli comp della psicologia e comportamento di uomini, animali ecc

rendere il lavoro con il calcolatore altrettanto facile e utile che del lavoro con persone capaci, abili e disponibili.

Poniamo definizione di IA: arte di creare macchine che svolgono funzioni che richiedono intelligenza quando svolte da esseri umani. Non definisce "Intelligenza", cosa significa "intelligente"?

Capitolo 1

Agenti Intelligenti

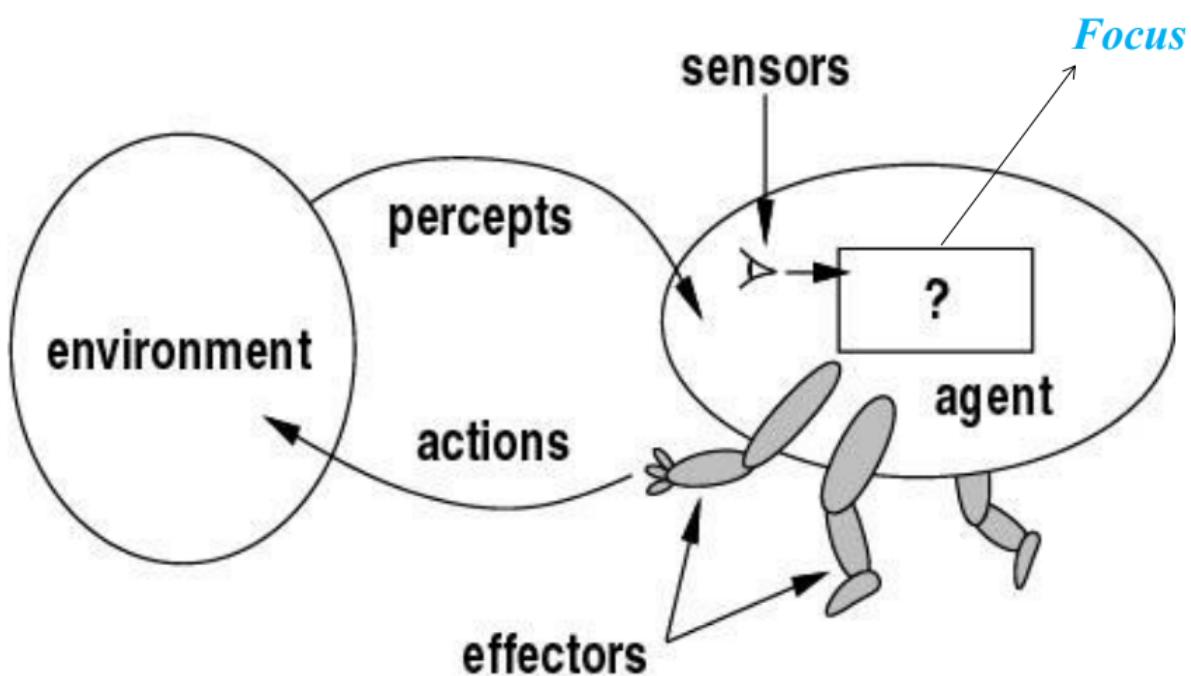
1.1 Intelligenza

L'intelligenza è vista come l'avere diverse capacità, durante il progresso nell'area di ricerca: buon senso, interazione con un ambiente, acquisizione di esperienza, comunicazione, ragionamento logico...

Considerazioni L'intelligenza quindi non è una collezione di tecniche per risolvere problemi **specifici**, ma per l'informatica consiste nel **fornire metodologie sistematiche per dotare le macchine di comportamenti intelligenti/razionali su problemi generali difficili**.

1.2 Agenti

Iniziamo con inquadrare gli **agenti**. L'approccio moderno dell'IA consiste della costruzione di agenti intelligenti. Questa visione ci offre un quadro di riferimento ed una prospettiva **diversa** all'analisi dei sistemi software. Il primo obiettivo sarà di costruire agenti per la risoluzione di problemi vista come una **ricerca in uno spazio di stati (problem solving)**



Ciclo percezione- azione

1.2.1 Caratteristiche

Sono qualcosa di più di un modulo software.

Situati Gli agenti sono situati in un ambiente da cui **ricevono percezioni** e su cui **agiscono** mediante **azioni (attuatori)**.

Sociali Gli agenti hanno **abilità sociali**: comunicano, collaborano e si difendono da altri agenti.

Credenze, obiettivi, intenzioni...

Corpo Gli agenti hanno un **corpo**, sono **embodied** fino a considerare i meccanismi delle emozioni.

1.2.2 Percezioni e Azioni

Percezione Una percezione è un input da sensori.

Sequenza percettiva Storia **completa** delle percezioni

La scelta delle azioni è **unicamente determinata dalla sequenza percettiva**.

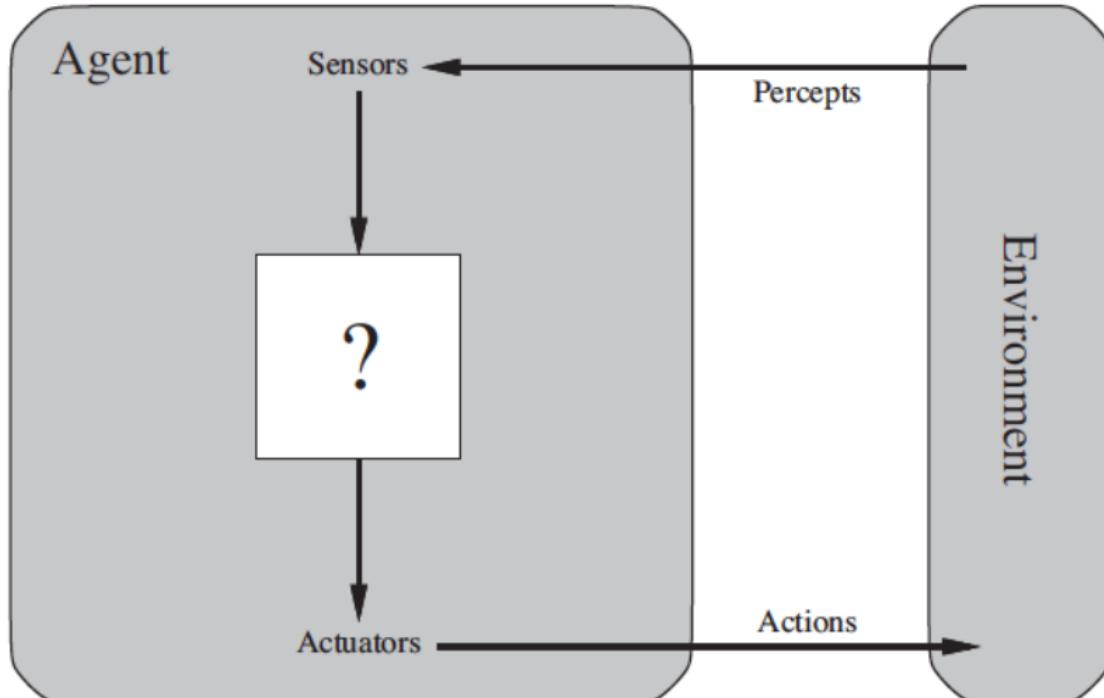
Funzione Agente Definisce l'azione da intraprendere per ogni sequenza percettiva e **descrive completamente** l'agente. Implementata da un **programma agente**.

$$\text{Sequenza Percettiva} \xrightarrow{f} \text{Azione}$$

Il compito dell'IA è progettare il programma agente.

1.2.3 Agente e ambiente

Architettura astratta



Esempi

Agente robotico Percepisce con camera, microfoni e sensori. Interagisce con motori, voce...

Agente finanziario Percepisce i tassi, le news. Interagisce con acquisti e scambi.

Agente di gioco Percepisce le mosse dell'avversario. Interagisce tramite le proprie mosse.

Agente diagnostico Percepisce i sintomi e le analisi dei pazienti. Interagisce fornendo la diagnosi.

Agente web Percepisce le query utente e le pagine web. Interagisce fornendo i risultati di ricerca.

1.2.4 Agenti Razionali

Agenti razionali Un agente razionale **interagisce con l'ambiente in maniera efficace**: *"fa la cosa giusta"*. L'agente razionale raggiunge l'obiettivo nella maniera più efficiente.

Serve quindi una **misura di prestazione**, di *come vogliamo che il mondo evolva*, a seconda del problema e considerato l'ambiente.

Esterna, perché bisogna definirla *prima* di agire. Non si può definire l'obiettivo dopo aver iniziato ad agire, altrimenti non è significativo.

Esempio: la volpe che non arriva all'uva.

Scelta dal progettista a seconda del problema e considerando l'effetto che ha sull'ambiente.

Razionalità La razionalità è relativa/dipende da:

Misura delle prestazioni

Conoscenze pregresse dell'ambiente

Percezioni presenti e passate (sequenza percettiva)

Capacità dell'agente (le azioni possibili)

Definizione Un **agente razionale**, quindi, **esegue l'azione che massimizza il valore atteso della misura delle prestazioni per ogni sequenza di percezioni**, considerando le sue percezioni passate e la sua conoscenza pregressa.

Non si pretende perfezione e conoscenza del futuro, ma massimizzare il risultato *atteso*. Potrebbero essere necessarie azioni di acquisizione di informazioni o esplorative (**non onniscienza**).

Le capacità dell'agente possono essere limitate (**non onnipotenza**).

Razionalità e apprendimento Raramente il programmatore può fornire a priori tutta la conoscenza sull'ambiente. L'agente razionale, quindi, **dove essere in grado di modificare il proprio comportamento con l'esperienza**, cioè con le percezioni passate.

Può migliorarsi esplorando, **apprendendo**, aumentando la propria autonomia per operare in ambienti differenti o mutevoli.

1.2.5 Agenti Autonomi

Un agente è **autonomo quando il suo comportamento dipende dalla sua esperienza**. Se il suo comportamento fosse determinato solo dalla propria conoscenza *built-int* allora sarebbe **non autonomo** e poco flessibile.

1.3 Ambienti

Definire un problema per un agente significa **caratterizzare l'ambiente in cui lavora**, cioè l'**ambiente operativo**. L'agente razionale è la soluzione del problema.

1.3.1 PEAS

Performance, prestazioni

Environment, ambiente

Actuators, attuatori

Sensors, sensori

Esempio Autista di taxi

Prestazione	Ambiente	Attuatori	Sensori
Arrivare alla destinazione, sicuro, veloce, ligio alla legge, confortevole, consumo minimo di benzina, profitti massimi	Strada, altri veicoli, clienti	Sterzo, acceleratore, freni, frecce, clacson	Telecamere, sensori, GPS, contachilometri, accelerometro, sensori del motore...

Formulazione PEAS dei problemi

Problema	P	E	A	S
Diagnosi medica	Diagnosi corretta	Pazienti, ospedale	Domande, suggerimenti, test, diagnosi	Sintomi, test clinici, risposte del paziente
Analisi immagini	Numero di immagini/zona correttamente classificate	Collezione di fotografie	Etichettatore di zone nell'immagine	Array di pixel
Robot "selezionatore"	Numero delle parti correttamente classificate	Nastro trasportatore	Raccogliere le parti e metterle nei cestini	Telecamera (pixel di varia intensità)
Giocatore di calcio	Fare più goal dell'avversario	Altri giocatori, campo di calcio, porte	Dare calci al pallone, correre	Locazione del pallone, dei giocatori e delle porte

1.3.2 Simulatore di Ambienti

Uno **strumento software** con il compito di:

Generare gli stimoli per gli agenti

Raccogliere le azioni in risposta

Aggiornare lo stato dell'ambiente

Opzionalmente, attivare altri processi che influenzano l'ambiente

Valutare le prestazioni degli agenti

Gli esperimenti su classi di ambienti (variando le condizioni) sono essenziali per valutare la capacità di generalizzare. La valutazione delle prestazioni è fatta tramite la media su più istanze.

1.3.3 Proprietà dell'Ambiente-Problema

- Osservabilità

Completemente osservabile: l'apparato percettivo è in grado di dare una conoscenza completa dell'ambiente o almeno tutto quello che serve a decidere l'azione.

Parzialmente osservabile: sono presenti limiti o inaccuratezze nell'apparato sensoriale. (Es. la videocamera di un rover vede solo parte dell'ambiente in un dato istante).

- Singolo/Multi-Agente

Distinzione tra agente e non agente: il mondo può cambiare anche attraverso **eventi**, non necessariamente per le azioni di agenti.

Multi-Agente Competitivo, come gli scacchi: comportamento randomizzato ma razionale.

Multi-Agente Cooperativo, o benigno: stesso obiettivo e comunicazione.

- Predicibilità

Deterministico: lo stato successivo è completamente determinato dallo stato corrente e dall'azione.

Stocastico: esistono elementi di incertezza con probabilità associata. Es: guida, tiro in porta.

Non deterministico: si tiene traccia di più stati possibili che sono risultato dell'azione, ma non in base ad una probabilità.

- Episodico:

l'esperienza dell'agente è divisa in episodi atomici indipendenti. In ambienti episodici non c'è bisogno di pianificare.

Sequenziale: ogni decisione influenza le successive.

- Statico:

il mondo non cambia mentre l'agente decide l'azione.

Dinamico: l'ambiente cambia nel tempo, va osservata la contingenza. Tardare equivale a non agire.

Semi-dinamico: l'ambiente non cambia ma la valutazione dell'agente sì. Es: scacchi con timer, se non agisco prima dello scadere perdo.

- Discreto/Continuo

Lo stato, il tempo, le percezioni e le azioni sono tutti elementi che possono assumere valori discreti o continui. Combinatoriale (nel discreto) *vs* infinito (nel continuo).

- Noto/Ignoto

Distinzione riferita allo stato di conoscenza dell'agente sulle leggi fisiche dell'ambiente. **L'agente conosce l'ambiente o deve compiere azioni esplorative?**

Noto ≠ osservabile: posso giocare a carte coperte, ma con regole note.

Ambienti reali Parzialmente osservabili, stocastici, sequenziali, dinamici, continui, multi-agente e ignoti.

1.4 Struttura di un Agente

$$\text{Agente} = \text{Architettura} + \text{Programma}$$

$$\text{Ag: P} \rightarrow \text{Az}$$

L'Agente associa **Azioni** alle **Percezioni**. Il **programma dell'agente** implementa la funzione **Ag**.

Programma Agente Pseudocodice del programma agente.

```
function Skeleton-Agent(percept) returns action
    static: memory #la memoria del mondo posseduta dall'agente
    memory <- UpdateMemory(memory, percept)
    action <- Choose-Best-Action(memory) #Cuore dell'IA
    memory <- UpdateMemory(memory, action)
    return action
```

1.4.1 Strutture di Agenti Caratteristici

Agente basato su tabella La scelta dell'azione è un accesso ad una tabella che **associa un'azione ad ogni possibile sequenza di percezioni**.

Vari problemi:

Le **dimensioni** possono essere proibitive: per giocare a scacchi, la tabella dovrebbe contenere un numero di righe nell'ordine di $10^{120} >> 10^{80}$ numero di atomi nell'universo osservabile.

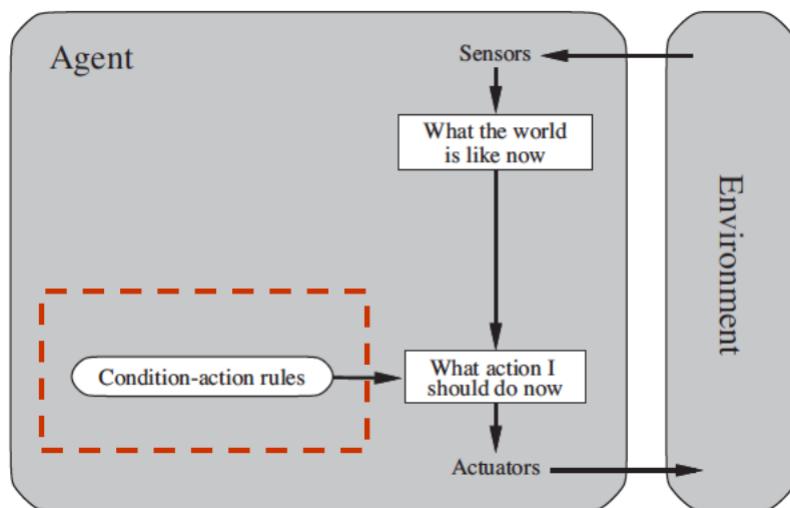
Difficile da costruire

Nessuna autonomia

Difficile da aggiornare, apprendimento complesso.

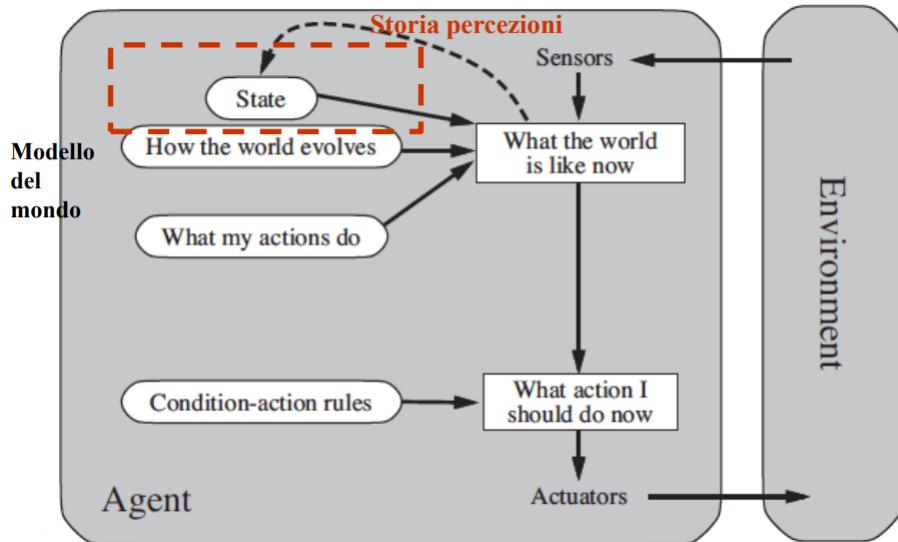
Con le IA vogliamo realizzare **automi razionali con un programma compatto**.

Agente Reattivo Semplice



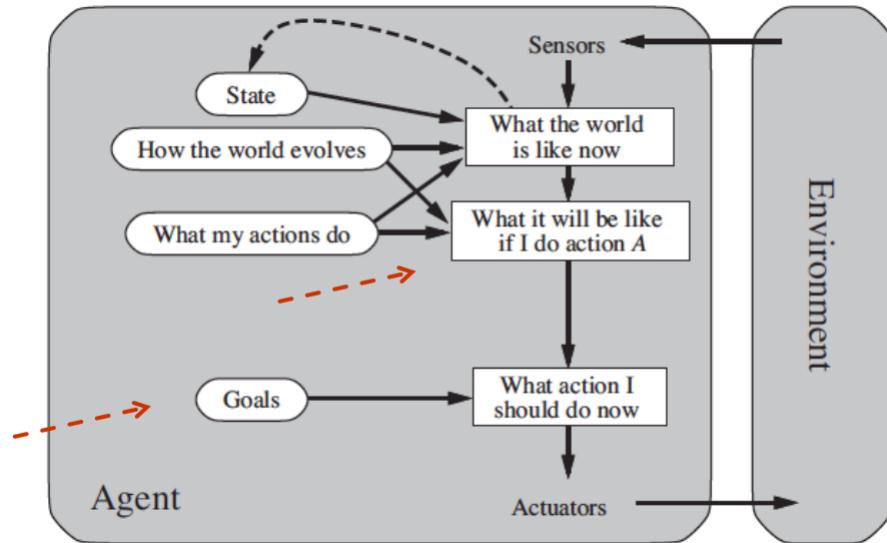
```
function Agente-Reattivo-Semplice(percezione) returns azione
    persistent: regole #insieme di regole condizione-azione (if-then)
    stato <- Interpreta-Input(percezione)
    regola <- Regola-Corrispondente(stato, regole)
    azione <- regola.Azione
    return azione
```

Agenti basati su modello



```
function Agente-Basato-su-Modello(percezione) returns azione
    persistent:      stato #descrizione dello stato corrente
                      modello #conoscenza del mondo
                      regole #insieme di regole condizione-azione
                      azione #azione piu recente
    stato <- Aggiorna-Stato(stato, azione, percezione, modello)
    regola <- Regola-Corrispondente(stato, regole)
    azione <- regola.Azione
    return azione
```

Agenti con obiettivo Bisogna pianificare una sequenza di azioni per raggiungere l'obiettivo. (In rosso sono indicate le parti aggiunte)



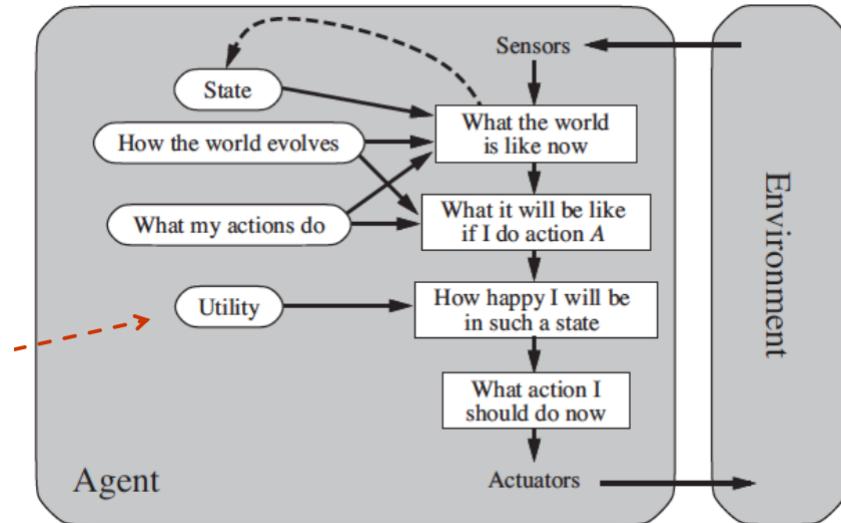
Sono guidati da un obiettivo nella scelta che intraprendono, è stato fornito un **goal esplicito**: per esempio una città da raggiungere.

A volte l'azione migliore dipende dall'obiettivo da raggiungere (*da che parte devo girare?*)

Devo **pianificare una sequenza di azioni** per raggiungere l'obiettivo. Sono meno efficienti ma **più flessibili** rispetto ad un agente reattivo. L'obiettivo può cambiare, non è codificato nelle regole.

Esempio classico: ricerca della sequenza di azioni per raggiungere una data destinazione.

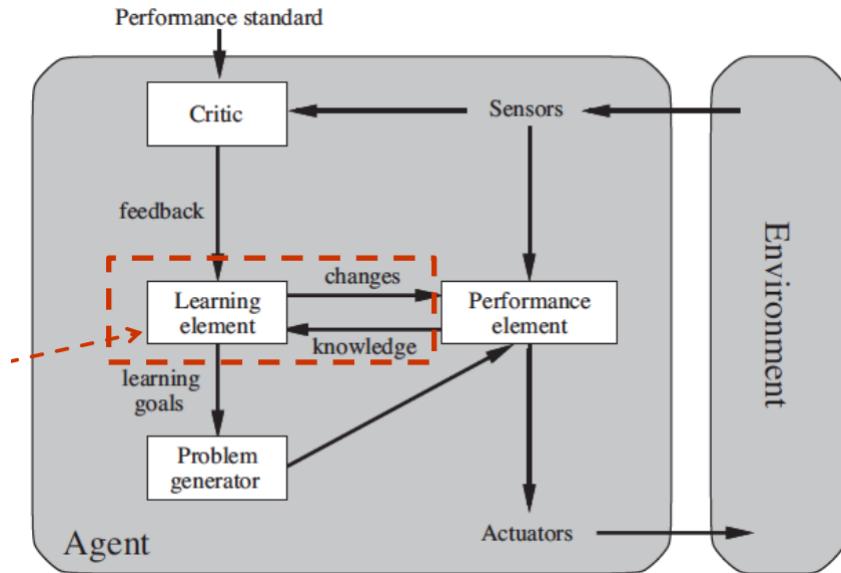
Agenti con valutazione di utilità



Obiettivi alternativi, o più modi per raggiungerlo: l'agente deve decidere verso quali muoversi, quindi è **necessaria una funzione di utilità** che associa ad uno stato obiettivo un numero reale.

Obiettivi più facilmente raggiungibili di altri: la funzione di utilità **tiene conto della probabilità di successo e/o di ciascun risultato (utilità attesa o media)**

Agenti che apprendono



Componente di apprendimento: produce cambiamenti al programma agente. Migliora le prestazioni, adattando i suoi componenti ed apprendendo dall'ambiente

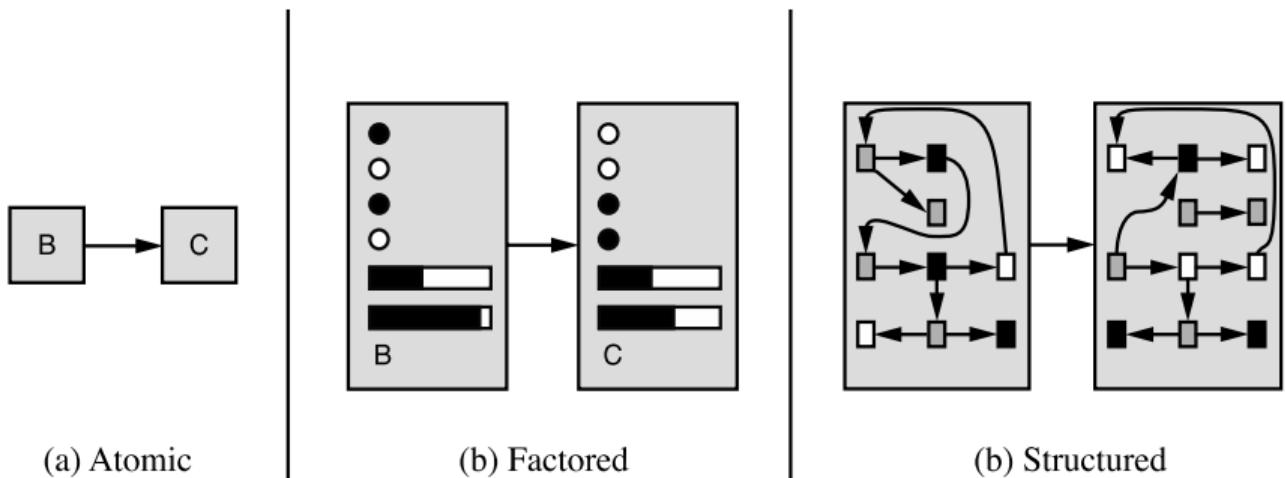
Elemento esecutivo: il programma agente

Elemento critico: osserva e dà feedback sul comportamento

Generatore di problemi: suggerisce nuove situazioni da esplorare

1.4.2 Tipi di rappresentazione

Stati e transizioni



Rappresentazione atomica (stati)

Rappresentazione fattorizzata (+ variabili e attributi)

Rappresentazione strutturata (+ relazioni)

Capitolo 2

Problem Solving

2.1 Agenti Risolutori di Problemi

Problem Solving Questi agenti adottano il paradigma della **risoluzione di problemi** come ricerca in uno spazio di stati (**problem solving**). Sono **agenti con modello** (storia, percezioni) che **adottano una rappresentazione atomica dello stato**. Sono **particolari agenti con obiettivo** che **pianificano l'intera sequenza di azioni** prima di agire.

2.1.1 Processo di risoluzione

Passi da seguire

1. **Determinazioni dell'obiettivo:** un insieme di stati dove l'obiettivo è soddisfatto.
2. **Formulazione del problema:** rappresentazione degli stati e delle azioni.
Fa parte del design "umano".
3. **Determinazione della soluzione** mediante ricerca: un piano d'azione
4. **Esecuzione del piano**
Soluzione algoritmica.

La determinazione dell'obiettivo e la formulazione del problema richiede **tanta intelligenza**, che in fase di design è **spostata sull'umano**. Gli algoritmi sono ancora **"stupidi"**.

Assunzioni sull'ambiente **Statico, osservabile** (so dove sono, es: *viaggio con la mappa*), **discreto** (insieme finito di azioni possibili), **deterministico** (una azione \Rightarrow un risultato. L'agente può eseguire il piano *"ad occhi chiusi"*, niente può andare storto)

Formulazione del problema Un problema può essere **definito formalmente** mediante cinque componenti:

1. **Stato iniziale**
2. **Azioni possibili** nello stato s : **Azioni(s)**
3. **Modello di transizione**
Risultato: stato \times azione \longrightarrow stato
Risultato(s, a): s' , uno stato **successore**
4. **Test obiettivo:** un insieme di stati obiettivo
Goal-Test: stato $\longrightarrow \{\text{true}, \text{false}\}$
5. **Costo del cammino:** somma dei costi delle azioni (costo dei passi).
Costo di un passo: $c(s, a, s')$, mai negativo.

1, 2 e 3 **definiscono implicitamente lo spazio degli stati**. Definirlo esplicitamente può essere molto oneroso, come in quasi tutti i problemi di IA.

2.2 Algoritmi di Ricerca

Il processo che cerca una sequenza di azioni che raggiunge l'obiettivo è detto ricerca.

Algoritmi Gli algoritmi di ricerca prendono in **input** un problema e restituiscono un cammino soluzione, un cammino che porta dallo stato iniziale allo stato goal.

Misura delle prestazioni Trova una soluzione? Quanto costa trovarla? Quanto è efficiente la soluzione?

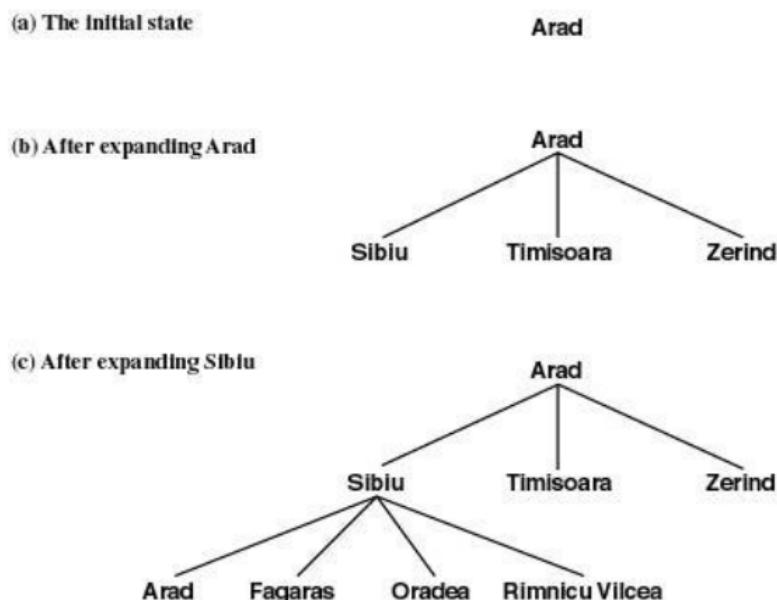
$$\text{Costo Totale} = \text{Costo della Ricerca} + \text{Costo del Cammino Soluzione}$$

Valuteremo algoritmi sul primo, ottimizzando il secondo.

2.2.1 Ricerca ad Albero

Generazione di un **albero di ricerca sovrapposto allo spazio degli stati**. Ricerca significa **approfondire l'opzione**, mettendo da parte le altre che verranno riprese se non trovo la soluzione.

Quindi l'albero viene generato esplorando i vari nodi partendo dallo stato iniziale. Il nodo è diverso dallo stato: per esempio, in un grafo rappresentante le città, se parto da città A ed esploro l'opzione nodo B, il nodo B avrà come figlio anche città A perché posso tornarci.



Algoritmo Ricerca ad albero, ossia senza controllare se i nodi (**stati**) siano già stati esplorati.

```

function Ricerca-Albero(problema) returns soluzione oppure fallimento
    #Inizializza la frontiera con stato iniziale del problema
    loop do
        if (frontiera vuota)
            return fallimento
        #Scegli* un nodo foglia da espandere e rimuovilo dalla frontiera
        if (nodo contiene uno stato obiettivo)
            return soluzione corrispondente
        #Espandi il nodo e aggiungi i successori alla frontiera

```

* = **strategia**: quale scegliere? I vari algoritmi si differenziano per la strategia di scelta.

Un **nodo** n è una **struttura dati con quattro componenti**

Stato, n.estado

Padre, n.padre

Azione effettuata per generarlo, n.azione

Costo del cammino dal nodo iniziale al nodo, n.costo-cammino

Indicata come $g(b) = \text{padre.costo-cammino} + \text{costo-passo ultimo}$

Frontiera Lista dei **nodi in attesa di essere espansi**, cioè le foglie dell'albero di ricerca. Implementata come una coda con operazioni:

- Vuota(coda)
- Pop(coda) estrae l'ultimo elemento (implementa la strategia)
- Inserisci(elemento, coda)

Diversi tipi di coda hanno differenti funzioni di inserimento e **implementano strategie diverse**.

FIFO → BF

Viene estratto l'elemento più vecchio, cioè in attesa da più tempo. Nuovi nodi aggiunti alla fine

LIFO → DF

Viene estratto l'ultimo elemento inserito. Nuovi nodi aggiunti all'inizio

Con priorità → UC, altri...

Viene estratto l'elemento con priorità più alta in base ad una funzione di ordinamento. All'aggiunta di un nuovo nodo si riordina.

Strategie non informate

- Ricerca in **ampiezza** (BF)
- Ricerca in **profondità** (DF)
- Ricerca in **profondità limitata** (DL)
- Ricerca con **apprendimento iterativo** (ID)
- Ricerca di **costo uniforme** (UC)

Strategie informate Anche dette di **ricerca euristica**: fanno uso di informazioni riguardo la distanza stimata della soluzione.

Valutazione di una strategia

Completezza: se la soluzione esiste viene trovata

Ottimalità (ammissibilità): trova la soluzione migliore, con costo minore

Complessità in tempo: tempo richiesto per trovare la soluzione

Complessità in spazio: memoria richiesta

2.2.2 Breadth-First

Ricerca in ampiezza Esplorare il grafo dello spazio degli stati a livelli progressivi di stessa profondità. Implementata con una coda FIFO. **Algoritmo su albero**:

```
function RicercaAmpiezzaA(problema)      returns soluzione oppure fallimento
    nodo = un nodo con stato = problema.stato-iniziale e costo-di-cammino = 0
    #Stati goal-tested alla generazione: maggior efficienza si ferma appena trova goal
    if (problema.TestObiettivo(nodo.Stato)) return Soluzione(nodo)
    frontiera = una coda FIFO con nodo come unico elemento
    loop do
        if (Vuota(frontiera)) return fallimento
        nodo = Pop(frontiera)
        for each azione in problema.Azioni(nodo.Stato) do #Espansione
            figlio = Nodo-Figlio(problema, nodo, azione) #costruttore: vedi AIMA
            if (Problema.TestObiettivo(figlio.Stato)) return Soluzione(figlio)
            frontiera = Inserisci(figlio, frontiera) #frontiera coda FIFO
```

Algoritmo su grafo evitando di espandere stati già esplorati:

```
function RicercaAmpiezzaG(problema) returns soluzione oppure fallimento
    nodo = un nodo con stato = problema.stato-iniziale e costo-di-cammino = 0
    if (problema.TestObiettivo(nodo.Stato)) return Soluzione(nodo)
    frontiera = una coda FIFO con nodo come unico elemento
    esplorati = insieme vuoto #gestisco stati ripetuti
    loop do
        if (Vuota(frontiera)) return fallimento
        nodo = POP(frontiera) #aggiungi nodo.Stato a esplorati
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione)
            if (figlio.Stato non in esplorati e non in frontiera)
                if (Problema.TestObiettivo(figlio.Stato)) return Soluzione(figlio)
                frontiera = Inserisci(figlio, frontiera) #in coda
```

Python

```
def breadth_first_search(problem): """Ricerca-grafo in ampiezza"""
    explored = [] # insieme degli stati già visitati (implementato come una lista)
    node = Node(problem.initial_state) #il costo del cammino è inizializzato nel costruttore
    if problem.goal_test(node.state):
        return node.solution(explored_set = explored)
    frontier = FIFOQueue() # la frontiera è una coda FIFO
    frontier.insert(node)
    while not frontier.isempty(): # seleziona il nodo per l'espansione
        node = frontier.pop()
        explored.append(node.state) # inserisce il nodo nell'insieme dei nodi esplorati
        for action in problem.actions(node.state):
            child_node = node.child_node(problem, action)
            if (child_node.state not in explored) and
            (not frontier.contains_state(child_node.state)):
                if problem.goal_test(child_node.state):
                    return child_node.solution(explored_set = explored)
                # se lo stato non è uno stato obiettivo allora inserisci il nodo nella frontiera
                frontier.insert(child_node)
    return None # in questo caso ritorna con fallimento
```

Analisi della complessità spazio-temporiale Assumiamo:

b = fattore di ramificazione (**branching**)

d = profondità del nodo obiettivo più superficiale (**depth**)
Più vicino all'iniziale

m = lunghezza massima dei cammini nello spazio degli stati (**max**)

Analisi:

Strategia **completa**

Strategia **ottimale** se gli operatori hanno tutti lo stesso costo k cioè $g(n) = k \cdot \text{depth}(n)$, dove $g(n)$ è il costo del cammino per arrivare ad n .

Complessità nel tempo (nodi generati)
 $T(b, d) = b + b^2 + \dots + b^d = O(b^d)$, con b figli per ogni nodo.

Complessità nello spazio (nodi in memoria): $O(b^d)$

2.2.3 Depth-First

Ricerca in profondità Implementata da una coda che mette i successori in testa alla lista (LIFO, pila o stack). Algoritmo generale visto all'inizio, su grafo o albero.

Analisi (su albero) Poniamo m lunghezza massima dei cammini nello spazio degli stati e b fattore di diramazione Tempo: $O(b^m)$ che può essere anche $> O(b^d)$

Spazio: $b \cdot m$, frontiera sul cammino perché vengono cancellati i rami completamente esplorati ma mantenuti i fratelli del path corrente.

Non completa (loop) e **non ottimale**, ma drastico risparmio di memoria.

BF, $d = 16 \rightarrow 10$ Esabyte

DF, $d = 16 \rightarrow 156$ Kilobyte

Analisi (su grafo) In caso di DF su grafo si perdono i vantaggi di memoria: torna a tutti i possibili stati (al caso pessimo diventa esponenziale come BF) per mantenere la lista dei visitati, ma così DF diventa **completa** in spazi degli stati finiti (al caso pessimo tutti i nodi vengono espansi).

Rimane non completa in spazi infiniti.

Possibile controllare anche solo i nuovi stati rispetto al cammino radice-nodo corrente senza aggravio di memoria. Si evitano i cicli finiti in spazi finiti ma non i cammini ridondanti.

2.2.4 Depth-First ricorsiva

Ancora più efficiente in occupazione di memoria perché mantiene solo il cammino corrente (m nodi al caso pessimo). Realizzata da un algoritmo ricorsivo "con backtracking" che non necessita di tenere in memoria b nodi per ogni livello, ma salva lo stato su uno stack a cui torna in caso di fallimento per fare altri tentativi. **Algoritmo su albero**:

```
function Ricerca-DF-A (problema) returns soluzione oppure fallimento
    return Ricerca-DF-ricorsiva(CreaNodo(problema.Stato-iniziale), problema)

function Ricerca-DF-ricorsiva(nodo, problema) returns soluzione oppure fallimento
    if problema.TestObiettivo(nodo.Stato) return Soluzione(nodo)
    else
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione)
            risultato = Ricerca-DF-ricorsiva(figlio, problema)
            if risultato != fallimento then return risultato
    return fallimento
```

Python

```
def recursive_depth_first_search(problem, node):
    """Ricerca in profondità ricorsiva"""
    #controlla se lo stato del nodo è uno stato obiettivo
    if problem.goal_test(node.state):
        return node.solution()
    #in caso contrario continua
    for action in problem.actions(node.state):
        child_node = node.child_node(problem, action)
        result = recursive_depth_first_search(problem, child_node)
        if result is not None: return result
    return None #con fallimento
```

2.2.5 Depth-Limited

Ricerca in profondità limitata Si va in profondità fino ad un certo livello predefinito l .

Completa per problemi di cui si conosce un limite superiore per la profondità della soluzione: ad esempio route-finding limitata dal numero di città - 1

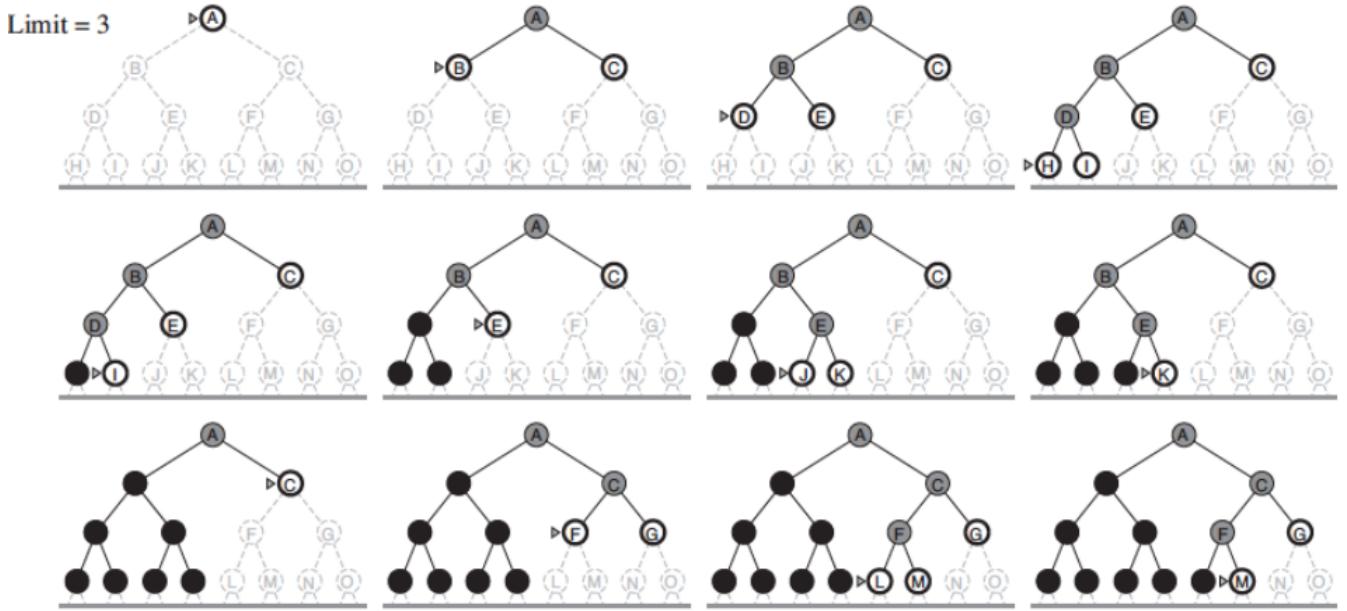
Completo se $d < l$

Non ottimale

Complessità in tempo: $O(b^l)$

Complessità in spazio: $O(b \cdot l)$

2.2.6 Iterative-Deepening



Analisi Miglior compromesso tra BF e DF. Nell'ID, i nodi dell'ultimo livello sono generati una volta, quelli del penultimo 2, del terzultimo 3... quelli del primo d volte.

$$\text{ID: } (d)b + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + b^d$$

Complessità in tempo: $O(b^d)$

Complessità in spazio: $O(b \cdot d)$, vs $O(b^d)$ del BF.

2.3 Direzione della Ricerca

Altro aspetto usato per ottimizzare la risoluzione di problemi, la **direzione della ricerca** è un problema ortogonale alla **strategia di ricerca**. La ricerca si può fare

In avanti, guidata dai dati come fatto fin'ora: si esplora lo spazio di ricerca dallo stato iniziale allo stato obiettivo

All'indietro o guidata dall'obiettivo: si esplora lo spazio di ricerca a partire da un goal e riconducendosi a sotto-goal fino a trovare uno stato iniziale.

Conviene procedere nella direzione in cui il fattore di diramazione è minore.

Si preferisce la **ricerca all'indietro** quando

l'**obiettivo è chiaramente definito** (es. theorem proving) o si possono **formulare una serie limitata di ipotesi**

i **dati del problema non sono noti** e la loro **acquisizione può essere guidata dall'obiettivo**

mentre si preferisce la **ricerca in avanti** quando

gli **obiettivi possibili sono molti** (es. design)

abbiamo una **serie di dati da cui partire**

Ricerca bidirezionale Si procede nelle due direzioni fino ad incontrarsi

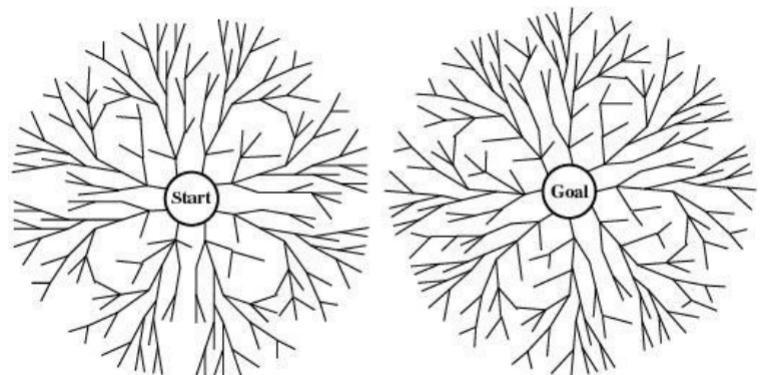
Complessità in tempo: $O(b^{d/2}) = O(\sqrt{b^d})$

Test intersezione in tempo costante, esempio: hash table

Complessità in spazio: $O(b^{d/2}) = O(\sqrt{b^d})$

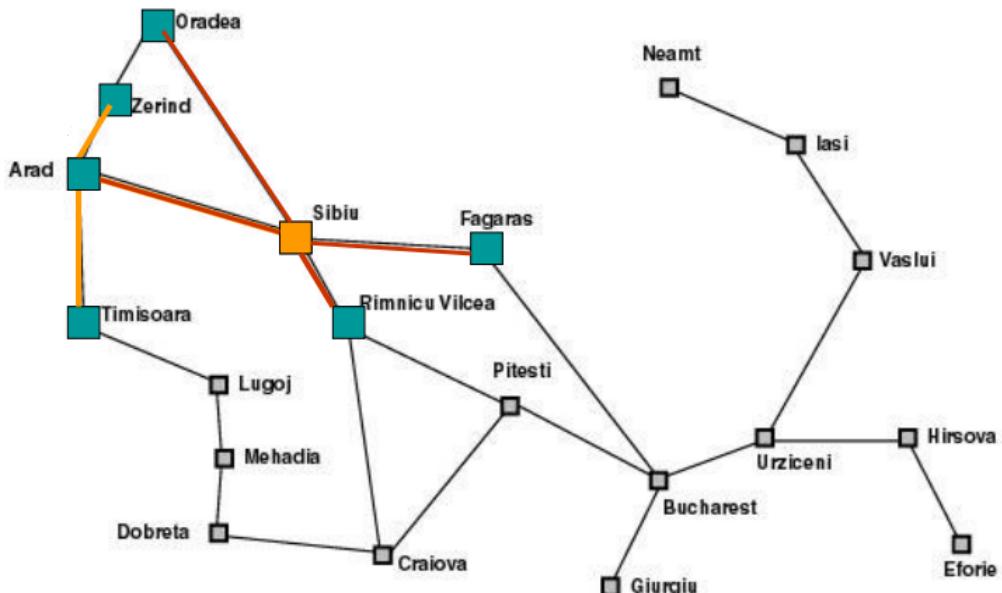
Almeno tutti i nodi in una direzione in memoria, esempio: usando BF

Non è sempre applicabile, ad esempio in casi di predecessori non definiti, troppi stati obiettivo...

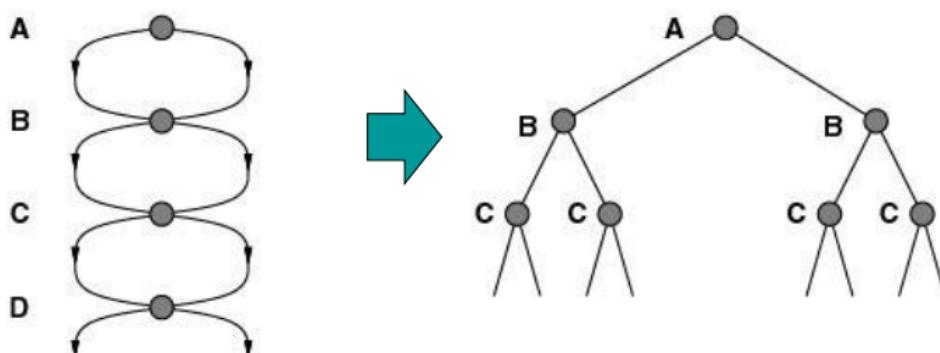


2.4 Problematiche

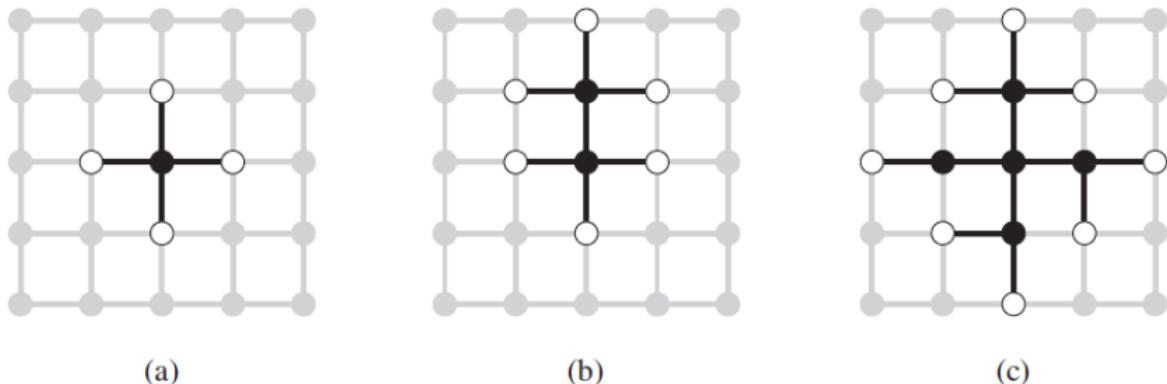
Cammini Ciclici I cammini ciclici potenzialmente rendono gli alberi di ricerca infiniti, anche se con stati finiti.



Ridondanze Su spazi di stati a grafo si generano più volte nodi con lo stesso stato nella ricerca, anche in assenza di cicli.



Un caso è la **ricerca nelle griglie** Visitare stati già visitati fa compiere lavoro inutile. Costo 4^d ma circa $2d^2$ stati distinti.



Come evitarlo?

Compromesso tra spazio e tempo Ricordare gli stati visitati **occupa spazio** ma ci **consente di evitare di visitarli di nuovo**. Gli algoritmi che dimenticano la propria storia sono destinati a ripeterla.

2.4.1 Tre soluzioni

In ordine crescente di costo ed efficacia:

Non tornare nello stato da cui si proviene: si elimina il genitore dai nodi successori.

Non evita i cammini ridondanti.

Non creare cammini con cicli: si controlla che i successori non siano antenati del nodo corrente.

Non generare nodi con stati già visitati/esplorati: ogni nodo visitato deve essere tenuto in memoria per una complessità $O(s)$ dove s è il numero di stati possibili (esempio: hash table per accesso efficiente)

Repetita Il costo può essere alto: in caso di DF la memoria torna da $b \cdot m$ a tutti gli stati, ma diventa una ricerca completa per spazi finiti. Ma **in molti casi gli stati crescono esponenzialmente** (scacchi...)

2.5 Uniform-Cost

Generalizzazione della ricerca in ampiezza (costi diversi tra passi): **si sceglie il nodo di costo $g(n)$ del cammino minore sulla frontiera**, si espande sui contorni di uguale costo (e.g. in km) invece che sui contorni di uguale profondità (BF). Implementata da una **coda ordinata per costo cammino crescente**. **Algoritmo su albero**:

```
function Ricerca-UC-A(problema) returns soluzione oppure fallimento
    nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
    frontiera = una coda con priorità con nodo come unico elemento
    loop do
        if Vuota?(frontiera) then return fallimento
        nodo = POP(frontiera)
        #Esame post-generaz e vedere costo minore, tipico per coda con priorità
        if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione)
            frontiera = Inserisci(figlio, frontiera) #in coda con priorità
    end
```

Algoritmo su grafo:

```

function Ricerca-UC-G(problema) returns soluzione oppure fallimento
    nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
    frontiera = una coda con priorita con nodo come unico elemento
    esplorati = insieme vuoto
    loop do
        if Vuota?(frontiera) then return fallimento
        nodo = POP(frontiera);
        if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
        aggiungi nodo.Stato a esplorati
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione)
            if figlio.Stato non in esplorati e non in frontiera then
                frontiera = Inserisci(figlio, frontiera) #in coda con priorita
            else if figlio.Stato in frontiera con Costo-cammino piu alto then
                sostituisci quel nodo frontiera con figlio

```

Analisi Ottimalità e completezza garantisce purché il costo degli archi sia maggiore di $\epsilon > 0$. Assunto C^* come il costo della soluzione ottima, allora $\lfloor C^*/\epsilon \rfloor$ numero di mosse al caso peggiore, arrotondato per difetto. Tendo ad andare verso tante mosse di costo ϵ prima di una che parta più alta ma poi abbia un path a costo più basso. Complessità: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.

Quando ogni azione ha lo stesso costo somiglia a BF ma con complessità $O(b^{1+d})$ perché esame e arresto solo dopo aver espanso anche l'ultima frontiera.

2.6 Confronto delle Strategie (albero)

Criteria	BF	UC	DF	DL	ID	Bidirez.
Completa?	Si	Si ¹	No	Si ³	Si	Si
Tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Spazio	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b \cdot m)$	$O(b \cdot l)$	$O(b \cdot d)$	$O(b^{d/2})$
Ottimale?	Si ²	Si ¹	No	No	Si ²	Si

¹ Per costi degli archi $\geq \epsilon > 0$

² Se gli operatori hanno tutti lo stesso costo

³ Per problemi di cui si conosce un limite alla profondità della soluzione (se $l > d$)

2.7 Conclusioni

Un agente per "problem solving" adotta un paradigma generale di risoluzione dei problemi:

Formula il problema, non automatico

Ricerca la soluzione nello spazio degli stati, automatico

Capitolo 3

Ricerca Euristica

In problemi di complessità esponenziale, come ad es. gli scacchi (10^{120} configurazioni) non è praticabile la ricerca esaustiva. Diventa quindi fondamentale **usare la conoscenza del problema e l'esperienza per riconoscere i cammini più promettenti**, evitando di generare gli altri (**pruning**).

Conoscenza Euristica La **conoscenza euristica** aiuta a fare scelte oculate:

Non evita la ricerca, ma la riduce

Consente, in genere, di trovare una **buona** soluzione in tempi accettabili

Sotto certe condizioni garantisce completezza e ottimalità

3.1 Funzione di Valutazione Euristica

La **conoscenza** del problema è data tramite una **funzione di valutazione** f , che include h detta **funzione di valutazione euristica**:

$$h : n \rightarrow R$$

R = insieme numeri reali. La funzione si applica al nodo, ma dipende solo dallo stato (n.Stato). Per confronto, g dipendeva anche dal cammino fino al nodo. Quindi, la funzione di valutazione

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

dove $g(n)$ è il costo del cammino visto con UC.

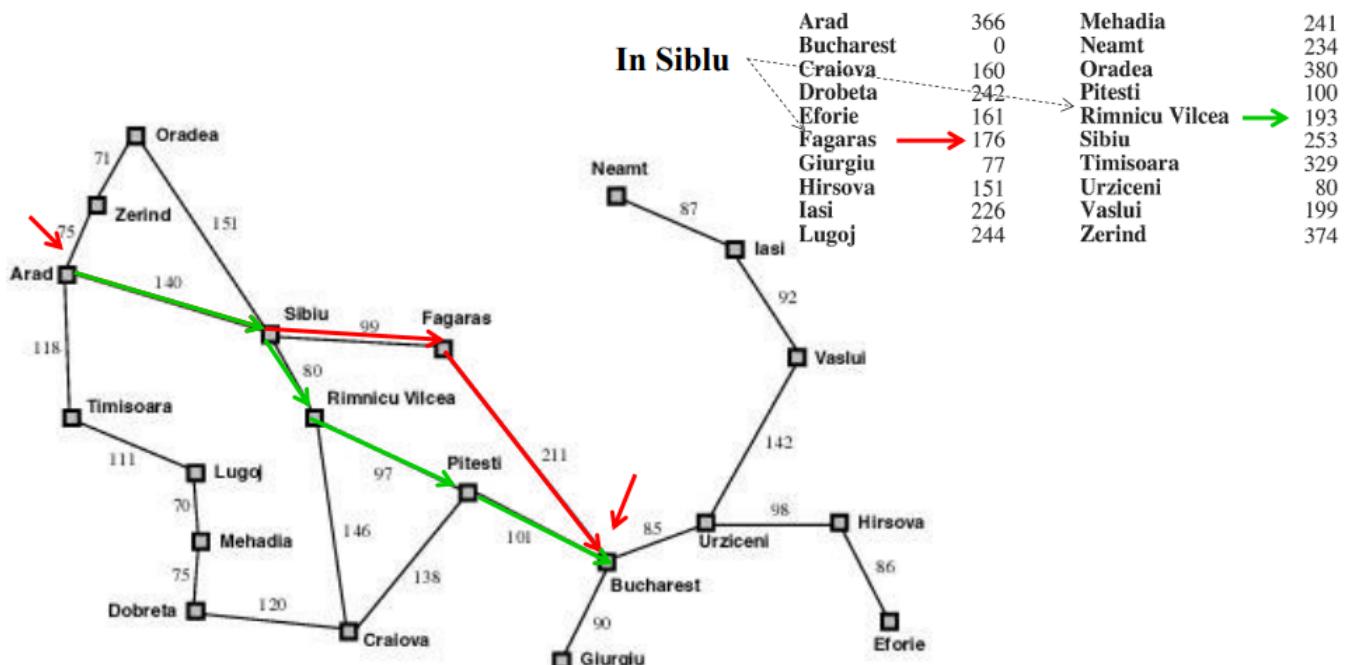
Esempio di euristica h Per procedere preferibilmente verso il percorso migliore, seguendo il "problem-specific information", nel problema del percorso più breve da città a città posso includere nel mio algoritmo le distanze in linea d'aria, oppure il vantaggio in pezzi nella dama o negli scacchi.

3.2 Best-First

Algoritmo di ricerca Best-First Heuristic utilizza lo **stesso algoritmo di Uniform-Cost** ma utilizzando f (stima di costo) per la coda con priorità. La **scelta di f determina la strategia di ricerca**: ad ogni passo si sceglie il nodo sulla frontiera per con valore di f migliore (**nodo più promettente**).

Nota Migliore significa "minore" in caso di un'euristica che stima la distanza della soluzione

Caso Speciale Greedy Best-First, si usa solo h ($f = h$)



Da Arad a Bucarest ...

Greedy best-first: Arad, Sibiu, Fagaras, Bucharest (450)

ma non è l'Ottimo: Arad, Sibiu, Rimnicu, Pitesti, Bucarest (418)

3.2.1 Algoritmo A

Si può dire qualcosa di f per avere garanzie di completezza e ottimalità?

Definizione Un algoritmo A è un algoritmo Best-First con una funzione di valutazione dello stato del tipo

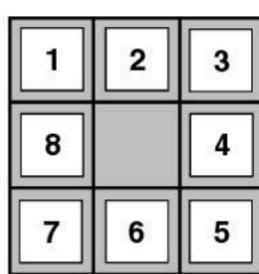
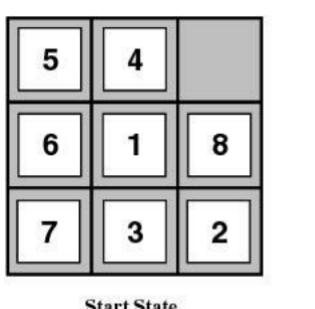
$$f(n) = g(n) + h(n)$$

con $h(n) \geq 0$ e $h(goal) = 0$. $g(n)$ è il costo del cammino percorso per raggiungere n e $h(n)$ è una stima del costo per raggiungere da n un nodo $goal$. Vedremo casi particolari dell'algoritmo A:

se $h(n) = 0$, cioè $f(n) = g(n)$, si ha **Ricerca Uniforme** (UC)

se $g(n) = 0$, cioè $f(n) = h(n)$, si ha **Greedy Best First**

Esempio Il gioco dell'otto



$$f(n) = \#\text{mosseFatte} + \#\text{caselleFuoriPosto}$$

$$f(start) = 0 + 7$$

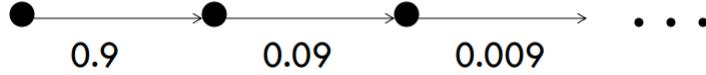
Dopo $\leftarrow, \downarrow, \uparrow, \rightarrow$ si ha $f = 4 + 7$: stesso stato ma g è cambiata.

$$f(goal) = ? + 0$$

Algoritmo A è completo

Teorema L'algoritmo A con la condizione $g(n) \geq d(n) \cdot \epsilon$, con $d(n)$ profondità e $\epsilon > 0$ costo minimo dell'arco, è **completo**.

La condizione ci garantisce che non si verifichino condizioni del tipo



e che il costo lungo un cammino non cresca "abbastanza", così da fermarsi per costi alti di g .

Dimostrazione Sia $[n_0 n_1 n_2 \dots n' \dots n_k = goal]$ un cammino soluzione. Sia n' un nodo della frontiera su un cammino soluzione $\rightarrow n'$ prima o poi verrà espanso. Infatti, esistono solo un numero finito di nodi x che possono essere aggiunti alla frontiera con $f(x) \leq f(n')$ (condizione su g).

Quindi, se non si trova una soluzione prima, n' verrà espanso e i suoi successori aggiunti alla frontiera. Tra questi, **anche il suo successore sul cammino soluzione**.

Il ragionamento si può ripetere fino a dimostrare che anche il nostro *goal* sarà selezionato per l'espansione.

3.2.2 Algoritmo A*: La Stima Ideale

Una **funzione di valutazione ideale (oracolo)** $f^*(n) = g^*(n) + h^*(n)$

$g^*(n)$ costo del **cammino minimo** da radice a n

$h^*(n)$ costo del **cammino minimo** da n a *goal*

$f^*(n)$ costo del **cammino minimo** da radice a *goal*, attraverso n

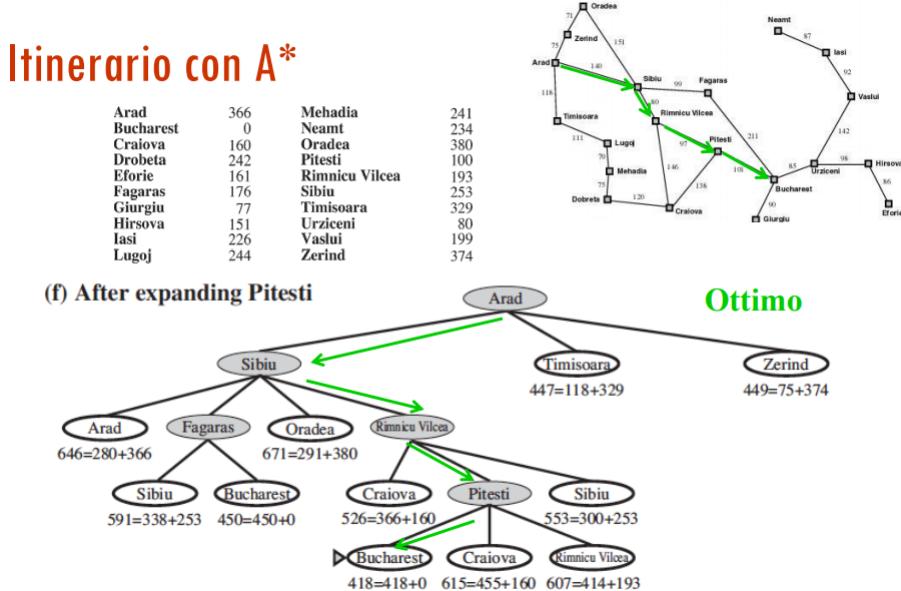
Normalmente $g(n) \geq g^*(n)$ (costo del cammino \geq cammino minimo) e $h(n)$ è una **stima** di $h^*(n)$: si può sotto o sovrastimare la distanza dalla soluzione.

Definizione Euristica ammissibile $\forall n \mid h(n) \leq h^*(n)$, h è una **sottostima**, ad esempio l'euristica della distanza in linea d'aria.

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

Teorema Gli algoritmi A* sono **ottimali**.

Corollario BF con passi a costo costante e UC sono **ottimali** ($h(n) = 0$)



Osservazioni La componente g fa sì che si abbandonino cammini che vanno troppo in profondità.

h sotto o sovra stima? Una sottostima può farci compiere lavoro inutile, ma **non fa perdere il cammino migliore**: quando trovo il nodo *goal* è il cammino migliore. Invece, una funzione che a volte sovrastima può **farci perdere la soluzione ottimale a causa di tagli per sovrastima**.

Ottimalità Nel caso di ricerca su albero, l'**uso di un'euristica ammissibile è sufficiente a garantire l'ammissibilità \Rightarrow ottimalità di A***.

Nel caso di ricerca su grafo, serve una proprietà più forte: la **consistenza**, anche detta **monotonicità**, perché rischio di scartare candidati ottimi (stato già incontrato) a meno che il primo espanso sia il migliore.

Definizione Euristica consistente se

$$h(goal) = 0$$

Consistenza locale: $\forall n \mid h(b) \leq c(n, a, n') + h(n')$ dove n' è un successore di n e $c(n, a, n')$ è il costo del cammino $n \rightarrow n'$ sull'arco a .

$$\Rightarrow f(b) \leq f(n')$$

Quindi se h è consistente, allora **f non decresce mai lungo i cammini**: da qui il termine monotonia. Esistono euristiche ammissibili che non sono monotone, ma sono rare.

Teorema Un'euristica monotona è ammissibile.

Le euristiche monotone garantiscono che la **soluzione meno costosa venga trovata per prima** e quindi **sono ottimali anche nel caso di ricerca su grafo**.

Non si devono recuperare, tra gli antenati, nodi con costo minore

Lista degli esplorati, stato già esplorato è sul cammino ottimo \Rightarrow posso evitare di inserire il corrente ripetuto senza perdere l'ottimalità

```
if (figlio.Stato non in Esplorati and non in Frontiera)
    Frontiera = Inserisci(figlio, Frontiera)
```

Per la frontiera, volendo evitare stati ripetuti, resta l'**if finale di UC**

```
if (figlio.Stato in Frontiera con costoCammino più alto)
    sostituisci quel nodo frontiera con il figlio
```

Ottimalità di A* Dimostrazione

1. $h(n)$ consistente \Rightarrow i valori di $f(n)$ lungo un cammino sono non decrescenti:

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

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

ma siccome $g(n) + c(n, a, n') = g(n')$, allora $g(n) + h(n) \leq g(n') + h(n')$

quindi $f(n) \leq f(n')$

2. Ogni volta che A* seleziona un nodo n per l'espansione, il cammino ottimo a tale nodo è stato trovato.

Se così non fosse, ci sarebbe un altro nodo n' della frontiera sul cammino ottimo (a n , ancora da trovare), con $f(n')$ minore (per la monotonia e n successivo di n').

Ma ciò non è possibile perché tale nodo sarebbe già stato espanso.

3. Quando seleziona nodo *goal* è cammino ottimo [$h = 0, f = C^*$]

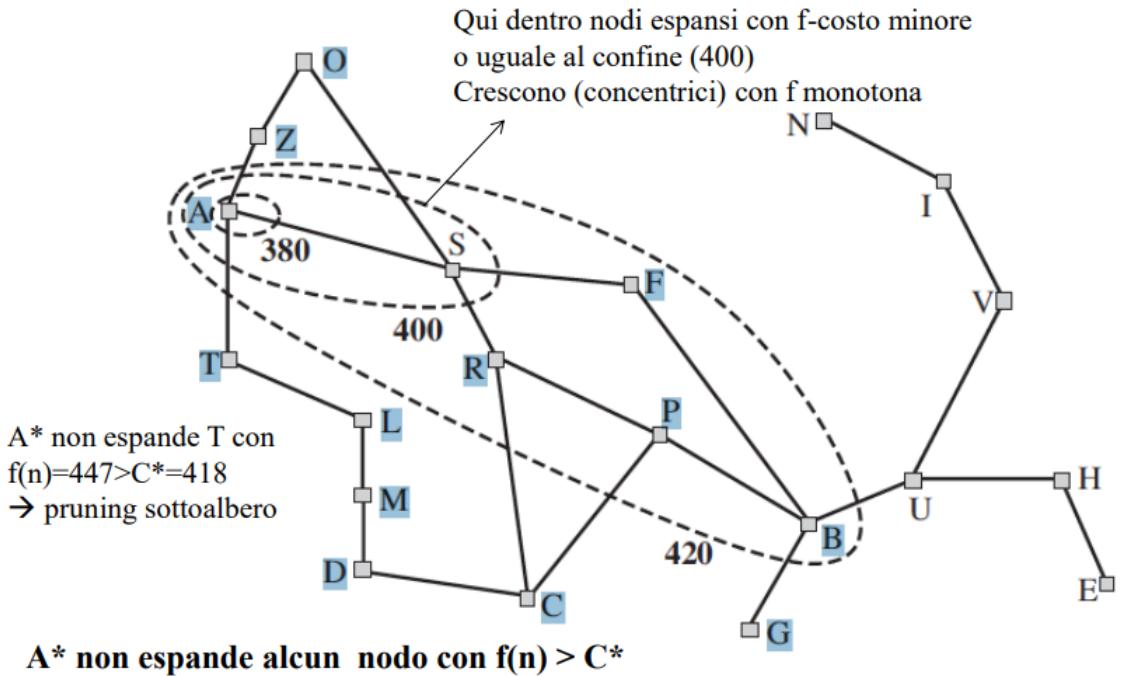
3.2.3 Perché A* è vantaggioso?

A* espande tutti i nodi con $f(n) < C^*$

A* espande *alcuni* nodi con $f(n) = C^*$

A* non espande alcun nodo con $f(b) > C^*$

Quindi alcuni nodi (e suoi sottoalberi) non verranno considerati per l'espansione, ma restiamo ottimali: **pruning**, un' h opportuna, **più alta possibile fra le ammissibili**, fa tagliare molto.



Più f è aderente, più taglio ottenendo ovali più stretti. Cercheremo quindi un' h più alta possibile tra le ammissibili. Se molto bassa, molti (sino a tutti) nodi restano $< C^* \Rightarrow$ espando tutti (a cerchi).

Pruning Il pruning dei sotto-alberi è il punto focale: non li abbiamo già in memoria ed evitiamo di generarli, e ciò è decisivo per i problemi di AI a spazio stati esponenziali.

3.2.4 Conclusioni su A*

Algoritmo Lo stesso usato per UC

Funzioni Usa $f = g + h$ per la coda di priorità, dove h e g soddisfano le condizioni per algoritmo A e h è una funzione euristica ammissibile per A*.

Sui grafi necessita di un'euristica monotona.

Completo Discende dalla completezza di A, perché A* è un A particolare

Ottimale Con euristica monotona

Ottimamente efficiente A parità di euristica nessun'altro algoritmo espande meno nodi senza rinunciare ad ottimalità

Problemi Quale euristica?

Occupazione in memoria: $O(b^{d+1})$

3.2.5 Casi speciali di A

$h(n) = 0$ si ha Uniform Cost, cioè $f(n) = g(n)$
Cioè g non basta

$g(n) = 0$ si ha Greedy Best First, cioè $f(n) = h(n)$
Ossia h non basta

3.3 Costruire le Euristiche di A*

3.3.1 Valutazione di funzioni euristiche

A parità di ammissibilità, una euristica può essere più efficiente di un'altra nel trovare il cammino soluzione migliore. Questo dipende da quanto **informata** è l'euristica, cioè dal grado di informazione posseduto.

$h(n) = 0$ **minimo** di informazione (BF, o UC)

$h^*(n)$ **massimo** di informazione (oracolo)

In generale, per le euristiche ammissibili,

$$0 \leq h(n) \leq h^*(n)$$

Teorema Se $h_1 \leq h_2$, i nodi espansi da A^* con h_2 sono un **sottoinsieme** di quelli espansi da A^* con h_1 .
Questo perché A^* espande tutti i nodi con $f(n) < C^*$ e sono meno per h maggiore (fa andare più nodi oltre C^*).
 \Rightarrow Se $h_1 \leq h_2$, allora A^* con h_2 è **almeno efficiente** quanto A^* con h_1 .

Un'euristica più informata (accurata) riduce lo spazio di ricerca (più efficiente) ma è tipicamente **più costosa da calcolare**.

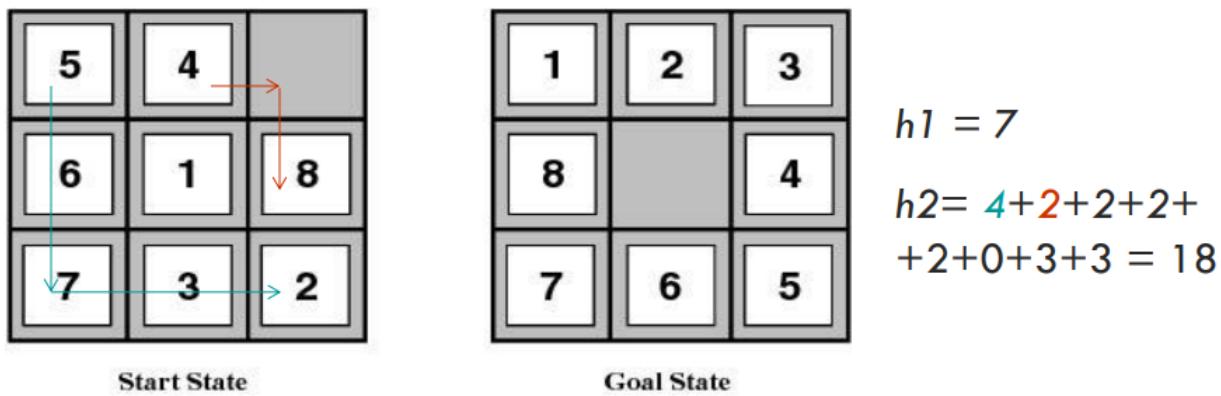
3.3.2 Confronto di euristiche ammissibili

Esempio Due euristiche ammissibili per il gioco dell'otto

h_1 conta il numero di caselle fuori posto

h_2 somma delle distanze Manhattan (orizz/vert) delle caselle fuori posto dalla posizione finale
Manhattan Distance: $h(x, y) = MD((x, y), (x_g, y_g)) = |x - x_g| + |y - y_g|$

$\Rightarrow h_2$ è **più informata** di h_1 , infatti $\forall n \Rightarrow h_1(n) \leq h_2(n)$. Si dice che h_2 **domina** h_1 .



3.3.3 Misura del potere euristico

Come valutare gli algoritmi di ricerca euristica

Fattore di diramazione effettivo Dato N numero di nodi generati, d profondità della soluzione, allora b^* è il fattore di diramazione di un albero uniforme con $N+1$ nodi, soluzione dell'equazione

$$N + 1 = b^* + (b^*)^2 + \dots + (b^*)^d$$

Sperimentalmente, una buona euristica ha un b^* abbastanza vicino ad 1 (< 1.5)

Esempio $d = 5$, $N = 52 \Rightarrow b^* = 1.92$

3.3.4 Capacità di esplorazione

Influenza di b^*

Con $b = 2$

$$d = 6 \quad N = 100$$

$$d = 12 \quad N = 10000$$

Con $b = 1.5$

$$d = 12 \quad N = 100$$

$$d = 24 \quad N = 10000$$

Quindi **migliorando di poco l'euristica si riesce, a parità di nodi espansi, a raggiungere una profondità doppia.**

Quindi

Tutti i problemi dell'IA, o quasi, sono di complessità esponenziale nel generare nodi (ad es. configurazioni possibili), ma c'è esponenziale ed esponenziale. 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.**

3.4 Come si inventa un'euristica?

Alcune strategie per ottenere euristiche ammissibili, da vedere man mano:

Rilassamento del problema

Massimizzazione di euristiche

Database di pattern disgiunti

Combinazione lineare

Apprendere dall'esperienza

3.4.1 Rilassamento del problema

Spazio degli stati con archi aggiunti.

Gioco dell'otto Nel gioco dell'otto, mossa da A a B possibile se **B adiacente ad A e B libera**.

h_1 e h_2 sono **calcoli della distanza esatta della soluzione** in versioni semplificate del puzzle: uno **spazio degli stati con archi aggiunti**

h_1 nessuna restrizione: sono sempre ammessi scambi tra caselle, si muove ovunque → numero di caselle fuori posto

h_2 solo prima restrizione: sono ammessi spostamenti anche su caselle occupate purché adiacenti → somma delle distanze Manhattan

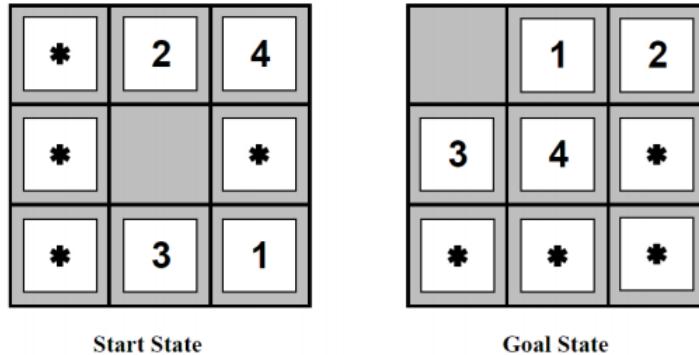
3.4.2 Massimizzazione di euristiche

Si hanno una serie di euristiche ammissibili h_1, h_2, \dots, h_k , **senza che nessuna domini un'altra**. Allora conviene prendere il **massimo dei loro valori**

$$h(n) = \max\{h_1(n), h_2(n), \dots, h_k(n)\}$$

Se le h_i sono ammissibili, anche la h lo è e **domina tutte le altre**.

Euristiche da sottoproblemi



Il **costo della soluzione ottima del sottoproblema** (sistemare 1, 2, 3, 4) è una **sottostima del costo del problema nel suo complesso** e più accurata della Manhattan.

Database di pattern: si memorizza ogni istanza del sottoproblema con relativo costo della soluzione. Si usa il database per calcolare h_{DB} , estraendo dal DB la configurazione corrispondente allo stato completo corrente.

Sottoproblemi multipli Potremmo poi fare la stessa cosa per altri sottoproblemi: 5-6-7-8, 2-4-6-8... ottenendo altre euristiche ammissibili.

Poi si può prendere il valore massimo: altra euristica ammissibile.

Ma potremmo sommarle ed ottenere un'euristica ancora più accurata?

3.4.3 Pattern Disgiunti

In generale no, perché le **soluzioni ai sottoproblemi interferiscono**: nel caso del gioco dell'otto, condividono alcune mosse perché se sposto 1-2-3-4 sposto anche 4-5-6-7.

La **somma delle euristiche in generale non è ammissibile** perché potremmo sovrastimare, avendo avuto aiuti mutui.

Si deve **eliminare il costo delle mosse che contribuiscono all'altro sottoproblema**: **databse di pattern disgiunti** consentono di sommare i costi (**euristiche additive**).

Sono molto efficaci: gioco del 15 in pochi ms. Ma difficile scomporre per il cubo di Rubik.

3.4.4 Apprendere dall'esperienza

Si fa girare il programma e si raccolgono i dati sottoforma di **coppie** `<stato, h*>`. Si usano i dati per apprendere come predire la h con **algoritmi di apprendimento induttivo**: da istanze note stimiamo h in generale.

Gli algoritmi di apprendimento si concentrano su caratteristiche salienti dello stato (*feature, x_i*). Esempio: numero tasselli fuori posto 5 → costo 14.

Combinazione di euristiche

Quando diverse caratteristiche influenzano la bontà di uno stato, si può usare una combinazione lineare

$$h(n) = c_1 x_1(n) + c_2 x_2(n) + \dots + c_k x_k(n)$$

Gioco dell'otto $h(n) = c_1 \#fuoriPosto + c_2 \#coppieScambiate$

Scacchi $h(n) = c_1 vantaggioPezzi + c_2 pezziAttaccante + c_3 regina + \dots$

Il **peso dei coefficienti può essere aggiustato con l'esperienza**, anche qui **apprendendo automaticamente da esempi di gioco**. $h(goal) = 0$ ma ammissibilità e consistenza **non** automatiche.

Capitolo 4

Algoritmi Evolutivi Basati su A*

4.1 Beam Search

Nel **best first** viene mantenuta tutta la frontiera. Se l'occupazione di memoria è eccessiva, si può ricorrere ad una variante: la **beam search**.

Beam Search La beam search **tiene ad ogni passo solo i k nodi più promettenti**, dove k è detto **ampiezza del raggio** (beam).

Non è completa.

4.2 IDA*

A* con approfondimento iterativo. IDA* combina A* con ID: ad ogni iterazione si ricerca in profondità con un limite (cut-off) dato dal valore della funzione f (e non dalla profondità).

Il limite **f-limit** viene aumentato ad ogni iterazione, fino a trovare la soluzione.

Punto Critico Di quanto viene aumentato f-limit.

Quale incremento? Cruciale la scelta dell'incremento per garantire l'ottimalità. In caso di azioni dal costo fisso, il limite viene incrementato dal costo delle azioni.

Ma in caso di costi variabili? Costo minimo? Si potrebbe, ad ogni passo, fissare il limite successivo al valore minimo delle f scartate (in quanto superavano il limite) all'interazione precedente.

Analisi **Completo e ottimale.** Occupazione in memoria $O(bd)$.

4.3 RBFS

Best-First Ricorsivo: simile a DF ricorsivo ma cerca di usare meno memoria, facendo del lavoro in più.

Tiene traccia del migliore percorso alternativo ad ogni livello. Invece di fare backtracking in caso di fallimento, interrompe l'esplorazione quando trova un nodo meno promettente secondo f . Nel tornare indietro **si ricorda il miglior nodo che ha trovato nel sottoalbero esplorato**, per poterci eventualmente tornare.

Memoria: lineare nella profondità della soluzione ottima.

4.4 A* con memoria limitata

L'idea è quella di utilizzare al meglio la memoria disponibile.

SMA* procede come A* fino ad esaurimento della memoria disponibile. A questo punto "dimentica" il nodo peggiore, dopo avere aggiornato il valore del padre.

A parità di f **si sceglie il nodo migliore più recente e si dimentica il nodo peggiore più vecchio.**

Ottimale se il cammino soluzione sta in memoria.

In algoritmi a memoria limitata, le limitazioni della memoria possono portare a compiere molto lavoro inutile: ad esempio, visitare ripetutamente gli stessi nodi. Diventa quindi **difficile stimare la complessità temporale effettiva**. Le **limitazioni della memoria**, quind, **possono rendere un problema intrattabile** dal punto di vista computazionale.

4.5 Conclusioni

Agenti in ambienti deterministici, osservabili, statici e completamente noti

Ricerca come **scelta della sequenza di azioni**, cioè cammino in uno spazio di stati, **che raggiunge obiettivo**
⇒ il cammino è la soluzione

Attività necessarie

Formulazione del problema

Scelta dell'algoritmo di ricerca adeguato

Identificazione della funzione di valutazione euristica più efficace

Capitolo 5

Oltre la Ricerca Classica

Risolutori "classici" Gli agenti risolutori di problemi *classici* assumono le condizioni viste in precedenza:

Ambienti completamente osservabili

Ambienti deterministici

Si trovano nelle condizioni di produrre un piano d'azione eseguibile "ad occhi chiusi": possono studiare la sequenza d'azioni *offline*, che può essere eseguita senza imprevisti per raggiungere l'obiettivo

5.1 Verso ambienti più realistici

La **ricerca sistematica**, o euristica, nello spazio di stati è **troppo costosa** → Metodi di ricerca locale
Assunzioni sull'ambiente da considerare:

Azioni non deterministiche

Ambiente parzialmente osservabile

→ Piani condizionali, ricerca AND-OR, stati credenza

Ambienti sconosciuti

Problemi di esplorazione (percezioni forniscono nuove informazioni dopo l'azione)

→ Ricerca **online**

5.2 Ricerca Locale

Assunzioni Gli algoritmi visti fin'ora esplorano gli spazi degli stati alla ricerca di un goal e restituiscono un **cammino soluzione**, ma a volte lo stato goal è la soluzione del problema. Gli **algoritmi di ricerca locale** sono adatti per problemi in cui

La sequenza di azioni non è importante: quello che conta è lo stato goal

Tutti gli elementi della soluzione sono nello stato ma alcuni vincoli sono violati

5.2.1 Algoritmi di ricerca locale

Non sono sistematici

Tengono traccia solo del nodo corrente e si spostano su nodi adiacenti

Non tengono traccia dei cammini, poiché non servono a lavoro finito

Efficienti in memoria

Possono trovare soluzioni ragionevoli anche in spazi molto grandi o infiniti, come nel caso di spazi continui

Utili per **risolvere problemi di ottimizzazione**

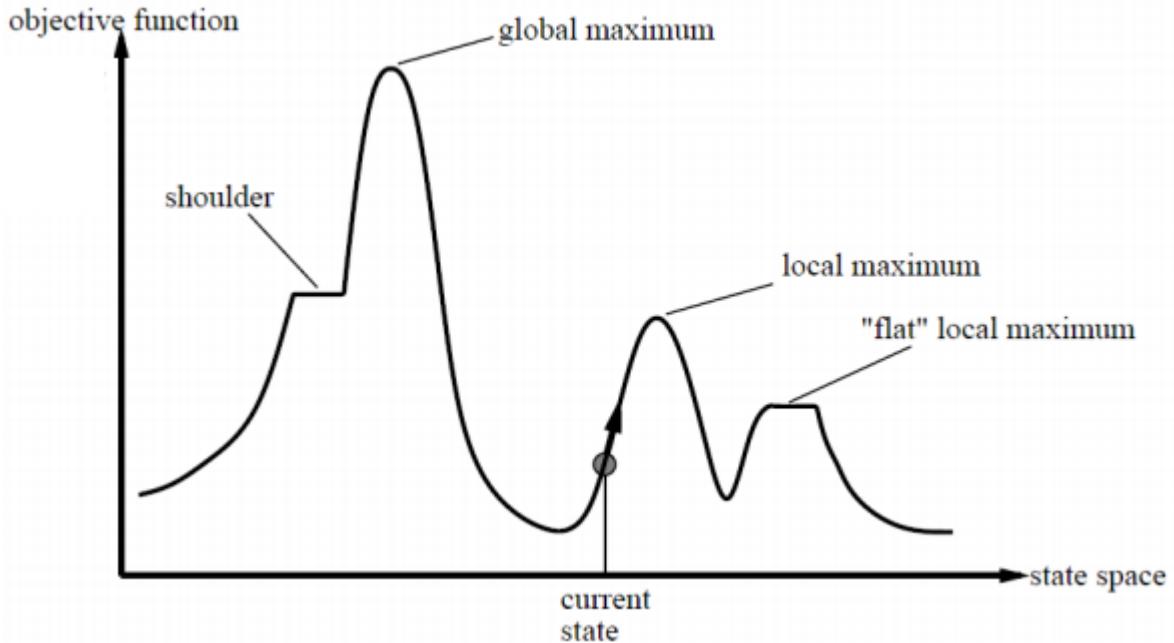
Stato migliore secondo una funzione obiettivo

Stato di costo minore

Esempio: training di un modello di Machine Learning

5.2.2 Panorama dello spazio degli stati

Esempio di una f euristica di costo della funzione obiettivo (non del cammino)



Uno stato ha una posizione sulla superficie e un'altezza corrispondente al valore della sua funzione di valutazione. Un algoritmo provoca un movimento sulla superficie.

Trovare avvallamento più basso (es: costo minimo) o il picco più alto (es: massimo di un obiettivo)

5.2.3 Algoritmo Hill Climbing

Anche detto **ricerca in salita, steepest ascent/descent**.

Ricerca locale **greedy**.

Vengono generati i successori e valutati. Viene scelto un nodo che migliora la valutazione dello stato attuale (**non si tiene traccia degli altri**, quindi non ho l'albero di ricerca in memoria). La scelta del nodo dipende dall'algoritmo:

Migliore → **Hill Climbing a salita ripida**

Uno a caso tra quelli che migliorano → **Hill Climbing Stocastico**

Il primo → **Hill Climbing con prima scelta**

Se non ci sono stati migliori, l'algoritmo termina con **fallimento**.

```
function Hill-climbing(problema) returns stato-massimo-locale
    nodo-corrente = CreaNodo(problema.Stato-iniziale)
    loop do
        vicino = successore di nodo-corrente di valore piu alto
        if (vicino.Valore <= nodo-corrente.Valore)
            return nodo-corrente.Stato #interrompe la ricerca
        nodo-corrente = vicino #altrimenti, se vicino migliore, continua
```

Si prosegue solo se il vicino più alto è migliore dello stato corrente, se tutti i vicini sono peggiori, si ferma. Non c'è frontiera a cui tornare, si tiene solo uno stato.

Tempo: numero di cicli variabile in base al punto di partenza.

Problema delle 8 Regine Con formulazione a stato completo

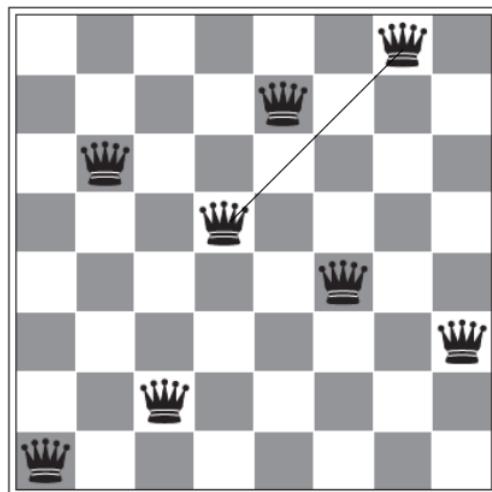
Costo h = numero di coppie di regine che si attaccano a vicenda

I numeri sono i valori dei successori (7×8 , 7 posizioni per ogni regina = su ogni colonna)

Tra i migliori (valore 12) si sceglie a caso

Minimo globale = 0

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



Un minimo locale

$$h = 1$$

Tutti gli stati successori non migliorano la situazione (minimo locale)

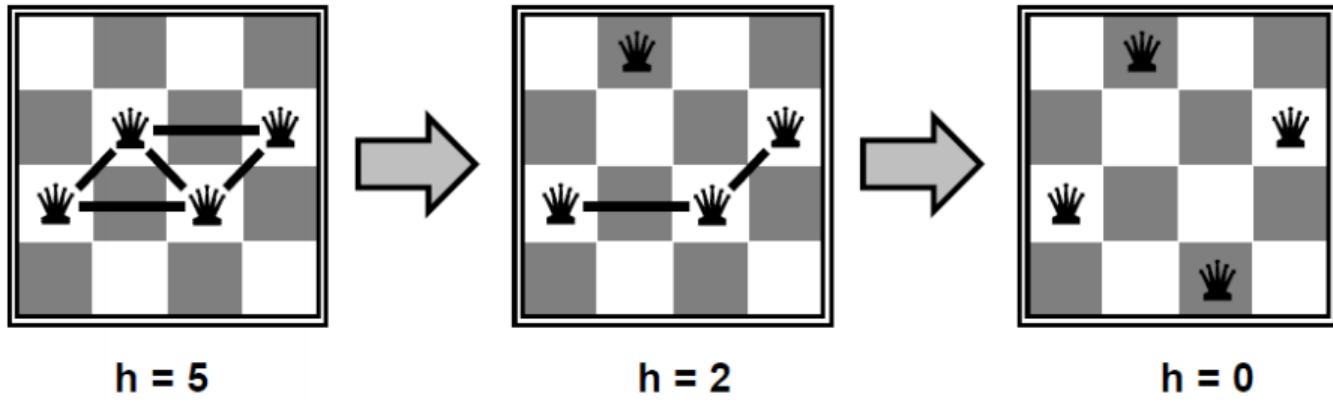
Per le 8 regine Hill-Climbing si blocca l'86% delle volte

Ma in media sono 4 passi per la soluzione, e 3 quando si blocca

Su $8^8 = 17$ milioni di stati

Esempio Successo in tre mosse

h qui è l'**euristica di costo della funzione obiettivo** da minimizzare



Problemi con Hill-Climbing

Se la funzione è da ottimizzare, i picchi sono massimi locali o soluzioni ottimali. Nel grafico si possono presentare: **massimi locali, plateau (pianori), spalle e crinali/creste.**

Miglioramenti

1. Consentire un **numero limitato di mosse laterali**
Ossia ci si ferma per $<$ nell'algoritmo, anziché per \leq
 \Rightarrow L'algoritmo sulle 8 regine ha successo nel 94% dei casi, ma impiega in media 21 passi
2. Hill-Climbing Stocastico: si **sceglie a caso tra le mosse in salita**, magari tenendo conto della pendenza.
Converge più lentamente ma a volte trova soluzioni migliori
3. Hill-Climbing con prima scelta: può generare le mosse a caso fino a trovarne una migliore dello stato corrente.
Più efficace quando i successori sono molti (es: migliaia)
4. Hill-Climbing con riavvio casuale (**random restart**): ripartire da un punto scelto a caso.
Se la probabilità di successo è p , saranno necessarie in media $\frac{1}{p}$ ripartenze per trovare la soluzione (Es.: 8 regine, $p = 0.14$, 7 iterazioni cioè 6 fallimenti e un successo).
Tendenzialmente completo: insistendo si generano tutte le possibilità.
Per le regine, 3 milioni di regine in meno di un minuto. Se funziona o no dipende molto dalla forma del panorama degli stati.

5.2.4 Algoritmo di Tempra Simulata

Simulated Annealing L'algoritmo di tempra simulata combina Hill-Climbing con una scelta stocastica ma **non del tutto casuale perché poco efficiente**. L'analogia è col processo di tempra dei metalli: portati a temperature molto elevate e raffreddati gradualmente consentendo di cristallizzare in uno stato a più bassa energia.

Tempra Simulata Ad ogni passo si **sceglie un successore a caso**:

Se migliora lo stato corrente, viene espanso

Se non lo migliora (caso in cui $\Delta E = f(n') - f(n) < 0$), quel nodo viene scelto con probabilità $p = e^{\Delta E/T}$, con p ovviamente $0 \leq p \leq 1$

(Si genera un numero casuale tra 0 e 1, se questo è $< p$ il successore viene scelto, altrimenti no)

p è **inversamente proporzionale al peggioramento**

T (**temperatura**) **decresce al progredire dell'algoritmo**, quindi anche p , secondo un piano definito.
Col progredire, rende improbabili le mosse peggiorative.

Analisi La probabilità di una mossa in discesa diminuisce col tempo, e l'algoritmo si comporta sempre di più come Hill-Climbing.

Se T viene decrementato abbastanza lentamente, con probabilità tendente ad 1 si raggiunge la soluzione ottimale.

Analogia: T corrisponde alla temperatura e ΔE alla variazione di energia

Parametri **Valore iniziale e decremento di T** sono parametri.

I valori per T sono **determinati sperimentalmente**: il **valore iniziale di T** è tale che per valori medi di ΔE , $p = e^{\Delta E/T}$ sia circa 0.5

5.2.5 Algoritmo Local Beam

Versione locale della beam search.

Si tengono in memoria k stati invece che uno solo. Ad ogni passo si generano i successori di tutti i k stati.

Se si trova un goal, ci si ferma

Altrimenti si prosegue con i k migliori tra questi

Note:

Diverso da K restart (che riparte da 0)

Diverso da beam search

5.2.6 Algoritmo Beam Search Stocastico

Si introduce un elemento di casualità, come in un processo di selezione naturale: **diversificare la nuova generazione**. In questa variante stocastica, si scelgono k successori ma **con probabilità maggiore per i migliori**. Terminologia:

Organismo, stato

Progenie, successori

Fitness (valore della f), idoneità

5.3 Algoritmi Genetici

L’Idea Sono varianti della beam search stocastica in cui gli stati successori sono ottenuti *combinando* due stati genitore anziché per evoluzione. Terminologia:

Popolazione di individui (stati)

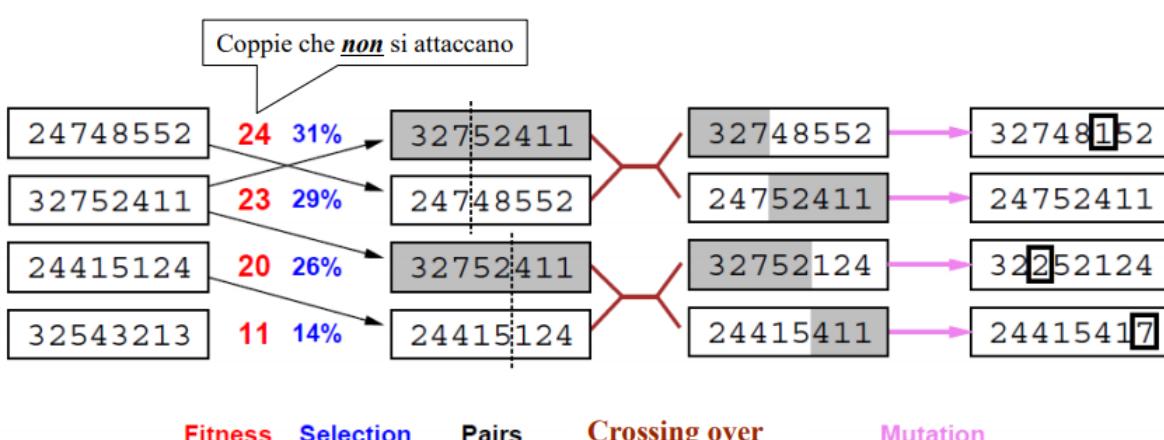
Fitness

Accoppiamenti e mutazione genetica

Generazioni

Funzionamento Popolazione iniziale: k stati/individui generati casualmente. Ogni **individuo è rappresentato come una stringa**: ad esempio 24 bit, o posizione nelle colonne ("24748552") per le 8 regine. Gli individui sono valutati da una funzione di **fitness**: ad esempio numero di coppie di regine che *non* si toccano. Si selezionano gli individui per gli accoppiamenti, con una probabilità proporzionale alla fitness. Le **coppie danno vita alla generazione successiva**: combinando materiale genetico (**crossover**) o con un meccanismo aggiuntivo di mutazione genetica (**casuale**). La popolazione ottenuta dovrebbe essere migliore e la cosa si ripete fino ad ottenere stati abbastanza buoni (**stati obiettivo**).

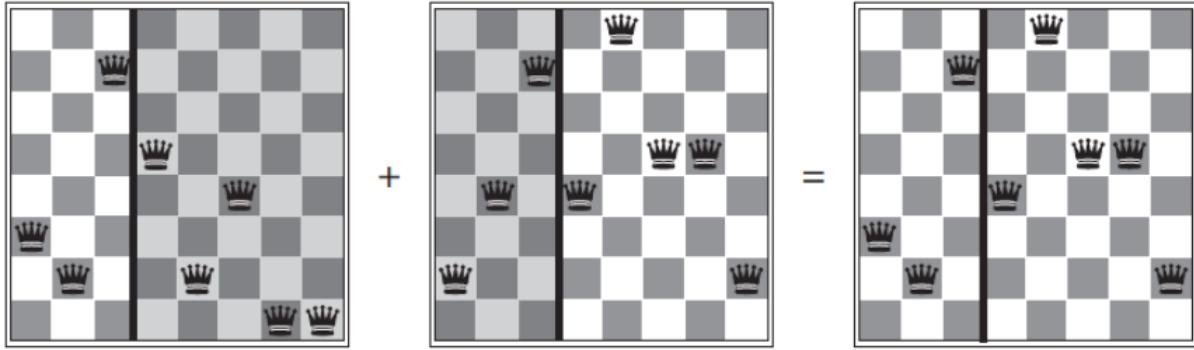
Esempio



Per ogni coppia viene scelto un punto di **crossing over** (la linea tratteggiata) e vengono generati due figli **scambiandosi dei pezzi del DNA**.

Viene infine effettuata una mutazione casuale che dà luogo alla prossima generazione.

Nascita di un figlio



Le parti chiare sono passate al figlio, le parti scure si perdono. Se i genitori sono molto diversi, anche i nuovi stati sono diversi. All'inizio spostamenti maggiori che poi si raffinano.

Algoritmi Genetici

Suggeritivi Area del **Natural Computing**. Usati molto in problemi reali (es.: configurazione di circuiti e scheduling dei lavori).

Combinano la tendenza a salire della beam search stocastica con l'interscambio di informazioni tra thread paralleli di ricerca (blocchi utili che si combinano).

Funziona meglio se il problema (soluzioni) ha componenti significative rappresentate in sottostringhe.

Punto critico: rappresentazione del problema in stringhe.

Spazi Continui Molti casi reali hanno spazi di ricerca continua, fondamentale per il Machine Learning! Lo stato è descritto da variabili continue x_1, \dots, x_n (vettore x), ad esempio la posizione in uno spazio 3D $x = (x_1, x_2, x_3)$.

Apparentemente è complicato: fattori di ramificazione infiniti con gli approcci precedenti. Ma in realtà ci sono molti strumenti matematici per spazi continui, che portano ad approcci anche molto efficienti.

5.3.1 Gradient

Se la f è **continua** e **differenziabile**, ad esempio quadratica rispetto il vettore x , allora il minimo/massimo lo si può cercare utilizzando il **gradiente**, che **restituisce la direzione di massima pendenza del punto**.

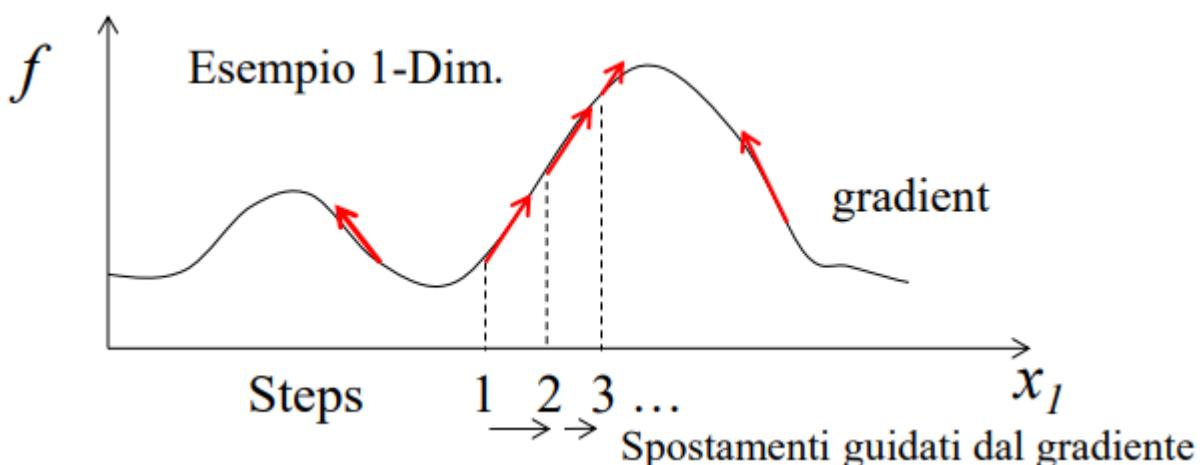
Data f obiettivo

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

Hill-Climbing iterativo: $x_{new} = x_{old} + \eta \nabla f(x)$ con η dimensione dello step.

Quantifica lo spostamento, senza cercarlo tra gli infiniti possibili successori.

Nota: in generale non è sempre necessario il min/max assoluto. Vedremo nel ML.



5.4 Ambenti più realistici

Problemi classici Gli agenti risolutori di problemi "classici" assumono:

Ambienti **completamente osservabili**

Azioni e ambienti **deterministici**

Piano generato è sequenza di azioni **eseguibile ad occhi chiusi**, generato *offline* ed eseguito senza imprevisti

Le **percezioni non servono**, se non nello stato iniziale

Soluzioni più complesse In un ambiente **parzialmente osservabile e non deterministico** le **percezioni sono importanti: restringono gli stati possibili e informano sull'effetto dell'azione**.

Più che un piano, l'**agente elabora una strategia** che **tiene conto delle diverse eventualità: un piano con contingenza**. Esempio: aspirapolvere con assunzioni diverse.

5.4.1 Azioni non deterministiche

Aspirapolvere imprevedibile Ci sono più stati possibili come risultato dell'azione.

Comportamento: se aspira in una stanza sporca la pulisce... ma **a volte** pulisce anche una stanza adiacente. Se aspira in una stanza pulita, **a volte** rilascia sporco.

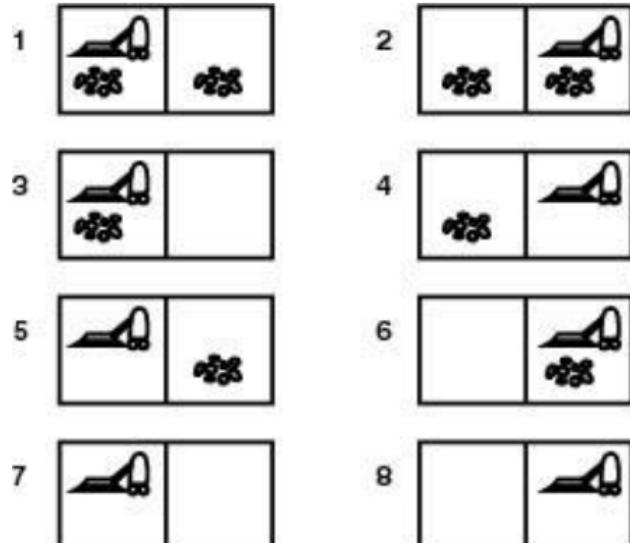
Variazioni necessarie al modello Il modello di transizione, quindi, **restituisce un insieme di stati**: l'agente non sa in quale si troverà. Il **piano di contingenza** sarà un piano condizionale e magari con cicli.

Esempio Nell'esempio

Risultati(Aspira, 1) = {5, 7}, cioè aspirando nello stato 1 posso finire nello stato 5 o 7.

Un possibile piano è

```
[ Aspira ,
  if (stato = 5):
    [ Destra , Aspira ]
  else:
    []
]
```



Da sequenza di azioni a piano (albero)

5.4.2 Come si pianifica

Alberi di Ricerca AND-OR

Nodi **OR**: scelte dell'agente (1 sola azione)

Nodi **AND**: le diverse contingenze (scelte dell'ambiente, più stati possibili), **da considerare tutte**

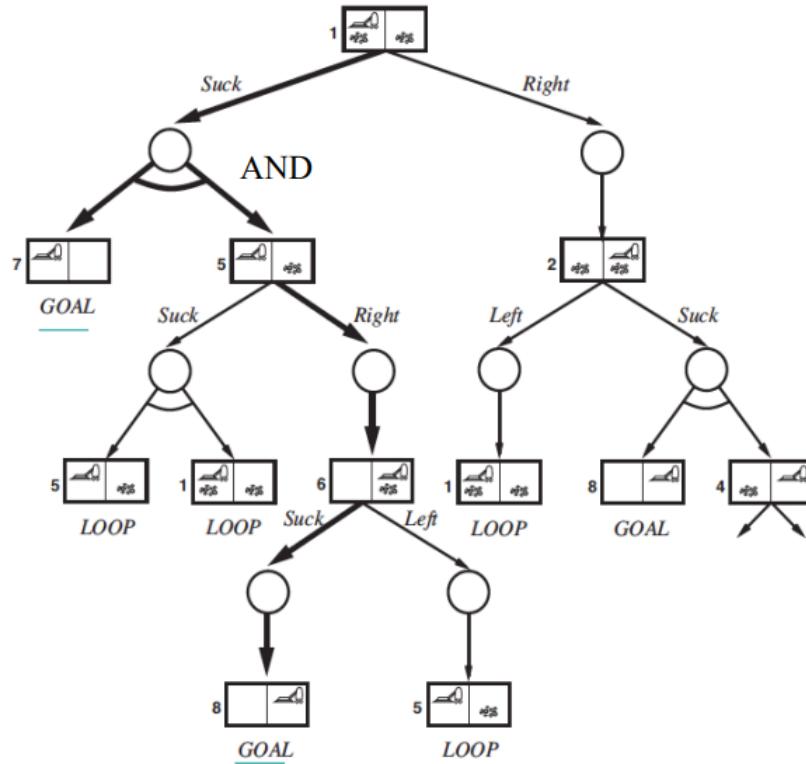
Una **soluzione ad un problema di ricerca AND-OR** è un **albero** che

Ha un nodo obiettivo in ogni foglia

Specifica un'unica azione nei nodi **OR**

Include tutti gli archi uscenti da nodi **AND**

Esempio



Archi in grassetto = soluzione (sottoalbero), la seguente:

Piano: [Aspira: **if** (stato = 5): [Destra , Aspira] **else**: []]

Capitolo 6

I Giochi con Avversario

Premessa Fin'ora abbiamo avuto *Problemi Solving* come ricerca. Il paradigma di base era: ambiente osservabile, deterministico, utente singolo e stati atomici.

Da adesso considereremo un **rilassamento delle assunzioni di base**: ambiente multi agente e rappresentazioni degli stati più complesse.

Ci occuperemo di **specializzazioni** del paradigma nell'ambito dei giochi con avversario: quindi i piani d'azione devono tenere conto dell'avversario.

Vedremo **problemi di soddisfacimento di vincoli**, sempre come ricerca in spazio di soluzioni, con stato a struttura fattorizzata.

Questo ci porterà a considerare **sistemi basati su conoscenza**: lo stato è una "base di conoscenza" a cui rivolgere domande, con una rappresentazione in un linguaggio espressivo e **tecniche di "ragionamento" con inferenze**: logica del prim'ordine e calcolo proposizionale.

6.1 Giochi con Avversario

- Regole semplici e formalizzabili
- Ambiente accessibile e deterministico
- Due giocatori, a turni alterni, a **somma zero** (se uno vince, l'altro perde: sommare i punteggi dà 0 o comunque un risultato costante), a informazione perfetta (tutti i giocatori conoscono lo stato attuale del gioco)
- Ambiente multi-agente competitivo: la presenza dell'avversario rende l'**ambiente strategico**, più difficile rispetto a quanto visto fin'ora
- Complessità e vincoli di tempo reale: mossa migliore nel tempo disponibile

⇒ Questa tipologia di giochi sono un po' più simili ai problemi reali

6.1.1 Ciclo *pianifica-agisci-percepisci*

Due agenti a turno Si può pianificare considerando le possibili risposte dell'avversario e le possibili risposte a quelle risposte... e così via.

Una volta decisa la mossa

Si **esegue** la mossa

Si **vede** cosa fa l'avversario

Si **pianifica** la prossima mossa

6.2 Giochi come problemi di ricerca

Stati: configurazioni del gioco

Player(s): a chi tocca muovere nello stato s

Stato iniziale: configurazione iniziale gioco

Actions(s): mosse legali in s

Result(s, a): stato risultante da una mossa

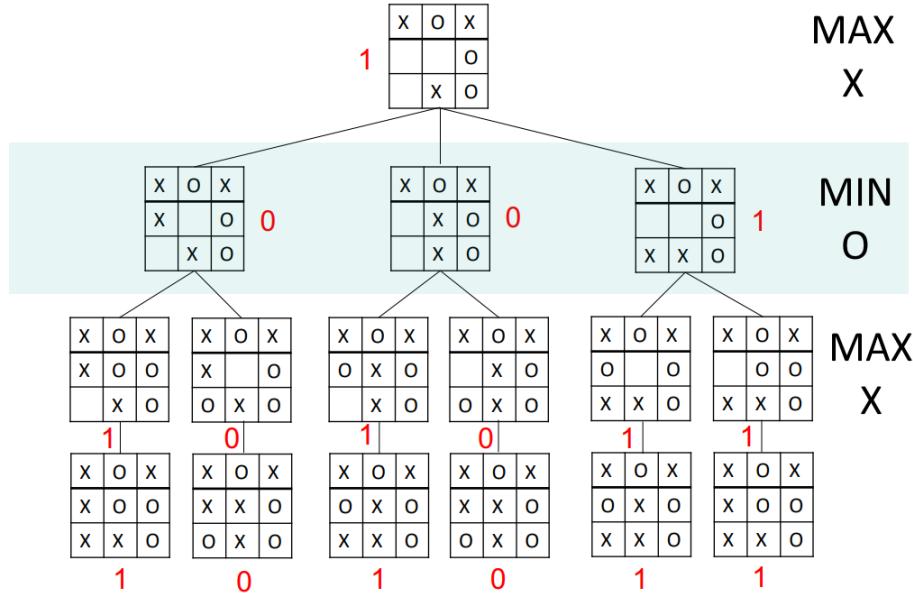
Terminal-Test(s): determina se stato è fine del gioco

Utility(s, p): utilità (o **pay-off**), cioè valore numerico degli stati terminali del gioco per il giocatore p
Es: 1, -1, 0, conteggio punti...

6.2.1 Algoritmo MinMax

Valuto gli stati terminali in base al punteggio/vittoria che ottengo, poi valuto gli stati preterminali in base allo stato terminale in cui mi portano (se mi portano a vittoria o meno a ritroso).

Valuto gli stati intermedi con il valore **minimo** dei risultati se tocca all'avversario (minimo rischio) e col valore **massimo** se tocca a me.



$$\text{Valore MinMax } \text{Minimax}(s) = \begin{cases} \text{Utility}(s, \text{Max}) & \text{se Terminal-Test}(s) \\ \max_{a \in \text{Action}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{se Player}(s) = \text{MAX} \\ \min_{a \in \text{Action}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{se Player}(s) = \text{min} \end{cases}$$

Come conviene esplorare l'albero di gioco? In profondità, perché hanno ampiezza esponenziale.

Algoritmo ricorsivo Di seguito

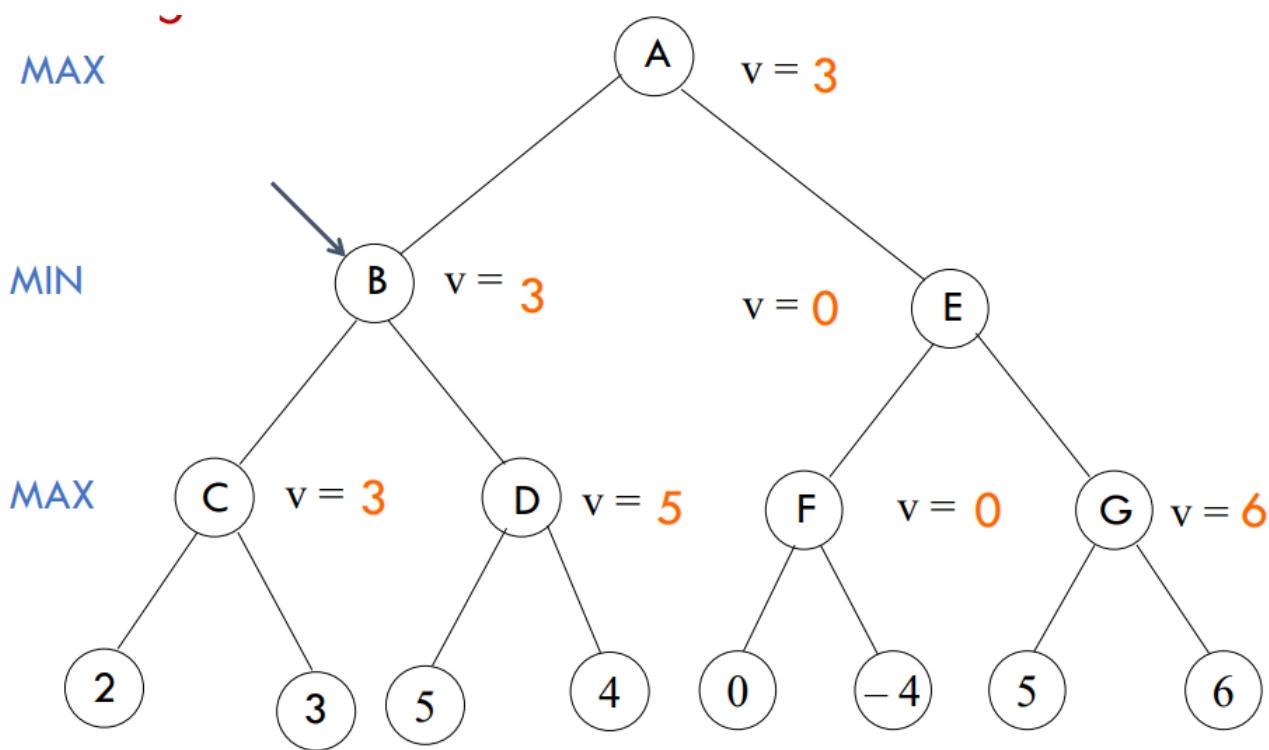
```

function MINIMAX-DECISION(state) returns an action
    return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(s, a))$ 

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
     $v \leftarrow -\infty$ 
    for each a in ACTIONS(state) do
         $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$ 
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
     $v \leftarrow \infty$ 
    for each a in ACTIONS(state) do
         $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$ 
    return v

```



Costo

Tempo: come DF $\rightarrow O(b^m)$

Spazio: $O(m)$

Per gli scacchi (10^{40} nodi) più di 10^{22} secoli \Rightarrow improponibile un'esplorazione sistematica del grafo, se non per giochi molto semplici

Necessarie **euristiche**

6.2.2 Algoritmo Min-Max Euristico (con orizzonte)

In casi più complessi, dove esplorare stati è troppo costoso (es. scacchi), occorre una **funzione di valutazione euristica**.

Strategia Guardare avanti d livelli

Si espande l'albero di ricerca un certo numero di livelli d , compatibile con il tempo e lo spazio disponibili

Si **valutano gli stati ottenuti** e si propaga indietro il risultato con la regola del Minimax

L'algoritmo è simile a prima ma

```
if Terminal-Test(s) then return Utility(s) → if Cutoff-Test(s, d) then return Eval(s)
```

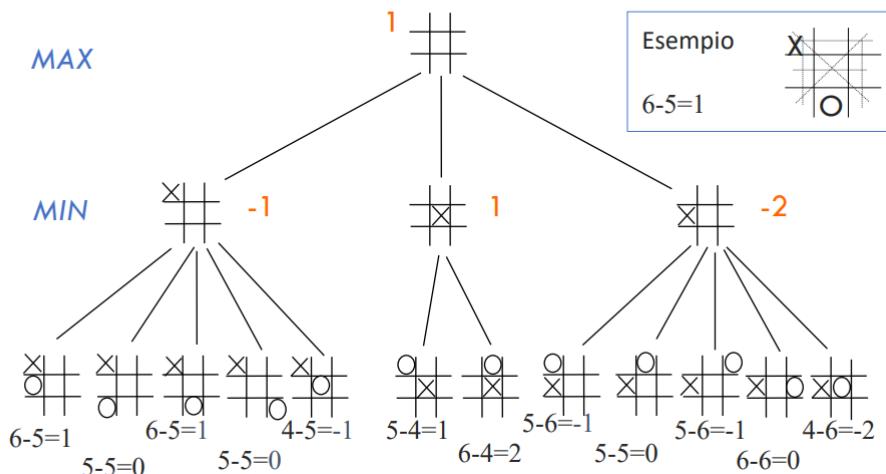
Posto d limite della profondità consentita

$$H\text{-Minimax}(s, d) = \begin{cases} \text{Eval}(s) & \text{se } \text{Cutoff-Test}(s, d) \\ \max_{a \in Action(s)} H\text{-Minimax}(\text{Result}(s, a), d + 1) & \text{se Player}(s) = \text{MAX} \\ \min_{a \in Action(s)} H\text{-Minimax}(\text{Result}(s, a), d + 1) & \text{se Player}(s) = \text{min} \end{cases}$$

|| filetto

$$\begin{aligned} \text{Eval}(s) &= X(s) - O(s) \\ X(s) &\quad \text{righe aperte per X} \\ O(s) &\quad \text{righe aperte per O} \end{aligned}$$

Una configurazione vincente per X viene stimata $+\infty$
Una vincente per O, $-\infty$



Funzione di Valutazione

La funzione di valutazione **Eval** è una **stima dell'utilità attesa** a partire da una certa posizione. La qualità della **funzione** è determinante.

deve essere **consistente** con l'utilità se applicata agli stati terminali del gioco (indurre lo stesso ordinamento)

deve essere **efficiente** da calcolare

deve **riflettere** le probabilità effettive di vittoria (A valutato meglio di B se da A ho più probabilità di vittoria rispetto a B)

combina probabilità con utilità dello stato terminale, può essere appreso con l'esperienza

Esempio Scacchi

Si potrebbe pensare di valutare caratteristiche diverse dello stato: valore del materiale (pedone 1, cavallo/alfiere 3, torre 5, regina 9...), buona disposizione dei pedoni, protezione del re...

Funzione lineare pensata $\text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$

Ma anche combinazioni non lineari delle caratteristiche: alfiere vale di più a fine partita, due alfieri valgono più del doppio di uno solo...

Problemi Noti

Stati Non Quiescenti: l'esplorazione fino ad un certo livello può mostrare una situazione molto vantaggiosa, ma la mossa successiva risulta estremamente svantaggiosa es. regina mangiata dalla torre.

Soluzione: non fermarsi agli stati non quiescenti ma esplorare un po' di più fino ad arrivare a stati in cui la funzione di valutazione non attende variazioni repentine (**stati quiescenti**)

Effetto Orizzonte: si privilegiano mosse **diverse che hanno solo l'effetto di spingere il problema oltre**, per evitare mosse disastrose che alla fine devono per forza accadere

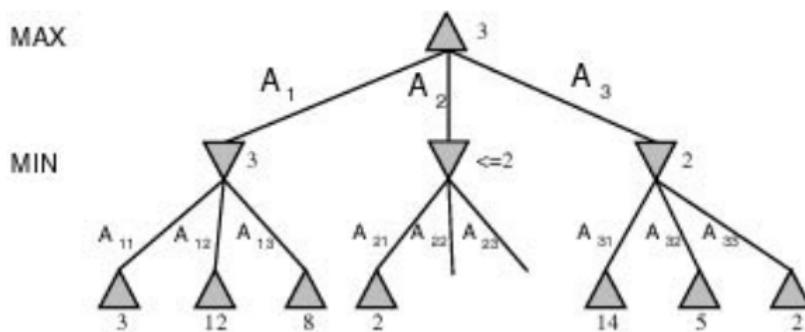
Ottimizzazione Dobbiamo esplorare ogni cammino? No, **esistono tecniche di potatura** che dimezzano la ricerca pur mantenendo decisione min-max corretta (**potatura alfa-beta**)

Potatura Alfa-Beta ridurre spazi ricerca algoritmi min-max

6.2.3 Potatura Alfa-Beta

Tecnica di *potatura* per ridurre l'esplorazione dello spazio di ricerca in algoritmi minmax

Idea



$$\begin{aligned}
 \minmax(\text{radice}) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) = \\
 &= \max(3, \min(2, x, y), 2) = \\
 &= \max(3, z, 2) = 3 \quad \text{con } z \leq 2
 \end{aligned}$$

Algoritmo Consideriamo un nodo v : se c'è una **scelta migliore sopra**, allora quel v non sarà mai raggiunto. Max passerà da α piuttosto che finire a v .

Implementazione Si va avanti in profondità fino al livello desiderato e, propagando indietro i valori, si decide se si può abbandonare l'esplorazione nel sotto-albero.

MaxValue e **MinValue** vengono invocate con **due valori di riferimento iniziali**

$\text{MaxValue} = \alpha$ (inizialmente $-\infty$)

$\text{MinValue} = \beta$ (inizialmente $+\infty$)

Rappresentano **rispettivamente la migliore alternativa per Max e per Min** fino a quel momento

I tagli avvengono quando, nel propagare indietro, si verifica:

$v \geq \beta$ per i nodi Max (taglio β)

$v \leq \alpha$ per i nodi Min (taglio α)

```

function ALPHA-BETA-SEARCH(state) returns an action
  v  $\leftarrow$  MAX-VALUE(state,  $-\infty$ ,  $+\infty$ )
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow -\infty$ 
  for each a in ACTIONS(state) do
    v  $\leftarrow$  MAX(v, MIN-VALUE(RESULT(s,a),  $\alpha$ ,  $\beta$ ))
    if v  $\geq \beta$  then return v  $\leftarrow$  taglio  $\beta$ 
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
  return v

function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow +\infty$ 
  for each a in ACTIONS(state) do
    v  $\leftarrow$  MIN(v, MAX-VALUE(RESULT(s,a),  $\alpha$ ,  $\beta$ ))
    if v  $\leq \alpha$  then return v  $\leftarrow$  taglio  $\alpha$ 
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return v

```

Ordinamento delle mosse La potatura ottimale si ottiene quando ad ogni livello sono generate prima le mosse migliori per chi gioca:

Per i **nodi Max** sono generate prima le mosse con valore più **alto**

Per i **nodi Min** sono generate prima le mosse con valore più **basso**

Complessità $O(b^{m/2})$

Alfa-Beta può arrivare a profondità doppia rispetto a MinMax, ma come avvicinarsi all'ordinamento ottimale?

Ordinamento Dinamico Usando un approfondimento iterativo si possono scoprire ad una iterazione delle informazioni utili per l'ordinamento delle mosse, da usare in una successiva iterazione (**mosse killer**).

Tabella delle trasposizioni: per ogni stato incontrato si memorizza la sua valutazione: situazione tipica è quando $[a_1, b_1, a_2, b_2]$ e $[a_1, b_2, a_2, b_1]$ portano nello stesso stato.

Altri miglioramenti

Potatura in avanti Esplorare solo **alcune mosse ritenute promettenti** e tagliare le altre: beam search, tagli probabilistici basati su esperienza

Database di mosse di apertura e chiusura: nelle prime fasi ci sono poche mosse sensate e ben studiate, mentre per le fasi finali il computer può esplorare in maniera esaustiva e ricordarsi le chiusure migliori

Giochi Multiplayer Invece di aver due valori ho un vettore di valori per nodo. Per propagare all'indietro prendo vettore che migliora la valutazione.

Giochi più complessi

Giochi stocastici, con un lancio di dadi

Giochi parzialmente osservabili: deterministici (mosse deterministiche ma non si conoscono gli effetti delle mosse dell'avversario, es: battaglia navale), **stocastici** (carte distribuite a caso come in molti giochi)

Capitolo 7

Problemi di Soddisfacimento di Vincoli

CSP Sono problemi con una struttura particolare, che si prestano ad algoritmi di ricerca specializzati. Grazie alla struttura si cerca meglio la soluzione: rappresentazione **fattorizzata**, con la quale iniziamo a dire qualcosa sulla struttura dello stato.

Classe ampia Questa classe di problemi è molto ampia, esistono **euristiche generali** da applicare e che consentono la risoluzione di istanze con dimensioni significative.

7.1 Formulazione

Problema descritto da tre componenti:

X, insieme di **variabili**

D, insieme di **domini**

Dove D_i è l'insieme dei valori possibili per X_i

C, insieme di **vincoli**, relazioni tra le variabili

Stato, un **assegnamento parziale/completo** di valori a variabili

$\{X_i = v_i, X_j = v_j, \dots\}$

Stato iniziale: $\{\}$

Azioni: assegnamento di un valore ad una variabile, tra quelli leciti

Soluzione (goal test): **assegnamento completo** (tutte le variabili hanno un valore) e **consistente** (i vincoli sono tutti soddisfatti)

Esempio Il problema delle 8 regine

$X = \{Q_1, \dots, Q_8\}$

$D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$

C_i i vincoli di non attacco

Esempio di vincolo fra Q_1 e Q_2 : $\langle(Q_1, Q_2), \{(1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (2, 5), \dots\}\rangle$

Formulazione incrementale o a stato completo

7.2 Strategie per problemi CSP

Fin'ora potevamo solo ricercare la soluzione nel grafo degli stati, guidati da una metrica definita sullo stato. Adesso possiamo:

Usare delle euristiche specifiche per questa classe di problemi

Fare delle inferenze che ci portano a restringere i domini, quindi a **limitare la ricerca**: **propagazione di vincoli**

Fare **backtracking intelligente**

Tipicamente **un mixto di queste strategie**.

7.2.1 Ricerca in problemi CSP

Ad ogni passo si assegna una variabile: la massima profondità della ricerca è fissata dal numero di variabili n .

Versione ingenua L'ampiezza dello spazio di ricerca è $|D_1| \times |D_2| \times \dots \times |D_n|$, dove $|D_i|$ è la cardinalità del dominio di X_i .

Il fattore di diramazione è pari a $n \cdot d$ al primo passo, $(n - 1) \cdot d$ al secondo... le foglie sarebbero $n! \cdot d^n$

C'è una drastica riduzione dello spazio di ricerca, dovuta al fatto che il **Goal-Test** è **commutativo**: l'ordine con cui si assegnano le variabili non è importante). In realtà, quindi, il fattore di diramazione è pari alla dimensione dei domini d (d^n foglie).

7.2.2 Backtracking

BT Ricerca con backtracking a profondità limitata: **controllo anticipato** della violazione dei vincoli. Intuile andare avanti fino alla fine e poi controllare, si può fare backtracking non appena si scopre che un vincolo è stato violato.

La ricerca è limitata naturalmente in profondità dal numero di variabili, quindi il metodo è completo.

Scelta delle variabili

MRV (minimum remaining values): scegliere la **variabile con meno valori legali residui**, cioè quella più vincolata. Si scoprono prima i fallimenti (**fail first**)

Euristica del grado: scegliere la **variabile coinvolta in più vincoli** con le altre variabili (quella più vincolante, o di grado maggiore).

Da usare a parità di MRV

Scelta dei valori: scegliere il **valore meno vincolante**, quello che esclude meno valori possibili per le altre variabili. Meglio valutare prima assegnamento che ha più probabilità di successo.

Propagazione dei vincoli

Verifica in avanti, Forward Checking: meno costosa, una volta assegnato il valore vado a vedere le variabili direttamente collegate nel grafo dei vincoli (non itero)

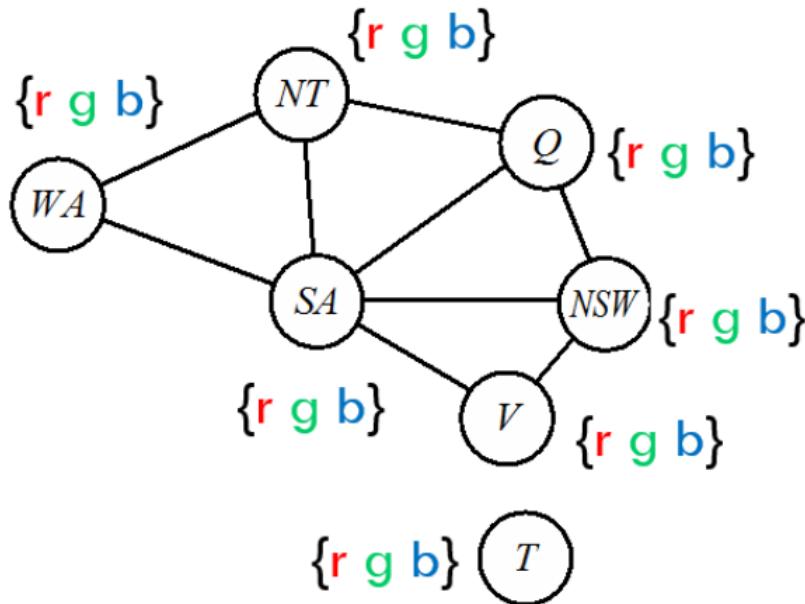
Consistenza di nodo e d'arco: più costosa ma anche più efficace, restringono i valori dei domini delle variabili tenendo conto dei vincoli unari e binari su tutto il grafo (itero finché tutti nodi e archi sono consistenti)

Backtracking ricorsivo per CSP

```

function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)
  
function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment                      trovata soluzione
  var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then                                controllo anticipato
      add {var = value} to assignment
      inferences  $\leftarrow$  INFERENCE(csp, var, value)
      if inferences  $\neq$  failure then                                         riduce i domini
        add inferences to assignment
        result  $\leftarrow$  BACKTRACK(assignment, csp)
        if result  $\neq$  failure then                                         nessun dominio è vuoto
          return result
        remove {var = value} and inferences from assignment                  si disfa lo stato
      return failure
  
```

Esempio di FC (Forward Check) Colorare mappa con Rosso, Verde, Blu.



Backtracking Cronologico Suppongo di avere $\{Q=R, NSW=G, V=B, T=R\}$ e cerco di assegnare SA. Il fallimento genera un backtracking cronologico: si provano tutti i valori alternativi per l'ultima variabile, T, continuando a fallire...

Questo perché **non è la T responsabile del fallimento ma le altre variabili**.

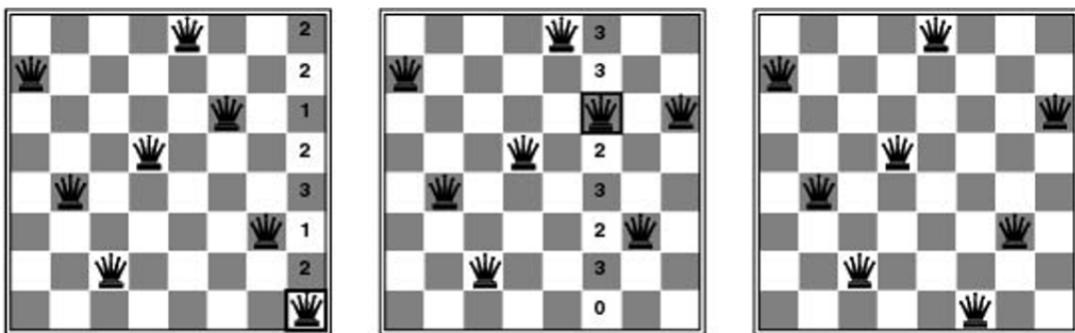
Quindi si può fare un backtracking "intelligente" guidato dalle dipendenze.

Backtracking guidato dalle dipendenze Considero le alternative solo per le variabili che hanno causato il fallimento $\{Q, NSW, V\}$, l'insieme dei conflitti.

Metodi CSP locali I problemi CSP possono essere affrontati con metodi locali. Esempio: **le regine**.

Euristica dei conflitti minimi (min-conflicts): si sceglie un valore che crea meno conflitti. Solitamente si sceglie a caso una delle regole che violano dei vincoli.

Molto efficace: 1 milione di regine in 50 passi!



Conclusioni Abbiamo visti due domini specifici per paradigma risoluzione problemi ricerca: giochi con avversario (complicazione: ambiente strategico e vincoli di tempo reale, non posso pianificare azioni da eseguire "a occhi chiusi") e CSP (grazie a rappresentazione stato più ricca tecniche problem solving specializzabili e usate per risolvere istanze di problemi di dimensioni maggiori)

Prossimamente: **sistemi basati su conoscenza**. Conoscenza implica capacità inferenziali, con rappresentazione stato ancora più ricco con linguaggio rappresentazione conoscenza. L'inferenza è anch'essa un problema di ricerca in uno spazio di stati.

Capitolo 8

Agenti Basati su Conoscenza

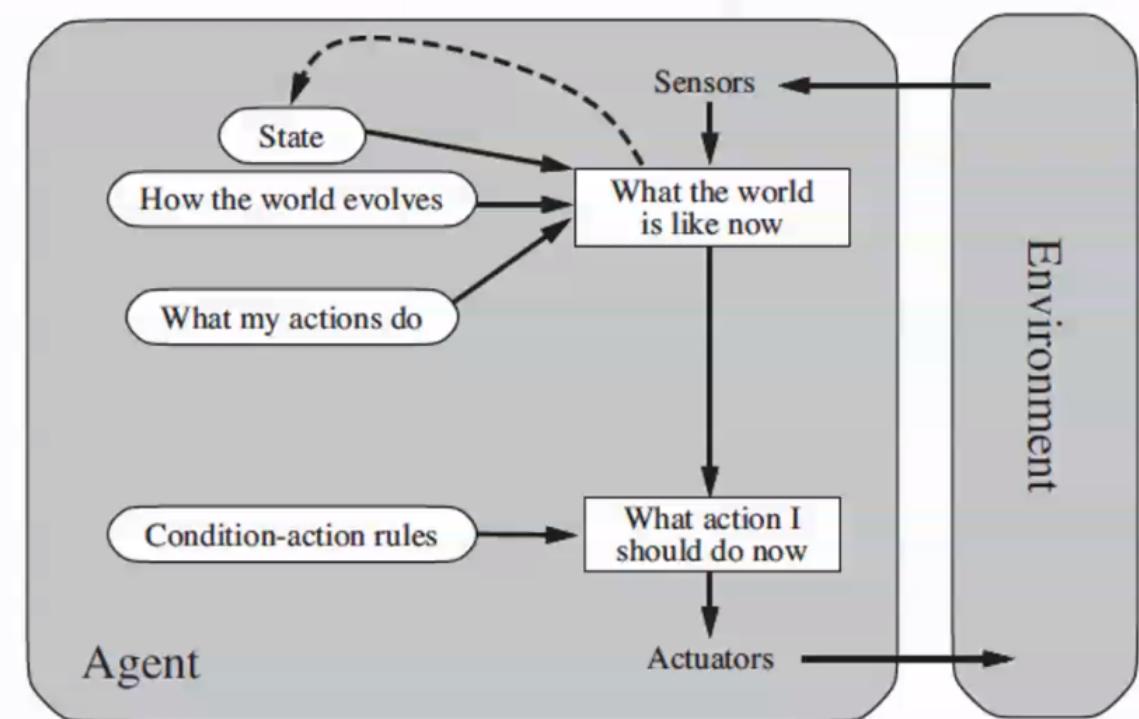
Fin'ora Abbiamo trattato agenti con stato e con obiettivo, in mondi osservabili, con stati atomici e azioni descrivibili in maniera semplice, facendo **enfasi sul processo di ricerca**.

Poiabbiamo trattato descrizioni **fattorizzate**, come nel CSP, che consentono di iniziare a *guardare dentro* lo stato, descritto come un **insieme di caratteristiche rilevanti** (**feature**)

Adesso Vedremo come costruire agenti dotati di capacità inferenziali. Cerchiamo di **migliorare la capacità di ragionamento**, dotando gli agenti di rappresentazioni del mondo più complesse non descrivibili semplicemente.

Agenti basati su conoscenza, con all'interno una **base di conoscenza** (**knowledge base**, o **KB**) con conoscenza espressa in forma esplicita e dichiarativa attraverso un **linguaggio**.

8.1 Agenti Knowledge-Based



La maggior parte dei problemi di I.A. sono *"knowledge-intensive"*. Il mondo è tipicamente **complesso**: serve una rappresentazione parziale e incompleta (**astrazione**) del mondo utile agli scopi dell'agente.

Per ambienti parzialmente osservabili e complessi ci servono linguaggi di rappresentazione della conoscenza più espresivi e **capacità inferenziali**.

La conoscenza può essere codificata a mano, acquisita o estratta da testi ed esperienza.

Approccio dichiarativo vs procedurale Tipicamente la conoscenza in una KB viene espressa in **forma dichiarativa**. L'alternativa è **codificare la conoscenza in maniera procedurale**, attraverso un programma che implementa il processo decisionale. Un agente KB (con conoscenza dichiarativa) è più flessibile: perché è più semplice ricevere conoscenza in maniera incrementale e modificare il comportamento con l'esperienza.

8.1.1 Il mondo del Wumpus

Esempio Agente in 1, 1: esplorare e trovare oro

Misura delle prestazioni

- +1000 se trova l'oro, torna in [1, 1] ed esce
- 1000 se muore
- 1 per ogni azione
- 10 se usa la freccia

Percezioni

- puzzo nelle caselle adiacenti al Wumpus
- brezza nelle caselle adiacenti alle buche
- luccichio nelle caselle con l'oro
- bump se batte in un muro
- urlo se il Wumpus viene ucciso
- L'agente non percepisce la sua locazione

Azioni

- Avanti
- A destra di 90 gradi
- A sinistra di 90 gradi
- Scaglia la freccia (solo una)
- Esce

Ambienti generati a caso, con [1, 1] safe.

4			
3		 	
2			
1	 START		
	1	2	3
			4

8.1.2 Knowledge-Base

KB Insieme di **enunciati** espressi in un linguaggio di rappresentazione.
L'agente interagisce con una KB con un'interfaccia funzionale **Tell-Ask**:

Tell per aggiungere nuovi enunciati alla KB

Ask per interrogare la KB

Retract per eliminare enunciati

Enunciati in KG rappresentano la conoscenza agente. Le **risposte** α devono essere tali che α è una conseguenza della KB.

Problema Il problema da risolvere: data una base di conoscenza KB che contiene una rappresentazione dei fatti ritenuti veri, vorrei sapere se un certo fatto α è vero di conseguenza, cioè vorrei sapere se

$$\text{KB} \models \alpha \quad : \quad (\text{conseguenza logica})$$

Programma di un agente KB Di seguito

```
function Agente-KB (percezione) returns azione
    persistent: KB, una base di conoscenza
        t, un contatore, inizialmente a 0, che indica il tempo
        TELL(KB, Costruisci-Formula-Percezione(percezione, t))
        azione = ASK(KB, Costruisci-Query-Azione(t))
        TELL(KB, Costruisci-Formula-Azione(azione, t))
        t = t + 1
    return azione
```

KB vs DB Una **base di conoscenza** è una **rappresentazione esplicita, parziale** e compatta in un linguaggio simbolico: contiene sia fatti esplicativi (es. *Pozzo in [3, 3]*) ma anche fatti generali o regole (es: brezza in caselle adiacenti a pozzi)

Una **base di dati** invece contiene solo fatti specifici e consente solo il recupero.

La differenza è che la **KB ha capacità inferenziale**: si possono **derivare nuovi fatti** da quelli memorizzati esplicitamente.

Trade-Off Fondamentale della Rappresentazione della Conoscenza Trovare il **giusto compromesso tra l'espressività** del linguaggio di rappresentazione e la **complessità** del meccanismo inferenziale.

Più un linguaggio è espressivo meno è efficiente il meccanismo inferenziale. Questi due obiettivi sono in contrasto, bisogna mediare e trovare un **compromesso**.

Formalismi per la Rappresentazione della Conoscenza

Un formalismo per la rappresentazione della conoscenza ha tre componenti:

Sintassi: un linguaggio composto da un vocabolario e da regole per la formazione delle frasi (**enunciati**)

Semantica: stabilisce la corrispondenza tra gli enunciati e fatti del mondo. Se un agente ha un enunciato α nella sua KB, allora crederà che il fatto corrispondente ad α sia vero nel mondo

Meccanismo Inferenziale: codificato o meno tramite regole di inferenza come nella logica, che ci consente di inferire nuovi fatti

Logica come linguaggio Qual è la complessità computazionale del problema $\text{KB} \models \alpha$ nei vari linguaggi logici? Quali algoritmi decisione e strategia ottimizzazione?

Linguaggi logici: calcolo proposizionale (POP) e logica dei predicati (FOL). Sono adatti per la rappresentazione della conoscenza? Da una parte sono anche troppo complessi, in particolare il FOL. Dall'altra, ci sono meccanismi non posseduti da FOL e POP ma che sono utili nelle KB.

Rivistazione di PROP e FOL per rappresentazione conoscenza, con attenzione ad algoritmi e complessità. **Contrazioni:** linguaggi a regole e **programmazione logica**.

8.1.3 Algoritmo TT-entails

$\text{KB} \models \alpha?$

Enumera tutte possibili interpretazioni KB (k simboli, 2^k possibili interpretazioni)

Per ciascuna interpretazione

Se non soddisfa KB, OK

Se soddisfa KB, si controlla che soddisfi anche α

Se si trova anche solo una interpretazione che soddisfa KB e non α , la risposta sarà NO

Algoritmo Di seguito

```

function TT-ENTAILS?(KB,  $\alpha$ ) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
           $\alpha$ , the query, a sentence in propositional logic

  symbols  $\leftarrow$  a list of the proposition symbols in KB and  $\alpha$ 
  return TT-CHECK-ALL(KB,  $\alpha$ , symbols, { })

```

```

function TT-CHECK-ALL(KB,  $\alpha$ , symbols, model) returns true or false
  if EMPTY?(symbols) then
    if PL-TRUE?(KB, model) then return PL-TRUE?( $\alpha$ , model)
    else return true // when KB is false, always return true
  else do
     $P \leftarrow$  FIRST(symbols)
    rest  $\leftarrow$  REST(symbols)
    return (TT-CHECK-ALL(KB,  $\alpha$ , rest, model  $\cup$  { $P = \text{true}$ })
            and
            TT-CHECK-ALL(KB,  $\alpha$ , rest, model  $\cup$  { $P = \text{false}$ }))

```

Esempio $(\neg A \vee B) \wedge (A \vee C) \models (B \vee C)$?

```

TT-CHECK-ALL(  $(\neg A \vee B) \wedge (A \vee C)$ ,  $(B \vee C)$ , [A, B, C], {} )

TT-CHECK-ALL(  $(\neg A \vee B) \wedge (A \vee C)$ ,  $(B \vee C)$ , [B, C], {A = true} )
TT-CHECK-ALL(  $(\neg A \vee B) \wedge (A \vee C)$ ,  $(B \vee C)$ , [C], {A = true, B = true} )
OK TT-CHECK-ALL(  $(\neg A \vee B) \wedge (A \vee C)$ ,  $(B \vee C)$ , [], {A = true, B = true, C = true} )
OK TT-CHECK-ALL(  $(\neg A \vee B) \wedge (A \vee C)$ ,  $(B \vee C)$ , [], {A = true, B = true, C = false} )
TT-CHECK-ALL(  $(\neg A \vee B) \wedge (A \vee C)$ ,  $(B \vee C)$ , [C], {A = true, B = false} )
OK TT-CHECK-ALL(  $(\neg A \vee B) \wedge (A \vee C)$ ,  $(B \vee C)$ , [], {A = true, B = false, C = true} )
OK TT-CHECK-ALL(  $(\neg A \vee B) \wedge (A \vee C)$ ,  $(B \vee C)$ , [], {A = true, B = false, C = false} )
TT-CHECK-ALL(  $(\neg A \vee B) \wedge (A \vee C)$ ,  $(B \vee C)$ , [B, C], {A = false} ) ...

```

Solo alla fine, **dopo aver provato tutti i possibili assegnamenti**, possiamo rispondere se $(B \vee C)$ è conseguenza logica.

8.2 Algoritmi per la soddisfacibilità (SAT)

Usano una KB come **insieme di clausole**, cioè insiemi di letterali:

{A, B} { \neg B, C, D} { \neg A, F}

La forma a clausole è la **forma normale congiuntiva (CNF)**, cioè una congiunzione di disgiunzioni letterali:

{ $A \vee B$ } \wedge { $\neg B \vee C \vee D$ } \wedge { $\neg A \vee F$ }

Non è restrittiva: è sempre possibile ottenerla con trasformazioni che preservano l'equivalenza logica.

Trasformazione in forma a clausole I passi sono:

- Eliminazione della \Leftrightarrow : $(A \Leftrightarrow B) = (A \Rightarrow B) \wedge (B \Rightarrow A)$
- Eliminazione dell' \Rightarrow : $(A \Rightarrow B) = (\neg A \vee B)$
- Negazioni all'interno (de Morgan):
 - $\neg(A \vee B) = (\neg A \wedge \neg B)$
 - $\neg(A \wedge B) = (\neg A \vee \neg B)$
- Distribuzione di \vee su \wedge : $(A \vee (B \wedge C)) = (A \vee B) \wedge (A \vee C)$

8.2.1 Algoritmo DPLL

DPLL Davis, Putman, Lovemann, Loveland. Parte da una KB in forma a clausole. Enumerazione in profondità di tutte le possibili interpretazioni alla ricerca di un modello. Tre miglioramenti rispetto TTEntails:

Terminazione anticipata: si può decidere sulla verità di una clausola anche con interpretazioni parziali, basta che un letterale sia vero.

Se A è *true*, lo sono anche $\{A, B\}$ e $\{A, C\}$ indipendentemente dai valori di B e di C .

Se anche una sola clausola è falsa, l'interpretazione non può essere un modello dell'insieme delle clausole.

Euristica dei **simboli puri** (o letterali) **puri**: un **simbolo puro** è un simbolo che **appare con lo stesso segno in tutte le clausole**. Ad esempio, nella KB $\{A, \neg B\}, \{\neg B, \neg C\}, \{C, A\}$ i simboli A e B sono puri.

Nel determinare un simbolo se è puro possiamo trascurare occorrenze simbolo in clausole già rese vere.

Se simbolo è puro può essere assegnato a *true* se positivo e *false* se negativo senza escludere modelli utili: se le clausole hanno un modello continueranno ad averlo anche dopo questo assegnamento. L'assegnamento è obbligato.

Euristica delle **clausole unitarie**: una **clausola unitaria** è una **clausola con un solo letterale non assegnato**. Ad esempio, $\{B\}$ è unitaria, ma anche $\{B, \neg C\}$ quando $C = \text{true}$.

Conviene assegnare prima valori ai letterali in clausole unitarie. L'assegnamento è univoco (*true* se positivo, *false* se negativo)

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*

inputs: s , a sentence in propositional logic

$\text{clauses} \leftarrow$ the set of clauses in the CNF representation of s

$\text{symbols} \leftarrow$ a list of the proposition symbols in s

return DPLL($\text{clauses}, \text{symbols}, \{\}$)

function DPLL($\text{clauses}, \text{symbols}, \text{model}$) **returns** *true* or *false*

if every clause in clauses is true in model **then return** *true*

if some clause in clauses is false in model **then return** *false*

$P, \text{value} \leftarrow \text{FIND-PURE-SYMBOL}(\text{symbols}, \text{clauses}, \text{model})$

if P is non-null **then return** DPLL($\text{clauses}, \text{symbols} - P, \text{model} \cup \{P=\text{value}\}$)

$P, \text{value} \leftarrow \text{FIND-UNIT-CLAUSE}(\text{clauses}, \text{model})$

if P is non-null **then return** DPLL($\text{clauses}, \text{symbols} - P, \text{model} \cup \{P=\text{value}\}$)

$P \leftarrow \text{FIRST}(\text{symbols}); \text{rest} \leftarrow \text{REST}(\text{symbols})$

return DPLL($\text{clauses}, \text{rest}, \text{model} \cup \{P=\text{true}\}$) **or**

DPLL($\text{clauses}, \text{rest}, \text{model} \cup \{P=\text{false}\}$)

Miglioramenti di DPLL DPLL è completo e termina sempre. Alcuni miglioramenti:

Analisi di componenti (sottoproblemi indipendenti): se le variabili possono essere suddivise in sottoinsiemi disgiunti

Ordinamento di variabili e valori: scegliere la variabile che compare in più clausole

Backtracking intelligente

Altre ottimizzazioni...

8.2.2 Metodi locali per SAT

Formulazione

Gli stati sono assegnamenti completi

L'obiettivo è un assegnamento che soddisfa tutte le clausole (un **modello**)

Si parte da un assegnamento casuale

Ad ogni passo **si cambia il valore di una proposizione (flip)**

Gli stati sono valutati contando il numero di clausole non soddisfatte, meno sono meglio è (o soddisfatte)

Algoritmi Ci sono molti minimi locali, per sfuggire ai quali bisogna introdurre delle perturbazioni casuali. Ad esempio, hill-climbing con riavvio casuale, simulated annealing...

C'è stata molta sperimentazione per trovare il miglior compromesso tra il grado di avidità e la casualità.

WalkSAT è uno degli algoritmi più semplici ed efficaci.

8.2.3 Algoritmo WalkSAT

Ad ogni passo **sceglie a caso una clausola non ancora soddisfatta**. Poi **sceglie un simbolo da modificare (flip)**, scegliendo con probabilità p (di solito 0.5) tra una delle due seguenti possibilità:

Passo casuale: sceglie un simbolo a caso da flippare

Passo di ottimizzazione: sceglie quello che rende più clausole soddisfatte

Si arrende dopo un numero predefinito di flip.

```

function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
            p, the probability of choosing to do a “random walk” move, typically around 0.5
            max_flips, number of flips allowed before giving up

  model  $\leftarrow$  a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
    if model satisfies clauses then return model
    clause  $\leftarrow$  a randomly selected clause from clauses that is false in model
    with probability p flip the value in model of a randomly selected symbol from clause
    else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure

```

Analisi Se *maxflips* è ∞ e l'insieme clausole è soddisfacibile, prima o poi termina. Ma **non può essere usato per verificare l'insoddisfacibilità** (è **incompleto**). Il problema è decidibile ma l'algoritmo è incompleto.

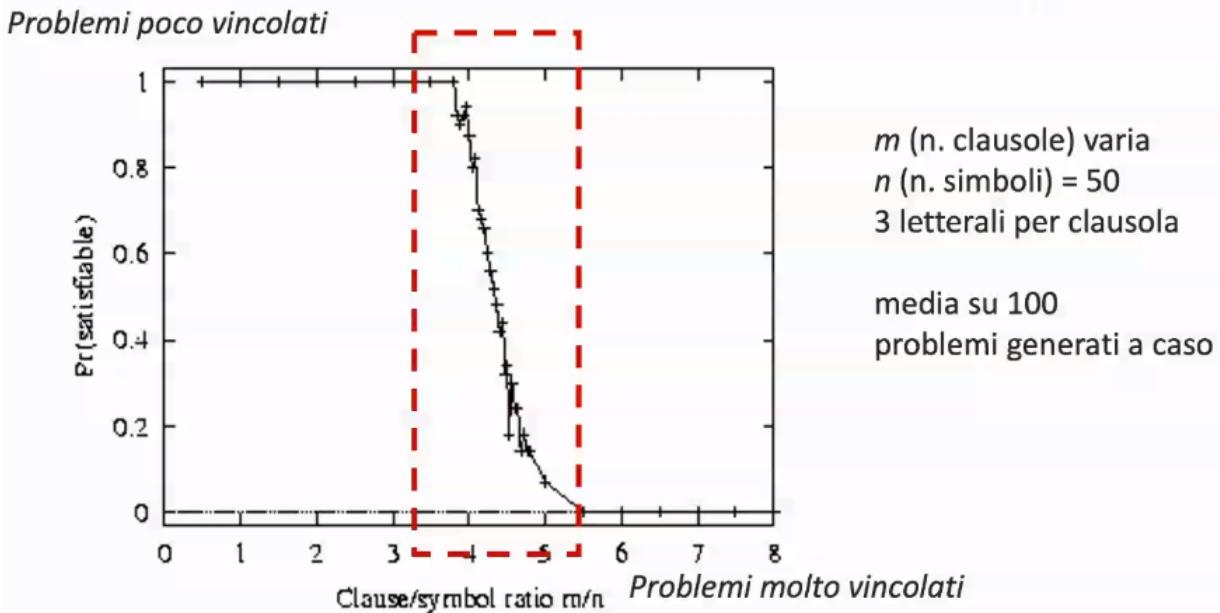
Va bene per cercare un modello, sapendo che c'è. Ma se è insoddisfacibile, non termina.

Lo si usa perché è efficiente in tempo e spazio.

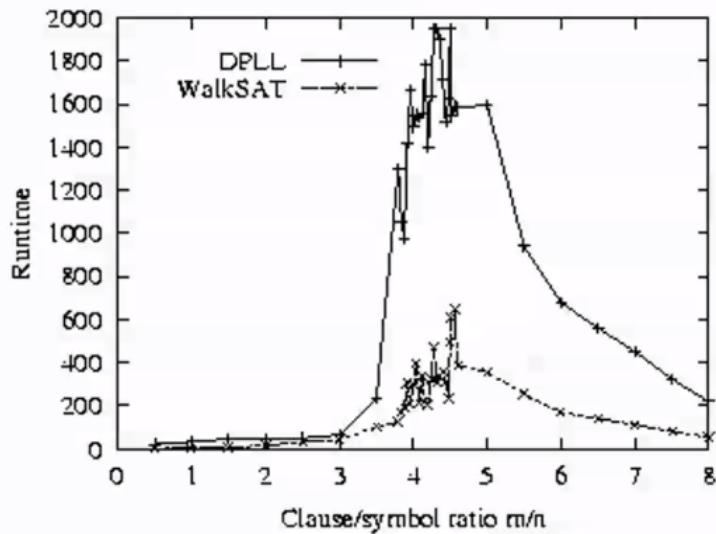
Problemi SAT difficili Se un problema ha molte soluzioni (sotto-vincolato) è più probabile che WalkSAT ne trovi una in tempi brevi. Per esempio, se ha 16 soluzioni su 32, un assegnamento ha il 50% di probabilità di essere giusto, quindi **2 passi in media**.

Più grande è il rapporto più vincolato è il problema.

Probabilità di soddisfabilità in funzione di $\frac{m}{n}$



Confronto tra DPLL e WalkSAT



8.3 Inferenza come Deduzione

Un altro modo per decidere se $KB \models A$ è usare un meccanismo di **dimostrazione (deduzione)**.

Si scrive $KB \vdash A$ (**A è deducibile da KB**) e la deduzione avviene specificando delle regole di inferenza: dovrebbero derivare solo le formule che sono conseguenza logica e **tutte** le formule che sono conseguenza logica.

Sono le proprietà della **correttezza** e **completezza**.

Correttezza Se $KB \vdash A$, allora $KB \models A$

Tutto ciò che è derivabile è conseguenza logica. Le regole preservano la verità

Completezza Se $KB \models A$, allora $KB \vdash A$

Tutto ciò che è conseguenza logica è ottenibile tramite il meccanismo di inferenza.

Dimostrazione come ricerca Come decidere ad ogni passo quale regola di inferenza applicare? A quali premesse? Come evitare esplosione combinatoria? Problema esplorazione spazio di stati. Ci riguarda perché vogliamo progettare algoritmi di inferenza. Una **procedura di dimostrazione** definisce direzione e strategia di ricerca.

Direzione Nella dimostrazione di teoremi **conviene procedere all'indietro**. Con un'applicazione in avanti delle regole di inferenza non controllata posso ottenere, ad esempio da $A, B : A \wedge B, A \wedge (A \wedge B) \dots$. All'indietro invece, se voglio dimostrare $A \wedge B$, si cerca di dimostrare A e poi B . Se voglio dimostrare $A \Rightarrow B$, assumo A e cerco di dimostrare $B \dots$

Strategia Anche assumendo insieme di regole di inferenza completo se l'algoritmo non è completo potrei non trovare soluzione. La complessità è alta: è un problema **decidibile ma NP-completo**.

8.3.1 Regola di risoluzione per PROP

Meno regole uso meglio è, senza rinunciare alla completezza. L'unica regola di inferenza è la regola di risoluzione, che presuppone la forma a clausole. Con due clausole, una P e l'altra che contiene $\neg P$, possiamo considerare l'OR degli altri elementi togliendo P e $\neg P$. Cioè

$$\frac{\{P, Q\} \quad \{\neg P, R\}}{\{Q, R\}}$$

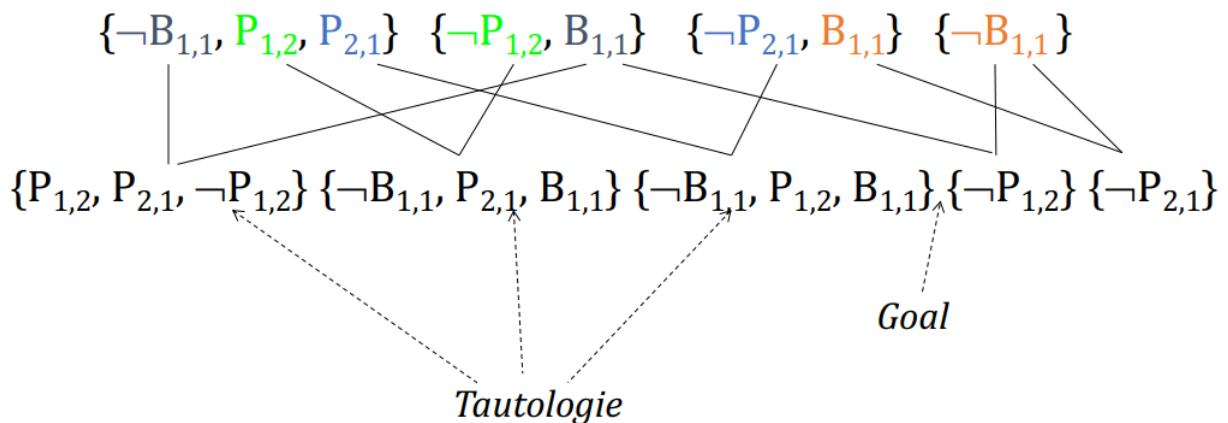
Corretta? Basta pensare ai modelli.

In generale Con l, m **letterali**, simboli di proposizione positivi o negativi. l_i e m_j sono **uguali ma di segno opposto**.

$$\frac{\{l_1, l_2, \dots, l_i, \dots, l_k\} \quad \{m_1, m_2, \dots, m_j, \dots, m_n\}}{\{l_1, l_2, \dots, l_{i-1}, l_{i+1}, \dots, l_k, m_1, m_2, \dots, m_{j-1}, m_{j+1}, \dots, m_n\}}$$

Caso particolare $\frac{\{P\} \quad \{\neg P\}}{\{\}}$: **clausola vuota o contraddizione**.

Grafo di risoluzione



Attenzione! $\{B, N\} \quad \{\neg B, \neg N\} \rightarrow \{\}$ NO!

Diventa o $\{N, \neg N\}$ oppure $\{B, \neg B\}$, un passo alla volta!

Siamo sicuri che basti una regola? Completezza: $KB \models \alpha \Rightarrow KB \vdash_{res} \alpha$? Non sempre. Un controesempio è $KB \models \{A, \neg A\}$ ma non è vero che $KB \vdash_{res} \{A, \neg A\}$
Ma ho due strumenti:

Teorema di risoluzione KB insoddisfacibile $\Leftrightarrow KB \vdash_{res} \{\}$ **completezza**

Teorema di refutazione $KB \models \alpha \Leftrightarrow (KB \cup \{\neg \alpha\})$ è insoddisfacibile

Nell'esempio $KB \cup FC(\neg(A \text{ or } \neg A))$ è insoddisfacibile? Sì, perché $KB \cup \{A\} \cup \{\neg A\} \vdash_{res} \{\}$ in un passo

Conclusioni Abbiamo visto che gli agenti KB che usano il calcolo proposizionale come linguaggio di rappresentazione possono decidere se $\text{KB} \models \alpha$. Il problema è decidibile ma intrattabile (NP al caso peggiore). Esistono algoritmi efficienti e completi che consentono di affrontare problemi di grosse dimensioni, ed esistono algoritmi locali particolarmente efficienti ma non completi. Oppure per via deduttiva.

8.3.2 Logica del Primo Ordine

8.3.3 Inferenza nella Logica del Prim'Ordine

Regole d'inferenza per \forall : istanziazione dell'Universale

$$\frac{\forall x \ A[x]}{A[g]}$$

Da $\forall x \ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$ si possono ottenere

$\text{King(john)} \wedge \text{Greedy(john)} \Rightarrow \text{Evil(john)}$

$\text{King(father(john))} \wedge \text{Greedy(father(john))} \Rightarrow \text{Evil(father(john))}$

Regola per \exists : Istanziazione dell'Esistenziale

$$\frac{\exists x \ A[x]}{A[K]}$$

Se \exists non compare nell'ambito di \forall , K è una costante nuova (costante di Skolem)

Altrimenti va introdotta una funzione (di Skolem) nelle variabili quantificate universalmente

$\exists x \ \text{Padre}(x, G)$

Grounding o proposizionalizzazione

Creo tante istanze delle formule quantificate universali quanti sono oggetti menzionati

Eliminare quantificatori esistenziali skolemizzando. KB diventa proposizionale e applicare algoritmi visti, ma problemi: costanti in numeri finiti, ma se ci sono funzioni il numero di istanze da creare è infinito $\text{john}, \text{padre(john)}, \text{padre(padre(john))}, \dots$

8.3.4 Teorema di Herbrand

Se KB soddisfa A allora c'è dim che coinvolge solo sottinsiemi finiti della KB proposizionalizzata.

Si può procedere incrementalmente

creo istanze con costanti

creo quelle con un solo livello di annidamento Padre(john)

creo quelle con due livelli padre(padre(j))

Se KB non soddisfa A, il processo non termina. **Semidecidibile**.

Forma a clausole costanti, funz, pred come definiti, escludiamo formule atomiche del tipo $(t_1 = t_2)$

Una clausola è insieme di letterali (formula atomica eventualmente negata) che rappresenta la loro **disgiunzione**.

Trasformazione in forma a clausole th: per ogni formula chiusa alfa del FOL è possibile trasformare in maniera effettiva un insieme di clausole $\text{FC}(\alpha)$ soddisfacibile sse alfa lo era (insoddisfacibile sse alfa lo era)

Esempio trasformazione skolemizzazione non preserva l'equivalenza

Unificazione operazione con la quale si determina se due espressioni possono essere rese identiche mediante una sostituzione di termini a variabili.

Risultato è la sostituzione che rende le due espressioni identiche, detta **unificatore**, o **FAIL** se le espressioni non sono unificabili

Sostituzione insieme finito di associazioni tra var e termini in cui ogni var compare una sola volta sulla sx. Es $\{x_1/A, x_2/f(x_3), x_3/B\}$ significa che A va sostituita a x_1 , $f(x_3)$ sost a $x_2 \dots$

A sx var a dx costanti e var, con la restrizione che la var sulla sx non può comparire anche sulla dx della stessa "coppia".

Applicazione Sia σ sost e A espressione, Asigma istanza generala dalla sost (delle var con corrisp espress) In AIMA SUBST(σ , A). Var sostuite simultaneamente e eseguo un solo passo di sostituzione

Espressioni unificabili se esiuste sostituz che le rende identiche (unificatore). In genere l'unificatore τ non è unico. Vorremmo l'unificatore più generale di tutti (MGU). **Teorema:** l'**unificatore più generale è unico** a meno dei nomi delle variabili (l'ordine non conta)

Algoritmo di unificazione input p,q espressioni, restituisce MGU Θ se esiste, `unify(p,q) = theta` tale che `subst(theta, p) = subst(theta, q)`, altrimenti `fail`.

Esplora in parallelo le due espressioni e costruisce l'unificatore strada facendo. Fallisce non appena trova espressioni non unificabili. Una causa del fallimento sono sost del tipo $x=f(x)$, questo controllo si chiama **occur-check**.

Algoritmo

Esempi

8.4 Definizione e Confronto di Euristiche Ammissibili

Mondo dei blocchi Problema classico, parte dei micromondi usati soprattutto per la pianificazione. Serie di blocchi su un tavolo impilati e vogliamo raggiungere certa config. final. Mosse sono spostare blocco su tavolo o su altro blocco, a condizione che blocco sia libero senza blocchi sopra e blocco destinazione anche.

Euristica H1 Numero di blocchi appoggiati su blocco sbagliato (incluso tavolo)

Euristica H2 Numero blocchi con supporto (torre sotto) sbagliato

Sono ammissibili? Sono ammissibili se servono almeno H_1 mosse per giungere in stato goal. H_2 è più accurata perché domina H_1 , valore più alto per ogni stato possibile. Questo perché $H_2 \Rightarrow H_1$, i blocchi contati in H_1 vengono contati anche in H_2 , ma H_2 ha casi non contati da H_1 .

Per trovare euristica bisogna contare quanto dista soluzione e usare questa come euristica.

Euristica = n ammissibile se servono almeno n mosse per arrivare in stato finale, ≥ 0 ovunque ma = 0 solo in stati goal.

Capitolo 9

Strategie di risoluzione

Abbiamo visto metodi di risoluzione KB in forma clausole unificazione e regola srisoluzione (estensioen rispetto regola PROP)

Come rendere più efficiente il meccanismo di risoluzione? Strategie di risoluzione: tecniche per esplorare in maniera efficiente grafo di risoluzione, possibilmente senza perdere completezza.

Un percorso che ci proterà a giustificare i sistemi a regole e le restrizioni del FOL associate.

9.1 Strategia di risoluzione

Distinzione tra

strategia di cancellazione: ci sono clausole da eliminare?

strategia di restrizione: posso usare ad ogni passo solo alcune clausole?

strategia di ordinamento: posso risolvere i letterali in un ordine specifico?

Tutto possibilmente senza perdere completezza

9.1.1 Strategie di Cancellazione

Rimuovere dalla KB clausole che non saranno mai utili nel processo di risoluzione

Clausole con Letterali Puri

Letterali che non hanno il loro negato nella KB

Non potranno mai essere risolte con altre clausole per ottenere {}, tanto vale eliminarle

Non si perdono soluz

Tautologie

Clausole con due letterali identici e complementari

Non basta che siano unificabili e di segno opposto

Tautologie possono essere generate quindi controllo da fare ad ogni passo

Clausole Sussunte

Sussunte (implicate)

In generale a sussume b se esiste sigma asigma in b

Se un'istanta di a con la sostituzione sigma è un sottoinsieme di b

b può essere ricavata da a, quindi b può essere eliminata senza perdere soluzioni. Clausole sussunte possono essere generate.

9.1.2 Strategie di Restrizione

Ad ogni passo si sceglie tra un sottoinsieme di possibili clausole

Risoluzione unitaria

Almeno una delle due clausole è unitaria (un solo letterale)

Converge molto rapidamente, perché ad ogni passo si elimina una clausola

Facile da implementare

Ma non è completa

Completa per **clausole di Horn**: clausole con **al più** un letterale positivo

Risoluzione lineare

Prendo ultima clausola generata con una clausola da input (della KB iniziale), oppure una clausola antenata

Risoluzione guidata dal goal/da insieme di supporto (sottinsieme della KB responsabile dell'insoddisfacibilità)

Almeno una delle due clausole appartiene a questo insieme o a suoi discendenti

Tipicamente, assumendo la KB iniziale consistente, si sceglie come insieme di supporto iniziale il negato del goal. È come procedere all'indietro dal goal. Strategia completa, per la refutazione.

9.1.3 Sottoinsieme a regole del FOL

Clausole di Horn definite Clausole con **esattamente** un letterale positivo.

Possano essere riscritte come fatti e regole

Una **KB a regole**

9.1.4 Sistemi a regole logici

Se la kb contiene solo clausole horn definite i meccanismi inferenziali sono più semplici, processo molto più giudicato senza rincuniare alla completezza: risolutori in tempo lineare per il caso proposizionale. Nota: è restrittivo e non coincide né con PROP né con FOL.

Uso delle regole in avanti e indietro

Backward Chaining

PROLOG ha kb a regole e procede applicando regole in avanti o indietro

Forward Chaining

9.1.5 Programmazione Logica

programmi logici sono kb costituiti di clausole horn definite, espressi come fatti e regole con sintassi alternativa altre convenzioni: in PL le variabili sono indicate con le lettere maiuscole e le costanti con le lettere minuscole
Rappresentazione del goal

Interpretazione dichiarativa

Interpretazione procedurale

9.1.6 Risoluzione SLD

Selection Linear Definit-Clauses struttura lineare ordinata basata su un insieme di supporto (clausola goal), lineare nell'input. SLD è completa per clausole di horn.

Alberi di Risoluzione SLD Albero SLD per il goal `antenato(lia, mark)`

Strategia di visita dell'albero sld e prolog Prolog genera albero di risoluzione con ricerca in profondità e backtracking su fallimento quindi non è una strategia completa. Programmatore ha possibilità di controllare ordine di generazione perché regole applicate nell'ordine in cui sono scritte nel file.