

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	Conslusioni 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	Assunzioni	37
5.2.2	Panorama dello spazio degli stati	39
5.2.3	Algoritmo Hill Climbing	39
6	I Giochi con Avversario	41
6.1	Giochi con Avversario	41
6.2	Ciclo <i>pianifica-agisci-percepisci</i>	41
6.3	Giochi come problemi di ricerca	41
6.4	Algoritmo Min-Max	42
6.5	Algoritmo Min-Max Euristico (con orizzonte)	42
6.6	Funzione di Valutazione	42
6.6.1	Ottimizzazione	42
6.6.2	Algoritmo Alfa-Beta	43
6.7	Problemi di Soddisfacimento di Vincoli	43
7	Agenti Basati su Conoscenza	45
7.1	Il mondo del Wumpus	46
7.2	Calcolo Proposizionale	47

Introduzione

Alessio Micheli, Maria Simi

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

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

Psicologia cognitiva: obiettivo comprensione 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

comprensione 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

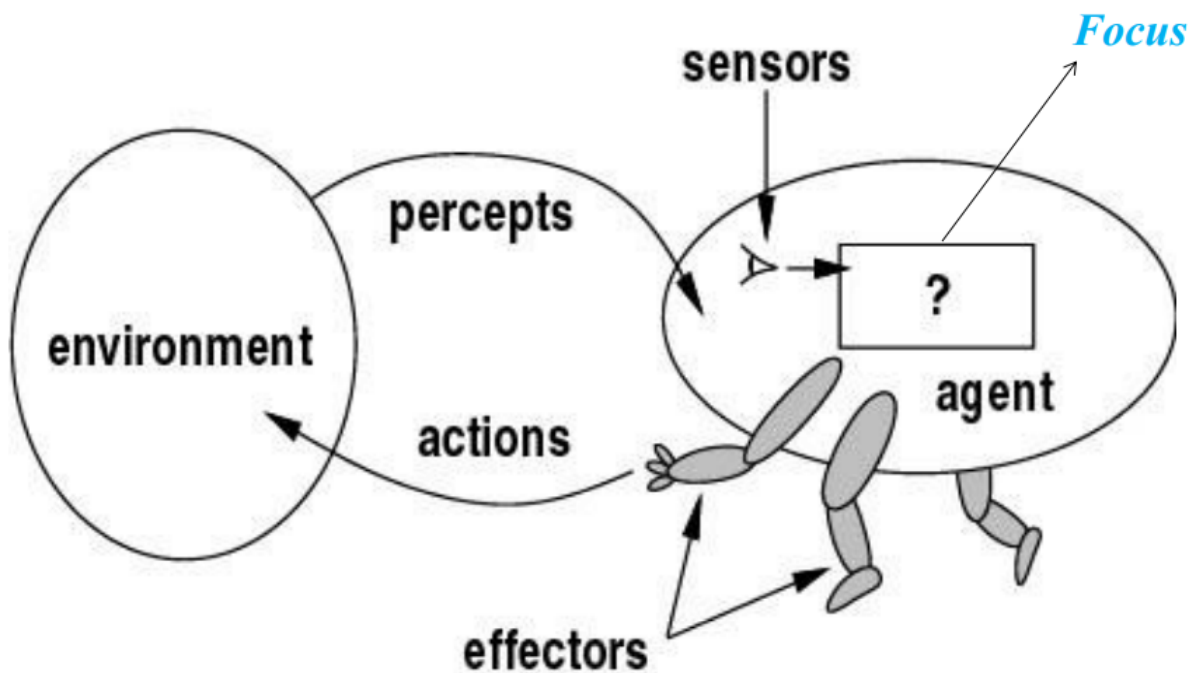
L'intelligenza è vista come l'avere diverse capacità, durante il progresso nell'area di ricerca: buon senso, interazione con un ambiente, acquisizione di esperienza, comunicazione, ragionamento logico...

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**.

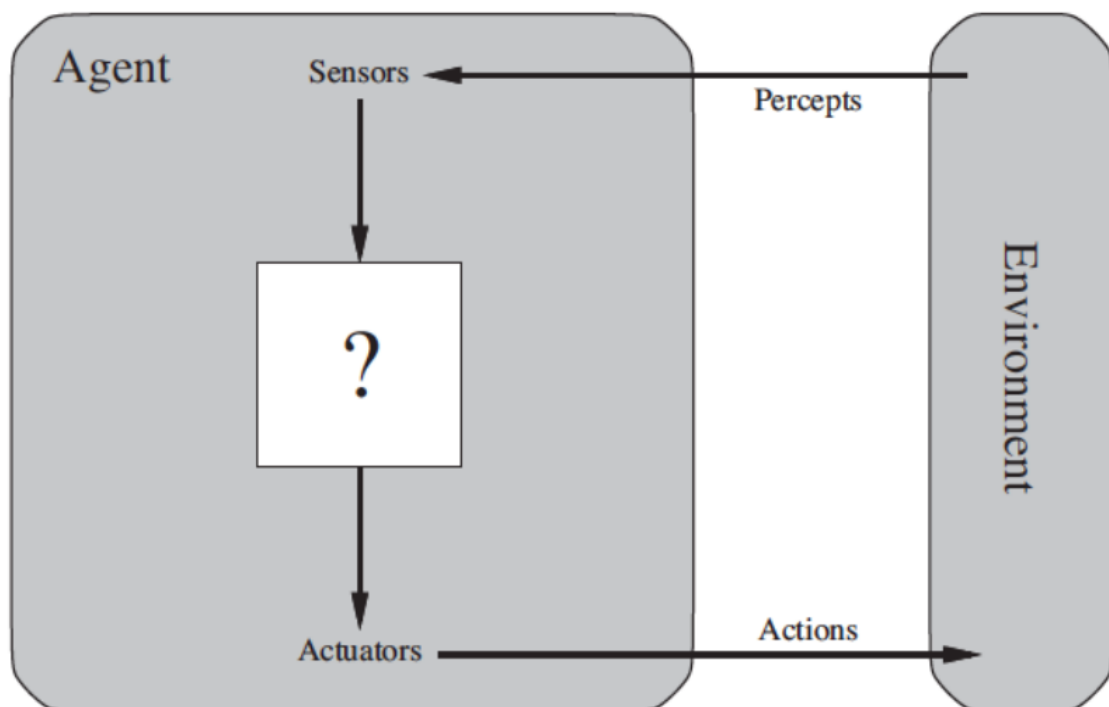
Funzione Agente Definisce l'azione da intraprendere per ogni sequenza percettiva e **descrive completamente l'agente**. Implementata da un **programma agente**.

$$\text{Sequenza Percettiva} \rightarrow^f \text{Azione}$$

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 onniscenza**).

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, **deve 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/zone 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 giocatore, 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**

Completamente 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 \neq 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

Agente = Architettura + Programma

$Ag: P \rightarrow Az$

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 $10^{120} \gg 10^{80}$ numero di atomi nell'universo osservabile.

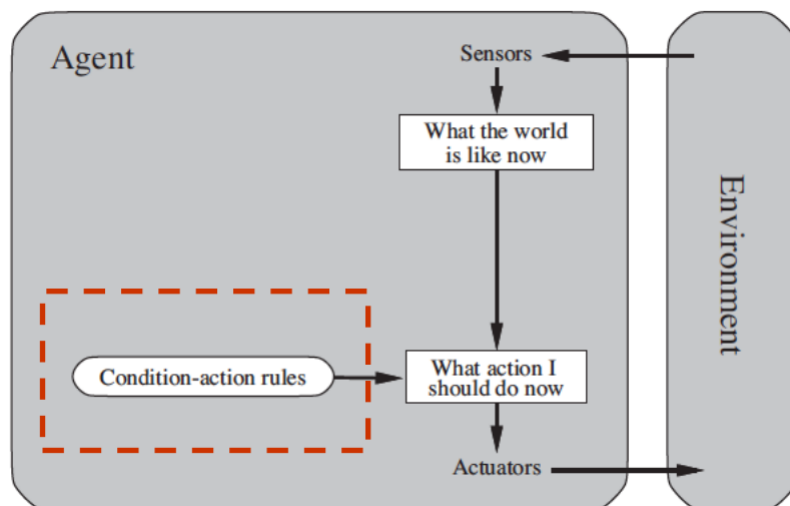
Difficile da costruire

Nessuna autonomia

Difficile da aggiornare, apprendimento complesso.

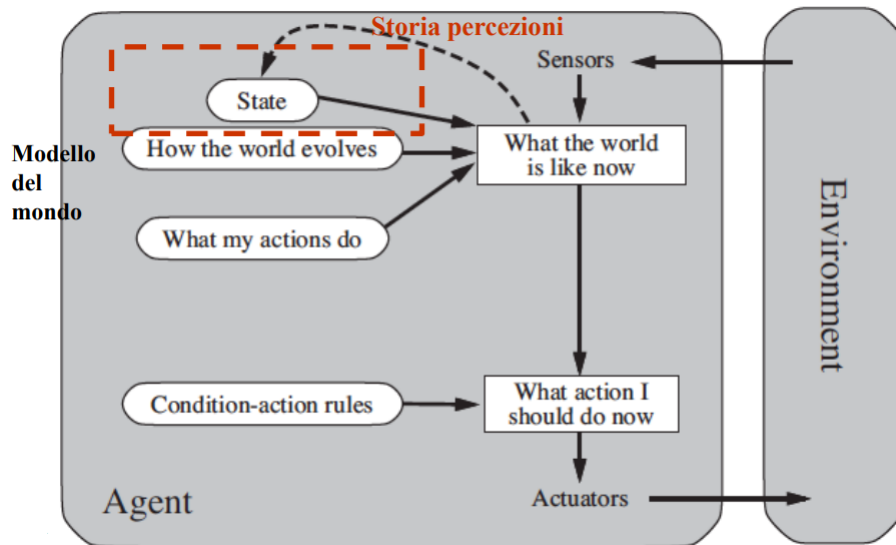
Con le IA vogliamo realizzare **automi razionali con un programma compatto**.

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

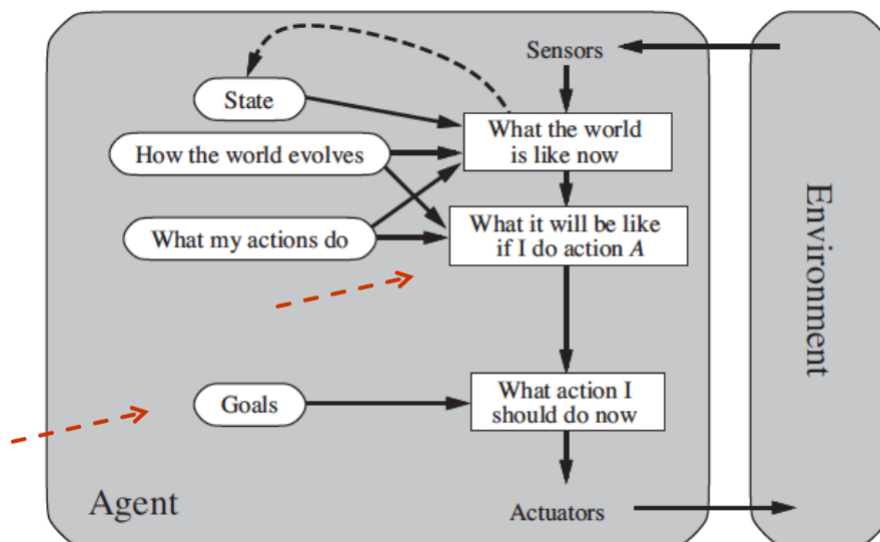


```

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)



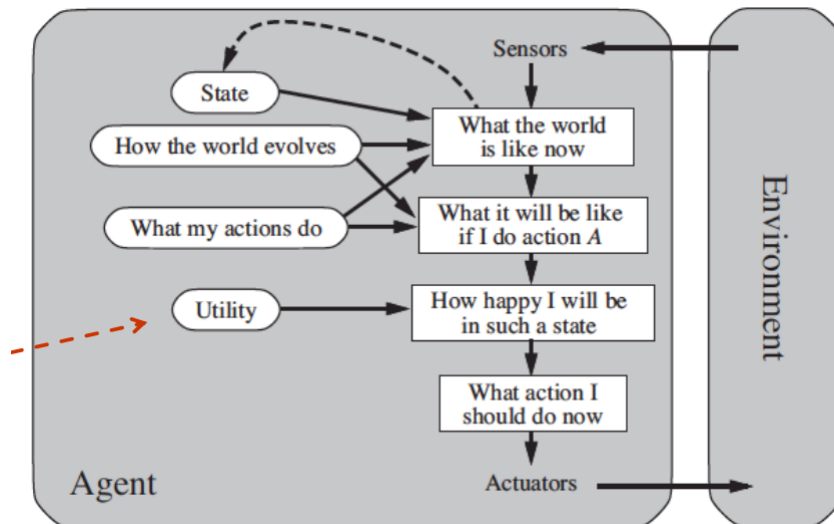
Sono guidati da un obiettivo nella scelta che intraprendono, è stato fornito un goal esplicito: per esempio una città da raggiungere.

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.

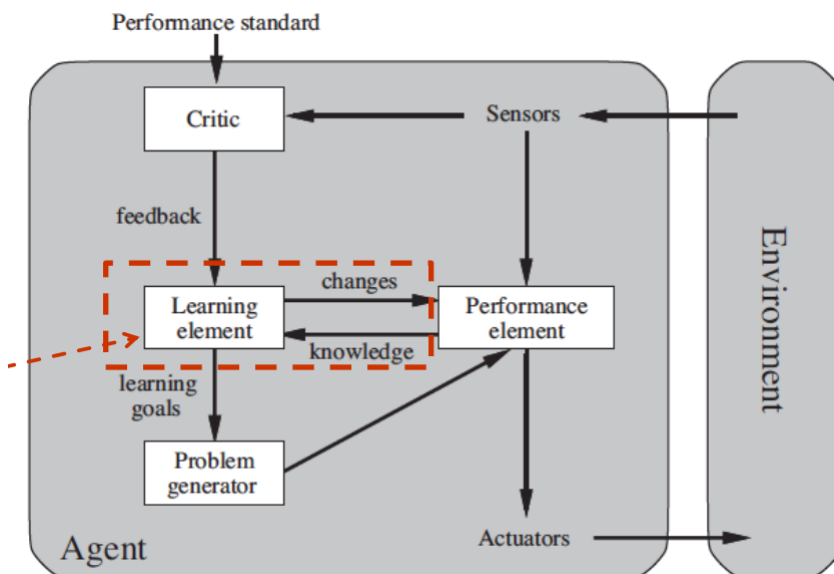
Agenti con valutazione di utilità



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

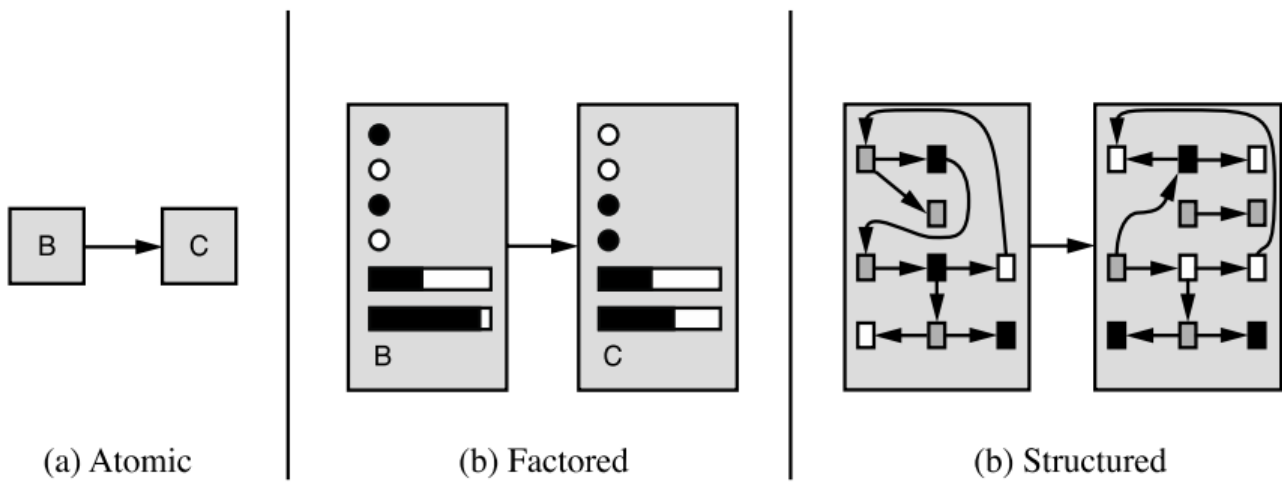
Elemento esecutivo: il programma agente

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)

Rappresntazione 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 (problemi 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 {true, 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?

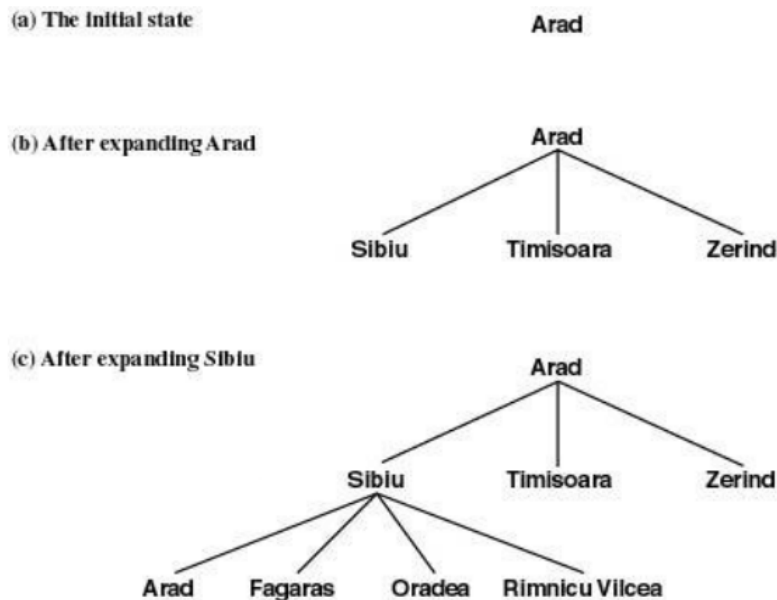
$$\text{Costo Totale} = \text{Costo della Ricerca} + \text{Costo del Cammino Soluzione}$$

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.stato

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 e' inizializzato nel costruttore
    if problem.goal_test(node.state):
        return node.solution(explored_set = explored)
    frontier = FIFOQueue() # la frontiera e' 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 e' 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-temporale 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 profondita' ricorsiva"""
    #controlla se lo stato del nodo e' 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

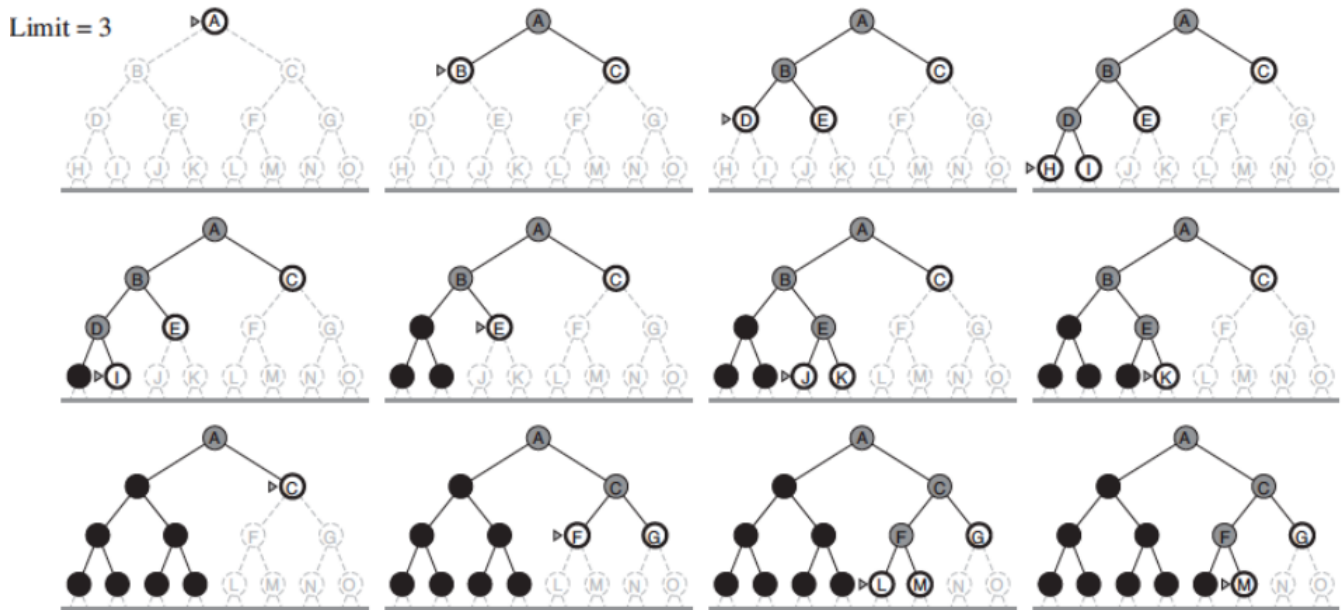
Completo se $d < l$

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.

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

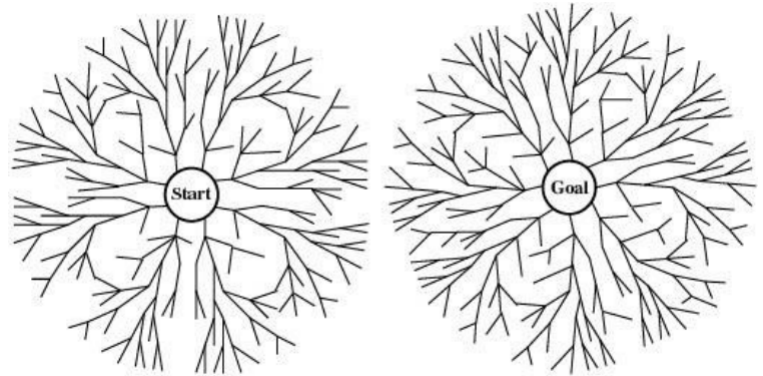
Complessità in tempo: $O(b^{d/2}) = O(\sqrt{b^d})$

Test intersezione in tempo costante, esempio: hash table

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

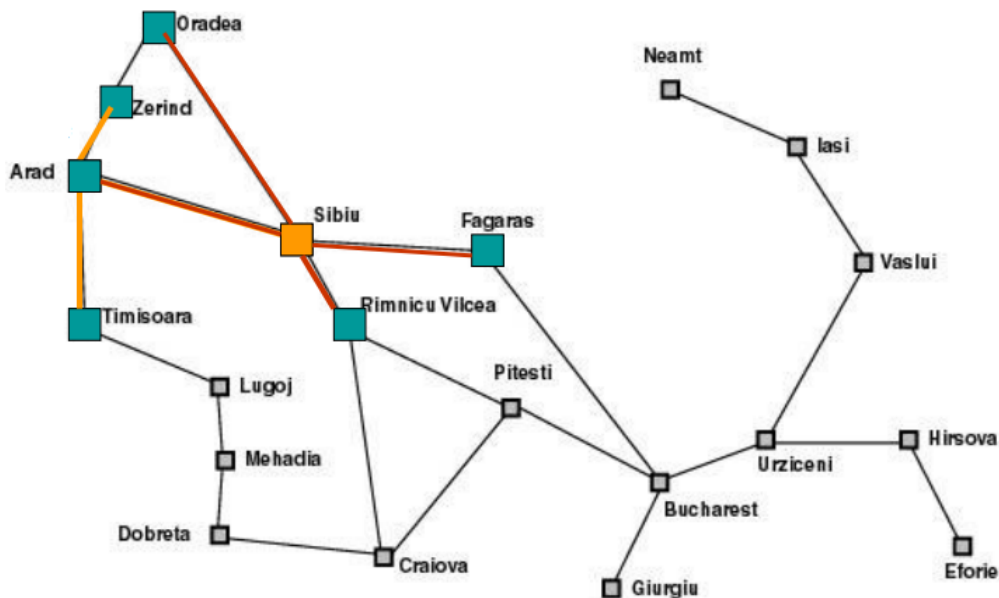
Almeno tutti i nodi in una direzione in memoria, esempio: usando BF

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

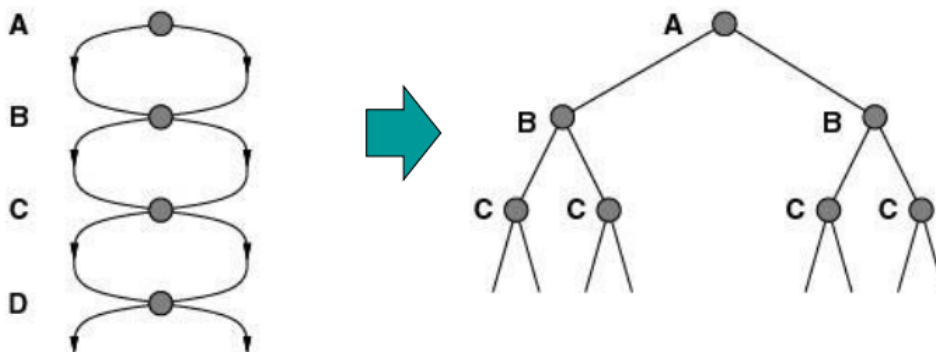


2.4 Problematiche

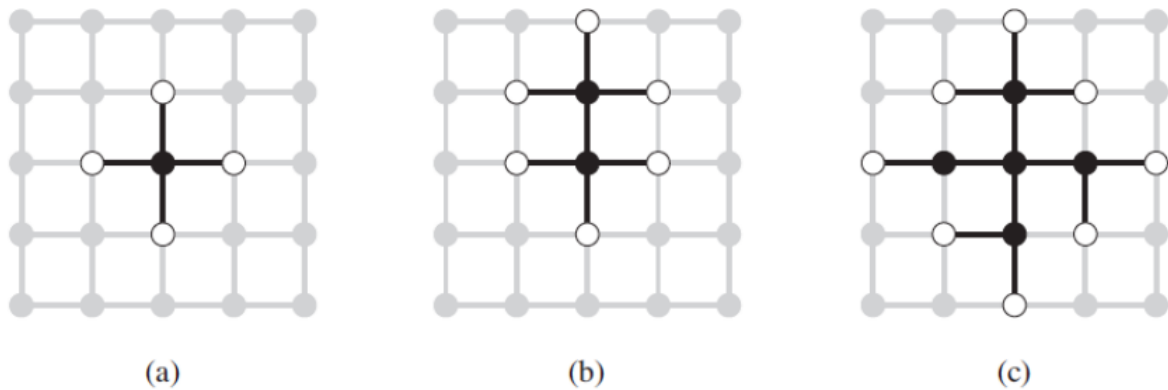
Cammini Ciclici I cammini ciclici potenzialmente rendono gli alberi di ricerca infiniti, anche se con stati finiti.



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 successivi.
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-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 priorit  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 priorit 
    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 priorit 
  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 priorit  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 priorit 
      else if figlio.Stato in frontiera con Costo-cammino piu alto then
        sostituisci quel nodo frontiera con figlio

```

Analisi Ottimalit  e completezza garantite purch  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 pi  alta ma poi abbia un path a costo pi  basso.

Complessit : $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.

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

2.6 Confronto delle Strategie (albero)

Criterio	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 profondit  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 ($n.\text{Stato}$). 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.

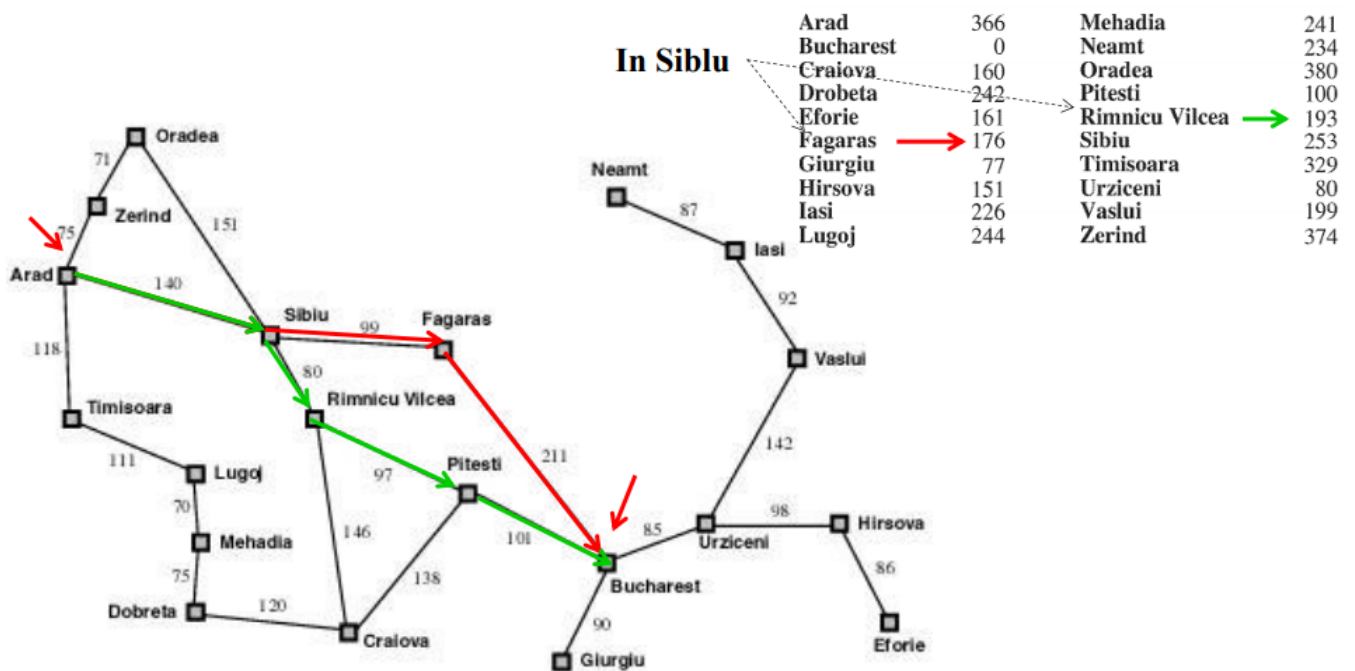
Esempio di euristica h Per procedere preferibilmente verso il percorso migliore, seguendo il "problem-specific information", nel problema del percorso più breve da città a città posso includere nel mio algoritmo le distanze in linea d'aria, oppure il vantaggio in pezzi nella dama o negli scacchi.

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

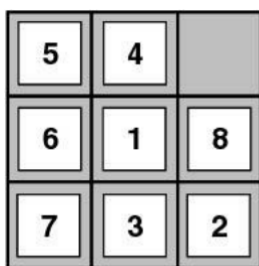
$$f(n) = g(n) + h(n)$$

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:

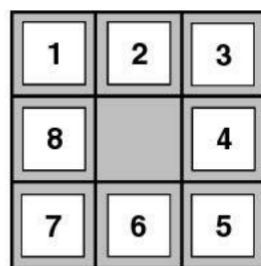
se $h(n) = 0$, cioè $f(n) = g(n)$, si ha **Ricerca Uniforme (UC)**

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

Esempio Il gioco dell'otto



Start State



Goal State

$$f(n) = \#mosseFatte + \#caselleFuoriPosto$$

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

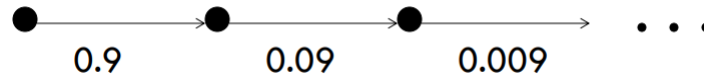
Dopo $\leftarrow, \downarrow, \uparrow, \rightarrow$ si ha $f = 4 + 7$: stesso stato ma g è cambiata.

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

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 $[n_0 n_1 n_2 \dots n' \dots n_k = \text{goal}]$ 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.

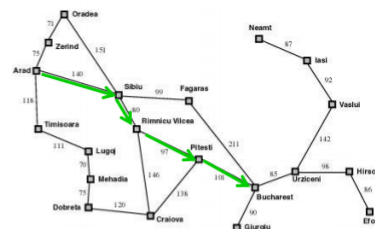
Definizione Algoritmo A*: un algoritmo A in cui h è una funzione euristica ammissibile.

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

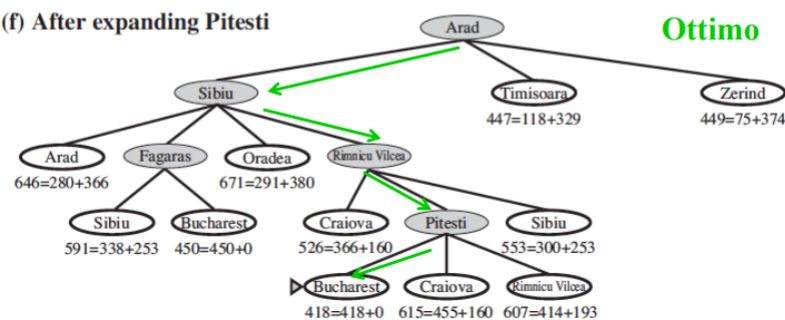
Corollario BF con passi a costo costante e UC sono **ottimali** ($h(n) = 0$)

Itinerario con A*

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



(f) After expanding Pitesti



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

h sotto o sopra 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: $\forall n | h(b) \leq c(n, a, n') + h(n')$ 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 piu 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 [$h = 0, f = C^*$]

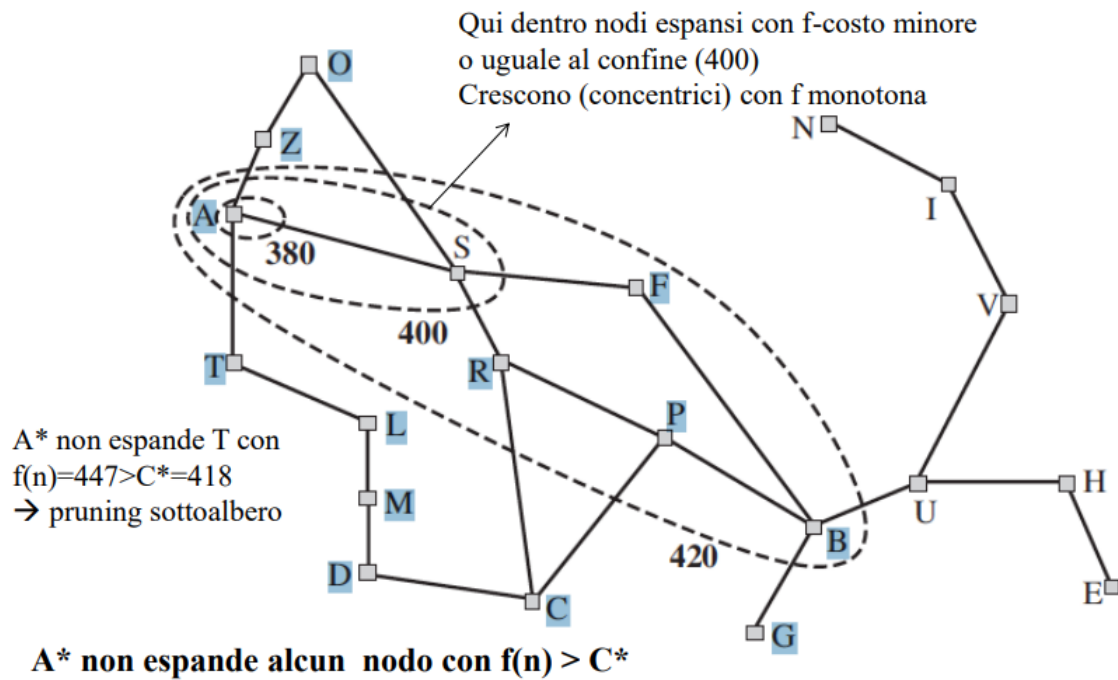
3.2.3 Perché A^* è vantaggioso?

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 necessità 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**.

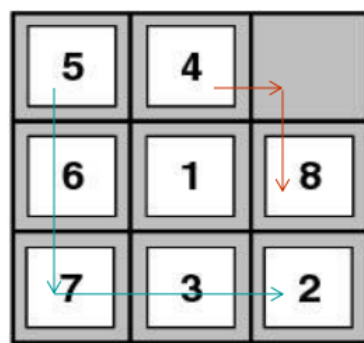
3.3.2 Confronto di euristiche ammissibili

Esempio Due euristiche ammissibili per il gioco dell'otto

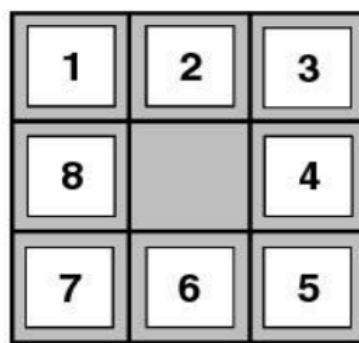
h_1 conta il numero di caselle fuori posto

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 .



Start State



Goal State

$$h_1 = 7$$

$$h_2 = 4 + 2 + 2 + 2 + 2 + 0 + 3 + 3 = 18$$

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 \rightarrow numero di caselle fuori posto

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

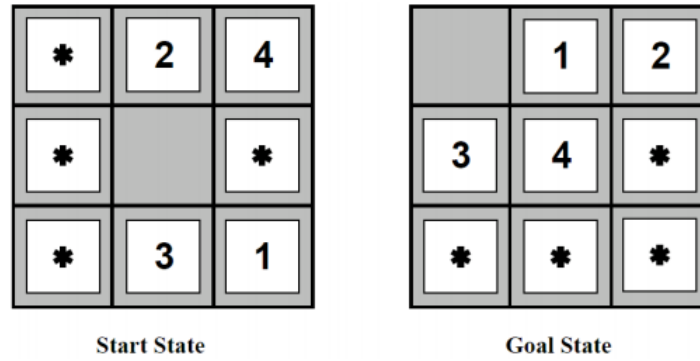
3.4.2 Massimizzazione di euristiche

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 spostato 1-2-3-4 spostato 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**: database 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 \rightarrow costo 14.

Combinazione di euristiche

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

$$h(n) = c_1x_1(n) + c_2x_2(n) + \dots + c_kx_k(n)$$

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

Scacchi $h(n) = c_1vantaggioPezzi + c_2pezziAttaccante + c_3regina + \dots$

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'iterazione 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**, quindi, **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**

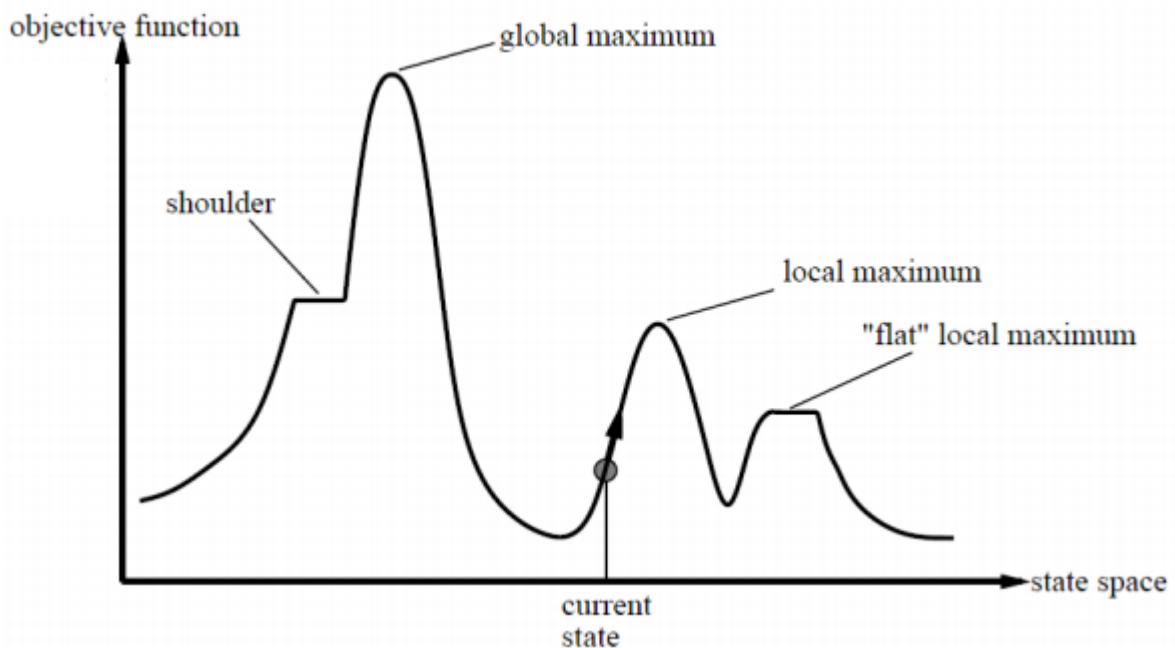
Stato migliore secondo una funzione obiettivo

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.

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 rilassamento delle assunzioni di base: ambiente multi agente e rappresentazioni degli stati più complesse. Ci occuperemo di **specializzazioni** del paradigma: giochi con avversario. I piani quindi 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 rappresentato in linguaggio espressivo. 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 risultato costante), a informazione perfetta (tutti i gioc conoscono stato gioco)

Ambiente multi-agente competitivo: presenza avversario rende ambiente strategico, più difficile rispetto a visto fin'ora

Complessità e vincoli di tempo reale: mossa migliore nel tempo disponibile

⇒ giochi più simili a prob reali

6.2 Ciclo *pianifica-agisci-percepisci*

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

Una volta decisa la mossa

Si esegue la mossa

...

6.3 Giochi come problemi di ricerca

Stati: config gioco

Player(s) a chi tocca muovere nello stato s

Stato iniziale: config iniziale gioco

Actions(s) mosse legali in s

Result(s, a) stato risultatne da una mossa

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

Utility(s, p) utilità (o pay-off) valore numerico degli stati terminali del gioco per p

Es: 1, -1, 0, conteggio punti. . .

6.4 Algoritmo Min-Max

Valuto stati terminali in base a punteggio/vittoria e valuto stati preterminali valutandoli in base a se mi portano a vittoria o meno a ritroso. Valuto stati intermedi con valore minimo tra gli stati risultanti.

Immagine tic-tac-toe

Valore minmax $\text{minimax}(s) =$

se Terminal-Test(s) , allora Utility(s, MAX)

se Player(s) = MAX , allora $\max_{a \in \text{Action}(s)} \text{Minimax}(\text{Result}(s, a))$

se Player(s) = MIN , allora $\min_{a \in \text{Action}(s)} \text{Minimax}(\text{Result}(s, a))$

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

Algoritmo ricorsivo . . .

6.5 Algoritmo Min-Max Euristico (con orizzonte)

Casi complessi in cui esplorare stati è troppo costoso (es. scacchi): occorre **funzione di valutazione euristica**.

Strategia: guardare avanti d livelli. . .

6.6 Funzione di Valutazione

Stima dell'utilità attesa a partire da una certa posizione. La qualità della funzione è determinante.

deve esser consistente con l'utilità se applicata a steati terminali del gioco (indurre lo stesso ordinamento)

dev'essere efficiente da calcolare

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

. . .

Esempio scacchi

Problemi Noti

Stati Non Quiescenti: esplorazione fino ad un livello può mostrare situaz. molto vantaggiosa, ma a mossa successiva es. regina mangiata dalla torre. Si possono riconoscere-

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

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

6.6.1 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

6.6.2 Algoritmo Alfa-Beta

Algoritmo ...

Ordinamento delle mosse

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

6.7 Problemi di Soddisfacimento di Vincoli

CSP caso particolare dei problemi di ricerca, grazie alla struttura si cerca meglio la soluzione.
Classe ampia, esistono euristiche generali da applicare che consentono risoluzioni di dimensioni significative.

Tre componenti ...

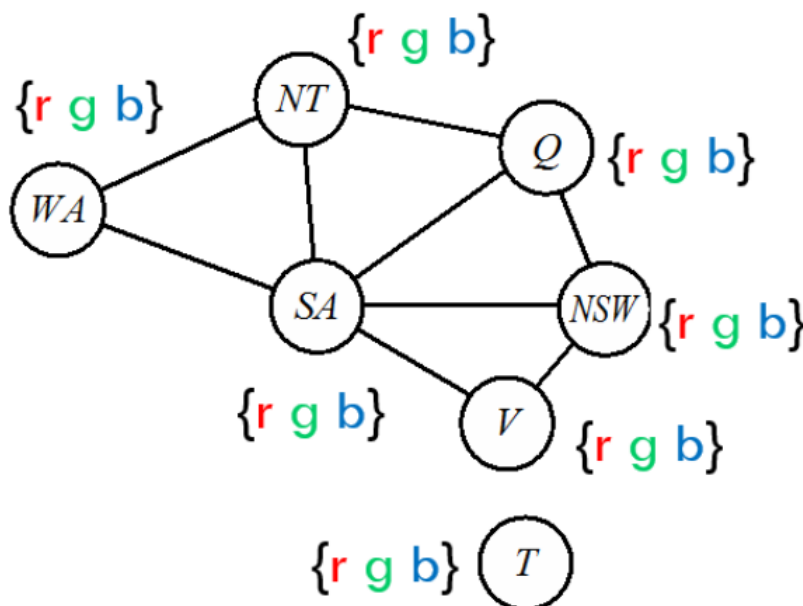
Backtracking ricorsivo per CSP Soluzione: trovare assegnamento per le variabili con valori dal dominio per arrivare ad assegnamento completo (tutte le variabili con valori) e consistente (soddisfa vincoli).
Scelta dei valori: scegliere il valore meno vincolante, quello che esclude meno valori possibili 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 un valore vado a vedere variabili direttamente collegate nel grafo dei vincoli

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

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



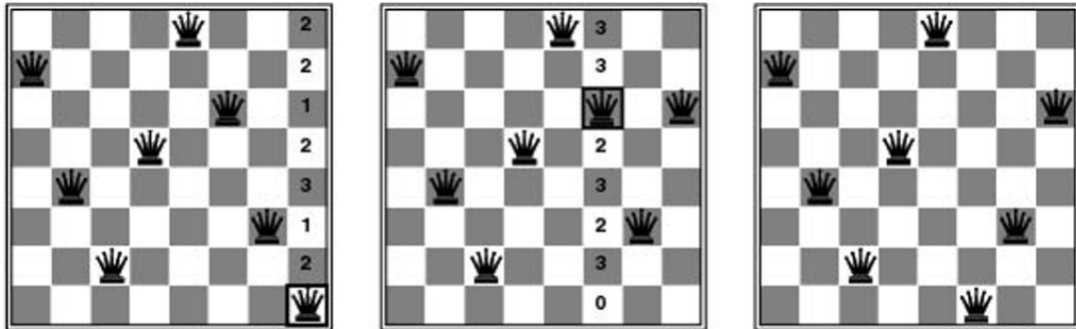
Backtracking Cronologico Suppongo di avere $\{Q=R, NSW=G, V=B, T=R\}$. Cerco di assegnare SA, il fallimento genera un backtracking cronologico: si provano tutti i valori alternativi per l'ultima variabile, T, continuando a fallire... Questo perché non è la T responsabile del fallimento ma le altre variabili. 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 regie 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)

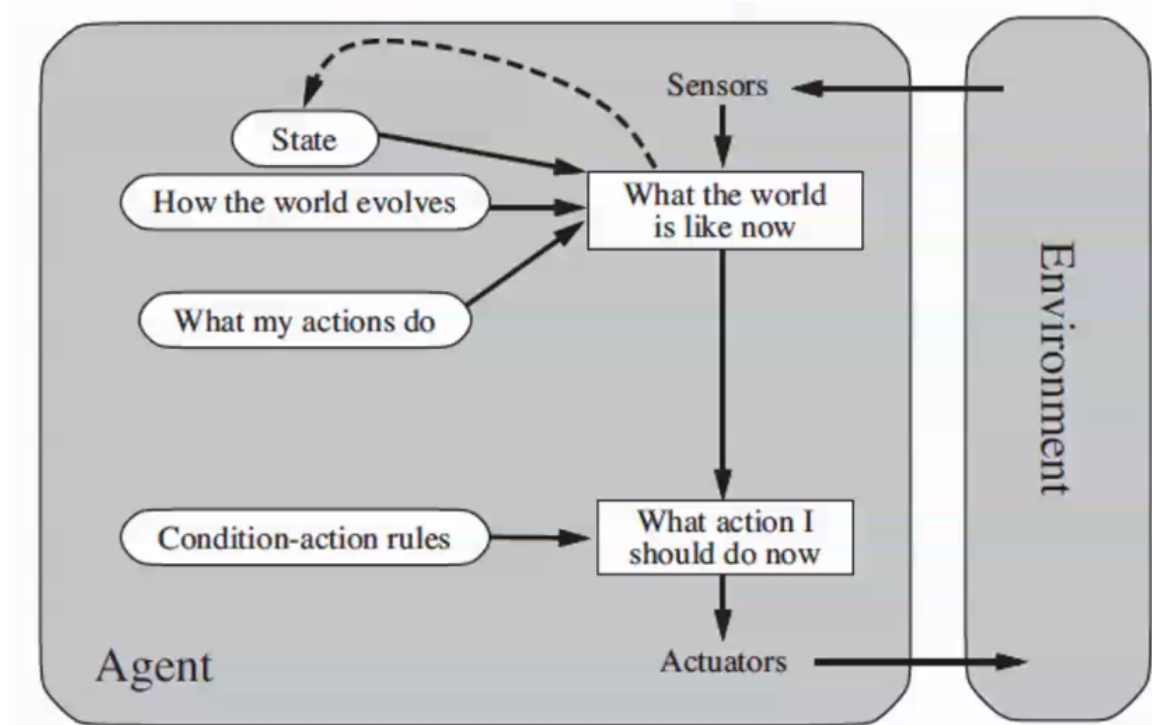
Prossimamente: **sistemi basati su conoscenza**. Conoscenza implica capacità inferenziali, con rappresentazione stato ancora più ricco con linguaggio rappresentazione conoscenza. L'inferenza è anch'essa un problema di ricerca in uno spazio di stati.

Capitolo 7

Agenti Basati su Conoscenza

Come costruire agenti dotati di capacità inferenziali.

Cerchiamo di migliorare capacità di ragionamento, dotando di rappresentazione del mondo più complessi non descrivibili semplicemente. Agenti basati su conoscenza, con all'interno una base di conoscenza KB con conoscenza espressa in forma esplicita e dichiarativa attraverso un linguaggio.



La maggiorparte prob I.A. sono "knowledge-intensive". Mondo tipicamente complesso: serve rappresentazione parziale e incompleta (astrazione) del mondo utile agli scopi dell'agente.

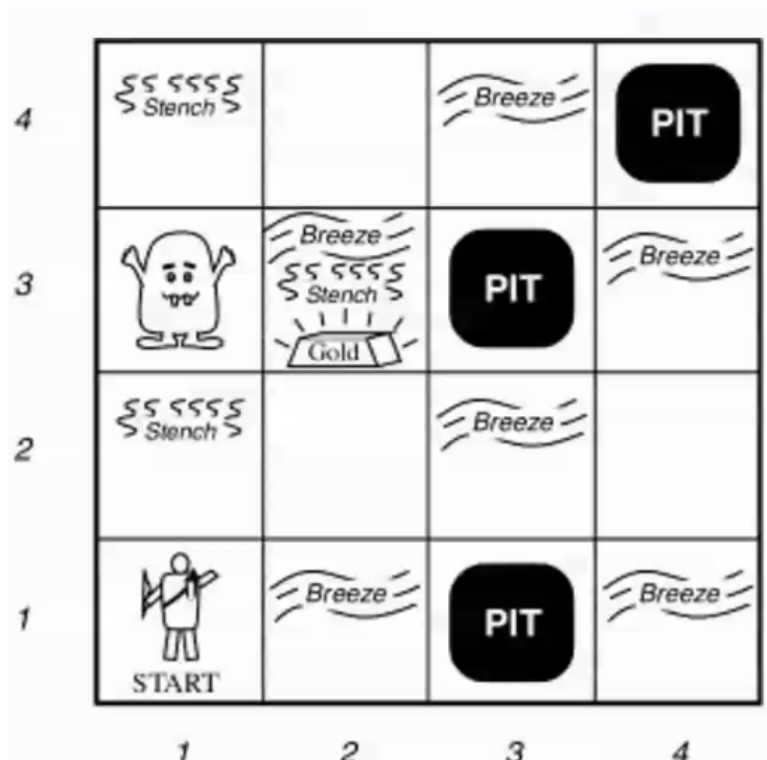
Per ambienti parzialmente osservabili e complessi ci servono linguaggi di rappresentazione della conoscenza più espressivi e capacità inferenziali.

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

Approccio dichiarativo vs procedurale tipicamente conoscenza in KB in forma dichiarativa. Alternativa è codificare conoscenza in maniera procedurale con programma che implementa processo decisionale. Un agente KB (con conoscenza dichiarativa) è più flessibile: perché più semplice ricevere conoscenza in maniera incrementale e modificare il comportamento con l'esperienza.

7.1 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] e 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 sbatte in un muro;
 - **urlo** se il Wumpus viene ucciso.
 - L'agente non percepisce la sua locazione.
- **Azioni:**
 - **avanti**
 - **a destra** di 90°, **a sinistra** di 90°
 - **afferra** un oggetto
 - **scaglia la freccia** (solo una)
 - **Esce**
- **Ambienti generati a caso ([1,1] safe)**

Uno scenario ...

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

Tell

Ask

Retract

Enunciati in KG rappresentano conoscenza agente...

Problema Il problema da risolvere: data conoscenza KB che contiene... vorrei sapere se un certo fatto α è vero di conseguenza, cioè se KB *soddisfa* α

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

Una base di dati invece solo base di dati e solo recupero.

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

Trade-Off Fondamentale della Rappresentazione della Conoscenza Trovare giusto compromesso tra linguaggio rappresentazione espressivo e la complessità del meccanismo inferenziale.

Più linguaggio è espressivo meno è efficiente il meccanismo inferenziale. Obiettivi in contrasto, bisogna mediare e trovare **compromesso**.

Formalismi per la R.C.

Sintassi

Semantica

Meccanismo inferenziale

Logica come linguaggio Qual è la complessità computazionale del problema KB *soddisfa* α 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.

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

7.2 Calcolo Proposizionale

Sintassi

Algoritmo TT-entails?

KB *soddisfa* α ?

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

...

Algoritmi per la soddisfacibilità (SAT) Usano KB come insieme di clausole.

Algoritmo DPLL 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

Euristica dei simboli (o letterali) puri

Euristica delle clausole unitarie

Simboli puri Simbolo con stesso segno in tutte le clausole. Es $\{A, \text{not}B\}$, $\{\text{not}B, \text{not}C\}$, $\{C, A\}$ A puro e B anche. Nel determinare un simbolo se è puro possiamo trascurare occorrenze simbolo in clausole già rese vere. Se simbolo è puro può essere assegnato a T se positivo, F se negativo senza eliminare modelli.

Clausole unitarie