Introduzione all'Intelligenza Artificiale

Federico Matteoni

A.A. 2019/20

Indice

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 comportamente 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

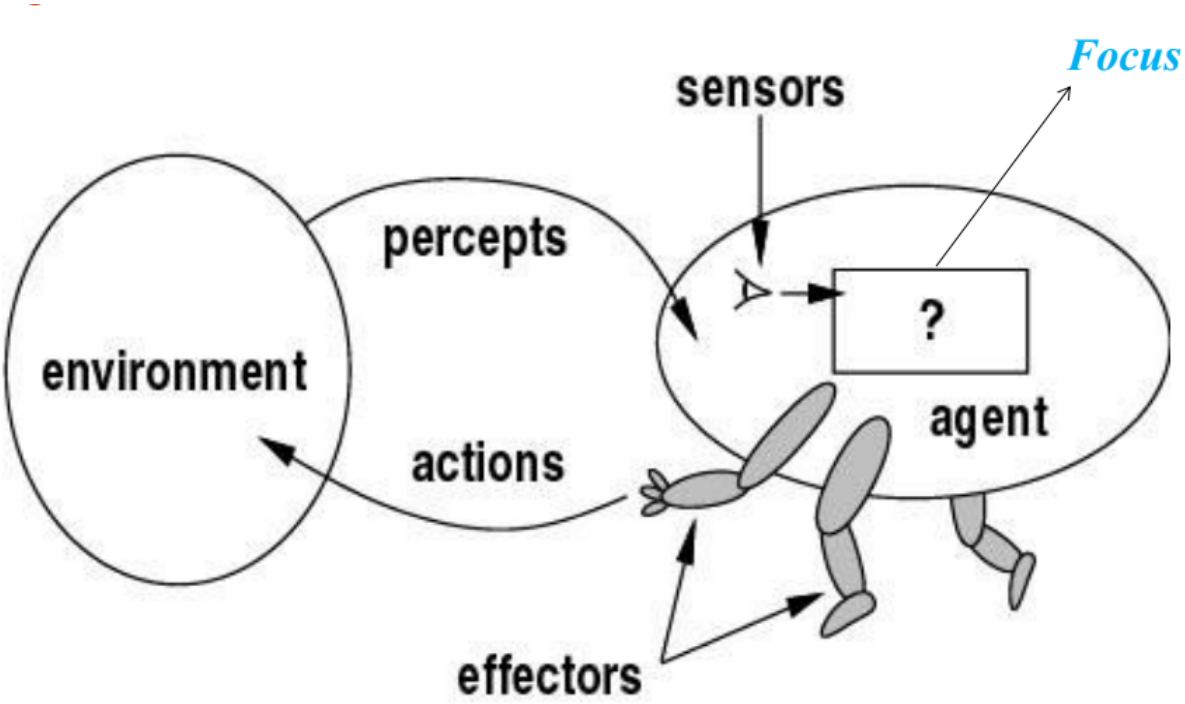
1.1 Intelligenza

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**.

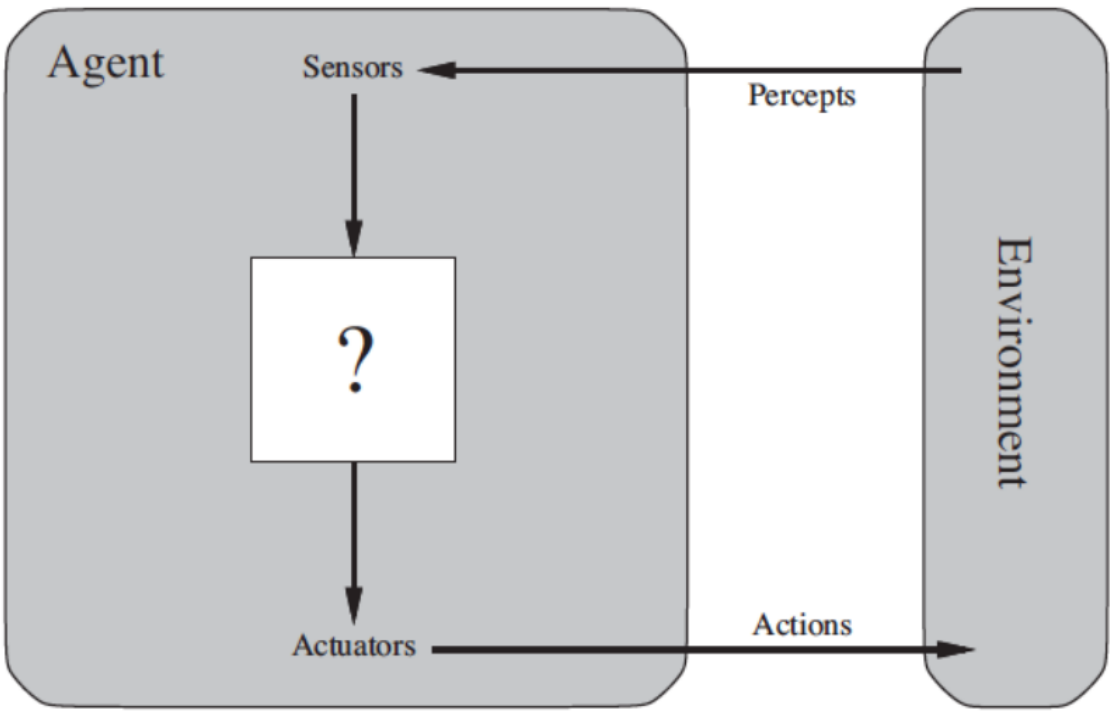
Funzione Agente Definisce l'azione da intraprendere per ogni sequenza percettiva e **descrive completamente l'agente**. Implementata da un **programma agente**.

$$\textbf{Sequenza Percettiva} \longrightarrow^{f} \textbf{Azione}$$

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 succesive.

- **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 si. 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 $\longrightarrow$ Az

L'**Ag**ente associa **Az**ioni alle **P**ercezioni. 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.

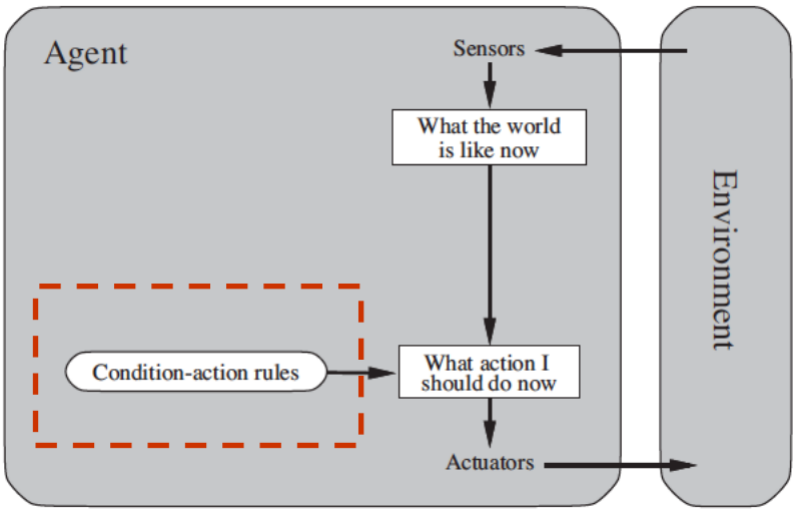
Difficile da costruire

Nessuna autonomia

Difficile da aggiornare, apprendimento complesso.

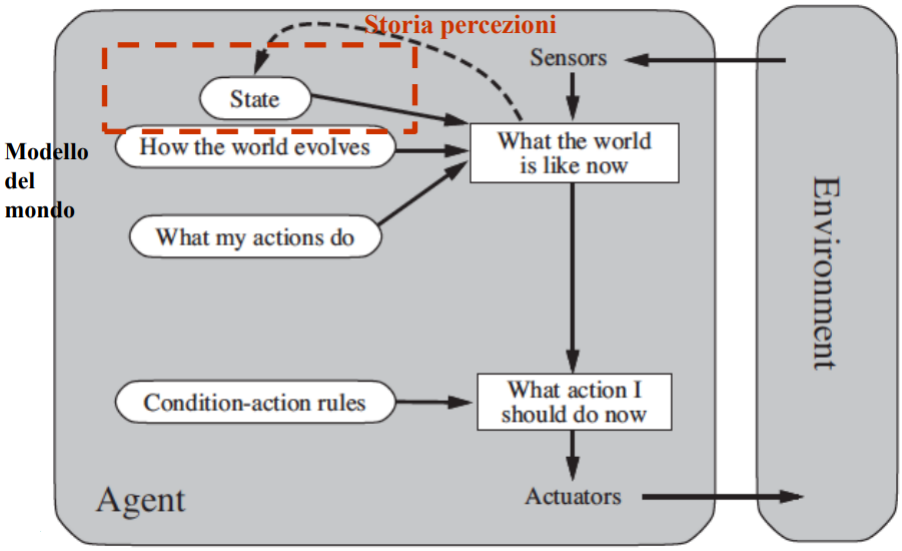
Con le IA vogliamo realizzare **automi razionali con un programma *compatto***.

Agente Reattivo Semplice



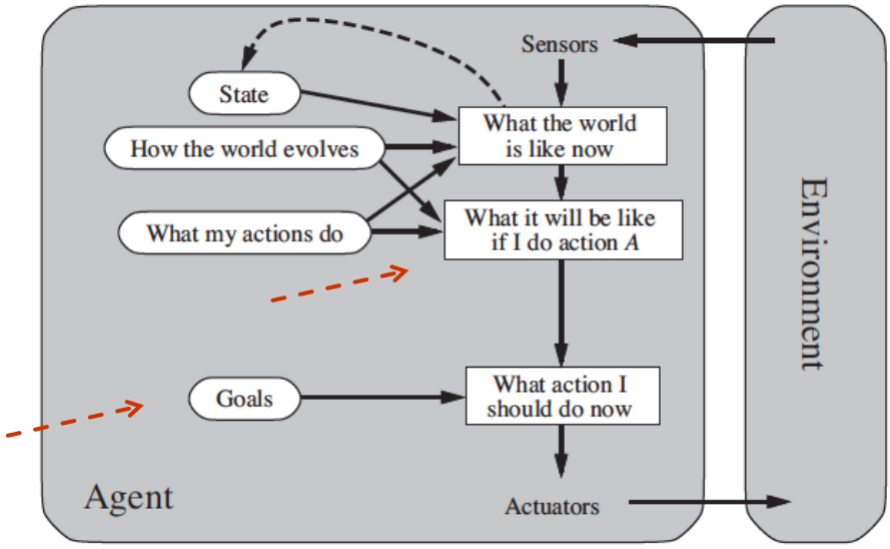

```
function Agente-Reattivo-Semplice(percezione) returns azione
        persistent: regole #insieme di regole condizione-azione (if-then)
        stato <- Interpreta-Input(percezione)
        regola <- Regola-Corrispondente(stato, regole)
        azione <- regola.Azione
        return azione
```

Agenti basati su modello



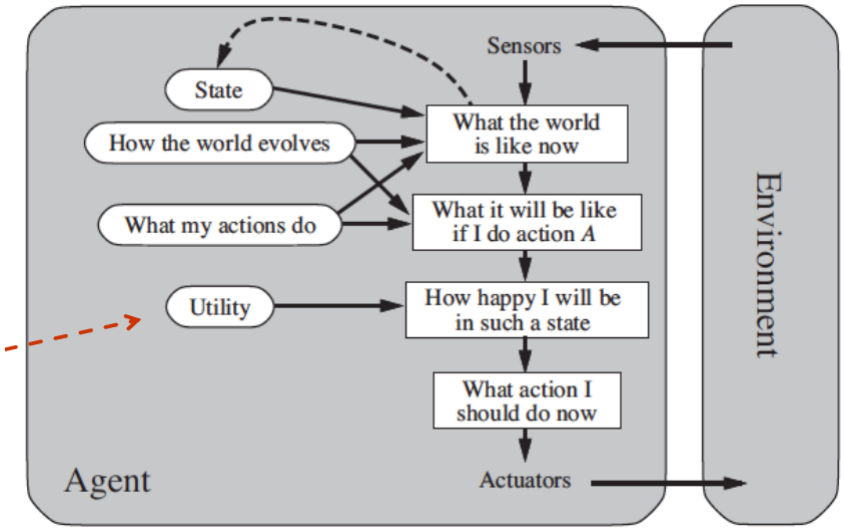

```
function Agente-Basato-su-Modello(percezione) returns azione
        persistent:     stato #descrizione dello stato corrente
                        modello #conoscenza del mondo
                        regole #insieme di regole condizione-azione
                        azione #azione piu recente
        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)



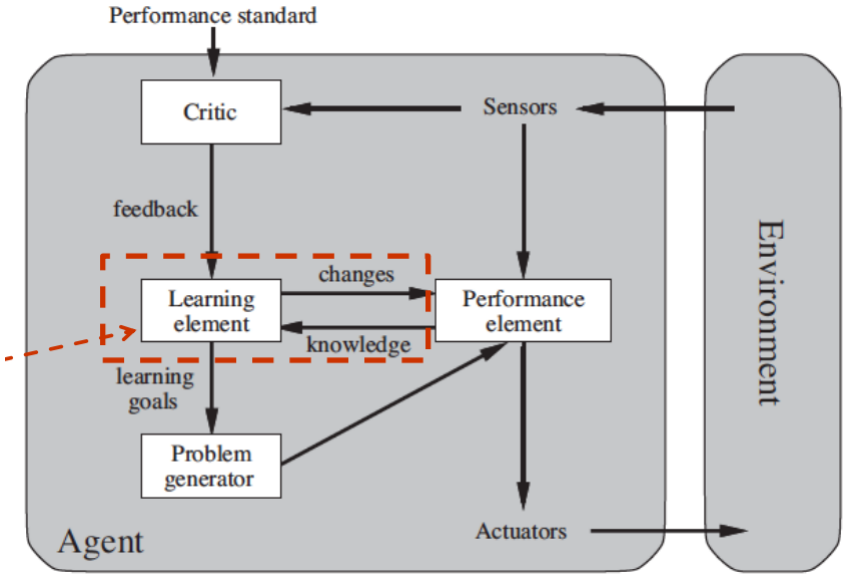
Sono **guidati da un obiettivo nella scelta che intraprendono**, è stato fornito un **goal esplicito**: per esempio una città da raggiungere.
A volte l'azione migliore dipende dall'obiettivo da raggiungere (*da che parte devo girare?*)
Devo **pianificare una sequenza di azioni** per raggiungere l'obiettivo. 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

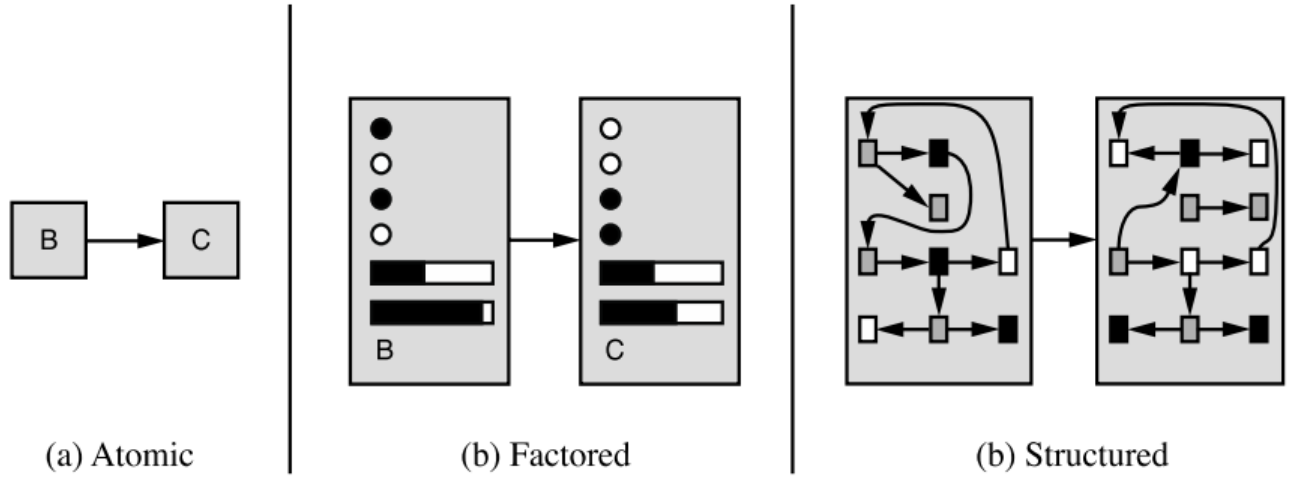
Elemento esecutivo: il programma agente

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)

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 ⇒ 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 × azione ⟶ stato
   `Risultato(s, a)`: `s'`, uno stato **successore**
4. **Test obiettivo**: un insieme di stati obiettivo
   `Goal-Test`: stato ⟶ {`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.

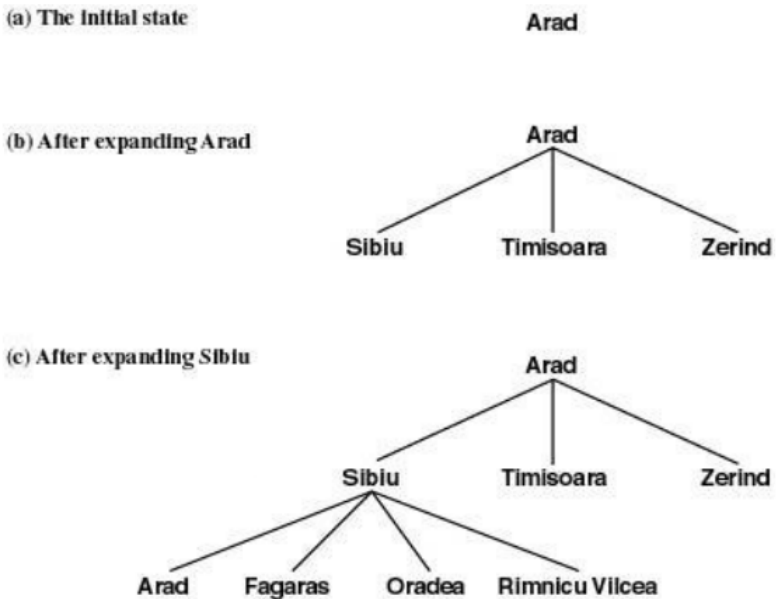
Misura delle prestazioni Trova una soluzione? Quanto costa trovarla? Quanto è efficiente la soluzione?

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

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) = padre.costo-cammino + 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 gia' 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 (**b**ranching)

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

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

Analisi:

Strategia **completa**

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

Complessità nel tempo (nodi generati)
$T(b, d) = b + b^2 + \ldots + b^d = O(b^d)$, 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·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 → 10 Esabyte

DF, d = 16 → 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
```

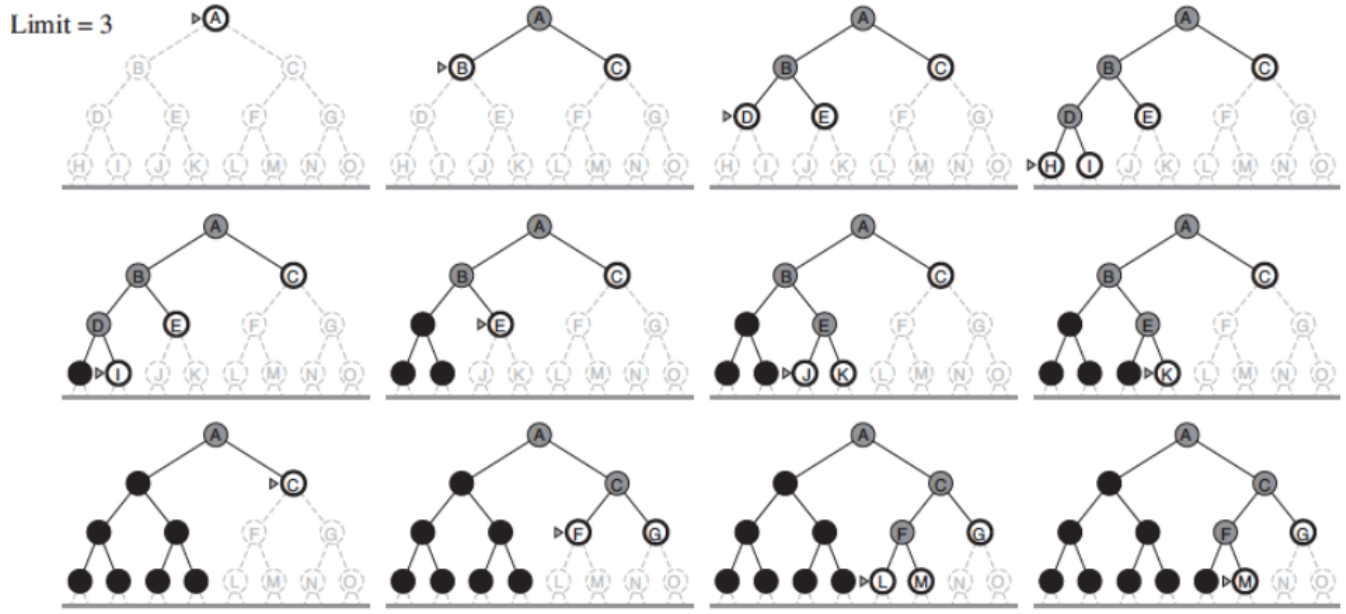
Python

```
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 poblemi 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·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 + ... + 3b^{d-2} + 2b^{d-1} + b^d
Complessità in tempo: O(b^d)
Complessità in spazio: O(b·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)

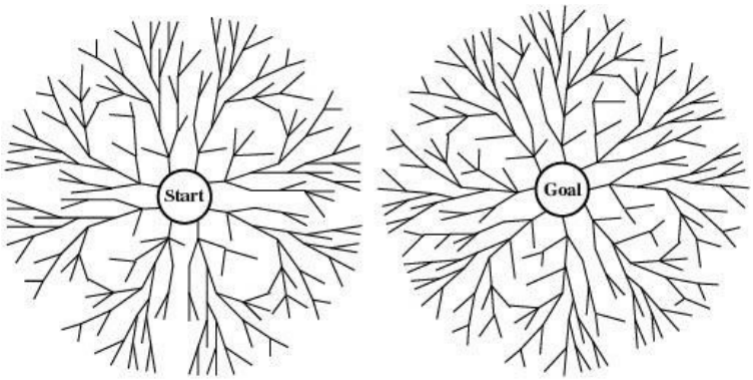
> abbiamo una **serie di dati da cui partire**

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

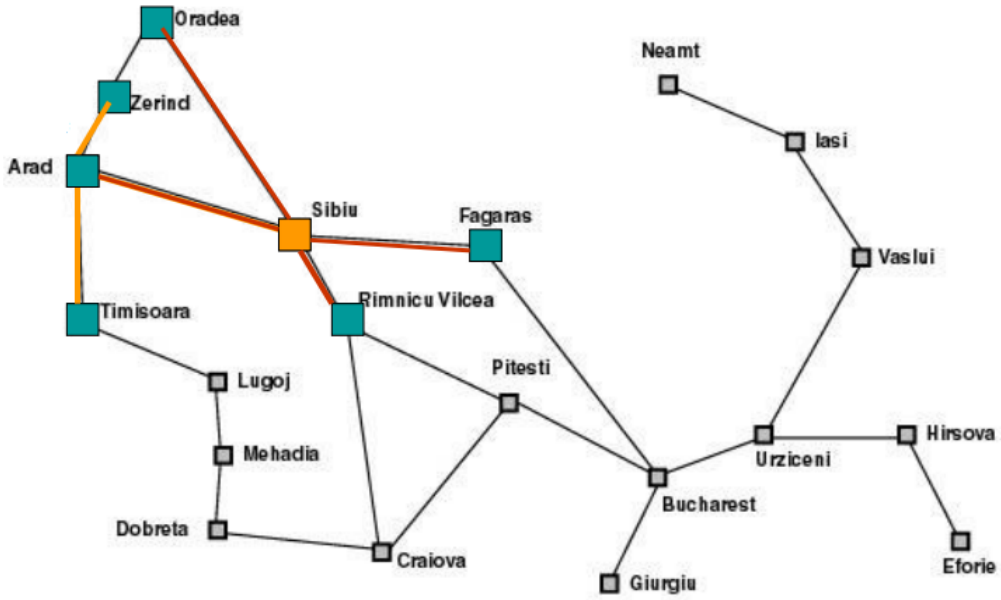
Complessità in spazio: $O(b^{d/2}) = O(\sqrt{b^d})$
Almeno tutti i nodi in una direzione in memoria, esempio: usando BF

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

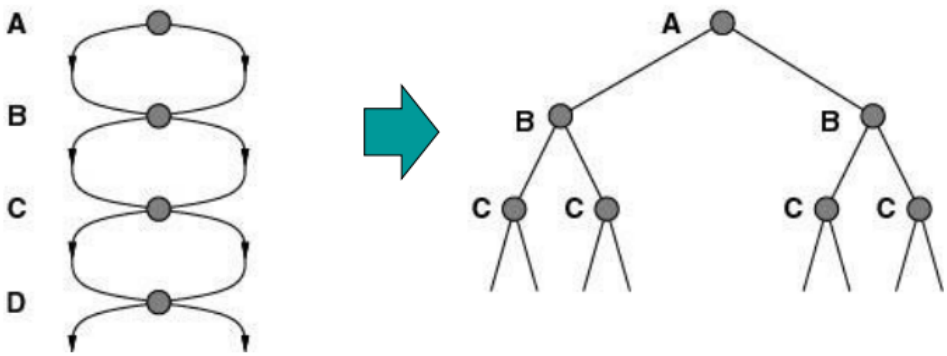


2.4 Problematiche

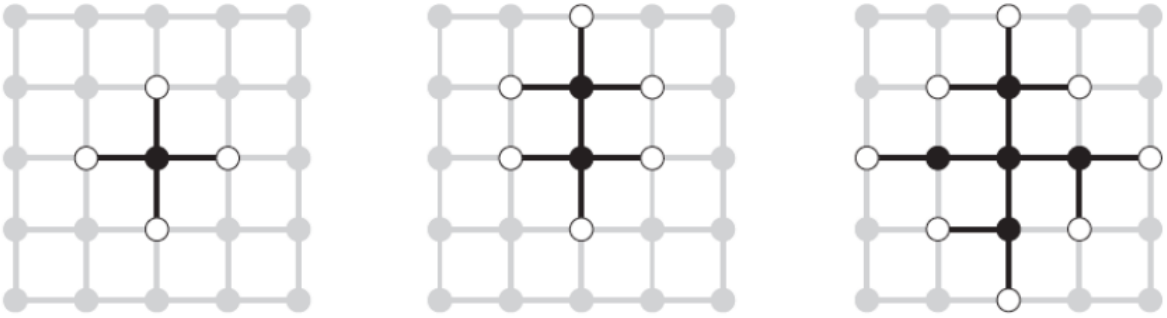
Cammini Ciclici I **cammini ciclici** potenzialmente rendono gli alberi di ricerca infiniti, anche se con stati finiti.



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.



(a) (b) (c)

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·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 priorita 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 priorita
                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 priorita
        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 garantite 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)

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 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**:

$$h : n \to R$$

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.

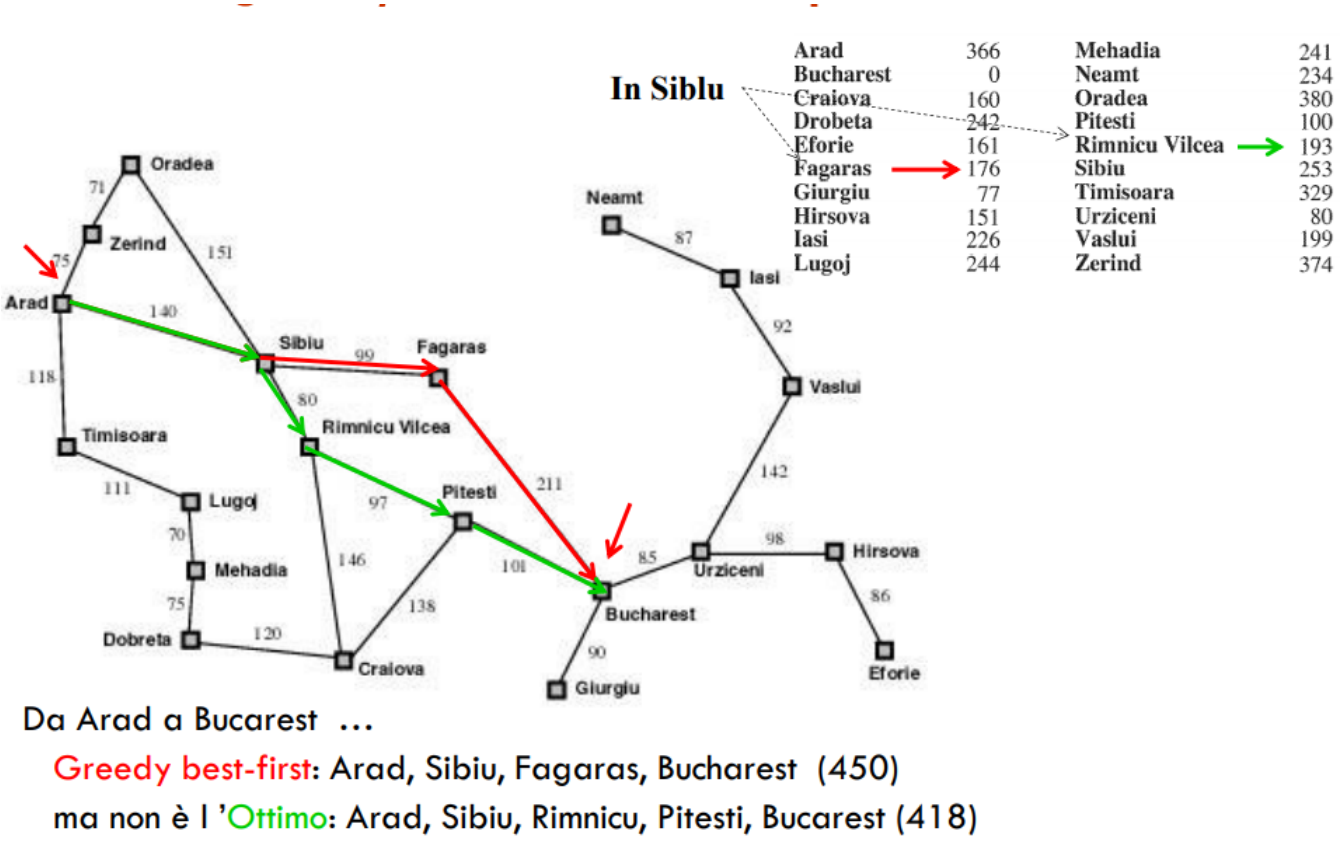
Esempio di euristica h Per procedere preferibilmente verso il percorso migliore, seguendo il "problem-specific information", nel problema del percorso più breve da città a città posso includere nel mio algoritmo le distanze in linea d'aria, oppure il vantaggio in pezzi nella dama o negli scacchi.

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)



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

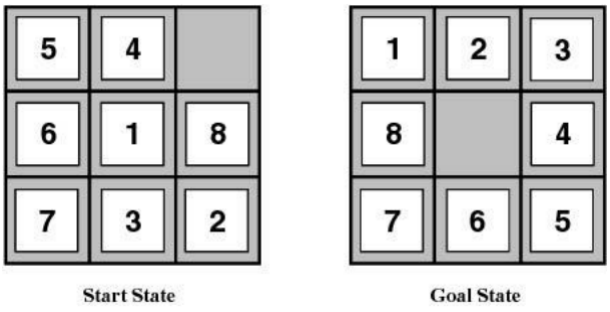
$$f(n) = g(n) + h(n)$$

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**



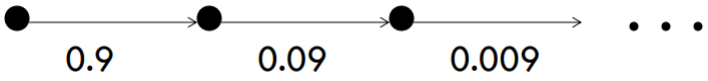
$f(n) =$ #mosseFatte + #caselleFuoriPosto

$f(start) = 0 + 7$
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 $[n_0\ n_1\ n_2 \ldots\ n'\ \ldots\ n_k = goal]$ 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

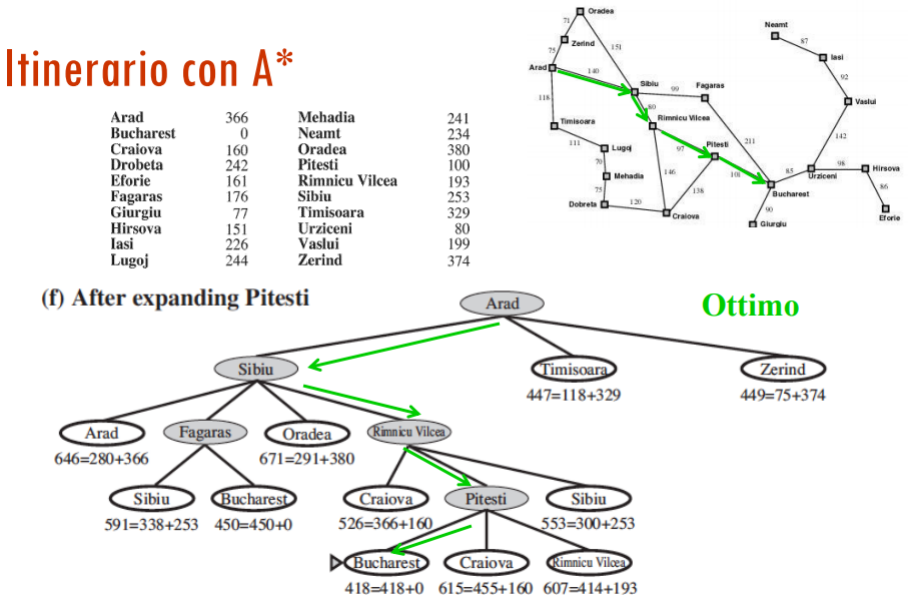
Normalmente $g(n) \geq g^*(n)$ (costo del cammino $\geq$ cammino minimo) e $h(n)$ è una **stima** di $h^*(n)$: si può sotto o sovrastimare la distanza dalla soluzione.

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 algortimo 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 ablero, l'**uso di un'euristica ammissibile è sufficiente a garantire l'ammissibilità ⇒ 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 $n \to n'$ 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 ⇒ 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 ⇒ 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 $[h = 0, f = C^*]$

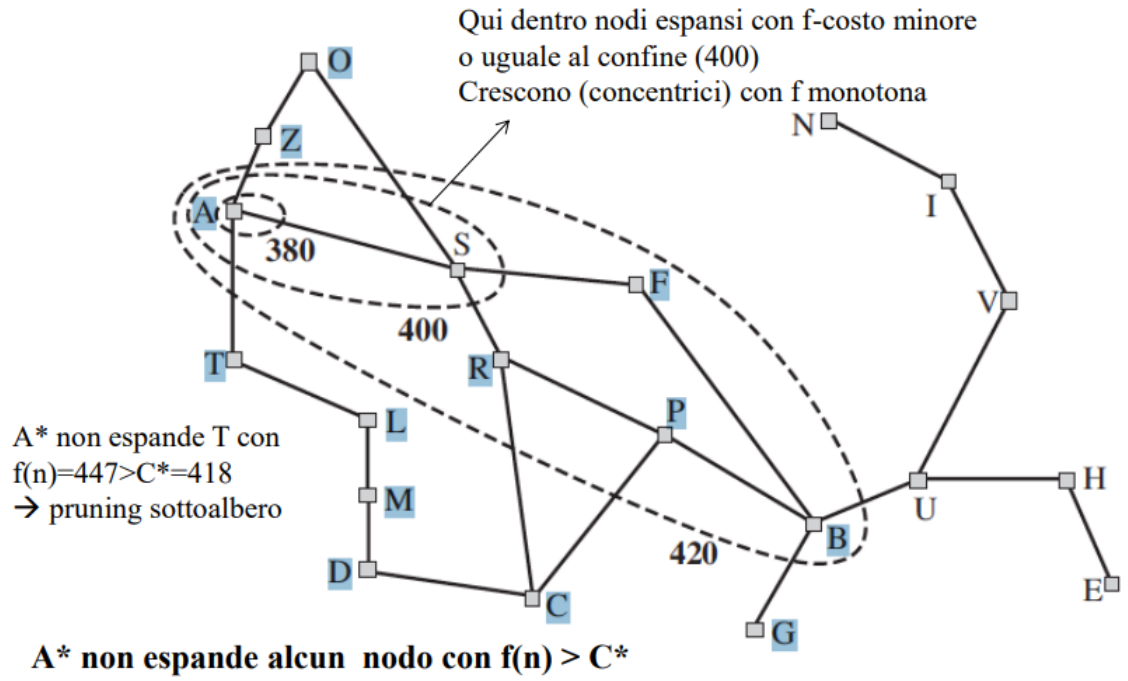
3.2.3 Perché A* è vantaggioso?

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 Conslusioni 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**.

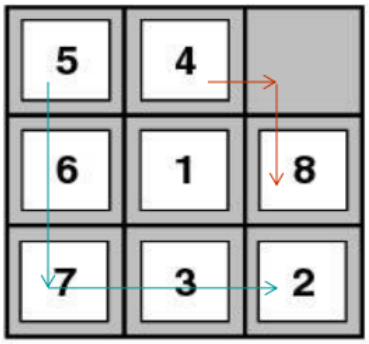
3.3.2 Confronto di euristiche ammissibili

Esempio Due euristiche ammissibili per il gioco dell'otto

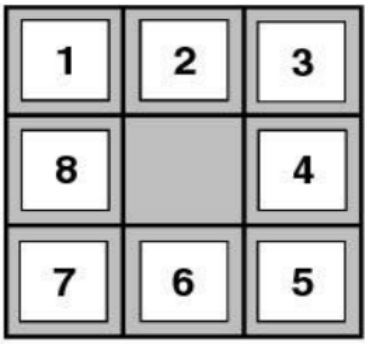
h_1 conta il numero di caselle fuori posto

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

$h1 = 7$

$h2 = 4+2+2+2+$
$+2+0+3+3 = 18$

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

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

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 $b = 2$

$d = 6 \quad N = 100$

$d = 12 \;\; N = 10000$

Con $b = 1.5$

$d = 12 \;\; N = 100$

$d = 24 \;\; N = 10000$

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

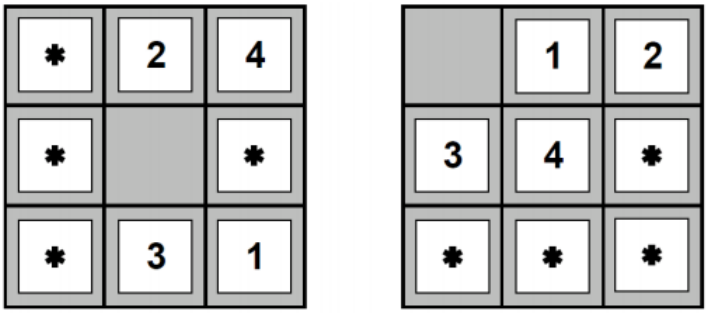
3.4.2 Massimizzazione di euristiche

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

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

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 $\rightarrow$ costo 14.

Combinazione di euristiche

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

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

Gioco dell'otto $h(n) = c_1 \# fuoriPosto + c_2 \# coppieScambiate$
Scacchi $h(n) = c_1 vantaggioPezzi + c_2 pezziAttaccante + c_3 regina + \ldots$
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.