Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso obbligatorio - 6 CFU

Introduzione intelligenza artificiale

Professore:
Prof. Alessio Micheli
Prof. Claudio Gallicchio

Autore:
Matteo Giuntoni

Anno Accademico 2023/2024

Contents

Introduzione all'Intelligenza Artificiale

Realizzato da: Matteo Giuntoni, Ghirardini Filippo

A.A. 2023-2024

---

1 Introduzione

1.1 Obiettivi dell'IA

1.1.1 Modellare

Modellare fedelmente l'essere umano:

- **Agire umanamente**: Test di Turing[1]
- **Pensare umanamente**: modelli cognitivi per descrivere il funzionamento della mente umana

1.1.2 Risultati

Raggiungere i risultati ottimali:

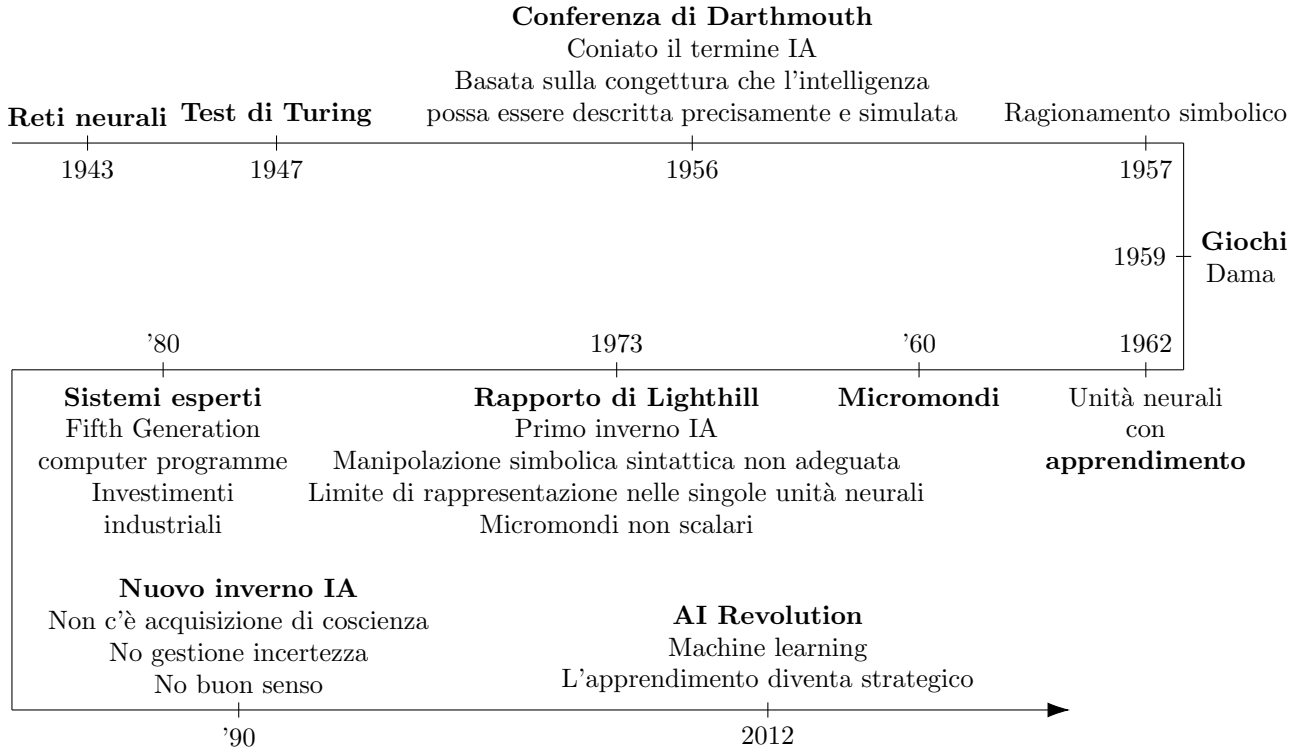
- Pensare razionalmente
- Agenti razionali: percepiscono l'ambiente, operano autonomamente e si adattano. Fanno la cosa giusta agendo in modo da ottenere il miglior risultato calcolando come agire in modo efficace e sicuro in una varietà di situazioni nuove. Ha alcuni vantaggi:

  1. Estendibilità e generalità
  2. Misurabilità dei risultati rispetto all'obiettivo

  I limiti dipendono dai rischi, dall'etica e dalla complessità computazionale.

1.2 Storia dell'IA

Nasce sin dall'antichità con il desiderio dei filosofi di sollevare l'uomo dalle fatiche del lavoro. Dal 1940 c'è un esplosione di popolarità che però si alterna tra periodi di crisi e di grandi avanzamenti.



[1] Ci sono due umani e una macchina. Tutti questi conversano tramite un computer. Se l'esaminatore non riesce a distinguere l'essere umano dalla macchina allora vince quest'ultima.

Esempio 1.2.1 (Scacchi)**.** Un esempio propedeutico è quello dell'applicazione dell'IA al gioco degli scacchi, definita **IA debole**. Negli anni '60 c'erano principalmente due opinioni al riguardo:

- Newell e Simon sostenevano che in 10 anni le macchine sarebbero state campioni negli scacchi
- Dreyfus sosteneva che una macchina non sarebbe mai stata in grado di giocare a scacchi

Nel 1997 la macchina Deep Blue sconfigge il campione mondiale di scacchi Kasparov. Viene naturale farsi alcune domande...

- Ha avuto **fortuna**?
- Ha avuto un **vantaggio psicologico**? La macchina eseguiva le mosse immediatamente e Kasparov si sentiva come l'ultimo baluardo umano.
- **Forza bruta**? La macchina calcolava 36 miliardi di posizioni ogni 3 minuti

Oggi l'Intelligenza Artificiale eccelle in tutti i giochi. L'ultimo a "cadere" è stato il Go nel 2016. Allo stesso tempo però il livello delle persone è aumentato giocando contro le macchine.

Definizione 1.2.1 (IA debole)**.** *Al contrario dell'IA forte, non ha lo scopo di possedere abilità cognitive generali, ma piuttosto di essere in grado di risolvere esattamente un singolo problema.*

1.3 Reti neurali

Le reti neurali sono caratterizzate da:

- **Flessibilità**: capacità di acquisizione automatica di conoscenza e di adattamento automatico a contesti diversi e dinamici
- **Robustezza**: capacità di trattare incertezza e rumorosità del mondo reale
- Rappresentazione appresa dai dati in forma **sub-simbolica**
- Possibilità di usare più strati di reti neurali con diversi livelli di astrazione (**Deep Learning**)

1.3.1 Deep Learning

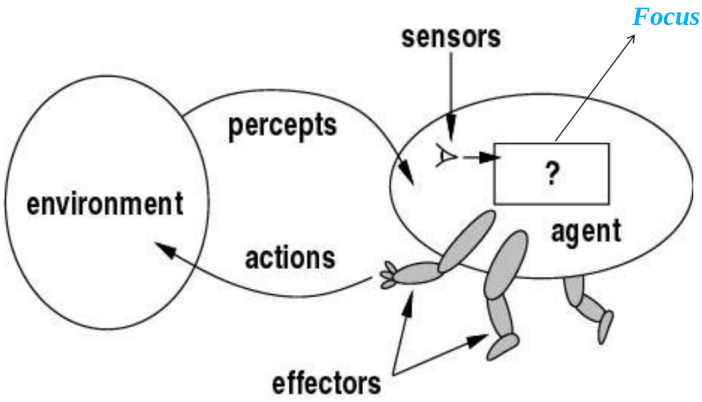
Abbinando alla capacità dei modelli di machine learning una grande quantità di dati e degli High Performance Computer, si è favorito molto il deep learning.
Dal 2010 le reti neurali profonde hanno iniziato a diffondersi molto nelle grandi industrie, riscuotendo successo ad esempio:

- **Computer vision**: ad esempio la classificazione del cancro della pelle
- **Natural Language Processing**: ad esempio IBM Watson o Google DeepL

Questa tecnologia ha raggiunto prestazioni a livello di quelle umane.

2 Agenti intelligenti

L'approccio moderno dell'IA (AIMA) è quello di costruire degli **agenti intelligenti**. La visione ad agenti offre n quadro di riferimento e una prospettiva più generale. È utile anche perché è **uniforme**.



Ciclo *percezione- azione*

Noi ci concentreremo sul programma che sta al centro dell'agente e che consiste in un ciclo di percezione-azione.

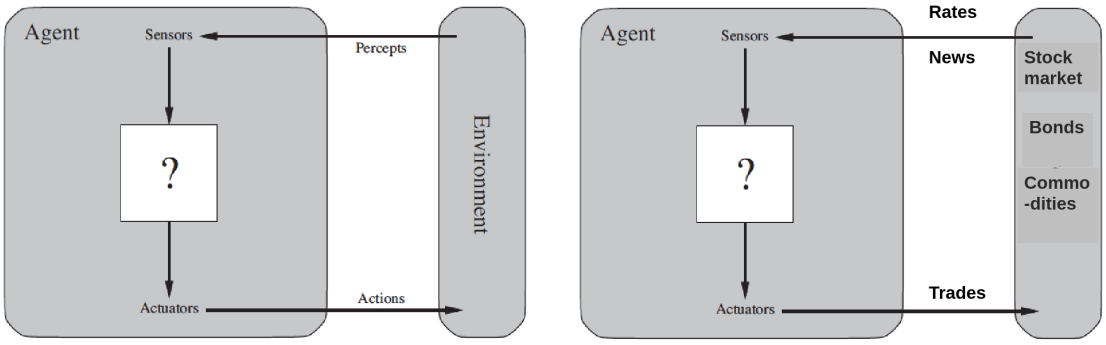
2.1 Caratteristiche

Un agente ha alcune caratteristiche:

- **Situati**: ricevono *percezioni* da un ambiente e agiscono mediante **azioni** (attuatori)

2.1.1 Percezioni e azioni

Le percezioni corrispondono agli **input** dai sensori. La **sequenza percettiva** sarà la storia completa delle percezioni.
La scelta dell'azione è *funzione* unicamente della sequenza percettiva ed è chiamata **funzione agente**.
Il compito dell'IA è costruire il programma agente.



2.2 Agente razionale

Un agente razionale interagisce con il suo ambiente in maniera **efficace** (fa la cosa giusta). Si rende quindi necessario un **criterio di valutazione** oggettivo dell'effetto delle azioni dell'agente. La valutazione della prestazione deve avere le seguenti caratteristiche:

- Esterna (come vogliamo che il mondo evolva?)
- Scelta dal progettista a seconda del problema considerando l'effetto desiderato sull'ambiente.
- (possibile) Valutazione su ambienti diversi.

Definizione 2.2.1 (Agente razionale)**.** *Per ogni sequenza di percezioni compie l'azione che massimizza il valore atteso della misura delle prestazioni, considerando le sue percezioni passate e la sua conoscenza pregressa.*

Osservazione 2.2.1. Si basa sulla razionalità e non sull'onniscenza e onnipotenza: non conosce alla perfezione il futuro ma può apprendere e hai dei limiti nelle sue azioni.

Raramente tutta la conoscenza sull'ambiente può essere fornita a priori dal programmatore. L'agente razionale deve essere in grado di modificare il proprio comportamento con l'esperienza. Può **migliorare** esplorando, apprendendo, aumentando l'autonomia per operare in ambienti differenti o mutevoli.

Definizione 2.2.2 (Agente autonomo)**.** *Un agente è autonomo nella misura in cui il suo comportamento dipende dalla sua capacità di ottenere esperienza e non dall'aiuto del progettista.*

2.3 Ambienti

Definire un problema per un agente significa innanzitutto caratterizzare l'ambiente in cui opera. Viene utilizzata la descrizione **PEAS**:

- **P**erformance
- **E**nviroment
- **A**ctuators
- **S**ensors

Prestazione	Ambiente	Attuatori	Sensori
Arrivare alla destinazione, sicuro, veloce, ligio alla legge, viaggio confortevole, minimo consumo di benzina, profitti massimi	Strada, altri veicoli, pedoni, clienti	Sterzo, acceleratore, freni, frecce, clacson, schermo di interfaccia o sintesi vocale	Telecamere, sensori a infrarossi e sonar, tachimetro, GPS, contachilometri, accelerometro, sensori sullo stato del motore, tastiera o microfono

L'ambiente deve avere le seguenti proprietà:

- Osservabilità:
  - Se è **completamente osservabile** l'apparato percettivo è in grado di dare conoscenza completa dell'ambiente o almeno tutto ciò che è necessario per prendere l'azione
  - Se è **parzialmente osservabile** sono presenti limiti o inaccuratezze dell'apparato sensoriale
- Agente singolo o multi-agente:
  - L'ambiente ad agente **singolo** può anche cambiare per eventi, non necessariamente per azioni di agenti
  - Quello **multi-agente** può essere *competitivo* (scacchi) o *cooperativo*
- Predicibilità:
  - **Deterministico**: quando lo stato successivo è completamente determinato dallo stato corrente e dall'azione (e.g. scacchi)
  - **Stocastico**: quando esistono elementi di incertezza con associata probabilità (e.g. guida)
  - **Non deterministico**: quando si tiene traccia di più stati possibili risultato dell'azione ma non in base ad una probabilità
- Episodico o sequenziale:

  - **Episodico**: quando l'esperienza dell'agente è divisa in episodi atomici indipendenti in cui non c'è bisogno di pianificare (e.g. partite diverse)
  - **Sequenziale**: quando ogni decisione influenza le successive (e.g. mosse di scacchi)

- Statico o dinamico:
  - **Statico**: il mondo non cambia mentre l'agente decide l'azione (e.g cruciverba)
  - **Dinamico**: cambia nel tempo, va osservata la contingenza e tardare equivale a non agire (e.g. taxi)
  - **Semi-dinamico**: l'ambiente non cambia ma la valutazione dell'agente sì (e.g. scacchi con timer)

- Valori come lo stato, il tempo, le percezioni e le azioni possono assumere valori **discreti** o **continui**. Il problema è combinatoriale nel discreto o infinito nel continuo.

- **Noto** o **ignoto**: una distinzione riferita alla conoscenza dell'agente sulle leggi fisiche dell'ambiente (le regole del gioco). È diverso da osservabile.

Definizione 2.3.1 (Simulatore)**.** *Un simulatore è uno strumento software che si occupa di:*

- *Generare stimoli*
- *Raccogliere le azioni in risposta*
- *Aggiornare lo stato*
- *Attivare altri processi che influenzano l'ambiente*
- *Valutare la prestazione degli agenti (media su più istanze)*

Gli esperimenti su classi di ambienti con condizioni variabili sono essenziali per ***generalizzare****.*

2.4 Programma agente

L'agente sarà quindi composto da un'architettura e da un programma. Il programma dell'agente implementa la funzione agente $Ag : Percezioni \rightarrow Azioni$.

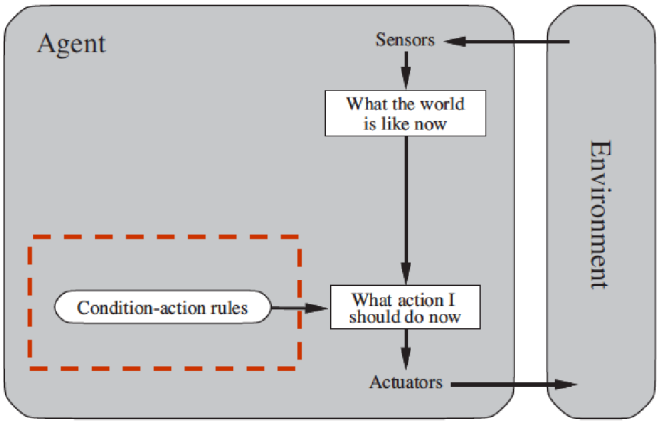
```
function Skeleton-Agent (percept) returns action
   static: memory, agent memory of the world
   memory <- UpdateMemory(memory, percept)
   action <- Choose-Best-Action(memory)
   memory <- UpdateMemory(memory, action)
   return action
```

2.4.1 Tabella

Un agente basato su tabella esegue una scelta come un accesso ad una tabella che associa un'azione ad ogni possibile sequenza di percezioni.
Ha una **dimensione ingestibile**, è difficile da costruire, non è autonomo ed è di difficile aggiornamento (apprendimento complesso).

2.4.2 Agenti reattivi

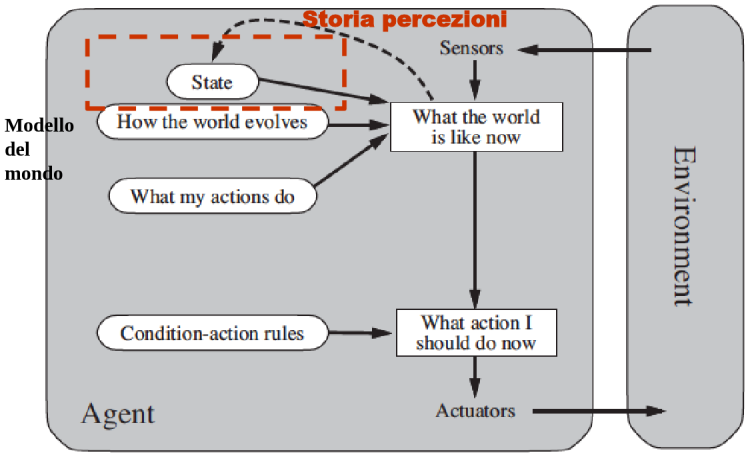
L'agente agisce in base a quello che percepisce senza salvare nulla in memoria.




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

2.4.3 Agenti basati su modello

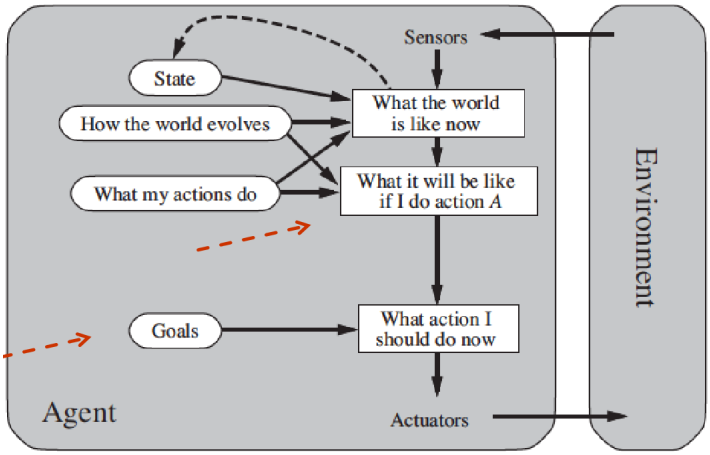
L'agente ha uno stato che mantiene la storia delle percezioni e influenza il modello del mondo.




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

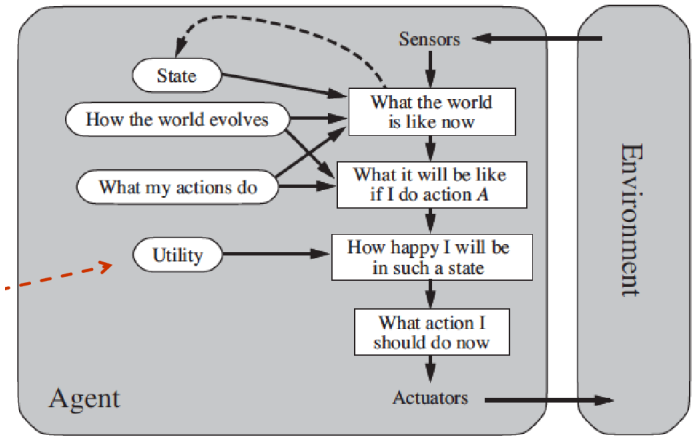
2.4.4 Agenti con obiettivo

Fin'ora l'agente aveva un obiettivo predeterminato dal programma. In questo caso invece viene specificato anche il **goal** che influenza le azioni. Abbiamo quindi più **flessibilità** ma meno efficienza.



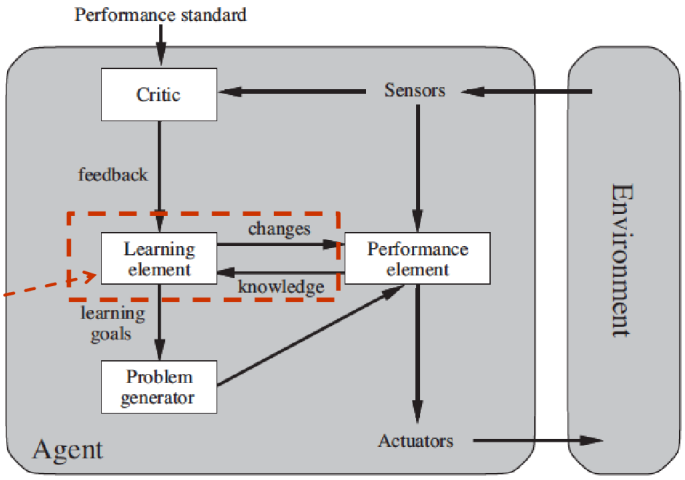
2.4.5 Agenti con valutazione di utilità

In questo caso ci sono **obiettivi alternativi** o più modi per raggiungerlo. L'agente deve quindi decidere verso dove muoversi e si rende necessaria una **funzione utilità** che associ ad un obiettivo un numero reale. La funzione terrà anche conto della probabilità di successo (**utilità attesa**).



2.4.6 Agenti che apprendono

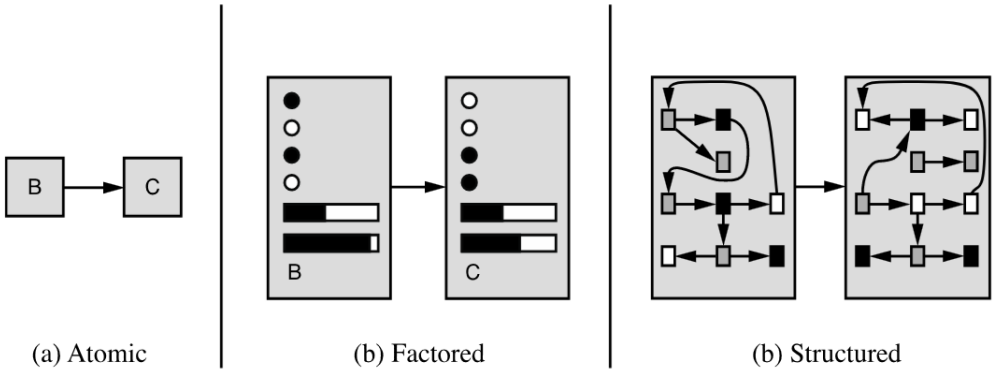
Questo tipo di agente include la capacità di **apprendimento** che produce cambiamenti al programma e ne migliora le prestazioni, adattando i comportamenti.
L'elemento **esecutivo** è il programma stesso, quello **critico** osserva e dà feedback ed infine c'è un generatore di problemi per suggerire nuove situazioni da esplorare.



2.4.7 Tipi di rappresentazione

Gli stati e le transizioni possono essere rappresentati in tre modi:

- **Atomica**: solo con gli stati
- **Fattorizzata**: con più variabili e attributi
- **Strutturata**: con l'aggiunta delle relazioni



3 Agenti risolutori di problemi

Gli agenti risolutori di problemi adottano il paradigma della risoluzione di problemi come **ricerca** in uno **spazio di stati**. Sono agenti con **modello** (storia percezioni e stati) che adottano una rappresentazione **atomica** degli stati.
Sono particolari gli agenti con **obiettivo** che pianificano l'intera sequenza di mosse prima di agire.

3.1 Processo di risoluzione

I passi da seguire sono i seguenti:

1. **Determinazione di un obiettivo**, ovvero un insieme di stati in cui l'obiettivo è soddisfatto
2. **Formulazione** del problema tramite la rappresentazione degli stati e delle azioni
3. Determinazione della **soluzione** mediante la ricerca
4. **Esecuzione** del piano

Esempio 3.1.1 (Viaggio con mappa)**.** Supponiamo di voler fare un viaggio. Il processo di risoluzione sarebbe il seguente:

1. Raggiungere Bucarest
2. - Azioni: guidare da una città all'altra
   - Stato: città su mappa

Assumiamo che l'ambiente in questione sia **statico**, **osservabile**, **discreto** e **deterministico** (assumiamo un mondo ideale).

3.2 Formulazione del problema

Un problema può essere definito formalmente mediante 5 componenti:

1. **Stato iniziale**
2. **Azioni** possibili
3. **Modello di transizione**: $ris : stato \times azione \to stato$, uno stato *successore* $ris(s, a) = s'$
4. **Test obiettivo** per capire tramite un insieme di stati obiettivo se il goal è raggiunto $test : stato \to \{true, false\}$
5. **Costo del cammino**: composto dalla somma dei costi delle azioni, dove un passo ha costo $c(s, a, s')$. Un passo non ha mai costo negativo.

I punti 1, 2 e 3 definiscono implicitamente lo **spazio degli stati**. Definirlo esplicitamente può essere molto costoso.

3.3 Algoritmo di ricerca

Gli algoritmi di ricerca prendono in input un problema e restituiscono un **cammino soluzione**.
Dobbiamo misurare le **prestazioni**: trova una soluzione? Quanto costa trovarla? Quanto è efficiente?

$$costo\_totale = costo\_ricerca + costo\_cammino\_sol$$

Esempio 3.3.1 (Arrivare a Bucarest)**.** Partiamo con la formulazione del problema:

1. **Stato iniziale**: la città di partenza, ovvero Arad
2. **Azioni**: spostarsi in una città collegata vicina

```
Azioni(In(Arad))={Go(Sibiu),Go(Zerind),...}
```

3. **Modello di transizione**:

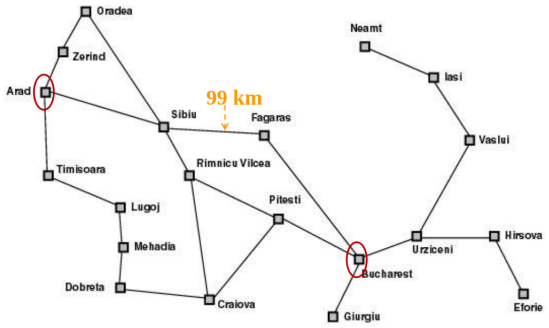
```
Risultato(In(Arad), Go(Sibiu)) = In(Sibiu)
```

4. **Test obiettivo**:

```
{In(Bucarest)}
```

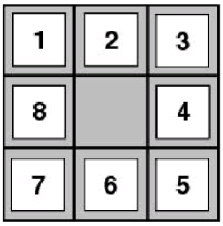
5. **Costo del cammino**: somma delle lunghezze delle strade

In questo esempio lo spazio degli stati coincide con la rete dei collegamenti tra le città.



Esempio 3.3.2 (Puzzle dell'8)**.** Partiamo con la formulazione del problema:

1. **Stati**: tutte le possibili configurazioni della scacchiera
2. **Stato iniziale**: una configurazione tra quelle possibili
3. **Obiettivo**: una configurazione del tipo



4. **Azioni**: le mosse della casella vuota
5. **Costo cammino**: ogni passo costa 1

In questo esempio lo spazio degli stati è un grafo con possibili cicli (ci possiamo ritrovare in configurazioni già viste). Il problema è NP-completo: per 8 tasselli ci sono $\frac{9!}{2} = 181.000$ stati.

Esempio 3.3.3 (8 regine)**.** Supponiamo di dover collocare 8 regine su una scacchiera in modo tale che nessuna regina sia attaccata da altre.

1. **Stati**: tutte le possibili configurazioni della scacchiera con 0-8 regine
2. **Goal test**: avere 8 regine sulla scacchiera, di cui nessuna è attaccata
3. **Azioni**: aggiungi una regina

In questo esempio lo spazio degli stati sono le possibili scacchiere, ovvero $64 \times 63 \times \ldots \times 57 \simeq 1.8 \times 10^{14}$. Proviamo ad utilizzare una formulazione diversa:

1. **Stati**: tutte le possibili configurazioni della scacchiera in cui *nessuna regina è minacciata*

2. **Goal test**: avere 8 regine sulla scacchiera, di cui nessuna è attaccata
3. **Azioni**: aggiungere una regina nella colonna vuota più a destra ancora libera in modo che non sia minacciata

Lo spazio degli stati passa a 2057, anche se comunque rimane esponenziale per k regine.
Vediamo infine un'ultima formulazione:

1. **Stati**: scacchiere con 8 regine, una per colonna
2. **Goal test**: nessuna delle regine già presenti è attaccata
3. **Azioni**: sposta una regina nella colonna se minacciata
4. **Costo cammino**: zero

Qui lo spazio degli stati è di qualche decina di milione.
Capiamo quindi che formulazioni diverse del problema portano a spazi di stati di dimensioni diverse.

Esempio 3.3.4 (Dimostrazione di teoremi)**.** Dato un insieme di premesse:

$$\{s, t, q \Rightarrow p, r \Rightarrow p, v \Rightarrow q, t \Rightarrow r, s \Rightarrow v\} \tag{1}$$

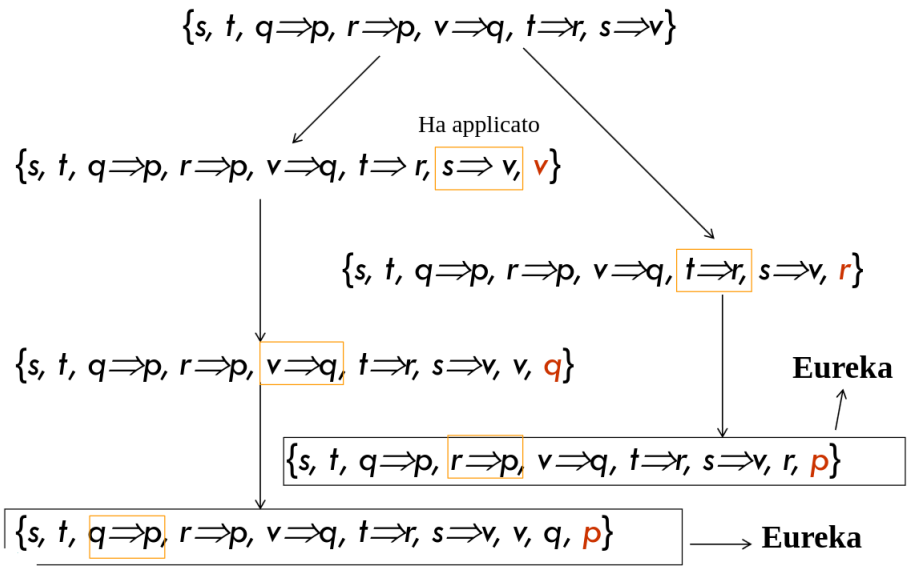
dimostrare una proposizione p utilizzando solamente la regola di inferenza *Modus Ponens*:

$$(p \wedge p \Rightarrow q) \Rightarrow q$$

Scriviamo la formulazione del problema:

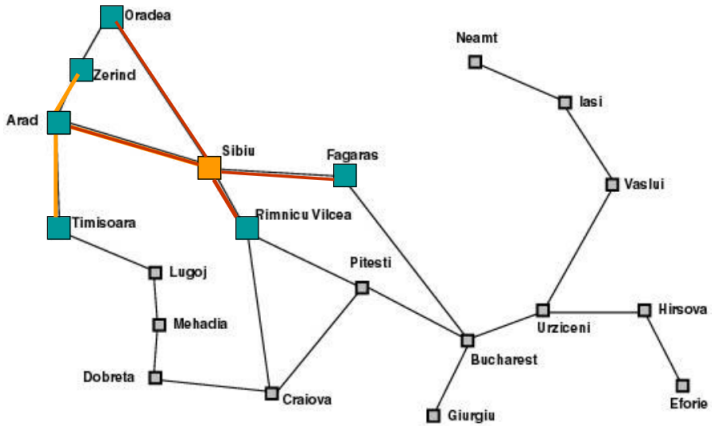
- **Stati**: insieme di proposizioni
- **Stato iniziale**: le premesse
- **Stato obiettivo**: un insieme di proposizioni contenente il teorema da dimostrare
- **Operatori**: l'applicazione del Modus Ponens

Lo spazio degli stati è quindi il seguente:



3.4 Ricerca della soluzione

La ricerca della soluzione consiste nella generazione di un **albero di ricerca** a partire dalle possibili sequenze di azioni che si sovrappone allo spazio degli stati.
Ad esempio per il caso di Bucarest:



Espandiamo ogni nodo con i suoi possibili successori (frontiera).

Osservazione 3.4.1. Si noti che un nodo dell'albero è diverso da uno stato. Infatti possono esitere nodi dell'albero di ricerca con lo stesso stato (si può tornare indietro).

La generazzione di un albero di ricera sovrapposto allo spazio degli stati:

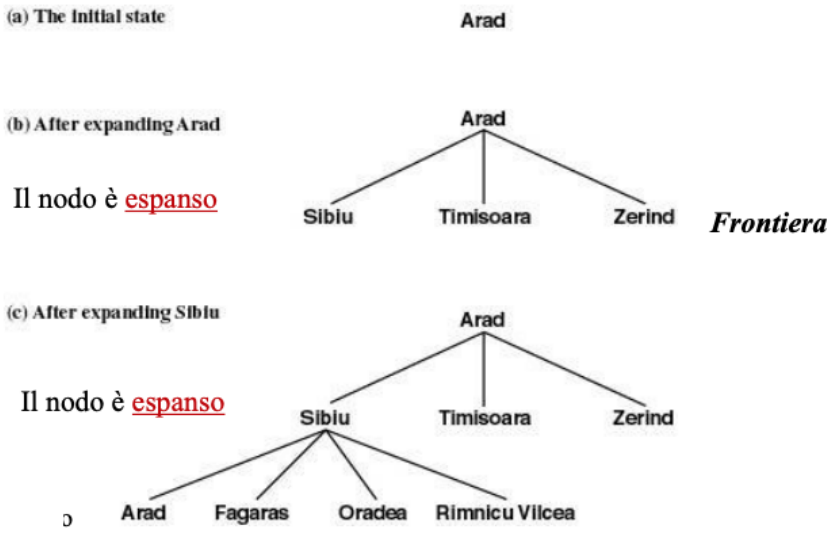


Figure 1: generazione di un albero di ricerca

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

Un nodo n in un albero di ricera è una struttura dati con quattro componenti:

1. Uno stato: n.stato
2. Il nodo padre: n.padre
3. L'azione effettuata per generarlo: n.azione
4. Il costo del cammino dal nodo iniziale al nodo: n.costo-cammino indicato come $g(n)$. (=padre.costo-cammino + costo-passo ultimo)

Abbiamo poi la struttura dati per la **frontiera** che è una lista dei nodi in attesa di essere espandi (le foglie dell'albero di ricerca). La frontiere è implementata come una coda con operazioni:

- Vuoto?(coda) = controlla se la coda è vuota
- POP(coda) = estrae il primo elemento.
- Inserisci(elemento, coda) = inserisce un elemento nella coda.
- Diversi tipi di coda hanno diverse funzioni di inserimento e implementano strategie diverse.
  - **FIFO** - First in Firt out $\Longrightarrow$ usato nella **BF (Breadth-first)**.
    Viene estratto l'elemento più vecchio (in attesa da più tempo); in nuovi nodi sono aggiunti alla fine.
  - **LIFO** - Last in Firt Out $\Longrightarrow$ usato nella **DF (deptht-first)**.
    Viene estratto il più recentemente inserito; i nuovi nodi sono inseriti all'inizio (pila).
  - **Coda con priorità** $\Longrightarrow$ usato nella **UC**, ed altri successivi.
    Viene estratto quello con priorità più alta in base a una funzione di ordinamento; dopo l'inserimento dei nuovi nodi si riordina.

4 Stategia ricerca non informative

Ora andremo a vedere diverse **strategie non informative**: Ricerca in ampiezza (BF), Ricerca in profondità (DF) Ricerca in profondità limitata (DL), Ricerca con approfondimento iterativo (ID), Ricerca di costo uniforme (UC). Successivamente le metteremo a confronto con strategie di **ricerca euristica (o informativa)** che fanno uso di informazio riguardo alla distanza stimata della soluzione.

La valutazione di una strategia verrà fatta andando a seguire i seguenti paramentri:

- **Completezza**: se la soluzione esiste viene trovata.
- **Ottimalità (ammissibilità)**: trovare la soluzione migliore con costo minore (per il costo del cammino soluzione).
- **Complessità in tempo**: tempo richiesto per trovare la soluzione (per il costo della ricerca)
- **Complessità in spazio**: memoria richiesta (per il costo della ricerca).

4.1 Ricerca in ampiezza (BF)

O come esplorare il grafo dello spazio degli stati a livelli progressivi di stessa profondità. Viene inoltre implementata con una coda che inserisce alla fine (FIFO).

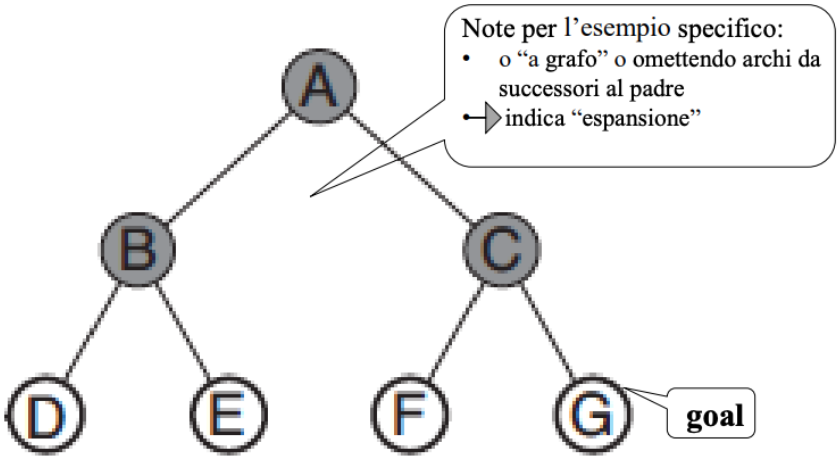


Figure 2: Ricerca in ampiezza

```
function Ricerca-Ampiezza (problema)
   return soluzione oppure fallimento
   nodo = un nodo con stato il problema.stati-iniziale e costo-di-cammino=0
   if problema.Test-Obiettivo(nodo.Stato) then return Soluzion(nodo)
   frontiera = una coda FIFO con nodo come unico elemento
   loop do
      if(Vuota?(frontiera)) then return fallimento
      nodo = POP(frontiera)
      for each azione in problema.Azioni(nodo.Stato) do
         figlio = Nodo-Figlio(problema, nodo, azione) [costruttore: vedi AIMA]
         if Problema.TestObiettivo(figlio.Stato) then return Soluzione(figlio)
         frontiera = Inserisci(figlio, frontiera) /* frontiera gestita come coda FIFO
```

Note 4.1.1. Nota che in questa versione i nodo.stato sono goal-tested al momento in cui sono generati, anticipato→ più efficiente, si ferma appena trova goal prima di espandere.

Una versione aggiornata dove evitiamo di espandere (nodi con) stati già esplorati:

```
function Ricerca-Ampiezza (problema)
   return soluzione oppure fallimento
   nodo = un nodo con stato il problema.stati-iniziale e costo-di-cammino=0
   if problema.Test-Obiettivo(nodo.Stato) then return Soluzion(nodo)
   frontiera = una coda FIFO con nodo come unico elemento
   esplorati = indieme vuoto //aggiunto per gestire gli stati ripetuti
   loop do
      if(Vuota?(frontiera)) then 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) [costruttore: vedi AIMA]
         if figlio.Stato non e in esplorati e non e in frontiera then // aggiunto check per
             vedere se e in frontiera
         frontiera = Inserisci(figlio, frontiera) /* frontiera gestita come coda FIFO
```

Abbiamo aggiunto **esplorati = insieme vuoto** e **if figlio.Stato non è in esplorati e non è in frontiera then** per gestire gli stati ripetuti.

Ora sempre lo stesso codice in uno script 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 del nodo
  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
```

4.1.1 Analisi complessità spazio-temporale (BF)

Inanzitutto assumiamo che:

- **b** = fatto di ramificazione (**b**ranching)
- **d** = profondità del nodo obiettivo piu superficiale (**d**epth)
- **m** = lunghezza massima dei cammini nello spazio degli stati (**m**ax)

La **strategia ottimale** è se gli operatori hanno tutti lo stesso costo k cioè $g(n) = k \cdot depth(n)$ dove $g(n)$ è il costo del cammino per arrivare a n.

La **complessità nel tempo** (nodi generati):

$$T(b,d) = 1 + b + b^2 + \cdots + b^d \rightarrow O(b?d) \text{ b figli per ogni nodo}$$

Note 4.1.2. Riflettere che il numero nodi cresce exp., non assumiamo di conoscere già il grafo ne una notazione di linearità nel numero nodi . Questo è tipico dei problemi in AI (pensate a quelli generati per le configurazioni dei giochi, con rappresentazione implicita dello spazio stati, non esplicitamente/staticamente in spazi enormi).

La **complessità nello spazio** (nodi in memoria): $O(b^d)$

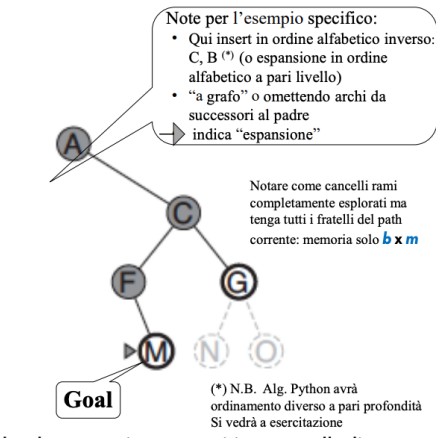
Esempio 4.1.1. b=10, 1 milione nodi al sec generati; 1 nodo occupa 1000 byte

Piu incisivo!

Profondità	Nodi	Tempo	Memoria
2	110	0,11 ms	107 kilobyte
4	11.100	11 ms	10,6 megabyte
6	10^6	1.1 sec	1 gigabyte
8	10^8	2 min	103 gigabyte
10	10^{10}	3 ore	10 terabyte
12	10^{12}	13 giorni	1 petabyte
14	10^{14}	3,5 anni	1 esabyte

Scala male: solo istanze piccole!

4.2 Ricerca in profondità (DF)



Viene implementata da una coda che mette i succerrosi in testa alla lista (LIFO, pila o stack). L'algoritmo è generale e può essere usato sia con alberi che con grafi. Notare come cancelli rami completamente esplorati ma tenga tutti i fratelli del path corrente: memoria solo $b \times m$

Analisi - Versione su albero.
Se $m \rightarrow$ lungehzza massima dei cammini nello spazio degli stati e $b \rightarrow$ fattore di diramazione.
Abbiamo che tempo: $O(b^m)$ [che può essere $> O(b^d)$].
Occupazione memoria: bm [frontiera sul cammino].

Analisi - Versione su grafo
In caso di DF con visita grafo si perderebbe i **vantaggi di memoria**: la memoria torna da bm a tutti i possibili stati (potenzialmente, caso pessimo, esponenziale come BF*) (per mantenere la lista dei visitati/esplorati), ma così DF diviene **completa** in spazi degli stati finiti (tutti i nodi verranno espansi nel caso pessimo).
In ogni caso resta non completa in spazi infiniti. È possibile controllare anche solo i nuovi nodi rispetto al cammino radice-nodo corrente senza aggravio di memoria (evitando però così i cicli in spazi finiti ma non i cammini ridondanti).

4.2.1 DF ricorsiva

Ancora più efficiente in occupazione di memoria perché mantiene solo il cammino corrente (solo m nodi nel 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 (generando i nodi fratelli al momento del backtracking).

```
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) then 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
```

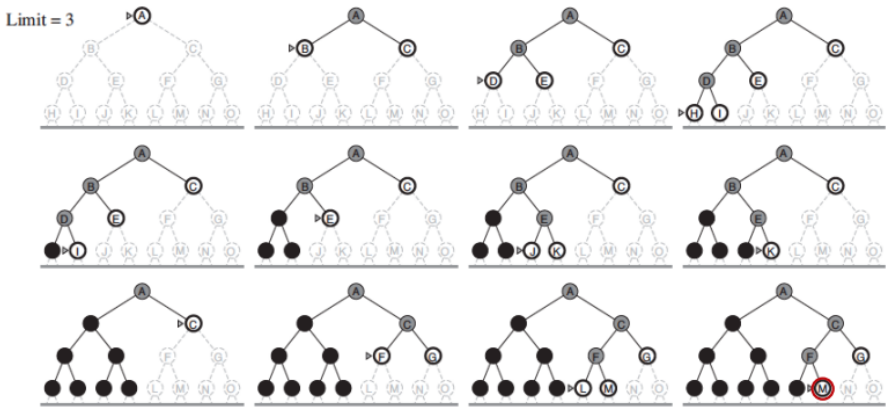
```python
def recursive_depth_first_search(problem, node):
  """Ricerca in profondita' ricorsiva """
  # controlla se lo stato del nodo e' uno stato obiettivo
  if problem.goal_test(node.state):
    return node.solution()
  # in caso contrario continua
  for action in problem.actions(node.state):
    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
```

4.2.2 Ricerca in profonditò limitata

Si va in profondità fino ad un certo livello predefinito l. È una soluzione definibile **completa** per problemi in cui si conosce un limite superiore per la profondità della soluzione (e.s Route-finding limitata dal numero di città - 1). È però completa se $d < l$ (d profondità nodo obbiettivo superficiale). Questa soluzione non è ottimale.
Complessità tempo: $O(b^l)$
Complessità spazio: $O(bl)$

4.3 Approfondimento iterativo (ID)

Si prova DF (DL) con limite di profondità 0, poi 1, poi 2, poi 3 ... fino a trovare la soluzione.



Miglior compromesso tra BF e DF.

$$BF : b + b^2 + \cdots + b^{d-1} + b^d \quad conb = 10 ed = 5$$

$$10 + 100 + 1000 + 10.000 + 100.000 = 111.110$$

ID: i nodi dell'ultimo livello generati una volta, quelli del penultimo 2, quelli del terzultimo 3 ... quelli del primo d volte.

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

$$= 50 + 400 + 3000 + 20.000 + 100.000 = 123.450$$

Complessita tempo: $O(b^d)$ (se esiste soluzione)
Spazio: $O(bd)$ (se esiste soluzione) versus $O(b^d)$ della BF.

Ergo: Vantaggi della BF (completo, ottimale se costo fisso oper. K), con tempi analoghi ma costo memoria analogo a quello di DF.

4.4 Direzione della ricerca

Un problema ortogonale alla strateiga è la direzione della ricerca:

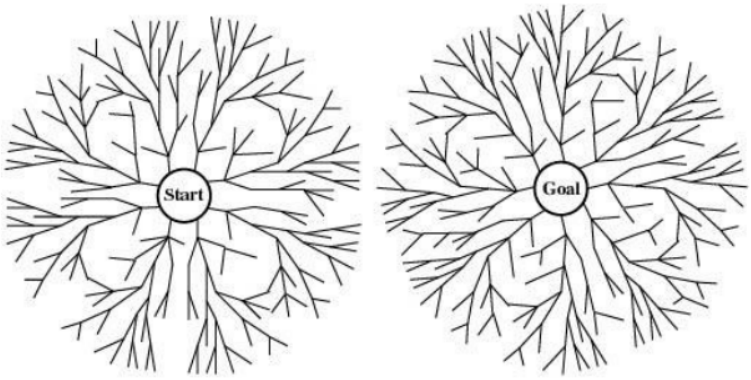
- Ricerca **in avanti** o **guidata dai dati**: si esplora lo spazio di ricerca dallo stato iniziale allo stato obiettivo.
- Ricerca **all'indietro** o **guidata dall'obbiettivo**: si esplora lo spazio di ricerca a partire da uno stato goal e riconducendosi a sotto-goal fino a trovare uno stato iniziale.

A questo punto bisogna capire quale direzione scegliere?
Conviene procedere nella direzione in cui il fattore di diramazione è minore.

- Si preferisce ricerca all'indietro quando, e.g l'obbiettivo è chiaramente definito (e.g theorem proving) o si possono formulare una seire limitata di ipotesi.
- Si preferisce ricerca in avanti quando e.g gli obbiettivi possibili sono molti (design).

4.4.1 Ricerca bidirezioanle

Si procede nelle dure direzioni fino ad incontrarsi.

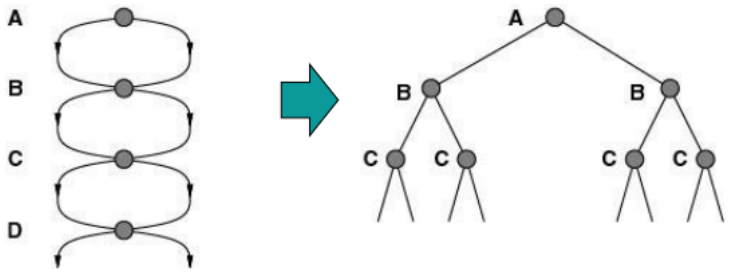


Complessità tempo: $O(b^{d/2})$ (assumendo test intersezione in tempo costante, es. hash table).
Complessità spazio: $O(b^{d/2})$ (almeno tutti i nodi una direzione in memoria, es usando BF).

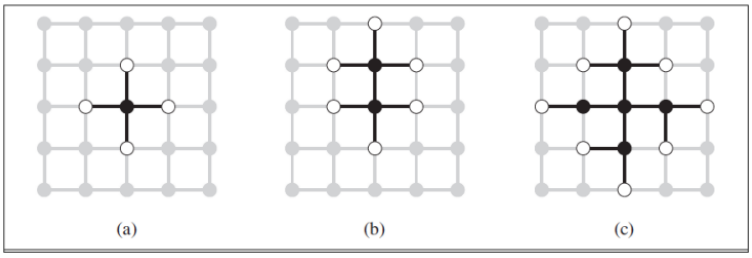
Note 4.4.1. Non sempre applicabile, es. predecessori non definiti, troppi stati obiettivo, ...

4.5 Ridondanze

Su spazi di stati a grafo si possono generare piu volte gli stesso nodi (o meglio nodi con stesso stato) nella ricerca, **anche in assenza di cicli** (cammini ridondanti).



Se vediamo per esempio il caso di ridondance nelle griglie spesso si vanno a visitare stati già visitati questa operazione fa compiere lavoro inutile. Come evitarlo?



Ricordare gli stati già visitati occupa spazio (es. lista **esplorati** in visita a grafo) ma ci consente di evitare di visitarli di nuovo. Gli algoritmi che dimenticano la propria storia sono desinati a ripeterlo! Abbiamo tre soluzioni, in ordine crescente di costo e di efficacia:

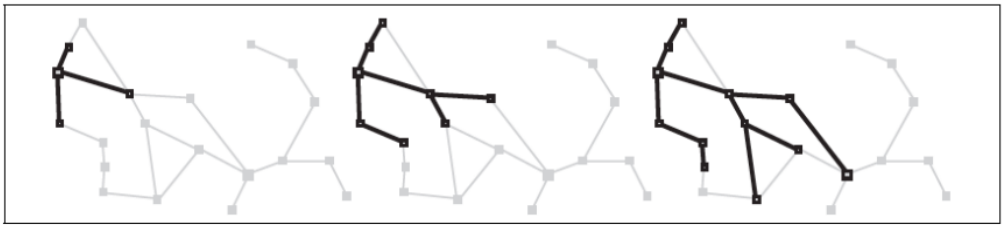
- Non tornare nello stato da cui si proviene: si elimina il genitore dai nodi successori (non evita i cammini ridondanti).
- Non creare cammini con cicli: si controlla che i successori non siano antenati del nodo corrente (detto per la DF).

- 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 (e.g. hash table per accesso efficiente).

Ricordare che il costo può essere alto: in caso di DF (profon.) la memoria torna da bm a tutti gli stati, ma diviene una ricerca completa (per spazi finiti). Ma in molti casi gli stati crescono exp. (gioco otto, scacchi, ...)

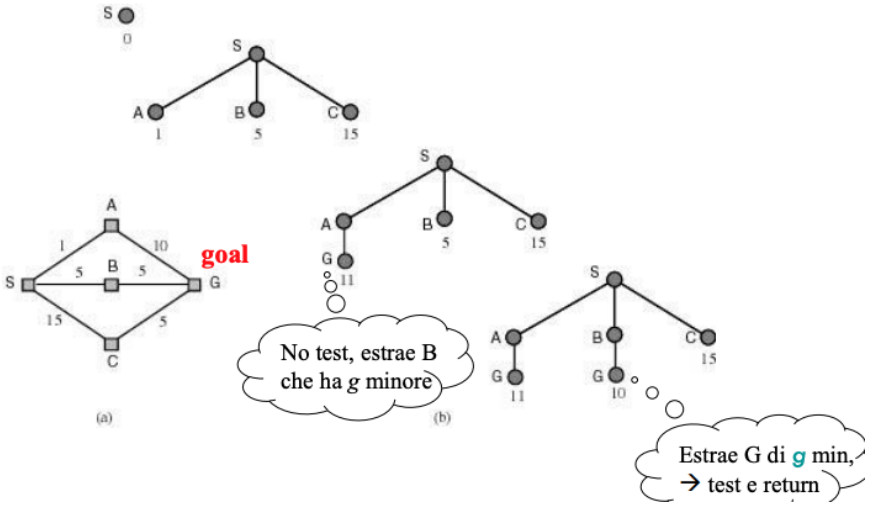
Note 4.5.1. Ricorda che in una ricerca su un grafo:

- Si mantiene una lista dei nodi (stati) visitati/esplorati (anche detta **lista chiusa**) [2]
- Prima di espandere un nodo si controlla se lo stato era stato già incontrato prima o è gia nella fronitera.
- Se questo succede, il nodo appena trovato non viene espanso.
- Ottimale solo se abbiamo la garanzia che il costo del nuovo cammino sia maggiore o uguale (cioò che il nuovo cammino non conviene)



La ricerca su grafo esplora ino stato al piu una volta. Una proprietà è che: la **frontiera** separa i nodi esplorati da quelli non-esplorati (ogni cammino dallo stato iniziale a inesplorati deve attraversare uno stato della frontiera).

Generalizzazione della ricerca in ampiezza (costi diversi tra passi): si sceglie il nodo di costo minore sulla frontiera (si intende il costo g(n) del cammino), si espande sui contorni di **uguale (o meglio uniforme) costo (e.g. in km)** invece che sui contorni di uguale profondità (BF).



[2] Ed. IV AIMA: introduce il termine di insieme di stati "raggiunti" che include sia (gli stati del)la frontiera che la lista degli esplorati

4.6 Ricerca di costo uniforme (UC)

Generalizzazione della ricerca in ampiezza (costi diversi tra passi): si sceglie il nodo di costo minore sulla frontiera (si intende il costo $g(n)$ del cammino), si espande sui contorni di **uguale (o meglio uniforme) costo (e.g. in km)** invece che sui contorni di uguale profondità (BF). Implementata da una coda ordinata per costo cammino crescente (in cima i nodi di costo minore). Codice di ricerca UC 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)
      // Posticipato* per vedere il costo minore su g (diverso da BF, ma tipico per coda
          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
```

Codice di ricerca UC 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);
      // posticipato per vedere il costo minore
      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 e in esplorati e non e in frontiera then
            frontiera = Inserisci(figlio, frontiera) /* in coda con priorita */
         else if figlio.Stato e in frontiera con Costo-cammino piu alto then // costo
             cammino piu alto g
            sostituisci quel nodo frontiera con figlio
```

Codice in python della ricerca:

```
def uniform_cost_search(problem): """Ricerca-grafo UC"""
  explored =[] # insieme (implementato come una lista) degli stati gia' visitati
  node =Node(problem.initial_state) # il costo del cammino e' inizializzato nel costruttore del nodo
  frontier =PriorityQueue(f =lambda x:x.path_cost) # la frontiera e' una coda coda con priorita'
  #lambda serve a definire una funzione anonima a runtime
  frontier.insert(node)
  while not frontier.isempty():
    # seleziona il nodo node = frontier.pop() # estrae il nodo con costo minore, per l'espansione
    if problem.goal_test(node.state):
      return node.solution(explored_set =explored)
    else: # se non lo e' inserisci lo stato nell'insieme degli esplorati
    explored.append(node.state)
      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)):
      frontier.insert(child_node)
    elif frontier.contains_state(child_node.state) and
       (frontier.get_node(frontier.index_state(child_node.state)).path_cost >child_node.path_cost):
```

```
        frontier.remove(frontier.index_state(child_node.state))
        frontier.insert(child_node)
    return None # in questo caso ritorna con fallimento
```

4.6.1 Analisi

Ottimalità e **completezza** garantite purchè il costo degli archi sia maggiore di $\epsilon > 0$. Appunto C^* come il costo della soluzione ottima, $\lfloor C^*/\epsilon \rfloor$ è il numero di mosse nel caso peggiore, arrotondato per difetto (e.g. attratto ad andare versio tante bassa)

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

Note 4.6.1. Quando ogni azione ha lo stesso costo UC somiglia a BF ma complessità $O(b^{1+d})$

Causa esame ed arresto posticipato, solo dopo aver espando frontiera, oltre la profondità del goal.

4.7 Confronto strategie

Criterio	BF	UC	DF	DL	ID	Bidir
Completa?	si	si(^)	no	si (+)	si	si (£)
Tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Spazio	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Ottimale?	si(*)	si(^)	no	no	si(*)	si (£)

(*) se gli operatori/archi hanno tutti lo stesso costo.
(∧) per costi degli archi $\geq \epsilon$>0.
(+)per problemi per cui si conosce un limite alla profondità della soluzione (se $l > d$).
(£)usando UC (o BF).

5 Ricerca euristica

La ricerca esaustiva non è praticabile in problemi di complesità esponenziale (e.g. $10^{1}20$ configurazioni in scacchi). Noi usiamo conoscenza del problemi es esperienza per riconoscere i cammini più promettenti, usiamo una stima del costo futuro, evitando di generare gli altri. La conoscenza euristica (dal greco "eureka") aiuta fare scelte "oculate", questa ovviamente però non evita la ricerca ma la riduce, consente in genere di trovare una buona soluzione in tempi accettabilili sotto certe condizioni garantisce completezza e ottimalità.

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

$$h : n \to R$$

La funzione si applica al nodo ma dipende solo dallo stato (n.Stato).

Note 5.0.1. Manteniamo la notazione in n per unifomritò con g; g dipende anche dal cammino fino al nodo.

$$f(n) = g(n) + h(n) \text{ove g(n) è il costo cammino visto con UC}$$

Per procedere preferibilmente verso il percorso migliore, seguendo problem-specific information, di stima del costo futuro:

- La città più vicina (o la città più vicina alla metà in linea d'aria - tabella esterna) nel problema dell'itinerario.
- Il numero della caselle fuori posto nel gioco dell'otto.
- Il vataggio in pezzi della dama o negli scacchi

Esempio 5.0.1. Mappa Romania dist. in linea d'aria.

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

5.1 Algoritmo di ricerca Best-first

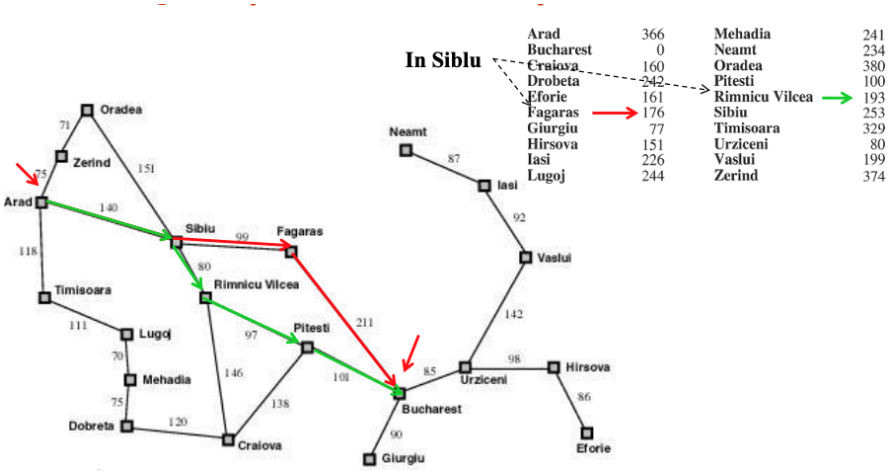
Il **best first - heuristic** usa lo stesso algoritmo di UC [3] ma con uso di f (stima di costo) per la coda con priorità. Una volta scelta f determina la strategia di ricerca. A ogni passo si sceglie il nodo sulla frontiera per cui il valore della f è migliore (il nodo più promettente).

Note 5.1.1. Migliore significa "minore" in caso di un'euristica che stima la distanza della soluzione

Un caso speciale: **greedy best-first**, su usa solo $h(f = h)$.

Esempio 5.1.1. Esempio di greedy best-first con $f = h$ Da Arad a Bucarest con **Greedy best-first**: Arad, sibiu, fagaras, bucharest (450) ma non è l'ottimale che sarebbe: Arad, Sibiu, Rimnicu, Pitesti, Bucarest (418).

[3] Warning: AIMA ed. IV ha usato uno schema di UC diverso e alcune proprietà cambiano



5.2 Algoritmo A

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

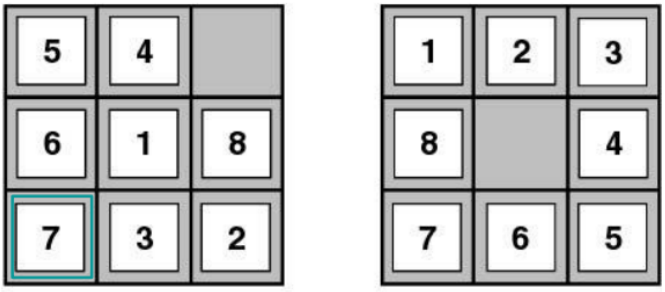
Definizione 5.2.1. *Un **Algoritmo A** è un algoritmo di best first con una fuzine di valutazione dello stato del tipo:*

$$f(n) = g(n) + h(n) \;\; con \;\; h(n) \geq 0 \;\; e \;\; h(goal) = 0$$

In questa definizione abbiamo che $g(n)$ è il costo del cammino percorso per raggiugnere n, mentre $h(n)$ una stima del costo per raggiungere da n un nodo goal (distanza).
Vedremo alcuni casi particolari dell'algoritmo A:

- Se $h(n) = 0[f(n) = g(n)]$ si ha **Ricerca Uniforme (UC)**.
- Se $g(n) = 0[f(n) = h(n)]$ si ha **Greedy Best First**.

Esempio 5.2.1. Esempio nel gioco dell'otto.



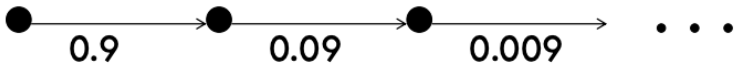
$f(n) =$ #mosse fatte $+$ #caselle-fuori-posto $\quad f(start) = 0 + 7 \quad f(goal - state) =? + 0$
Dopo $\leftarrow, \downarrow, \uparrow, \rightarrow$ abbiamo che $f = 4 + 7$, stesso stato, g è cambiato.

Teorema 5.2.1. L'algoritmo A con la condizione:

$$g(n) \geq d(n) \cdot \epsilon \quad (\epsilon > 0 \text{ costo minimo arco})$$

è completo.

Note 5.2.1. La conzione ci garantisce che non si verifichino situazioni strane del tipo: e quindi che il costo lungo un cammino non cresca "abbastanza" (se cresce abbastanza possiamo fermare quel path per costo alto di g).



Dimotrazione 5.2.1. 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: n' prima o poi sarà espanso. Infatti esistono solo un numero finito di nodi x che possono essere aggiunti alla frontiera con $f(x) \leq f(n')$ (è la condizione sulla crescita di g, scritta precedentemetne, tale che non esista una catena infinita di archi e nodi che possa aggiungere con costo sempre $\leq f(n')$).

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 nodo goal sarà selezionato per l'espanzione.

5.3 Algoritmo A^*

La funzione di valutazione ideale (oracolo):

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

Con $g*(n)$ il 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:

$$g(n) \geq g*(n) \quad e \quad h(n) \text{ è una stima di } h*(n)$$

($g(n) \geq g*(n)$ rappresenta costo cammino $\geq$ costo migliore). Si può andare in sottostima (e.g. linea d'aria) o sovrastima della distanza dalla soluzione.

Definizione 5.3.1 (Euristica ammissibile)**.**

$$\forall n t.c. h(n) \leq h^*(n) \quad h \text{ è una sottostima}$$

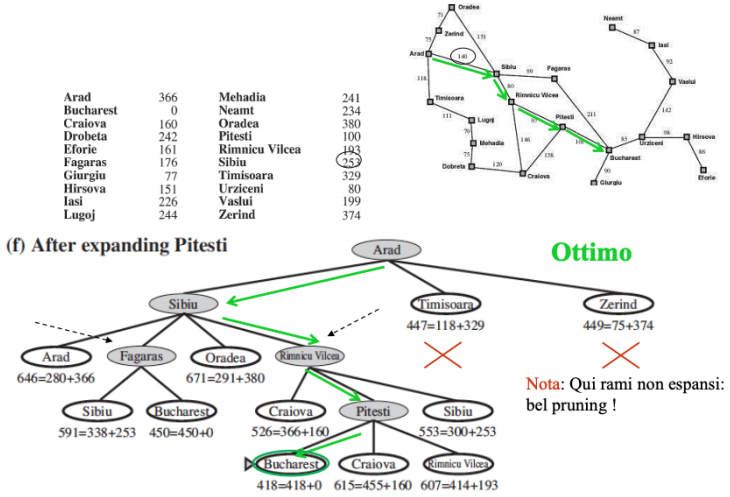
Esempio 5.3.1. L'euristica della distanza in linea d'aria.

Definizione 5.3.2 (Algoritmo A^*)**.** *Un algoritmo A in cui h è una funzione euristica ammissibile.*

Teorema 5.3.1. Gli algoritmi A^* sono **ottimali**.

Corollario 5.3.1.1. *$BF^{(+)}$ e UC sono ottimali ($h(n) = 0$).*

Esempio 5.3.2. Itinerario con A^* (ad albero).



Osservazione 5.3.1. Alcune osservazioni su A^*

1. Rispetto a greedy best-first, la componente g fa si che si abbandonino cammini che vanno troppo in profondità.
2. Ha sotto o sovra stima?
   (a) Una sottostima (h) può farci compiere del lavoro inutile (tenendo anche candidati non buoni), però non ci fa perdere il cammino migliore (quando prendo nodo goal è il cammino migliore).
   (b) Una funzione che qualche volta sovrastima può farci perdere la soluzione ottimale (taglio per causa di sovrastima, invece era buona)

5.3.1 Ottimalità su A^*

Nel caso di ricerca a/su albero l'uso di un'euristica ammissibile è sufficente a garantire l'ottimalità su A^*. Nel caso di ricerca su grafo (con UC come visto) serve una proprietà più forte: la **consistenza** (detta anche **monotonicità**).

Per evitare rischio di scartare candidati ottimi (stato già incontrato) ai vuol evitare, causa uso della lista esplorati, di far sparire, o meglio non considerare al momento dell'espansione, candidati ottimali. Cerchiamo quindi condizioni per garantire che il ptimo espanso sia il migliore.

Definizione 5.3.3. *Un euristica* ***consistente*** $[h(goal) = 0]$ *(consistenza locale).*

$$\forall n \text{ } t.c. \text{ } h(n) \leq c(n, a, n') + h(n') \text{ } dove \text{ } n' \text{ } è \text{ } un \text{ } successore \text{ } di \text{ } n$$

Ne segue che $f(n) \leq f(n')$

Note 5.3.1. Se h è consistente la f non decresce mai lungo i cammini, da cui il termine **monotona**.

Teorema 5.3.2. Un'euristica monotona è ammissibile.

Esistono euristiche ammissibili che non sono monotone, ma sono rare. Le euristiche monotone garantiscino che la soluzione meno costosa venga trovata per rima 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 allora posso evitare di inserire il corrente ripetuto senza perdere l'ottimalità.

```
if figlio.Stato non e in esplorati e non e in frontiera then
    frontiera = Inserisci(figlio, frontiera)
```

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

```
if figlio.Stato e in frontiera con Costo-cammino piu alto then
    sostituisci quel nodo frontiera con figlio
```

Andiamo ora a verificare l'ottimalità di A^* supponendo di avere il teorema, con h consistente.

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

   se $h(n) \leq c(n, a, n') + h(n')$ $\Rightarrow$ (def. consistenza sommando g(n)) $g(n) + h(n) \leq g(n) + c(n, a, n') + h(n')$

   ma siccome abbiamo $g(n) + c(n, a, n') = g(n')$ allora

   $$g(n) + h(n) \leq g(n') + h(n') \Rightarrow f(n) \leq f(n') \Rightarrow f \text{ monotona}$$

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 m ella frontiera sul cammino ottimo (a n, ancora da trovare con un cammino ottimo), con $f(m)$ minore (per la monotonia e n successore di m); ma ciò non è possibile perché tale nodo sarebbe già stato espanso (si espande prima un nodo con f minore).

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

Quindi, detto questo, perché A^* è vantaggioso?

- A^* espande tutti i nodi con $f(n) < C^*$ (C^* = costo ottimo)
- A^* espande alcuni nodi con $f(n) = C^*$.
- A^* **non espande alcun nodo con** $f(n) > C^*$

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

Più f è aderente a stima ottimale, più taglio! Ovali più stretti. Cercheremo quindi una h il più alta possibile tra le ammissibili Se molto bassa molti (sino a tutti i) nodi restano minore di $C^* \rightarrow$ espando tutti (a cerchi). Il pruning sotto-alberi è il punto focale: non li abbiamo già in memoria e evitiamo di generarli (decisivo per i problemi di AI a spazio stati esponenziali.

In riassunto L'algoritmo è quello degli schemi usati per UC, Usando f = g+h per la coda con priorità, ove h e g soddisfano quanto allo slide 9 [A], ove h è una funzione euristica ammissibile [A^*], e considerando le condizioni dette per ottenere l'ottimalità su grafi.

- A^* è **completo**: discende dalla completezza di A (A^* è un algoritmo A particolare).
- A^* con euristica monotona è **ottimale**.
- A^* è **ottimamente efficiente**: a parità di euristica nessn altro algoritmo espande meno nodi (senza rinunciare a ottimalità)

I problemi principali sono la scelta dell'euristica e ancora l'occupazione di memoria che nel caso pessimo resta esponenziale come visto per gli altri algoritmi di ricerca con stesso schema, causa frontiera.

5.4 Sotto-casi speciali: US e Greedy Best First

Ci sono due casi particolai dell'algoritmo A:

1. Se $h(n) = 0$ $[f(n) = g(n)]$ si ha Uniform Cost (UC), ossia g non basta (si può migliorare).
2. Se $g(n) = 0$ $[f(n) = h(n)]$ si ha Greedy Best Fist, ossia h non basta (già visto all'inizio).

5.4.1 UC vs A^*

5.5 Costruire le euristiche di A^*

Partiamo dalle valutazione di funzioni euristiche. A parità di ammissibilità, una euristica può essere più efficiente di un'altra nel trovare il cammino soluzione migliore (visitare meno nodi). Questo dipende da quanto informata è l'euristia (dal **grado di informazione posseduto**)

- $h(n) = 0$ minimo di informazione (BF o UC)
- $h^*(n)$ massimo di infomrazione (oracolo)

In generale, per le euristiche ammissibili:

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

Teorema 5.5.1. Se $h_1 \leq h_2$, i nodi espansi[4] da A^* con h_2 sono un sottoinsieme di quelli espandi da A^* con h_1.

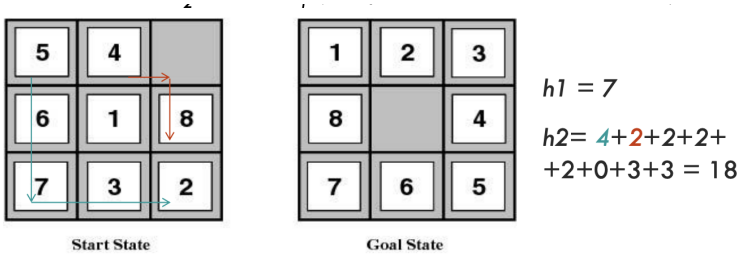
Se $h_1 \leq h_2$, A^* con h_2 è almeno efficiente quanto A^* con h_1. Un'euristica più informata (accurata) riduce lo spazio di ricerca (è più efficiente), ma è tipicamente più costosa da calcolare (e.g. un caso estremo ?)

[4] Ricorda che A^* espande tutti i nodi con $f(n) = g(n) + h(n) < C^*$, e sono meno per h maggiore (h maggiore fa andare più nodi oltre C^*).

Esempio 5.5.1. Due euristiche ammissibili per il gioco dell'8 potrebbero essere le seguenti:

- h_1: conta il numero di caselle fuori posto
- h_2: somma delle distanze **Manhattan** (orizzonatle/verticale) delle caselle fuori posto dalla posizione finale.

h_2 è più informata di h_1, infatti $\forall n \ . \ h_1(n) \leq h_2(n)$, quindi si dice che h_2 **domina** h_1 (utile per confrontte tra ammissibili)



Definizione 5.5.1. *La somma delle distanze Manghattan si definisce come:*

$$h((x,y)) = MD((x,y),(x_g,y_g)) = |x - x_g| + |y - y_g|$$

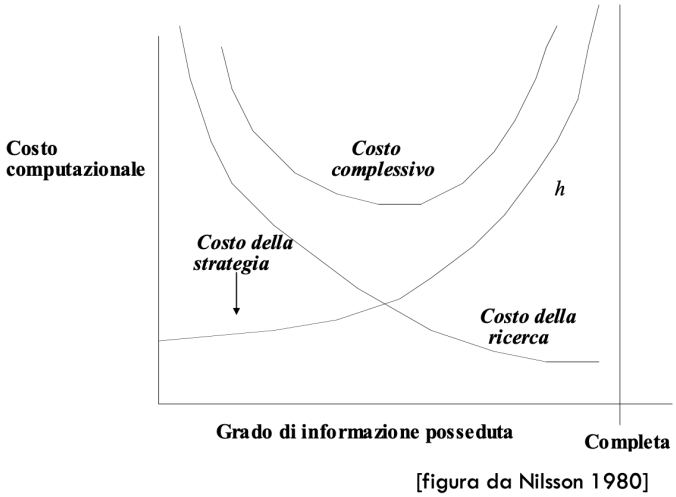


Figure 3: Costo ricerca vs costo euristico

Ora capiamo come valutare gli algoritmi di ricerca euristica. Introfuciamo il **fattore di diramazione effettivo** b^*, N: numero di nodi generati, d: profondità della soluzione. b^* è il fattore di diramazione di un albero uniforme con $N+1$ nodi, solizione dell'equazione:

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

Sperimentalmente una buona euristica ha un b^* abbastanza vicino a 1 (¡1.5)

Esempio 5.5.2. Ricodando l'esempio dal gioco dell'otto: Sono riportati: Nodi generati e fattore di diramazione effettivo (b^*, verde) I dati sono mediati, per ogni d, su 100 istanze del problema [AIMA].

Nella **capacità di esplorazione**, l'influenza di b^*:

- Con b=2: d=6 e N = 100 d=12 e N = 10.0000
- Con b=1.5: d=12 N = 100 d=24 e N = 10.000

d	ID (appr. it. non inf)	A*(h1)	A*(h2)
2	10 (2,43)	6 (1,79)	6 (1,79)
4	112 (2,87)	13 (1,48)	12 (1,45)
6	680 (2,73)	20 (1,34)	18 (1,30)
8	6384 (2,80)	39 (1,33)	25 (1,24)
10	47127 (2,79)	93 (1,38)	39 (1,22)
12	3644035 (2,78)	227 (1,42)	73 (1,24)
14	Nodi generati b*	539 (1,44)	113 (1,23)
...	-	...	...

migliorando di poco l'euristica si riesce, a parità di nodi espansi, a raggiungere una profondità doppia di esplorazione mosse!

Quindi abbiamo che:

1. Tutti i problemi dell'IA (o quasi) sono di complessità esponenziale ... (nel generare nodi, i.e. configurazioni possibili) ma c'è esponenziale e esponenziale!
2. L'euristica può migliorare di molto la capacità di esplorazione dello spazio degli stati rispetto alla ricerca cieca.
3. Migliorando anche di poco l'euristica si riesce ad esplorare uno spazio molto più grande (più in profondità).

Per inventare un euristica ci sono alcune strategie, che aiutano appunto ad ottenre euristiche amissibili: Rilassamento del problema, Massimizzazione di euristiche, Database di pattern disgiunti, Combinazione lineare, Apprendere dall'esperienza.

Esempio 5.5.3. Il rilassamento del problema Nel gioco dell'8 mossa da A a B possibile se: **1. B adiacente a A**, **2. B libera**.
h_1 e h_2 sono calcoli distanza esatta della soluzione in versioni semplificate del puzzle:

- h_1 (nessuna restrizione, ne 1 ne 2): sono sempre ammessi scambi a piacimento tra caselle (si muove ovunque) $\rightarrow$ numero caselle fuori posto.
- h_2 (solo restrizione 1): sono ammessi spostamenti anche su caselle occupate, purché adiacenti $\rightarrow$ somma delle distanze Manhattan.

Se si hanno una serie di euristiche ammissibili $h_1, h_2, \ldots, h_k$ **senza che nessuna "domini" un'altra** allora conviene prendere il massimo dei lori valori:

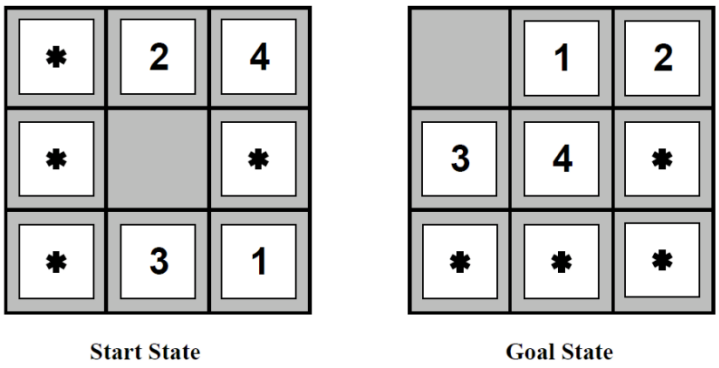
$$h(n) = max(h_1(n), h_2(n), \ldots, h_k(n))$$

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

5.6 Euristiche da sottoproblemi

Costo della soluzione ottima al sottoproblema (di sistemare 1,2,3,4) è una sottostima del costo per il problema nel suo complesso (e.g. rilevatesi più accurata della Manhattan). **Database di pattern**: memorizzare ogni istanza del sottoproblema con relativo costo della soluzione. Usare poi questo database per calcolare hDB (estraendo dal DB la configurazione corrispondente allo stato completo corrente).

Potremmo poi fare la stessa cosa per altri sottoproblemi: 5-6-7-8, 2-4-6-8, ottenendo altre euristiche ammissibili, poi prendere il valore massimo: ancora una euristica ammissibile. Ma potremmo sommarle e ottenere un'euristica ancora più accurata?



In generale no perchè le soluzioni ai sottoproblemi interferiscono (condividono alcune mosse, se sposto 1-2-3-4, spostero anche 4-5-6-7) e la somma delle euristiche in generale non è ammissibile (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) [e.g. solo costo mosse su1-2-3-4], sono molto efficaci: gioco del 15 in pochi ms ma per esempio difficile scomporre per cubo Rubik.

Bisogna eseguire un apreddimento dall'esperienza. Quindi far girare il programma, raccogliere dati: coppie $< stato, h^* >$. Usare i dati per apprendere a 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, xi) [e.g. apprendiamo che da numero tasselli fuori posto 5 $\rightarrow$ costo 14, etc].

Quando diverse caratteristiche influenzano la bontà di uno stato, si può usare una combinazione lineare per combinare le euristiche:

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

Esempio 5.6.1. Gioco dell'8: $h(n) = c_1$ #fuori-posto $+ c_2$#coppie-scambiate
Scacchi: $h(n) = c_1$ vant-pezzi $+ c_2$ pezzi-attacc. $+ c_3$ regina $+ \ldots$

Il peso dei coefficienti può essere aggiustato con l'esperienza, anche qui apprendendo automaticamente da esempi di gioco. $h(goal) = 0$ (e.g. gioco dell'8) ma ammissibilità e consistenza non automatiche.

5.7 Algoritmi evoluti basati su A^*

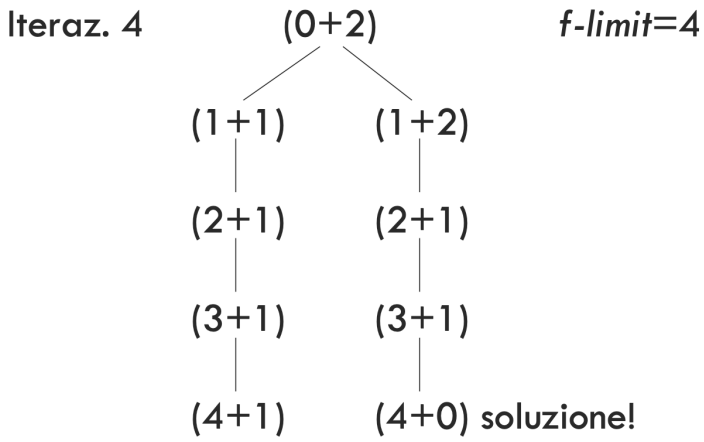
Ci sono una serie di algoritmi basati su A^* che possono andare a portare ad un miglioramento dell'occupazione della memoria. Fra questi abbiamo: Beam search, A* con approfondimento iterativo (IDA*), ricerca best-first ricorsiva (RBFS), A* con memoria limitata (MA*) in versione semplice (SMA*).

5.7.1 Beam search

Nel Best First viene tenuta tutta la frontiera; se l'occupazione di memoria è eccessiva si può ricorrere ad una variante: la Beam search. La Beam Search tiene ad ogni passo solo i k nodi più promettenti, dove k è detto l'ampiezza del raggio (beam). La Beam Search non è completa.

5.7.2 IDA^*

L'IDA^* è un 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.



Esempio 5.7.1.

Cruciale la scelta dell'incremento per garantire l'ottimalità:

- Nel caso di costo delle azioni fisso è chiaro: il limite viene incrementato del costo delle azioni.
- Nel caso che i costi delle azioni siano variabili? O costo minimo, oppure si potrebbe ad ogni passo fissare il limite successivo al valore minimo delle f scartate (in quanto superavano il limite) all'iterazione precedente.

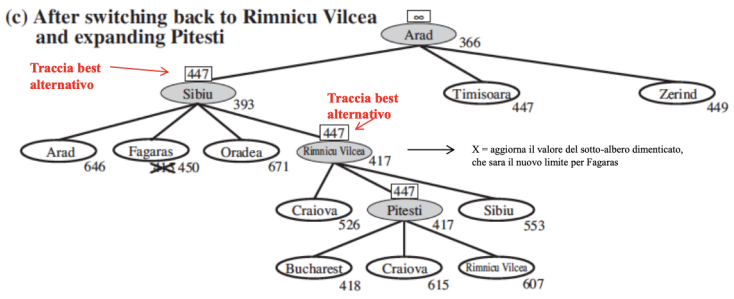
IDA^* è sia completo che ottimale.

- Se le azioni hanno costo costante k (caso tipico 1) e f-limit viene incrementato di k.
- Se le azioni hanno costo variabile e l'incremento di f-limit è $\leq \epsilon$ (minimo costo degli archi).
- Se il nuovo f-limit = min. valore f dei nodi generati ed esclusi all'iterazione precedente.

L'occupazione di memoria è $O(bd)$.

5.7.3 Best-first ricorsivo (BRFS)

Simile a DF ricorsivo: cerca di usare meno memoria, facendo del lavoro in più. Tiene traccia ad ogni livello del **migliore percorso alternativo**. Invece di fare backtracking in caso di fallimento (DF si ferma solo in fondo) 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 profondita delle sol. ottima.



Esempio 5.7.2.

```
    function Ricerca-Best-First-Ricorsiva(problema)
        returns soluzione oppure fallimento
        return RBFS(problema, CreaNodo(problema.Stato-iniziale), infinito) // all inizio
            f-limite e un valore molto grande

function RBFS (problema, nodo, f-limite)
    returns soluzione oppure fallimento e un nuovo limite all f-costo // restituisce due
        valori
    if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
    successori = [ ]

    for each azione in problema.Azioni(nodo.Stato) do
        aggiungi Nodo-Figlio(problema, nodo, azione) a successori // genera i successori
    if successori vuoto then return fallimento, infinito

    for each s in successori do // valuta i successori
        s.f = max(s.g + s.h, nodo.f) // un modo per rendere monotona f
    loop do
        migliore = il nodo con f minimo tra i successori
        if migliore.f > f_limite then return fallimento, migliore.f
        alternativa = il secondo nodo con f minimo tra i successori
        risultato, migliore.f = RBFS(problema, migliore, min(f_limite, alternativa))
        if risultato != fallimento then return risultato
```

5.7.4 A^* con memoria limitata (versione semplice)

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 conclusione in algoritmi a memoria limitata (IDA* e SMA*) le limitazioni della memoria possono portare a compiere molto lavoro inutile [esp. ripetuta stessi nodi]. Difficile stimare la complessità temporale effettiva. Le limitazioni di memoria possono rendere un problema intrattabile dal punto di vista computazionale.

6 Agenti basati su coscenza

Abbiamo già visto agenti con stato e con obiettivo in mondi osservabili con stati atomici e azioni descrivibili in maniera semplice, mettendo enfasi sul processo di ricerca.

Ora andremo ad parlare di come migliorare le **capacità di ragionamento** degli agenti, dotandoli di rappresentazioni di mondi complessi e astratti, non descrivibili semplicemente. Agenti **basati su conoscenza**, dotati di una KB (**Knowledge Base**) con conoscenza espressa in maniera esplicita e dichiarativa.

6.1 Introduzione

La maggior parte dei problemi di I.A. sono "knowledge intensive". Il mondo è tipicamente complesso: ci serve una rappresentazione parziale e incompleta (un'astrazione) del mondo funzionale 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 ma anche estratta dai testi o appresa dall'esperienza o estratta/elicitata dagli esperti.

La KB racchiude tutta la conoscenza necessaria a decidere l'azione da compiere in forma **dichiarativa**. Un agente basato su conoscenza può essere costruito dicendogli (TELL) ciò che deve sapere. SI inizia con una base di conoscneza vuota si aggiungono progressivamente formule alla base di conoscneza, una alla volta.

L'alternativa (approccio procedurale) è scrivere un programma che implementa il processo decisionale, una volta per tutte. Un agente KB è più flessibile: più semplice acquisire conoscenza incrementalmente e modificare il comportamento con l'esperienza.

Esempio 6.1.1. Il mondo del Wumpus. Il mondo del Wumpus è una caverna fatta di stanze connesse tra di loro da passaggi. Il wumpus è una bestia puzzolente che mangia chiunque entri nella stanza in cui si trova. Il wumpus può essere ucciso dall'agente, che ha solo una freccia a disposizione.

- Ci sono stanze con dei pozzi: se l'agente entra in una di queste stanze cade nel pozzo e muore. Il Wumpus non muore nel pozzo.
- In una delle stanze si trova l'oro e l'obiettivo dell'agente è di trovare l'oro e tornare a casa con l'oro, sano e salvo.
- L'agente non conosce l'ambiente, né la sua locazione. Solo all'inizio sa dove si trova (in [1,1]).

Abbiamo poi delle misure di prestazioni:

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

L'**ambiente** è strutturato in una griglia 4 x 4 di stanze, circondata da pareti di delimitazione. L'agente comincia sempre dalla posizione [1,1], rivolto verso est (dx) ([1,1] è safe). Le posizioni dell'oro e del Wumpus sono scelte casualmente tra tutti I riquadri tranne quello iniziale. Tutti I riquadri (tranne quello iniziale) hanno una probabilita' 0.2 di contenere un pozzo.

Le **azioni** disponibili sono: avanti, a destra di 90°, a sinistra di 90°, afferra un oggetto, scaglia la freccia (solo una), esce. Le cose che vengono **percepite** (sesori) sono:

- fetore nelle caselle adiacenti al Wumpus.
- brezza nelle caselle adiacenti ai pozzi.
- Luccichio nelle caselle con l'oro.

- Bump se sbatte in un muto
- Urlo se il wumpus viene ucciso.
- L'agente NON percepisce la sua locaizone.

6.2 Agenti basati sulla conoscenza

Un agente basato su conoscenza mantiene una **base di conoscenza** (KB): un insieme di **enunciati** (formule) espressi in un linguaggio di rappresentazione. Interagisce con la KB mediante una interfaccia funzionale Tell-Ask:

- Tell: per aggiungere nuovi enunciati a KB.
- Ask: per interrogare la KB.
- Retract: per eliminare enunciati.

Gli enunciati nella KB rappresentano le **opinioni/credenze dell'agente**. Le risposte α devono essere tali che α **discende necessariamente** dalla KB.

Il problema: data una base di conoscenza KB, contenente una rappresentazione dei fatti che si **ritengono veri**, come dedurre che un certo fatto a è vero di conseguenza?

$$KB| = \alpha \quad \text{conseguenza logica}$$

```
//Prende in input una percezione e restituisce un azione
function Agente-KB (percezione) returns un azione
    //Mantiene in memoria una base di conoscenza KB che puo contenere una conoscenza
        iniziale.
    persistent: KB, una base di conoscenza
    t, un contatore, inizialmente a 0, che indica il tempo

    // Comunica la percezione all KB
    TELL(KB, Costruisci-Formula-Percezione(percezione, t ))

    // Chiede alla KB quale azioni deve compiere
    azione <- ASK(KB, Costruisci-Query-Azione( t ))

    // Comunica alla KB che l azione e stata compiuta al tempo t
    TELL(KB, Costruisci-Formula-Azione(azione, t ))
    t <- t + 1
    return azione
```

Base di conoscenza vs base di dati:

- **base di dati**: solo fatti specifici, solo recupero (retrieval).
- **Base di conoscenza**: una rappresentazione esplicita, parziale e compatta, in un linguagio simbolico, che contiene: fatti di timpo specifico (casella [1,1] ok, c'è un posso in [3,1]), fatti di tipo generale, o regole (c'è brezza nella caselle adiacenti ai pozzi).

Qullo che caratterizza una KB è la capacità inferenziale, derivare nuovi fatti da quelli memorizzati esplicitamente (es. c'è un pozzo in [3,1] il wumpus è in [1,3]).
Sfortunatamente pià il linguaggio è **espressivo**, meno **efficiente** è il meccanismo inferenziale. § Il problema 'fondamentale' in R.C. è trovare il giusto compromesso tra: **espressività** del linguaggio di rappresentazione, **complessità** del meccanismo inferenziale.
Questi due obiettivi sono in contrasto e si tratta di mediare tra queste due esisgenze.

6.3 Logica

Le basi di conoscenza sono costituite da enunciati (formule). Le formule sono espresse secondo le regole della sintassi, che specifica quali di esse sono "ben formate". La semantica di una formula esprime il "significato" della formula. Un modello e' una configurazione dei valori di verità che si possono assegnare alle variabili convolte in una formula.

Un formalismo per la rappresentazione della conoscneza ha tre componenti:

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

7 Calcolo proposizionale

7.1 Logica Proposizionale

Definizione 7.1.1 (Sintassi)**.** *La sintassi definisce quali sono le frasi leggittime (ben formate) del linguaggio.*

$$\begin{aligned} formula &\longrightarrow formulaAtomica \mid formulacomplessa \\ formulaAtomica &\longrightarrow True \mid False \mid simbolo \\ simbolo &\longrightarrow P \mid Q \mid R \mid \ldots \\ formulaComplessa &\longrightarrow \neg formula \text{ not (negazione)} \\ &\mid (formula \wedge formula) \text{ and (congiuzione)} \\ &\mid (formula \vee formula) \text{ or (disgiunzione)} \\ &\mid (formula \Rightarrow formula) \text{ implicazione} \\ &\mid (formula \Leftrightarrow formula) \text{ se e solo se} \end{aligned}$$

Esempio 7.1.1. $((A \wedge B) \Rightarrow C)$. Possiamo oettere le parentesi assumendo questa precedenza tra gli operatori:

$$\neg \ > \ \wedge \ > \ \vee \ > \Rightarrow, \Leftrightarrow$$

$\neg P \wedge Q \vee R \Rightarrow S$ è la stessa cosa di $(((\neg P) \vee (Q \wedge R)) \Rightarrow)$. Per esempio dal mondo del Wumpus $P_{1,1}$ c'è il posso in $[1,1]$, $W_{2,3}$ il wumpus è in $[2,3]$.

Definizione 7.1.2 (Semantica)**.** *La semantica specifica le regole usate per determinare il valore di verità di una formula nei confronti di un particolare modello.*

Nella logica proposizionale, un modello specifica il valore di verità (True o False) di ogni simbolo proposizionale.

8 Ricerca locale

Gli agenti risolutori di problemi "classici" assumono: ambienti completamente osservabili, ambienti deterministici, sono nelle condizioni di produrre offline un piano (una sequenza di azioni) che può essere eseguito senza imprevisti per raggiungere l'obiettivo.

La ricerca sistematica, o anche euristica, nello spazio di stati è troppo costosa. Inoltre spesso le assunzioni sull'ambiente sono da riconsiderare infatti in ambienti realistizi le azioni sono non deterministiche e ambiente parzialmente osservabile oppure abbiamo addirittura ambienti sconosciuti e probemi di esplorazione (ess. ricerca online).

Per effetturare una **ricerca locale** bisogna fare alcune assunzione. Gli algoritmi visti esplorano gli spazi di ricerca 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:

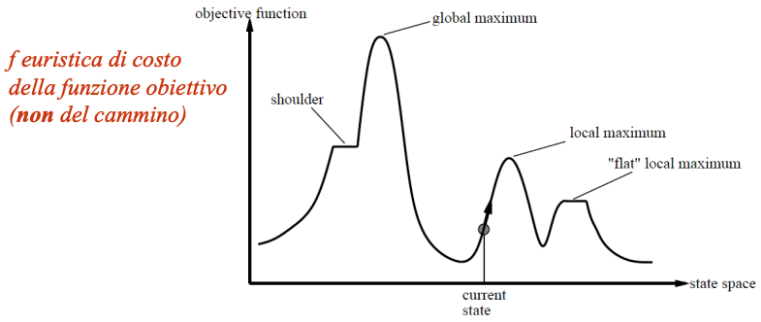
- La sequenza di azioni non è importante: quello che conta è unicamente lo stato goal.
- Tutti gli elementi della soluzione sono nello stato ma alcuni vincoli sono violati (Es. le regine nella formulazione a stato completo). Ci interessa di ottenere la soluzione ma non il path

Gli algoritmi di ricerca locale inoltre non sono sistematici, tengono traccia solo del nodo corrente e si spostano su nodi adiacenti. Non tengono traccia dei cammini (non servono in unscita!).

1. Efficienti in occupazione di memoria.
2. Possono trovare soluzioni ragionevoli anche in spazi molto grandi e inifini, come nel caso di spazi continui.

Sono molto utili per risolvere probelmi di ottimazzione: lo stato migliore secondo una funzione obiettivo (f), lo stato di costo minore (ma non il path).

Esempio 8.0.1. Minimizare numero di regine sotto attacco (f all'obiettivo?) oppure training di un modello di Machine Learning.



Uno stato ha una posizione sulla superficie e una altezza che corrisponde al valore della f. di valutazione (f. obiettivo). Un algoritmo provoca movimento sulla superficie. Trovare l'avvallamento più basso (e.g. min costo) o il picco più alto (e.g. max di un obiettivo).

8.1 Ricerca in salita (Hill climbing)

È una ricerca locale di tipo **greedy**. Vengono generati i successori e valutati; viene scelto un nodo che migliora la valutazione dello stato attuale (non si tiene traccia degli altri [no albero di ricerca in memoria]).

- Il migliore fra i sucessori è **Hill climbing a salita rapida/ripida**.
- Uno a caso (tra quelli che salgono) si dice **Hill climbing stocastico** (anche dipendendo da pendenza)

- Mentre il primo è il **Hill climbing con prima scelta** (il primo generato tra tanti possibili)

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

```
function Hill-climbing (problema)
    returns uno stato che e un massimo locale [esercizio: fare con min.]
    nodo-corrente = CreaNodo(problema.Stato-iniziale)

    loop do
        vicino = il successore di nodo-corrente di valore piu alto
        if vicino.Valore <= nodo-corrente.Valore then
            return nodo-corrente.Stato // interrompe la ricerca
        nodo-corrente = vicino
        // (altrimenti, se vicino e migliore, continua)
```

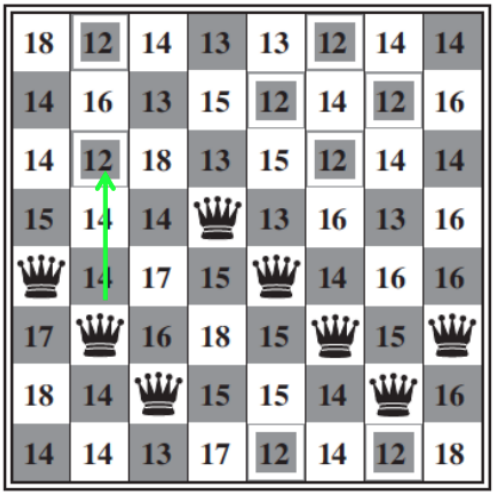
Note 8.1.1. Si prosegue solo se il vicino (più alto) è migliore dello stato corrente → se tutti i vicini sono peggiori o uguali si ferma.

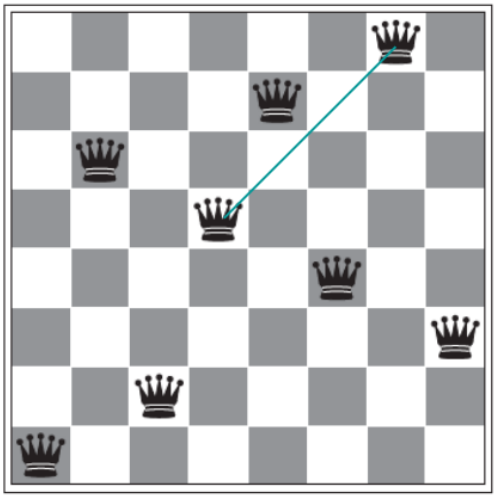
Non c'è frontiera a cui ritornare, si tiene 1 solo stato. Il tempo si calcolo come numero di cicli variabile in base al punto di partenza. Il codice in python è il seguente:

```python
def hill_climbing(problem): """ Ricerca locale - Hill-climbing."""
    current =Node(problem.initial_state)
    while True:
        neighbors =[current.child_node(problem, action) for action in
                    problem.actions(current.state)]
    if not neighbors: # se current non ha successori esci e restituisci current
        break

    # scegli il vicino con valore piu' alto (sulla funzione problem.value)
    neighbor =(sorted(neighbors,key =lambda x:problem.value(x), reverse =True))[0]
    if problem.value(neighbor) <=problem.value(current):
        break
    else:
        current =neighbor # (altrimenti, se vicino e migliore, continua)
    return current
```

Esempio 8.1.1. Il problema delle 8 regine. Costo h (stima euristica del costo f): numero di coppie di regine che si attaccano a vicenda (nell'esempio valore 17). Si cerca il minimo, i numeri sono i valori dei successori (7x8) [7 posizioni per ogni regina, su ogni colonna], tra i migliori (di pari valore 12) si sceglie a caso, i imposta anche il minimo globale a 0. Vediamo poi nella seconda figura un esempio

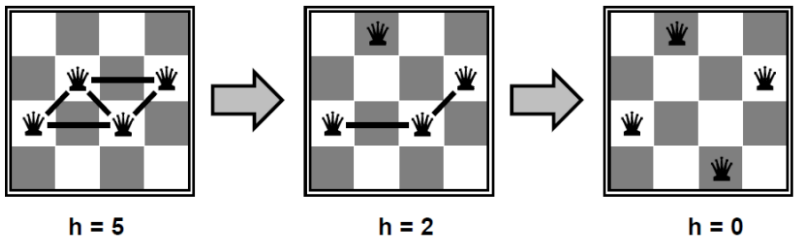




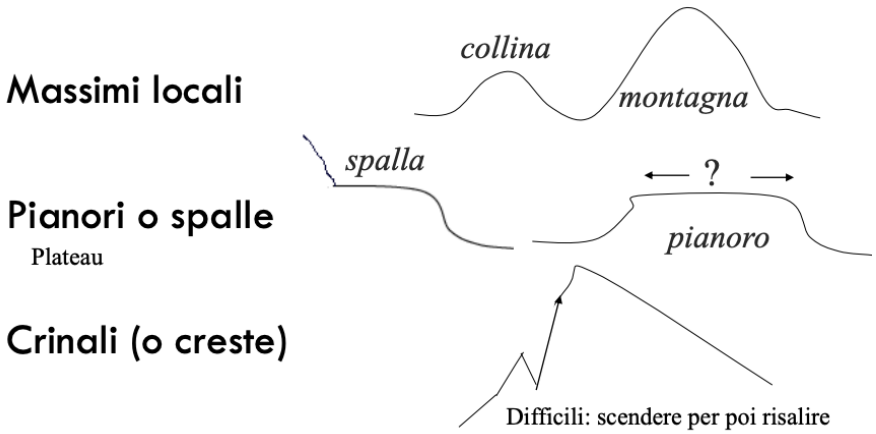
negativo con minimo locale. Settiamo $h = 1$, tutti gli stati successori non migliorano la situazione.

Per le 8 regine Hill-climbing si blocca l'86% delle volte, ma in media solo 4 passi per la soluzione e 3 quando si blocca, su $8^8 = 16.8$ milioni di stati.

Esempio 8.1.2. Sempre sulla stessa base dell'esempio sopra questo è un esempio positivo, con successo in tre mosse. h qui è l'euristica di costo della funzione obiettico (da minimizzare).



Alcuni problemi con Hill-climbing ci sono se la f è da ottimizzare, infatti in questo caso i picchi sono massimi locali o soluzioni ottimali.



Alcuni miglioramenti che si possono apportare solo i seguenti:

1. Consentire (un numero limitato di) mosse laterali (ossia ci si ferma per ¡ nell'alg. invece che per $\leq\rightarrow$ continua anche a parità di h). L'algoritmo sulle 8 regine ha successo nel 94%, ma impiega in media 21 passi.

2. Hill-climbing stocastico: si sceglie a caso tra le mosse in salita (magari tenendo conto della pendenza). Converge più lentamente ma a volte trova soluzioni migliori.

3. Hill-climbing con prima scelta. Può generare le mosse a caso, uno alla volta, fino a trovarne una migliore dello stato corrente (si prende solo il primo che migliora). Come la stocastica ma utile quando i successori sono molti (e.g. migliaia o ben oltre), evitando una scelta tra tutti.

4. Hill-Climbing con riavvio casuale (random restart): ripartire da un punto scelto a caso. Se la probabilità di successo è p saranno necessarie in media 1/p ripartenze per trovare la soluzione (es. 8 regine, $p = 0.14 \rightarrow 7$ ripartenze [1/p] per avere 6 fail e un successo). Per le regine: caso con 3 milioni di regine in meno di un minuto! Se funziona o no dipende molto dalla forma del panorama degli stati (molti min loc. abbassano p, si blocca spesso)

8.2 Tempra simulata

L' algoritmo di tempra simulata (Simulated annealing) [Kirkpatrick, Gelatt, Vecchi 1983] combina hill-climbing con una scelta stocastica (ma non del tutto casuale, perché poco efficiente...). Analogia con il processo di tempra dei metalli in metallurgia. I metalli vengono portati a temperature molto elevate (alta energia/stocasticità iniziale) e raffreddati gradualmente consentendo di cristallizzare in

uno stato a (più) bassa energia. (esempio di cross-fertilization tra aree scientifche diverse).

In questo algoritmo ad ogni passo si sceglie un successore n' a caso:

- se migliora lo stato corrente viene espanso.
- se no (caso in cui $\Delta E = f(n') - f(n) \leq 0$) quel nodo viene scelto con probabilità $p = e^{\Delta E/T}[0 \leq p \leq 1]$ [Si genera un numero casuale tra 0 e 1: se questo è ¡ p il successore viene scelto, altrimenti no].

Ossia: p è inversamente proporzionale al peggioramento Infatti se la mossa peggiora molto, ΔE alto neg., la p si abbassa. T (temperatura) decresce col progredire dell'algoritmo (quindi anche p) secondo un piano definito. Col progredire rende improbabili le mosse peggiorative.

Per quanto riguarda l'analisi della tempra simulata, La probabilità p di una mossa in discesa diminuisce col tempo e l'algoritmo si comporta sempre di più come Hill Climbing. Se T viene decrementato abbastanza lentamente con prob. tendentente ad 1 si raggiunge la soluzione ottimale. Analogia col processo di tempra dei metalli T corrisponde alla temperatura ΔE alla variazione di energia.

I paramentri da inserire sono il valore iniziale e decremento di T, ed i valori per T determinati sperimentalmente: il valore iniziale di T è tale che per valori medi di $\Delta E, p = e^{\Delta E/T}$ sia all'incirca 0.5.

8.3 Ricerca local beam

La versione locale della beam search. Si tengono in memoria k stati, anziché uno solo ed ad ogni passo si generano i successori di tutti i k stati Se si trova un goal ci si ferma, altrimenti si prosegue con i k migliori tra questi

Note 8.3.1. Diverso da K restart (che riparte da zero), diverso anche da beam search

Si introduce un elemento di casualità, come in un processo di selezione naturale (diversificare la nuova generazione). Nella **variante stocastica** della local beam, si scelgono k successori, ma con probabilità maggiore per i migliori. Tra le terminolgie usate abbiamo: organismo [stato], progenie [successori], fitness [il valore della f], idoneità.

8.4 Algoritmi generici/evolutivi

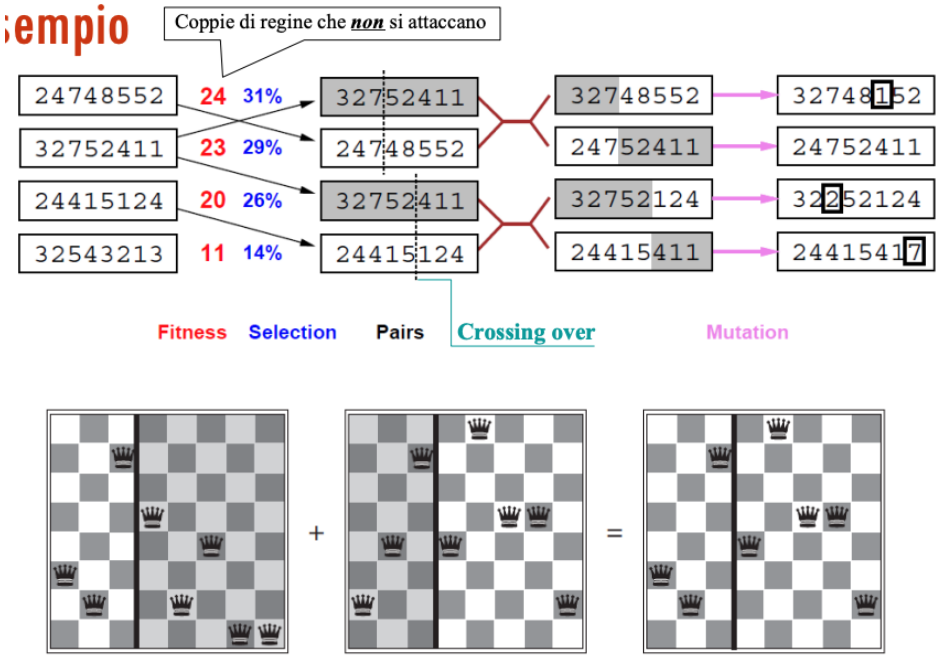
Sono varianti della beam search stocastica in cui gli stati successori sono ottenuti combinando due stati genitore (anziché per evoluzione). Un po' di terminologia che si usera: **popolazione di individui** [stati], **fitness**, **accoppiamenti** + **mutazione genetica**, **generazioni**.

Allora per spiegare il funziomento vediamo insanzitutto che la **popolazione** iniziale è formata da:

- k stati/**individui** generati casualmente.
- ogni individuo è rappresentato come una stringa (esempio: posizione nelle colonne ("24748552") stato delle 8 regine o con 24 bit*).

A questo punto gli individui sono valutati da una **funzione di fitness** (Esempio: n. di coppie di regine che non si attaccano).
Poi si **selezionano** gli individui per gli "**accoppiamenti**" con una probabilità proporzionale alla fitness. Le coppie danno vita alla **generazione** successiva combinando materiale genetico (crossover) con un meccanismo aggiuntivo di mutazione genetica (casuale). La popolazione ottenuta dovrebbe essere migliore. La cosa si ripete fino ad ottenere stati abbastanza buoni (stati obiettivo) o finche non miglioriamo più.

Esempio 8.4.1. Per ogni coppia (scelta con probabilità(blu) proporzionale alla fitness(rosso)) viene scelto un punto di crossing over(verde) e vengono generati due figli scambiandosi pezzi (del DNA) Viene infine effettuata una mutazione(rosa) casuale che dà luogo alla prossima generazione La fitness progressivamente tenderà a favorire generazioni migliori. Emula i meccanismi genetici ma anche l'evoluzione della specie. La nascita di un figlio aviene come



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

In conclusione gli algoritmi genetici sono suggestivi (area del Natural computing: e.g. swarm, ...). Usati in molti problemi reali e.g. configurazione di circuiti e scheduling di lavori, fra i loro vantaggi abbiamo che combinano: tendenza a salire della beam search stocastica, interscambio info (indirettamente) tra thread paralleli di ricerca (blocchi utili che si combinano). Funziona meglio se il problema (soluzioni) ha componenti significative rappresentate in sottostringhe. Il maggior punto critico è la rappresentazione del problema in stringhe.

8.5 Spazi continui

Molti casi reali hanno spazi di ricerca continua e.g. fondamentale per Machine Learning! Stato descritto da variabili continue $x_1, \ldots, x_n$, vettore x. Prendiamo ad esemio movimenti in spazio 3D, con posizione data da $x = (x_1, x_2, x_3)$, apparentemente ostico, fattori di ramificazione infiniti con gli approcci precedenti, in realtà molti strumenti matematici per spazi continui, che portano ad approcci anche molto efficienti...

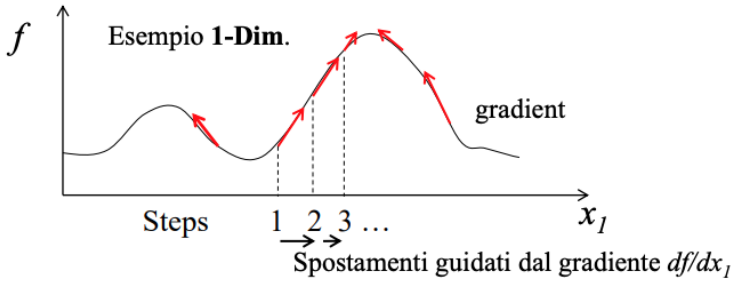
Se la f è continua e differenziabile, e.g. quadratica rispetto a x (vettore). Il minimo o massimo si può cercare utilizzando il **gradiente**, che restituisce la direzione di massima pendenza nel punto. Data f obiettivo su 3D

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

Nell Hill climbing iterativo: $x_{new} = x + \eta \nabla f(n)$ Dove il '+' per salive (maximization), il '-' per scendere (minimization), la η indica lo step size (e.g constante positiva ¡1) Quantifica direzione e spostamento, senza cercarlo tra gli infiniti possibili successori!

Note 8.5.1. non sempre è necessaio il min/max assoluto: vedremo nel ML

Esempio 8.5.1. Discesa verso il minimo. $f(x) = x^2 \quad f'(x) = 2x$
Discesa di gradiente verso il minimo. $x_{new} = x - \eta \nabla f(x) \rightarrow \ (1-d) \ \rightarrow x_{new} = x - \eta f'(x)$.
Mettiamoci in $x = 2$, la derivata vale $2 \times 2 = 4$, mi devo spostare di $-\eta f'(x) = -\eta 4$, quindi ad esempio ($\eta = 0.2$) ottengo $-\eta f'(x) = -0.2 \times 4 = -0.8$ andando a sinistra al punto $x_{new} = 2 - 0.8 = 1.2$ avvicinandomi quindi al minimo.



8.6 Assunzioni sull'ambiente da considerare

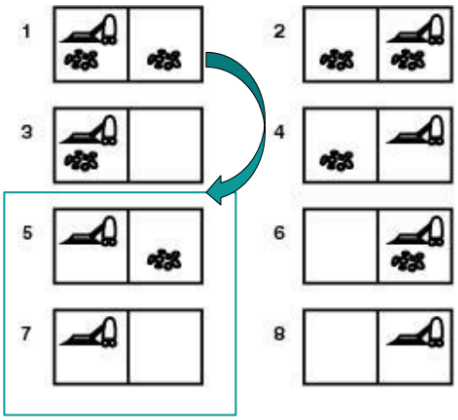
Negli ambienti più realistiti gli agenti risolutori di problemi "classici" assumono:

- Ambienti completamente osservabili.
- Azioni/ambienti deterministici.
- Il piano generato è una sequenza di azioni che può essere generato offline e eseguito senza imprevisti.
- Le percezioni non servono se non nello stato iniziale.

In un ambiente parzialmente osservabile e non deterministico le **percezioni** sono importanti: restringono gli stati possibili, informano sull'effetto dell'azione. Più che un piano l'agente può elaborare una "strategia", che tiene conto delle diverse eventualità: un **piano con contigenza**.

Esempio 8.6.1. L'aspirapolvere con assunzioni diverse. L'spirapolvere imprevedibile: ci sono piùstatipossibili risulato dell'azione.

- Comportamento: Se aspira in una stanza sporca, la pulisce maa talvolta pulisce anche una stanza adiacente. Se aspira in una stanza pulita, a volte rilascia sporco.
- Variazioni necessarei al modello: Il modello di transizione restituisce un **insieme di stati**: l'agente non sa in quale si troverà. Il piano di **contingenza** sarà un piano condizionale e magari con cicli.



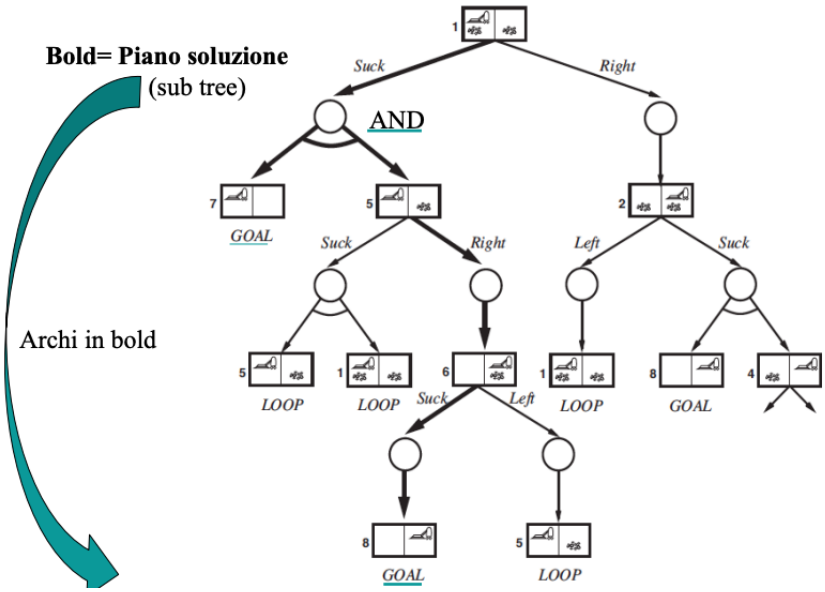
Abbiamo che Risultati(aspira, 1) = $\{5, 7\}$. Il piano possibile potrebbe essere:

```
Aspira, if stato = 5
   then [Destra, Aspira]
   else []
```

La soluzione è da sequenza di azioni a piano (albero).

8.6.1 Alberi di ricerca AND-OR

Nodi OR le scelte dell'agente [1 sola azione]. **Nodi AND** le diverse contingenze (le scelte dell'ambiente, piu stati possibili), da considerare tutte. Una soluzione a un problema di ricerca ANDOR è un albero che: ha un nodo obiettivo in ogni foglia specifica un'unica azione nei nodi OR include tutti gli archi uscenti da nodi AND (tutte le contingenze).



Esempio 8.6.2. Un piano di ricerca potrebbe essere il seguente:

```
Aspira, if stato = 5 then [Destra, Aspira] else []
```