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:

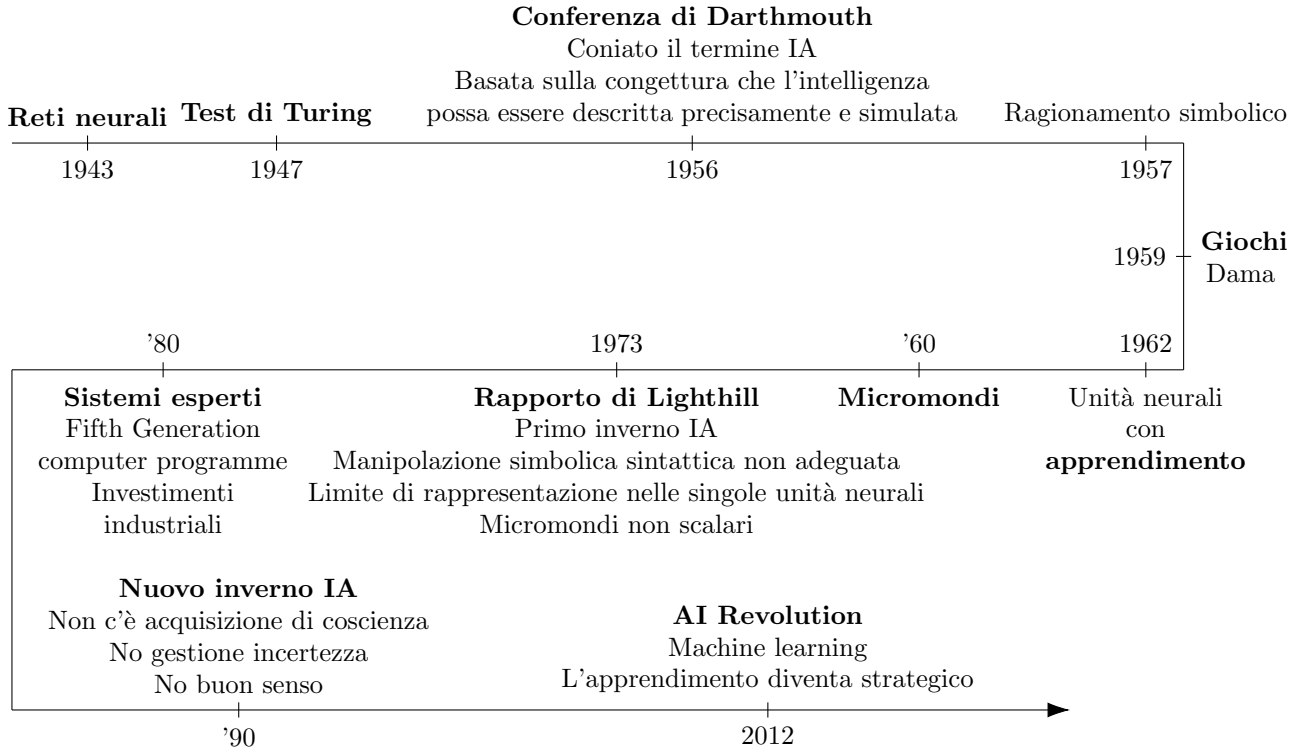
- Pensare razionalmente
- Agenti razionali: percepiscono l'ambiente, operano autonomamente e si adattano. Fanno la cosa giusta agendo in modo da ottenere il miglior risultato calcolando come agire in modo efficace e sicuro in una varietà di situazioni nuove. Ha alcuni vantaggi:

  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

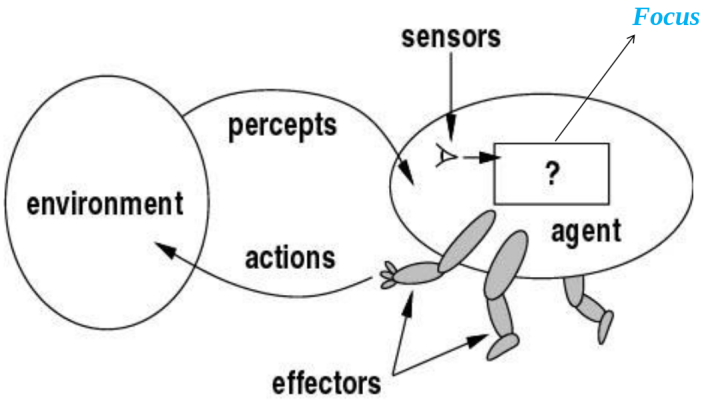
Abbinando alla capacità dei modelli di machine learning una grande quantità di dati e degli High Performance Computer, si è favorito molto il deep learning.
Dal 2010 le reti neurali profonde hanno iniziato a diffondersi molto nelle grandi industrie, riscuotendo successo ad esempio:

- **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.

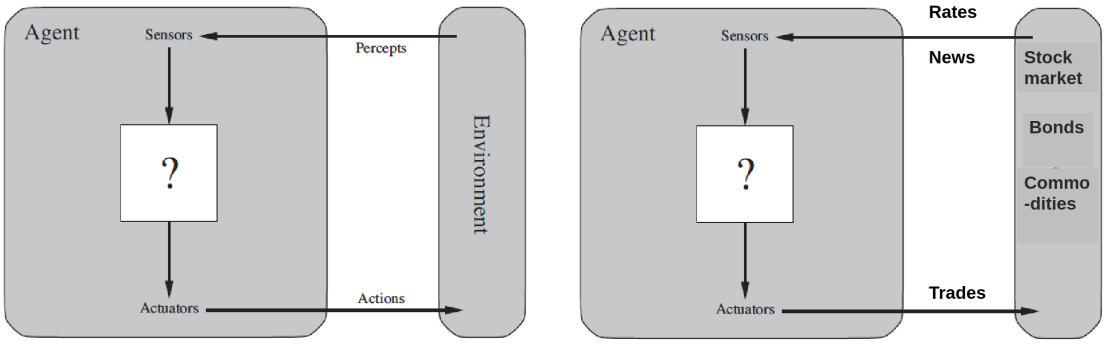
2.1 Caratteristiche

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 $Ag : Percezioni \rightarrow Azioni$.

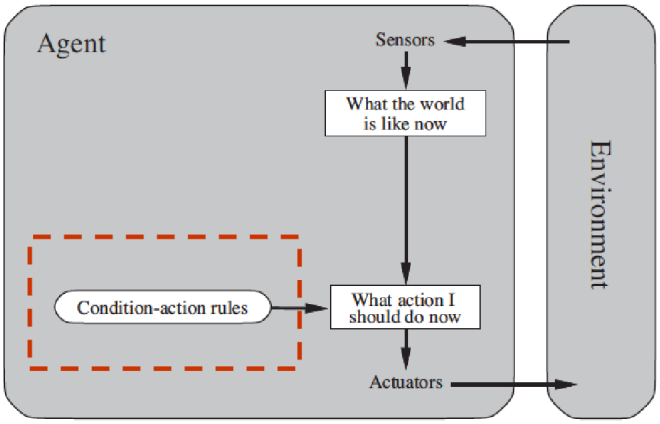
```
function Skeleton-Agent (percept) returns action
   static: memory, agent memory of the world
   memory <- UpdateMemory(memory, percept)
   action <- Choose-Best-Action(memory)
   memory <- UpdateMemory(memory, action)
   return action
```

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

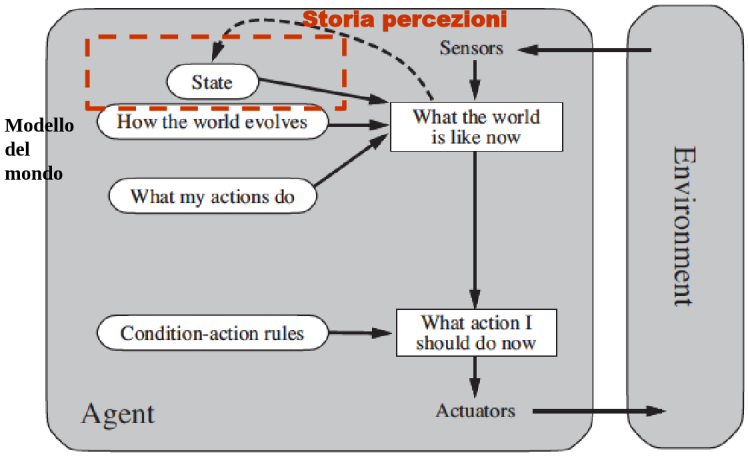
L'agente agisce in base a quello che percepisce senza salvare nulla in memoria.




```
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

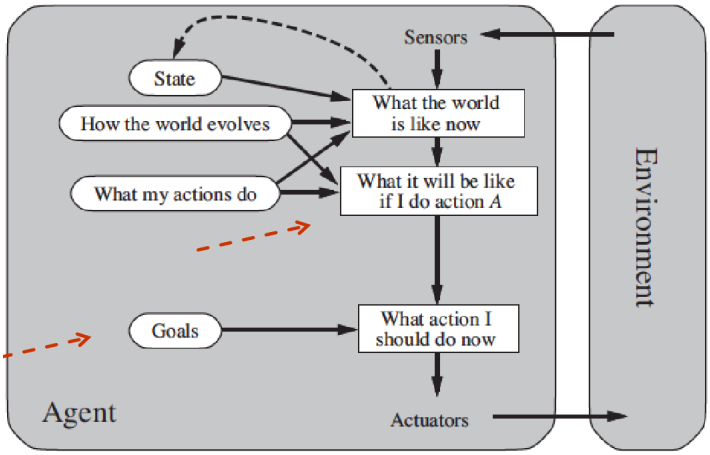
L'agente ha uno stato che mantiene la storia delle percezioni e influenza il modello del mondo.




```
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
```

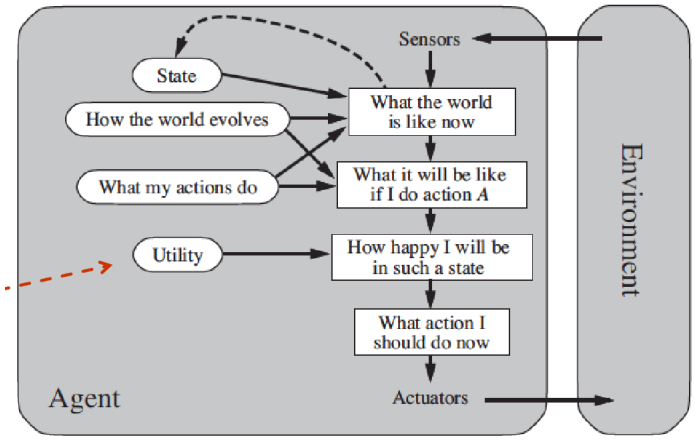
2.4.4 Agenti con obiettivo

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.



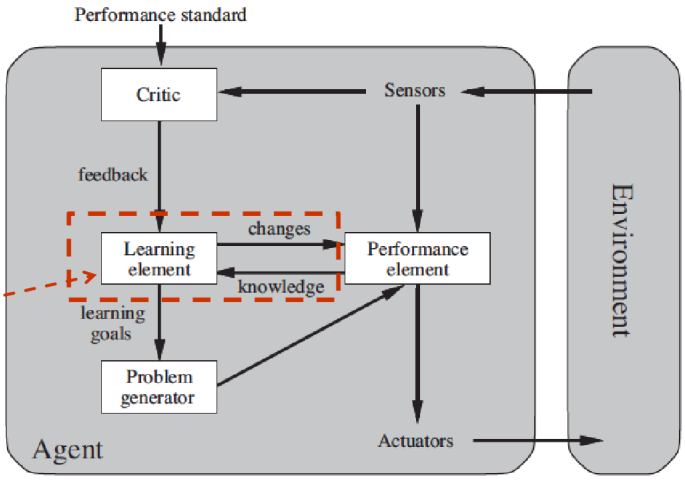
2.4.5 Agenti con valutazione di utilità

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

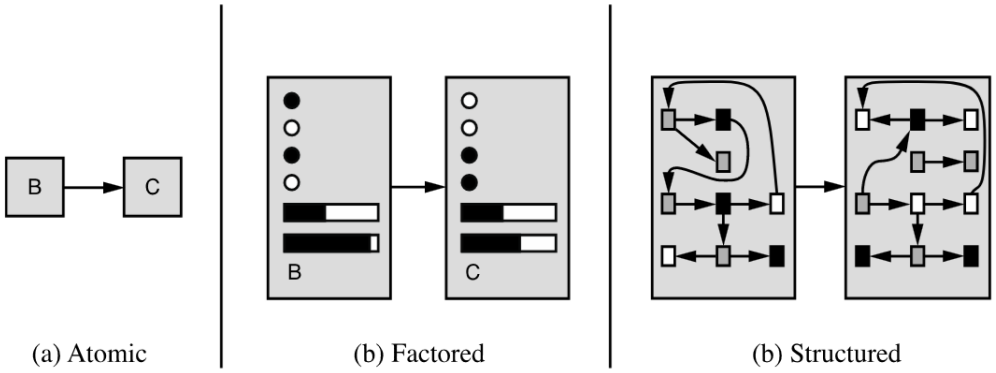
Questo tipo di agente include la capacità di **apprendimento** che produce cambiamenti al programma e ne migliora le prestazioni, adattando i comportamenti.
L'elemento **esecutivo** è il programma stesso, quello **critico** osserva e dà feedback ed infine c'è un generatore di problemi per suggerire nuove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

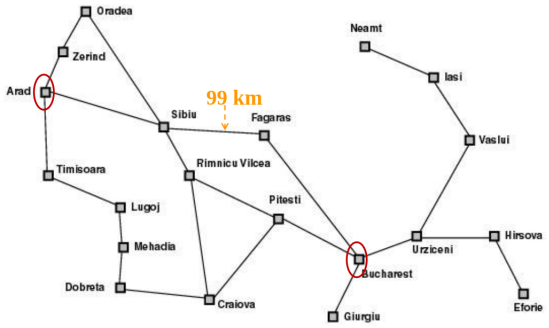
```
Risultato(In(Arad), Go(Sibiu)) = In(Sibiu)
```

4. **Test obiettivo**:

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

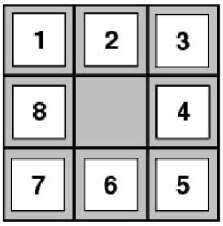
5. **Costo del cammino**: somma delle lunghezze delle strade

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:

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

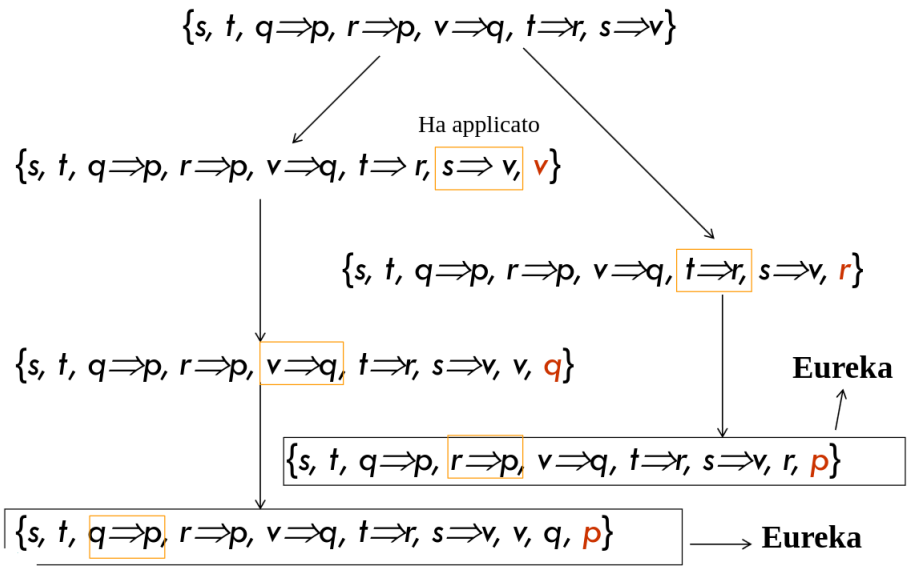
dimostrare una proposizione p utilizzando solamente la regola di inferenza *Modus Ponens*:

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

Scriviamo la formulazione del problema:

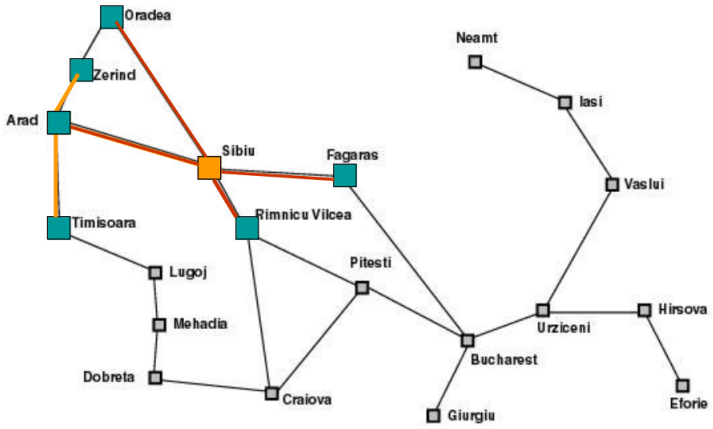
- **Stati**: insieme di proposizioni
- **Stato iniziale**: le premesse
- **Stato obiettivo**: un insieme di proposizioni contenente il teorema da dimostrare
- **Operatori**: l'applicazione del Modus Ponens

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:

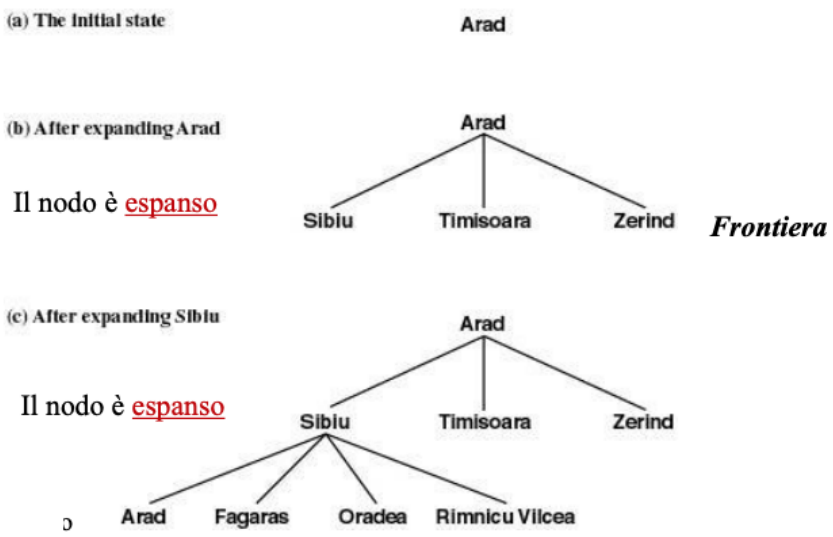


Figure 1: generazione di un albero di ricerca

```
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).

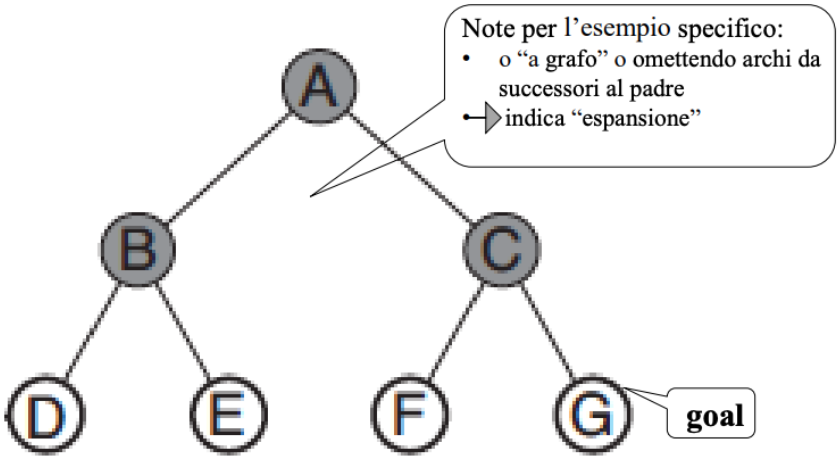


Figure 2: Ricerca in ampiezza

```
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):

$$T(b,d) = 1 + b + b^2 + \cdots + b^d \rightarrow O(b?d)$$ 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): $O(b^d)$

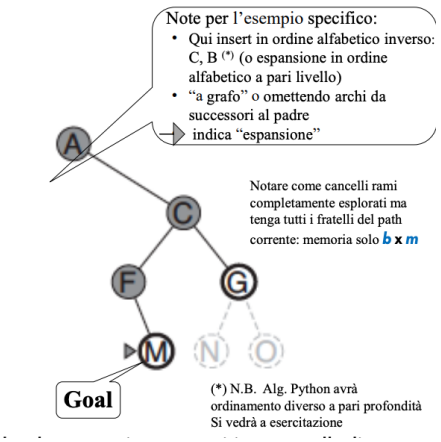
Esempio 4.1.1. b=10, 1 milione nodi al sec generati; 1 nodo occupa 1000 byte

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
```

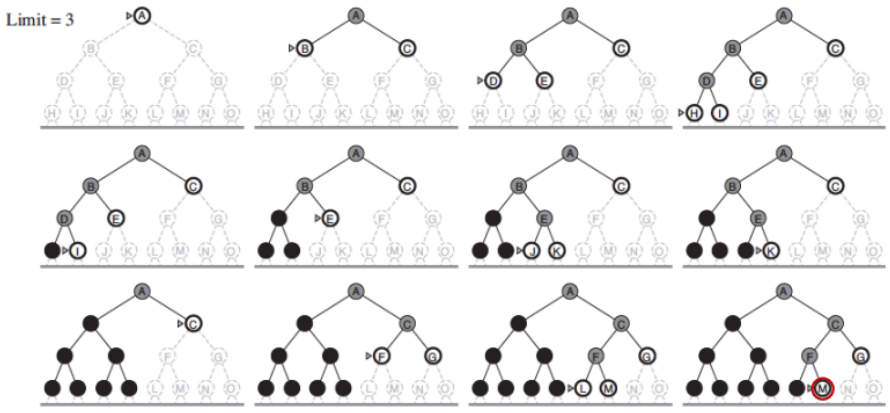
```python
def recursive_depth_first_search(problem, node):
  """Ricerca in profondita' ricorsiva """
  # controlla se lo stato del nodo e' uno stato obiettivo
  if problem.goal_test(node.state):
    return node.solution()
  # in caso contrario continua
  for action in problem.actions(node.state):
  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

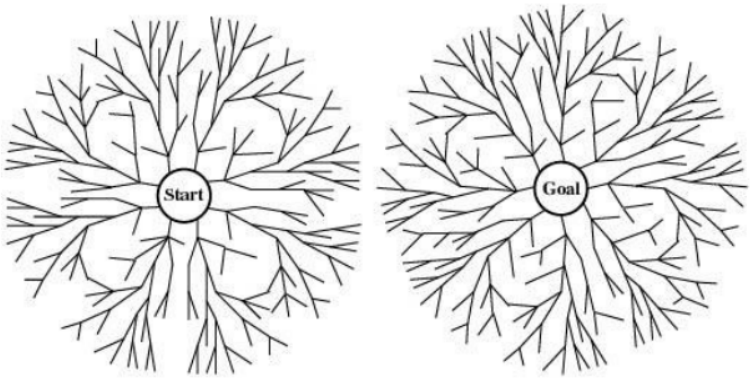
- Ricerca **in avanti** o **guidata dai dati**: si esplora lo spazio di ricerca dallo stato iniziale allo stato obiettivo.
- Ricerca **all'indietro** o **guidata dall'obbiettivo**: si esplora lo spazio di ricerca a partire da uno stato goal e riconducendosi a sotto-goal fino a trovare uno stato iniziale.

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.

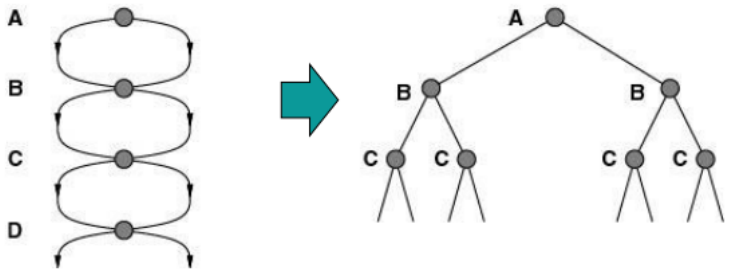


Complessità tempo: $O(b^{d/2})$ (assumendo test intersezione in tempo costante, es. hash table).
Complessità spazio: $O(b^{d/2})$ (almeno tutti i nodi una direzione in memoria, es usando BF).

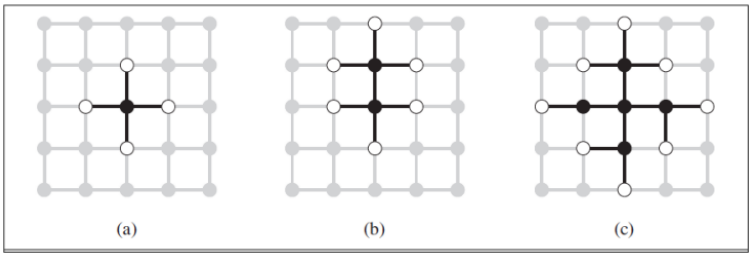
Note 4.4.1. Non sempre applicabile, es. predecessori non definiti, troppi stati obiettivo, ...

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, ...)

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:

```python
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).