

Appunti di Introduzione all'Intelligenza Artificiale Unipi - Parte 1

Raffaele Apetino

Marzo 2020

Contents

1 Premessa	2
2 Introduzione	2
2.1 Test di Turing	2
3 Agenti Intelligenti	3
3.1 Caratteristiche	3
3.2 Agenti Razionali	4
3.3 Agenti Autonomi	4
4 Ambienti	4
4.1 Descrizione PEAS dei problemi	4
4.2 Proprietà dell'ambiente	5
4.3 Simulatore di ambienti	5
5 Tipi di Agente	6
5.1 Struttura generale di un agente	6
5.2 Agente basato su tabella	6
5.3 Agenti reattivi semplici	6
5.4 Agenti basati su modello	7
5.5 Agenti con obiettivo	7
5.6 Agenti con valutazione di utilità	8
5.7 Agenti che apprendono	8
6 Agenti risolutori di problemi	9
6.1 Formulazione di un problema	9
6.2 Algoritmi di ricerca	9
6.3 Il problema dell'itinerario	9
6.3.1 Formulazione del problema dell'itinerario	10
6.4 Il problema dell'aspirapolvere	10
6.4.1 Formulazione del problema dell'aspirapolvere	10
7 Ricerca della soluzione (non informata)	11
7.1 Strutture dati per gli algoritmi di ricerca	11
7.2 Ricerca ad Albero	11
7.3 Strategie non informate VS Strategie informate "euristiche"	12
7.4 Valutazione di una strategia	12
7.5 Problemi cammini ciclici e ridondanze	12

7.6	Ricerca in ampiezza - BF	13
7.6.1	BF-Albero	13
7.6.2	BF-Grafo	13
7.6.3	Analisi complessità BF	13
7.7	Ricerca in profondità - DF	14
7.7.1	Analisi complessità DF-Albero	14
7.7.2	Analisi complessità DF-Grafo	14
7.8	Ricerca in profondità ricorsiva - DF con backtracking	15
7.9	Ricerca in profondità limitata - DL	15
7.9.1	Analisi complessità DL	15
7.10	Approfondimento Iterativo - ID	15
7.10.1	Analisi complessità ID	15
7.11	Ricerca Bidirezionale - Bidir.	16
7.11.1	Analisi complessità Bidir.	16
7.12	Ricerca di costo uniforme - UC	16
7.12.1	UC-Albero	17
7.12.2	UC-Grafo	17
7.12.3	Analisi complessità UC	17
7.13	Confronto finale delle strategie	18
8	Ricerca Euristica	18
8.1	Algoritmo Best-First - BFH	18
8.2	Algoritmo Greedy-Best-First - GBF	19
8.3	Algoritmo A	19
8.3.1	Completezza Algoritmo A	19
8.3.2	Dimostrazione Completezza di A	20
8.4	Algoritmo A^*	20
8.4.1	Esempio A^* sul problema dell'itinerario	20
8.4.2	Osservazioni su A^*	20
8.4.3	Ottimalità di A^*	21
8.4.4	Euristica Consistente (o monotona)	21
8.4.5	Dimostrazione ottimalità di A^*	21
8.4.6	Vantaggi di A^*	21
8.5	Costruire le euristiche di A^*	22
8.6	Valutare Algoritmi di ricerca euristica	22
8.7	Inventare euristiche	23
8.8	Algoritmi evoluti basati su A^*	23
8.8.1	Beam Search	23
8.8.2	A^* con approfondimento iterativo - IDA^*	23
8.8.3	Best-First ricorsivo - RBFS	24
8.8.4	A^* con memoria limitata - SMA^*	24
9	Ricerca Locale	25
9.1	Spazio degli stati	25
9.2	Algoritmo Hill-Climbing	25
9.2.1	Problemi e Miglioramenti per Hill-Climbing	26
9.3	Algoritmo Tempra Simulata (Simulated Annealing)	26
9.4	Algoritmo Local Beam	27
9.4.1	Local Beam Search Stocastica	27
9.5	Algoritmi Genetici - GA	27
9.6	Gradiente (Hill-Climbing iterativo)	28

10 Oltre la ricerca classica	29
10.1 Problema dell'aspirapolvere non deterministico	29
10.2 Alberi AND-OR	29

1 Premessa

Questi appunti sono stati controllati più volte, ma come ben si sa:

Sa chi sa che nulla sa, e chi sa che nulla sa ne sa più di chi ne sa.

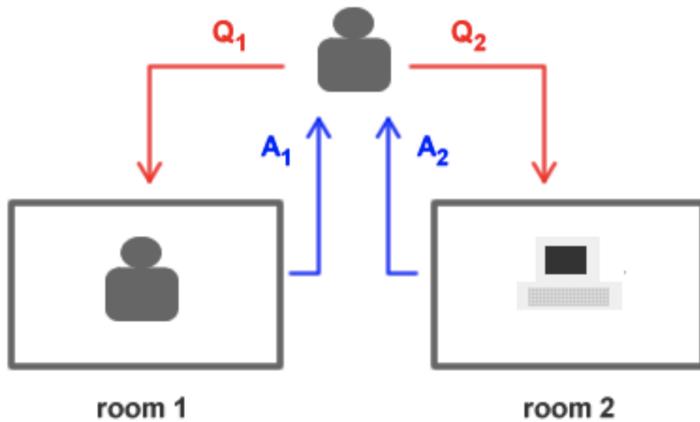
Quindi invito a guardare per bene le slide del corso e studiare su quelle. Consiglio di sfruttare questi appunti per un ripasso veloce. Per segnalare errori mandatemi una mail a r.apetino at studenti.unipi.it
∨ apriete una Issue su GitHub ∨ fate una Pull Request ∨ cercatemi su Telegram.

2 Introduzione

L'intelligenza artificiale si occupa della comprensione e riproduzione del comportamento intelligente. L'approccio psicologico (psicologia cognitiva) ha come obiettivo la comprensione dell'intelligenza umana e quindi risolvere i problemi con gli stessi processi usati dall'uomo. L'approccio informatico è quello di costruire entità dotate di razionalità, cioè si occupa dell'automazione del comportamento intelligente. Quest'ultimo viene eseguito attraverso la meccanizzazione del ragionamento e la comprensione mediante modelli computazionali della psicologia e del comportamento degli uomini.

C'è però una domanda molto importante riguardo a cosa sia l'intelligenza: capacità di ragionamento? Buon senso? Capacità sociali e di comunicazione? Capacità di comprendere e provare emozioni?

2.1 Test di Turing



In due stanze separate ci sono una persona ed un computer, fuori da queste stanze con le porte chiuse, c'è una seconda persona che fa domande ad entrambi. Il test di Turing si basa su dopo quanto tempo l'uomo esterno riesce a capire chi è uomo e chi macchina.

Da qui ci viene una domanda fondamentale, dobbiamo dotare i computer di senso comune? (esistono già dei progetti nominati CYC e OpenMind) Una definizione di senso comune possiamo darla, è la capacità di un uomo di poter riconoscere in modo immediato ricorrendo all'uso della ragione naturale. Quindi possiamo anche dare una definizione di intelligenza: è la qualità mentale che consiste nell'abilità di apprendere dall'esperienza, di adattarsi a nuove situazioni, comprendere e gestire concetti astratti, utilizzare la conoscenza per agire sul proprio ambiente.

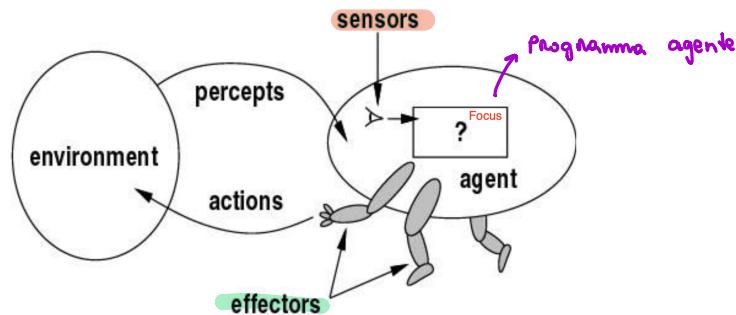
L'intelligenza artificiale è pericolosa? Andrebbe regolata? Le macchine dovrebbero avere un'etica? Ma su quale etica dovrebbero basarsi? L'intelligenza collettiva può essere estratta o inferita dai dati?

3 Agenti Intelligenti

L'approccio moderno dell'IA si basa sulla costruzione di agenti intelligenti e sulla creazione del programma agente da parte del programmatore. La visione ad agenti ci offre un quadro di riferimento e una prospettiva diversa dall'analisi dei sistemi software. Il nostro primo obiettivo è realizzare agenti per la risoluzione di problemi vista come ricerca in uno spazio di stati.

3.1 Caratteristiche

Un agente è qualcosa in grado di **percepire l'ambiente attraverso i sensori** e **agire su quell'ambiente sfruttando gli attuatori**.



Agenti Intelligenti:

- Sono situati: **ricevono percezioni da un ambiente (tramite input dei sensori), agiscono sull'ambiente mediante azioni (sfruttando gli attuatori)**.
La sequenza percettiva è la storia completa delle percezioni. La scelta dell'azione è funzione unicamente della sequenza percettiva
- Gli agenti hanno abilità sociale: **sono capaci di comunicare, collaborare, difendersi da altri agenti**.
- Gli agenti hanno **credenze, obiettivi, intenzioni...**
- Gli agenti sono embodied: **hanno un corpo, fino a considerare i meccanismi delle emozioni**.

In termini matematici diciamo che il comportamento dell'agente è descritto dalla funzione agente. **La funzione agente è una descrizione matematica che definisce l'azione da compiere per ogni sequenza percettiva, il programma agente è una implementazione concreta di questa funzione**. Il nostro compito è proprio quello di progettare il programma agente.

$$\text{Funzione Agente: } f : \text{seq. percettiva} \rightarrow \text{A}$$

$$f(\alpha) = A$$

3.2 Agenti Razionali

Un agente razionale interagisce con il suo ambiente in maniera efficace cioè "fa la cosa giusta". Serve quindi un criterio di valutazione oggettivo dell'effetto delle azioni dell'agente¹, come ad esempio il costo minimo di un cammino per arrivare alla soluzione. La razionalità è relativa alla misura delle prestazioni, alle conoscenze pregresse dell'ambiente e alle capacità dell'agente.

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.

Non ti prende conoscenza

Raramente tutta la conoscenza sull'ambiente può essere fornita "a priori", l'agente razionale deve essere in grado di modificare il proprio comportamento con l'esperienza (percezioni passate oppure percezioni che è in grado di apprendere in futuro).

3.3 Agenti Autonomi

Modificare il proprio comportamento con l'esperienza implica la creazione di agenti autonomi:

Un agente è autonomo nella misura in cui il suo comportamento dipende dalla sua esperienza.

Un agente il cui comportamento fosse determinato solo dalla sua conoscenza built-in, sarebbe non autonomo e poco flessibile.

4 Ambienti

Definire un problema per un agente significa caratterizzare l'ambiente in cui l'agente opera.

4.1 Descrizione PEAS dei problemi

- Performance (prestazione, obiettivo)
- Environment (ambiente, dove attua le sue azioni)
- Actuators (attuatori, meccanismi con cui agisco sull'ambiente)
- Sensors (sensori, percezioni)

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, acelerometro, sensori sullo stato del motore, tastiera o microfono

Figure 1: Esempio descrizione PEAS di un agente guidatore di taxi

¹vedremo che una azione ha come conseguenza la creazione di un nuovo stato nell'ambiente

4.2 Proprietà dell'ambiente

- L'ambiente è osservato dall'agente che ne apprende le sue caratteristiche.
 - **Completamente osservabile:** conoscenza completa dell'ambiente, non c'è bisogno di mantenere uno stato del mondo esterno.
 - **Parzialmente osservabile:** sono presenti limiti o inaccuratezze che riguardano la conoscenza del mondo. (*ho bisogno di mantenere uno stato del mondo esterno*)
 - **Non osservabile:** se l'agente non ha sensori
- Il mondo può cambiare anche per eventi, non necessariamente per azioni di agenti.
 - **Agente singolo.**
 - **Multi-agente:** può essere a sua volta **competitivo**, oppure **cooperativo** quindi con lo stesso obiettivo (comunicano).

- Si possono predire i cambiamenti del mondo.
 - **Deterministico:** se lo stato successivo è completamente determinato dallo stato corrente e dall'azione.
 - **Stocastico:** esistono elementi di incertezza con associata probabilità. (*Stato succ. dato dalle prob.*)
 - **Non deterministico:** non si può sapere come evolve il mondo quindi si tiene traccia di più stati possibili risultanti da una azione eseguita. (*Nessun legame tra stato corrente e successivo*)
- L'ambiente si può caratterizzare anche come
 - **Episodico:** l'esperienza dell'agente è divisa in episodi atomici indipendenti.
 - **Sequenziale:** ogni decisione influenza le successive.
- L'ambiente può cambiare nel corso del tempo.
 - **Statico:** il mondo non cambia mentre l'agente è fermo e sta decidendo l'azione.
 - **Dinamico:** il mondo cambia nel tempo, tardare equivale a non agire.
 - **Semi-dinamico:** l'ambiente non cambia ma la valutazione dell'agente si.
- L'ambiente è caratterizzato da valori.
 - **Discreto:** i valori del mondo sono limitati (ad esempio gli stati possono essere di numero finito)
 - **Continuo:** i valori del mondo possono essere infiniti (ad esempio il tempo può essere infinito)
- Lo stato di conoscenza dell'agente può essere:
 - **Noto:** conosco l'ambiente, questo non significa che sia osservabile (ad esempio in un gioco di carte, le carte sono note ma se sono coperte non sono osservabili!)
 - **Ignoto:** devo compiere azioni esplorative per conoscerlo tutto.

Gli ambienti reali sono parzialmente osservabili, stocastici, sequenziali, dinamici, continui, multi-agente, ignoti.

4.3 Simulatore di ambienti

E' uno strumento software che si occupa di generare stimoli per gli agenti, raccogliere le azioni di risposta, aggiornare lo stato dell'ambiente e valutare le prestazioni dell'agente.

Function agente (percezione):
 memoria = Aggiornamemoria (memoria, percezione)
 azione = ScegliAzione (memoria)
 memoria = Aggiornamemoria (memoria, azione)
 return azione

5 Tipi di Agente

5.1 Struttura generale di un agente

Struttura Agente = Architettura² + Programma
 Funzione Agente³ = Ag : Percezioni → Azioni

Il programma dell'agente implementa la funzione Ag.

Architettura: Parte hardware dell'agente

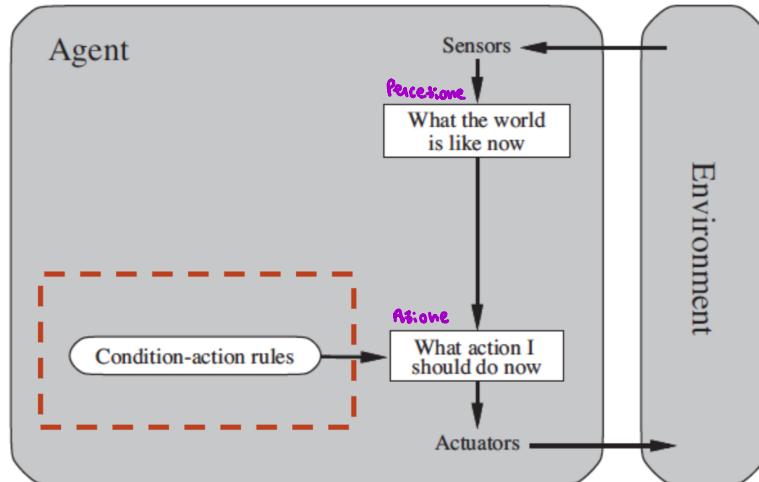
Programma: Impl. funzione agente

5.2 Agente basato su tabella

La scelta dell'azione è un accesso a una tabella che associa una azione ad ogni possibile sequenza di percezioni. Ci sono ovvi problemi:

1. Dimensione: per giocare a scacchi la tabella ha un numero di righe molto maggiore di 10^{80} perciò la situazione è ingestibile.
2. Difficile da costruire.
3. Nessuna autonomia.
4. Difficile da aggiornare, quindi l'"apprendimento" diventa complesso.

5.3 Agenti reattivi semplici



Sono presenti delle regole if-then costruite a priori che mi dicono quale azione fare in base allo stato e alle regole in quel dato istante.

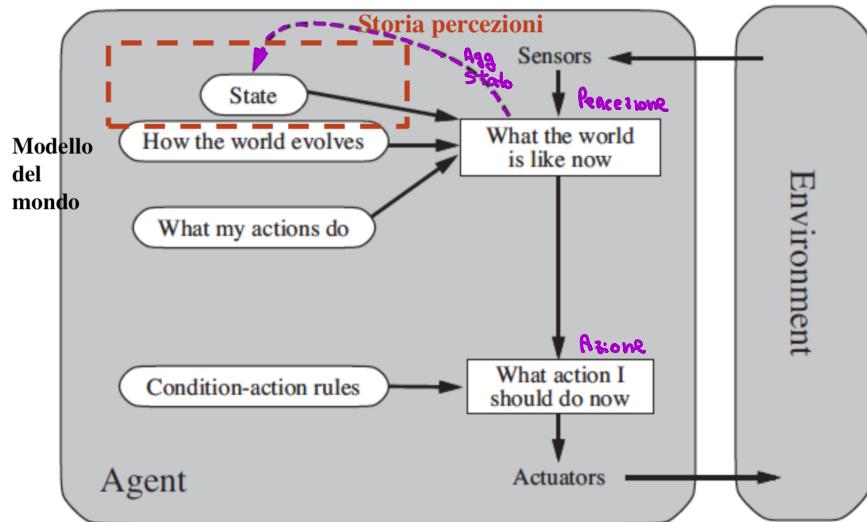
Si basa unicamente sulla percezione che arriva dall'ambiente

Regola = Azione da svolgere

²è la parte hardware dove gira il nostro programma agente

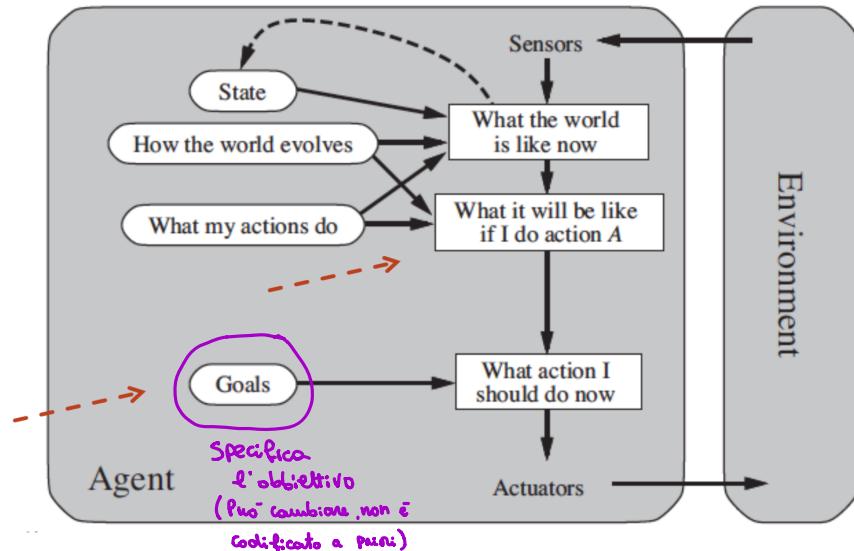
³La funzione agente definisce l'azione da compiere per ogni sequenza percettiva

5.4 Agenti basati su modello



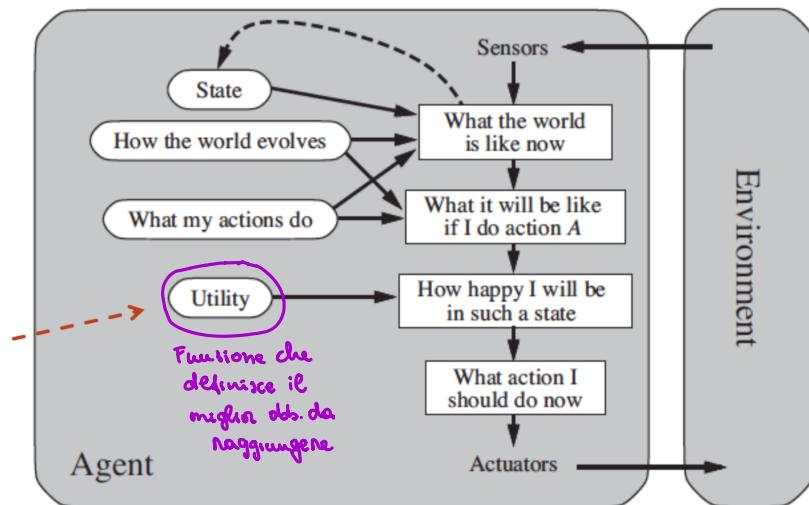
E' presente un modello del mondo che comprende lo stato aggiornato con la storia delle percezioni. Sono ancora presenti le regole if-then.

5.5 Agenti con obiettivo (Scelego l'azione che mi avvicina all'obiettivo)



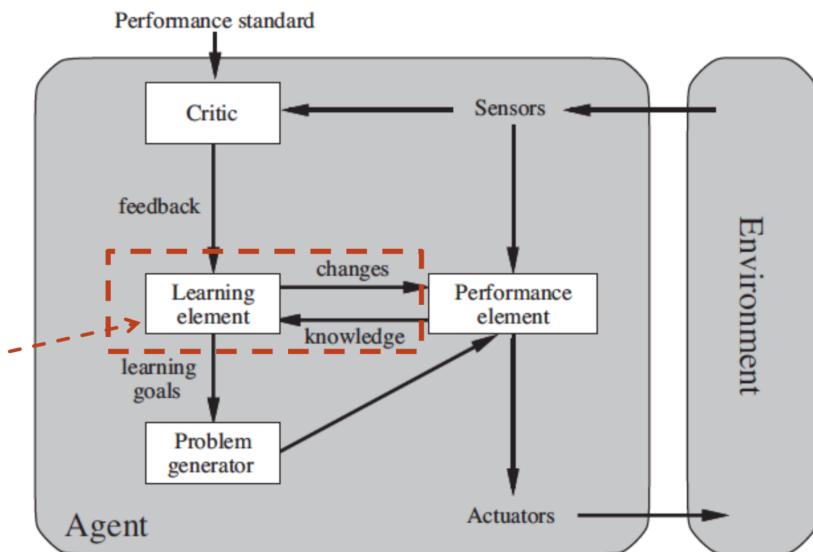
Sono agenti guidati da un obiettivo nella scelta dell'azione (viene fornito un goal esplicito, ad esempio raggiungere una città). L'azione migliore dipende da quale obiettivo bisogna raggiungere e pianificano le proprie azioni in base al goal. L'agente si preoccupa di capire come sarà il mondo dopo aver eseguito una azione.

5.6 Agenti con valutazione di utilità (Scelgo l'azione migliore in base alle funz. di utilità)



Ci sono obiettivi alternativi magari più facilmente raggiungibili. L'agente deve decidere verso quali di questi muoversi. E' necessaria una funzione di utilità che associa ad uno stato un numero reale. La funzione di utilità tiene conto anche della probabilità di successo e di utilità attesa.

5.7 Agenti che apprendono



E' presente una componente di apprendimento che produce cambiamenti al programma agente, migliora le prestazioni adattando i suoi componenti, apprendendo dall'ambiente. L'elemento esecutivo è il programma agente, l'elemento critico osserva e dà feedback sul comportamento. Infine è presente un generatore di problemi, suggerisce nuove situazioni da esplorare.

6 Agenti risolutori di problemi (ambiente statico, osservabile, discreto e deterministico.)

Adottano il paradigma della risoluzione di problemi come ricerca in uno spazio di stati. Sono agenti con modello che adottano una rappresentazione atomica dello stato, hanno un obiettivo e pianificano l'intera sequenza di mosse prima di agire.

Processo di risoluzione di un problema:

1. Determinare un obiettivo (un insieme di stati tali che l'obiettivo è soddisfatto)
2. Formulare un problema (rappresentazione degli stati e delle azioni)
3. Determinare soluzione mediante ricerca
4. Esecuzione soluzione

Assumiamo un ambiente statico, osservabile, discreto e deterministico.

6.1 Formulazione di un problema

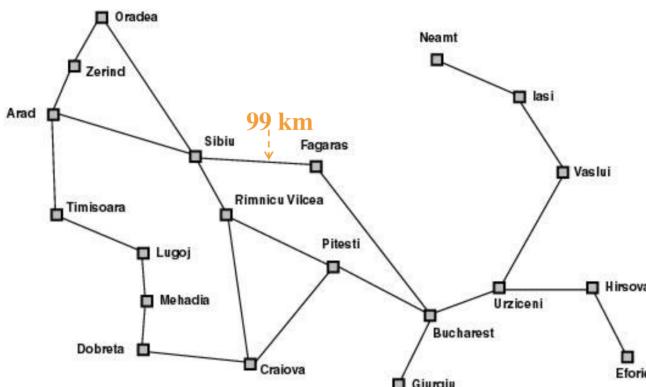
Un problema può essere definito formalmente mediante 5 componenti:

1. Stato iniziale \rightarrow Nodo
2. Azioni possibili nello stato ($Azioni(stato)$) \rightarrow Azioni uscenti da un nodo
3. Modello di transizione: $Risultato(stato, azione) = nuovo_stato \rightarrow$ Nodo
4. Test obiettivo: insieme di stati obiettivo ($GoalTest(stato) = \{true, false\}$) \rightarrow Nodi/o obiettivo
5. Costo del cammino: somma dei costi delle azioni, costo di passo definito come $c(s, a, s') \rightarrow$ Somma costi anche fino a s'
1, 2 e 3 definiscono implicitamente lo spazio degli stati.

6.2 Algoritmi di ricerca

Il processo che cerca una sequenza di azioni che raggiunge l'obiettivo è detto ricerca. Gli algoritmi di ricerca prendono in input un problema e restituiscono un cammino soluzione. La misura delle prestazioni è definita come: Costo totale = costo della ricerca + costo del cammino soluzione. Valuteremo gli algoritmi riguardo la ricerca cercando di ottimizzare il cammino soluzione.

6.3 Il problema dell'itinerario



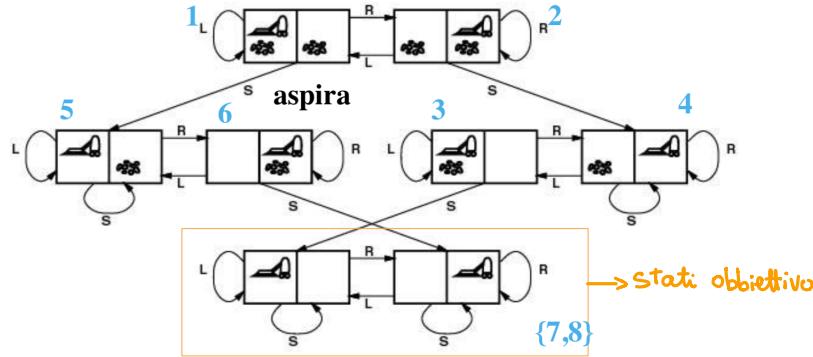
Il problema dell'itinerario riguarda la ricerca del percorso più breve da una città di partenza a una città di arrivo.

6.3.1 Formulazione del problema dell'itinerario

1. Stati: le città = **Nodi**
2. Stato iniziale: la città da cui si parte ($In(Arad)$)
3. Azioni: spostarsi su una città vicina collegata ($Azioni(In(Arad)) = \{Go(Sibiu), Go(Zerind), \dots\}$) = **Anelli vicini**
4. Modello di transizione ($Risultato(In(Arad), Go(Sibiu)) = In(Sibiu)$) \rightarrow **Nodo risultante da un'azione**
5. Test obiettivo $\{In(Bucarest)\} \rightarrow$ **Obiettivo**
6. Costo del cammino: somma delle lunghezze delle strade

Lo spazio degli stati coincide con la rete (grafo) di collegamenti tra città.

6.4 Il problema dell'aspirapolvere



Il problema dell'aspirapolvere riguarda la pulizia di due stanze adiacenti con il minimo sforzo.

6.4.1 Formulazione del problema dell'aspirapolvere

1. Stati: sono determinati sia dalla posizione dell'aspirapolvere che dalla posizione dello sporco

1	2
3	4
5	6
7	8

2. Stato iniziale: qualsiasi stato può essere uno stato iniziale
3. Percezioni: Sporco - Non sporco
4. Azioni: Sinistra (L) - Destra (R) - Aspira (S)
5. Modello di transizione: Aspira(stanza) = stanza pulita, Destra = si sposta nella stanza a destra, Sinistra = si sposta nella stanza a sinistra
6. Test obiettivo: rimuovere lo sporco (stati 7 o 8)
7. Costo del cammino: ogni azione ha costo 1

7 Ricerca della soluzione (non informata)

Si tratta di generare un albero di ricerca sovrapposto allo spazio degli stati (generato da possibili sequenze di azioni).

7.1 Strutture dati per gli algoritmi di ricerca

Gli algoritmi di ricerca hanno bisogno di strutture dati per tenere traccia di come l'albero di ricerca si sta formando.

Un nodo n è una struttura dati con quattro componenti: (Attributi nodo di un albero)

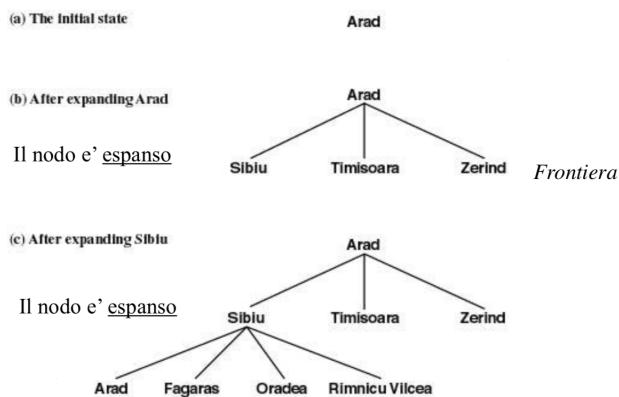
1. Uno stato: n.estado
2. Il nodo padre: n.padre
3. L'azione effettuata sul padre per generarlo: n.azione
4. Il costo del cammino a partire dal nodo iniziale: n.costocammino definito come:
$$g(n) = n.padre.costocammino + \text{costo dell'ultimo passo}$$

La frontiera è lista dei nodi in attesa di essere espansi (le foglie dell'albero di ricerca). Essa può essere implementata come una coda FIFO (viene estratto l'elemento più vecchio della coda), LIFO (viene estratto quello più recentemente inserito) o con priorità (viene estratto quello con priorità più alta).

Sulla frontiera sono definite le seguenti operazioni:

- Vuota(coda) // mi dice se la coda è vuota
- Pop(coda) // estrae il primo elemento dalla coda in base alla strategia utilizzata
- Inserisci(elemento, coda) // inserisce un elemento della coda

7.2 Ricerca ad Albero



Nella ricerca ad albero non controlliamo se i nodi (stati) siano già stati esplorati (questo controllo lo vedremo successivamente sui grafî). Come prima operazione inizializzo la frontiera con lo stato iniziale (Arad), ad ogni ciclo controllo se la frontiera è vuota, se lo è FAIL altrimenti scelgo un nodo della frontiera e lo rimuovo. Se il nodo rimosso è contenuto negli stati obiettivo allora ho trovato la soluzione, altrimenti espando il nodo e aggiungo i successori alla frontiera. Come si può notare dall'immagine la ricerca ad albero può portare alla creazione di loop.

Il problema quindi è quale tra i nodi della frontiera scelgo?

7.3 Strategie non informate VS Strategie informate "euristiche"

Strategie non informate: (Algoritmi di ricerca di una soluzione)

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

Le strategie informate "euristiche" le vedremo dopo, fanno uso di informazioni riguardo alla distanza stimata dalla soluzione.

7.4 Valutazione di una strategia

Dobbiamo definire un criterio per capire quale algoritmo di ricerca è migliore di un altro.

Metriche
di un algo
di ricerca,

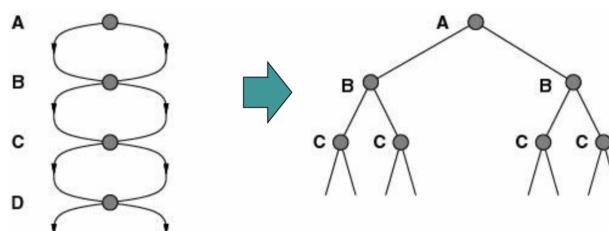
- Completezza: l'algoritmo garantisce di trovare una soluzione quando esiste?
- Ottimalità (ammissibilità): trova la soluzione migliore con costo minore
- Complessità in tempo: tempo richiesto per trovare la soluzione
- Complessità in spazio: memoria richiesta per trovare la soluzione

Per descrivere la complessità di ogni algoritmo sfrutteremo le seguenti definizioni che valgono sia per i grafi che per gli alberi:

- minimo
- b = fattore di ramificazione (branching, numero di nodi successori)
 - d = profondità del nodo obiettivo più superficiale (depth)
 - m = lunghezza massima dei cammini nello spazio degli stati (max)

7.5 Problemi cammini ciclici e ridondanze

I cammini ciclici rendono gli alberi di ricerca infiniti. Su spazi di stati a grafo si generano più volte gli stessi nodi (o meglio nodi con stesso stato) nella ricerca, anche in assenza di cicli.



Ricordare gli stati già visitati occupa spazio ma ci consente di evitare di visitarli di nuovo. Ci sono tre soluzioni:

1. Non tornare nello stato da cui si proviene, si elimina il padre dai nodi successori (ma non evita cammini ridondanti).
2. Per evitare di creare cammini con cicli si controlla che i successori non siano antenati del nodo corrente.
3. Per non generare nodi con stati già visitati/esplorati teniamo in memoria ogni nodo visitato con complessità in spazio di $O(\text{StatiPossibili})$.

DSS:

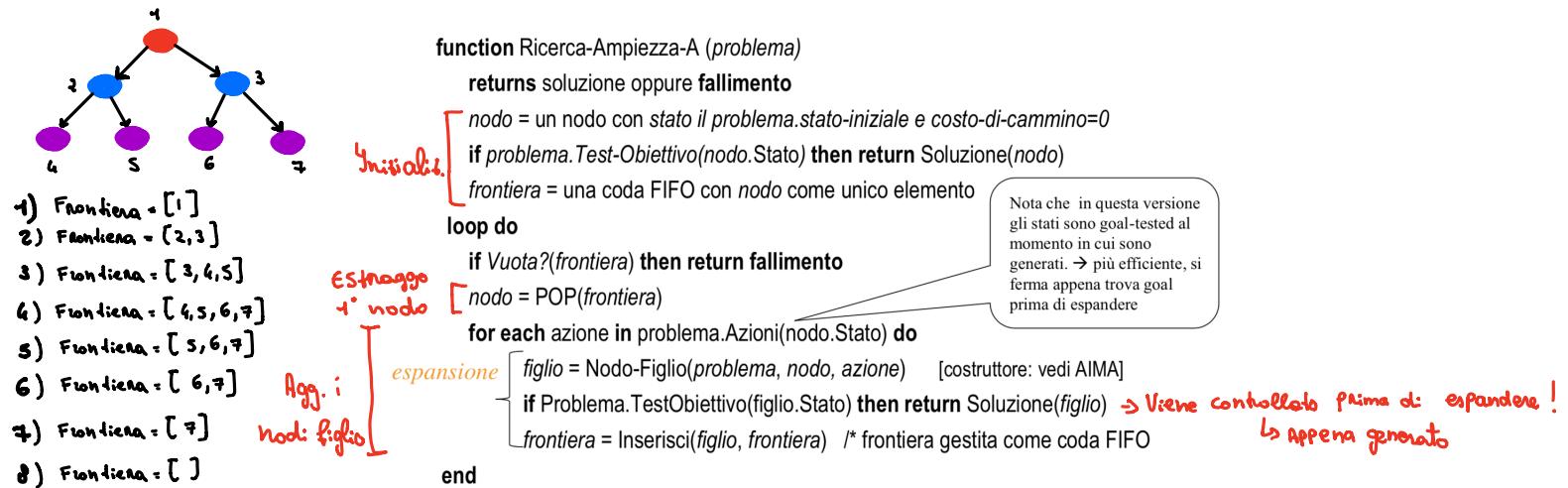
Algo ad albero: fanno generare anche nodi già esplorati

Algo a grafo: Non generano nodi già esplorati o che sono già in frontiera (tengono una lista degli esplorati)

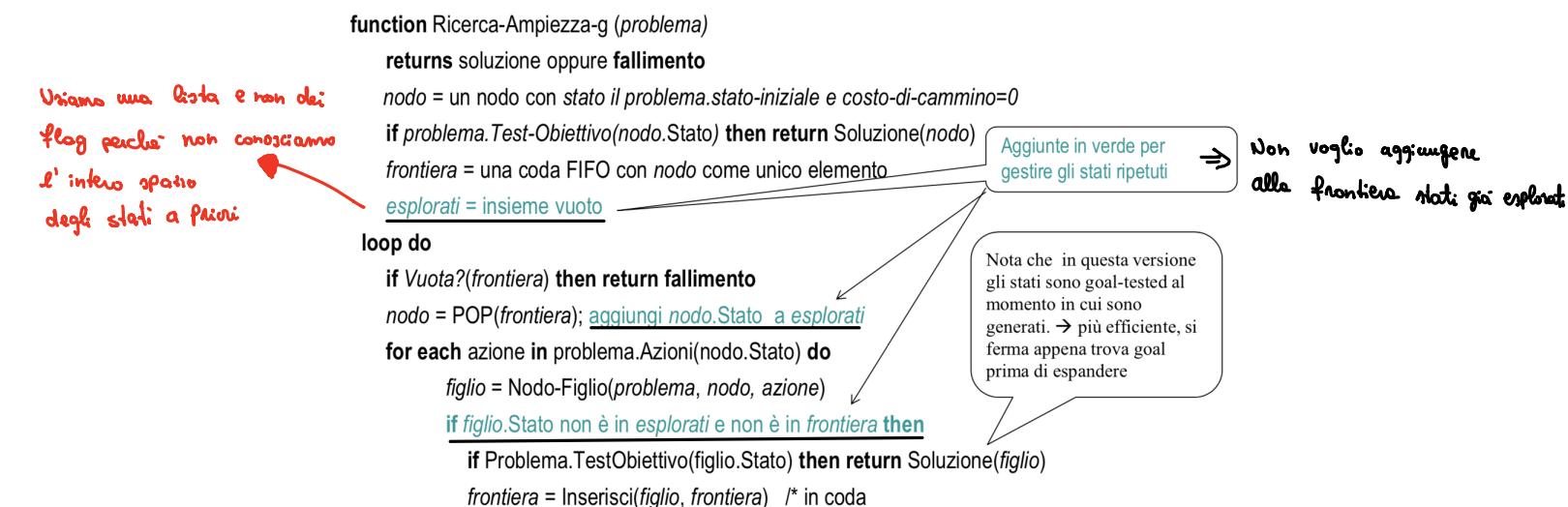
7.6 Ricerca in ampiezza - BF

Esplorare il grafo dello spazio degli stati a livelli progressivi di stessa profondità. La frontiera è implementata con una coda che inserisce alla fine (FIFO).

7.6.1 BF-Albero



7.6.2 BF-Grafo (evito dei cicli che mi posso trovare nella ricerca sull'albero)



7.6.3 Analisi complessità BF

- Strategia Completa: SI (4 cicli non intaccano la compl. ma posso andare a esp. + volte gli stessi nodi)
- Strategia Ottimale: SI se gli operatori hanno tutti lo stesso costo k⁴
- Complessità Tempo: $O(b^d)$ (b figli per ogni nodo all'altezza d)
- Complessità Spazio: $O(b^d)$ (occupa un sacco di memoria)

⁴cioè g(n) = k*depth(n) dove g(n) è il costo del cammino per arrivare a n

Dovuto alla frontiera (hw $O(b^d)$ nodi in frontiera da espandere)

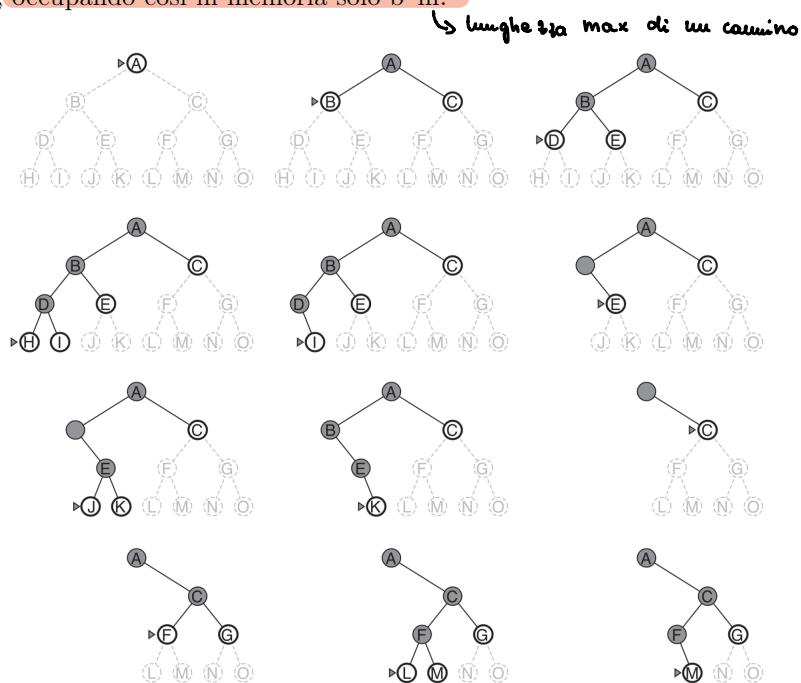
$T(b,d) = b + b^2 + b^4 + b^8 + \dots + b^d = O(b^d)$

l'ul profondità
↓ numero di espansioni
↑ nodi generati da ogni nodo
complexità esponenziale

BFS = buono per istante piccole perché occupa troppa mem. (e anche tempo in minor impatto)

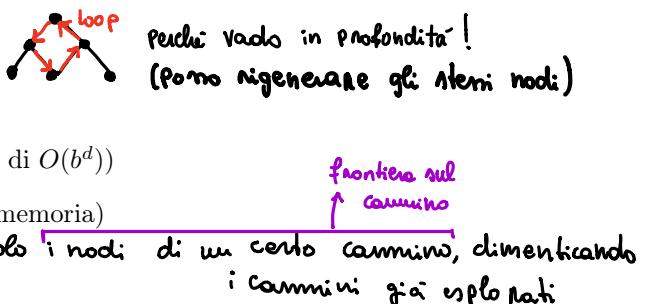
7.7 Ricerca in profondità - DF

La ricerca Depth-First esplora il grafo dello spazio degli stati arrivando in profondità⁵ per ogni nodo corrente sulla frontiera. La frontiera è implementata con una coda che inserisce i successori in testa alla lista (LIFO). Cancella rami già completamente esplorati ma tiene in memoria tutti i successori del path corrente, occupando così in memoria solo b^*m .



7.7.1 Analisi complessità DF-Albero

- Strategia Completa: NO, si possono creare dei loop
- Strategia Ottimale: NO
- Complessità Tempo: $O(b^m)$ (che può essere maggiore di $O(b^d)$)
- Complessità Spazio: $O(b * m)$ (drastico risparmio di memoria)



7.7.2 Analisi complessità DF-Grafo

In caso di DF con visita su grafo si perdono i vantaggi della memoria: la memoria torna ad essere $O(b^d)$ ma così DF diventa completa su spazi degli stati finiti (al caso pessimo estende tutti i nodi) resta comunque non completa su spazi infiniti.

Perché bisogna tenere l'intera lista degli esplorati → *Non puo' rigenerare i nodi (No loop)*

⁵fino a che non ha più successori

Non ha una frontiera, genera i nodi uno figlio uno per volta e fa backtracking con lo stack
 Per questo $O(n)$ di memoria

7.8 Ricerca in profondità ricorsiva - DF con backtracking

Ancora più efficiente in occupazione di memoria perché mantiene in memoria solo il cammino corrente (solo m nodi al caso pessimo). L'algoritmo è realizzato 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.

```
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 → Analizzo Quando torno indietro
        return fallimento
```

7.9 Ricerca in profondità limitata - DL \rightarrow Si usa DF fino a un certo livello l

Il fallimento della ricerca DF in uno spazio di stati infinito viene mitigato in parte se si procede in profondità fino ad un certo livello predefinito l .

Evita i loop della DF!

7.9.1 Analisi complessità DL

- Strategia Completa: SI solo per problemi in cui si conosce un limite superiore per la profondità della soluzione cioè se $l > d$. Se sceglio $l < d$ la ricerca risulta incompleta.
- Strategia Ottimale: NO
- Complessità Tempo: $O(b^l)$
- Complessità Spazio: $O(b * l)$

Livello accelto → Livello minimo obiettivo

7.10 Approfondimento Iterativo - ID

Ad ogni iterazione aumenta il limite della ricerca in profondità limitata (DL) e rincomincio dalla radice.

7.10.1 Analisi complessità ID

- Strategia Completa: SI
- Strategia Ottimale: SI se gli operatori hanno tutti lo stesso costo
- Complessità Tempo: $O(b^d)$ $T(b,d) = d \cdot b + (d-1)b^2 + \dots + 1 \cdot b^d = O(b^d)$
- Complessità Spazio: $O(b * d)$ \rightarrow Rimuove problema BF dello spazio

Miglior compromesso tra BF e DF, ma i nodi dell'ultimo livello sono generati una volta, quello del penultimo due, ..., quelli del primo d volte.

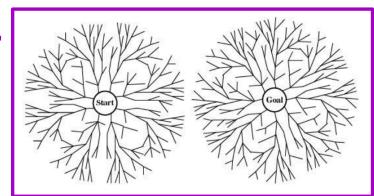
7.11 Ricerca Bidirezionale - Bidir.

Un problema che possiamo valutare è la **direzione della ricerca**.

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

In quale direzione conviene procedere? Conviene procedere nella direzione in cui il fattore di **ramificazione è minore**. Procediamo in avanti quando gli obiettivi sono molti e abbiamo una serie di dati da cui partire. Procediamo all'indietro quando l'obiettivo è chiaramente definito oppure i dati del problema non sono noti e la loro acquisizione può essere guidata dall'obiettivo.

Nella ricerca bidirezionale si procede nelle due direzioni fino ad incontrarsi.

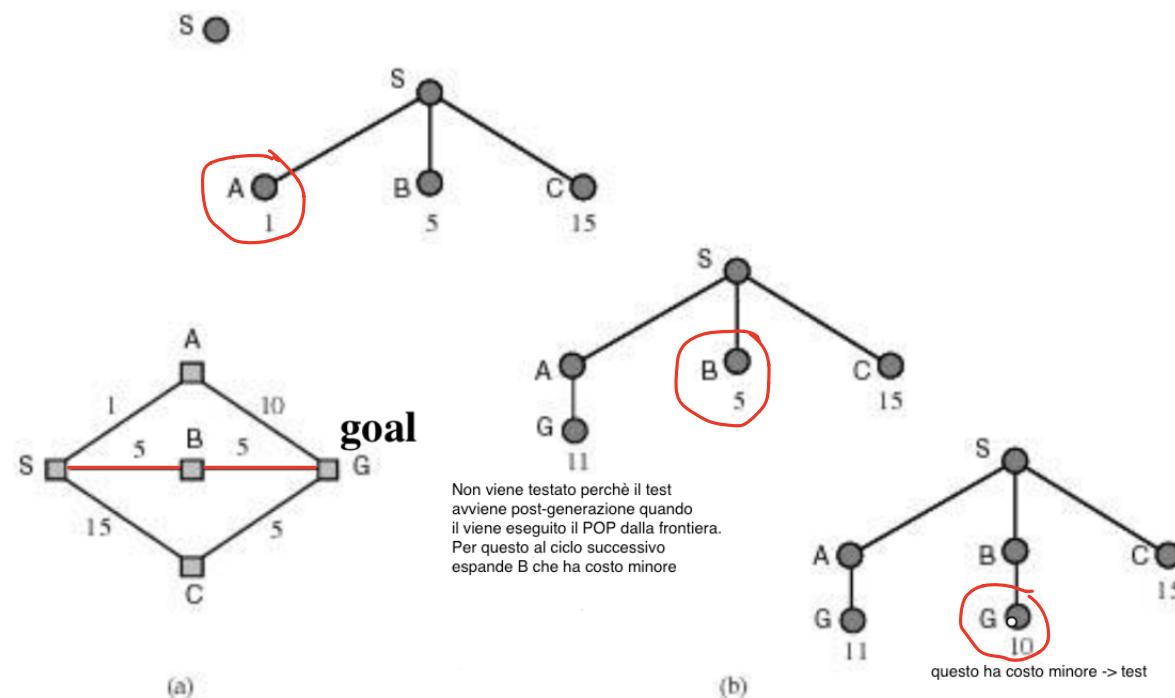


7.11.1 Analisi complessità Bidir.

- Strategia Completa: SI (se il branching è finito e si usa la ricerca BF)
- Strategia Ottimale: SI (se i costi sono tutti uguali e si usa la ricerca BF)
- Complessità Tempo: $O(b^{d/2}) = O(\sqrt{b^d})$
- Complessità Spazio: $O(b^{d/2}) = O(\sqrt{b^d}) \rightarrow$ Salvo almeno tutti i nodi in una direzione in memoria

7.12 Ricerca di costo uniforme - UC

E' una generalizzazione della ricerca in ampiezza dove i costi di ogni operatore sono diversi. Si sceglie il nodo di costo minore sulla frontiera (costo $g(n)$) e si espande. La frontiera è implementata da una coda ordinata per costo cammino crescente (cioè per primi i nodi di costo g minore). I nodi vengono goal-testati quando vengono scelti per l'espansione e non quando sono generati!



7.12.1 UC-Albero

```

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

7.12.2 UC-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 priorità con nodo come unico elemento
    esplorati = insieme vuoto
    loop do
        if Vuota?(frontiera) then return fallimento
        nodo = POP(frontiera);
        if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
        aggiungi nodo.Stato a esplorati
        for each azione in problema.Azioni(nodo.Stato) do
            figlio = Nodo-Figlio(problema, nodo, azione)
            if figlio.Stato non è in esplorati e non è in frontiera then
                frontiera = Inserisci(figlio, frontiera) /* in coda con priorità
            else if figlio.Stato è in frontiera con Costo-cammino più alto then
                sostituisci quel nodo frontiera con figlio
    
```

g(n)

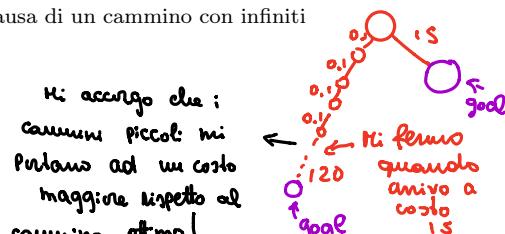
Se il figlio ha costo minore
di quello stesso nodo in frontiera,
ri restituirlo!

7.12.3 Analisi complessità UC

- Strategia Completa: SI se il costo degli archi x sia tale che $x \geq \alpha > 0$ ⁶
 - Strategia Ottimale: SI se il costo degli archi x sia tale che $x \geq \alpha > 0$
 - Complessità Tempo: $O(b^{1+\lfloor C^*/\alpha \rfloor})$
 - Complessità Spazio: $O(b^{1+\lfloor C^*/\alpha \rfloor})$
- Non ci sono cammini nulli o negativi.*
- se ho delle mosse piccole, mi fermo quando il costo ↑ diventa più alto di un altro cammino*

Assumendo C^* come costo della soluzione ottima e $\lfloor C^*/\alpha \rfloor$ come numero di mosse al caso peggiore, arrotondato per difetto.

⁶dove α è una costante relativamente piccola, questo ci evita che UC non termini a causa di un cammino con infiniti passi di costo 0



Nota: Quando tutti gli archi hanno costo uguale: la sua complessità è $O(b^{l+d})$
perché avviene solo dopo aver esplorato!

7.13 Confronto finale delle strategie

	BF	UC	DF	DL	ID	Bidir.
Completa	SI	SI (-)	NO	SI (+)	SI	SI
Ottimale	SI (*)	SI (-)	NO	NO	SI (*)	SI
Tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\alpha \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Spazio	$O(b^d)$	$O(b^{1+\lfloor C^*/\alpha \rfloor})$	$O(b*m)$	$O(b*l)$	$O(b*d)$	$O(b^{d/2})$

(*) se gli operatori hanno tutti lo stesso costo

(-) per costo degli archi x tale che $x \geq \alpha > 0$

(+) per problemi per cui si conosce un limite alla profondità della soluzione (se $d < l$)

8 Ricerca Euristica

La ricerca esaustiva non è praticabile in problemi di complessità esponenziale, come ad esempio gli scacchi che portano ad una complessità di 10^{120} . Possiamo invece usare la conoscenza del problema e l'esperienza per riconoscere cammini più promettenti, evitando di generarne di inutili e costosi. La conoscenza euristica non evita la ricerca ma la riduce e sotto certe condizioni può essere completa e ottimale.

Per indirizzare la ricerca utilizziamo una funzione di valutazione $f(n)$ che ci indica la qualità dello stato. La funzione di valutazione f include h , detta funzione di valutazione euristica.

Funzione di valutazione: $f(n) = g(n) + h(n)$ dove:

- $h : n \rightarrow \mathbb{R}$ (la funzione si applica al nodo n , ma dipende solo dallo stato - n .Stato)
- $g(n)$ è il costo del cammino

Esempi di euristica h possono essere la distanza in linea d'aria tra due città, numero di caselle fuori posto nel gioco dell'otto, numero di pezzi sulla scacchiera...

8.1 Algoritmo Best-First - BFH

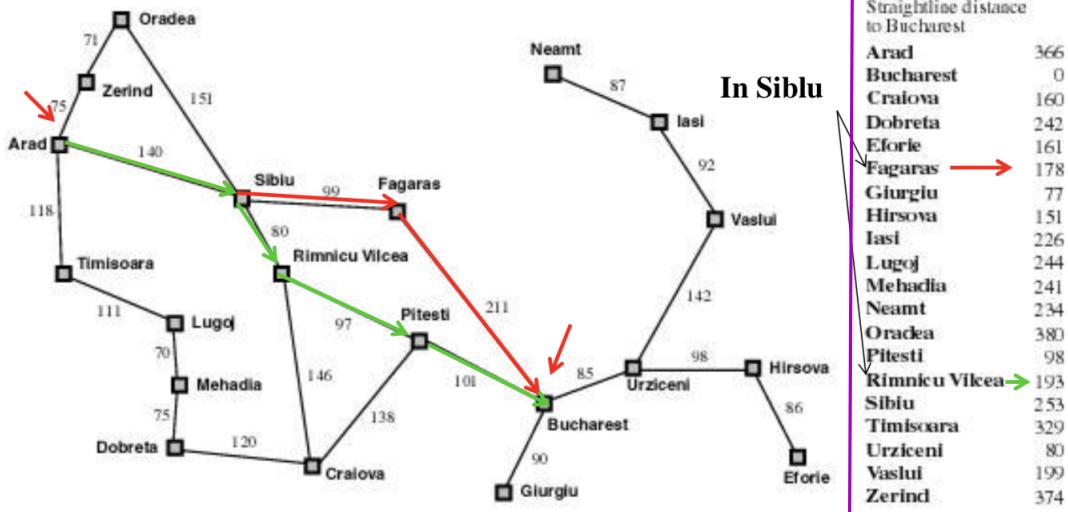
Best-First-Heuristic si basa sullo stesso algoritmo di Uniform Cost (UC) ma con uso della funzione di valutazione f per la coda con priorità. La scelta di $f(n)$ quindi determina la strategia di ricerca. Ad ogni passo si sceglie il nodo sulla frontiera per cui il valore della f è migliore (migliore può assumere significati diversi in base al problema, ad esempio nel problema dell'itinerario significa "minore"). La $h(n)$ contenuta nella funzione di valutazione ha come valore il costo stimato del percorso sul cammino dal nodo n al nodo goal.

Si sceglie il modo con $f(n) = g(n) + h(n)$ migliore in base al problema

8.2 Algoritmo Greedy-Best-First - GBF

È un caso speciale dell'algoritmo Best-First in cui la funzione di valutazione è uguale alla funzione di valutazione euristica, cioè $f(n) = h(n)$. In pratica l'algoritmo cerca di espandere il nodo più vicino al goal secondo l'euristica.

Sceglie il nodo successivo in base alla distanza in linea d'aria da Bucharest



Ci troviamo In(Arad) e aggiungiamo alla frontiera tutti i nodi adiacenti, la coda è ordinata secondo l'euristica della distanza in linea d'aria perché $f(n) = h(n)$. Da In(Arad) quindi andiamo Go(Sibiu), e ci troviamo In(Sibiu), vengono aggiunti i nodi adiacenti alla coda e per primo troviamo Faragás, Go(Faragás) e poi Go(Bucharest) perché ha distanza (ovviamente) 0 dal goal ed è primo nella coda. Notiamo però che l'algoritmo Greedy-Best-First ha trovato un cammino ma non l'ottimo, che sarebbe quello in verde.

8.3 Algoritmo A → Si sceglie una funzione euristica $h(n)$!

Un algoritmo A è un algoritmo Best-First⁷ con una valutazione dello stato del tipo:

$$f(n) = g(n) + h(n) \text{ dove:}$$

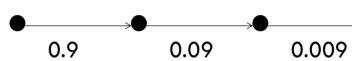
- $h(n) \geq 0$
- $h(\text{goal})=0$
- $g(n)$ è il costo del cammino per arrivare a n
- $h(n)$ è una stima del costo per raggiungere il goal da un nodo n

L'algoritmo A nel caso in cui $h(n) = 0$ ottengo $f(n) = g(n) \rightarrow$ Uniform Cost. (uc)

Nel caso in cui $g(n) = 0$ ottengo $f(n) = h(n) \rightarrow$ Greedy-Best-First.

8.3.1 Completezza Algoritmo A

Teorema: L'algoritmo A è completo se $g(n) \geq d(n) * \alpha$ dove $d(n)$ è la profondità al nodo n e $\alpha > 0$ è il costo minimo dell'arco. Questo ci garantisce che non ci siano situazioni del tipo



e che il costo lungo un
cammino non cresca "all'infinito"

⁷la frontiera è ordinata secondo il valore di f

Questo perché non esiste una catena infinita di nodi con costo $\leq f(n_i)$
 ↑ (per ipotesi)

8.3.2 Dimostrazione Completezza di A

Sia $n_1, n_2, \dots, n_i, \dots, n_k$ un cammino soluzione.

n_i è un nodo sulla frontiera di un cammino soluzione quindi prima o poi sarà espanso.

Se non verrà trovata una soluzione prima, il nodo n_i sarà espanso e i nodi del figlio aggiunti alla frontiera, tra questi anche il successore sul cammino soluzione. Il ragionamento si ripete fino a dimostrare che anche il nodo goal sarà selezionato per l'espansione.

8.4 Algoritmo A^*

L'algoritmo è identico a UC ad eccezione del fatto che A^* utilizza $f(n) = g(n) + h(n)$ per ordinare la coda. (dove h e g hanno le proprietà dell'algoritmo A e h è euristica ammissibile → def in seguito)

Definiamo una funzione di valutazione ideale detta anche oracolo, che idealizza la funzione di valutazione perfetta (cammino minimo).

Funzione di valutazione ideale: $f^*(n) = g^*(n) + h^*(n)$ dove

- $f^*(n)$ è il costo del cammino minimo da radice a goal passando per n
- $g^*(n)$ costo del cammino minimo da radice a n
- $h^*(n)$ costo del cammino minimo da n a goal

Normalmente $g(n) \geq g^*(n)$ perché il costo del cammino è \geq del costo del cammino migliore. Mentre $h(n)$ è una stima di $h^*(n)$, si può andare in sottostima o sovrastima dalla distanza della soluzione.

Una euristica è ammissibile se $\forall n. h(n) \leq h^*(n)$ cioè h è sottostima (lower-bound condition).

Importante

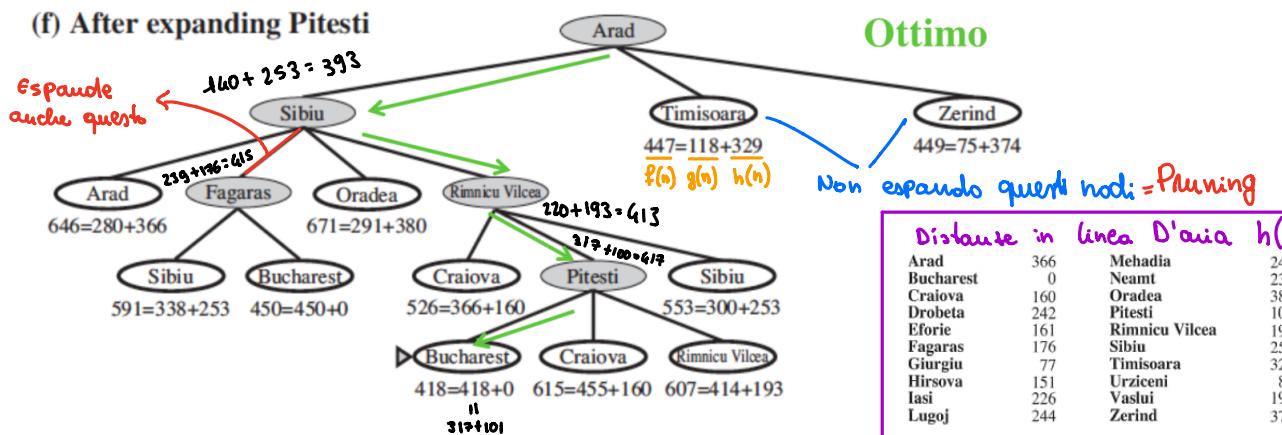
Possiamo quindi ora dare una definizione di un algoritmo A^* :

Un algoritmo A^* è un algoritmo A in cui h è una funzione euristica ammissibile. (Gli algoritmi A^* sono ottimali).

(quindi anche BreadthFirst con passi a costo costante e UC sono ottimali perché sono A^* con $h(n)=0$)

8.4.1 Esempio A^* sul problema dell'itinerario

Ho come euristica la distanza in linea d'aria, quindi è una sottostima \Rightarrow Euristica Ammissibile.



8.4.2 Osservazioni su A^*

La componente $g(n)$ invece fa sì che si abbandonino cammini che vanno troppo in profondità.

Una euristica sottostima può farci fare del lavoro inutile ma non ci fa perdere il cammino migliore, una sovrastima invece può farci perdere la soluzione ottimale a causa di un taglio che però in realtà poteva essere buono.

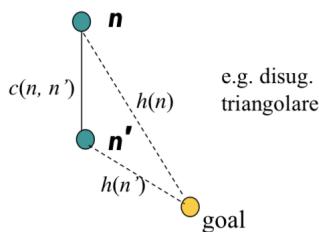
8.4.3 Ottimalità di A^*

Come abbiamo già detto gli algoritmi A^* sono ottimali.

- nel caso di ricerca su albero l'uso di una euristica ammissibile è sufficiente a garantire l'ammissibilità e quindi l'ottimalità di A^* .
- nel caso di ricerca su grafo abbiamo bisogno di definire una proprietà chiamata consistenza (o monotonicità).

8.4.4 Euristiche Consistenti (o monotone)

Un'eufistica consistente ci arricchisce che la prima volta che raggiungiamo uno stato è sul cammino ottimo, quindi non abbiamo mai da raggiungere uno stato alla frontiera e mai cambiare la lista reached.



Una eufistica è consistente se:

- $h(\text{goal}) = 0$
- $\forall n. h(n) \leq c(n, \text{azione}, n') + h(n')$ dove n' è successore di n e $h(n')$ consistenza locale ne segue che $f(n) \leq f(n')$

Se $h(n)$ è consistente la $f(n)$ non decresce mai lungo i cammini, da cui il termine monotona.

Una eufistica monotona è ammissibile (ci sono però eufistiche ammissibili non monotone ma sono molto rare). Le eufistiche monotone garantiscono che la soluzione meno costosa venga trovata per prima e quindi sono ottimali anche nel caso di ricerca su grafo.

8.4.5 Dimostrazione ottimalità di A^*

Se $h(n)$ è consistente, i valori di $f(n)$ lungo un cammino sono crescenti.

$$\begin{aligned} h(n) &\leq c(n, a, n') + h(n') \\ g(n) + h(n) &\leq g(n) + c(n, a, n') + h(n') // sommo g(n) da entrambe le parti \\ g(n) + h(n) &\leq g(n') + h(n') // perché g(n) + c(n, a, n') = g(n') \\ f(n) &\leq f(n') \end{aligned}$$

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

Non è possibile che esista un nodo n' della frontiera sul cammino ottimo con $f'(n)$ minore, poiché sarebbe già espanso

8.4.6 Vantaggi di A^*

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

Assumendo C^* come costo della soluzione ottima.

Pruning: alcuni nodi non vengono espansi a causa delle regole sopra scritte risparmiando così memoria e rimanendo ottimali. Quindi una $h(n)$ opportuna fa tagliare molto.

$$\text{Distanza di Manhattan} = |x-x_g| + |y-y_g|$$

Cercheremo quindi una $h(n)$ il più alta possibile tra le ammissibili poiché se molto bassa i nodi restano sempre minori di C^* e li espando tutti.

A^* è completo (discende dalla completezza di A), A^* con euristica monotona è ottimale. Il problema quindi è quale euristica utilizzare? Lo spazio in memoria è ancora molto grande $O(b^{d+1})$ (il costo in tempo dipende dall'euristica utilizzata).

8.5 Costruire le euristiche di A^*

A parità di ammissibilità una euristica può essere più efficiente di un'altra nel trovare il cammino soluzione migliore. Questo dipende dal grado di informazione posseduto dall'euristica.

$$h(n) = 0 \text{ // grado di informazione minimo}$$

$$h^*(n) \text{ //massimo grado di informazione (oracolo)}$$

In generale per le euristiche ammissibili $0 \leq h(n) \leq h^*(n)$.

Più informata significa più efficiente:

Se $h_1 \leq h_2$, i nodi espansi da A^* con h_2 sono un sottoinsieme di quelli espansi da A^* con h_1 . Cioè A^* con h_2 è almeno efficiente quanto A^* con h_1 . (In questo caso si dice che h_2 domina su h_1)

Una euristica più informata (accurata) riduce lo spazio di ricerca ma è tipicamente più costosa da calcolare (più informata \rightarrow taglia di più).

8.6 Valutare Algoritmi di ricerca euristica

Un modo per caratterizzare la qualità di una euristica è il fattore di diramazione effettivo b^* .

Se:

- N : numero di nodi generati da A^*
- d : profondità della soluzione

ogni nodo ha lo stesso numero di figli (tutte le foglie)

allora b^* è il fattore di diramazione effettivo che avrebbe un albero uniforme di profondità d per ottenere un albero con $N + 1$ nodi.

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

Ad esempio, se A^* trovasse una soluzione alla profondità 5, usando 52 nodi, il fattore di diramazione effettivo b^* sarebbe di 1.92.

Sperimentalmente una buona euristica ha un b^* abbastanza vicino a 1 (< 1.5) in modo tale da rendere problemi con dimensionalità alta, risolvibili in un tempo computazionale ragionevole.

Possiamo vedere come cambia in base ad alcuni dati:

- con $b^* = 2$ otteniamo a profondità $d = 6$, $N = 127$
- con $b^* = 1.7$ otteniamo a profondità $d = 6$, $N = 57$

Possiamo notare come migliorando di poco l'euristica allo stesso livello abbiamo espanso la metà dei nodi. Quindi migliorando l'euristica a parità di nodi espansi riesco a raggiungere una profondità doppia.

OSS:

- 1) Quasi tutti i prob. dell'IA sono di costo esponenziale, ma c'è Esonenziale e Esonenziale
- 2) L'euristica può migliorare di molto l'esplorazione degli spazi rispetto alla ricerca cieca
- 3) Migliorando l'euristica esploro uno spazio molto più grande

8.7 Inventare euristiche

- **Rilassamento del problema:** rimuovo dei vincoli (versioni semplificate). Ad esempio nel gioco dell'otto rimuovo i vincoli di mosse solo nelle caselle adiacenti libere (distanza Manhattan dalla sua corretta posizione).
- **Massimizzazione euristiche:** se ho k euristiche ammissibili senza che alcuna sia migliore dell'altra allora ogni volta conviene prendere il massimo dei loro valori: $h(n) = \max\{h_1(n), h_2(n), \dots, h_k(n)\}$. Se tutte le h_i sono ammissibili allora anche h lo è, e domina su tutte le altre.
- **(Sottoproblemi) Database di pattern:** ho dei database in cui sono memorizzati dei pattern di sottoproblemi con relativo costo e soluzione. Nel caso di database di pattern normali la somma delle euristiche di questi sottoproblemi non è accurata perché potrebbe essere che due soluzioni tra due sottoproblemi interferiscono l'una con l'altra. Nel caso invece di database di pattern disgiunti è consentita la somma dei costi.
- **Combinazione lineare:** si esegue una combinazione lineare di euristiche diverse: $h(n) = c_1h_1(n) + c_2h_2(n) + \dots + c_kh_k(n)$
- **Apprendere dall'esperienza:** faccio girare il programma raccogliendo dati in forma di coppie (stato, h^*) e uso i dati per capire come predire la $h(n)$ con algoritmi di apprendimento induttivo.

8.8 Algoritmi evoluti basati su A^*

Sono algoritmi che si preoccupano di occupare meno memoria⁸.

■ nodi da espandere
↑ con $f(n)$ migliore

8.8.1 Beam Search

La Beam Search salva in frontiera ad ogni passo solo i k nodi più promettenti dove k è l'ampiezza del raggio (), ovviamente l'algoritmo non è più completo così facendo.

8.8.2 A^* con approfondimento iterativo - IDA*

IDA* combina A^* con ID, ad ogni iterazione si ricerca in profondità con un limite (cut-off) dato dal valore della funzione $f(n)$ e non dalla profondità. Il valore f-limit viene aumentato ad ogni iterazione fino a trovare la soluzione (come in ID si aumenta di un livello alla volta). Ma di quanto aumento?

- Nel caso di costo fisso di ogni azione il limite viene aumentato di questo costo.
- Nel caso in cui i costi siano variabili il limite viene aumentato del costo minimo oppure ad ogni passo scelgo il valore minimo delle $f(n)$ scartate perché in quanto superavano il limite all'iterazione precedente.

IDA* è completo e ottimale se:

- le azioni hanno costo costante k e l'f-limit viene aumentato di k
- le azioni hanno costo variabile e l'incremento di f-limit è \leq del costo minimo degli archi (\mathcal{E})
- il nuovo f-limit è il minimo valore $f(n)$ dei nodi generati ed esclusi all'iterazione precedente

IDA* occupa $O(b * d)$ memoria.

⁸Le limitazioni di memoria possono rendere un problema intrattabile dal punto di vista computazionale

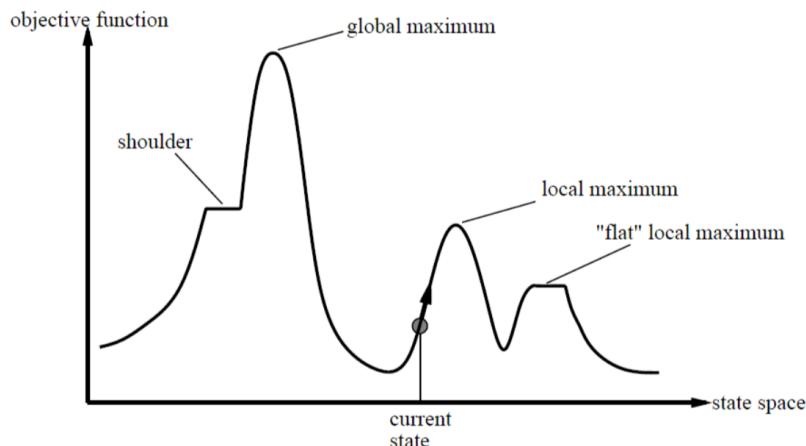
9 Ricerca Locale

Vogliamo migliorare al max la funz. obiettivo

Gli algoritmi di ricerca locale sono adatti per problemi in cui la sequenza delle azioni non è importante, Essi non sono sistematici, tengono traccia solo del nodo corrente e si spostano su nodi adiacenti. Non tengono traccia dei cammini poiché non servono in uscita, questo li rende efficienti in memoria e possono trovare soluzioni ragionevoli anche in spazi molto grandi e infiniti (continui).

9.1 Spazio degli stati

Lo stato migliore viene valutato da una funzione obiettivo f .



Uno stato ha una altezza che corrisponde al valore della funzione obiettivo (funzione costo euristico). Un algoritmo provoca movimento sulla superficie e il suo compito è quello di trovare il minimo o il massimo della funzione in base alla richiesta del problema (se stiamo cercando un costo cerchiamo il minimo).

9.2 Algoritmo Hill-Climbing

E' un algoritmo di ricerca locale greedy. Vengono generati i successori e valutati, viene scelto un nodo che migliora la valutazione dello stato attuale (non si tiene traccia degli altri nodi). Quale nodo scelgo?

- il migliore: Hill-Climbing a salita rapida.
- uno a caso tra quelli che migliorano: Hill-Climbing stocastico.
- il primo trovato: Hill-Climbing con prima scelta.

Se non ci sono stati successori migliori l'algoritmo termina. Non c'è frontiera a cui ritornare, si tiene in memoria solo lo stato corrente.

```

function Hill-climbing (problema)
    returns uno stato che è un massimo locale
    nodo-corrente = CreaNodo(problema.Stato-iniziale)
    loop do
        vicino = il successore di nodo-corrente di valore più alto
        if vicino.Valore  $\leq$  nodo-corrente.Valore then
            return nodo-corrente.Stato // interrompe la ricerca
        nodo-corrente = vicino
        // (altrimenti, se vicino e' migliore, continua)

```

Figure 2: Codice algoritmo Hill-Climbing

9.2.1 Problemi e Miglioramenti per Hill-Climbing

Essendo Hill-Climbing un algoritmo greedy, prende il nodo successivo senza pensare a dove lo porterà in futuro. Per questo l'algoritmo può arrestarsi su una soluzione che in realtà è solo un massimo/minimo locale oppure ritrovarsi su "pianori". Possiamo però eseguire dei miglioramenti:

- Consentire un numero limitato di mosse laterali, cioè l'algoritmo si modifica fermandosi quando $nodoVicino.Valore < nodoCorrente.Valore$ invece che \leq .
- Si sfrutta Hill-Climbing stocastico scegliendo a caso tra le mosse in salita. L'algoritmo converge più lentamente ma a volte trova soluzioni migliori.
- Si sfrutta Hill-Climbing con prima scelta, può generare mosse a caso fino a trovarne una migliore dello stato corrente, diventa più efficace quando i successori sono molti.
- Si sfrutta Hill-Climbing con ravvio casuale (random restart), cioè si riparte da un punto scelto a caso. Se la probabilità di successo è p saranno necessarie in media $1/p$ ripartenze per trovare la soluzione. Questo algoritmo è tendenzialmente completo.

9.3 Algoritmo Tempra Simulata (Simulated Annealing)

Hill-Climbing non esegue mai mosse in discesa, quindi viene facile capire che è garantita l'incompletezza. L'algoritmo di Tempra Simulata combina Hill-Climbing con una scelta stocastica ben studiata, ad ogni passo si sceglie un successore a caso:

- se migliora lo stato viene espanso (migliora lo stato significa che la funzione di valutazione è maggiore, cioè $[\Delta E = f(n') - f(n)] \geq 0$)
- altrimenti se peggiora lo stato (caso $[\Delta E = f(n') - f(n)] < 0$) scelgo il nodo n' con probabilità $p = e^{\Delta E/T}$ dove T è un parametro che nel progredire dell'algoritmo decresce, facendo così decresce anche p rendendo improbabili le mosse peggiorative.

Il valore per cui T decresce è dato in input come parametro¹⁰, se T diminuisce lentamente si raggiunge un ottimo globale con probabilità tendente ad 1.

¹⁰I valori di T iniziali determinati sperimentalmente sono tali che $p = e^{\Delta E/T}$ sia all'incirca 0,5

9.4 Algoritmo Local Beam

Tenere un solo nodo in memoria può sembrare una soluzione estrema al problema dello spazio... Local Beam la versione locale della Beam Search, si tengono in memoria K stati anziché uno solo. Si parte con K stati generati randomicamente e ad ogni passo si generano tutti i successori dei K stati:

- Se si trova un goal ci si ferma
- Altrimenti si prosegue con i K migliori tra quelli della lista completa

È diverso dalla Beam Search normale perché tengo in memoria solo i K migliori del passo corrente e non di tutta la storia. Possiamo notare come se $K=1$ abbiamo Hill-Climbing con prima scelta mentre se $K=\infty$ ho Hill-Climbing a salita rapida.

9.4.1 Local Beam Search Stocastica

Si introduce un elemento di casualità. Al posto di scegliere i migliori K successori dalla lista completa, si scelgono i K successori in modo randomico ma con probabilità maggiore di prendere i migliori.

9.5 Algoritmi Genetici - GA

Gli algoritmi genetici sono varianti della Beam Search Stocastica in cui gli stati successori sono ottenuti combinando due stati genitore. Chiamiamo fitness il valore della funzione obiettivo.

Abbiamo una popolazione iniziale, cioè un insieme di K stati iniziali, generati casualmente. Ogni individuo (stato) è rappresentato da una stringa definita su un alfabeto finito. Ogni individuo viene valutato da una funzione di fitness, si selezionano gli individui per gli accoppiamenti sulla base di una probabilità proporzionale alla fitness. Le coppie scelte¹¹ danno vita ad una "generazione" successiva con la combinazione di due metodi:

- Crossover (combinando materiale genetico): per ogni coppia viene scelto un punto di crossing over e ogni stato viene diviso in due parti ottenendo in totale 4 parti. Vengono poi generati due figli scambiandosi i pezzi ottenuti.
- Con un meccanismo casuale aggiuntivo di mutazione genetica: viene infine effettuata una mutazione casuale che dà luogo alla prossima generazione

La popolazione ottenuta dovrebbe esser migliore.

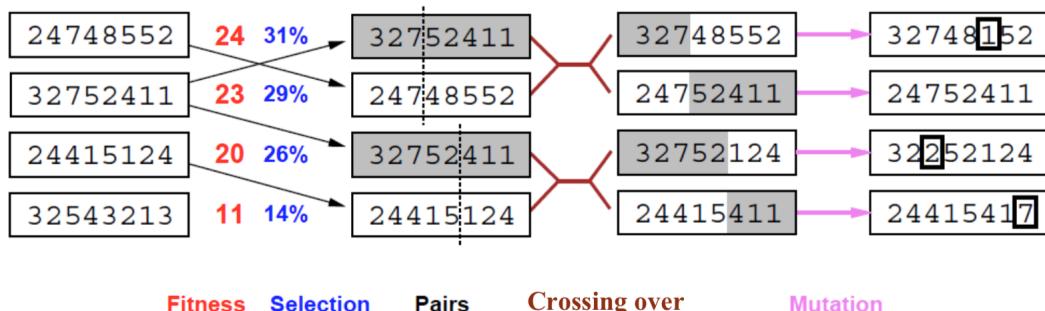


Figure 3: Esempio crossover + mutazione casuale

Gli algoritmi genetici combinano la tendenza a salire della Beam Search Stocastica e la possibilità di interscambio di informazioni tra thread paralleli. Però il punto critico di questi algoritmi è la rappresentazione del problema in stringhe.

¹¹Nel problema delle regine sceglio le coppie che non si "attaccano"

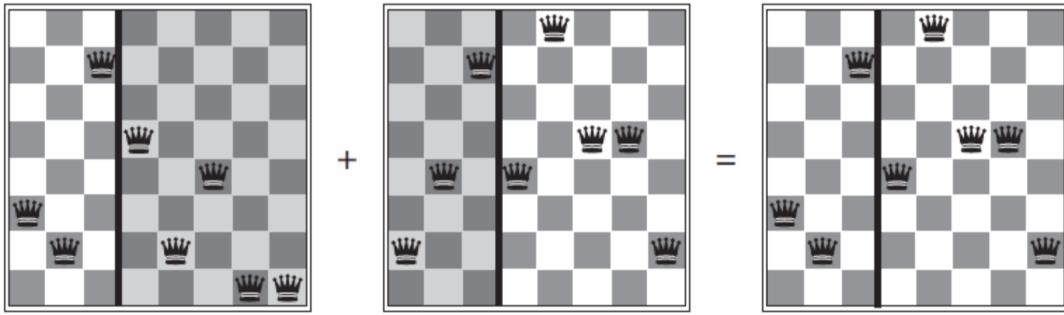


Figure 4: Esempio sul problema delle 8 regine

9.6 Gradiente (Hill-Climbing iterativo)

Molti casi nel mondo reale hanno spazi di ricerca infiniti. Lo stato viene descritto da variabili continue x_1, x_2, \dots, x_n rappresentate anche da un vettore x (ad esempio nello spazio 3D abbiamo $x = (x_1, x_2, x_3)$). Avere più dimensioni può sembrare ostico, ma abbiamo molti strumenti matematici che ci permettono di semplificare.

Se la funzione è continua e differenziabile, il minimo e il massimo può essere cercato utilizzando il gradiente, che restituisce la direzione di massima pendenza nel punto.

Data f obiettivo:

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

dove ∂ è la derivata parziale e ∇ è inteso come calcolo vettoriale. In alcuni casi possiamo trovare il massimo risolvendo l'equazione $\nabla f = 0$ (calcolo la derivata, vedo dove si annulla e ottengo i punti di min/max), ma in altri casi non è possibile risolverla in forma chiusa. Definiamo Hill-Climbing iterativo come

$$x_{new} = x_{old} + \eta \nabla f(x)$$

dove η è lo step-size (di quanto aumentare).

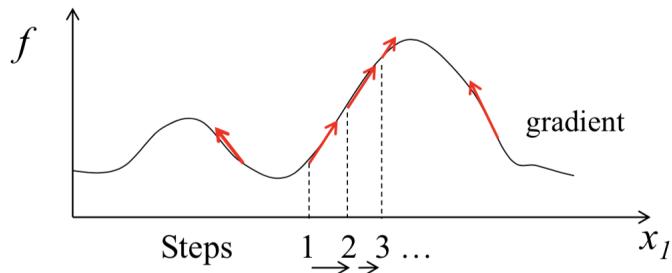


Figure 5: Esempio con una dimensione x_1 , gli spostamenti sono guidati dal gradiente

10 Oltre la ricerca classica (Ambienti parz. osservabili e non deterministici)

Gli agenti risolutori di problemi "classici", visti fino ad ora, assumono ambienti completamente osservabili e deterministici. Questo gli permette di esplorare offline lo spazio degli stati in ricerca di un goal e restituendo una sequenza di azioni che può essere eseguito senza imprevisti per raggiungere l'obiettivo. Per avvicinarci alla realtà dobbiamo riconsiderare il nostro ambiente valutando azioni non deterministiche e ambienti parzialmente osservabili.

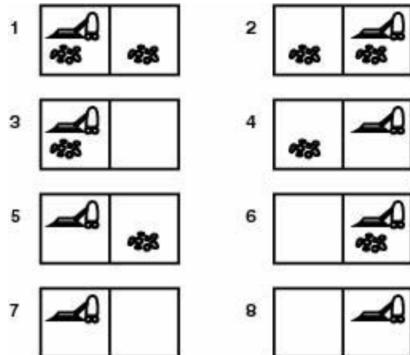
In un ambiente parzialmente osservabile e non deterministico le percezioni sono importanti perché restringono gli stati possibili e informano sull'effetto dell'azione eseguita. L'agente che si trova in questo ambiente può elaborare una strategia che tiene conto delle differenti eventualità: un piano di contingenza.

10.1 Problema dell'aspirapolvere non deterministico

Analizziamo il comportamento dell'aspirapolvere:

- Se aspira in una stanza sporca → la pulisce, ma talvolta pulisce anche la stanza adiacente.
- Se aspira in una stanza pulita a volte rilascia dello sporco.

Il modello di transizione restituisce un insieme di stati ma l'agente non sa in quale si troverà. Il piano di contingenza sarà un piano (un albero) condizionale con probabili cicli.



Analizzando lo spazio degli stati, Risultato(Aspira,1)¹² mi può portare negli stati 5 o 7. Per rappresentare il problema possiamo usare gli alberi di ricerca AND-OR.

10.2 Alberi AND-OR

- Nodi OR → scelte dell'agente.
- Nodi AND → le diverse eventualità da considerare tutte (ambiente non deterministico!).

Una soluzione a un problema di ricerca AND-OR è un albero che ha un nodo obiettivo in ogni foglia, specifica un'unica azione nei nodi OR e include tutti gli archi uscenti da nodi AND che visita (possiamo notare che sono le caratteristiche della soluzione in foto sottostante).

¹²esegue l'azione di aspirare mentre si trova nello stato 1

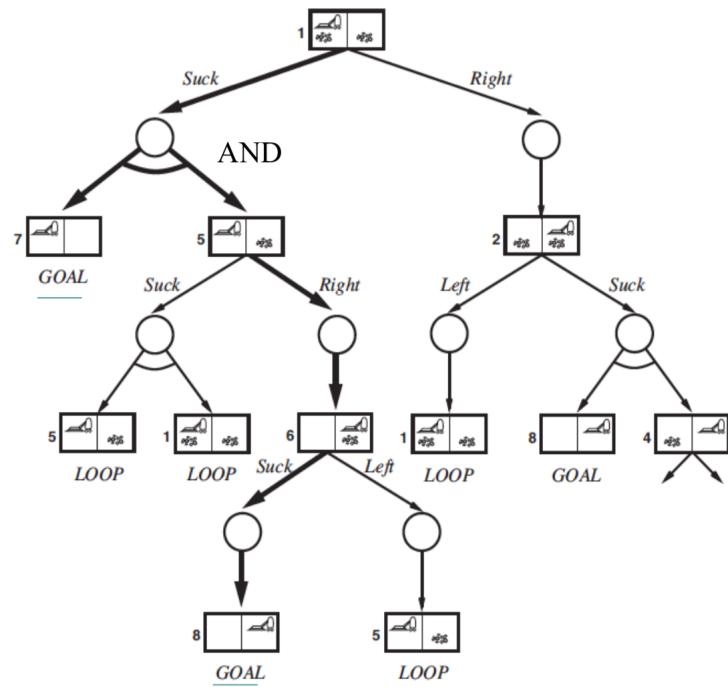


Figure 6: Esempio albero AND-OR problema dell’aspirapolvere. La soluzione è in grassetto.