Roberto Esposito
15 December 2020

Formulazione di un problema

La progettazione dell'agente consiste in 3 fasi:

1. **Formulazione**;
2. **Ricerca**;
3. **Esecuzione**;

La **ricerca di una soluzione** consiste nel trovare una soluzione ovvero una sequenza di azioni, tali azioni formano così un albero di ricerca con lo stato iniziale alla radice, i rami sono azioni e i nodi corrispondono a stati nello spazio degli stati del problema.

Dato lo stato iniziale bisogna controllare che l'obiettivo non sia stato raggiunto dopodiché si considereranno le varie azioni, espandendo lo stato corrente e generando così un nuovo insieme di stati.

Si parla di **cammino ciclico** quando uno stato può essere ripetuto più volte ed in alcuni casi questi cicli possono causare il fallimento degli algoritmi, andando in loop oppure trasformando il problema computazionalmente troppo oneroso. Un cammino ciclico non è mai migliore di quello che si ottiene dallo stesso cammino rimuovendo un ciclo ovviamente se i costi dei passi sono non negativi.

I cammini ciclici sono un caso particolare dei cammini ridondanti. Un **cammino è ridondante** se è un cammino in cui esistono due o più modi per passare da uno stato all'altro. In alcuni casi è impossibile farne a meno ad esempio quando le azioni sono reversibili. In altri termini seguendo cammini ridondanti si può trasformare un problema trattabile in uno intrattabile.

Per evitare di addentrarsi in cammini ridondanti è necessario ricordarsi dove si è passati introducendo una struttura dati chiamata **insieme esplorato** o lista chiusa. Tale tecnica ha una proprietà ovvero la **frontiera** separa il grafo degli stati nella regione esplorata e in quella inesplorata, ogni cammino che passa da un nodo esplorato ad un nodo inesplorato deve sicuramente passare per la frontiera.

Si definiscono nodi **foglia/intermedi** ovvero i nodi privi di figli in un determinato momento della ricerca, poi successivamente possono non essere più foglia se vengono espansi.

L'insieme di tutti i nodi foglia in un determinato momento della ricerca è detto frontiera.

Strutture Dati

Per ogni nodo n dell'albero abbiamo bisogno di 4 componenti:

1. n.stato: lo stato dello spazio degli stati a cui corrisponde il nodo;

2. n.padre: il nodo dell'albero di ricerca che ha generato il nodo corrente;
3. n.azione: l'azione applicata al padre per generare il nodo;
4. n.costo-cammino: il costo g(n) del cammino che va dallo stato iniziale al nodo;

Una volta memorizzati i singoli nodi è necessaria una struttura dove conservare tutti i nodi, la struttura dati più adatta è una coda:

1. **LIFO** - last in first out
2. **FIFO** - first in first out
3. **Coda con priorità**

Valutazione Algoritmi di Ricerca

Completezza: l'algoritmo garantisce di trovare una soluzione, se questa esiste ?

Ottimalità: l'algoritmo garantisce di trovare la soluzione ottima ?

Complessità temporale: l'algoritmo quanto tempo impiega per trovare una soluzione ?

Complessità spaziale: di quanta memoria ha bisogno l'algoritmo per trovare una soluzione ?

Variabili che si usano per valutare la complessità:

1. b = fattore di ramificazione/branching factor
2. d = profondità della soluzione a costo minimo
3. m = massima profondità dello spazio degli stati

Algoritmi di Ricerca Non Informata

Gli algoritmi di ricerca non informata fanno riferimento alle strategie di ricerca che non dispongono di informazioni aggiuntive sugli stati oltre a quella fornita nella definizione del problema.

La principale differenza fra le varie strategie di ricerca consiste nell'ordine in cui vengono espansi i nodi.

La natura di questi algoritmi non è in grado di stimare quanto un nodo non obiettivo sia "promettente" per la risoluzione del problema a differenza degli algoritmi di ricerca informata e euristica.

Ricerca in Ampiezza

Questa ricerca consiste di:

1. Espandere il nodo radice

2. Espandere i nodi generati dalla radice
3. Espandere i loro successori

Ricerca in ampiezza: strategia di ricerca in cui tutti i nodi di profondità d sono espansi prima di quelli di profondità d+1.

La ricerca in ampiezza è una strategia sistematica di ricerca ovvero è in grado di trovare sempre una soluzione se essa esiste. È in grado di trovare sempre la soluzione con il cammino più breve, ma bisogna sottolineare che il cammino più breve non è necessariamente quello con il costo minore.

Viene sempre scelto per l'espansione il nodo non espanso più vicino alla radice.

Questa ricerca può essere implementata con una semplice coda FIFO per la frontiera. Di conseguenza i nuovi nodi vanno in fondo alla coda e i nodi vecchi vengono espansi per primi.

Completezza: è completo perché prima o poi analizza tutti i nodi, per arrivare alla soluzione d devo aver espanso tutti i nodi alle profondità precedenti.

Ottimalità: se il costo di cammino è una funzione monotona non decrescente allora la ricerca è ottima, ma in generale non lo è.

Complessità temporale: è esponenziale O(b elevato a d)

Complessità spaziale: è esponenziale perché l'algoritmo memorizza ogni nodo espanso e quindi sarà necessariamente O(b elevato a d)

Ricerca di Costo Uniforme

Anziché meno espandere il nodo meno profondo si espande il nodo n con il minimo costo di cammino g(n). In questo caso non parleremo di coda FIFO, ma di coda a priorità, in cima ci saranno i nodi di costo minore.

Due principali differenze implementative rispetto alla ricerca in ampiezza:

1. Il test obiettivo è applicato ad un nodo non appena è selezionato per l'espansione e non quando viene generato per la prima volta
2. Si aggiunge un test obiettivo nel caso in cui sia trovato un cammino migliore per raggiungere un nodo che attualmente si trova sulla frontiera

Completezza: trova sempre una soluzione andando ad espandere tutti nodi, a meno di NoOp.

Ottimalità: è ottimo perché ritorna sempre la soluzione di costo minore.

Complessità temporale: la complessità dipende dal costo di cammino e non dalla profondità per definire questo parametro si considera C* ovvero il costo della soluzione ottima e ipotizziamo che ogni azione costi almeno epsilon. La complessità temporale sarà uguale a O(b elevato a 1+C*/epsilon). C*/epsilon sta a rappresentare il costo di ogni passo fatto dall'algoritmo

Complessità spaziale: il ragionamento è analogo anche in questo caso e quindi la complessità sarà esponenziale.

Questa ricerca è più efficace di quella in ampiezza perché sia completa che ottimale, ma in alcuni casi risulta essere meno efficiente perché arrivati ad una soluzione va ad analizzare percorsi alternativi di costo minore.

Ricerca in Profondità

È l'opposto della ricerca in ampiezza, i nodi vengono espansi scendendo sui singoli rami.

Ricerca in profondità: viene sempre espanso prima il nodo più profondo nella frontiera corrente dell'albero di ricerca.

Praticamente si raggiunge subito il livello più profondo dell'albero dove i nodi non hanno successori. L'espansione di tali nodi li rimuove dalla frontiera per cui la ricerca "torna indietro" (backtracking) per riconsiderare il nodo più profondo che ha successori ancora non espansi.

In questo caso si utilizza una coda LIFO poiché verrà sempre espanso il nodo generato.

Completezza: se vi è al presenza di cicli o lo spazio degli stati ha profondità infinita l'algoritmo non terminerà mai. E quindi non è completo in tal caso

Ottimalità: automaticamente non è ottimo perché può ritornare un nodo obiettivo lontano dalla radice non tenendo conto di un eventuale presenza di un nodo obiettivo vicino alla radice

Complessità temporale: è sempre di tipo esponenziale O(b elevato a d)

Complessità spaziale: è lineare O(bd)

Con il prossimo algoritmo se ne vede un miglioramento.

Ricerca in Profondità Limitata

Opera in modo identico alla ricerca in profondità, ma si pone un limite per evitare che l'algoritmo non termini.

Ricerca in profondità limitata: con questa strategia un nodo viene espanso solo se la lunghezza del cammino corrispondente è minore rispetto al massimo stabilito.

Con questa tecnica nascerà una nuovo problematica ovvero se la profondità della soluzione è alla profondità x e il limite nostro è x-1 allora l'algoritmo non sarebbe completo.

In questo caso bisogna essere in grado di conoscere il problema per trovare una soglia ne troppo piccola ne troppo grande.

Completezza: l'algoritmo non è completo come appena detto.

Ottimalità: non è ottimo quando la soglia è minore della profondità massima quindi sempre perché altrimenti non avrebbe senso utilizzarla.

Complessità temporale: O(b elevato a l) dove l è il limite stabilito.

Complessità spaziale: O(bl) perché è necessario memorizzare un solo cammino dalla radice ad un nodo foglia, insieme ai rimanenti nodi fratelli non espansi per ciascun nodo sul cammino. Una volta che un nodo è stato espanso può essere rimosso dalla memoria non appena tutti i suoi discendenti sono stati esplorati completamente.

Ricerca con Approfondimento Iterativo

È un ulteriore miglioramento dell'algoritmo precedente

Ricerca con approfondimento iterativo: con questa strategia il limite viene incrementato progressivamente finché un nodo obiettivo non è identificato.

Questa strategia è un misto tra ampiezza e profondità perché per arrivare ad un livello si espande in ampiezza prima di aumentare la profondità.

Si preferisce questa tecnica quando lo spazio di ricerca è grande e la profondità della soluzione non è nota.

Completezza: è completo perché la ricerca lo troverà dopo aver espanso tutti i nodi che lo precedono

Ottimalità: se il costo di cammino è una funzione monotona non decrescente della profondità del nodo allora possiamo dire che la ricerca è ottima, ma in generale non lo è.

Complessità temporale: nel caso pessimo sarà O(b elevato a d)

Complessità spaziale: stesso ragionamento della ricerca con profondità limitata e quindi sarà O(bd)

Ricerca Bidirezionale

Si effettuano due ricerche in parallelo, la prima guidata dai dati e la seconda guidata dall'obiettivo.

Ricerca bidirezionale: strategia di ricerca in cui viene eseguita una prima ricerca in avanti dallo stato iniziale e l'altra all'indietro dallo stato obiettivo.

È meglio perché O(b elevato a d/2) + O(b elevato a d/2) < O(b elevato a d)

È necessario che le due ricerche si intersechino

Primo problema bisogna definire cos'è un predecessore: predecessore di uno stato x tutti gli stati che hanno x come successore. Definire un predecessore non è sempre così semplice

Secondo problema bisogna spiegare cosa si intende per obiettivo nella ricerca all'indietro basti pensare al problema delle 8 regine ad esempio qual è l'obiettivo della ricerca all'indietro ?

Il migliore tra quelli visti, ma difficile da implementare.

Completezza: si

Ottimalità: si

Complessità temporale: O(b elevato a d/2) + O(b elevato a d/2) < O(b elevato a d)
Complessità spaziale: O(b elevato a d/2) + O(b elevato a d/2) < O(b elevato a d)

Algoritmi di Ricerca Informata

La differenza principale rispetto agli algoritmi di ricerca non informata è l'avere a disposizione **conoscenze specifiche** del problema, in modo tale da essere più efficienti. In generale l'approccio che si utilizza è un approccio best-first (prima il migliore): l'algoritmo viene modificato in maniera tale da consentire l'espansione di nodi sulla base di una funzione di valutazione f(n) questa funzione utilizzerà conoscenze specifiche del problema in modo da individuare i nodi più promettenti espandendoli prima. In generale in questa strategia è identica alla ricerca a costo uniforme, ma la differenza è che la coda priorità non è più ordinata in base a costo di passo, ma in base a quanto un nodo è promettente. In cima alla coda si avranno i nodi più promettenti, mentre verso la fine della code vi sono quelli meno promettenti. Molti algoritmi di questo tipo includono una funzione euristica che indica il costo del cammino più conveniente dallo stato n allo stato obiettivo ed è h(n) e stima quanto è promettente il nodo n. Queste funzioni euristiche sono il modo in cui viene inserita informazione all'interno dell'algoritmo di ricerca. Rispetto alla ricerca a costo uniforme non cambia molto per quanto riguarda lo pseudocodice, cambia solo la coda a priorità e l'istruzione che manipolava quale nodo successivo prendere in considerazione.

ALGORITMO GREEDY

Tra gli algoritmi best-first vediamo come prima l'**algoritmo greedy** (goloso) il quale va sempre alla ricerca del nodo più vantaggioso ignorando le alternative, andando sempre alla ricerca di qualcosa più promettente e sono tra gli algoritmi più utilizzati per problemi di ottimizzazione. Gli algoritmi greedy sono utilizzati anche per l'ottimizzazione per i casi di test da lanciare in un ambiente di continuos integration

(testing di sviluppo di software complessi). L'algoritmo greedy cerca sempre di espandere il nodo più vicino all'obiettivo perché si pensa che espandendolo questo porti più rapidamente ad un nodo soluzione. Quindi la funzione di valutazione dell'algoritmo greedy è esattamente h(n), f(n) = h(n). [Esempio Bucarest: f(n)=h(n)=città più vicina in linea d'aria, è informato perché conosciamo qualcosa in più rispetto all'ambiente]. L'algoritmo greedy può trovare delle soluzioni, ma queste non sono necessariamente le migliori possibili. Inoltre l'algoritmo non è neanche completo in quanto rischia anche di entrare in "vicoli ciechi" entrando poi successivamente in un loop. Per quanto riguarda la complessità temporale ha complessità esponenziale b elevato alla m. Si può lavorare sull'euristica diminuendone la complessità. Per quanto riguarda la complessità spaziale anch'essa sarà esponenziale in quanto conserva tutti i nodi nel caso pessimo. Si può utilizzare, ma per problemi complessi non sarà efficiente.

A*

Un altro algoritmo best-first è l'**A*** che differisce dalla ricerca greedy in quanto A* combina il costo di cammino con la stima di costo potenziale, f(n) = costo + promessa, f(n)=g(n)+h(n). Siccome g(n) è il costo del cammino del nodo iniziale al nodo n ed h(n) rappresenta il costo stimato del cammino più conveniente da n all'obiettivo si può concludere dicendo che f(n) è il costo stimato della soluzione più conveniente che passa per n. Dal punto di vista pratico l'algoritmo è sempre uguale alla ricerca a costo uniforme, l'unica differenza è sempre l'ordinamento dato dalla funzione f(n). Se si è in cerca di una soluzione meno costosa allora si sceglie la f(n) più bassa. Ci possono essere casi speciali: se h(n)=0 allora si ha la ricerca uniforme, viceversa se g(n)=0 allora si ha la ricerca best first greedy. Le prestazioni di questo algoritmo: ottimalità dipende da due fattori (h(n) deve essere un'euristica **ammissibile** cioè che non sbagli mai per eccesso la stima del costo per arrivare all'obiettivo, il secondo fattore è che l'euristica deve essere **consistente** cioè per ogni nodo n e per ogni successore n' generato dall'azione a quando si trovava nello stato n il costo stimato per andare da n all'obiettivo sia: h(n) <= c(n, a, n') + h(n') ovvero una specie di disuguaglianza triangolare). Non funziona quindi con grafi con costi negativi. Per quanto riguarda la completezza richiede ovviamente un numero finito di nodi di

costo minore o uguale a C*(soluzione ottima) condizione vera se tutti i costi dei passi superano un valore finito epsilon e se b è finito. La complessità temporale anche in questo caso è esponenziale ma in maniera diversa dall'algoritmo greedy poiché in questo caso a parità di euristica nessun altro algoritmo espande meno nodi e nel caso pessimo è O(b elevato alla epsilon) per questo è più efficiente rispetto al greedy. La complessità spaziale rimane essere sempre la stessa e nel caso pessimo sarà esponenziale in m O(b elevato alla m). Con A* si riduce la complessità temporale, ma continua ad esserci il problema della complessità spaziale. Vengono viste di seguito variazioni di questo algoritmo.

BEAM SEARCH

Il nuovo algoritmo è la **Beam Search** ed è una variante di best first e non conserva in memoria tutti i nodi generati, ma si salva solo i k più promettenti dove k viene definito ampiezza del raggio (riducendo il raggio diminuisce la memoria). L'ampiezza del raggio è definita in base ad una euristica solo i nodi nel raggio vengono conservati. La Beam Search è a tutti gli effetti una soluzione greedy poiché andrà comunque ad espandere un nodo che ritiene essere più utile all'obiettivo. Con la Beam Search ad ogni livello l'algoritmo genera tutti i successori, ma in più li ordinerà in ordine decrescente in base all'euristica (utilizza una coda a priorità) dopodiché andrà ad escludere tutti quei nodi che hanno un valore minore di una certa soglia salvando solo i primi k nodi di ogni livello che poi saranno espansi ignorando i restanti. Maggiore sarà il raggio minore sarà il numero di noti potati, la scelta del raggio influenza la complessità spaziale dell'algoritmo. Questo algoritmo ottimizza la complessità spaziale che in questo caso sarà O(kb) però risulterà essere né completo né ottimale.

IDA*

Il prossimo algoritmo che viene analizzato è l'**IDA*** (iterative deepening A*) riprende il concetto dell'approfondimento iterativo (un nodo viene espanso solo se il suo livello è minore del valore limite taglio <— ricerca in profondità limitata; ricerca ad approfondimento iterativo —> il limite viene può essere incrementato piano piano). La differenza principale sta nel valore di taglio che non è più basato sulla

profondità del cammino, ma sul costo f. In altri termini il nuovo valore di taglio è l'f-costo minimo tra quelli di tutti i nodi che hanno superato il valore di taglio nell'iterazione precedente, fondamentalmente cambia relativamente poco rispetto alla ricerca ad approfondimento iterativo, cambia il modo in cui viene assegnato il taglio. Essendo un'estensione di A* ne mantiene le proprietà di completezza ed ottimalità. Il vero vantaggio lo si ottiene in termini di occupazioni di memoria perché in questo caso l'algoritmo eredita anche le proprietà dell'algoritmo ad approfondimento iterativo in quanto verrà memorizzato un solo cammino dalla radice ad un nodo foglia, insieme ai rimanenti nodi fratelli non espansi per ciascun nodo sul cammino. Una volta che il nodo è stato espanso può essere rimosso dalla memoria non appena tutti i suoi discendenti sono stati esplorati completamente, O(bd)/O(bm) [dubbio prof]. L'IDA* è da preferire alla Beam Search.

BFRS

Un altro algoritmo è la ricerca **Best-First ricorsiva BFRS** ed è una variante della ricerca best-first classica, usando però uno spazio lineare. Ad ogni iterazione l'algoritmo tiene traccia del miglior percorso alternativo. Invece di fare backtracking in caso di fallimento, interrompe l'esplorazione quando trova un nodo meno promettente (definito sulla base di f). Per quanto riguarda l'implementazione non ci sono grandi variazioni rispetto alla ricerca ricorsiva in profondità, ma invece di continuare a seguire il cammino si utilizza f-limite per tenere traccia del valore del migliore cammino alternativo. Tale ricerca è ottimale nel caso in cui l'euristica definita è ammissibile (sempre stesso ragionamento per eccesso visto prima) anche qui il vantaggio è in termini di occupazione di memoria in quanto è lineare nella profondità della più profonda soluzione ottima, O(bd). La complessità temporale è difficile da definire poiché dipende sia dalla funzione euristica che selezioniamo sia dalla frequenza dei cambiamenti del cammino ottimo, ma nel caso pessimo sarà esponenziale.

Il problema di IDA* e di best-first ricorsiva si comportano in maniera opposta agli altri algoritmi e quindi memorizzano troppe poche informazioni. Entrambi gli algoritmi dimenticano la maggior parte di ciò che fanno rischiando di espandere più e più volte gli stati già visitati in precedenza, influendo sulla complessità temporale.

SMA*

Un compromesso tra memorizzare tutto e memorizzare troppo poco è lo **SMA*** (Simplified Memory Bounded A*) che invita ad utilizzare meglio la memoria, procede come A* fino all'esaurimento della memoria quando poi si arriva alla fine si libera il nodo peggiore (quello con l'f-valore più alto). La cosa interessante, simile alla RBFS, memorizza nel nodo padre il valore del nodo dimenticato, tenendo traccia della radice di un sottoalbero dimenticato ricordandosi così quale era il cammino migliore del sottoalbero dimenticato. SMA* rigenererà un sottoalbero dimenticato quando tutti gli altri cammini avranno un comportamento peggiore di quello da lui dimenticato. SMA* è completo se la profondità del primo nodo soluzione è inferiore della dimensione della memoria espressa in nodi, se SMA* non termina la memoria prima di trovare una soluzione allora è completo. Se c'è una soluzione ottima raggiungibile, SMA* è ottimale. Il problema di questo algoritmo la ricerca passerà tra molti ipotetici cammini saltando da una parte all'altra. Quando questo accade il problema potrebbe diventare intrattabile e la complessità temporale potrebbe esplodere e si potrebbe arrivare a un problema non risolvibile nella pratica. Seppure ci sono vantaggi in complessità spaziale si potrebbero avere problemi elevati in complessità temporale.

OSSERVAZIONI SU FUNZIONE EURISTICA

La funzione euristica non è sempre semplice da definire però a patto di trovare un'euristica ammissibile e consistente è possibile utilizzare algoritmi di ricerca completi e ottimi. Come si definisce una buona funzione euristica ? Una buona euristica può migliorare di molto la capacità di esplorazione dello spazio degli strati. Migliorando anche di poco un'euristica si riesce ad esplorare uno spazio degli stati molto più vasto. Spesso bisogna "inventare" euristica seguendo delle linee guida come:

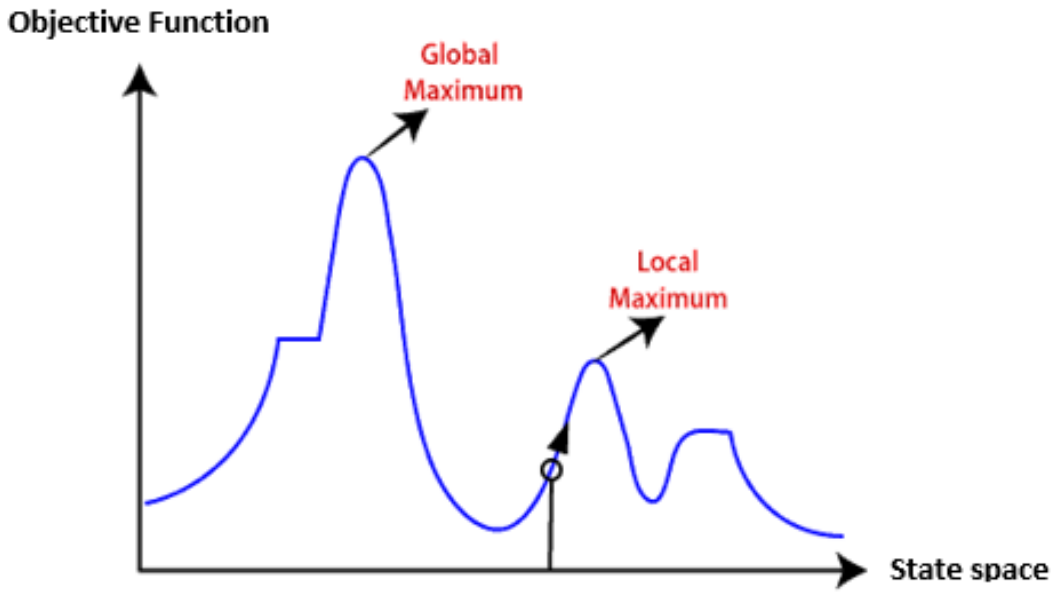
- **rilassamento dei vincoli**, se il problema è troppo complicato si può "rilassare" alcuni dei vincoli rendendo il problema meno complesso aiutando a definire un'euristica sufficientemente buona;
- **Massimizzazione di euristiche**: si può fare un mix di euristiche, se si hanno h1, h2, …, hk si possono combinare tali euristiche attraverso ad esempio una massimizzazione di esse, $h(n)=\max(h1(n), h2(n), \ldots, hk(n))$;

- **Apprendere dall'esperienza**: si possono volendo utilizzare tecniche di apprendimento per migliorare il modo in cui la ricerca viene effettivamente eseguita;

- **Combinare euristiche**: si può usare una combinazione di euristiche del tipo h(n)= c1*h1(n)+…+ck*hk(n);

Algoritmi di Ricerca Locale

Gli algoritmi di ricerca locale non hanno un test obiettivo, ma affidano le loro azioni ad una funzione obiettivo che indica quanto la ricerca sta migliorando nelle iterazioni. Questo li rende adatti alla risoluzione di classici problemi di ottimizzazione laddove non esiste un vero e proprio test obiettivo. La scelta della funzione obiettivo è fondamentale per l'efficienza del sistema e definisce l'insieme delle soluzioni che possono essere raggiunte da una soluzione s in un singolo passo di ricerca dell'algoritmo. È da notare che la ricerca locale si basa sull'**esplorazione iterativa** delle "soluzioni vicine" che possono migliorare quella corrente mediante modifiche locali. Si definisce **Struttura dei vicini** (**neighborhood**): una struttura dei vicini è una funzione F che assegna a ogni soluzione s dell'insieme di soluzioni S un insieme di soluzioni N(s) sottoinsieme di S. È importante sottolineare come la soluzione trovata da un algoritmo di ricerca locale non è sempre la soluzione ottima, ma può essere anche un "ottimo locale" cioè ottima rispetto ai cambiamenti locali. Lo spazio degli stati viene visualizzato in modo completamente visto fino a come è stato visto ora, adesso si avrà un piano cartesiano in cui l'asse delle x rappresenta i vari stati mentre l'asse delle y il valore della funzione obiettivo in un determinato stato.



A one-dimensional state-space landscape in which elevation corresponds to the objective function

Lo stato corrente indica quali sono gli altri possibili stati raggiungibili dalla ricerca, l'obiettivo di questa tipologia di algoritmi è quello di migliorare ad ogni passo il valore della funzione obiettivo. Si possono trovare **punti piatti/plateau**, ovvero massimo locale piatto, cioè andando avanti con gli stati il valore della funzione obiettivo rimane sempre lo stesso. O anche una **spalla/shoulder** ed è una porzione dello spazio di ricerca che è un massimo locale piatto, ma ha uno spigolo in salita e per alcuni algoritmi sarà complicato arrivare al massimo globale perché venendo da sinistra troveranno un massimo locale ed andando avanti troveranno un punto piatto e quindi non continueranno la loro ricerca. Per questo questi algoritmi possono ritornare una soluzione sub-ottima e non ottima. Gli algoritmi di ricerca locale si possono utilizzare sia in caso di massimizzazione che in caso di minimizzazione. Un algoritmo di ricerca locale completo è un algoritmo che trova sempre un obiettivo se questo esiste. Un algoritmo di ricerca locale è un algoritmo che trova sempre un minimo/massimo globale. Un **massimo locale** è una soluzione s tale che data una funzione di valutazione f avveri la seguente equazione: $\forall s' \in N(s), f(s) \geq f(s')$. Un **massimo globale** è una soluzione s tale che data una funzione di valutazione f avveri la seguente equazione: $\forall s, f(s_{opt}) \geq f(s)$. Viceversa per minimo locale e globale. Più è largo il neighborhood più è probabile che un massimo/minimo locale sia anche globale —> più è largo il neighborhood e migliore sarà la qualità della soluzione.

Hill-Climbing

Questo algoritmo cercherà di "scalare" lo spazio di ricerca per trovare un max/min globale. L'algoritmo termina quando il nodo vicino abbia un valore minimo rispetto a quello attuale. L'algoritmo mantiene in memoria solo il nodo corrente (stato & valore della sua funzione obiettivo). L'algoritmo non guarda al di là degli stati immediatamente vicini a quello corrente. Tale algoritmo viene alcune volte definito "ricerca locale greedy" perché trova uno stato "buono" senza pensare a come andrà avanti. L'Hill-Climbing non sempre riesce a trovare un ottimo globale per diverse ragioni:

- se ci troviamo in vicinanza di un massimo locale, si sale sul picco rimanendo bloccati nel max locale;
- la presenza di plateau ferma l'algoritmo non capendo che spostandosi di poco potrebbe raggiungere un eventuale massimo globale;
- i ridge "creste" rappresentano delle porzioni dello spazio di ricerca che presentano una brusca variazione e l'algoritmo analizzerà una serie di massimi locale da cui è difficile uscire;

Massimi locali, plateau e creste deteriorano le prestazioni dell'algoritmo Hill-Climbing. Statisticamente il problema delle 8 regine viene risolto solo nel 14% dei casi, ma l'algoritmo impiega solo 4 passi per trovare una soluzione. Tutto sommato l'Hill-Climbing non è da escludere a prescindere quando vanno soluzioni sub-ottime.

Alcuni miglioramenti dell'Hill-Climbing, se ci troviamo in una spalla possiamo "perturbare" sperando che ci troviamo in una spalla e non in un plateau, ma vi può essere il rischio che si entra in un ciclo infinito infatti si può stabilire un limite massimo al numero di mosse laterali che si possono effettuare. Questo migliora il tasso di successo dell'algoritmo, impattando la sua efficienza. Nel caso delle 8-regine si avrà un tasso di successo del 94%, ma il numero di mosse salirà a 21. Esistono diverse altre varianti dell'Hill-Climbing migliorandone l'efficacia e degradandone l'efficienza quindi dipende dalla tipologia di problema da affrontare.

Hill-Climbing Stocastico

Questa variante sceglie a caso tra tutte le mosse che vanno verso l'alto; non viene scelta immediatamente la mossa più conveniente, aumentando la possibilità di navigare lo spazio di ricerca in modo più esteso.

Hill-Climbing con Prima Scelta

Genera mosse a caso fino a trovarne una migliore dello stato corrente, cercando di allargare il raggio di azioni cercando altri stati.

Hill-Climbing con Riavvio Casuale

Può consentire alla ricerca di ripartire da un punto a caso dello spazio di ricerca, avendo però difficoltà nella generazione di uno stato casuale. Questa variante consente all'algoritmo di avviare una serie di ricerche di Hill-Climbing (come se si ripetesse più di una volta). Se ci sono pochi massimi locali e plateau allora troverà una soluzione molto velocemente, sfortunatamente non sempre è così. Ad esempio i problemi NP-hard un numero esponenziale di massimi locali, ma si può usare in questo caso con soluzioni sub-ottime.

Simulated Annealing

Un algoritmo che non "scende" mai sarà sicuramente incompleto in quanto può restare bloccato in un massimo locale. Contrariamente un'esplorazione completamente casuale è completa, ma non efficiente. Si possono combinare queste due alternative "estreme" creando un algoritmo che scende qualche volta. In metallurgia, l'annealing è un processo termodinamico che viene utilizzato per indurire i metalli o il vetro riscaldandoli ad altissime temperature per poi gradualmente raffreddarli permettendo al materiale di cristallizzare. Questa è una similitudine di ciò che avviene in questo processo, inizialmente l'algoritmo scuote molto la ricerca cercando di fare molti passi all'interno della ricerca per poi man mano assolidare all'Hill-Climbing classico concentrandosi solo su una soluzione che risulta essere ottima, ciò avviene con una certa probabilità p che decresce all'aumentare del tempo. La probabilità decresce esponenzialmente in base alla cattiva qualità della soluzione. All'inizio cambiamo con molta probabilità, successivamente decresce sempre di più. Se la temperatura si abbassa molto lentamente allora l'algoritmo troverà un ottimo globale con probabilità tendente a 1.

Local Beam

È una variante della Beam Search nel caso di un algoritmo di ricerca locale. Memorizzare un solo nodo può risultare eccessivo e si può memorizzare solo un certo numero k di nodi promettenti. Utilizzando il concetto di Beam Search lo si può applicare nella ricerca locale tenendo traccia di k stati e all'inizio partirà con k stati casualmente ed a ogni passo vengono generati i successori di tutti i k stati. Se uno di questi stati è l'obiettivo allora la ricerca termina altrimenti vengono scelti i k successori migliori dalla lista e ricomincia. In linea teorica si può vedere come Hill-Climbing a riavvio casuale con la differenza fondamentale è che prima i k processi di ricerca erano indipendenti, in questo caso la ricerca è di tipo informata come se ci fossero k ricerche in parallelo. Local Beam può essere vista come parallelizzazione della Hill-Climbing con riavvio casuale. La Local Beam potrebbe soffrire di una carenza di diversificazione tra i k stati, concentrandosi in una regione dello spazio andando a finire in un ottimo locale.

Local Beam Stocastica

Invece di seguire i k successori migliori tra i candidati se ne scelgono k a caso, esplorando lo spazio di ricerca in maniera più efficace. Con maggiore probabilità scelgo i successori migliori, ma potrebbe succedere che scelgo un nodo diverso che mi fa esplorare lo spazio. Questo approccio ricorda vagamente il processo di selezione naturale, infatti questo processo ha ispirato gli algoritmi generici che si rifanno alla teoria evolutiva di Darwin.

Algoritmi Genetici

Darwin introduce il concetto di teoria dell'evoluzione dove organismi della medesima specie evolvono tramite un processo di selezione naturale. È possibile riassumere la teoria dell'evoluzione in 4 principi:

1. Gli individui con tratti ereditatili che ben si adattano all'ambiente tendono a sopravvivere;
2. Una generazione di individui che sopravvive tende a produrre più individui;
3. Fra gli individui esiste una variazione di caratteristiche osservabili e tale variazione può essere passata dai genitori ai figli;
4. Se popolazioni della medesima specie smettono di interagire tra loro allora avviene la spedizione, ovvero le popolazioni avranno accumulato così tante differenze da essere classificate come specie diverse;

Noi ci soffermeremo sull'1 e sul 3 introducendo anche il concetto di mutazione. Definiamo l'ottimizzazione come tipo di ricerca volto a trovare un punto di ottimo tra le diverse alternative. Intorno agli anni '50 si iniziano a diffondere gli algoritmi evolutivi (non sono gli algoritmi genetici). Gli EA sono un sottotipo di algoritmi nature-ispired, non necessariamente ispirati alla teoria dell'evoluzione, ma possono ispirarsi ad altri fenomeni della natura (Ant

Colony Optimization, Particle Swarm Optimization). Gli algoritmi genetici nascono nel 1975 e inizialmente vengono definiti genetic plans, ma alla fine non cambia praticamente nulla.

Si definisce algoritmo genetico una procedura ad alto livello ispirato alla genetica per definire un algoritmo di ricerca. Con GA si definisce la procedura che definisce un algoritmo di ricerca e non è un algoritmo. Essendo gli GA una meta-euristica non è problem-specific (non risolve una singola classe di problemi di ricerca, ma molteplici), è flessibile (lo si può modificare facilmente), non garantisce l'ottimalità (produce soluzioni sub-ottimali). Un GA evolve una popolazione di individui producendo soluzioni sempre migliori finché non si raggiunge un ottimo/sub-ottimo/condizione di terminazione. Si seguono 3 passi: selezione, crossover, mutazione. La funzione obiettivo nei GA si chiama funzione di fitness cioè funzione di sopravvivenza, grado di sopravvivenza. Il fatto che si chiamano genetici significa che si ispirano alla natura e non che sono applicati alla bio-informatica, volendo li si possono applicare in quel caso, ma non solo lì.

I GA sono caratterizzati da:

1. **Codifica** degli individui come stringhe di lunghezza finita;
2. Sono **population-based** (non si evolve una singola soluzione, ma un insieme di esse contemporaneamente)
3. Sono **ciechi** (non è necessario avere informazioni di basso livello del problema);
4. Presenza di elementi di **casualità** che guidano la ricerca, ma non si arriva ad una random search;

Bisogna seguire la sequenze di 3 operazione in un algoritmo genetico:

1. **Selezione**: si copiano alcuni individui nella successiva generazione;
2. **Crossover**: accoppiamento di individui (parents) per crearne nuovi (offsprings) da aggiungere alla generazione. I genitori "incrociano" il loro corredo genetico per dare vita ai figli;
3. **Mutazione**: modifica casuale di una parte di tutto il corredo genetico;

La mutazione è importante perché selezione e crossover possono portare avanti soluzioni ottimali e con la mutazione si può dare un "pizzico" di casualità per esplorare meglio lo spazio di ricerca. La condizione di terminazione dipende da noi ad esempio possiamo continuare fino a quando migliora, nel momento in cui peggiora allora ci fermiamo, scartiamo questa soluzione e riprendiamo l'ultima che era la migliore fino a quel momento.

Setup

Size population:

- se fissa bisogna scegliere il numero;
- se variabile, è bene dare una dimensione massima;

Size Mating Pool: numero di individui selezionati e che partecipano alla riproduzione.

Probabilità Crossover: con quale probabilità avviene il crossover tra due genitori;

Probabilità Mutazione: con quale probabilità avviene una mutazione;

Inizializzazione:

- la prima generazione si crea in modo random;

- la prima generazione si crea basandosi su euristiche (se il GA è meno cieco allora si può evitare di iniziare con individui pessimi);

Tipologie di Selezione

Quando si passa la selezione si è ammessi alla riproduzione accedendo alla mating pool

-**Roulette Wheel**: si assegna una porzione di una ruota a ciascun individuo a seconda della sua fitness relativa (gli individui con fitness alta hanno una maggiore possibilità di essere scelti). La ruota solitamente la si gira il numero di volte pari al numero di individui, un individuo può vincere più volte. Si rischia che un individuo troppo forte viene sempre scelto, senza avere grande diversità, provocando convergenza prematura (l'algoritmo si blocca troppo presto). E non va bene con funzioni di fitness negative

-**Random**, ma non si usa mai.

-**Rank (Ordered Selection)**: si ordinano gli individui rispetto alla fitness e ogni individuo ottiene un rango e sulla base del rango si riceve una probabilità di essere scelti. Si favoriscono gli individui più forti, ha meno possibilità di convergenza prematura.

-**Truncation**: si ordinano gli individui e si prendono i primi M con M < n. Vengono selezionati i primi M e in questo caso un individuo può essere scelto al più una volta. Toglie componenti di casualità ed è un problema. Qual è il valore di M ? Altro problema.

- **K-Way (Tournament)**: vengono selezionati K individui in modo casuale ,con K < n, il migliore tra questi K passa la selezione definitivamente. Si ripete finché non si arriva ad M tornei (e quindi M vincitori). Si possono avere selezioni ripetute. Si fanno M tornei ognuno con K individui. Il problema è scegliere M e K. Si può implementare una pressione di selezione: invece di far vincere sempre il migliore di un torneo, ogni individuo ha una probabilità di vittoria proporzionata alla sua fitness (ogni torneo diventa una mini Roulette Wheel). Si possono anche impedire le ripetizioni: una volta partecipato ad un torneo che abbia vinto oppure che abbia perso non posso partecipare ad altri tornei.

Tipologie di Crossover

-**Single Point**: si sceglie un punto di taglio e si prendono i geni prima del punto di taglio di un genitore e poi si prendono i geni successivi al punto di taglio dell'altro genitore.

-**Two Point**: scegliere due punti di taglio (sempre in modo casuale).

-**K Point**: generalizzazione di Single/Two Point.

-**Uniform**: ciascun gene i-esimo viene scelto casualmente tra i due geni i-esimi dei genitori (casualità massima).

-**Arithmetic**: ciascun gene i-esimo è il valor medio dei geni i-esimi dei genitori. I due figli saranno gemelli però infatti a tal punto si definisce un peso per i due genitori e si fa la media pesata anziché quella aritmetica. Il primo figlio usa il peso alpha per il primo genitore ed il peso 1-alpha per il secondo genitore. Il secondo figlio il viceversa. (Un figlio somiglia più a un genitore piuttosto che un altro).

Inoltre è possibile avere anche più di due genitori infatti studi empirici hanno dimostrato che possibilità con n > 2 non bisogna sottovalutarle.

Tipologie di Mutazione

Si può scegliere in modo casuale quali individui mutare e quali individui no.

-**Bit Flip**: si cambia il valore di un bit da 0 diventa 1 oppure da 1 diventa 0;

-**Random Resetting**: cambio casuale di un gene ad un altro valore ammissibile;

-**Swap**: scambio di due geni, scelti casualmente;

-**Scramble**: permutazione casuale di un sottoinsieme di geni;

-**Inversion**: si scegli un sottoinsieme e lo si "ribalta";

Stopping Condition/Budget Di Ricerca

Con quale criterio interrompere l'algoritmo.

- **Tempo di Esecuzione**: l'evoluzione termina se si è superato un tempo di esecuzione T. Allo stop o si decide di restituire l'ultima generazione ottenuto o la migliore ultima;
- **Costo**: se è presente una funzione di costo allora al raggiungimento di tale costo l'evoluzione termina;
- **Numero Generazioni**: si itera per massimo X generazioni;
- **Assenza Miglioramenti**: se per Y generazioni consecutive non ci sono stati miglioramenti (significativi) allora l'evoluzione termina;
- **Ibride**: combinare più stopping condition;
- **Problem-Specific**: individuare stopping condition relative ad un determinato problema;

Gli algoritmi genetici possono avere tanti campi di applicazione: ML, DL (NeuroEvoluzione), Algoritmi come TSP, Bio-informatica (analisi del DNA), Testing (SBST, SBSE).

Pro VS. Contro

+Fitness molto flessibile.

+Ottimizza sia nel continuo che nel discreto.

+Valido per problemi multi-obiettivo.

+Esplora rapidamente lo spazio di ricerca ed ottiene in poco tempo soluzioni sub-ottime.

+Ben parallelizzabile.

+Buona explainability, facile da spiegare.

-Fitness molto flessibili non si presta bene a problemi di decisione [si/no] (Workaround: tecnica di conversione di problemi di decisione in problemi di ottimizzazione).

-La fitness viene valutata molte volte e le prestazioni ne possono risentire se la codifica è complessa (si possono utilizzare delle approssimazioni per non avere questo problema).

-Convergenza Prematura (per ovviare si può introdurre la diversità. È bene che GA faccia exploitation [favorire individui] e exploration [navigare nello spazio di ricerca]).

-Non sempre raggiungono l'ottimo.

-Bisogna considerare numerosi parametri.

Tecniche di Improvement

-**Elitism**: tecnica che permette di salvare sicuramente il migliore (o i migliori k) individuo/i e copiarli nella generazione successiva, così facendo siamo sicuri che la nuova generazione non sia peggiore della precedente.

-**Conoscenza problem-specific**: non sono indispensabili però ci possono aiutare notevolmente.

Problemi Multi-Obiettivo

Problemi multi-obiettivo: un individuo non avrà più una funzione fitness, bensì un vettore di fitness di dimensione n. Tra due individui chi scelgo ? Non devo confrontare due valori, ma due vettori.

A tal proposito si introduce il concetto di Fronte di Pareto. Si indica con **fronte di Pareto** un insieme di individui di una popolazione non migliori tra loro, ma migliori di tutti gli altri individui al di fuori dal fronte. Gli elementi presenti nell'insiemi si possono chiamare **ottimi secondo Pareto**. Volendo si può preferire un individuo piuttosto che un altro quando ad esempio le dimensioni del fronte di Pareto iniziano ad aumentare notevolmente.

Ciò avviene grazie al **Preference Sorting**, ovvero una tecnica che permette di fornire un ordinamento totale tra gli individui nel fronte di Pareto attraverso l'uso di una Funzione di Preferenza che ad esempio deriva dalla conoscenza del problema (la funzione di preferenza è diversa dalle funzioni di fitness). Il numero di criteri di preferenza dipende tutto da noi. Un'altra strategia è la strategia dell'archivio **Archive strategy**: strategia che mantiene una popolazione aggiuntiva che non evolve, contenente gli individui che riescono a soddisfare degli obiettivi mai soddisfatti nelle precedenti iterazioni. Al termine del budget (quando termina il tutto) invece di restituire soltanto l'ultima generazione ritorno l'archivio perché so già che contiene un insieme di individui molto forti quanto unici. Questa strategia per applicarla è necessario avere conoscenze approfondite sulle fitness (sapere quando ho raggiunto l'ottimo globale).

Lo scopo nei problemi multi-obiettivo è quello di evitare di avere troppe fitness ed è consigliato usare funzioni indipendenti tra loro, ma dove ciò non è possibile e vi sono dipendenze si può applicare la **Dynamic Target Selection**: tecnica che, ad ogni iterazione, ci permette di restringere l'insieme delle funzioni di fitness da valutare a seconda del risultato dell'iterazione precedente (Es: ho 100 problemi e gli ultimi 30 non possono essere risolti senza i primi 70 allora risolvo prima i 70 e poi mi concentro sui restanti 30 facendo attenzione a non perdere la soluzione ai primi 70).

Varianti

-**Steady-State GA**: non genera nuove generazioni, ma mantiene sempre la medesima popolazione. Non si effettua la selezione e una sola coppia di genitori si accoppia e produce 2 figli, vengono mutati e poi rimpiazzano i genitori (evoluzione lenta e meno caotica). Il costo computazionale ad ogni iterazione è molto più basso, spesso si adotta in problemi multi-obiettivo.

-**Memetic Algorithm**: nell'algoritmo genetico posso poi lanciare un algoritmo di ricerca locale, riducendo convergenza prematura.

-**Adaptive GA**: trovare la combinazione di parametri è difficile, a tal proposito tale scelta di parametri può avvenire runtime (fa tutto da solo).

-**Interactive GA**: (esempio di human in the loop): interviene manualmente l'uomo per la scelta dei parametri ad ogni iterazione.

Agenti per Ricerca Online

Fino a questo momento abbiamo studiato agenti che utilizzano algoritmi di ricerca offline, ovvero agenti che cercano una soluzione prima di entrare nel mondo reale per poi eseguirla. Nel caso della ricerca online, un agente opera alternando computazione e azione: prima esegue un'azione poi osserva l'ambiente per determinare l'azione seguente. La ricerca online è l'ideale per domini dinamici o semi-dinamici. Inoltre la ricerca online è anche adatta quando l'ambiente non è deterministico e quindi l'agente si può concentrare su attività di calcolo sulle contingenze che si verificano realmente piuttosto che quelle probabili, si adattano in base a quello che succede effettivamente nell'ambiente. Teoricamente sono ottimi, ma praticamente non sempre sono un bene questi agenti perché più un agente pianifica in anticipo meno spesso si troverà in situazioni inaspettate e che potrebbero provocare azioni non efficaci. Però la ricerca online diventa una necessità con ambienti ignoti, in cui gli stati e gli effetti non sono noti in anticipo all'agente e quindi l'agente affronta un problema di esplorazione e dovrà considerare le sue azioni come veri e propri esperimenti. L'agente affronta un problema di esplorazione. Esempi per questi algoritmi sono i robot che devono esplorare un nuovo edificio, labirinti, analisi dei dati finanziari, bot. Definiamo gli algoritmi di ricerca online come un algoritmo in grado di analizzare un flusso di dati che arriva in input in tempo reale ed elaborare un processo al fine di determinare un'azione da restituire in output.

Sono detti "online" perché riescono ad elaborare "in linea" il flusso di dati. Negli algoritmi di ricerca online un'agente conosce solamente tre cose:

1. AZIONI(s), restituisce una lista delle azioni permesse nello stato s;
2. c(s, a, s') ovvero una funzione di costo di passo il quale non è noto a priori, ma sarà noto solo dopo aver effettivamente effettuato il passo;
3. TEST-OBIETTIVO(s) ovvero la funzione che verifica che uno stato s sia un obiettivo;

Quindi l'agente non è in grado di determinare RISULTATO(s, a) se non trovandosi effettivamente in s ed eseguendo a.

Solitamente lo scopo dell'agente è raggiungere uno stato obiettivo minimizzando il costo oppure esplorare l'ambiente. Il costo è tipicamente dato dal costo del cammino effettivamente percorso dall'agente. Una soluzione si valuta buona se il costo ottenuto è uguale al costo del cammino che l'agente potrebbe seguire se conoscesse in anticipo lo spazio di ricerca. Il risultato di questo confronto prende il nome di rapporto di competitività ed il nostro scopo è quello di tenerlo il più basso possibile. Inoltre in alcuni casi può essere infinito ad esempio quando le azioni sono irreversibili. In tal caso la ricerca online potrebbe arrivare ad un vicolo cieco da cui non è possibile raggiungere uno stato obiettivo. Il problema dei vicoli ciechi è un problema serio per gli algoritmi in quanto si parla di occasioni di azioni irreversibili che portano al fallimento dell'agente. Il problema del vicolo cieco può essere usato a proprio vantaggio nel contesto dell'adversary argument: possiamo immaginare un "avversario" che costruisce lo spazio degli stati mentre l'agente lo esplora, posizionando obiettivi e ostacoli a piacimento. Questa tecnica può essere utilizzata per misurare la bontà di questi algoritmi in quanto tale tecnica mira a "mettere in difficoltà" quanto più possibile un algoritmo cercando di fargli eseguire quanti più passi possibili per raggiungere un obiettivo. Nel nostro caso lo spazio degli stati è esplorabile in modo sicuro, ovvero si può sempre arrivare ad uno stato obiettivo da ogni stato raggiungibile, in poche parole non vi sono vicoli ciechi. Quindi logicamente uno spazio degli stati con azioni reversibili sono rappresentati come grafi NON orientati, prima invece erano grafi orientati, perché non si poteva tornare indietro. Per quanto riguarda il funzionamento un agente online riceve una percezione che gli comunica lo stato raggiunto e può arricchire così la mappa. La versione corrente della mappa verrà utilizzata per decidere l'azione successiva e quest'alternanza è la sostanziale differenza dagli algoritmi di ricerca offline. Le caratteristiche degli agenti online portano l'agente ad eseguire delle azioni tipiche degli algoritmi locali e il metodo di esplorazione è di ordine locale.

Algoritmo di Ricerca in Profondità

Un algoritmo facilmente adattabile in questo contesto è la DFS, il nodo successivo è sempre figlio di quello appena espanso ad eccezione di quando fa backtracking. Inoltre ogni volta che c'è un'azione non ancora esplorata, la ricerca in profondità la proverà. Quando l'agente ha provato tutte le azioni in uno stato: nel caso della ricerca offline semplicemente cancellerà dalla coda quello stato; mentre in quella online, invece, l'agente dovrà tornare sui propri passi,

dal punto di vista tecnico la tabella dovrà avere una tabella che elenca, per ogni stato, gli stati predecessori a cui l'agente non è ancora ritornato con il backtracking.

Hill-Climbing

Tale algoritmo è già un algoritmo online. Ma come abbiamo visto precedentemente può rischiare di bloccarsi in un minimo locale e quindi non sarebbe in grado di restituire una soluzione nel caso in cui un nodo successivo abbia costo maggiore del corrente. Una delle varianti dell'Hill-Climbing era quella con riavvio casuale, ma in questo caso non si può attuare poiché l'agente non può trasferirsi in un nuovo stato dello spazio degli stati può solo tornare sui suoi passi.

Vi è una nuova variante ovvero la Random-Walk e in questo caso l'agente sceglie a caso una delle azioni possibili dello stato corrente. In aggiunta si possono anche prediligere le mosse che non sono state ancora provate. Con questo algoritmo prima o poi si troverà un obiettivo e completerà la sua esplorazione a patto che lo spazio degli stati sia finito. Ma tale processo potrebbe essere estremamente lento, abbassando le prestazioni dell'algoritmo.

Una variante che si può utilizzare è l'Hill-Climbing con memoria all'interno della quale si terrà in memoria una funzione che calcola la miglior stima corrente del costo che serve per raggiungere un obiettivo da ogni stato visitato, secondo un'euristica. Si tiene in memoria una tabella dei valori dell'euristica stimati, quando si esplora un figlio si aggiorna il valore dell'euristica del padre sulla base delle nuove informazioni che scopre, continuando in questo modo l'agente si muove nello spazio degli stati appiattendo il minimo locale.

Ricerca con Avversari

In questo capitolo si parla di problemi multi-agente quindi avremo due agenti. In tale contesto ogni agente deve necessariamente considerare quello che fa l'altro così da poter quantificare il proprio benessere. Iniziamo a discutere con ambienti competitivi, in cui gli obiettivi degli agenti sono in conflitto. Questo dà origine ai problemi di ricerca con avversari i quali vengono indicati come giochi.

La **teoria dei giochi** è una branca dell'economia che considera ogni ambiente multi-agente come un gioco, indipendentemente dal fatto che l'interazione sia cooperativa o competitiva, a patto che l'influenza di ogni agente sugli altri sia significativa.

Gli ambienti con quantità molto grandi di agenti sono spesso chiamati economie piuttosto che giochi.

Prima classificazione dei giochi:

- Giochi con **informazione "perfetta"**: giochi in cui gli stati del gioco sono completamente espliciti agli agenti;

- Giochi con **informazione "imperfetta"**: giochi in cui gli stati del gioco sono solo parzialmente esplicitati;

Seconda classificazione dei giochi:

- Giochi **deterministici**: giochi in cui gli stati del gioco sono determinati unicamente dalle azioni degli agenti;

- Giochi **stocastici**: giochi in cui gli stati del gioco sono determinati anche da fattori esterni (basti pensare ai giochi in cui c'è un dado);

I giochi più comuni sono quelli che vengono definiti giochi a somma zero e giochi con informazione perfetta; Con **somma zero** significa che il guadagno finale ha esattamente somma zero, i valori di utilità alla fine della partita avranno valore sempre uguale, ma di segno opposto.

Più formalmente si può definire un gioco come un problema di ricerca costituito da:

- **s0**: stato iniziale;
- **GIOCATORE(s)**: definisce il giocatore a cui tocca fare una mossa nello stato s;
- **AZIONI(s)**: restituisce l'insieme delle mosse lecite in uno stato s;
- **RISULTATO(s, a)**: il modello di transizione, che definisce il risultato di una mossa;
- **TEST-TERMINAZIONE(s)**: un test di terminazione che restituisce true se la partita è finita e false altrimenti. Definiamo "stati terminali" gli stati che fanno terminare una partita;
- **UTILITÀ(s, p)**: una funzione utilità chiamata anche funzione obiettivo o funzione di playoff che definisce il valore numero finale per un gioco che termina nello stato terminale s per un giocatore p. In altri termini, la funzione da il punteggio finale da assegnare ai giocatori.

Lo stato iniziale, azioni e funzione risultato definiscono l'**albero di gioco**, un albero i cui nodi sono stati del gioco e gli archi le mosse.

In un normale problema di ricerca la soluzione ottima è costituita da una sequenza di mosse che portano ad uno stato obiettivo. In una ricerca con avversari ogni giocatore dovrà elaborare una strategia che lo porti alla vittoria. Si definisce, a tal proposito, una strategia ottima la quale porta ad un risultato che è almeno pari a quello di qualsiasi altra strategia, assumendo che si stia giocando contro un giocatore infallibile.

Dilemma del prigioniero fu inventato da John Von Neumann, ma divenne popolare con John Nash il quale vinse il premio Nobel per l'Economia: due criminali vengono accusati di aver commesso di porto abusivo di armi e, per questo, gli investigatori li arrestano e li chiudono in due celle separate impedendo loro di comunicare. Ad ognuno di loro vengono date due scelte: collaborare, non collaborare. Gli investigatori spiegano ai due criminali che se:

-se solo uno dei due collabora accusando l'altro, chi ha collaborato evita la pena, l'altro però viene condannato a 7 anni di carcere;

-se entrambi accusano l'altro, vengono entrambi condannati a 6 anni;

-se nessuno dei due collabora, entrambi vengono condannati a 1 anno, perché comunque già colpevoli di porto abusivo di armi;

La migliore strategia in questo gioco non cooperativo è quella che entrambi collaborano. La scelta di collaborare è, quindi, la migliore possibile per uno dei due, in assenza di informazioni su ciò che farà l'altro. Questo avviene perché lo scopo di entrambi è quello di minimizzare la propria condanna e quindi collaborando rischiano da 0 a 6 anni, mentre non collaborando rischiano da 1 a 7 anni. La teoria dei giochi afferma che vi è un solo equilibrio (collabora, collabora). Tale equilibrio prende il nome di **equilibrio di Nash**, ovvero quando ognuno fa ciò che è meglio per se stesso a questo punto la società non giungerà alla soluzione migliore.

La scelta migliore per la "**società**", i due prigionieri, sarebbe di non parlare, ma questa scelta non è un equilibrio di Nash perché basta semplicemente che uno parla per migliorare la sua situazione. L'unico equilibrio di Nash in questo caso è quello in cui entrambi prendono 6 anni a testa. Se il compagno non parla allora è libero, se il compagno parla prende 6 anni anziché 7. Secondo Adam Smith, la teoria economica "classica" diceva che ogni azione individuale accresce la ricchezza complessiva del gruppo, ma la teoria dei giochi dimostra che invece non è così. Se ognuno si comporta in modo tale da fare ciò che è meglio per se, il risultato al quale si giunge è un equilibrio di Nash che non rappresenta la soluzione migliore per la società e può rappresentare un'allocazione inefficiente delle risorse.

Ottimo Paretiano: si parla di ottimo paretiano quando si è raggiunta una situazione in cui non è possibile migliorare la condizione di una persona senza peggiorare la condizione di un'altra. L'ottimo di Pareto non porta a una situazione migliore per la società.

Le strategie egoistiche non portano ad una soluzione ottimale per la società né se non ci si cura di ciò che succede agli altri e neanche se si è disposti a collaborare a patto di non avere svantaggi. Per raggiungere una situazione migliore si deve imporre a qualcuno di rinunciare a qualche cosa, essere egoisti non porta mai ad una soluzione ottimale per tutti.

Dinamiche dominanti: parliamo di dinamiche dominanti quando un giocatore fa il meglio che può indipendentemente da ciò che farà l'altro.

Il dilemma dei beni pubblici è una generalizzazione del dilemma del prigioniero ed include N giocatori. Se ogni giocatore rispetta, nel consumo individuale, la capacità auto-rigenerazione del bene pubblico, esso perdurerà nel tempo a beneficio di tutti. Tuttavia il singolo giocatore è spinto a comportarsi in maniera opportunistica.

MiniMax

Il problema è quello di come poter trovare una strategia ottima. Dato un albero di gioco la strategia ottima può essere determinata analizzando il valore minimax di ogni nodo, che scriveremo MINIMAX(n).

Valore Minimax: il valore minimax di un nodo corrisponde all'utilità di trovarsi nello stato corrispondente, assumendo che entrambi gli agenti giochino in modo ottimo da lì alla fine della partita.

In Minimax, Max tende sempre a massimizzare mentre Min tende sempre a minimizzare.

MINIMAX(n) = UTILITÀ(s, MAX) se TEST-TERMINALE(s)

MINIMAX(n) = max AZIONI(s) VALORE-MINIMAX-RISULTATO(s, a) se GIOCATORE(s) = MAX;

MINIMAX(n) = min AZIONI(s) VALORE-MINIMAX-RISULTATO(s, a) se

GIOCATORE(s) = MIN

La strategia ottima massimizza il risultato di MAX nel caso pessimo ovvero nel caso in cui MIN ha fatto la sua scelta migliore ovvero quella che minimizza. Se MIN non minimizza allora MAX farà sicuramente meglio, massimizzando ancora di più i suoi valori minimax.

Performance dell'algoritmo minimax:

-**Completezza**: l'algoritmo è completo se l'albero è finito;

-**Ottimalità**: è ottimale nel caso in cui l'avversario è ottimo, nel caso in cui l'avversario non gioca in modo ottimo continua anche lì ad essere ottimo;

-**Complessità temporale**: esponenziale O(b elevato a m);

-**Complessità spaziale**: O(bm) perché memorizzo solo un cammino alla volta e non bisogna memorizzare l'intero albero di gioco;

L'algoritmo minimax si può applicare anche nei giochi multi-player, in tal caso si alternano gli N giocatori. È costituito da un vettore di N elementi ed ogni giocatore tende a massimizzare il valore della posizione corrispondente nel vettore, i-esimo giocatore massimizza la i-esima posizione del vettore.

Si possono creare anche altre complicazioni dell'algoritmo nel caso in cui ad esempio vi sono delle alleanze fra i giocatori e quindi bisogna considerare anche queste situazioni, le alleanze possono essere anche dinamiche e quindi cambiano nel corso del gioco.

Potatura alpha-beta

È possibile calcolare la decisione minimax corretta senza analizzare tutti i nodi dell'albero di gioco. In questo caso si parla di potatura, ovvero la tecnica che consente di evitare di prendere in considerazione grandi porzioni dell'albero, la tecnica specifica che analizzeremo si chiama potatura alpha-beta che applicata ad un albero minimax restituisce lo stesso risultato della tecnica minimax standard, ma "pota" i rami che non possono influenzare la decisione finale.

La potatura alpha-beta ci può portare alla potatura di interi sotto-alberi rendendo la potatura più efficace.

Consideriamo un nodo n da qualche parte dell'albero tale che il giocatore abbia la possibilità di spostarsi in quel nodo. Se esiste una scelta migliore m a livello del nodo padre o di qualunque altro nodo precedente, allora n non sarà mai raggiunto in tutta la partita. Di

conseguenza possiamo potare n non appena abbiamo raccolto abbastanza informazioni per poter giungere a questa conclusione.

Alpha: il valore della scelta migliore per MAX che abbiamo trovato sin qui in un qualsiasi punto di scelta lungo il cammino;

Beta: il valore della scelta migliore per MIN che abbiamo trovato sin qui in un qualsiasi punto di scelta lungo il cammino;

L'efficacia della potatura alpha-beta dipende in modo particolare dall'ordinamento delle mosse, infatti adesso vediamo delle tecniche di ordinamento dinamico delle mosse.

Ricerca ad Approfondimento Iterativo

Potrebbe essere una buona idea quella di applicare una valutazione best-first ed analizzare i nodi più promettenti. In tal caso si può dimezzare la complessità temporale diventando O(b elevato a m/2). In altre parole una modifica di questo tipo ci consente di risolvere un albero profondo il doppio rispetto a quello risolto da minimax.

In mancanza di una strategia a priori si possono analizzare schemi dinamici. Un modo per ottenere informazioni dalla mossa corrente è quello di utilizzare una ricerca ad approfondimento iterativo. Con questa strategia si cerca uno strato in profondità e si registra il miglior cammino di mosse. Poi si cerca uno strato ancora in profondità, ma si utilizza il cammino registrato allo scopo di fornire informazioni per l'ordinamento delle mosse (al livello k non esploro per il livello k+1 il primo nodo, ma quello che al livello k ha ad esempio un valore molto promettente).

Le mosse migliori sono spesso chiamate **mosse killer** e si parla di euristica della mossa killer quando queste vengono provate per prime.

Con la ricerca ad approfondimento iterativo possiamo esplorare in ampiezza e decidere come variare l'ordinamento in base ai valori minimax.

Trasposizioni

Un cammino è ridondante quando lo stesso stato può essere contenuto più volte nelle spazio.

Un gioco per natura ad esempio potrebbe avere dei cammini ridondanti. Quando due sequenze di mosse si concludono con la stessa configurazione praticamente quelle due sequenze diverse danno la stessa configurazione alla fine [a1,b1,a2,b2] e [a1,b2,a2,b1]. A questo punto vale la pena memorizzare la valutazione di una configurazione in una tabella cash la prima volta che questa viene incontrata. La tabella delle trasposizioni è molto simile alla lista esplorati. Si può decidere di ordinare le mosse successive sulla base della conoscenza già acquisita esplorando una trasposizione. Se l'albero di gioco dovesse avere milioni di stati, nella tabella si dovranno memorizzare solo alcune delle trasposizioni, altrimenti è troppo grande la tabella. Le trasposizioni da mantenere possono essere scelte secondo diverse strategie. Una strategia possibile è quella di definire una funzione euristica.

Decisioni Imperfette in Tempo Reale

L'algoritmo minimax genera l'intero spazio di ricerca, mentre quello alpha-beta ci permette di potarne buona parte. In ogni caso comunque bisognerà condurre la ricerca fino agli stati terminali. Questa profondità risulta normalmente ingestibile, perché le mosse devono essere calcolate in tempo ragionevole, ad esempio nei giochi a tempo.

Claude Shannon propose di "tagliare" la ricerca prima di raggiungere le foglie applicando una funzione di valutazione euristica degli stati (quando è buono lo stato in cui mi trovo senza arrivare alla fine). I nodi non terminali così diventano delle foglie. Il minimax a questo punto subisce due modifiche:

1. La funzione di utilità viene sostituita da EVAL, che fornisce una stima dell'utilità della posizione raggiunta;
2. Il test di terminazione viene sostituito con un test di taglio (cutoff test) che decide quando applicare EVAL;

La nuova formulazione porta il problema in questo modo:

H-MINIMAX(n)= EVAL(s) se TEST-TAGLIO(s)

H-MINIMAX(n)= max AZIONI(s) H-MINIMAX(RISULTATO(s, a)) se GIOCATORE(s)=MAX;

H-MINIMAX(n)= min AZIONI(s) H-MINIMAX(RISULTATO(s, a)) se GIOCATORE(s)=MIN;

La difficoltà principale sarà la definizione di una funzione di valutazione.

Funzione di Valutazione

La funzione di valutazione deve ordinare gli stati terminale nello stesso modo della funzione di utilità. Gli stati che sono vittorie devono avere una valutazione migliore dei pareggi che a loro volta devono essere migliori delle sconfitte.

La funzione di valutazione deve essere veloce da calcolare.

Per gli stati non terminale la funzione di valutazione deve essere sensata e dire quale è la probabilità reale di vincere una partita.

La maggior parte delle funzioni di valutazione si basano sulle caratteristiche di uno stato. Ogni caratteristica definisce una classe di equivalenza, ovvero l'insieme di stati aventi lo stesso valore per tutte le caratteristiche, se considero un elemento in una classe di equivalenza piuttosto che un altro alla fine non mi cambia niente.

In molti casi la funzione calcola valori separati per ogni caratteristica e poi li combina insieme per formare il valore finale. Così facendo EVAL(s) la si può vedere come una funzione lineare pesata EVAL(s)=w1f1(s)+…+wnfn(s) dove wi è il peso e fi è una caratteristica. Ciò è possibile se i contributi individuali delle caratteristiche sono indipendenti dagli altri anche se oggigiorno in alcuni casi si usano combinazioni non lineari.

Test di Taglio

Nella definizione del test di taglio, dobbiamo fare attenzione ai cosiddetti **stati non quiescenti** ovvero stati che portano ad una variazione brusca/repentina del valore delle mosse.

Per tale motivo la funzione di valutazione dovrebbe essere applicata solo a posizioni quiescenti, ovvero quelle per cui sia improbabile il verificarsi di grandi variazioni di valore nelle mosse immediatamente successive. A volte può essere inclusa nell'algoritmo una ricerca di quiescenza, in cui gli stati non quiescenti vengono espansi fino a raggiungere posizioni quiescenti.

Varianti Potatura alpha-beta

Potatura in avanti: sulla base di una valutazione si esplorano solo alcuni stati che risultano essere particolarmente promettenti escludendo gli altri. Un approccio in questo senso è la Beam Search dove considero solo le prime k mosse migliori, definite sulla base della funzione di valutazione. Un altro approccio è quello dei **tagli probabilistici** i quali sono basati sull'esperienza, vengono utilizzate statistiche ricavate dalle precedenti esperienze per ridurre la possibilità che la mossa migliore venga potata.

Database di mosse di apertura e chiusura: l'idea di base è quella che all'inizio di ogni partita ci sono solitamente poche mosse sensate e ben studiate per cui è inutile esplorarle tutte. Stesso ragionamento anche per le fasi finali. Nel gioco degli scacchi viene utilizzato un database di questo tipo.

Giochi Stocastici

Nella vita reale si verificano molti eventi esterni che ci mettono in situazioni impreviste, come ad esempio quando vi è anche la presenza di elementi casuali. A tal proposito si parla di giochi stocastici.

È possibile adattare gli algoritmi visti fino ad ora introducendo i cosiddetti "nodi possibilità" accanto a quelli di scelti. Così facendo nel calcolare i valori di MIN e MAX dovremo tenere conto delle probabilità dell'esperimento casuale. In altri termini non si calcola più i valori di minimo e di massimo, ma i valori di massimo e minimo attesi. I nodi saranno "pesati" nel senso che assumeranno come valore la somma del valore su tutti i risultati pesati in base alla probabilità di ciascuna azione. Per gli stati terminali, il valore verrà calcolato nello stesso modo visto in precedenza e quindi non dovremo tenere conto della probabilità.

Giochi Parzialmente Osservabili

Questa tipologia di giochi nasce dal fatto che vi è l'impossibilità di accedere alle scelte fatte dall'avversario, basti pensare ai giochi di carte quando le carte dell'avversario non sono note.

Si potrebbe pensare questi giochi come un caso particolare dei giochi stocastici (immaginiamo di avere un dado con tantissime facce ed esprimere i nodi possibilità come i nodi possibilità per ogni possibile azione dell'avversario). Questa analogia non è praticamente fattibile a

livello di efficienza perché si avrebbero tantissimi nodi da considerare. Una soluzione potrebbe essere quella di considerare tutte le possibili distribuzioni di carte nascoste e risolviamo una per una come se fosse un gioco completamente osservabile. A questo punto basta scegliere la mossa che ha il miglior risultato calcolato in media su tutte distribuzione.

Purtroppo in molti casi il numero di distribuzioni è troppo grande. Un'eventuale alternativa sarebbe l'approssimazione Monte Carlo: invece di sommare tutte le distribuzioni, prendiamo un campione casuale di N distribuzioni in cui la probabilità di s di apparire nel campione è proporzionale alla probabilità P(s) (immaginati una Roulette Wheel dove un campione può essere preso solo una volta però, SBAGLIATO SE SI DICE AL PROF).

Problemi di Soddisfacimento dei Vincoli

Nei problemi di ricerca standard abbiamo sempre considerato uno stato come una rappresentazione atomica nella quale non poter entrare, considerando gli stati come rappresentazioni atomiche. All'interno dei problemi di **soddisfacimento dei vincoli** si cerca di rilassare il concetto di rappresentazione atomica di uno stato e allo stesso tempo modificare il test obiettivo, il quale in precedenza considerava solamente l' "esterno" dello stato mentre in questo caso si va ad analizzare l'interno e quindi la sua struttura. Lo stato si rappresenta con una rappresentazione fattorizzata: una serie di variabili ognuna delle quali ha un valore.

Un problema è risolto solo quando ogni variabile ha un valore che soddisfa tutti i vincoli su di essa, a tal proposito si parla di problemi di soddisfacimento di vincoli (Constraint Satisfaction Problem).

Un'altra **differenza** dagli algoritmi standard è che tengono in considerazione la struttura interna degli stati ed euristiche di uso generale anziché specifiche del problema per la soluzione di problemi complessi (in questo caso si può sfruttare la rappresentazione interna dello stato evitando di utilizzare euristiche specifiche per il problema che stiamo analizzando). L'idea principale di questi algoritmi è quella di eliminare ampie porzioni dello spazio di ricerca tutte allo stesso tempo andando ad individuare delle combinazioni variabili-valori che violano i vincoli.

Un problema CSP è costituito da 3 componenti:

1. X, insieme di variabili {X1, …, Xn};
2. D, insieme di domini {D1, …, Dn}, uno per ogni variabile Xi;
3. Insieme di vincoli che specificano combinazioni di valori ammesse;

Ogni dominio Di è costituito da un insieme di valori ammessi per la variabile Xi,

{v1, …, vk}.

Ogni vincolo Ci è costituito da una coppia <ambito, rel> dove ambito è una tuple di variabili che partecipano nel vincolo e rel è una relazione che definisce i valori che tali variabili possono assumere. Una relazione può essere espressa come elenco esplicito di tutte le tuple di valori che soddisfano il vincolo o come una relazione astratta che supporta due operazioni:

1. Controllare se la tuple è membro della relazione;
2. Enumerare i membri stessi;

La risoluzione di un problema di soddisfacimento di vincoli consiste nella definizione di uno spazio degli stati e il concetto di soluzione.

Ogni stato in un problema CSP è definito dall'assegnamento di valori ad alcune o tutte le variabili ovvero {Xi=vi, Xj=vj, …, Xn=vn}.

L'assegnamenti di valori in uno stato in un problema di CSP gode di due proprietà:

1. Consistenza: un assegnamento che non viola nessun vincolo è chiamato assegnamento consistente o legale;
2. Completezza: un assegnamento è completo se a tutte le variabili è assegnato un valore, ovvero se tutte le variabili sono valorizzate;

Una soluzione di un problema CSP è un assegnamento completo e consistente. Il test obiettivo si deve definire tenendo conto di un'assegnazione completa e consistente delle variabili.

Un problema CSP fornisce una rappresentazione naturale di molti problemi reali. Se si dispone di un risolutore di problemi CSP è più facile risolvere altri problemi utilizzando sempre lo stesso senza doverlo adattare in modo particolare.

Questi risolutori sono molto molto **generici** ed è un grande vantaggio.

Il secondo vantaggio è quello **dell'efficienza**. Infatti i risolutori CSP riescono ad eliminare rapidamente porzioni dello spazio che non rispettano i vincoli.

È possibile visualizzare un CSP come un grafo di vincoli i cui nodi sono le variabili del problema e un arco connette ogni coppia di variabili che partecipano ad un vincolo.

Un risolutore di un problema CSP sarà sicuramente più veloce rispetto agli algoritmi di ricerca standard perché ad esempio nel caso della colorazione dell'Australia se ad uno stato ho assegnato un colore posso automaticamente escludere tutti gli stati adiacenti che prevedono il colore blu perché non soddisferebbe il vincolo.

Nella forma più semplice di CSP le variabili hanno domini discreti e finiti, ma vi possono essere delle varianti come riportate di seguito. Un dominio discreto può essere infinito basti pensare agli interi o le stringhe. In questo caso bisogna affidarsi ad un linguaggio di specifica dei vincoli in quanto non si possono descrivere i vincoli enumerandoli tutti quanti. I CSP con domini continui sono comuni nel mondo reale e si possono risolvere con la programmazione lineare. Altre varianti possono riguardare i vincoli anziché il dominio delle variabili. Si parla di vincolo unario quando il vincolo interessa una singola variabile; i vincoli binari mettono in

relazione due variabili ed infine un vincolo globale se interessa un numero arbitrario di variabili. Un'altra categoria di vincoli è quella di preferenza. Anziché avere vincoli assoluti, la cui violazione impedisce di trovare una soluzione, abbiamo dei vincoli che indicano delle preferenze ed in tal caso non si parla esplicitamente di CSP, ma di problemi di ottimizzazione di vincoli.

Formulazione del problema di ricerca standard:

- Stato iniziale: un insieme vuoto {} che corrisponde ad un assegnamento nullo dei valori;
- Azioni: la funzione assegna un valore ad una variabile non assegnata che è compatibile con l'assegnamento corrente. Se non esiste un assegnamento legale allora la funzione non restituisce nessun valore, portando al fallimento;
- Test obiettivo: controlla che l'assegnamento corrente è completo;

Questa formulazione di problemi è valida per ogni tipologia di CSP a prescindere dalla variante e di tutto il resto appresso.

Ogni soluzione apparirà a profondità n, se abbiamo n variabili. In altri termini, ad ogni passo corrisponderà un avanzamento di un livello dell'albero di ricerca. Possiamo utilizzare a tal punto un algoritmo di ricerca in profondità per arrivare alla soluzione.

A livello di efficienza però si presenta un problema perché se abbiamo n variabili con un dominio di dimensione d, il fattore di ramificazione al primo livello è n*d poiché uno qualsiasi dei d valori può essere assegnato a ognuna delle n variabili. Al livello successivo, il fattore di ramificazione sarà (n-1)*d e così via. In generale avremo quindi n!*d foglie. In questo caso un DFS non sarà efficace e a tal proposito vedremo come implementare soluzioni più efficienti.

Ricerca con Backtracking

Proprietà della commutatività: un problema è commutativo se l'ordine di applicazione di un qualsiasi insieme di azioni non ha effetto sul risultato finale.

I CSP sono commutativi perché assegnando valori alle variabili si ottiene sempre lo stesso assegnamento parziale indipendentemente dall'ordine degli assegnamenti. Ad esempio in un problema di colorazione {a=rosso, b=blu} = {a=blu, b=rosso} sono identici.

Questa proprietà ci permette di evitare di considerare ad ogni livello l'assegnamento di n variabili, limitando la ricerca ad una sola variabile per volta.

La ricerca in profondità per CSP con assegnamento di singole variabili è chiamata **ricerca con backtracking**.

La ricerca con backtracking rappresenta l'algoritmo di ricerca non informata di base per la risoluzione di problemi di soddisfacimento di vincoli.

La ricerca con backtracking è quindi un algoritmo di ricerca in profondità che assegna valori ad una variabile per volta e torna indietro quando non ci sono più valori legali da assegnare,

se viene rilevata un'inconsistenza allora l'algoritmo registra il fallimento del tentativo di assegnazione e tornerà a considerare percorsi alternativi.

La principale differenza tra la ricerca con backtracking e quella in profondità standard consiste nella definizione del problema: se il problema è vincolato e vale la proprietà commutativa allora l'algoritmo potrà considerare singoli assegnamenti ad ogni livello, cosa che non viene fatta nella ricerca standard. La ricerca con backtracking è una specializzazione della ricerca in profondità standard.

Backtracking cronologico: in caso di fallimento si effettua la scelta più semplice possibile, ovvero quella dia ornare indietro alla variabile precedenti e provare ad usare un valore diverso.

Come migliorare l'efficienza dell'algoritmo di base ? Inoltre in alcuni casi è possibile scomporre il problema in sotto-problemi, aumentando ulteriormente l'efficienza.

Quale variabile si deve assegnare al prossimo passo ?

-Opzione 1: si sceglie la variabile con il numero minimo di valori legali. Così facendo la ricerca favorisce le variabili che hanno meno chance di trovare valori che rispettano i vincoli del problema;

-Opzione 2: si scegli la variabile che ha più vincoli con le variabili non assegnate. Così facendo la ricerca risolve prima l'assegnamento delle variabili che potrebbero rivelarsi più critiche;

In quale ordine bisogna testare i valori che può assumere ?

-Opzione 1: testare prima il valore meno vincolante, in questo modo la ricerca tenterà di assegnare alla variabile il valore che meno condizionerà il proseguitò della ricerca;

-Opzione 2: tecnica del forward checking: la tecnica tiene traccia dei rimanenti valori legali per variabili non assegnate, di fatto portando alla potatura delle scelte legali nei successivi passi, una specie di tabella con i valori legali per le variabili non assegnate, portando alla potatura delle scelte legali nei passi successivi;

Come possiamo ridurre le chance di fallimento ?

Le chance di fallimento in un CSP possono essere ridotte con l'inferenza chiamata propagazione dei vincoli.

Propagazione dei vincoli: utilizzo dei vincoli per ridurre il numero di valori legali per ogni variabile, che a sua volta può ridurre i valori legali per un'altra variabile e così via. Così facendo si utilizzano i vincoli per effettuare ragionamenti sui valori legali ammissibili, riducendo lo sforzo di ricerca di una soluzione.

La propagazione dei vincoli la si può intrecciare con la ricerca oppure essere utilizzata come passo di elaborazione preliminare. In alcuni casi, l'elaborazione preliminare può risolvere l'interno problema senza la necessità di eseguire la ricerca.

Il concetto che sta alla base della propagazione dei vincoli è la **consistenza locale**. Se consideriamo ogni variabile come nodo in un grafo e ogni vincolo binario come un arco del grafo, il processo di forzare la consistenza locale in ogni parte del grafo causa l'eliminazione dei valori inconsistenti nel grafo stesso.

L'arco-consistenza è la forma più semplice di propagazione dei vincoli e ha l'obiettivo di rendere ogni arco consistente.

Una variabile in un CSP è arco-consistente se ogni valore del suo dominio soddisfa i suoi vincoli binari. Xi è arco-consistente rispetto a Xj se per ogni variabile del dominio Di c'è un valore nel dominio Dj che soddisfa il vincolo binario sull'arco (Xi, Xj).

Detto in parole povere, per ogni variabile Xi c'è qualche valore legale per le variabili Xj.

Attraverso l'arco-consistenza si può definire l'insieme dei valori validi che la ricerca potrà utilizzare, avendo effetto sia sulla velocità di completamento della ricerca sia sulla riduzione delle chance di fallimento.

Possiamo in qualche modo sfruttare la struttura del problema ?

Conoscendo la struttura del problema si possono sfruttare alcune proprietà del problema migliorandone la struttura. Ad esempio se vi sono nodi senza vincoli o cose di questo tipo possiamo scomporre il problema in sotto-problemi i quali possono essere identificati come componenti connesse del grafo dei vincoli.

La scomposizione è importante perché se ad esempio abbiamo una serie di sotto-problemi con c variabili su un totale di n. Avremo quindi n/c sotto-problemi avremo quindi al massimo d elevato a c foglie anziché d elevato a n.

A questo punto la soluzione di un caso pessimo di un problema costa quindi $d^c * \frac{n}{c}$ anziché d^n, implicando così che la complessità crescerà linearmente con n piuttosto che esponenzialmente.

La scomposizione non è sempre fattibile ed individuabile, ma tuttavia esistono altre strutture di grafi facilmente risolvibili come quelle ad albero. I CSP con strutture ad albero sono risolvibili linearmente al crescere con il numero delle variabili. In alcuni casi è possibile ridurre i grafi di vincoli generici è possibile fare dei ragionamenti in maniera tale da modificare o fondere nodi del grafo.