Appunti di Intelligenza Artificiale

By *@Thisisfava*

2024/2025

Contents

1 Intelligenza

Dare una definizione generale e universale di intelligenza è davvero difficile, dal momento che l'intelligenza può manifestarsi in diversi modi e atteggiamenti. è stata data, tuttavia una **definizione operativa** che permette di descrivere diverse intelligenze rispetto a **come fare** e **cosa fare**.

1.1 Definizione Operativa

La definizione operativa di cui parlavo prima è riassumibile nella seguente tabella

	Umanamente	**Razionalmente**
Pensare	Codificare il funzionamento della mente in un programma	Un programma che usa deduzioni logiche per risolvere il problema
Agire	Un programma che ha comportamento umano	Un programma che prende "buone" decisioni

Pensare Umanamente. Consiste nel ottenere un programma che pensa come il nostro cervello; è impossibile dal momento che ancora oggi non lo conosciamo del tutto.

Pensare Razionalmente. Consiste nel formalizzare tutta la conoscenza tramite assiomi/regole logiche per poter dedurre/inferire ragionamenti

Agire Umanamente. Consiste nell'emulare il comportamento umano (sbagliando, commettendo imprecisioni, ecc...). I captcha sfruttano questa capacità per distinguere i bot dagli umani.

Agire Razionalmente. Consiste nella capacità da parte dell'agente di prendere delle decisioni che lo portano al raggiungimento dei suoi obiettivi.
Nota Bene: Questa è la definizione che utilizzeremo nel corso.

1.2 Problema dell'agire umanamente

Per quanto possa essere facile da realizzare e utile un agente che agisce umanamente (perchè quest'intelligenza è profondamente legata al task per il quale sono costruiti), esiste un problema che li affligge (soprattutto nei casi più complessi): l'impossibilità di determinare il percorso di ragionamento che ha portato l'agente a prendere questa o quella decisione. Il problema è così critico che l'intera industria dell'autonomous driving è stata rallentata.

1.3 Test di Turing

Spesso, tuttavia, la tabella sopra proposta risulta essere poco pratica e molto astratta. Alan Turing propose, invece, un esperimento che permettesse di determinare se l'agente è intelligente o meno a partire dall'essere intelligente (si spera) per definizione: l'essere umano. Una formulazione del Test di Turing è la seguente:
Dati A e B agenti intelligenti (tipicamente un uomo e una donna), e C l'agente di cui testare l'intelligenza:

- C deve indovinare il sesso di A e B
- B collabora con C
- A inganna C

Se swappando A con B si ottengono le stesse percentuali di successo, l'agente pensa umanamente. Questo perchè, per superare il test l'agente dovrebbe possedere le seguenti capacità:

- **interpretazione del linguaggio naturale** per comunicare con l'esaminatore nel suo linguaggio umano
- **rappresentazione della conoscenza** per memorizzare quello che sa o sente
- **ragionamento automatico** per utilizzare la conoscenza memorizzata in modo da rispondere alle domande e trarre nuove conclusioni
- **apprendimento** per adattarsi a nuove circostanze, individuare ed estrapolare pattern

2 Storia dell'IA

Studiare la storia dell'Intelligenza Artificiale è utile per capire quali sono stati i problemi agli approcci usati in passato per capire il successo dei nuovi approcci più moderni.

2.1 Nascita

A differenza di molte altre discipline, l'IA ha una data e un luogo di nascita: il convegno di Dartmouth, organizzato dallo scienziato McCarty, nel **1956**. A quel convegno parteciparono molti informatici, psicologi, statistici, matematici, che avevano il sentore di star affrontando tutti lo stesso problema ma da punti di vista differenti. Alla fine del convegno, questo sentore venne confermato e si diede un nome a questo importante problema: **Artificial Intelligence**

2.2 Prima era: l'Approccio Simbolico

Dal 1956 alla fine degli anni '60 ci fu il primo boom dell'IA: tale successo è dovuto alla realizzazione di agenti che "ragionassero" tramite manipolazioni simboliche e sintattiche, e tramite regole di inferenza logica. I primi risultati furono così tanto promettenti che si investì molto in questa tecnologia. Il fatto è che il grande entusiasmo, col tempo, non venne assecondato dai grandi limiti di questa tecnologia:

- **Nessuna conoscenza specifica**: tali tecnologie si basavano SOLO ed ESCLUSIVAMENTE sulla manipolazione sintattica e regole logiche, per cui problemi reali, di dimensioni nemmeno troppo grandi, risultavano impossibili dal momento che queste intelligenze non avevano alcuna conoscenza del dominio applicativo

- **Esplosione combinatoria**: alcuni problemi venivano risolti dall'IA provando diverse combinazioni dei dati di un problema; tuttavia questo approccio portava spesso a tempi di calcolo impraticabili (es: problemi NP-HARD). Dunque su istanze di problemi leggermente più grandi tali intelligenze non scalavano.

- **Limiti di rappresentazione della conoscenza:** l'approccio simbolico usato risultava davvero difficile da implementare per far fronte all'incertezza della realtà e alle situazioni ambigue. Per cui i sistemi realizzati erano molto rigidi e poco scalabili.

Di fronte a tutti questi limiti, i fondi alla ricerca sull'IA vennero immediatamente congelati e si assistette al **primo inverno**

2.3 Seconda era: I Sistemi Esperti

Verso la fine degli anni'70, si pensò che il problema riscontrato nell'approccio precedente fosse dovuto solo alla poca conoscenza del dominio applicativo. Per cui iniziarono a diffondersi agenti detti **Sistemi esperti** poichè **conoscevano** specificatamente il dominio per cui erano realizzati (quindi non vi era alcuna forma di ragionamento). Erano dotati di un sistema di reasoning (IF-THEN-ELSE) per assistere la gestione aziendale e per altre operazioni. I risultati ottenuti da questi sistemi furono così incredibili che si decise di reinvestire TANTISSIMO. Alcune aziende già negli anni '80 aveva realizzato team IA, per lo sviluppo di sistemi esperti, con centinaia di membri. Ma ecco che, nuovamente, l'Hype generato non fu accompagnato dai risultati sperati:

- **Nessun Apprendimento Automatico**: queste macchine, dal momento che mappavano staticamente ad ogni situazione una risposta, non potevano adattarsi alle varie situazioni per cui, ad ogni nuova situazione era necessario che degli esperti aggiornassero le regole di reasoning
- **Non scalabilità**: anche quest'approccio è affetto dal problema di scalabilità; se il primo approccio permetteva una certa forma di ragionamento logico, questo approccio è fondato solo ed unicamente sulla conoscenza. Il fatto è che tale forma di conoscenza rigida (IF-THEN) non può tener conto delle infinite situazioni incerte della realtà.
- **Difficile formalizzazione**: con la crescita di dimensioni dei sistemi esperti, gli scienziati iniziarono ad avere difficoltà a formalizzare regole sensate e coerenti con le precedenti (il ragionamento umano non è quasi mai algoritmico e lineare).
- **Costi Eleveati**: per tutti questi motivi, la manutenzione e aggiornamento di questi sistemi risultò essere, nel tempo, COSTOSISSIMA.

Ed ecco che arrivò il **secondo inverno dell'IA**.

2.4 Un Nuovo Approccio

In generale, possiamo dire, che i primi due approcci esplorati per l'IA fossero fallimentari perchè cercavano di risolvere problemi tipicamente umani con il paradigma tradizionale dell'informatica:

1. Analizzo il problema
2. Creo l'algoritmo
3. Passo i dati del problema all'algoritmo
4. Ottengo il risultato

Il nuovo approccio usato è invece detto **Machine Learning**:

1. Passo i dati di un problema e le soluzioni corrispondenti (detta Esperienza) ad un computer
2. Tale genererà un programma che possa trasformare i dati in input nelle soluzioni date

3. Tale programma potrà essere eseguito su nuovi input (con o senza buoni risultati)

Purtroppo nei primi anni un tale approccio non era minimamente affrontabile per alcuni motivi:

- **Mancanza di Dati**: per addestrare bene una rete neurale è necessaria una quantità di dati umani IMMENSA, cosa che negli anni 60-80 era impossibile. Oggi, grazie ad internet e ai social network, in rete sono disponibili una quantità quasi infinita di dati (soprattutto testuale).
- **Mancanza di Potenza Computazionale**: per addestrare in tempi utili una rete è necessaria una grande potenza computazionale, potenza che nei decenni successivi si è sviluppata grazie all'industria dei Videogames e delle Schede Grafiche.

3 Agenti

3.1 Definizione

"Un agente è un ente immerso nell'ambiente. L'agente percepisce l'ambiente tramite ***percettori*** *ed agisce tramite* ***attuatori****"*[1].
Tramite la percezione, l'ambiente modifica lo stato dell'agente. Tramite l'azione, l'agente modifica lo stato dell'ambiente. Percezioni e azioni possono essere concepite come flussi di informazioni.

3.2 Caratteristiche dell'Ambiente

Un ambiente può essere:

- **Fully/Partially Observable**: nel primo caso l'agente conosce tutto lo stato dell'ambiente (es: a scacchi). Nel secondo caso, l'agente ne conosce solo una parte (es: il mondo reale)
- **Single/Multi Agent**: in un ambiente l'agente può essere unico e indipendente oppure deve interagire con altri agenti (in competizione, in cooperazione, ecc...)
- **Deterministico/Stocastico**: nel primo caso, l'agente conosce apriori l'effetto di ogni azione sull'ambiente. Nel secondo caso, l'effetto può essere solo stimato.
- **Statico/Dinamico**: l'ambiente può non cambiare o cambiare
- **A tempo discreto/continuo**
- **Conosciuto/Sconosciuto**: l'agente può conoscere le regole e le caratteristiche del dominio in cui opera o deve scoprirle man mano.

[1] Definizione di Russel-Narvig

3.3 Tipi di Agenti

3.3.1 Agente Reflex

L'agente reattivo è sicuramento quello più semplice (ma anche usato): è basato sulla statica mappatura di percezione-azione. La particolarità di questo agente è l'assenza di stato: la mappatura non cambierà mai perchè non viene tenuta in considerazione lo storico delle percezioni (è come se fosse un circuito combinatorio, una funzione ben definita).

3.3.2 Agente Reflex con Modello

Più adatto in ambienti parzialmente osservabili, l'Agente Reflex con Modello tiene traccia della parte dell'ambiente che non può osservare nell'istante corrente. Per poter far ciò, l'agente deve poter conoscere:

- **Le leggi che descrivono l'ambiente**: (o quelle sufficienti) per poter determinare come può evolvere l'ambiente a prescindere dalle azioni
- **Gli effetti delle azioni**: per poter determinare l'effetto delle azioni dell'agente sull'ambiente

Questi 2 tipi di conoscenza vengono detti **Modelli del Mondo**

3.3.3 Agente basato su Goal

Un altro tipo di Agente è quello basato su Goal, ossia, quello che conosce l'obiettivo da raggiungere e deve poter calcolare, per ogni azione quanto l'agente si avvicina a tale obiettivo. Questa operazione è semplice se il calcolo si può fare in pochi passi, ma se si deve valutare lo storico delle azioni, in questo caso non lo è più. Esiste un'intera branca dell'IA che si occupa della Ricerca e Pianificazione.

Il concetto di obiettivo, tuttavia, è limitante: in base ad un obiettivo si possono scartare gli stati che ci allontanano e gli stati che ci fanno avvicinare all'obiettivo, ma ancora è difficile **quantificare** la distanza dall'obiettivo. Esistono diversi parametri che possiamo utilizzare per quantificare la bontà di un'azione:

- **Valore Atteso**: Un primo che viene usato per quantificare la bontà di un'opzione tra le tante è il valore atteso (il prodotto tra la vincita e la sua probabilità, prendendo come esempio quello della lotteria)
- **Propensione/Avversione al rischio**: un altro parametro riguarda quanto è propenso al rischio l'agente; in base a quello si può prediligere la minima vincita con alta probabilità o massima vincita con bassa probabilità
- **Utilità**: questo parametro permette di quantificare quanto è utile un premio rispetto ad un altro (es: 100$ per un povero sono molto utili. Per un ricco, invece, sono poco utili).

3.3.4 Agente basato su Utilità

Un agente basato su Utilità è un agente con modello le cui decisioni vengono prese non per raggiungere un obiettivo, ma per massimizzare il grado di "contentezza" dell'agente stesso. Per apprezzare meglio il concetto di Utilità e come questo abbia avuto importanti ripercussioni in ambito IA è necessario introdurre una serie di formalismi.

3.4 Relazione di Preferenza

3.4.1 Definizioni

- $S = \{s_1, s_2, \dots\}$ insieme degli stati possibili
- $s_i \succeq s_j$ è detta **preferenza debole** di s_i a s_j
- $s_i \succ s_j$ è detta **preferenza (stretta)** di s_i a s_j
- $s_i \sim s_j$ è detta **indifferenza** di s_i a s_j

Grazie alla probabilità e alle definizioni precedenti è possibile definire il concetto di **Lotteria** l (che può essere vista anche come una funzione di massa):

$$l = [p_1 : s_1, p_2 : s_2, \dots] \tag{1}$$

dove s_i è il possibile esito della lotteria, $p_i : s_i$ è la probabilità che si verifichi s_i. Inoltre è necessario che $\sum_i p_i = 1$

3.4.2 Proprietà

Affinchè una relazione sia di preferenza deve rispettare le seguenti proprietà

- **Completezza**: è necessario che ogni preferenza sia comparabile

$$s_1 \succ s_2 \vee s_1 \succeq s_2 \vee s_1 \sim s_2 \quad \forall s_1, s_2 \tag{2}$$

- **Transitività**: Dati $s_1 \succ s_2$ e $s_2 \succ s_3$ allora:

$$s_1 \succ s_3 \tag{3}$$

- **Sostituibilità**: Dati 2 stati s_a e s_b indifferenti, le lotterie, che contengono sia il primo stato che il secondo stato con la stessa probabilità q, devono essere altrettanto indifferenti

$$[q : s_a, p_1 : s_1, \dots] \sim [q : s_b, p_1 : s_1, \dots] \quad \text{se } s_a \sim s_b \tag{4}$$

  Inoltre è necessario che $q + \sum_i p_i = 1$

- **Decomponibilità (Not Fun In Gambling)**: giocare alle varie lotterie in una qualsiasi sequenza non deve influenzare le probabilità di vincita. In parole povere, le varie sottolotterie di una lotteria devono essere indipendenti tra loro. Formalizzando:
  Detta p_i^e la probabilità con cui la lotteria e seleziona lo stato s_i, allora date 2 lotterie l_1, l_2 se $l_1 \sim l_2$ allora
$$p_i^{l_1} = p_i^{l_2} \tag{5}$$

- **Monotonicità**: Dati stati preferibili e delle probabilità, la lotteria preferibile è quella che assegna la probabilità maggiore allo stato più preferibile, ossia:
  Se $s_1 \succ s_2$ e $p > q$ allora:
$$[p : s_1, (1-p) : s_2] \succ [q : s_1, (1-q) : s_2] \tag{6}$$

- **Continuità**: Dati 3 stati, uno più preferibile dell'altro, esisterà sempre un valore tra 0 e 1 che renda la lotteria, tra il più preferibile e il meno preferibile, indifferente allo stato "intermedio", ossia:
  Dati $s_1 \succ s_2 \succ s_3$, $\exists p \in [0, 1]$ tale che:
$$s_2 \sim [p : s_1, (1-p) : s_3] \tag{7}$$

3.5 Teorema di Neumann-Morgenstein

Data una relazione di preferenza che rispetta le proprietà (2),(3),(4),(5),(6),(7) allora $\exists u : \mathcal{L} \longrightarrow [0, 1]$ (dove $\mathcal{L}$ è l'insieme delle possibili lotterie) tale che:

$$s_1 \succ s_2 \Longleftrightarrow u(s_1) > u(s_2) \tag{8}$$

$$u([p_1 : s_1, \dots]) = \sum_i p_i u(s_i) \tag{9}$$

La u viene detta *funzione di utilità* mentre $u(s_i)$ è detta *utilità dello stato i*. In parole povere, il teorema afferma che, data una relazione di preferenza che rispetta quelle proprietà, è possibile definire una funzione che associa ad ogni stato un'utilità, dunque se uno stato è preferibile ad un altro, la sua utilità sarà maggiore; inoltre l'utilità di una lotteria è definibile come il valore atteso della funzione utilità. Dunque, per raggiungere lo stato di massima felicità, l'agente deve massimizzare una funzione (che è appunto u). Dunque, grazie a questo importante teorema, siamo riusciti a ricondurre una forma di ragionamento nella massimizzazione di una funzione (che è un problema facilmente attaccabile)

4 Problemi di Search

L'insieme dei problemi di Search è un insieme di problemi legati all'inferenza (piuttosto che al Machine Learning). Tali problemi vengono formulati e risolti da un agente per trovare il percorso che li porterà ad uno stato obiettivo; per fare ciò, considereremo un ambiente che è:

- **Statico** dal momento che assumiamo che durante la ricerca il mondo non cambi (altrimenti la ricerca sarà inutile)
- **A Singolo Agente** per semplificare la situazione
- **Completamente Osservabile** per poter conoscere lo stato iniziale dell'agente
- **Discreto** in modo da poter descrivere i passi risolutivi in maniera discreta
- **Deterministico** perchè ad ogni azione devo essere sicuro del suo effetto per la computazione dello stato successivo

4.1 Formulazione del Problema

Per la descrizione dei problemi di Search useremo queste convenzioni:

- $S = \{s_1, s_2, \dots\}$ è detto *Insieme degli stati* (deve essere finito)
- $s_i \in S$ è detto *stato iniziale*
- $s_G \in S$ è detto *stato di Goal* (può essere più di uno)
- $A(s_i) = \{a, b, c, \dots\}$è *l'insieme delle azioni possibili allo stato i*
- $f(s_i, a)$ con $s_i \in S$ e $a \in A(s_i)$ è detto *modello di Transizione* o *Funzione Successore*; corrisponde allo lo stato successivo
- $c(s_i, a, f(s_i, a))$ è detto *costo Additivo*. Una cosa da tenere a mente per il costo additivo è che i vari costi devono poter essere tutti sommabili (non possono essere grandezze diverse)

Un'altra formulazione interessante del problema usa il *Grafo degli Stati*, dove gli archi sono le azioni e i nodi sono gli stati.

4.2 Classificazione dei problemi

In base a cosa cercare, i problemi di Search possono essere di 3 tipi:

- **Fattibilità**
  I problemi di Fattibilità hanno come obiettivo di rispondere la domanda: *Esiste un percorso che mi porta da s_i ad un s_G?*. In questi problemi bisogna quindi esplorare un qualunque percorso che mi porti all'uscita del labirinto, senza sapere necessariamente la sequenza di azioni o quella più efficiente/interessante.

- **Approssimazione**
  I problemi di Approssimazione, invece, ricercano una soluzione che soddisfi alcune garanzie (es: "il percorso trovato dev'essere al massimo il 30% peggiore dell'ottimo"). Chiaramente questi tipi di algoritmi sono difficili da progettare dal momento che dimostrare tali garanzie è davvero arduo.

- **Ottimizzazione**
  Sono problemi che richiedono il percorso più bello/efficiente/interessante rispetto a tutti gli altri (e di dimostrarlo) che mi porti allo stato obiettivo.

In generale dobbiamo dire che, negli ultimi 2 casi, il risultato del problema di Search è un albero la cui radice è lo stato iniziale e in cui un ramo porta allo stato obiettivo.

4.3 Approccio Esaustivo/Esplicito

Un primo approccio che potremmo formulare, per risolvere tali problemi, è quello di precalcolare tutti i possibili percorsi e selezionare quello più efficiente. Chiaramente questo approccio è INUTILIZZABILE in problemi di dimensione reale (ma nemmeno troppo grandi): tutti i possibili passi per la risoluzione del Cubo di Rubik, per esempio, sono circa $4,33 * 10^{43}$ permutazioni, cosa che non è possibile contenere tutta in memoria.

4.4 Approccio Implicito

In questo caso, invece, piuttosto che enumerare tutti i possibili percorsi, si esplorano solo quelli più "interessanti" a partire dallo stato iniziale. Dunque, piuttosto che tenere in memoria tutti i possibili percorsi, tengo solo quelli che effettivamente ho esplorato ed eventualmente scarto quelli non ottimi.

4.5 Caratteristiche di un Algoritmo

D'ora in poi, gli algoritmi di Search che mostreremo nel corso verranno valutati in base ai seguenti parametri:

- **Correttezza**: questa proprietà afferma che, se l'algoritmo restituisce un risultato, questo dev'essere conforme alle specifiche (cioè, se si richiede, per esempio, di trovare il percorso migliore, il risultato dell'algoritmo DEVE RESTITUIRE IL PERCORSO MIGLIORE)

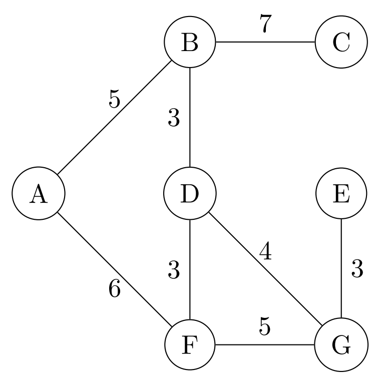
- **Completezza**: se esiste una soluzione al problema, l'algoritmo la troverà sempre. In altri termini, l'algoritmo deve sempre terminare con una risposta in un tempo finito. Nel caso in cui i passi del problema sono infiniti (ovviamente contabili), la completezza viene detta **Sistematicità** e va dimostrata.

- **Complessità Spaziale**: l'uso (nel caso peggiore) della risorsa spaziale (memoria disponibile) per la risoluzione del problema in funzione della dimensione dell'input

- **Complessità Temporale**: l'uso (nel caso peggiore) della risorsa temporale (passi da eseguire) per la risoluzione del problema in funzione della dimensione dell'input

4.6 Problema di Riferimento

Per presentare i seguenti algoritmi, ci riferiremo sempre a questo grafo, in cui gli archi sono bidirezionali (ossia, per ogni X, Y stati del problema $c(X, a, Y) = c(Y, b, X)$ con a l'arco di transizione da X a Y e b l'arco di transizione da Y ad X)



I vari algoritmi che presenteremo differiscono per come rispondono alla domanda: "*Dato che ho ispezionato il nodo, proseguo o vado indietro?*"

4.7 Ricerca Non Informata

4.7.1 Depth-First-Search

Un algoritmo classico di ricerca su grafo è la ricerca in profondità. In questa versione, la DFS è dotata di *Backtracking.* Possiamo dire in generale che l'approccio di questo algoritmo è aggressiva, poichè ricerca subito in profondità la soluzione (potrebbe metterci molto o potrebbe metterci poco), ossia non c'è garanzia. Inoltre possiamo osservare che gli alberi generati dalla DFS sono stretti e lunghi.
Funzionamento:

1. Si parte dal nodo iniziale A (che sarà poi radice dell'albero finale)
2. Se il nodo da esplorare ha dei figli, si aggiungono all'albero i vari figli; in caso ve ne sia più di uno, un *Tie-Breaker* spesso usato è basato sull'ordine lessicografico (quindi, per esempio, se esploriamo A, il primo figlio da esplorare sarà B)
3. Si torna indietro se: il nodo è foglia, oppure se il nodo è già stato visitato

Analisi:

- **Correttezza**: La dfs restituisce sempre un albero (grazie all'uso del Backtracking e all'eliminazione dei loop)
- **Completezza**: La dfs restituisce sempre un albero in cui un ramo contiene lo stato obiettivo (se esiste il percorso)
- **Complessità Temporale**: Dato b il *Branching Factor* e d la *profondità massima*, la complessità di tale algoritmo è esponenziale, ossia $O(b^d)$
- **Complessità Spaziale**: La memoria usata è quella necessaria per generare l'albero; in questo caso la complessità è $O(d)$ ossia la profondità massima del percorso dallo start al goal.

4.7.2 Breath-First-Search

Un altro algoritmo classico di ricerca su grafo è la ricerca in ampiezza. Anche la BFS è dotata di *Backtracking*. Possiamo dire in generale che l'approccio di questo algoritmo è conservativa, poichè ricerca sempre allo stesso livello, garantendo di visitare tutti i nodi. Possiamo inoltre dire che gli alberi generati dalla BFS sono larghi e corti.
Funzionamento:

1. Si parte dal nodo iniziale A (che sarà poi radice dell'albero finale)
2. Se il nodo padre ha dei figli da esplorare, vengono tutti aggiunti all'albero; Si prosegue poi ai figli del primo nodo figlio e così via. Il *Tie-Breaker* che possiamo usare è ancora quello basato sull'ordine lessicografico.
3. Si torna indietro se: il nodo è foglia, oppure se il nodo è già stato visitato

Analisi:

- **Correttezza**: Per lo stesso motivo della DFS
- **Completezza**: Per lo stesso motivo della DFS
- **Complessità Temporale**: Dato b il *Branching Factor* e q la *profondità minima*, la complessità di tale algoritmo è esponenziale, ossia $O(b^q)$ (quindi sempre minore, nel caso peggiore, della DFS)
- **Complessità Spaziale**: In questo caso, dato che non viene allocata altra memoria se non il grafo stesso, la complessità spaziale sarà $O(n)$ con n il numero di nodi.

4.7.3 DFS/BFS Ottimizzati

Gli ultimi 2 algoritmi possono essere ottimizzati introducendo nuove strutture dati, usate per evitare di rivisitare i nodi e quindi per generare alberi più piccoli:

EQL (Enqueued List). La EQL, anche detta *lista di accodamento*, è una lista che tiene traccia di tutti i nodi già visitati; è detta di accodamento perchè ad ogni nuova visita, il nodo visitato viene aggiunto alla fine; quando si deve vistare un nodo si verifica che questo non sia già nella lista; se lo è, tutto il sottoalbero relativo non verrà esplorato. Questa operazione è detta **Potatura** o **Pruning**

Frontiera. Definiamo frontiera l'insieme dei nodi foglia non ancora espansi dell'albero; è detta così dal momento che, per la **Separation Property**, separa la parte dell'albero esplorata da quella ancora non esplorata.
Implementazioni della Frontiera:

- **Caso BFS**: la frontiera viene implementata come **Queue** (una coda FIFO)
- **Caso DFS**: la frontiera viene implementata come **Stack** (una coda LIFO)

Analisi:

- **Correttezza**: L'uso delle EQL pota solo i sottoalberi già esplorati, per cui la correttezza non viene compromessa
- **Completezza**: L'algoritmo è ancora completo per il motivo precedente
- **Complessità Temporale/Spaziale**: anche se abbiamo introdotto queste strutture dati per ottimizzare le operazioni, in realtà la complessità nel caso peggiore non cambia. Grazie a queste ottimizzazioni, tuttavia, è possibile usare in un tempo ragionevole i 2 algoritmi

4.8 UCS (Uniform Cost Search)

Nella presentazione di BFS e DFS, abbiamo sempre ipotizzato di dover risolvere problemi di Fattibilità (paragrafo 4.2), per cui i 2 algoritmi hanno dovuto solo esplorare il grafo fino a generare un nodo obiettivo del problema. Per risolvere il problema di Ottimizzazione, invece, dovremo utilizzare l'approccio conservativo della BFS per progettare un algoritmo che trovi sempre il percorso ottimo.

Cost To Go Function. Per descrivere l'implementazione della UCS è necessario innanzitutto definire la *Cost-to-Go Function* $g(V)$. Dato un vertice V, $g(V)$ è il costo complessivo per arrivare dallo stato A allo stato V tramite un percorso p.

4.8.1 Algoritmo (Ad alto livello)

Inizialmente si aggiunge alla Frontiera il nodo di Partenza A e dopodichè

- Si calcolano per ogni nodo della frontiera il $g(V)$
- Si seleziona il nodo da espandere con $g(V)$ più piccolo
- Ci si ferma solo quando **espandiamo** un nodo Goal

4.8.2 Ottimalità di UCS

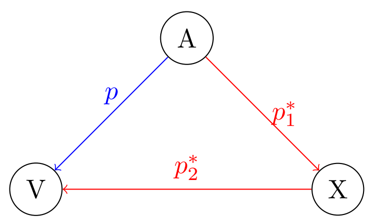
Si può dimostrare che UCS non solo è **corretto** e **completo** e che seleziona il percorso **ottimo**, ma che *"Ogni volta che UCS seleziona per la prima volta un nodo per l'espansione, il percorso che porta ha quel nodo ha costo minimo"*

Ipotesi.

1. UCS seleziona sempre per la prima volta dalla frontiera un nodo V da espandere ottenuto tramite percorso p con costo minore

2. Il percorso p non è ottimo

Dimostrazione. Se p non è ottimo, allora deve esistere sulla frontiera un nodo X tale che la somma dei percorsi $p_1^* = A \longrightarrow X$ e $p_2^* = X \longrightarrow V$ sia $p^* = p_1^* + p_2^*$ e $g(p^*) < g(p)$. Dal momento che $g(p^*) = g(p_1^*) + \Delta p_2^*$ allora $g(p_1^*) + \Delta p_2^* < g(p)$; Δp_2^* è per definizione una quantità non negativa, dunque $g(p_1^*) < g(p)$, ossia UCS ha selezionato prima un percorso p che è peggiore di p_1^*, e ciò porta ad un **assurdo** perchè noi abbiamo definito l'algoritmo in maniera diversa, viene violata la prima ipotesi. Allora $p = p*$.



4.8.3 UCS con EXL

Una prima forma di ottimizzazione della UCS la otteniamo introducendo una struttura dati detta *Expansion List* o *Lista delle Espansioni* Questa lista è molto simile alla EQL, solo che in questo caso si aggiunge un nodo solo quando deve essere espanso, e non quando viene generato.

4.9 Ricerca Informata

Si parla di ricerca **non informata** quando dobbiamo affrontare un problema di Search avendo a disposizione **solo** il grafo e il criterio di scelta del prossimo nodo da esplorare.
Si parla, invece, di ricerca **informata** quando oltre alle informazioni precedenti abbiamo anche una stima $f(V)$ della bontà del nodo V, ossia quanto è distante tale nodo dall'obiettivo. Tramite queste stime è possibile realizzare algoritmi con approccio **Best-First** in cui il criterio di scelta di un nodo da esplorare avviene in base alla minimizzazione di $f(V)$.

4.9.1 A*

L'algoritmo A* è un algoritmo nato negli anni '60 per permettere al primo prototipo di Robot autonomo (Shakey) di arrivare da un punto A ad un punto B col percorso più breve possibile. A* è praticamente identico a UCS, tranne per il fatto che il cost-to-go da ottimizzare è $f(s) = g(s) + h(s)$ dove $h(s)$ è detta **euristica di distanza** dall'ottimo. L'euristica dev'essere una funzione:

- Computabile in tempo costante
- Ammissibile (ossia deve sottostimare il vero valore, o meglio la stima dev'essere ottimista)

Trovare un'euristica sensata al problema, tuttavia, potrebbe non essere semplice.

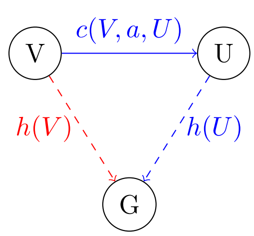
4.9.2 A* con EXL

Dal momento che A* può essere visto come una generalizzazione di UCS (UCS è il caso in cui l'euristica è sempre nulla), possiamo equipaggiare anche A* con una lista delle espansioni per ottimizzare il tempo di ricerca. Purtroppo, in questo caso, non è sempre detto che con la EXL e l'euristica si conservi sempre l'ottimalità (perchè è possibile che si selezionino prima percorsi da espandere più lunghi di altri) ed è per questo che l'euristica h debba essere anche **consistente**.

Euristica Consistente La consistenza (o monotonicità) di un'euristica è una proprietà più forte dell'ammissibilità (quindi la implica) e afferma che:
Dati 2 nodi V, U e un'azione a, allora

$$h(V) \leq c(V, a, U) + h(U) \ \forall V, U \tag{10}$$

La proprietà può essere interpretata come una disuguaglianza triangolare in questa forma (G è il nodo goal):



Ossia, la stima di un nodo non può essere peggiore del costo per arrivare al nodo successivo + la stima del nodo successivo.

4.9.3 Monotonicità di f

Assumendo che h sia consistente, allora possiamo dimostrare che f sia monotona non decrescente.

Dimostrazione. Consideriamo i nodi V, U e un arco che li collega a (simili al disegno precedente) generati dopo p passi dall'algoritmo A*. Sappiamo che $f(U) = g(U) + h(U)$ e che $g(U) = g(V) + c(V, a, U)$; sostituendo otteniamo che $f(U) = h(U) + g(V) + c(V, a, U)$. Allora dalla definizione di consistenza sappiamo che

$$c(V, a, U) + h(U) \geq h(V)$$
$$\underbrace{h(U) + c(V, a, U) + g(V)}_{f(U)} \geq \underbrace{h(V) + g(V)}_{f(V)}$$

Dunque

$$f(U) \geq f(V)$$

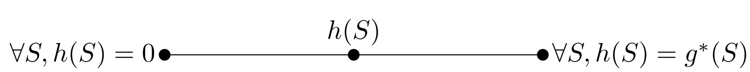
4.9.4 Ottimalità di A*

Ipotesi

1. A* seleziona come primo nodo da espandere V, ottenuto tramite un percorso p
2. p non è ottimo ($p \neq p^*$)

Dimostrazione Se p non è ottimo per ipotesi 2, allora deve esistere necessariamente sulla frontiera un nodo X che si trova sul cammino ottimo p^* verso V. Lungo ogni percorso, abbiamo dimostrato prima, che f è monotona non decrescente dunque $f(V) \geq f(X)$; allora V non può essere stato scelto prima di X: ASSURDO, $p = p^*$

4.10 Progettare un'Euristica



Per progettare un'euristica, come abbiamo detto, è necessario non solo che sia ammissibile ma che la sua computazione sia efficiente (computata in tempo costante). Chiaramente, più è efficiente l'euristica, meno stretta è, quindi dobbiamo trovare il compromesso giusto per avere una buona euristica. Tale ha come estremi:

- L'Estremo sinistro (caso in cui l'euristica è **triviale**)
- L'Estremo destro (caso in cui è il problema ad essere **triviale** e l'euristica conosce già il percorso completo)

Realizzare una buona euristica per A* ci permette di diminuire di molto la dimensione degli alberi e permette di risparmiare tempo e spazio occupati, quindi il nostro obiettivo è di ottenere l'euristica migliore, ossia:
Dati h_1, h_2 euristiche tali che $\forall S, h_1(S) \leq h_2(S)$ allora h_2 domina h_1. Inoltre, anche se non per tutti gli S un'euristica domina l'altra, è possibile creare un'euristica dominante su h_1, h_2, ossia:

$$h_3 = \max\{h_1, h_2\}$$

Un modo che abbiamo per progettare un'euristica è risolvendo un **Problema Rilassato**

4.10.1 Problema Rilassato

Dato un problema P, il suo rilassamento $\hat{P}$ è una sua versione più semplice, ottenuta *rilassando* alcuni suoi vincoli; per esempio, in una mappa, il problema rilassato della ricerca del percorso l'abbiamo ignorando tutti gli ostacoli, ipotizzando di poter attraversare gli edifici. Per questo motivo vale che per ogni S, U nodi e a azione:

$$\hat{c}(S, a, U) \leq c(S, a, U)$$

Risolvere un problema rilassato può essere una buona euristica.

4.10.2 Limiti di A*

Il problema di A* è, essenzialmente, che è troppo rigido: nel caso in cui sulla frontiera vi siano MOLTI nodi tutti con un valore di f molto simile, allora A* perderà molto tempo a espanderli tutti quanti. Può essere anche che un nodo arrivi all'ottimo in un certo numero di passi, mentre un altro nodo ci arrivi con un numero di passi decisamente inferiore. La formulazione di A* più flessibile è il **Focal Search**

4.11 Focal Search

Per l'implementazione del Focal Search è necessario implementare una nuova euristica $\hat{h_F}(n)$ che stima il costo **computazionale** (e non in termini di passi) per raggiungere un goal. Questa euristica poi, rispetto all'euristica h, deve **sovrastimare** (quindi essere pessimista). Possiamo risolvere diversi problemi NP-HARD con questa tecnica (vedi il TSP, potrebbe chiederlo all'esame)

4.11.1 Terminologia

Ai fini di spiegare il funzionamento del Focal Search è necessario introdurre un po' di notazione:

- **F** la frontiera (la stessa di A*)
- $n_{\text{best}} = \arg\min_{n \in F} f(n)$ è il nodo in frontiera che minimizza la f (non necessariamente l'ottimo n^*)

- $\omega \geq 1$ è un parametro che impostiamo noi, in funzione del quale possiamo definire il limite di subottimalità della soluzione
- **OPT** è il costo del percorso minimo
- FOCAL $\subseteq F$ la lista focale, che è una sottolista definita nella seguente maniera:
$$\text{FOCAL} = \{n \in F | f(n) \leq \omega f(n_{best})\}$$

Sulla basta di ciò possiamo definire anche la **regola di espansione**:

$$n_{\text{next}} = \arg \min_{n \in \text{FOCAL}} \hat{h}_F(n)$$

Tale regola impone l'espansione del nodo che è più vicina alla soluzione, ossia che richiede meno passi computazionali per arrivare alla soluzione; dal momento che ci stiamo facendo guidare dall'euristica di costo computazionale $\hat{h}_F(n)$ e non dall'euristica di costo $h(n)$, sicuramente l'ottimalità viene perduta ma tale subottimalità può essere quantificata in funzione di ω

4.11.2 Descrizione Algoritmo

Il focal search procede nella stessa maniera di A* con l'unica differenza che i nodi da espandere non sono tutti quelli in frontiera, ma quelli in lista focale; in particolare definiamo Il nodo selezionato per l'espansione e (nodo che minimizza l'euristica di costo computazionale). Possiamo dunque notare che:

1. $f(n^*) \leq \text{OPT}$ per definizione, dal momento che l'euristica di costo dev'essere SEMPRE ammissibile (quindi deve sottostimare).
2. $f(n_{\text{best}}) \leq f(n^*)$ perchè, per definizione, n_{best} deve minimizzare la funzione di costo f
3. $f(e) \leq \omega f(n_{\text{best}})$ questo perchè $e \in \text{FOCAL}$ e quindi per definizione ha questa proprietà

Unendo tutte queste osservazioni, possiamo affermare che:

$$f(e) = g(e) \leq \omega f(n_{\text{best}}) \leq \omega f(n^*) \leq \omega \text{OPT} \tag{11}$$

Allora deduciamo che

$$g(e) \leq \omega \text{OPT} \tag{12}$$

Ossia $g(e)$ è al massimo omega volte il costo dell'ottimo; questa proprietà è davvero interessante perchè ci permette di impostare ω in base al problema e avere il giusto trade-off prestazioni/subottimalità. Per qquesto motivo il focal search è un algoritmo **BSS (Bounded Suboptimal Search)** perchè la subottimalità è limitata in questo caso da ω; è possibile esprimere $\omega = 1 + \epsilon$ con ϵ la percentuale di subottimalità massima attesa.

4.11.3 Limiti di Focal: Trashing

Un grande problema del focal search è che, quando all'inizio cominciamo a valutare le $f(n)$ e a realizzare la lista focale, è possibile che un n_{best} rimanga in lista focale con basso $f(n)$ (poichè ancora il costo computato $g(n)$ è basso), ma l'euristica di costo computazionale $\hat{h}_F(n)$ è molto alta (poichè ci troviamo all'inizio, quindi molto distanti dal goal). A causa di ciò, eventuali nodi figli espansi non entreranno in lista focale perchè avranno una $f(n) > \omega f(n_{\text{best}})$ per cui vi entreranno solo dopo molto tempo.
Per questo motivo la regola di espansione del focal search è stata rivista: piuttosto che usare $f(n)$ per realizzare la lista focale, vengono prese in considerazione un'euristica h ammissibile e un'euristica $\hat{h}$ inammissibile. Solo dopo viene valutata la $f(n)$. Questo approccio è detto **EES (Explicit Extimation Search)**

5 Adversarial Search

Gli algoritmi di Search finora presentati hanno sempre tenuto conto di un ambiente Single-Agent in cui un obiettivo si raggiungeva trovando un percorso in un grafo. Tutto cambia se invece consideriamo un ambiente Multi-Agent, in cui più agenti cercano di ottimizzare la propria utilità e ogni azione influenza tutti gli agenti. Tuttavia è sempre possibile modellare un ambiente Multi-Agente, ossia un **gioco**, come un **Problema di Decisione**.
In particolare in un gioco:

- Ogni agente deve ottimizzare una propria funzione di utilità
- Ogni azione influenza le azioni di tutti gli altri agenti
- Nella strategia da adottare bisogna tenere in considerazione anche le strategie adottate dagli avversari

Un'altra differenza che possiamo riscontrare con i problemi di Search è che, alla fine del gioco, non si raggiunge un goal ma si vince una *ricompensa*, o *payoff*, e ciò dipende dal risultato del gioco.

5.1 Caratteristiche di un gioco

In teoria dei giochi, un gioco può essere caratterizzato da:

- **Numero di giocatori**: a 2 (caso più semplice) oppure a n giocatori (caso più difficile)
- **Tipi di Agenti**: *razionali* (cioè che in ogni occasione scelgono sempre di ottimizzare la propria utilità) o $\epsilon-$*razionali* (ossia che scelgono di trovare una soluzione peggiore al massimo ϵ)
- **Struttura Sequenziale**: a *turni* o *ad azioni sequenziali* (in realtà è stato dimostrato che una struttura a turni è una generalizzazione della struttura ad azioni sequenziali)
- **Esito delle azioni**: *deterministico* o *stocastico*
- **Struttura dei Payoff**: *a Somma Costante* oppure *a Somma Generica*
- **Ad Informazione**: *completa*, se si conoscono tutti gli obiettivi e azioni degli altri agenti, altrimenti *incompleta*
- **Memoria del Gioco**: *perfetta* se ogni agente conosce TUTTE le mosse compiute dagli altri agenti anche in precedenza, altrimenti *imperfetta*

Dal momento che i giochi possono essere molto complessi e le interazioni tra gli agenti non facili da studiare, sono nati 2 rami di teoria dei giochi che studiano i comportamenti degli agenti in modo diverso:

- **Teoria dei Giochi Competitiva**: studia i comportamenti degli agenti individualmente, come se ogni agente considerasse solo il proprio tornaconto
- **Teoria dei Giochi Cooperativa**: studia le dinamiche delle formazioni delle coalizioni.

Nota Importante

! In questo corso considereremo giochi: a 2 giocatori, ad agenti razionali, a turni, deterministico, ad informazione perfetta e a somma zero (ossia costante).

5.1.1 Formalizzazione di un Gioco

Descrivo qui la notazione usata per descrivere i giochi e i relativi algoritmi all'interno di questo corso:

- **Insieme di Stati**: $S = \{s_1, s_2, \ldots, s_n\}$
- **Stato Iniziale**: $s_k \in S$
- **Agenti**: $I = \{i_1, i_2\}$
- **Azioni Possibili**: $A = \{a_1, a_2, \ldots, a_n\}$
- **Turno**: è l'insieme delle azioni possibili $A(s_k)$ di un giocatore $I(s_k)$ con $s_k \in S$
- **Modello di Transizione**: $f(s_k, a)$ con $a = A(I(s_k))$ l'azione effettuata in un turno e s_k lo stato precedente
- **Stato Terminale**: $s_t \in S$ è stato terminale se e solo se verifica un predicato T che determina la fine di un gioco, ossia se $T(s_t) = 1$
- **Payoff**: $u_i(s_t)$ è il payoff del giocatore i nello stato terminale s_t

5.1.2 Gioco A Somma Zero

Per presentare il primo algoritmo di risoluzione di un gioco , prenderemo in considerazione un gioco a somma zero:
Dato un gioco a 2 agenti, tale si definisce *a somma zero* se e solo se:

$$u_i(s_t) + u_2(s_t) = 0 \ \forall s_t \in S \tag{13}$$

Dove s_t è un qualunque stato terminale. Considerare un gioco a 2 agenti a somma zero permette di semplificare molto i calcoli dal momento che è possibile riscrivere l'utilità di un agente in funzione dell'utilità dell'altro agente. In particolare, massimizzare il proprio payoff significa minimizzare il payoff avversario dal momento che l'avversario perde tanto quanto vinco, ossia:

$$\max\{u_2\} = \max\{-u_1\} = -\min\{u_1\} \tag{14}$$

5.1.3 Strategie

Quello che abbiamo finora descritto sono le *regole* del gioco; un'altra importante parte dei giochi riguarda le *strategie* adottate dagli agenti per massimizzare il payoff. Le strategie si dividono in 2 grandi categorie:

- **Strategia Pura**: se la strategia σ_i, del giocatore i, sceglie, secondo un criterio, un'azione possibile in uno stato del gioco appartenente a $A(s_k)$, ossia:
$$\sigma_i : \{s_k \in S | I(s_k) = i\} \longrightarrow A(s_k) \tag{15}$$
- **Strategia Mista**: se la strategia σ_i, del giocatore i, sceglie un'azione possibile in uno stato del gioco appartenente a $A(s_k)$ secondo uno spazio di probabilità $\Pi(A(s_k))$, ossia:
$$\sigma_i : \{s_k \in S | I(s_k) = i\} \longrightarrow \Pi(A(s_k)) \tag{16}$$

5.1.4 Albero di Gioco

Sebbene un problema di ricerca e un gioco siano sostanzialmente differenti, è ancora possibile modellare un gioco a mo' di albero, detto appunto albero di gioco, ed applicare un algoritmo di ricerca tra quelli presentati precedentemente per trovare la strategia ottima dell'agente. In questo albero, ogni nodo è detto *nodo di decisione* e i nodi foglia sono il nodo terminale del gioco con annesso payoff ai vari giocatori. Una strategia è un percorso all'interno dell'albero, ossia una sequenza di mosse che porta ad uno stato terminale.

5.2 Minimax

Minimax è un algoritmo **Corretto e Completo** che trova sempre la strategia ottima per massimizzare i payoff di un agente e lo fa esplorando l'albero di gioco. Lavora con l'assunzione che l'agente avversario sia **razionale**. Quest'algoritmo:

- è basato su DFS (4.7.1)
- è basato su Backtracking
- effettua il **backup/riporto** dei valori al nodo padre

5.2.1 Descrizione Gioco

Il gioco risolto da questa prima implementazione di Minimax è la seguente:

- **Agenti** $I = \{\text{MAX}, \text{MIN}\}$
- **Gioco a Somma Zero**
- Max avrà sempre utilità non negativa e le conseguenze di MIN saranno sempre negative

5.2.2 Descrizione Algoritmo

1. Si effettua una DFS sull'albero di gioco
2. Ogni volta che si esplora totalmente un sottoalbero:
   - se il nodo padre di tale sottoalbero è di **MAX**, allora riporto al padre il valore **maggiore** tra quelli riportati dai figli
   - se il nodo padre di tale sottoalbero è di **MIN**, allora riporto al padre il valore **minore** tra quelli riportati dai figli
3. Arrivati al nodo radice si trova il *Valore di Gioco.*
4. Si trova la strategia (ossia il percorso) che genera al payoff uguale al valore di gioco.

Bisogna tenere a mente che tale algoritmo lavora sempre con l'assunzione che l'agente avversario sia razionale, perchè se non lo è, l'algoritmo non garantisce di trovare il payoff ottimo; tuttavia ci dà una garanzia: il payoff sarà SEMPRE almeno il valore di gioco.

5.2.3 Minimax a n agenti

La strategia adottata da Minimax può essere estesa a più agenti, tuttavia dovremmo fare alcuni cambiamenti: non ha più senso trattare giochi a somma zero (poichè a più giocatori non ci dà alcun vantaggio nei calcoli); inoltre il payoff di un gioco non può essere più espresso da uno scalare ma da un vettore di payoff, detto profilo del payoff, di dimensione pari al numero di giocatori.

5.3 Ottimizzazioni di Minimax

5.3.1 α, β Pruning

Sebbene l'algoritmo Minimax appena presentato sia corretto e completo, è evidente la sua inefficienza per 2 motivi:

- L'algoritmo deve sempre attraversare tutto l'albero per ottenere il *valore di gioco*
- In caso di alberi di grandi dimensioni, la complessità cresce notevolmente

Così come abbiamo fatto per gli algoritmi di ricerca, è possibile ottimizzare Minimax potando alcuni sottoalberi e risparmiando dunque tempo di computazione (NB: nonostante ciò, la complessità dell'algoritmo rimane esponenziale).

Algoritmo. Per potare i vari sottoalberi, l'algoritmo deve assegnare ad ogni nodo dell'albero una coppia di valori $[\alpha, \beta]$:

- α rappresenta il **minimo valore garantito** per il giocatore **MAX** ad un certo nodo dell'albero
- β rappresenta il **massimo valore garantito** per il giocatore **MIN** ad un certo nodo dell'albero

Durante **l'esplorazione** dell'albero:

- Quando un nodo viene esplorato per la **prima volta**, l'algoritmo gli assegna i valori $[-\infty, +\infty]$, poi:
  - Se il nodo da esplorare è un nodo MAX, viene aggiornato il suo valore di α con il valore più alto dei suoi sottoalberi
  - Se il nodo da esplorare è un nodo MIN, viene aggiornato il suo valore di β con il valore più basso dei suoi sottoalberi
- Quando un nodo viene **completamente esplorato** allora vale che: $\alpha = \beta$.
- Invece, se si scopre che il nodo **appena esplorato** presenta un valore v che è:
  - $v < \alpha$ se il padre è un nodo **MAX**
  - $v > \beta$ se il padre è un nodo **MIN**

  Allora tutti gli altri sottoalberi del nodo padre vengono **potati** e non esplorati.

Analisi L'efficacia del pruning, dunque, dipende molto dallo scoprire le *"killer moves"*, ossia dei valori α, β stringenti per la potatura dei sottalberi. Se i valori più stringenti si trovano solo alla fine dell'esplorazione, il pruning non è più apprezzabile e la computazione risparmiata è minima. Valutiamo, in particolare, la complessità nei 2 casi:

- **Caso Pessimo**: i valori stringenti α, β di un nodo vengono scoperti negli ultimi sottoalberi, per cui non vi è alcuna potatura (stessa complessità di una DFS $O(b^d)$ (vedi 4.7.1))
- **Caso Ottimo**: Per ogni nodo MAX, tutti i suoi alberi vengono esplorati, mentre per ogni nodo MIN, il primo sottoalbero fornisce già i valori α, β per il pruning di tutti gli altri sottoalberi, dunque, è possibile dimostrare, che la complessità è $O(b^{d/2})$, ossia che nello stesso tempo, un MINIMAX con pruning esplora il quadrato dei nodi di un semplice MINIMAX

Rimane dunque evidente il fatto che per la maggior parte dei giochi è **impossibile** esplorare l'albero di gioco e trovare sempre, in tempo ragionevole, il valore di gioco; anzi, se lo fosse, il gioco non sarebbe più interessante (come il tris). Inoltre Minimax risulta essere poco utile quando il tempo per decidere una mossa risulta essere molto limitato.

Esiti di una partita. In generale possiamo dire che una partita tra 2 intelligenze artificiali A e B può concludersi in uno dei seguenti modi:

- A si arrende
- B si arrende
- A e B patteggiano

5.3.2 Funzioni di Valutazione e CUTOFF

Un problema sicuramente noto del pruning α, β è che per trovare un valore α o β, l'algoritmo deve arrivare fino ad una foglia dell'albero, cosa assolutamente impossibile per alberi di gioco davvero grandi (come gli scacchi o il go). Un'intuizione che propose Shannon verso la fine degli anni '50 fu quella di interrompere la valutazione di un nodo (se contiene troppi sottoalberi) e farlo diventare un nodo foglia il cui valore è una stima della bontà del nodo calcolata da una **Funzione di Valutazione**, ossia un'euristica $v(s)$ con s il nodo attuale. La decisione di continuare ad esplorare un nodo non terminale tramite Minimax o approssimarlo tramite Funzione di valutazione viene fatta da un predicato (una funzione booleana), ossia $\text{CUT}(s, d)$ con s il nodo attuale e d la profondità massima di ricerca:

- se $\text{CUT}(s, d) = \text{TRUE}$, allora approssimo il valore del nodo con l'euristica $v(s)$
- altrimenti continuo ad esplorarlo tramite MINIMAX

Man mano che scendo nell'albero, il valore d decrementa fino ad arrivare a 0. Per quanto riguarda **l'euristica** $v(s)$ esistono 2 principali approcci:

Euristiche basate sull'esperienza. Un modo per determinare la stima $v(s)$ del nodo s è **apprendendo** i vari pesi delle feature da un dataset di partite e, dunque, combinare con questi pesi le varie feature; il dataset di partite può essere generato facendo simulare alla macchina milioni di partite e, a partire dall'esito di una serie di mosse, quantificare l'importanza di ogni feature per vincere il gioco. (Problema di regressione)

Combinazione Lineare. Quando non si ha a disposizione un grande dataset di partite o le risorse computazionali per analizzarlo, è possibile affidarsi ad un vettore di pesi $\langle \omega_1, \omega_2, \dots, \omega_n \rangle$ delle n feature dato da un esperto. Tali tecniche, tuttavia, funzionano solo se:

- l'euristica è **efficiente**
- è possibile determinare **a mano** le varie feature

Si stanno però facendo molti studi per permettere ad una macchina di apprendere anche quali feature considerare (e non solo i pesi) nella descrizione di uno stato tramite il **deep learning**. Per quanto riguarda le funzioni di **CUT** esistono alcuni approcci utilizzabili:

Iterative Deepening. Una possibile implementazione di CUT è tramite la funzione di *approfondimento iterativo* della DFS e consiste nell'incrementare d ad ogni nuovo sottoalbero esplorato, cioè consiste nell'aumentare il *budget di esplorazione* e a scommettere sempre di più su un ramo, scendendo via via più in profondità

Quiescient Search. Un altro approccio utilizzabile per CUT risiede nel riconoscimento dei nodi **Quiescienti** e di quelli **Non Quiescenti**. Un nodo è quiescente se non è *interessante*, ossia una qualunque mossa non stravolge gli esiti del match; con questo approccio dunque, tutti gli stati potenzialmente quiescienti vengono approssimati, mentre quelli più interessanti vengono esplorati con MINIMAX.

5.3.3 Tabella delle Trasposizioni

Molto spesso, esplorando l'albero di gioco, è possibile che certe mosse siano state già esplorate, o per simmetria, o perchè ci si è arrivati tramite un ordine diverso di mosse. Nel caso **peggiore**, esistono $m!^n$ stati diversi, dove m sono il numero di mosse per generare gli stati, n sono i giocatori, ma nella stragrande maggioranza dei giochi gli stati non saranno tutti diversi. Per evitare di ricalcolare ogni volta la strategia per mosse già considerate in precedenza è possibile salvare le mosse in una **Tabella delle Trasposizioni**:
La tabella delle trasposizioni è una **HashTable** che conserva, per ogni stato $\overline{s}$ esplorato:

- il valore $\overline{v}$ del nodo
- la mossa $\overline{a}$ che ha generato $\overline{s}$
- la profondità $\overline{d}$ a cui è arrivato l'algoritmo per esplorare $\overline{s}$ e rappresenta la forza con cui l'algoritmo ha esplorato quello stato.

Il valore $\overline{v}$ può essere:

- Il valore MINIMAX effettivo se tutto il sottoalbero di $\overline{s}$ è stato esplorato
- Un α/β bound altrimenti

Funzionamento. Ragioniamo ora sull'algoritmo che sfrutta **Iterative Deepening**, **Pruning** α, β e **Transposition Table**:
Consideriamo la funzione di *decisione* $D(s_k, d_{\mathrm{MAX}}, \mathrm{TT})$ dove:

- s_k è lo stato k-esimo
- d_{MAX} è la massima profondità di esplorazione
- TT è la tabella di trasposizione

Per ogni stato s selezionato dalla DFS:

- se $s \notin$ TT allora si aggiunge in TT il record $\langle \overline{s} \leftarrow s, \overline{a} \leftarrow a, \overline{d} \leftarrow d \rangle$
- altrimenti:
  - se $\overline{d} \geq d_{\text{MAX}}$ allora interrompo l'esplorazione del nodo e lo stimo con l'euristica v
  - se $\overline{d} < d_{\text{MAX}}$ allora lo espando

La scelta di espandere un nodo già esplorato in precedenza con valore di profondità maggiore serve per migliorare la stima, generare bound più forti e dunque aumentare la possibilità di potatura.

Nonostante tutte le ottimizzazioni presentate, è possibile dimostrare che andando sempre più in profondità nell'albero, il tempo risparmiato dalla Tabella di trasposizione diventa trascurabile.

5.3.4 Forward Pruning (PROBCUT)

Quando l'albero di gioco è troppo profondo, riuscire ad esplorare anche un ramo può risultare troppo costoso, per cui la funzione di valutazione può intervenire non solo in *orizzontale* ma anche in *verticale*: tramite alcune condizioni, per esempio in caso di posizioni particolarmente svantaggiose, è possibile troncare completamente la ricerca in profondità e stimare il valore del nodo. Chiaramente troncare così la ricerca non garantisce sempre di trovare l'ottimo perchè c'è una probabilità di troncare il ramo con il valore MINIMAX più alto, nonostante ciò viene utilizzato quando la risorsa tempo diventà molto più importante della precisione delle mosse. Questo algoritmo è il fratello del Beam Search, ossia il **PROBCUT**.

5.4 Expectimax

Finora i giochi considerati sono stati deterministici: a partire dall'assunzione di razionalità dell'agente avversario era facile prevedere la mossa scelta da costui nell'albero di gioco, per cui era possibile sempre massimizzare il valore di MINIMAX. Il discorso cambia quando entra in gioco una variabile aleatoria che rende **stocastica** la natura del gioco. Per modellare questo tipo di giochi, si introduce un nuovo giocatore detto *Nature* $\mathcal{N}$ che ha le seguenti caratteristiche:

- Tutti i player conoscono la strategia di $\mathcal{N}$
- La strategia di $\mathcal{N}$ è mista e la sua distribuzione Π di probabilità è nota
- $\mathcal{N}$ non riceve *payoff*

Dal momento che non si sa apriori l'esito delle mosse di $\mathcal{N}$, l'algoritmo può solo massimizzare il **Valore Atteso** dello stato; per questo motivo l'algoritmo MINIMAX dei giochi Stocastici è detto **EXPECTIMAX**. Anche in questo caso possiamo implementare **pruning** questa volta basati su bound di valore atteso dei nodi.

5.4.1 Monte Carlo Tree Search (MTSC)

Sebbene si possa pensare che l'incertezza e stimare troppo possano creare troppi ostacoli, in realtà questa volta non è cosi: se il modello probabilistico su cui si basa la stima è corretta, e se la stima è non deviata e consistente, alla lunga la stima tenderà al valore effettivo di MINIMAX del nodo. Su questo principio si basa il **Monte Carlo Tree Search**: quest'algoritmo, oltre a selezionare ed espandere un nodo, introduce una fase di **simulazione**, ossia gioca, a partire da un determinato stato, milioni di partite (giocate a caso) per "raffinare" sempre di più la stima del valore atteso di una particolare azione tra quelle disponibili

5.4.2 Multi-Armed Bandit Problem (MAB)

Chiaramente ci aspettiamo che il MTSC scelga una mossa in un tempo finito, per cui il numero di simulazioni che potrà effettuare per le varie azioni possibili sarà limitato. Sarà necessario quindi distribuire questa risorsa tra le varie azioni in modo da non escludere alcuna possibilità e dunque perdere l'ottimalità.

Approccio Naive. Si potrebbe scegliere di equidistribuire le simulazioni a tutte le azioni, pure a quelle meno promettenti. In questo modo, alla fine, si sceglie la mossa $a^* = \arg\max_i\{\omega_i\}$.

Quest'approccio funziona, ma si può fare di meglio: quando devo esplorare tutte le azioni? Quando scommetto di più sull'azione migliore? Questo viene detto **Dilemma Exploration/Exploitation** e possiamo formularlo con il problema **Multi-Armed Bandit Problem** che recita cosi:

Un bandito si trova in un casinò per arricchirsi. Per farlo deve investire le sue risorse su K slot machines e, per massimizzare i guadagni, deve scommettere il più possibile sulla slot machines con valore atteso più alto. Purtroppo però i valori attesi di tutte le slot machines sono ignoti all'inizio, per questo il bandito dovrà alternare fasi di ***esplorazione*** *in cui cerca di stimare il valore atteso di tutte le slot machines, e fasi di* ***exploitation****, in cui scommette tanto sulla macchina che sembra più promettente.*

Esistono diverse soluzioni a questo problema, tra cui:

Upper Confidence Bound. Questa soluzione consiste nel scegliere la slot machine i che **massimizza** il coefficiente UCB_i, che è definito nella seguente maniera:

$$UCB_i = \omega_i + c\sqrt{\frac{\log N}{N_i}} \tag{17}$$

Dove:

- ω_i è la winning rate della slot machine i-esima (rappresenta la fase di exploitation)
- c è un fattore che bilancia l'esplorazione e l'exploitation
- $\sqrt{\frac{\log N}{N_i}}$ determina il fattore di esplorazione, dove N è il numero di slot machines e N_i è il numero di scommesse sulla slot i-esima

Grazie a questo coefficiente, l'algoritmo tenderà inizialmente a preferire slot machine con poche scommesse e, se si vince, ω_i aumenterà di conseguenza. Il vantaggio di implementare MTCS con UCB è:

- L'algoritmo diventa **Anytime**, cioè, imponendo un tempo limite di decisione, l'algoritmo ha sempre una risposta da dare.
- L'algoritmo è **informato** senza usare alcuna **euristica**

6 Constraint Satisfaction Problem (CSP)

Il Problemi CSP rappresentano una categoria di problemi in cui si è informati rigurado la natura del problema (cioè i vincoli) e bisogna creare uno stato "a tavolino" che rispetti tutti i vincoli imposti dal problema. Chiaramente, problemi del genere sono affrontabili da algoritmi di Search, tuttavia sono notevolmente inefficienti rispetto agli algoritmi di CSP (tipo l'AC-3). In questi algoritmi, in particolare, si inizia con un set di variabili nulle che descrivono lo stato del problema; di volta in volta, si scelgono degli assegnamenti che vanno a rispettare i vincoli imposti dal problema.

6.1 Formalizzazione

Di seguito presento tutta la formalizzazione matematica che utilizziamo per rappresentare un problema:

- $X = \{X_1, X_2, \ldots, X_n\}$ è uno **stato** del problema
- Ogni variabile X_i è definita in un Dominio D_i
- Esistono C vincoli

Rappresentazione dei Vincoli. Un primo modo di rappresentare un vincolo è tramite tupla:

$$\langle \text{k Variabili}, \text{Relazione} \rangle$$

Con $k \leq n$ elenco delle variabili coinvolte in un vincolo e, per quanto riguarda la relazione, questa può essere di vario tipo (di disuguaglianza, uguaglianza, ecc...)

Tipologie di Vincoli. Esistono diverse tipologie di vincoli:

- **Vincoli Unari**: riguardano una sola variabile
- **Vincoli Binari**: riguardano solo 2 variabili
- **Vincoli n-ari**: riguardano n variabili

Possiamo dare una rappresentazione del problema a noi abbastanza familiare: il **grafo dei vincoli**. Per farlo, tuttavia, è necessario trasformare tutti i vincoli n-ari in vincoli binari, e per farlo dovremmo aggiungere un certo numero di **variabili ausiliarie**. Questo è possibile se le variabili in questione hanno **domini finiti**.

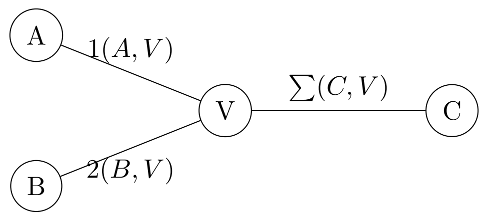
Esempio. Ipotizziamo di voler "binarizzare" il vincolo $A+B=C$ con A, B, C 3 variabili del problema. Per rendere binario il vincolo, dobbiamo introdurre una nuova variabile V con dominio $D_A \times D_B$, ossia una tupla che associa sempre un valore del dominio di A e un valore del dominio di B. Possiamo definire questi vincoli di **assegnamento** nella seguente maniera:

$$i(A, V)$$

E corrisponde essenzialmente a $V = D_A[i]$ cioè al valore i-esimo del dominio di A. Analogamente vale per B. Per legare V e C dobbiamo intrudurre un **vincolo di somma** ossia:

$$\sum(C, V)$$

Ossia che vincola C ad essere la somma di V.
A partire da questa formalizzazione possiamo costruire il corrispettivo grafo dei vincoli:



Risolvere un CSP significa:

- Trovare uno **stato obiettivo**
- Dimostrare che è **insoddisfacibile**

6.2 Tecniche per il CSP

Dal momento che noi conosciamo apriori i vincoli del problema, tali informazioni ci permettono di fare potature nell'grafo dei vincoli molto più efficaci, ossia riducendo i domini delle variabili, eliminando i valori che violano i vincoli del problema; questa tecnica è detta **Constraint Propagation** e ve ne sono di 3 tipi:

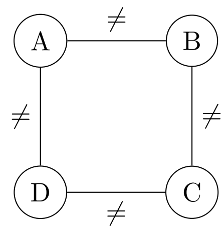
Consistenza di Nodo. Si applica ai **vincoli unari**: un nodo si dice consistente se il dominio non contiene valori che violano un vincolo unario. Il concetto di consistenza di nodo permette di ridurre un dominio escludendo i valori inammissibili.

Consistenza ad Arco. Si applica ai **vincoli binari**: un nodo X_i si dice *consistente ad arco* a X_j se per ogni valore in D_i esiste almeno un valore in D_j che soddisfa il vincolo binario. Grazie a questo concetto di consistenza, è possibile escludere da un dominio tutti i valori che impediscono la consistenza ad arco con l'altra variabile. Una cosa da mettere in evidenza è che un vincolo binario **deve essere letto in entrambe le direzioni**, per esempio, $A > B$ deve essere letto sia come $A > B$ sia come $B < A$, in questa maniera possiamo togliere da D_A tutti i valori per cui non esiste alcun valore minore in D_B, così come posso escludere da D_B tutti i valori che sono maggiori di ogni altro elemento in D_A.

Caso Patologico:

Dati i domini:

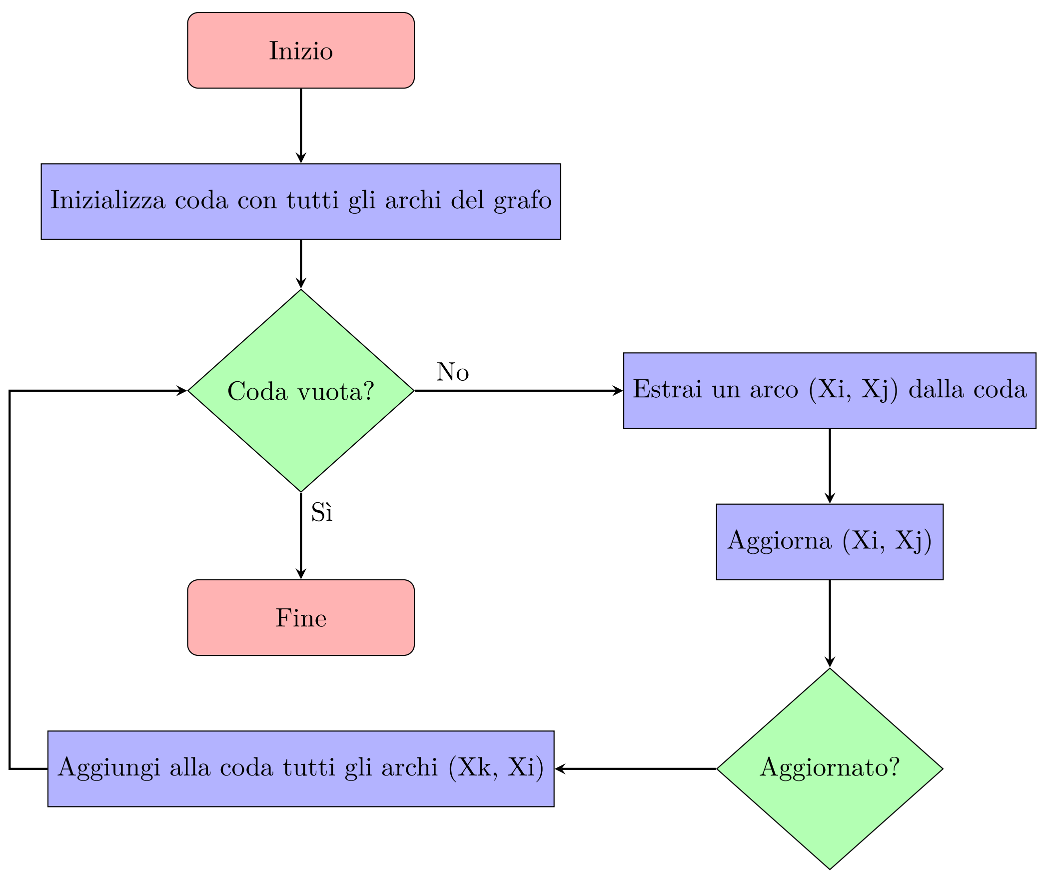
- $D_A = \{2, 4\}$
- $D_D = \{2, 4\}$
- $D_B = \{4, 5\}$
- $D_C = \{4, 5\}$



In realtà l'intero grafo è già **Arc-Consistent**

6.2.1 AC-3

Un famoso algoritmo che risolve CSP sfruttando le arc-consistencies è l'**AC-3**:



6.2.2 Path Consistency

Possiamo considerare la **Node Consistency** come consistenza di **ordine 0** e la **Arc Consistency** una consistenza di **ordine 1**. Con questo ragionamento possiamo estendere il concetto di consistenza a **ordine 3**, anche detta **path-consistency** e ad un generico **ordine n**.

Definizione. Per ogni assegnamento (A, B) legale (ossia arc-consistent), C è path consistent con (A, B) sse $\exists v \in C$ tale che:

$$\begin{cases} C \text{arc-consistent con} A C \text{arc-consistent con} B \end{cases} \tag{18}$$

6.2.3 Backtracking Search

Un altro algoritmo usato per la risoluzione dei CSP è il Backtracking Search: dato un vettore di n variabili $\Gamma = \langle x_1, x_2, \ldots, x_n \rangle$, ad ogni step si prova ad assegnare un valore ad una variabile. In particolare:

1. **Start**: si parte con l'assegnamento nullo $\Gamma = \emptyset$
2. **Espansione del Nodo**: fintanto che non ho trovato un assegnamento valido, scelgo un valore per una variabile
3. **Backtracking**: se mi accorgo che l'assegnamento mi porta ad un vicolo cieco, ritorno indietro e provo con un nuovo assegnamento.

Chiaramente l'algoritmo affronta un approccio esaustivo e, nel caso peggiore, il numero di assegnamenti da provare è m^n dove m è la dimensione del dominio e n è il numero di variabili. A differenza dei problemi di search, tuttavia, in questo caso non siamo interessati all'ordine degli assegnamenti ed è per questo motivo che possiamo scegliere arbitrariamente un ordine sia per i valori che per le variabili senza invalidare l'ammissibilità di Γ

6.2.4 Ordering Delle Variabili

Per ottimizzare il backtracking, possiamo implementare delle euristiche che scelgono il prossimo nodo da espandere:

Ordine Lessicografico è un ordine statico, unfair e quasi mai utile.

Ordine Random è dinamico e a differenza del precedente è più fair

Euristica di Grado (EG) con questa euristica si sceglie sempre prima la variabile con il **maggior numero di vincoli** che vincolano variabili ancora non assegnate

Minimum Remain Value (MRV) Si scelgono le variabili con i domini più piccoli

I classici solutori CSP basati su backtracking di solito sono basati su **EG** e in caso di spareggio vengono usati **EG + MRV**. Il grande vantaggio di usare EG o MRV è il fatto che conducono subito a **vicoli ciechi**, cioè sollecitano il principio di **fail-first** in modo da escludere a monte molti percorsi fallimentari e non accorgersene troppo tardi.

6.2.5 Euristica per i Valori

Anche per i valori da assegnare è possibile realizzare delle euristiche apposite, come il **Least Constraining Value (LCV)**: questa euristica sceglie i valori che escludono **meno valori possibili** nei domini delle altre variabili.

Implementare **EG/MRV** e **LCV**, che hanno tendenze opposte, funzionano bene insieme: se le prime 2 euristiche ci avvicinano ai vicoli ciechi la LCV ci allontana scegliendo i valori che soddisfano più vincoli possibili. In questa maniera si evitano a tutti i costi i backtracking.

Forward Checking Questa tecnica, basata sulla **constraint propagation**, consiste nell'escludere apriori tutti i nodi che, se espansi, so già che portino a vicoli ciechi.

6.3 Local Search

Nel mondo del CSP possiamo dire che esistano 2 scuole di pensiero: la prima scuola è quella *costruttiva*, cioè a partire da un assegnamento vuoto ci costruiamo man mano lo stato ammissibile. Un altra scuola di pensiero ragiona al contrario: a partire da un assegnamento con valori random inammissibile lo si aggiusta man mano fino ad avere uno stato ammissibile; questo approccio è detto **ricerca locale**.

6.3.1 Min Conflicts

Quest'algoritmo è basato su ricerca locale:

1. **Start**: si parte inizialmente con uno stato le cui variabili hanno assegnamenti tutti casuali.

2. **Step**: fintanto che non si è raggiunto uno stato ammissible, si trasforma l'assegnamento Γ_1 in Γ_2 tale che viola un numero di vincoli minore rispetto all'assegnamento precedente (ossia cerchiamo la soluzione che è **localmente migliore**).

Teoricamente quest'algoritmo è pessimo:

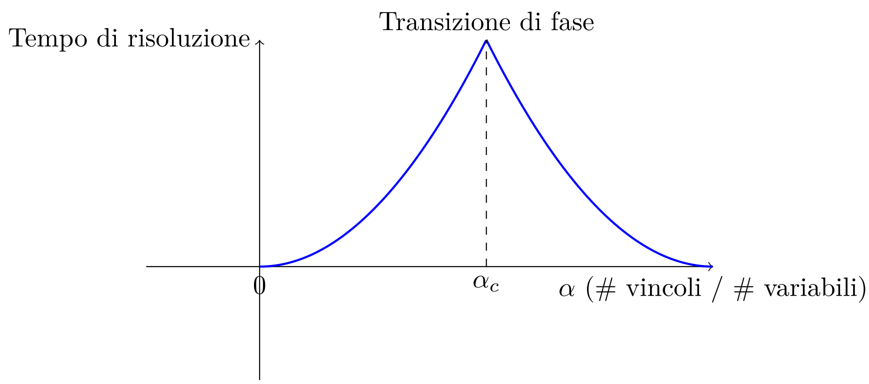
- Non è un algoritmo **Anytime**

- Non garantisce di trovare una soluzione ottima **tempo finito (anche esponenziale)**.

Eppure, empiricamente, è **molto efficiente**: riesce a risolvere il problema delle n regine in **qualche decina di iterazioni**. Possiamo dire in generale che:

- **Minconflicts** funziona bene nelle istanze di problema con **molte variabili e pochi vincoli**
- **Backtracking Search** funziona bene nelle istanze di problema con **molti vincoli e poche variabili**

Questo perchè, a differenza di molte altre classi di problemi, i problemi CSP sono fortemente influenzati non direttamente dal numero di variabili o di valori ma dal loro rapporto. Esiste infatti un grafico che riesce a mettere in relazione la difficoltà di un problema di CSP con il rapporto di variabili/vincoli, ed emerge che esiste un punto critico in cui la difficolta cresce vertiginosamente.



7 Markov Decision Processes

Tutti i problemi finora affrontati ricadevano sempre entro la famiglia dell**inferenza**. D'ora in poi tratteremo l'altro ramo dell'IA, ossia l'**apprendimento**. La prima classe di problemi che affronteremo in questo ambito sono i **Markov Decision Processes**.

Descrizione Generale.

- Gli agenti hanno come obiettivo il raggiungimento di uno **stato terminale** (può essere unico, multiplo o assente se l'agente dev'essere sempre attivo)
- L'ambiente è **Stocastico** e **Single-Agent**
- Esiste un **Orizzonte Temporale** entro cui agire

Formalizzazioni.

- **Insieme degli Stati**: $S : \{s_1, s_2, \dots\}$

- **Stato Iniziale**: $s_i \in S$
- **Stato Terminale**: $s_T \in S$
- **Azioni possibili per lo stato s**: $A(s)$
- **Modello di Transizione Stocastica**: dato $s \in S, a \in A(s), s' \in S$ con la transizione da s a s' con azione a, il modello di transizione f indica la probabilità di finire proprio sullo stato s', ossia:

$$f(s, a, s') = \mathbb{P}(s'|s, a) \tag{19}$$

- **Reward**: ad ogni transizione l'agente riceve un reward (il corrispondente di costo negli algoritmi di search) che dev'essere **additivo**
- **Orizzone H**: ossia le epoche di decisione per un MDP; oltre quest'epoca, tutto non viene più considerato. Se $H = \infty$ allora il processo termina se e solo se l'agente raggiunge uno stato terminale

7.1 Markovianità

Dato un processo stocastico che genera una **sequenza aleatoria** $s_1, s_2, \ldots, s_n$ (aleatoria perchè l'ambiente è stocastico), un processo si dice **Markoviano** se e solo se *l'esito di un'azione dipende solo dallo stato corrente e non dagli stati precedenti*, ossia:

$$\mathbb{P}(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = \mathbb{P}(s_{t+1}|s_t, a_t) \tag{20}$$

In parole povere, se un processo è markoviano, tutte le informazioni necessarie per potere scegliere la mossa migliore è già contenuta nello stato attuale stesso. Assumere che un processo sia markoviano, inoltre permette di facilitare di molto i calcoli non perdendo di generalità (questo perchè la markovianità è un'assunzione realistica in molti contesti).

7.2 Risoluzione di un MDP

In un problema MDP, quello che si cerca è la **policy** π:

$$\pi : S \to A$$

Questa è una funzione **deterministica** che mappa ad ogni stato un'azione. Il nostro obiettivo è, una volta risolto il problema, quello di rendere l'agente un **agente reflex** che sa già, ad ogni stato, le azioni da compiere. Naturalmente la difficoltà sta nel trovare una policy che **ottimizzi** un parametro (ad esempio il reward).

7.3 Stimare la bontà di una policy

Rispetto ai problemi precedenti, stimare la bontà di una strategia o un percorso risultava essere piuttosto intuitiva. Qui le cose si complicano per 2 motivi:

- Le transizioni sono stocastiche, dunque i reward sono **incerti**
- Viene imposto un **orizzonte temporale** h

La **policy ottima** deve dunque **massimizzare il valore atteso della somma dei reward**.

Definizione

- Data una sequenza $s_1, \dots, s_n$ di stati generati da π
- Dati un reward associato ad ogni stato $R_1, R_2, \dots, R_n$
- Data $U(s_1, s_2, \dots, s_n) = R_1 + R_2 + \cdots + R_n$

La policy ottima π^* deve generare il valore atteso della somma dei reward non peggiore di ogni altra policy:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}\left[U(s_0, s_1, \dots, s_n)\right] \tag{21}$$

Possiamo dunque notare come i reward assegnati influenzino notevolmente la policy:

- per $R < 0$, possiamo notare che la policy ottima induca l'agente a trovare lo stato terminale migliore nella maniera più efficiente
- per $R \ll 0$ possiamo notare che la policy ottima induca l'agente a trovare un qualsiasi stato terminale nel minor numero di passi possibili
- per $R > 0$ la policy ottima indurrà l'agente a non raggiungere mai lo stato terminale

7.4 Reward Scontati

Ponendo all'infinito l'orizzonte, l'unico modo che ha l'agente di terminare è quello di finire su uno stato terminale; dunque se la policy lo porta ad uno stato terminale, il valore atteso della somma dei reward è finita. Tuttavia, nel caso in cui ciò non accadesse, il valore atteso della somma dei reward risulta essere infinita (perchè transita in infiniti stati con altrettanti infiniti reward). Per cui dobbiamo introdurre una nuova definizione di somma di reward, ossia la **somma di reward scontati**:

$$U(s_0, s_1, \dots) = \gamma^0 R_0 + \gamma^1 R_1 + \cdots = \sum_{t=0}^{H} \gamma^t R_t \tag{22}$$

Dove $0 \leq \gamma \leq 1$ e viene detto **fattore di sconto**. La combinazione sopra descritta ha la forma di un **decadimento esponenziale** e scegliendo un apposito valore di γ possiamo bilanciare l'importanza tra l'immediato futuro e il futuro più remoto:

- per $\gamma \approx 1$ la somma dei reward si comporta esattamente come combinazione lineare con peso unitario dei reward che avevamo introdotto precedentemente, e dunque vengono preferite le scelte più vantaggiose nel futuro più lontano.
- per $\gamma \approx 0$ la somma dei reward tiene in considerazione i termini più vicini temporalmente, per cui sono più importanti le scelte che influenzano l'immediato futuro.

Si può benissimo dimostrare che per $H = +\infty$ la serie converge:

$$\sum_{t=0}^{+\infty} y^t R_t = \frac{R_{\text{MAX}}}{1-\gamma} \tag{23}$$

7.5 Policy Stazionaria

Paradossalmente, considerare problemi in cui l'orizzonte è finito, rende la risoluzione del problema molto complessa: questo perchè è necessario considerare non solo i reward per stimare la bontà di una policy, ma anche il budget di mosse rimaste. Dal momento che il **passato** conta nelle stime delle policy in problemi a orizzonti finiti, la **Markovianità si perde** poichè la policy non è più **stazionaria**, ossia, non considera solo il reward del nodo in un tempo qualsiasi. Proprio per tutti questi motivi, gli MDP che affronteremo hanno:

- $H = \infty$
- Somma di reward scontata

7.6 Ricerca Policy Ottima in DP

Arrivati a questo punto, dobbiamo calcolare in qualche modo $\pi^* : S \rightarrow A$ che masasimizza il **valore atteso della somma di reward scontati**; analizziamo ora 2 principali approcci:

Programmazione Dinamica. La Programmazione Dinamica è un approccio risolutivo che consiste nel suddividere un problema P in tanti sottoproblemi $P_1, P_2, \ldots$ e risolverli separatamente. Dalle sottosoluzioni si ricava la soluzione al problema P di partenza. I problemi che possono essere risolti tramite prog. Dinamica devono rispettare il **principio di ottimalità di Bellman** ossia:

> Nella soluzione ottima di P sono contenute le soluzioni dei sottoproblemi $P_1, P_2, \ldots$ e viceversa

Non tutti i problemi rispettano questo principio, es:

- Cammino Minimo su Grafo (rispetta)
- Cammino Massimo su Grafo pesato (non rispetta)

Si può dimostrare (per fortuna) che MDP rispetta il principio di Bellman, allora esiste una procedura risolutiva realizzabile tramite Programmazione Dinamica. Per fare ciò dobbiamo introdurre 2 nuove metriche per determinare la bontà di π^*:

- La **Value function** $V^*(s)$ di uno stato s: indica il valore atteso della somma scontata dei reward da s in avanti seguendo la policy ottima π^*. Possiamo osservare che $V^*(s)$ è indipendente dal tempo dal momento che π^* è stazionaria.
- L'**Action value function** $Q^*(s, a)$: misura il reward che ottengo eseguendo a (che non necessariamente dev'essere prescritta dalla policy ottima) e ipotizzando che dal nodo successivo in poi seguo la policy ottima

Esiste una relazione tra le 2 funzioni:

$$V^*(s) = \max_a Q^*(s, a) \tag{24}$$

Conoscere $V^*(s)$ significa dunque conoscere la policy ottima, dunque significa risolvere il problema.
Dalla definizione di Q^* possiamo dedurre un'altra relazione tra la value function e l'action function, ossia:

$$Q^*(s, a) = \gamma R(s) + V^*(s') \tag{25}$$

Ora, diamo una definizione più precisa di Q^*:

$$Q^*(s, a) = \sum_{s'} \underbrace{\mathbb{P}(s' \mid s, a)}_{\substack{\text{è la probabilità} \\ \text{di transitare su } s'}} [\underbrace{R(s, a, s')}_{\substack{\text{è il reward} \\ \text{immediato}}} + \underbrace{\gamma V^*(s')}_{\substack{\text{è lo sconto} \\ \text{del reward successivo}}}] \tag{26}$$

Da tutte queste equazioni possiamo ottenere l'equazione che le riassume tutte:

$$V^*(s) = \max_a \sum_{s'} \mathbb{P}(s'|s, a)[R(s, a, s') + \gamma V^*(s')] \tag{27}$$

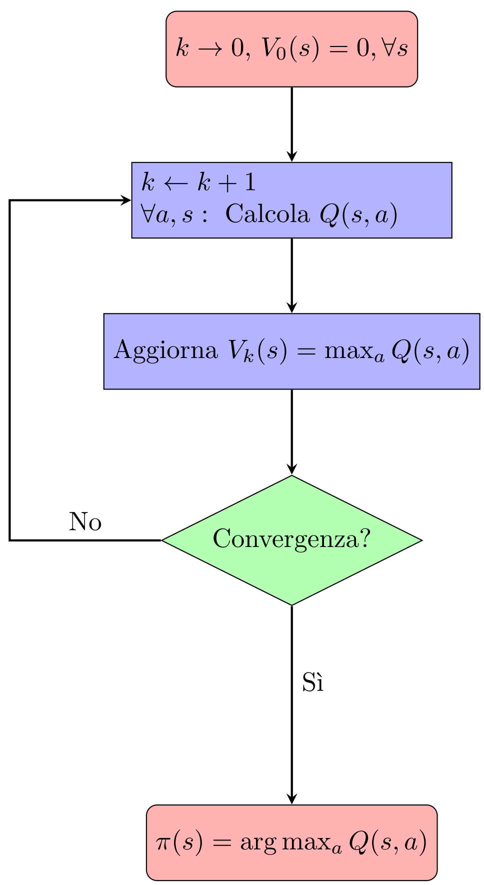
E viene detta **Equazione di Bellman**. Possiamo vedere $V^*(s')$ come il sottoproblema da risolvere mentre il sottoproblema terminale è lo stato terminale e ha valore pari a 0. Un'altra considerazione che possiamo fare è che possiamo mettere a sistema tutte le $V^*(s)$ per ogni s ed ottenere un sistema. SI può dimostrare che la soluzione a questo sistema è automaticamente la soluzione ottima, dal momento che l'equazione di bellman ammette un'unica soluzione.

7.6.1 Approccio Value Iteration

Questo primo approccio di risoluzione tramite programmazione dinamica dell MDP si basa sulla costruzione, iterazione dopo iterazione, di una value function V_k che convergerà a V^*:

- per $\mathbf{k = 0}$: $V_0(s) = 0\ \forall s$
- per $\mathbf{k > 0}$: $V_{k+1}(s) \leftarrow \max_a \underbrace{\sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')]}_{Q(s,a)}$

Quell'ultima operazione viene detta **Bellman Update** e l'operatore $\rightarrow$ è detta **Operatore di Bellman**. La risoluzione del problema avviene calcolando ad ogni iterazione la value function che **monotonicamente convergerà** a V^*. Quindi, ad una certa iterazione, se V_k soddisfa il **test di convergenza** allora abbiamo trovato la value function.



Il **Test di Convergenza** possiamo implementarlo come la condizione:

$$|\max_s\{V_{k+1}(s) - V_k(s)\} - \min_s\{V_{k+1}(s) - V_k(s)\}| \leq \varepsilon \tag{28}$$

Ossia *lo span (differenza tra val massimo e minimo tra i V_k di 2 iterazioni dev'essere meno di un epsilon)*, oppure:

$$\sum_s |V_{k+1}(s) - V_k(s)| \leq \varepsilon \tag{29}$$

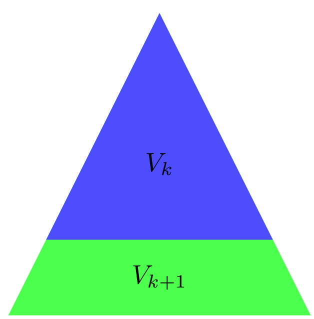
Ossia *la somma delle differenze tra i value function tra 2 iterazioni dev'essere meno di epsilon.* Sebbene quest'algoritmo sia efficace (dimostreremo dopo perchè), non sempre è efficiente, perchè V_k potrebbe **convergere lentamente**.

7.6.2 Dimostrazione di Convergenza

Osservando l'operatore di Bellman:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')] \tag{30}$$

Possiamo accorgerci di una grande somiglianza con i **giochi stocastici** e, ancora meglio, con **Expectimax** (dal momento che quell'algoritmo puntava a massimizzare il valore atteso del reward di ogni mossa). Possiamo ricondurre un MDP ad un **gioco stocastico ad 1 giocatore MAX**. Quindi possiamo dire che calcolare $V_k(s)$ corrisponde a risolvere un gioco stocastico di livello k. Tra l'albero di gioco a profondità k e $k+1$ possiamo visualizzare questa relazione:



Inoltre possiamo vedere l'albero V_k come un albero di altezza $k+1$ che ha nell'ultimo livello $(k+1)$ i reward tutti nulli, per cui V_k e V_{k+1} differiscono solo per i reward che all'ultimo livello sono (o potrebbero essere) $\neq 0$.
Sappiamo che un qualsiasi Reward R è limitato:

$$R_{min} \leq R \leq R_{max}$$

Dunque i casi possibili sono 2:

Caso Migliore Ogni reward $R = R_{max}$, per cui $V_{k+1}(s) = V_k(s) + \gamma^k R_{max}$

Caso Peggiore Ogni reward $R = R_{min}$, per cui $V_{k+1}(s) = V_k(s) + \gamma^k R_{min}$
Allora un qualsiasi reward sarà sempre tra il caso migliore e il caso peggiore:

$$V_k + \gamma^k R_{min} \leq V_{k+1} \leq V_k + \gamma^k R_{max} \tag{31}$$

Ora facendo tendere $k \to +\infty$ otteniamo:

$$V_k + 0 \leq V_{k+1} \leq V_k + 0 \tag{32}$$

Ossia, V_k non cambia dunque abbiamo trovato una soluzione all'equazione di Bellman e che sarà necessariamente ottima.

7.6.3 Policy Extraction

Una volta calcolata la value function, dovremmo semplicemente estrarre la policy a partire dai valori calcolati applicando la formula:

$$\pi^* = \arg\max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')] \tag{33}$$

Possiamo però notare quanto sia inefficiente l'algoritmo per 2 grandi motivi:

- Nel caso peggiore dovremmo considerare per ogni azione qualsiasi transizione ($O(|S|^2 \times |A|)$)
- Non è semplice trovare un ε giusto per il test di convergenza

Possiamo tuttavia notare che, già dalle prime iterazioni, avevamo individuato la policy ottima, ancora prima della value function. Per questo motivo possiamo cambiare totalmente approccio e iterare sulle policy.

7.6.4 Policy Iteration

Questo è un algoritmo che possiamo descrivere così:

- Scelgo inizialmente una policy π casuale
- **Policy Evaluation**: calcolo $V^\pi(s)$
- **Policy Extraction**: estraggo una nuova policy π' a partire da $V^\pi(s)$

Nonappena succede che $\pi = \pi'$, ho trovato la policy ottima. Dal momento che itero sulla policy, $V^\pi(s)$ è :

$$V^\pi(s) = Q(s, \pi(s)) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')] \tag{34}$$

Non essendoci il max, quella è un'equazione lineare. Mettendo a sistema le equazioni per ogni stato, otteniamo un sistema lineare da risolvere.

8 Reinforcement Learning

Potrebbe capitare che l'agente che deve risolvere l'MDP= $\langle P, R\rangle$ (dove P sono le probabilità di transizione e R sono i reward) senza conoscere P e R. In ambiente sconosciuto, dunque, l'agente non potrebbe applicare dynamic programming per risolvere il problema, o almeno all'inizio. All'inizio, un agente può solo assumere che le azioni siano stocastiche e che esistano dei reward (deterministici). In queste condizioni l'agente deve alternare fasi di **esplorazione**, in cui costruire una **stima del mondo** e questo può portare l'agente a fare decisioni con reward molto bassi, perchè conoscere il mondo ha più valore, e fasi di **Exploitation** ossia di massimizzazione dei reward (che è il suo obiettivo principale).

8.1 Passive Reinforcement Learning

In questa prima fase, presenteremo una forma più semplice dell'RL, ossia **l'RL passivo**: un agente, in un ambiente sconosciuto, conosce una policy π e deve stimarne la *bontà*, o meglio la **value function** V^*. A differenza dei casi precedenti, qui non si conoscono nè $P(s'|s,a)$ nè $R(s',a,s)$ per cui dovremo stimare. Il problema di tali approcci è che dipendono troppo dall'esperienza, per cui, con nessuna modalità di reasoning, tali agenti non sapranno rispondere ad un input se non l'hanno già incontrato in passato.

8.1.1 Adaptive Dynamic Programming

In questo primo approccio all Passive Reinforcement Learning, **Model Based**:

1. Esploro il mondo usando la policy π in un **episodio**, ossia una sequenza di mosse, e colleziono i reward e le probabilità in **dataset**

2. Costruisco una **stima** di P e di R dal Modello Del Mondo che ho raccolto nel dataset, ossia:

$$\widehat{P}(s'|s,a) = \frac{\#(s,a,s')}{\#(s,a)} \tag{35}$$

$$\widehat{R}(s,a,s') = R(s,a,s') \tag{36}$$

3. Calcolo della nuova policy ottima π^* usando l'approccio dynamic programming sul modello stimato

8.1.2 Stima Diretta

Un altro approccio usando la Passive RL **Model Free** è la **Stima Diretta**, ossia:

1. Genero un dataset dall'esplorazione

2. Per ogni stato s misuro i reward accumulati per tutto l'episodio

3. La stima sarà la **media aritmetica**:

$$\widehat{V}^{\pi}(s) = \frac{1}{k}\sum_{i=1}^{k} \mathrm{TotR}_i^s \tag{37}$$

dove TotR_i^s rappresenta il reward totale accumulato all'episodio i partendo dallo stato s

Il problema di questo approccio è che le stime vengono fatto facendo la (sbagliata) assunzione che gli stati siano indipendenti tra loro.