

# Intelligenza Artificiale I

Federico Falcone

November 1, 2025

## 1 Agenti

### 1.1 Agenti e ambienti

Un **agente** è qualsiasi cosa possa essere vista come un sistema che percepisce il suo **ambiente** attraverso i **sensori** e agisce su di esso mediante **attuatori**. Usiamo il termine **percezione** per indicare i dati che i sensori di un agente percepiscono. La **sequenza percettiva** di un agente è la storia completa di tutto ciò che esso ha percepito nella sua esistenza. In generale, *la scelta dell'azione di un agente in un qualsiasi istante può dipendere dalla conoscenza integrata in esso e dall'intera sequenza percettiva osservata fino a quel punto, ma non da qualcosa che l'agente non abbia percepito*. Il comportamento di un agente quindi è descritto dalla **funzione agente**, che descrive la corrispondenza tra una qualsiasi sequenza percettiva e una specifica azione.

Possiamo immaginare di rappresentare in forma di **tabella** la funzione agente che descrive un certo agente. La tabella è una descrizione **esterna** dell'agente. **Internamente**, la funzione agente di un agente artificiale sarà implementata da un **programma agente**, il quale è l'implementazione della funzione agente.

### 1.2 Comportarsi correttamente: il concetto di razionalità

Un **agente razionale** è un agente che fa la cosa giusta.

#### 1.2.1 Misure di prestazione

Valutiamo il comportamento di un agente considerandone le *conseguenze*. Ciò si chiama **conseguenzialismo**. Quando un agente viene inserito in un ambiente, genera una sequenza di azioni in base alle percezioni che riceve. Questa sequenza di azioni porta l'ambiente ad attraversare una sequenza di **stati**: se tale sequenza è desiderabile, significa che l'agente si è comportato bene. Questa nozione di desiderabilità è catturata da una **misura di prestazione** che valuta una sequenza di stati dell'ambiente.

#### 1.2.2 Razionalità

In un dato momento, ciò che è razionale dipende da quattro fattori:

- la misura di prestazione che definisce il criterio di successo;
- la conoscenza pregressa dell'ambiente da parte dell'agente;
- le azioni che l'agente può effettuare;
- la sequenza percettiva dell'agente fino all'istante corrente.

Questo porta alla definizione di **agente razionale**: Per ogni possibile sequenza di percezioni, un agente razionale dovrebbe scegliere un'azione che massimizzi il valore atteso della sua misura di prestazione, date le informazioni fornite dalla sequenza percettiva e da ogni ulteriore conoscenza dell'agente.

#### 1.2.3 Onniscienza, apprendimento e autonomia

Un agente **onnisciente** conosce il risultato *effettivo* delle sue azioni e può agire di conseguenza.

Intraprendere azioni mirate a modificare le percezioni future, chiamato **information gathering** è una parte importante della razionalità. Un esempio è formato dall'**esplorazione**.

Un agente razionale non si deve limitare solo a raccogliere informazioni, ma deve essere anche in grado di **apprendere** il più possibile sulla base delle proprie percezioni.

## 1.3 La natura degli ambienti

### 1.3.1 Proprietà degli ambienti operativi

Gli ambienti devono essere:

- **completamente osservabile/parzialmente osservabile**
- **Agente sigolo/multiagente**: gli ambienti multiagente possono essere **competitivi** oppure **co-operativi**.
- **deterministico/non deterministico (stocastico)**: è deterministico quando lo stato successivo dell'ambiente è completamente determinato dallo stato corrente e dall'azione eseguita dall'agente.
- **episodico/sequenziale** episodico significa che l'esperienza dell'agente è divisa in episodi atomici. In ogni episodio l'agente riceve una percezione e poi esegue una singola azione. Ogni episodio non dipende dalle azioni intraprese in quelli precedenti. In quelli sequenziali ogni decisione può influenzare tutte quelle successive.
- **statico/dinamico**: se l'ambiente può cambiare mentre un agente sta decidendo come agire è dinamico.
- **discreto/continuo**: la distinzione si applica allo stato dell'ambiente, al modo in cui è gestito il tempo, alle percezioni e azioni dell'agente.
- **noto/ignoto**: si riferisce allo stato di conoscenza dell'agente delle "leggi fisiche" dell'ambiente stesso.

## 1.4 La struttura degli agenti

Il compito dell'intelligenza artificiale è progettare il **programma agente** che implementa la funzione agente, che fa corrispondere la percezione alle azioni. Diamo per scontato che questo programma sarà eseguito da un dispositivo computazionale dotato di sensori e attuatori fisici; questa prende il nome di **architettura agente**:

*agente = architettura + programma*

### 1.4.1 Programmi agente

I programmi agente prendono come input la percezione corrente dei sensori e restituiscono un'azione agli attuatori.

### 1.4.2 Agenti reattivi semplici

Questi agenti scelgono le azioni sulla base della percezione *corrente*, ignorando tutta la storia percettiva corrente.

### 1.4.3 Agenti reattivi basati su modello

Il modo più efficace di gestire l'osservabilità parziale, per un agente, è *tener traccia della parte del mondo che non può vedere nell'istante corrente*. Questo significa che l'agente deve mantenere una sorta di **stato interno** che dipende dalla storia delle percezioni e che quindi riflette almeno una parte degli aspetti non osservabili dello stato corrente.

Aggiornare l'informazione dello stato interno al passaggio del tempo richiede che il programma agente possieda due tipi di conoscenza. Prima di tutto, deve avere informazioni sull'evoluzione del mondo nel tempo, suddivisibili approssimativamente in due parti: gli effetti delle azioni dell'agente e le modalità di evoluzione del mondo indipendentemente dall'agente. Questa conoscenza sul "funzionamento del mondo", viene chiamata **modello di transizione** del mondo.

In secondo luogo, ci servono informazioni su come lo stato del mondo si rifletta nelle percezioni dell'agente. Questo tipo di conoscenza è chiamato **modello sensoriale**.

Il modello di transizione e il modello sensoriale, insieme, consentono a un agente di tenere traccia dello stato del mondo, per quanto possibile date le limitazioni dei sensori. Un agente che utilizza tali modelli prende il nome di **agente basato su modello**.

#### 1.4.4 Agenti basati su obiettivi

Conoscere lo stato corrente dell'ambiente non sempre basta e decidere che cosa fare. Oltre che alla descrizione dello stato corrente l'agente ha bisogno di qualche tipo di informazione riguardante il suo **obiettivo** (goal), che descriva situazioni desiderabili.

Talvolta scegliere un'azione in base a un obiettivo è molto semplice, quando questo può essere raggiunto in un solo passo. Altre volte è più difficile. La **ricerca** e la **pianificazione** sono sottocampi dell'IA dedicati proprio a identificare le sequenze di azioni che permettono a un agente di raggiungere i propri obiettivi.

Benchè un agente basato su obiettivi sembri meno efficiente, d'altra parte è più **flessibile**, perchè la conoscenza che guida le sue decisioni è rappresentata esplicitamente e può essere modificata.

#### 1.4.5 Agenti basati sull'utilità

Gli obiettivi forniscono solamente una distinzione binaria tra stati "contenti" e "scontenti", laddove una misura di prestazione più generale dovrebbe permettere di confrontare stati del mondo differenti e misurare precisamente la contentezza che potrebbero portare all'agente. Per descrivere ciò utilizziamo il termine **utilità**. Una **funzione di utilità** di un agente è un'internalizzazione della misura di prestazione. Purchè la funzione di utilità interna e la misura di prestazione esterna concordino, un agente che sceglie le azioni per massimizzare l'utilità sarà razionale in base alla misura di prestazione esterna.

## 2 Risolvere i problemi con la ricerca

Quando l'azione giusta da compiere non è subito evidente, un agente uò avere la necessità di *guardare avanti*, cioè considerare una **sequenza** di azioni che formano un cammino che porterà a uno stato obiettivo. Questo tipo di agente è chiamato **agente risolutore di problemi** e il processo computazionale che effettua è la **ricerca**.

Gli agenti risolutori di problemi utilizzano rappresentazioni **atomiche** in cui gli stati del mondo sono considerati come entità prive di una struttura interna visibile agli algoritmi per la risoluzione dei problemi. Gli agenti che utilizzano rappresentazioni di stati **fattorizzate** o **strutturate** sono solitamente chiamati **agenti pianificatori**.

### 2.1 Agenti risolutori di problemi

Se l'agente non ha informazioni sufficienti sull'ambiente, ovvero se l'ambiente è **ignoto**, non può fare altro che eseguire una delle azioni scelte a caso. Con tali informazioni a disposizione, l'agente può eseguire un processo di risoluzione del problema in quattro fasi:

- **Formulazione dell'obiettivo;**
- **Formulazione del problema:** l'agente elabora una descrizione degli stati e delle azioni necessarie per raggiungere l'obiettivo, ovvero un modello astratto della parte del mondo interessata;
- **Ricerca:** prima di effettuare qualsiasi azione nel mondo reale, l'agente simula nel suo modello sequenze di azioni, continuando a cercare finchè trova una sequenza che raggiunge l'obiettivo: tale sequenza si chiama **soluzione**;
- **Esecuzione:** l'agente ora può eseguire le azioni specificate nella soluzione, una per volta.

#### 2.1.1 Problemi di ricerca e soluzioni

Un **problema** di ricerca può essere definito formalmente come segue:

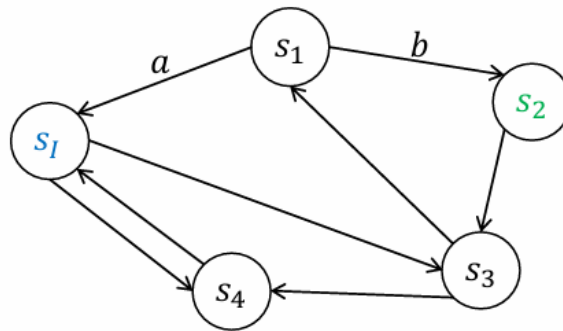
- Un insieme di possibili **stati** in cui può trovarsi l'ambiente. Lo chiamiamo **spazio degli stati**.
- Lo **Stato iniziale** in cui si trova l'agente inizialmente.
- Un insieme di uno o più **stati obiettivo**. A volte è unico, a volte è un piccolo insieme, a volte è definito da una proprietà che è soddisfatta da molti stati.
- Le **azioni** possibili dell'agente. Dato uno stato  $s$ ,  $AZIONI(s)$  restituisce un insieme finito di azioni che possono essere eseguite in  $s$ . Diciamo che ognuna di queste azioni è **applicabile** in  $s$ .

$$A(s) = \{a, b, c, \dots\}$$

- Un **modello di transizione** che descrive ciò che fa ogni azione. Dato uno stato di partenza  $s_j$  e un'azione  $a \in A(s_j)$ , indica uno stato di arrivo  $f(s_j, a)$ : rappresenta la conseguenza dello svolgere l'azione  $a$  nello stato  $s$ .
- Una **funzione di costo dell'azione**, denotata da  $c(s_j, a, f(s_j, a))$ , restituisce il costo numerico di applicare l'azione  $a$  nello stato  $s_j$  per raggiungere lo stato  $s'$ .

Una sequenza di azioni forma un **cammino**; una **soluzione** è un cammino che porta dallo stato iniziale a uno stato obiettivo. Assumiamo che i costi delle azioni siano additivi. Una **soluzione ottima** è quella che ha il costo minimo.

Lo spazio degli stati può essere rappresentato come un **grafo** in cui i vertici rappresentano gli stati e i collegamenti orientati tra di essi rappresentano le azioni.



### 2.1.2 La formulazione dei problemi

La formulazione del problema è un **modello**, ovvero una descrizione matematica astratta.

Il processo di rimozione dei dettagli da una rappresentazione prende il nome di **astrazione**. Per una buona formulazione del problema serve il giusto livello di dettaglio.

Come faccio a specificare un problema di search?

- **Approccio esaustivo/esplicito**: fornire il grafo degli stati in modo completo specificando tutte le transizioni possibili. Il più delle volte questa non è un'opzione percorribile a causa della natura combinatoria dello spazio degli stati.
- **Approccio implicito**: possiamo specificare lo stato iniziale e la funzione di transizione in una forma **compatta**. Il grafo degli stati si "svela" man mano che le azioni vengono valutate.

Ci serve una procedura efficiente per controllare se uno stato generato è il goal: **goal check**.

## 2.2 Algoritmi di ricerca

Un **algoritmo di ricerca** riceve in input un problema di ricerca e restituisce una soluzione o un'indicazione di fallimento. Consideriamo algoritmi che sovrappongono un **albero di ricerca** al grafo dello spazio degli stati, formando vari cammini a partire dallo stato iniziale e cercando di trovarne uno che raggiunga uno stato obiettivo. Ciascun **nodo** nell'albero di ricerca corrisponde a uno stato nello spazio degli stati e i rami dell'albero di ricerca corrispondono ad azioni. La radice dell'albero corrisponde allo stato iniziale del problema.

È importante comprendere la distinzione tra spazio degli stati e albero di ricerca. Lo spazio degli stati descrive l'insieme degli stati nel mondo e le azioni che consentono le transizioni da uno stato a un altro. L'albero di ricerca descrive i cammini tra questi stati per raggiungere l'obiettivo. Nell'albero di ricerca possono esserci più cammini per raggiungere qualsiasi stato, ma per ogni nodo dell'albero c'è un cammino univoco per tornare alla radice.

### 2.2.1 Obiettivi della ricerca

Un algoritmo di ricerca **esplora il grafo degli stati fin quando non trova la soluzione desiderata**. Nella versione di **fattibilità** quando viene visitato un nodo di goal viene restituito il percorso che ha portato a quel nodo. Nella versione di **ottimizzazione** quando viene visitato un nodo di goal, se qualsiasi altro possibile percorso per quel nodo ha un costo maggiore, viene restituito il percorso che ha portato a quel nodo.

Non basta visitare un nodo di goal, l'algoritmo deve ricostruire il percorso che ha seguito per arrivarci: deve tenere traccia della sua ricerca. Tale traccia può essere mappata su un sotto-grafo di  $G$ , detto **albero** di ricerca.

### 2.2.2 Come si valuta un algoritmo di ricerca?

Possiamo valutare un algoritmo di ricerca lungo diverse dimensioni:

- Correttezza
- Completezza
- Complessità in termini di spazio
- Complessità in termini di tempo

#### 2.2.3 Correttezza

Garanzia che se l'algoritmo restituisce una soluzione, questa è conforme alle caratteristiche specificate nella formulazione del problema.

L'algoritmo dice che c'è una soluzione. È vero? E la soluzione che l'algoritmo ha calcolato conduce veramente ad un goal?

#### 2.2.4 Completezza

Garanzia che se una soluzione esiste allora l'algoritmo la trova **sempre**.

*L'algoritmo termina sempre? E se dice che non ci sono soluzioni, è vero?*

La completezza di solito si dimostra facendo vedere che la ricerca nello spazio degli stati + in grado di visitare tutti gli stati possibili, ap atto di concedere un tempo arbitrariamente lungo.

**Se lo spazio degli stati è infinito?** Possiamo chiederci se la ricerca è **sistematica**:

- se la risposta è *sì* l'algoritmo deve terminare;
- se la risposta è *no*, va bene se non termina ma tutti gli stati raggiungibili devono essere visitati nel limite: man mano che il tempo va all'infinito, tutti gli stati vengono visitati.

#### 2.2.5 Complessità spaziale e temporale

**Complessità spaziale:** come cresce la quantità di memoria richiesta dall'algoritmo di ricerca in funzione della dimensione del problema (caso peggiore)?

**Complessità temporale:** come cresce il tempo richiesto (numero di operazioni) dell'algoritmo di ricerca in funzione della dimensione del problema (caso peggiore)?

**Trend asintotico:**

- La complessità viene descritta con una funzione  $f(n)$ .
- Ai fini dell'analisi risulta conveniente adottare quella che si chiama la notazione "O-grande".
- $n$  di solito codifica la dimensione di una istanza del problema.

## 3 Search: UCS e A\*

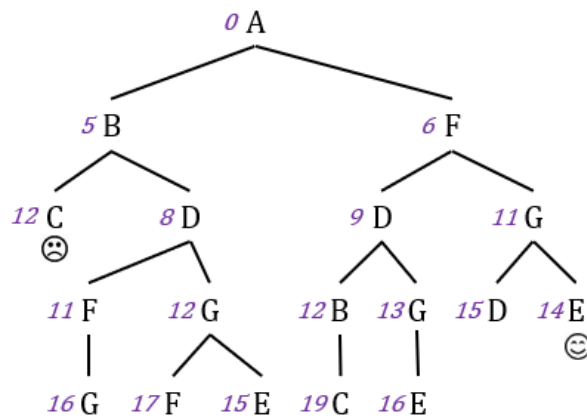
### 3.1 Uniform Cost Search (UCS)

Nell'albero di ricerca, teniamo traccia del nostro accumulato sul percorso dal nodo iniziale a ogni nodo  $V$ :  $g(V)$ . Non consideriamo EQL.

L'UCS consiste nella selezione (espansione) del nodo con  $g$  minore ancora da esplorare (sulla frontiera).

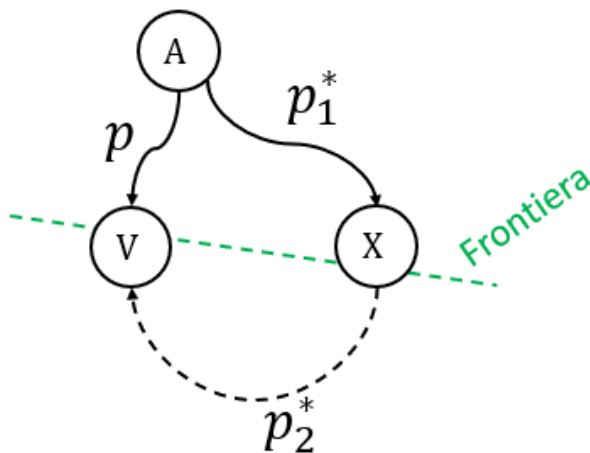
Goal check: se il **nodo selezionato per l'espansione** è un goal, mi fermo e restituisco la soluzione.

Ci si pone la domanda: abbiamo trovato il percorso ottimale per l'obiettivo? Per dare una risposta, possiamo ispezionare il grafico.



Come si nota dal grafico, sì, trova la soluzione ottimale.  
Anzi possiamo affermare che: **ogni volta che UCS seleziona per la prima volta un nodo per l'espansione, il percorso che, sull'albero di ricerca, porta a quel nodo ha un costo minimo.**

### 3.1.1 Ottimalità di UCS



**Overload della notazione:** estendo la notazione di  $g$  rendendola applicabile anche ai path  $g(XBYB \dots \beta Z) = g(Z)$ .

**Ipotesi:**

1. UCS seleziona per la prima volta dalla frontiera un nodo  $V$  che è stato generato attraverso un percorso  $p$ ;
2. il percorso  $p$  non è il percorso ottimo per raggiungere  $V$ :  $p^* \neq p$ ;

Dato il secondo punto e la **separation property** della frontiera, sappiamo che deve esistere un nodo  $X$  sulla frontiera, generato attraverso un cammino  $p_1^* + p_2^*$ .

$p^*$  è il **path ottimo**, quindi  $g(p_1^*) < g(p_1^*) + \Delta p_2^* < g(p)$ .

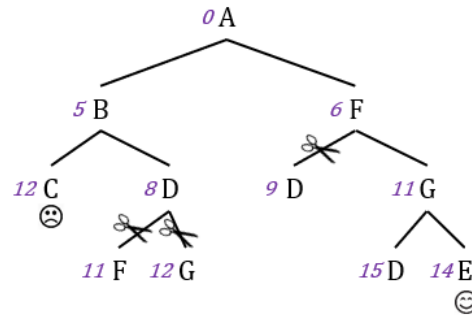
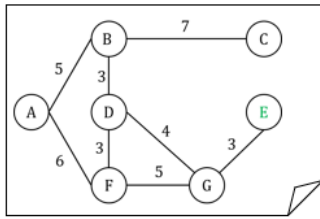
I costi sono tutti positivi, quindi  $g(p_1^*) < g(p_1^*) + \Delta p_2^* < g(p) \Rightarrow g(p_1^*) < g(p)$ .

Questo implica che  $g(X) < g(V)$ , **la prima ipotesi è violata.**

Se quando selezioniamo per la prima volta un nodo scopriamo il percorso ottimo, non c'è motivo di selezionare lo stesso nodo una seconda volta, introduciamo quindi la lista dei **nodi espansi: EXL**.

Ogni volta che selezioniamo un nodo per l'estensione:

- Se il nodo è già in EXL, lo **scartiamo**.
- Altrimenti lo estendiamo e lo inseriamo in EXL.



$EXL = \{\emptyset\}$

$EXL = \{A\}$

$EXL = \{A, B\}$

$EXL = \{A, B, F\}$

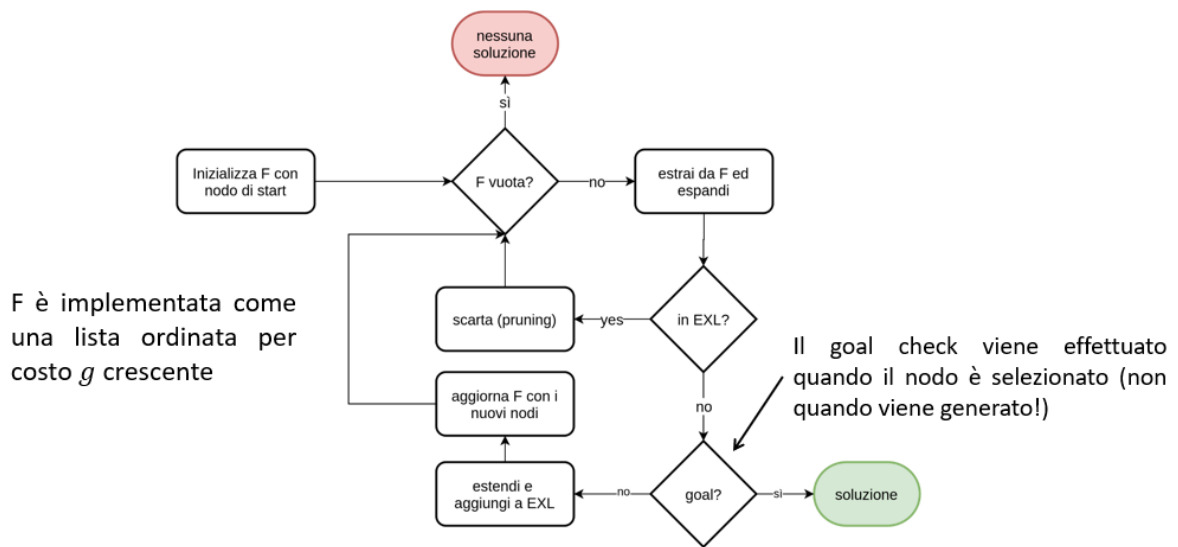
$EXL = \{A, B, F, D\}$

$EXL = \{A, B, F, D, G\}$

$EXL = \{A, B, F, D, G, C\}$

$EXL = \{A, B, F, D, G, C, E\}$

### 3.1.2 Implementazione



## 3.2 Ricerca informata

### 3.2.1 Ricerca non informata e non informata

Gli algoritmi di ricerca decidono quale nodo espandere attraverso delle regole che applicano in funzione della conoscenza del problema e del processo di ricerca svolto fino al tempo presente.

Una ricerca è **non informata** se utilizza solo la conoscenza del problema che è specificata nella sua definizione.

Una ricerca è **informata** va oltre alla definizione del problema sfruttando della conoscenza aggiuntiva: ciò che quel grafo, quelle connessioni e quei costi rappresentano nel mondo reale, oltre il formalismo agnostico che li esprime.

Dato un generico stato  $S$ , usando questa conoscenza, un algoritmo informato **stima** la bontà di  $S$  attraverso una funzione  $f(S)$  e guida la ricerca usando  $f$ .

Approccio **best-first**: espandere prima gli stati che hanno una  $f$  migliore.

Esistono diversi algoritmi di ricerca best-first, la differenza la fa il **come**  $f$  è definita.

### 3.3 A\*

La forma più diffusa di ricerca best-first è la **ricerca A\***. La valutazione dei nodi viene eseguita combinando  $g(n)$ , il costo per raggiungere il nodo, e  $h(n)$  (**euristica**), il costo per andare da lì all'obiettivo:

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

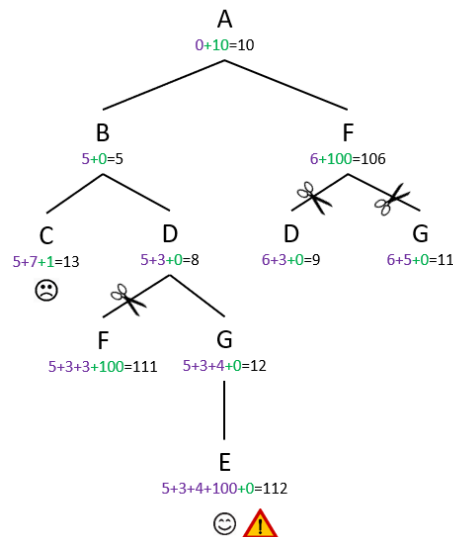
Dal momento che  $g(n)$  fornisce il costo di cammino dal nodo iniziale al nodo  $n$ , e  $h(n)$  rappresenta il costo stimato del cammino più conveniente da  $n$  all'obiettivo, risulta  $f(n)$  = costo stimato della soluzione più

conveniente che passa per  $n$ . Se stiamo cercando di trovare la soluzione meno costosa, quindi una cosa ragionevole è provare per primo il nodo col valore più basso di  $g(n)$  e  $h(n)$ . In effetti rilta che questa strategia è molto più che ragionevole, a patto che la funzione auristica  $h(n)$  soddisfi certe condizioni, la ricerca  $A^*$  è sia completa che ottima.

### 3.3.1 Ammisibilità di $A^*$

$A^*$  è ottima se  $h(n)$  è una **euristica ammissibile**, ovvero se  $h(n)$  *non sopravvaluta* mai il costo reale di una soluzione che passa per il nodo  $n$ .

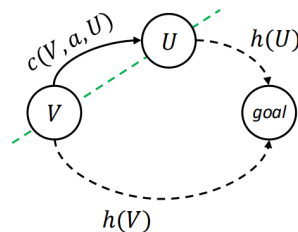
### 3.3.2 $A^*$ con EXL



Grazie all'ammissibilità manteniamo la stessa proprietà di ottimalità che abbiamo dimostrato con UCS: se non sovrastimiamo non possiamo scartare il path ottimo.

Se lavoriamo con EXL l'ammissibilità **non garantisce** l'ottimalità. Per risolvere ciò bisogna aggiungere all'euristica una proprietà più stringente: la **consistenza**.

Siano  $V$  e  $U$  due stati connessi da una azione  $a$ . Una euristica  $h$  è **consistente** se per ogni possibile coppia di  $V$  e  $U$  vale la seguente disuguaglianza:  $h(V) \leq c(V, a, U) + h(U)$ . (disuguaglianza triangolare)  
Disuguaglianza triangolare Afferma che ogni lato del triangolo non può mai essere più lungo della somma degli altri due.



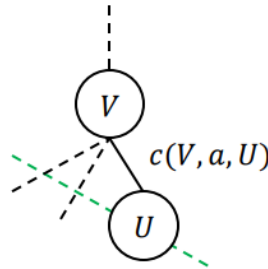
### 3.3.3 Ottimalità di $A^*$

Se assumiamo che  $h$  sia consistente, possiamo costruire una dimostrazione per l'ottimalità di  $A^*$ . Cominciamo con il derivare una proprietà di  $f$ :

1. Consideriamo due stati  $V$  e  $U$  connessi da un'azione  $a$  che l'algoritmo ha generato uno dopo l'altro sull'albero di ricerca.
2. Per definizione  $f(U) = g(U) + h(U)$ .
3. Per definizione di  $g$  e nostra azzunzione in 1,  $g(U) = g(V) + c(V, a, U)$ .
4. Sostituendo in 2:  $f(U) = g(V) + c(V, a, U) + h(U)$ .



5. Per la proprietà di **consistenza**  $c(V, a, U) + h(U) \geq h(V)$ .
6. Sommando  $g(V)$  a entrambi i termini:  $g(V) + c(V, a, U) + h(U) \geq g(V) + h(V)$ .
7.  $f(U) \geq f(V) \Rightarrow$  lungo ogni percorso nell'albero di ricerca la  $f$  è monotono non decrescente.

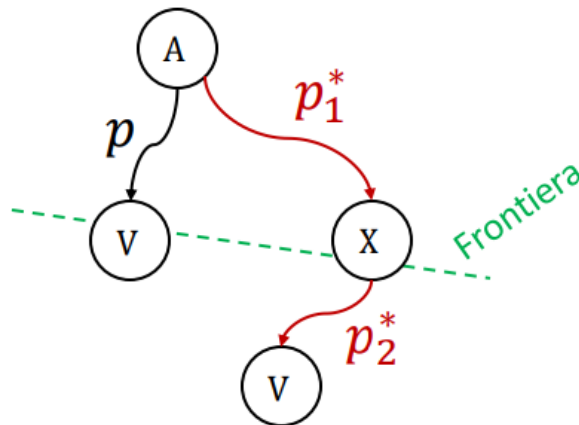


Estendo la notazione di  $f$  esplicitando il path su cui si calcola il costo  $g$ :  $f(p, n) = g(p) + h(n)$

**Ipotesi:**

1.  $A^*$  seleziona per la prima volta dalla frontiera un nodo  $V$  che è stato generato attraverso un percorso  $p$ .
2. il percorso  $p$  non è il percorso ottimo per raggiungere  $V$ :  $p^* \neq p$

Dato 2 e la **separation property** della frontiera, sappiamo che deve esistere un nodo  $X$  sulla frontiera che si trova sul cammino ottimo  $p^* = p_1^* + p_2^*$  verso  $V$ ;



1.  $g(p) > g(p^*)$  perché sia  $p^*$  che  $p$  sono path che portano a  $V$ , ma il primo è ottimo mentre il secondo no;
2.  $f(p^*, V) \geq f(p_1^*, X)$  dalla consistenza di  $h$  ( $f$  monotona non decrescente);
3.  $g(p) + h(V) > g(p^*, V) + h(V)$  sommando lo stesso termine ai membri di 1;
4.  $f(p, V) > f(p^*, V) \geq f(p_1^*, X)$  mettendo insieme 3, 2 e la definizione di  $f$ ;
5.  $f(p, V) > f(p_1^*, X)$  quindi  $V$  non può essere scelto prima di  $X$ , l'ipotesi 1 è violata

### 3.4 Progettare un'eutistica

Per capire come trovare un metodo per costruire buone euristiche, cerchiamo prima di capire come si può valutare una euristica: ci sono euristiche che sono migliori di altre? E come si stabilisce?

Possiamo immaginare la nostra  $h$  come un punto in un intervallo limitato:



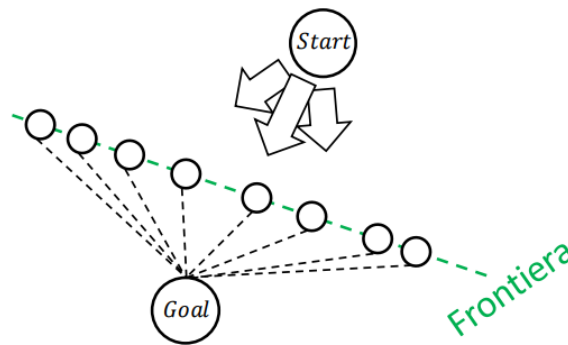
## 4 Bounded Suboptimal Search e varianti di A\*

### 4.1 Limiti di A\*

L'uso di euristiche ammissibili costituisce un vantaggio perchè garantisce l'ottimalità. Può essere anche essere un grosso limite per mancanza di flessibilità.

Consideriamo, ad esempio, una situazione in cui in frontiera non ci sono un gran numero di nodi:

- ciascun nodo rappresenta un path verso il goal:
- ciascun path ha più o meno lo stesso costo, uno solo ha il costo minimo.



A\* spenderà un sacco di tempo a distinguere il path ottimo tra tutti questi path che sostanzialmente sono equivalenti. Non si accontenta di un path sub-ottimale, anche quando il suo costo dista di pochissimo dall'ottimo.

Come possiamo equipaggiare A\* con un po' di flessibilità?.

### 4.2 Focal Search

Supponiamo di avere a disposizione una seconda euristica, **non ammissibile**, che chiamiamo  $\hat{h}_F$ .

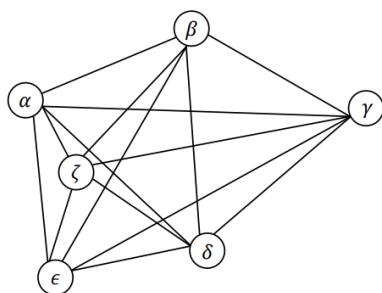
$\hat{h}_F(n)$  è una stima del **costo computazionale** necessario per completare la ricerca del percorso ottimo verso il goal.

Ricordando che, in generale, in un problema di search le azioni **non** hanno costi uniformi, possiamo dire che dato un nodo  $n$ :

- $h(n)$  stima ottimisticamente il costo rimanente da spendere per arrivare al goal;
- $\hat{h}_F(n)$  stima il numero di azioni ancora da compiere per arrivare al goal, ovvero la lunghezza della parte di soluzione ancora da costruire.

Più è lunga la parte di soluzione da costruire, più lavoro dovrò fare per costruirla! Questa valutazione non è legata al costo rimanente.

**Esempio: Traveling Salesman Problem**

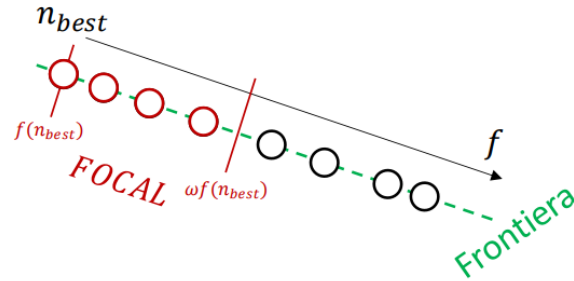


- Nodi  $\rightarrow$  città
- Collegamenti  $\rightarrow$  strade, ciascuna strada ha un costo diverso e positivo (grafo completo)
- Soluzione: ciclo Hamiltoniano di costo minimo ( $\mathcal{NP}$ -Hard)
- **Problema di search (?)**
  - Stato  $\rightarrow$  percorso parziale (es.  $n = (\alpha \rightarrow \zeta \rightarrow \delta)$ )
  - Costo  $\rightarrow$  somma dei collegamenti percorsi
  - Azioni  $\rightarrow$  spostamenti verso una città non precedentemente visitata
  - Se  $x$  è la città non ancora visitata più lontana dall'ultima città nel path ( $\delta$ )  
$$h(n) = \text{dist}(\delta, x) + \text{dist}(x, \alpha)$$
  - $\hat{h}_F(n)$  numero di città rimanenti da visitare

- $F$ : la lista di nodi in frontiera, quella usata da A\*.
- $n_{best} = \text{argmin}_{n \in F} f(n)$

- $\omega \geq 1$ , un parametro che scegliamo noi
- $FOCAL \subseteq F$  sotto-lista definita così:

$$FOCAL = \{n \in F | f(n) \leq \omega f(n_{best})\}$$



**Regola di espansione:** scegliere la  $FOCAL$  il nodo che minimizza  $\hat{h}_F, n_{next} = \operatorname{argmin}_{n \in FOCAL} \hat{h}_F(n)$ . Tutto il resto resta uguale ad  $A^*$ .

**Perdiamo l'ottimalità!** Quando l'algoritmo seleziona per l'espansione un nodo di goal  $e$  potrebbe non aver trovato il percorso ottimo (potremmo non aver scelto il nodo con  $f$  minima in frontiera e quindi potremmo aver saltato una linea di costo!)

**La persita di ottimalità non è arbitrariamente grande**, ma controllabile con il parametro  $\omega$ .

#### 4.2.1 Focal Search è una BSS

- $e$ : il nodo di goal selezionato da FOCAL (fa terminare la ricerca)
- $n^*$ : il nodo che conduceva al goal su path ottimo, è in frontiera ma (assumiamo) non sia stato scelto
- $OPT$  è il costo della soluzione ottima
- $f(n^*) \leq OPT$  perchè  $f$  usa un'euristica ammissibile
- $f(n_{best}) \leq f(n^*)$  per definizione di  $n_{best}$
- $f(e) \leq \omega f(n_{best})$  per costruzione dell'algoritmo Focal Search
- Mettendo insieme le disuguaglianze di cui sopra otteniamo:

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

Il costo della soluzione trovata da Focal Search può essere al più  $\omega$  volte peggiore dell'ottimo  
BSS: Bounded Subotimal Search

### 4.3 Problema del trashing

La funzione  $f$  può solo crescere. Questo implica, in  $FOCAL$  search due effetti:

- in  $FRONTIER$  il nodo  $n_{best}$  tenderà a stare a profondità bassa e quindi ad avere un valore alto di  $\hat{h}_F$  (resta in FOCAL a lungo senza esser scelto)
- i nodi generati da una espansione tenderanno a non entrare in  $FOCAL$  per molto tempo (fino a quando  $n_{best}$  non viene rimosso).

Effetto "fisarmonica":  $FOCAL$  viene svuotata, nel momento in cui viene rimosso  $N_{best}$  viene riempita di nuovo, poi ancora svuotata, ...

Causa del problema: Focal Search usa  $f$  per riempire  $FOCAL$ , questo sembrerebbe l'unico modo per garantire il bound della subottimalità, **ma non lo è**.

Idea: usare  $f$  solo per garantire il bound e usare, in ordine,  $\hat{h}_F$ , e una euristica "aggressiva"  $\hat{h}$  per scegliere il nodo da espandere, EES (Explicit ESTimation Search, 2011)

## 4.4 Altri algoritmi di ricerca

I vari concetti che definiscono gli algoritmi di ricerca visti in precedenza, possono essere visti come il loro building blocks.

Nulla vieta di combinarli in modi alternativi per definire nuovi algoritmi di ricerca.

DFS, ma spareggio usando $h$	<b>Hill Climbing</b>
BFS, ma una volta che ho espanso tutti i nodi al livello $k$ , mantengo nel livello $k + 1$ solo i $w$ nodi migliori secondo $h$	<b>Beam</b>
DFS limitando la profondità massima a 1, poi a 2, poi a 3 ...	<b>Iterative Deepening</b>
Iterative Deepening, ma anziché limitare la profondità uso $f$	<b>IDA*</b>
A* dal nodo di goal cercando una strada verso quello iniziale	<b>D*</b>
Come A* ma usando $f(n) = h(n)$	<b>Greedy Search</b>

## 5 Giochi: nozioni di base, Minimax e alfa-beta pruning

### 5.1 Adversarial Search

L'adversarial search è una **ricerca con avversari** creata in un **ambiente competitivo** in cui vi sono due o più agenti con obiettivi in conflitto.

Consiste in un ambiente multi-agente in cui gli obiettivi degli altri obiettivi non sono necessariamente concordanti con quelli del nostro agente di riferimento. In questi casi si parla di un sistema multi-agente e l'interazione tra di esse può essere chiamata **gioco**.

### 5.2 Giochi

Un gioco è un problema di decisione **interattivo**. Ogni agente ha le proprie preferenze (utilità) individuali sugli stati del mondo. Le azioni di un agente influenzano l'ambiente e quindi, indirettamente, anche gli altri agenti.

Nel cercare una sequenza di azioni verso lo stato desiderato, l'agente da noi controllato deve considerare le **strategie** degli altri decisori.

Esistono diversi tipi di giochi:

- 2 o  $n$  giocatori;
- Agenti **razionali**,  $\epsilon$ -razionali;
- Struttura sequenziale: **turni**, azioni simultanee, ...;
- **Deterministico** o stocastico: tutte le azioni hanno effetti prevedibili?
- Struttura dei payoff: **somma costante** o somma generica;
- **Informazione completa**/incompleta: i giocatori conoscono/non conoscono gli obiettivi e le azioni disponibili agli altri agenti;
- **Informazione perfetta**/imperfetta: i giocatori sono/non sono informati di tutto quello che è successo ad ogni punto del gioco.

Lo studio del comportamento individuale dei singoli agenti si chiama **teoria dei giochi competitivi**.

Lo studio delle dinamiche nella formazione di coalizioni si chiama **teoria dei giochi cooperativa**.

I giochi più studiati in IA sono quelli che gli studiosi di teoria dei giochi chiamano deterministici, a due giocatori, a turni, con **informazione perfetta** (completamente osservabile), **a somma zero** (ciò che a vantaggio di un giocatore danneggia l'altro).

Consideriamo giochi formalizzati così:

- **Stati:**  $S = \{s_1, s_2, \dots\}$  insieme o **spazio** degli stati, dove  $s_k \in S$  è lo stato **iniziali** (la situazione di partenza in cui trovano gli agenti e ambiente).
- **Giocatori e azioni possibili:** insieme di agenti  $I = \{i_1, i_2\}$ , insieme di azioni disponibili  $A = \{a_1, a_2, a_3, \dots\}$ .
- **Turni e azioni legali:** dato  $s_k \in S$ ,  $I(s_k) \in I$  è il giocatore che hanno diritto di compiere un'azione (turno) e  $A(s_k)$  è l'insieme di azioni che può intraprendere.
- **Modello di transizione:** dato  $s_k \in S$  e  $a \in A(I(s_k))$ ,  $f(s_k, a) \in S$  indica il prossimo stato del gioco e cioè il risultato della mossa  $a$ .
- **Stati terminali:** dato uno stato  $s_k$ ,  $T(s_k) = 1$  se  $s_k$  è uno stato terminale, dove il gioco termina e un payoff viene inviato ad ogni giocatore; vale 0 altrimenti.
- **utilità:** dato uno stato terminare  $s_t$ ,  $u_i(s_t)$  è il payoff che il giocatore  $i$  riceve in quello stato.

Tutti questi punti prendono il nome di **meccanismo**.

Questa formalizzazione implica: 2 giocatori, struttura sequenziale a turni alternati e azioni deterministiche.

I giochi hanno anche altre caratteristiche:

- **Informazione completa:** entrambi i giocatori hanno accesso al meccanismo, conoscono azioni possibili e utilità del proprio avversario.
- **Informazione perfetta:** dato uno stato corrente del gioco  $s_j$  ogni giocatore ha accesso allo stato e alla sequenza di azioni che, a partire dallo stato iniziale, ha portato dino ad  $s_k$ .
- **A "somma zero":** significa che in ogni stato terminale  $s_t$  vale  $u_1(s_t) + U_2(s_t) = 0$ .
  - schema di competizione pura: il guadagno di un agente corrisponde ad una egual perdita dell'avversario;
  - da un punto di vista matematico **è equivalente ad un gioco a somma costante**  $u_1(s_t) + u_2(s_t) = C$
  - Interpretazione: ogni giocatore paga una quota di  $\frac{C}{2}$  per entrare nel gioco, poi  $C$  viene ridistribuito a seconda dell'esito;
  - Il gioco a somma costante  $C$  può essere sempre trasformato in un gioco a somma zero rinormalizzando i payoff con una trasformazione affine;
  - In un gioco a somma zero l'utilità del secondo giocatore è implicita, si indica solo con  $u_1$ , come se ci fosse un trasferimento di utilità da  $i_1$  a  $i_2$ . **Esempio:**  $u_1 = 5 \rightarrow i_2$  paga 5 a  $i_1$ ,  $u_1 = -5 \rightarrow i_1$  paga 5 a  $i_2$ .
  - Entrambi **massimizzano la loro utilità**, ma per  $i_2$  (quello di cui non esplicitiamo l'utilità) vale:  $\max\{u_2\} = \max\{-u_1\} = -\min\{u_1\}$ , quindi possiamo dire che cercherà di minimizzare  $u_1$  così da ricevere il miglior posto possibile  $-u_1$ .

Il meccanismo descrive le "regole" del gioco, ma rappresenta solo una parte della sua descrizione. L'altra parte è data dalle **strategie**.

La strategia di un giocatore  $i$  specifica il comportamento di quel giocatore in ogni possibile stato del gioco, dato  $s_k$ , tale per cui  $I(s_k) = i$ ,  $\sigma_i(s_k)$  è la strategia di  $i$  nello stato  $s_k$ . Se la strategia coincide con una singola azione, quella da giocare in quel caso si chiama **strategia pura**  $\sigma_i : \{s_k | I(s_k) = i\} \rightarrow A(s_k)$ . Se invece la strategia è una distribuzione di probabilità su più azioni, si chiama **strategia mista**  $\sigma_i : \{s_k | I(s_k) = i\} \rightarrow \Pi(A(s_k))$  (dove  $\Pi(Q)$  è lo spazio di distribuzioni di probabilità sugli elementi di  $Q$ ).

Ci concentreremo principalmente sulle **strategie pure**

**Strategia ottima:** quella strategia tale per cui ogni strategia diversa non introduce miglioramenti contro un avversario **infallibile**.

Per ricolmare un gioco bisogna calcolare la strategia ottima.

### 5.2.1 Albero di gioco

Il meccanismo dà origine all'**albero di gioco**. Esso è un **albero di ricerca** che si ottiene sviluppando tutte le possibili sequenze di mosse alternate, in cui ogni nodo è un **nodo di decisione**: uno dei due giocatori deve fare la sua mossa.

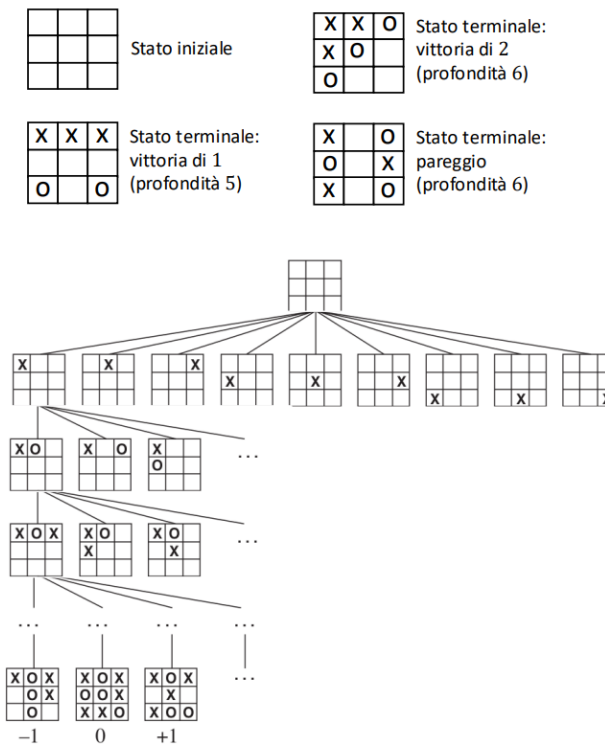
Ogni branch rappresenta un gioco (partita). Dallo stato iniziale fino a quello terminale dove il gioco termina e i payoff sono distribuiti, ogni foglia è uno stato terminale su cui indichiamo il payoff del giocatore  $i_1$  (quello del giocatore  $i_1$  è uguale all'opposto). **Esempio: il gioco del tris, Giocatore  $i_1$ : X, giocatore  $i_2$ : O. Il branching massimo è  $b=9$  e profondità massima  $d=9$ . Gli stati sono:**

- $\leq 3^9 = 19683$
- rimuovendo gli stati illegali ne restano 5478, al netto delle simmetrie sussistono di fatto 5478, al netto delle simmetrie sussistono di fatto 765 diverse situazioni strategiche in cui decide.

Numero di nodi dell'albero di gioco:  $\leq 1 + 9 + (9 * 8) + (9 * 8 * 7) + (9 * 8 * 7 * 6) + \dots = 986410$  se non facciamo ottimizzazioni è una stima ragionevole.

Numero di giochi (ovvero i branch dell'albero o i suoi nodi terminali):

- $9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 9! = 362880$
- numero esatto 255168, al netto delle simmetrie 26830.



### 5.3 Minimax

Nei problemi di search l'albero di ricerca rappresentava/supportava il processo di inferenza per trovare una strada o la strada ottima per il goal. L'albero di gioco ha una funzione analoga: rappresenta/supporta il ragionamento strategico.

Per adesso assumiamo che trovare la strategia ottima sia un problema di search che tiene conto dell'avversario: **adversarial search**. L'approccio di base con cui si risolve la classe di giochi che stiamo considerando è dato dall'algoritmo **minimax**, sostanzialmente una DFS sull'albero di gioco con un **ritorno all'indietro** dei valori di utilità.

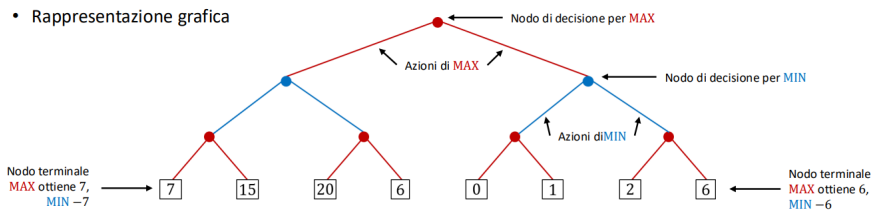
È un approccio esautistico e inefficiente che possiamo adottare solo per giochi di dimensioni limitate, ma è un approccio esatto, corretto e completo; sta alla base dei metodi più sofisticati.

Introduciamo la notazione per semplificare la descrizione dell'algoritmo:

$I = \{MAX, MIN\}$ , il giocatore  $i_1$  si chiama **MAX** e il suo obiettivo nel gioco è ottenere un valore di utilità il più alto possibile. Il giocatore  $i_2$  si chiama **MIN** e il suo obiettivo è far sì che l'utilità di **MAX** sia la più bassa possibile (implicitamente massimizza la sua utilità che, sotto l'assunzione di somma zero

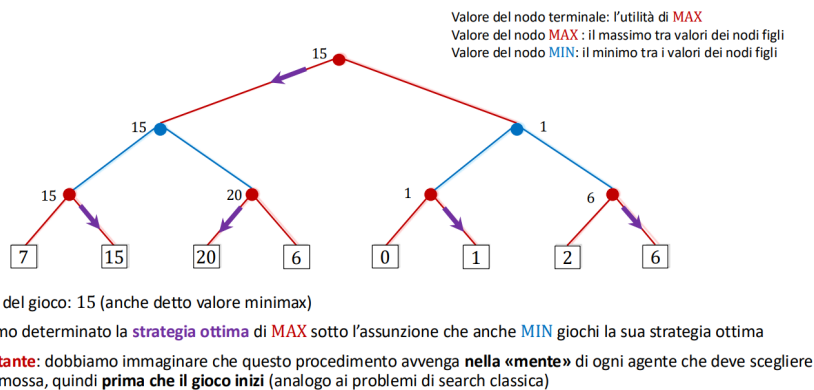
,equivale a minimizzare l'utilità dell'avversario).

Per semplicità **MAX** avrà sempre utilità  $\geq 0$  (e di conseguenza quelle di **MIN** saranno sempre  $< 0$ ). Consideriamo giochi piccoli per ragioni di spazio, ma le considerazioni si generalizzano su ogni gioco.



Procedimento:

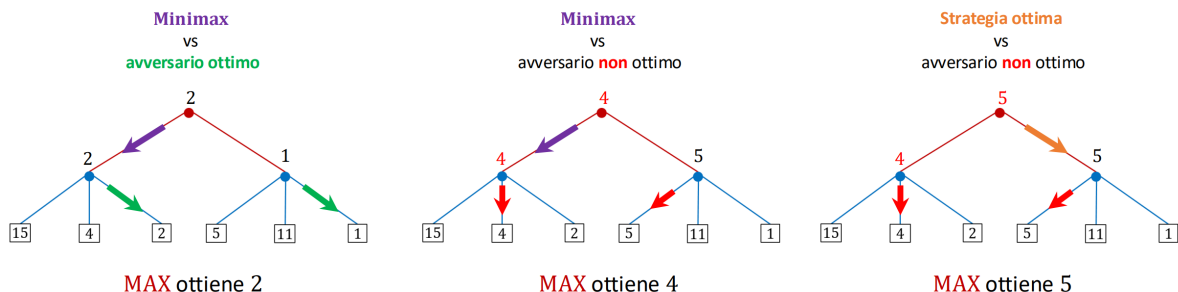
1. Eseguire una DFS sull'albero di gioco
2. Riposrto all'indietro dei valori



**Lato negativo**: bisogna attraversare tutto l'albero. Il risolutore, a differenza della mente umana ha un limite molto maggiore di ragionamento per sviluppare la propria mossa. Spesso la decisione è estremamente controintuitiva. Questo approccio va bene finché il numero di decisioni non aumenta.

### 5.3.1 Minimax con avversario non razionale

Minimax ragiona per trovare la strategia ottima assumendo che l'avversario giochi a sua volta la sua strategia ottima, date le nostre assunzioni sul meccanismo equivale a dire che l'avversario è **razionale**. Cosa succede se MAX ragiona assumendo un avversario razionale, ma poi, nel momento del gioco, il suo avversario non lo è? In tal caso la strategia Minimax non peggiora, ma **non è garantita che sia quella ottima**.



Minimax è una strategia **conservativa**. Il valore Minimax è un Lower Bound sul valore del gioco.

### 5.3.2 Minimax con più agenti

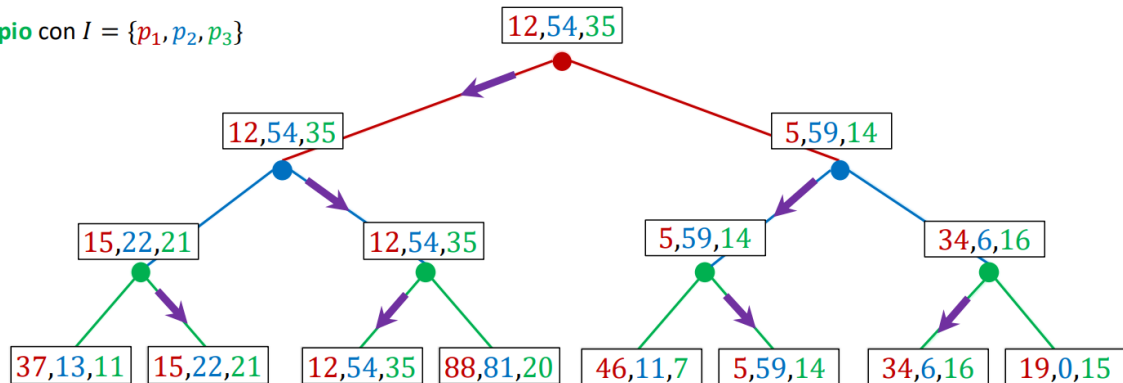
Minimax può essere facilmente esteso ad una classe di giochi equivalente a quella che abbiamo considerato, ma dove passiamo da 2 a  $n$  agenti giocatori. Non possiamo più rappresentare le utilità rispetto ad un



solo giocatore perchè il concetto di somma zero diventa più complicato.  
In ogni stato terminale ripostiamo un vettore di  $n$  utilità, una per ciascun giocatore.

Ciascun giocatore si comporterà come un MAX rispetto alla sua utilità

**Esempio** con  $I = \{p_1, p_2, p_3\}$



L'algoritmo sceglierà quale valore propagare in base al payoff che otterrebbe il giocatore che ha eseguito la mossa.

## 5.4 $\alpha - \beta$ pruning

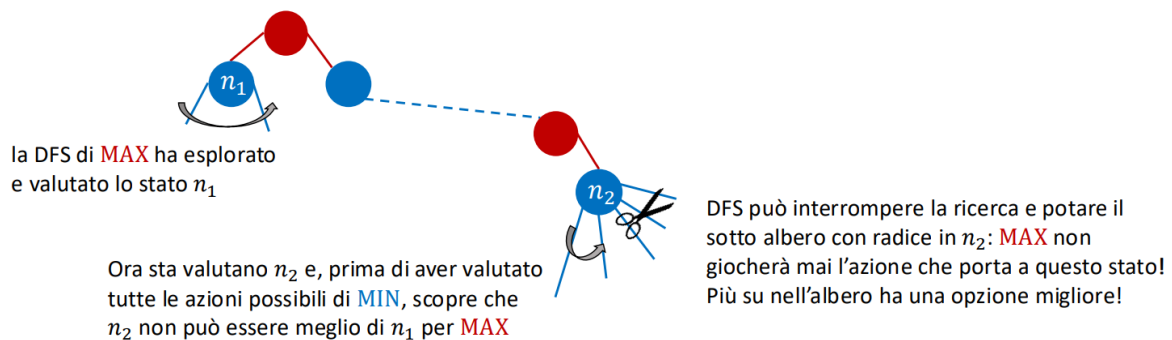
### 5.4.1 Migliorare le performance di Minimax

Minimax ha il problema di eseguire una DFS esaustiva su tutto l'albero di gioco per poter determinare la sua mossa. La complessità è quindi esponenziale nella profondità dell'albero di gioco:  $O(b^d)$ .

Per certi giochi quindi la complessità combinatoria elevata rappresenta un limite **impossibile** da valicare con l'approccio Minimax.

**Idea:** nei problemi di search abbiamo definito delle regole di **pruning** che ci hanno aiutato a migliorare l'efficienza pratica degli algoritmi: possiamo definire regole analoghe per la risoluzione di giochi?

Consideriamo questa situazione su un albero di gioco esplorato da Minimax



Questo schema può essere formalizzato e codificato in una regola di pruning.

### 5.4.2 Potatura $\alpha - \beta$

**Principio dell' $\alpha - \beta$  pruning:** man mano che la DFS procede, teniamo traccia, su ogni nodo del branch corrente, di ciò che sappiamo sul valore di quel nodo. Fino a che la DFS non ha esplorato tutti i nodi figli, non conosciamo con esattezza il valore del nodo. Ma qualcosa sappiamo!

Anziché un valore singolo, usiamo un intervallo di valori  $[\alpha, \beta]$ : indica che, stando a quanto scoperto dalla DFS fino a quel momento, il valore Minimax di quel nodo è compreso nell'intervallo (inizialmente è  $[-\infty, +\infty]$ ).

In molti casi, per scartare un nodo dall'albero di gioco non ci interessa conoscere l'esatto valore Minimax, ci basta  $[\alpha, \beta]$ .

- $\alpha$ : il minimo valore garantito per MAX.
- $\beta$ : il massimo valore garantito per MIN.

### 5.4.3 Esempio

