Intelligenza Artificiale I

Federico Falcone

November 6, 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 **sequenzza 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 conoscenscena 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 **aagente 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 **consequenzialismo**. 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 presazione che definisce il criterio di successo;
- la conoscenza pregressa del'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 ateso della sua misura di prestazione, date le informazioni fornite dalla sequenza percettiva e da ogni ulteriore conoscenza dell'agente.

1.2.3 Onniscenza, apprendimento e autonomia

Un agente **onniscente** 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 perte 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 id **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 **cooperativi**.
- **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 sula 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'istnate 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 ossevabili 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 seconod 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 **agesnete 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 bassati 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 presazione esterna concordino, un agente che sceglie le zioni 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 ch eporterà 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 ll'obiettivo, ovvero un modello astratto della parte del mondo interessata;
- **Ricerca**: prima di effettuare qualsiasi azione nel modno 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

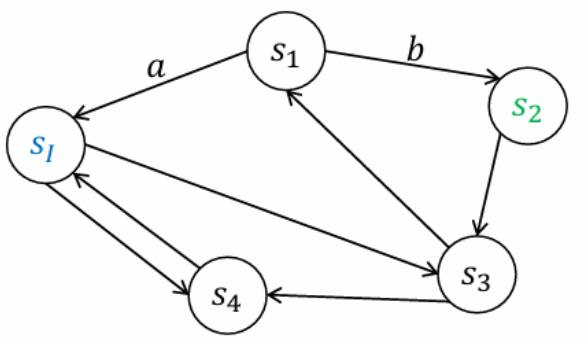
Un **problema** di ricerca può essere definito formalmente come segue:

- 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 pià **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 questa azioni è **applicabile** in s.

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

- Un **modello di transizione** che descrive ciò che fa ogni azione. Dato uno stato di partenza s_j e un'azione $a \in A(s_i)$, indica uno stato di arrivo $f(s_i, a)$: rappresenta la conseguenza dello svolgere l'azione a nello stato s.
- Una **funzione di costo dell'azione**, denotata da $c(s_j, a, f(s_i, a)$, restituisce il costo numerico di applicare l'azione a nello stato s_i 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 casua 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 ocntrollare 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. Considretiamo algoritmi ch esovrappongono un **albero di ricerca** al grafo dello spazio degli stati, formando vari cammini a partire dallo stato iniziale e carcando 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 del'albero di ricerca corrispondo ad azioni. La radice dell'alvvero 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 delgi 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 quasiasi 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 riscostruire 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 temrpo arbitrariamente lungo.
Se lo spazio degli stati è infinito? Possiamo chiederci se la riceca 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

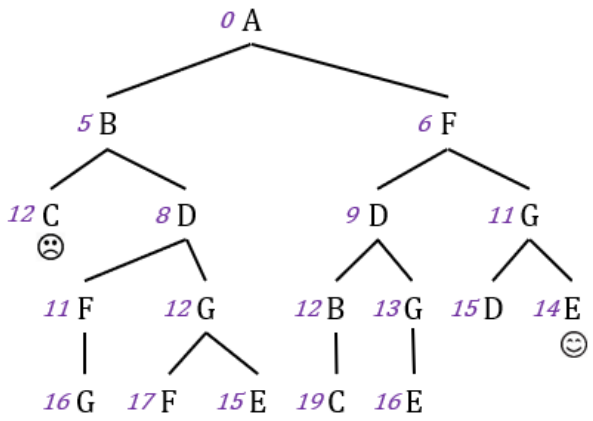
Complessità spaziale: come cresce la quantità di memoria richiesta dall'algoritmo di ricerca in funzione della dimensione del problema (caso peggiore)?
Complessità temporale: come cresce il tempo richiesto (numero di operazioni) dell'algoritmo di ricerca in funzione della dimensione del problema (caso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

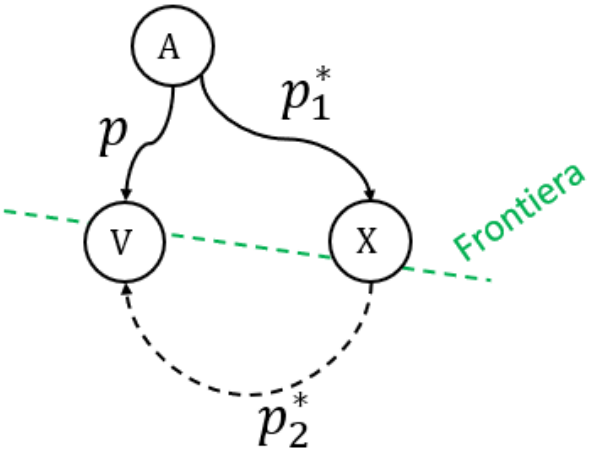
3.1 Uniform Cost Seatch (UCS)

Nell'albero di ricerca, teniamo traccia del nostro accumlato sul percorso dal nodo iniziale a ogni noto V: $g(V)$. Non consideriamo EQL.
L'UCS consiste nella selezione (espansione) del nodo con g minore ancora da espolare (sulla frontiera).
Goal check: se il **nodo selezionato per l'espandione** 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

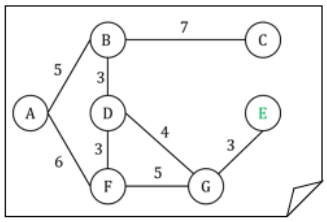


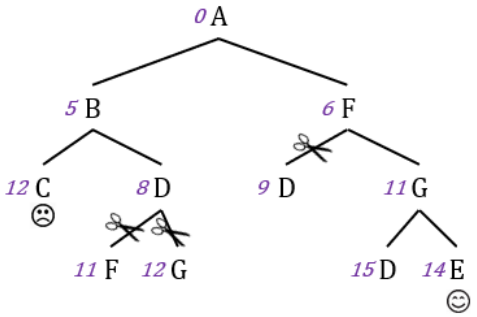
Overload della notazione: estendo la notazione di g rendendola applicabile anche ai path $g(X ß Y ß \dots ß Z) = g(Z)$).
Ipotesi:

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 odo per l'estensione:

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





$EXL = \{\emptyset\}$

$EXL = \{A\}$

$EXL = \{A, B\}$

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

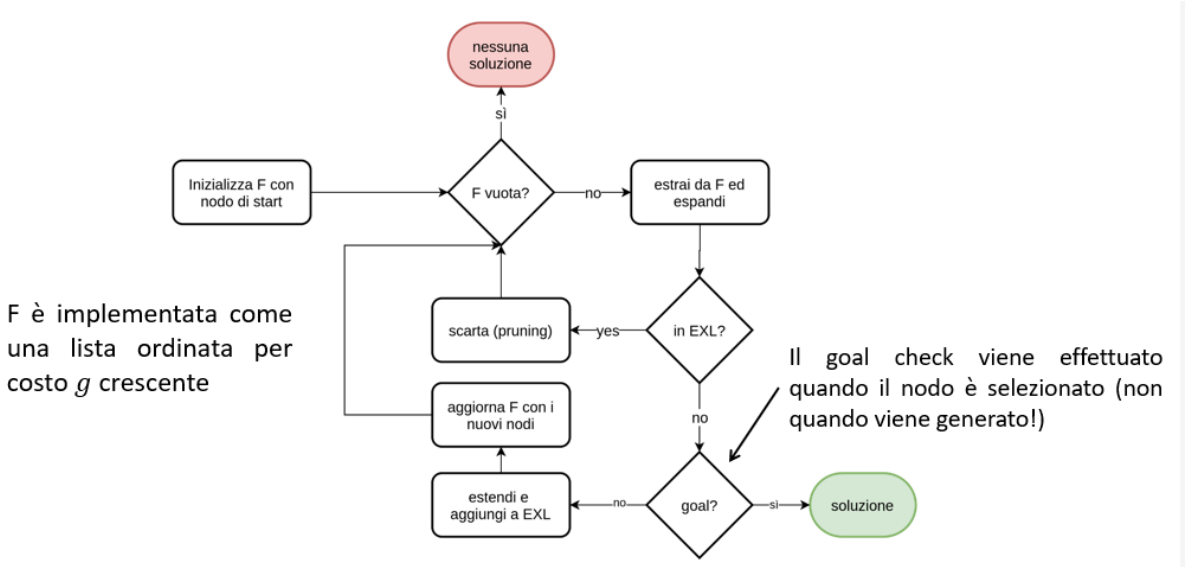
$EXL = \{A, B, F, D\}$

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

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

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

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 algorimti 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:

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

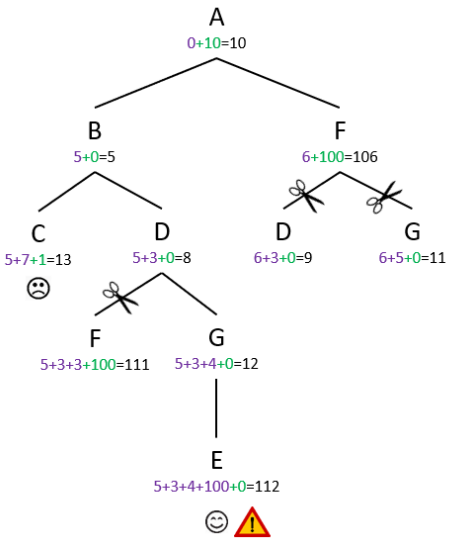
Dal momento che $g(n)$ fornisce il costo di cammino dal nodo iniziale al nodo n, e $h(n)$ rappresenta il costo stimato del cammino più conveniente da n all'obiettivo, risulta $f(n) =$ costo stimato della soluzione più

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.

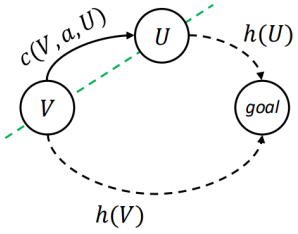
3.3.1 Ammisibilità di A*

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'ammisibilità 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 diseguaglianza: $h(V) \leq c(V, a, U) + h(U)$. (diseguaglianza triangolare)
Disuguaglianza triangolare Afferma che ogni lato del triangolo non può mai essere più lungo della somma degli altri due.

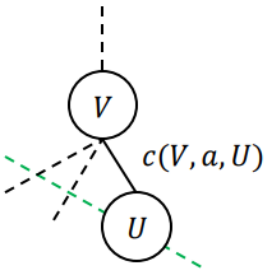


3.3.3 Ottimalità di A*

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. Sostituiendo in 2: f(U)=g(V) + c(V,a,U)+h(U).

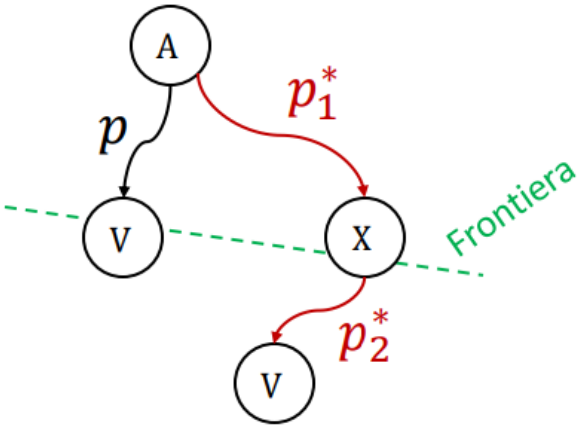
5. Per la proprietà di **consistenza** $c(V, a, U) + h(U) \geq h(V)$.
6. Sommando $g(V)$ a entrambi i termini: $g(V) + c(V, a, U) + h(U) \geq g(V) + h(V)$.
7. $f(U) \geq f(V) \Rightarrow$ **lungo ogni perorso nell'albero di ricerca la f è monotono non decrescente**.



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 vola 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: $p^* \neq p$

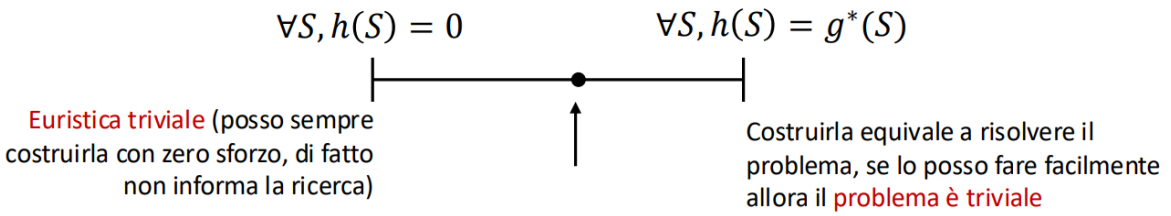
Dato 2 e la **separation property** della frontiera, sappiamo che deve esistere un nodo X sulla frontiera che si trova sul cammino ottimo $p^* = p_1^* + p_2^*$ verso V;



1. $g(p) > g(p^*)$ perch+ 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. $f(p, V) > f(p^x, V) \geq f(p_1^x, X)$ mettedno insieme 3, 2 e la definizione di f;
5. $f(p, V) > f(p_1^*, X)$ quindi V non può essere stato scelto prima di X, l'ipotesi 1 è violata

3.4 Progettare un'eutistica

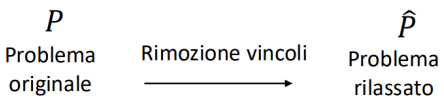
Per capire come trovare un metodo per costruire buone euristiche, cerchiamo prima di capire come si può valutare una euristia: ci sono euristiche che sono migliori di altre? E come si stabilisce?
Possiamo immaginare la nostra h come un punto in un intervallo limitato:



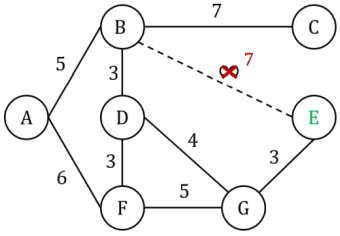
Vorrei spingere il più possibile a destra il punto, mantendendo però l'**efficienza** nel computare h. Perchè?
A* espande tutti gli stati S tali per cui $f(S) < g^*(goal)$ cioè tutti gli stati S tali per cui $h(S) < g^*(goal) - g(S)$. QUinsi se per ogni stato $S, h_1(S) \leq h_2(S)$ allora h_2 domina h_1 e A* con h_2 non espanderà ppiù nodi di A* con h_1 (sempre assumendo che entrambe le euristiche siano consistenti e calcolabili in tempo efficiente). É facile vedere che date due euristiche consistenti h_1 e h_2 dove nessuna domina l'altra, possiamo costruire una terza euristica che le domina entrambe: $h_3 = max\{h_1, h_2\}$
Il principio "euristiche più grandi sono migliori" ci permette di valutare le euristiche o di combinarle, ma come si costruisce una euristica da zero?
Tale compito sembra essere piuttosto complesso: l'euristica srutta la struttura di un problema e deve soddisfare una serie di vincoli.

3.5 Rilassamento di un problema

Dato un problema P, un **rilassamento di P** è una versione più semplice di P in cui alcuni vincoli sono stati rimossi.



I costi nel problema rilassato non sono mai maggiori dei costi nel problema originale
$\hat{c}(S, a, U) \leq c(S, a, U)$



Idea:

1. Costruire un rilassamento di P: $\hat{p}$.
2. Eseguire A* (o un qualsiasi altro algoritmo di ricerca) sul problema rilassato e trovare il costo ottimo da ogni nodo S per arrivare al goal: $\hat{h}^*(S)$.
3. Assegnare $h(S) = \hat{h}^*(S)$ ed eseguire A* con tale euristica.

Possiamo facilmente definire un problema di rilassamento, si tratta solo di rimuovere vincoli/riscrivere i costi. Ma cosa succede alle proprietà delle'euristica e all'ottimalità di A*?

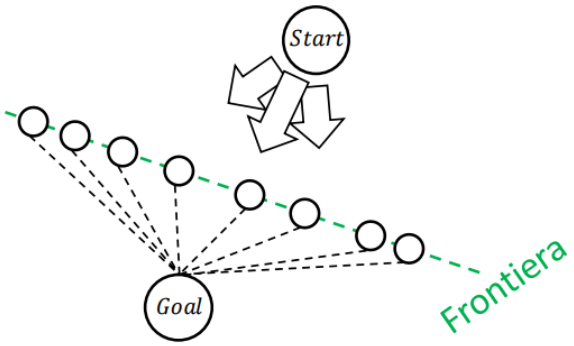
1. Nel problema rilassato risolvo per il costo ottimo da ogni stato $\hat{h}^*(S) \leq \hat{c}(S, a, U) + \hat{h}^*(U)$.
2. Dalla nostra idea $h(S) \leq \hat{c}(S, a, U) + h(U)$.
3. Dalla definizione di rilassamento $\hat{c}(S, a, U) \leq c(S, a, U)$.
4. **L'euristica h è consistente** $h(S) \leq c(S, a, U) + h(U)$.

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.



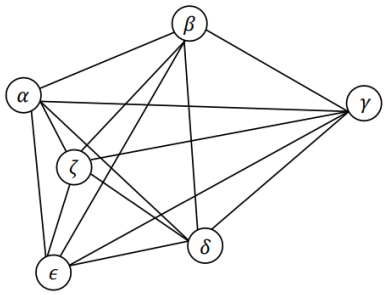
A* spenderà un sacco di tempo a distinguere il path ottimo tra tutti questi path che sostanzialmente sono equivalenti. NOn si accontenta di un path sub-ottimale, anche quando il suo costo dista di pochissimo dall'ottimo.
Come possiamo equipaggiare A* con un po' di flessibilità?.

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

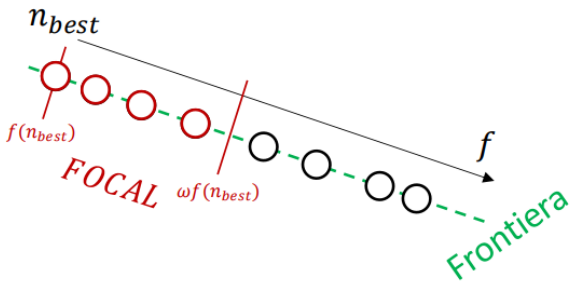


- Nodi → città
- Collegamenti → strade, ciascuna strada ha un costo diverso e positivo (grafo completo)
- Soluzione: ciclo Hamiltoniano di costo minimo ($\mathcal{NP}$-Hard)
- **Problema di search (?)**
  - Stato → percorso parziale (es. $n = \langle \alpha \to \zeta \to \delta \rangle$)
  - Costo →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

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

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

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



Regola di espansione: scegliere la $FOCAL$ il nodo che minimizza $\hat{h}_F, n_{next} = argmin_{n \in FOCAL} \hat{h}_F(n)$.
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 ω.

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 ω 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	**Hill Climbing**
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	**Beam**
DFS limitando la profondità massima a 1, poi a 2, poi a 3 ...	**Iterative Deepening**
Iterative Deepening, ma anziché limitare la profondità uso f	**IDA***
A* dal nodo di goal cercando una strada verso quello iniziale	**D***
Come A* ma usando $f(n) = h(n)$	**Greedy Search**

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 conglitto.
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 l ostato desiderato, l'agente da noi controlalto deve considerare le **strategie** degli altri decisori.
Esistono diversi tipi di giochi:

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

Lo studio del comportamente 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**: $S = \{s_1, s_2, \ldots\}$ 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 $A = \{a_1, a_2, a_3, \ldots\}$.
- **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 terminiali**: 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 acceso 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 trasformto 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 trasferimetno 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 comportamente 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 prinicipalmente sulle **strategie pure**
Strategia ottima: quella strategia tale per cui ogni strategia diversa non introduce miglioramenti contro un avversario **infallibile**.
Per ricolvere un gioco bisogna calcolare la strategia ottima.

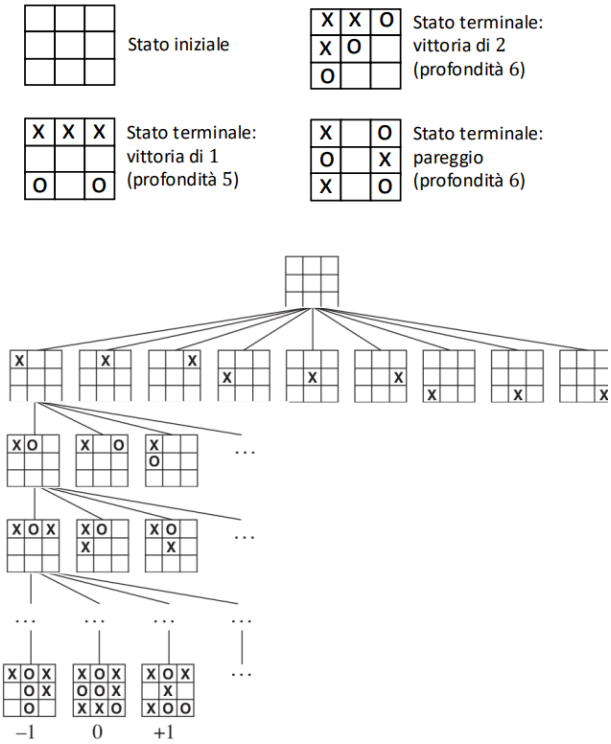
5.2.1 Albero di gioco

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'oppsoto). Esempio: il gioco del tris, Giocatore i_1: X, giocaotre i_2:0. 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 simmetriche 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: $\leq 1 + 9 + (9 * 8) + (9 * 8 * 7) + (9 * 8 * 7 * 6) + \ldots = 986410$ se non facciamo ottimizzazioni è una stima ragionevole.
Numero di giochi (ovvero i branch dell'albero o i suoi nodi terminali):

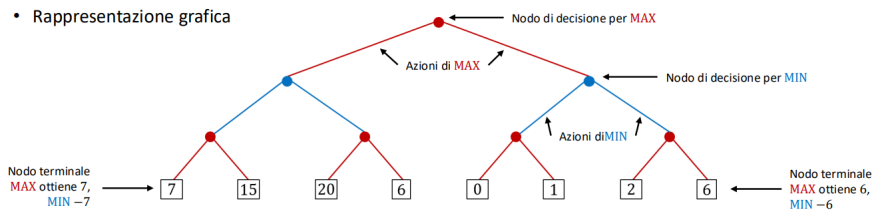
- $9 * 8 * 7 * 6 * 5 * 4 * 3 * 2* = 9! = 362880$
- numero esatto 255168, al netto delle simmetrie 26830.



5.3 Minimax

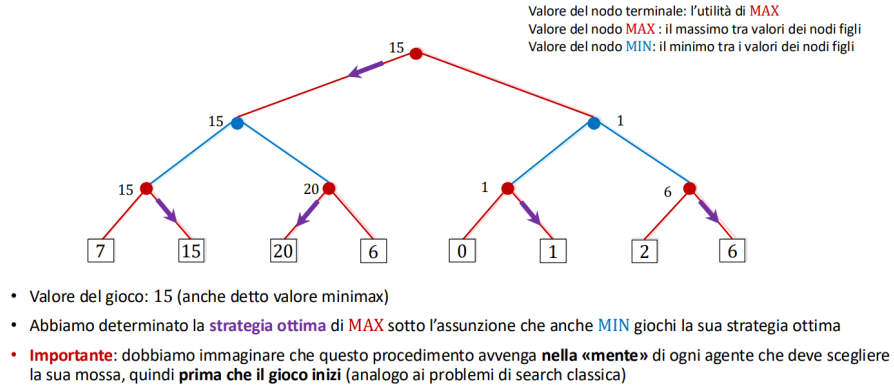
Nei problemi di search l'albero di ricerca rappresentava/supportava il processo di inferenza per trovare una strada o la strada ottima per il goal. L'albero di gioco ha una funzione analoga: rappresenta/supporta il ragionamento strategico.
Per adesso assumiamo che trovaer la strategia ottima sia un problema di search che tiene conto dell'avverario: **aversarial search**. L'approccio di base con cui si risolve la classe di giochi che stiamo considerando è dato dall'algormitmo **minimax**, sostanzialmente una DFS sull'albero di gioco con un **ritorno all'indietro** dei valori di utilità.
È un approccio esautistivo e inefficiente che possiamo adottare solo per giochi di dimentisioni limitate, ma è un approccio esatto, corretto e completa; sta alla base dei metodi più sofisticati.
Introduciamo della 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 ≥ 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:

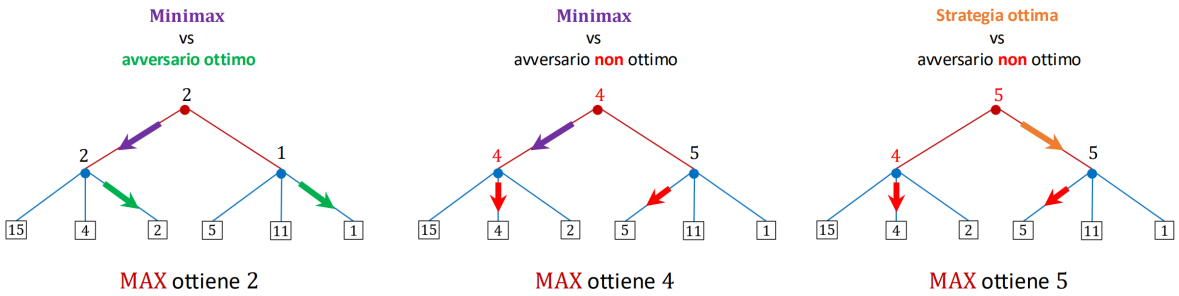
1. Eseguire una DFS sull'albero di gioco
2. Riposrto all'indietro dei valori



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 finchè 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 dure 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.

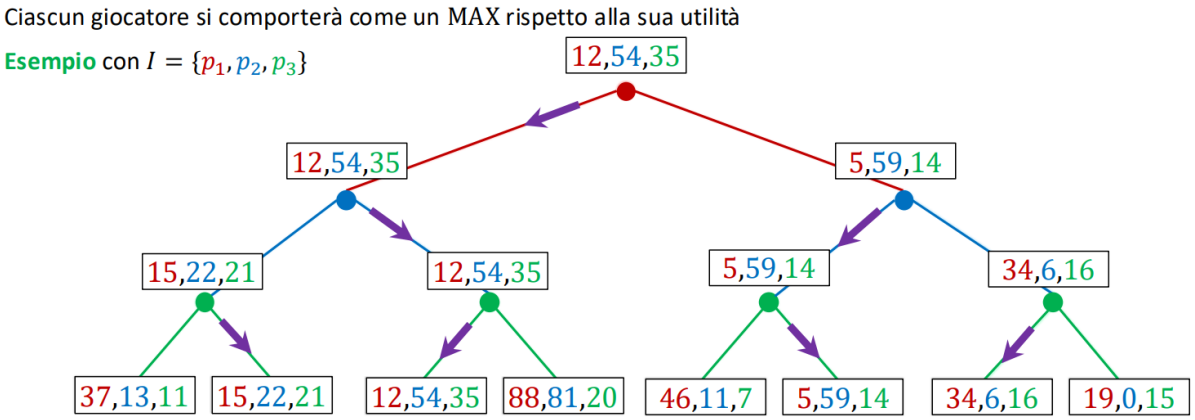


Minmax è una strategia **conservativa**. Il valore Minimax è un Lower Bound sul valore del gioco.

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 pià 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.

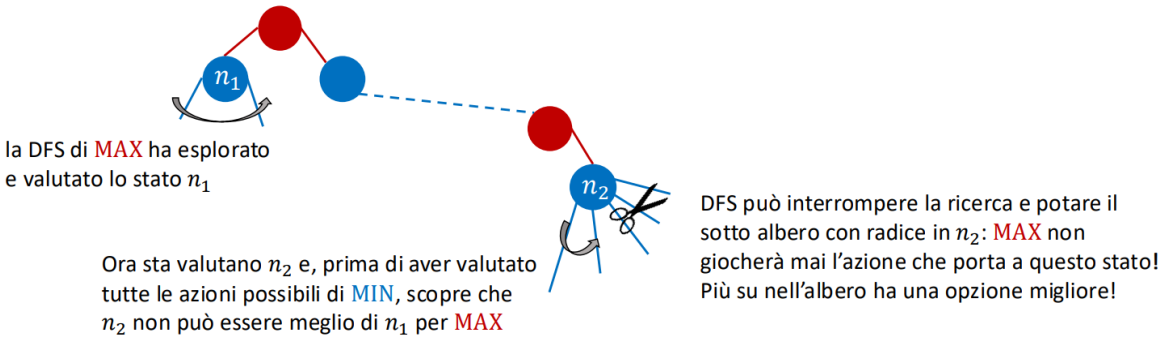


L'algoritmo sceglierà quale valore propagare in base al payoff che otterebbe il giocatore che ha eseguito la mossa.

5.4 $\alpha - \beta$ pruning

5.4.1 Migliorare le performance di Minimax

Minimax ha il problema di eseguire uan 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 **prnunig** 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 albeero di gioco esplorato da Minimax



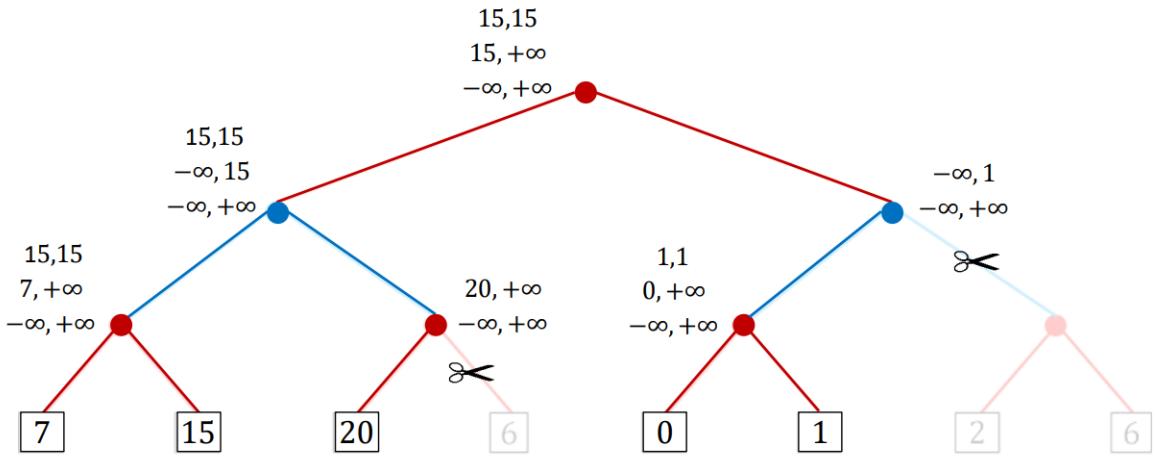
Questo schema può essere formalizzato e codificato in una regola di pruning.

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!
Anzichè 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 $[\alpha, \beta]$.

- α: il minimo valore garantito per MAX.
- β: il massimo valore garantito per MIN.

5.4.3 Esempio



La potatura $\alpha - \beta$ può introdurre diversi vantaggi, ma ha un problema di fondo sul quale ci si può fare poco: **almeno una porzione dell'albero di gioco va esplorata fino ai nodi terminali**. Questa soluzione risulta impossibile in molti casi reali per due ragioni:

1. Complessità elevatissima dell'albero di gioco;
2. Tempo limitato entro cui va presa la decisione su quale mossa fare.

C'è un aspetto più di fondo: risolvere il gioco non è poi così interessante. Immaginiamo due giocatori ideali e cioè entrambi in grado di risolvere il gioco all'esatto, la partita può andare solo in tre modi: uno dei due si arrende o si mettono d'accordo per pareggiare.
Ricapitolando:

1. Sarebbe bello avere un **ordering perfetto** delle mosse, ma per averlo equivale a risolvere all'esatto;
2. Sarebbe bello poter risolvere **all'esatto**, ma non abbiamo abbastanza risorse computazionali.

Domanda sul problema 1: sarebbe possibile calcolare un ordine delle azioni man mano che la ricerca procede? Non l'ordine perfetto, ma uno euristico, informato da ciò che la ricerca ha scoperto fino a quel momento.
Domanda sul problema 2: sarebbe possibile assegnare un valore numerico anche ad un nodo non terminale p dell'albero di gioco, in modo da stimare il valore Minimax senza dover scendere in profondità fino ad almeno una foglia del sotto-albero con radice in p?
La risposta ad entrambe le domande è sì e la tecnica per farlo rende la potatura α, β più efficiente combinando tre elementi:

- funzioni di **valutazione** e **cutoff**;
- tabelle delle **trasposizioni**;
- **iterative deepening**.

5.4.4 Funzioni di valutazione e cutoff

Dato un nodo s dell'albero di gioco, anzichè calcolare il suo valore Minimax fornisco una stima di quel valore applicando una funzione di valutazione $v(s)$. La funzione v è un'**euristica**: produce una stima della qualità di quel nodo, codifica, in una computazione **efficiente**, quello che un giocatore umano fa quando cerca di intuire la bontà del trovarsi in una particolare situazione del gioco.
Approccio: data v e un **test di cutoff** $CUT(s, d)$ per un nodo s alla profondità d, se il test restituisce false, eseguo Minimax da quel nodo, altrimenti fermo Minimax e restituisco $v(s)$.
Introduciamo incertezza: la complessità computazionale mi impedisce di calcolare il valore esatto di ogni nodo, anche se, viste le assunzioni sul meccanismo, sarebbe possibile. Se la funzione di valutazione è **efficace**, posso riuscire comunque a identificare buone mosse.

Posso fermare la ricerca dopo un tempo ragionevole e fornire comunque un'azione da giocare!
Posso usare v per ordinare le azioni e valutare prima quelle che sembrano migliori!

Dato uno stato si possono definire delle **feature** che lo descrivono. Ad esempio scacchi: $(n_1, n_2, \ldots, n_6)$, dove n_i è il numero di pezzi del tipo i rimasti nella scacchiera.

- funzione basata sull' **esperienza** (tante partite che abbiamo giocato e salvato in un dataset):
  - Ogni profilo di feature p definisce una classe di equivalenza tra stati;
  - Dall'esperienza sappiamo che di tutte le occasioni in cui ci siamo trovati in uno stato con profilo p, nel 50% dei casi abbiamo vinto, nel 20% abbiamo perso, mentre nel restante 30% abbiamo pareggiato;
  - Quindi se s è un nodo con profilo p, possiamo assegnargli il valore $v(s) = 0,5 * 1 + 0,2 * (-1) + 0,3 * 0 = 0,3$.
- funzione basata sulla **combinazione** di feature:
  - Se chiamo $f_1(s), f_2(s), \ldots, f_n(s)$ le n feature che ho definito per un generico stato s, posso aggregarle per ottenere un valore:

$$v(s) = \omega_1 f_1(s) + \omega_2 f_2(s) + \ldots + \omega_n f_n(s)$$

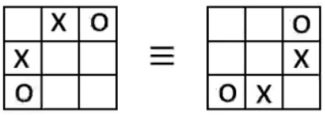
Nella pratica queste funzioni sono convenienti se:

- Riusciamo a definire buone feature **manualmente**.
- Si può calcolare in modo **efficiente** le funzioni di valutazione (tempo polinomiale con esponente basso).

Non sempre questi due scenari si verificano, in certi casi non è ovvio come buone feature siano definite, serve un approccio alternativo per identificarle (es. deep learning).
Come definire il test di cutoff $CUT(s, d)$? Basato su soglia di profondità **massima**.
Identificazione degli **stati quiescenti**: gli stati in cui non ci si aspettano grandi cambiamenti in termini di valore.
Quiescent search: esegue un cutoff restituendo $v()$ solo per gli stati "quieti", dove non ci sono minacce o opportunità imminenti. Se uno stato non è quiescente, si va avanti in profondità.
L'arroccio della quiescent search sfrutta una caratteristica ricorrente in molti giochi: gli episodi rilevanti si sviluppano in regioni di profondità relativamente compatte e distanziate:

5.4.5 Trasposizoni

Molti nodi nell'albero di ricerca sono strategicamente equivalenti perchè identici o per via di simmetrie e regolarità.
Esempio: questi due nodi sono equivalenti a meno di una rotazione, dal punto di vista strategico sono lo stesso



In altri casi uno stesso stato può essere raggiunto da sequenze diverse di mosse, in generale uno stato che richiede m mosse da parte di ogni giocatore può ammettere fino a $m!^n$ diverse sequenze (dove n è il numero di giocatori).
Trasposition table: hash table dove per ogni stato $\overline{s}$ valutato dalla ricerca mantengo questi campi:

- Il valore dello stato $\overline{v}$ e la mossa $\overline{a}$ associata a quel valore.
- La profondità massima $\overline{d}$ da cui è arrivata la ricerca che ha valutato quel nodo (dipende dal cutoff).

Se durante la ricerca (con limite di profondità $\overline{d}$) tutto il sotto-albero è stato esplorato fino ai nodi terminali, allora $\overline{v}$ è il valore Minimax del nodo; in tutti gli altri casi è un bound ($\geq$ o $\leq$ a seconda se il nodo è MIN o MAX)
È possibile $\overline{v}$ sia stato determinato dopo una potatura α (se il nodo è MIN) o β (se il nodo è MAX), quindi non è detto che tutte le azioni siano state valutate.

5.5 Giochi stocastici, incertezza

Introduciamo l'**incertezza** nel nostro modello di gioco, questa nuova caratteristica ha un duplice ruolo:

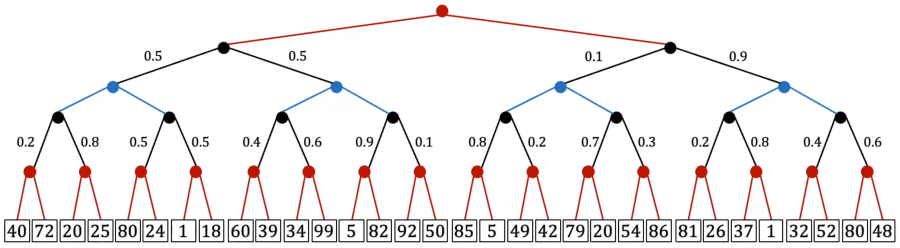
- ci permette di avere un modello più generale con cui descrivere i giochi dove alcune dinamiche sono random;
- ci permetterà di definire un nuovo approccio per la risoluzione dei giochi non più basato sulla ricerca esatta, ma su una ricerca "probabilistica" che può essere usato sia con giochi stocastici che non.

Come si introduce l'incerteza nei giochi? Si introduce un nuovo giocatore: **la Natura:** $\mathcal{N}$.
$\mathcal{N}$ può essere pensata come un giocatore analogo a tutti gli altri: avrà i suoi turni di gioco e, di conseguenza, i suoi nodi di decisione sull'albero di ricerca (detti anche **nodi di chance**).
Tuttavia, rispetto agli altri giocatori presenta **tre** differenze fondamentali:

- La sua strategia di gioco è **fissa a priori** ed è conosciuta da tutti gli altri giocatori da prima che il gioco inizi (come se $\mathcal{N}$, prima di iniziare la partita, dichiarasse pubblicamente la sua strategia e, durante il gioco, rispettase la promessa).
- La sua strategia di gioco è **mista** (una distribuzione di probabilità sulle azioni), quindi nel momento di decidere l'azione si lanciano dei dadi.
- $\mathcal{N}$ non riceve payoff, è totalmente agnostica rispetto alle dinamiche del gioco , quindi la sua strategia è un **fenomeno dato** non il risultato di un ragionamento rispetto a delle preferenze.

È sostanzialmente l'ambiente, a cui non frega di vincere o di perdere

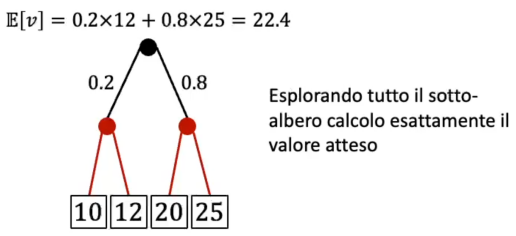
Come cambia la rappresentazione grafica del gioco?

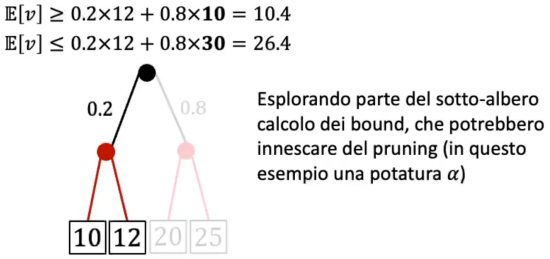


Le mosse colorate in nero sono quelle dell'ambiente, nodi di chance. Gli agenti si trovano in un ambiente **stocastico**

5.5.1 Expectimax

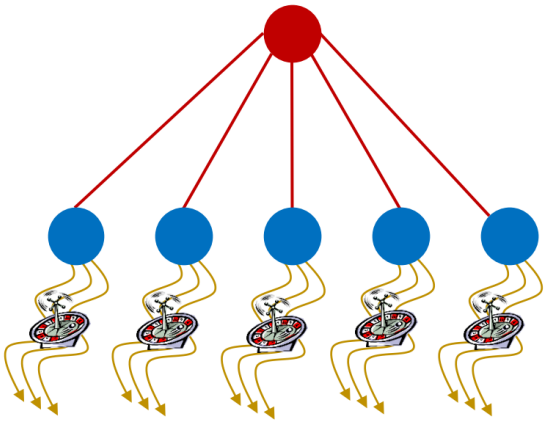
Visto che $\mathcal{N}$ non ha un comportamento strategico, la logica Minimax è ancora valida: possiamo estendere l'algoritmo in modo che gestisca l'incertezza.
Idea: Minimax dove il valore dei nodi di chance non è determinato con il *max* o *min* dei valori sottostanti, ma con il **valore atteso** $E[X]$ definito come la somma dei valori sottostanti, ciascuno moltiplicato per la probabilità di finire in quel nodo.
Nel caso di somma-zero a due giocatori con turni alternati (che includano anche $\mathcal{N}$), l'algoritmo Minimax si estende naturalmente e diventa Expectimax.
Problema: la potatura α, β sotto un nodo di chance (e quindi in generale) **non si può applicare** perchè la scelta dell'azione non è guidata da un giocatore. Il valore che "sale" dai nodi sottostanti cambia se ne scartiamo alcuni. Questo non valeva con MAX o MIN perchè il massimo/minimo tra n valori rimane lo stesso se, tra quegli n, si scartano i valori che non sono il massimo/minimo.
Se ci sono date assunzioni sui payoff però si può fare!
Esempio supponiamo di sapere che $10 \leq u_1 \leq 30$ in qualsiasi terminale.





5.5.2 Stimare il valore di un'azione

L'incertezza non è una caratteristica del problema che dobbiamo gestire, ma può essere "sfruttata" nella definizione di un metodo risolutivo alternativo.
Con l'eccezione di $PROBCUT$, tutti i metodi che abbiamo studiato si basano su inferenze esatte; tutti questi approcci soffrono di un problema di scalabilità.
Idea: trasformare il processo di inferenza in un **processo di stima**: non cerco l'azione migliore, ma stimo la bontà di ogni azione e scelgo quella con il valore stimato più alto.
Se il processo di stima è ben definito, con il tempo, i valori stimati convergono al valore esatto.

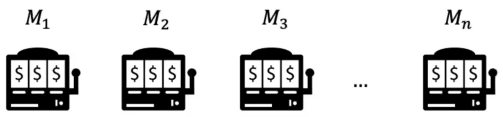


- Considero un'azione e lo stato s in cui mi porterebbe.
- Simulo un numero molto alto di partite a partire da s: gioco contro me stesso scegliendo le azioni in modo casuale.
- La media dei punteggi finali è una stima del valore di s.
- Ripeto per ogni azione e scelgo quella con la stima più alta.
- Questa idea si chiama Monte Carlo Tree Search (MCTS).

5.5.3 Monte Carlo Tree Search (MCTS)

Caso semplificato: stimare la **winning rate** di un'azione a_i, cioè la probabilità di vincere la partita se mi trovassi nello stato, che indico con s_i, in cui tale azione mi porta.
Posso scegliere tra n azioni $(a_1, a_2, \dots, a_n)$ e ho un tempo limitato che consente di svolgere un totale di N simulazioni (supponiamo di non poter parallelizzare).
Appoccio naive: eseguo $\frac{N}{n}$ simulazioni per ogni s_i e per ciascuno ottengo la stima della winning rate $\omega_i = \frac{nW_i}{N}$ dove W_i è il numero di simulazioni in cui ho vinto; alla fine gioco $a^* = argmax_i\{\omega_i\}$.
Problema: sto suddividendo lo sforzo in modo uniforme e non informato, potrei spendere troppo tempo su azioni che è chiaro che nono sono buone e, viceversa, troppo poco tempo su azioni che sembrerebbero buone e che richiederebbero maggior precisione di stima (più simulazioni delle altre).
Soluzione: concentrare più sforzo (numero di simulazioni) verso le azioni che si configurano come migliori.
ATTENZIONE! Devo stare attento a non escludere del tutto le azioni che non sembrano buone perchè, magari, con più simulazioni posso scoprire che in realtà sono migliori! Questo è un problema fondamentale dell'informatica: **EXPLORATION VS ESPLOITATION** dilemma.

5.5.4 Multi-Armed Bandit Problem (MAB)



MAB

- ci sono n slot machines, ciascuna con un vincita attesa $\mathbb{E}[M_i]$ diversa
- Ho a disposizione N giocate, voglio massimizzare la mia vincita totale
- Exploration: giocare su slot machines diverse per scoprire se sono profittevoli o no
- Exploitation: giocare sulle slot machines che credo essere profittevoli

Il nostro problema è un MAB:

- Slot machine $M_i \leftrightarrow$ azione a_i
- N giocate $\leftrightarrow$ N simulazioni
- Voglio massimizzare la mia vincita totale $\leftrightarrow$ voglio scoprire l'azione migliore
- Exploration: provare diverse azioni
- Exploitation: confermare la bontà delle azioni che sembrano buone

Come spendere al meglio le N giocate? **Risultato fondamentale**: simula l'azione che massimizza l'**Upper Confidence Bound** $UCB_i = \omega_i + c\sqrt{\frac{logN}{N-i}}$ dove N_i è il numero di simulazioni svolte per a_i e c è un parametro che bilancia exploration ed exploitation.

5.5.5 MCTS con UCB

Vantaggi:

- **Anytime**: fermando l'algoritmo ad un qualsiasi tempo t otteniamo la decisione migliore possibile con le informazioni raccolte fino a quel momento.
- Non necessita euristiche, funzioni di valutazione, definizione di feature, ...
- **Informato**: la ricerca si concentra maggiormente sui branch più promettenti (senza tralasciare mai completamente tutti gli altri).