Teoria dei giochi

Algoritmi per l'intelligenza artificiale

Vincenzo Bonnici
Corso di Laurea Magistrale in Scienze Informatiche
Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Universitá degli Studi di Parma

2025-2026

Teoria dei giochi

Partendo dal **Problem Solving**:

- **game playing**: introduciamo nel dominio del problema altri agenti in competizione
- **planning**: complichiamo introducendo stati (congiunzioni di fatti), e operatori (legami tra fatti-condizioni e fatti-effetti, non tra stati)

- Von Neumann & Morgenstern (1944) "Theory of Games and Economic Behaviour"
  - **Teoria della decisione**: analizzare il comportamento individuale le cui azioni hanno effetto diretto (scommesse e mondo dei puzzle)
  - **Teoria dei giochi**: analizzare il comportamento individuale le cui azioni hanno effetto che dipende dalle scelte degli altri (mondo dei giochi a più giocatori)

Teoria dei giochi

- Classificazione secondo **condizioni di scelta**:
  - Giochi con informazione **perfetta**
    - Gli stati del gioco sono completamente espliciti per gli agenti.
  - Giochi con informazione **imperfetta**
    - Gli stati del gioco sono solo parzialmente esplicitati.
- Classificazione secondo **effetti della scelta**:
  - Giochi **deterministici**
    - Gli stati sono determinati unicamente dalle azioni degli agenti
  - Giochi **stocastici**
    - Gli stati sono determinati anche da fattori esterni (es: dadi)

	Informazione Perfetta	Informazione Imperfetta
Giochi deterministici	Scacchi, Go, Dama, Otello, Forza4	MasterMind (è un gioco o un puzzle?)
Giochi stocastici	Backgammon, Monopoli	Scarabeo, Bridge, Poker… (giochi di carte) Risiko

Teoria dei giochi

Altre classiicazioni:

- Numero giocatori (tutti multiagenti!)
- Politica del turno di giocata
  - Diacronia (turni definiti/indefiniti)
  - Sincronia
- Ambienti discreti / continui
- Ambienti statici / dinamici
- Ambienti episodici / sequenziali
- Giochi a somma zero

L'uomo agisce in un ambiente continuo, dinamico, sequenziale, a scelte sincroniche e con informazione imperfetta.

Giochi e problem solving

Si può analizzare un gioco come un problema di ricerca anche se è multiagente?
Esempio, gli **scacchi**:

- X = tutti gli stati della scacchiera
- X_0 = lo stato di inizio gioco
- $SCS(x)$ =le mosse legali ad uno stato
- $T(x)$ = scacco matto
- g = numero di mosse

Problemi:

- g non è determinante
- $SCS(x)$ è sotto controllo solo per metà delle mosse e spesso non è reversibile
- $T(x)$ non è sufficiente per definire la terminazione. Serve una **funzione di utilità** sulla terminazione. Es: vittoria = $+1$, patta = 0, sconfitta = -1
- Obiettivo dell'agente: definire una strategia che raggiunga $T(x) = +1$

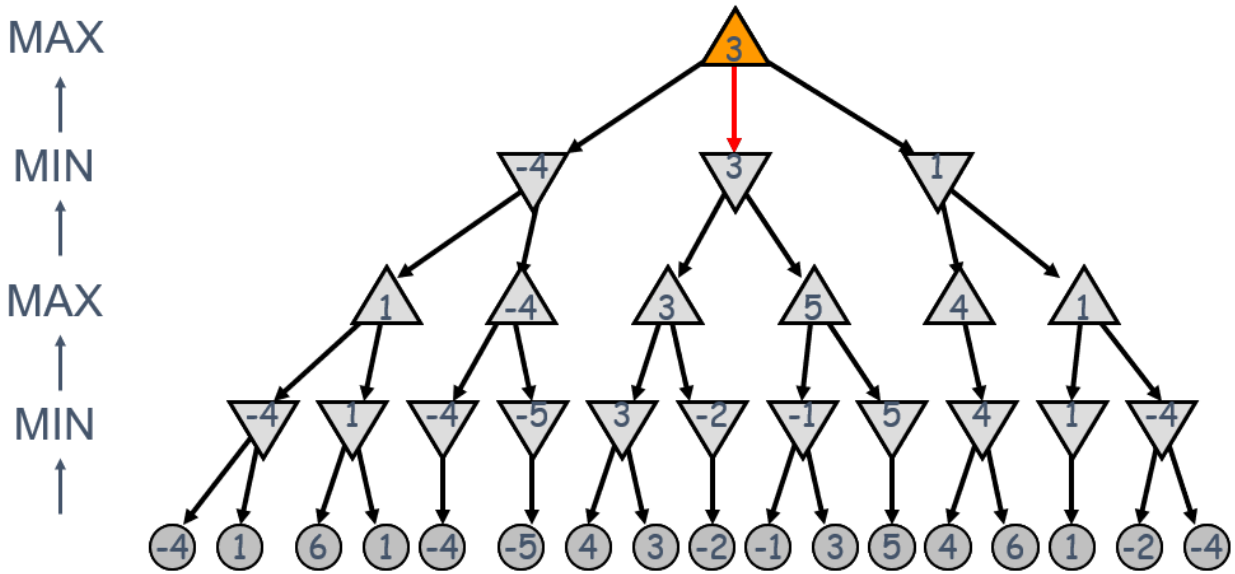
Giochi e problem solving

- Per inserire un gioco ad informazione perfetta in uno schema classico di search si considera che:
  - Esiste un avversario che va simulato
  - L'avversario minimizza il nostro utile
- L'albero di ricerca si sviluppa su 2 giocatori:
  - MAX(noi)  e MIN (l'avversario) 
- L'obiettivo è raggiungere uno stato terminale di quest'albero con la massimizzazione dell'utilità
  - se l'avversario inizia per primo: lui diventa MAX e noi MIN con lo scopo di minimizzare l'utilità

Algoritmo Minimax

- Nei giochi ad informazione perfetta si può ottenere la strategia perfetta con una ricerca esaustiva. (Von Neumann '28, Shannon '50)

- Funzionamento:
  - Si costruisce l'albero delle mosse fino ai nodi terminali
  - Si applica la funzione di utilità $U(x)$ ai nodi terminali
  - Si usano i valori per calcolare l'utilità dei nodi superiori:
    - $U(nodo_{sup}) = MAX\ U(nodo_{inf})$ se la mossa spetta a MAX
    - $U(nodo_{sup}) = MIN\ U(nodo_{inf})$ se la mossa spetta a MIN



Algoritmo Minimax

```
> MAX = true, MIN = false
> MINIMAX(X, MAX)

MINIMAX(nodo, agente)
```

```
figli[] = SCS(nodo, agente)
for all (figli){
    if(END_test(figlio) == true) {
        figlio.utilità = UTILITY_test(figlio)
    }
    else figlio.utilità = MINIMAX(figlio, !agente)

    if(agente==MAX && figlio.utilità > best)
      best = figlio.utilità
    if(agente==MIN && figlio.utilità < best)
      best = figlio.utilità
} return best
```

Proprietà:

- E' **completo** in grafi finiti
- E' **ottimale** se MIN è ottimale (e se ci sono più avversari).
- Se MIN non è ottimale non si può garantire l'ottimalità, ma...
- Ha **complessità spaziale** $O(bm)$ perché la **ricerca è in profondità**.

Algoritmo Minimax e gli scacchi

- Negli schacchi "Unfortunately, the number of possible positions in the chess tree surpasses the number of atoms in the Milky Way." (Claude Shannon)
- In generale: complessità temporale $O(bm) \mapsto$ negli scacchi 35^{100}
- In problemi reali non lo si può usare. E' utile solo come base teorica.

Minimax + taglio di profondità

- Limitare la ricerca ad una profondità max (dipendente dalla memoria e dal tempo disponibile)
- Per valutare l'utilità dei nodi foglia serve una **funzione di valutazione**, cioè un'**euristica** (visto che non sono nodi terminali)
- Far risalire fino alla radice le stime usando minimax ed effettuare la scelta

Euristiche:

- Funzioni **lineari** pesate del tipo $w_1 f_1 + w_2 f_2 + \cdots + w_n f_n$
  - per esempio negli scacchi: 1 punto x Pedone, 3 x Alfiere, 3 x Cavallo, 5 x Torre, 9 x Regina
  - Vantaggi: la linearità permette rapidità di calcolo
  - Svantaggi: povertà espressiva (es: Cavallo forte nelle aperture e al centro, Alfiere nelle chiusure, i valori delle combinazioni di pezzi non sono lineari)
- Funzioni **non-lineari**
  - Es. ottenuti da "learning", ma come definire i TARGET?

Minimax + taglio agli stati quiescenti

Quiescenza = proprietà di uno stato la cui euristica di utilità non varia molto con l'applicazione degli operatori.

- Arrivati alla profondità di **taglio**:
  - Per i nodi foglia **quiescenti** si applica il taglio
  - Per i nodi **non quiescenti** si approfondisce l'albero con una ricerca di quiescenza
  - Al termine della ricerca si applica il taglio

Problemi:

- Vogliamo arrivare a profondità 6 in una partita di scacchi (3 mosse MAX, 3 MIN)
  - $b = 35$, numero di nodi $= 35^6 \mapsto 1.85 \cdot 10^9$
  - con un calcolatore veloce in grado di eseguire 10^6 mosse al secondo ci vogliono 30 minuti.
  - otteniamo un giocatore mediocre

Alfa-beta pruning (McCarthy '56)

- Si può ottenere la mossa MAX senza osservare esaustivamente l'albero, perché:
  1. DATO $U(n_0) = \alpha \mapsto$ utilità minimax del nodo n_0 su cui sceglie MAX
  2. affinchè la scelta conclusiva di MAX sia $\neq \alpha$ almeno 1 nodo n ("fratello" di n_0) deve avere $U(n) > \alpha$
  3. affinchè $U(n) > \alpha$ per ogni nodo n' successore di n deve valere $h(n') > \alpha$
  4. QUINDI: appena un successore di n possiede $U(n') \leq \alpha$ il sottoalbero restante può essere potato
- stesso discorso vale per MIN

Alfa-beta pruning

- Nella ricerca nell'albero:
  - Si usano 2 variabili:
    - α = valore maggiore di MAX al tempo attuale
    - β = valore minore di MIN al tempo attuale
  - Calcolando MAX si pota il sottoramo di un nodo se un suo figlio ha valore inferiore ad α;
    se invece tutti i figli hanno valore maggiore il minimo diventa α
  - Calcolando MIN si pota il sottoramo di un nodo se un suo figlio ha valore maggiore a β;
    se invece tutti i figli hanno valore minore il massimo diventa β

Alfa-beta pruning

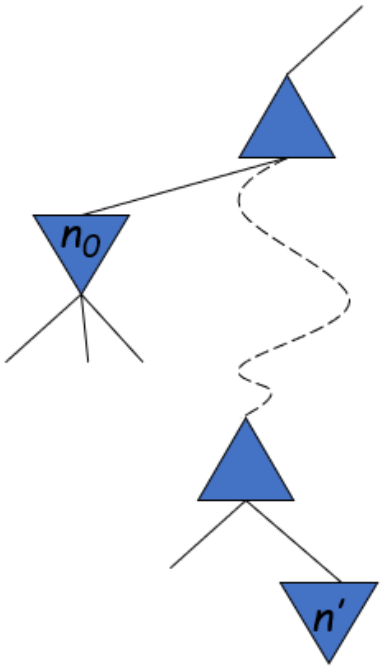
```
MAX-VALUE(nodo, α, ß)
  if CUTOFF-TEST(nodo) then return EVAL(nodo)
  v ← - ∞
  for all figli in SCS(nodo) {
          v ← max(v, MIN-VALUE(figlio, α, ß) )
          if v ≥ ß then return v
          α ← max(v,α)
  }
  return v
```

v = *utilità del nodo*

```
MIN-VALUE (nodo, α, ß)
  if CUTOFF-TEST(nodo) then return EVAL(nodo)
  v ← + ∞
  for all figli in SCS(nodo) {
          v  ← min(v, MAX-VALUE(s, α, ß))
          if v ≤ α   then return v
          ß ← min(v,ß)
  }
  return v
```

Alfa-beta pruning

Caso generale:
Se n_0 è migliore di n' allora n' non verrà mai raggiunto durante il gioco e quindi tutto il sottoramo corrispondente può essere potato.



Alfa-beta pruning

Efficacia della potatura:

- Dipende dall'**ordinamento dei nodi**
  - Ordinamento migliore: $O(b^{\frac{1}{2}m})$
  - Ordinamento pessimo: $O(b^m)$
  - Ordinamento medio: $O(b^{\frac{3}{4}m})$

Nel caso degli scacchi:

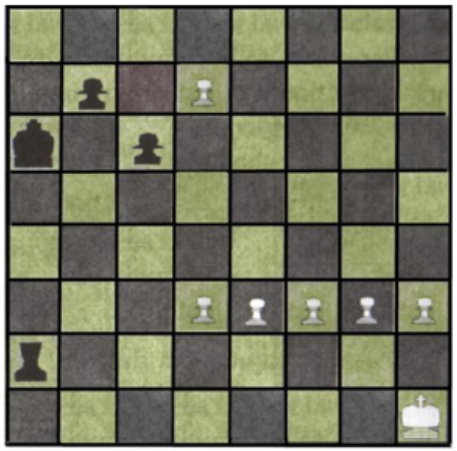
- Fasi di apertura (b $\approx$ 35, poniamo m $= 10$)
  - Minimax: n. nodi: ca. 2.700.000.000.000.000
  - Alfa-beta: n. nodi: ca. 380.000.000.000
- Fasi centrali (b $\approx$ 18, poniamo m $= 10$)
  - Minimax: n. nodi: ca. 3.500.000.000.000
  - Alfa-beta: n. nodi: ca. 2.600.000.000
- $\mapsto$ Un buon calcolatore (10^6 mosse/sec) sceglie una mossa in 4 minuti!

Altri problemi da affrontare

- Problema dell'orizzonte
- Eccessiva fiducia nell'euristica
- Eventi stocastici
- Giochi multiplayer
- Branching Factor e potenza di calcolo

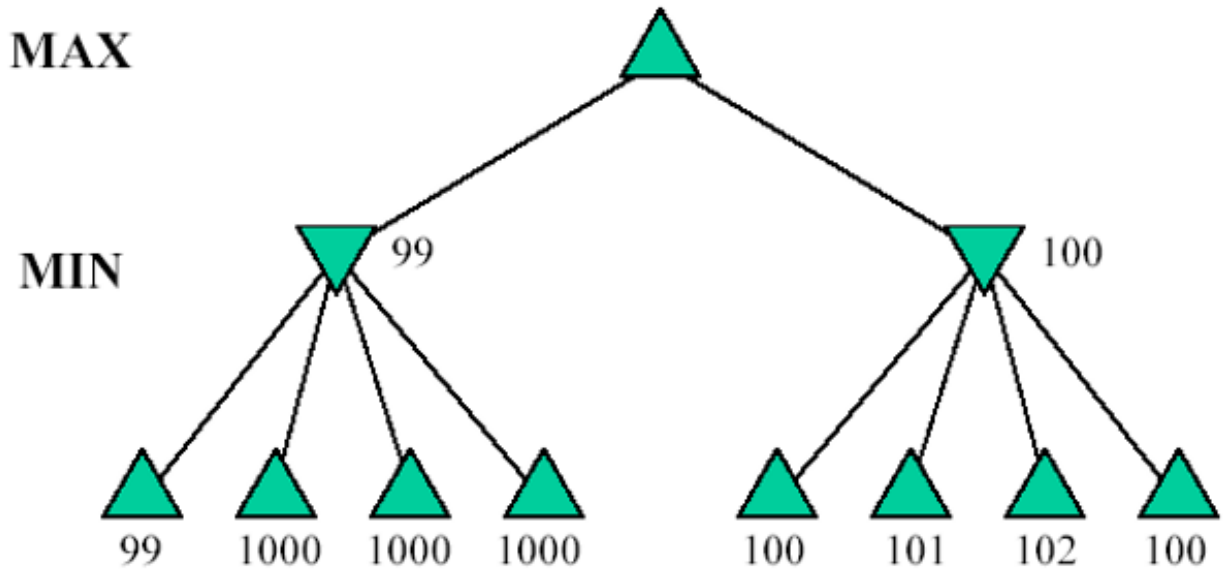
Problema dell'orizzonte

- Un lungo periodo di quiescenza può precedere un rapido ed inevitabile peggioramento dell'utilità
- Se il taglio in profondità è avvenuto in questa "zona", valuta positivamente uno stato che è invece disastroso



Eccessiva fiducia nell'euristica

- Una valutazione molto irregolare tra nodi "fratelli" è rischiosa, soprattutto usando Alpha-Beta
- Servirebbe un'ulteriore ricerca nel sottoramo per accertarsi della bontà della valutazione



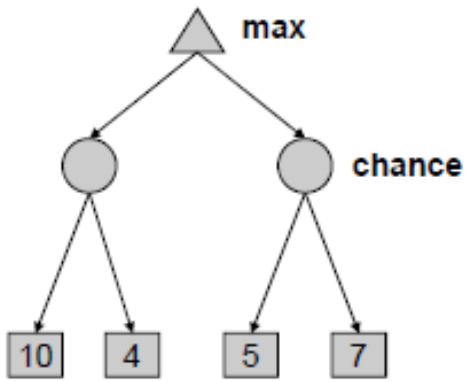
Alberi di ricerca ExpectiMAX

E se non sappiamo quale sia il risultato di un'azione sarà? Per esempio:

- In solitario, la carta successiva è sconosciuta
- In pacman, i fantasmi agiscono in modo casuale

Possiamo applicare un ricerca expectiMAX

- cambia i nodi, come i nodi min, eccetto che l'outcome è casuale
- calcola le utilità attese
- scegli i nodi max come in minimax
- i nodi cambiati prendono la media (aspettativa/expectation) di valore dei figli



Utilità massima attesa

Principio di massima utilità attesa: un agente dovrebbe scegliere l'azione che massimizza la sua utilità attesa, data la sua conoscenza

- Principio generale per il processo decisionale
- Spesso preso come definizione di razionalità

Un richiamo del concetto di Probabilità:

- oggettivista / frequentista:
  - Medie su esperimenti ripetuti
  - Per esempio. stima empiricamente P(pioggia) dall'osservazione storica
  - Affermazione su come andranno gli esperimenti futuri (nel limite)
  - Nuove evidenze cambiano la classe di riferimento
  - Fa pensare a eventi intrinsecamente casuali, come tirare i dadi
- soggettivista/bayesiana:
  - Gradi di convinzione sulle variabili non osservate
  - Per esempio, la convinzione di pacman che il fantasma girerà a sinistra, dato lo stato
  - Impara spesso le probabilità dalle esperienze passate (più avanti)
  - Nuove prove aggiornano le convinzioni (più avanti)

Valore atteso e Utilità

Il **valore atteso** (expectation) di una funzione è il suo output medio, ponderato da una data distribuzione sugli input

Le **utilità** sono funzioni dai risultati (stati del mondo) a numeri reali che descrivono le preferenze di un agente.

- In una partita, può essere semplice (+1/-1)
- Riepilogano gli obiettivi dell'agente
- Teorema: qualsiasi insieme di preferenze tra i risultati può essere riassunto come funzione di utilità (a condizione che le preferenze soddisfino determinate condizioni)

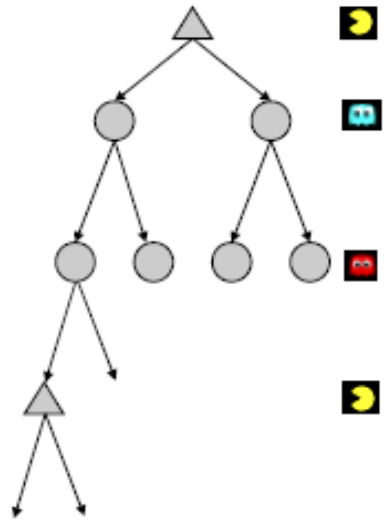
In generale, scriviamo algoritmi in cui le utilità sono implicite e invece emergono le azioni.
Perché, quindi, non lasciamo gli agenti decidere le loro utilità?

Ricerca ExpectiMAX

Nella ricerca Expectimax, abbiamo a modello probabilistico di come la volontà dell'avversario (o dell'ambiente) può comportarsi in qualsiasi stato

- Il modello potrebbe essere una semplice distribuzione uniforme (tira un dado)
- Il modello potrebbe essere sofisticato e richiedono una grande quantità di calcolo
- Abbiamo un nodo per ogni risultato fuori dal nostro controllo: avversario o ambiente
- Il modello potrebbe portare a probabili azioni contraddittorie!

Per ora, supponiamo per qualsiasi stato di avere magicamente una distribuzione per assegnare le probabilità alle azioni dell'avversario/risultati ambientali



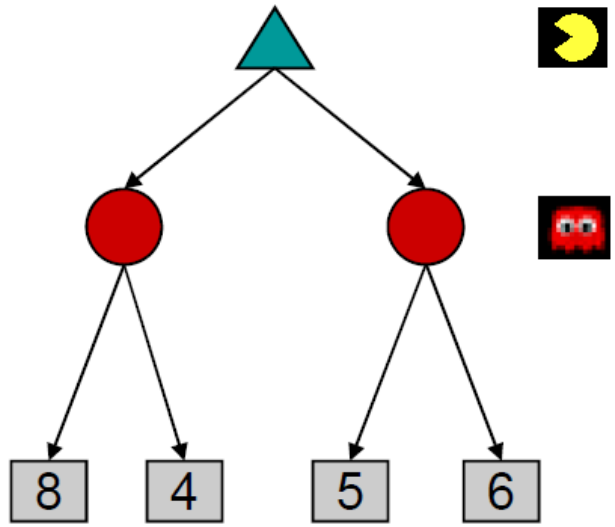
Avere una convinzione probabilistica su l'azione di un agente non significa quell'agente sta lanciando delle monete!

Ricerca ExpectiMAX

```
def value(s)
    if s is a max node return maxValue(s)
    if s is an exp node return expValue(s)
    if s is a terminal node return evaluation(s)

def maxValue(s)
    values = [value(s') for s' in successors(s)]
    return max(values)

def expValue(s)
    values = [value(s') for s' in successors(s)]
    weights = [probability(s, s') for s' in successors(s)]
    return expectation(values, weights)
```



Giochi multiplayer

- Possiamo generalizzare gli algoritmi per giochi "2-player-perfect-information":
  - Requisito: non ci deve essere accordo tra i giocatori

- Esempio 1: "la dama cinese"
  - 6 giocatori muovono a turno
  - ogni giocatore cerca di occupare completamente l'angolo opposto

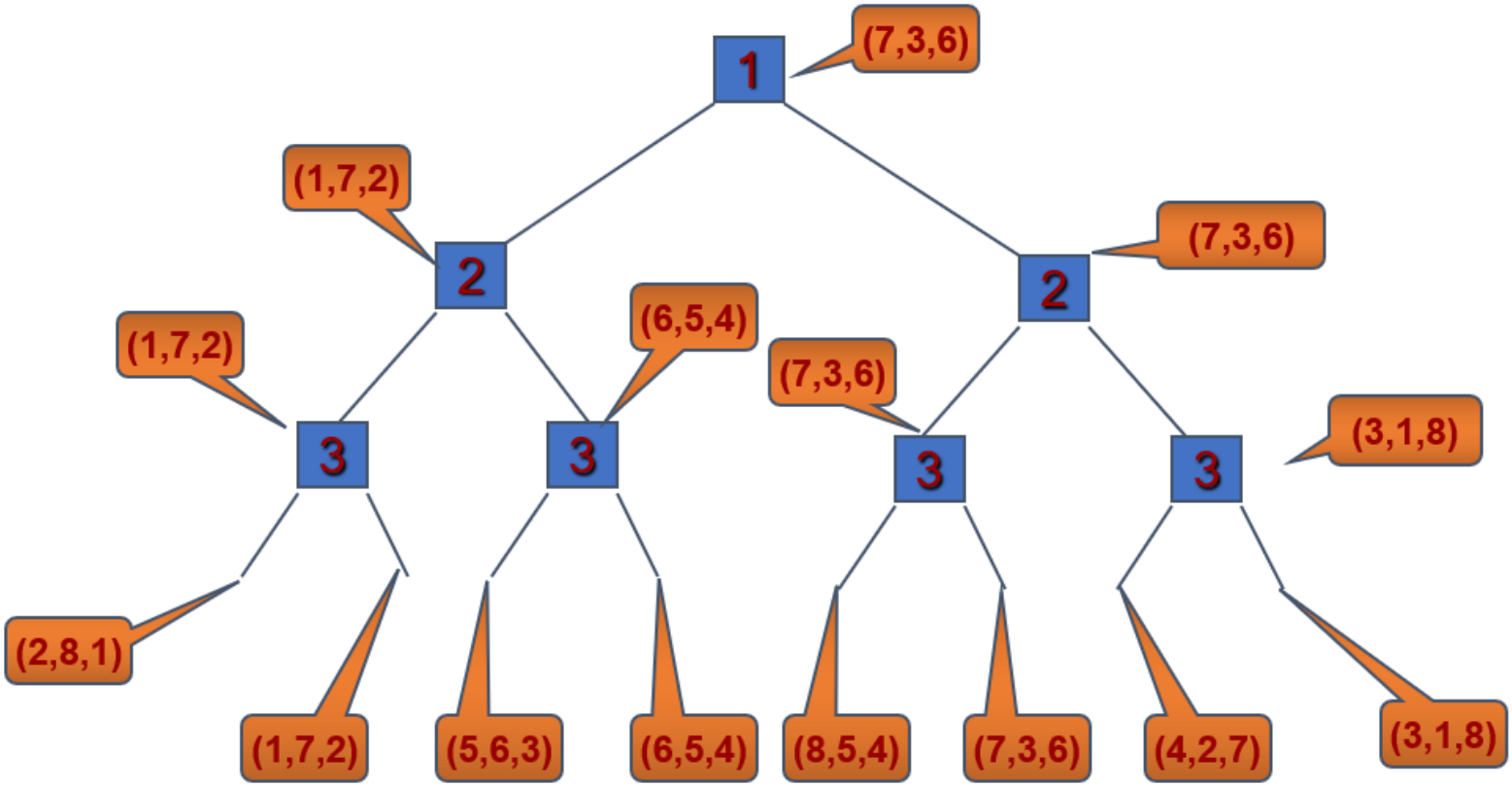
- Esempio 2: "3-players Othello"
  - 3 giocatori muovono a turno
  - ogni giocatore deve "conquistare" il massimo della scacchiera

Algoritmo MAX^N

- Assunzioni:
  - I giocatori muovono a turni
  - Ogni giocatore mira a massimizzare il proprio utile
  - Ogni giocatore è indifferente all'utile degli avversari
- Funzione di valutazione:
  - Restituisce una n-tupla di valori di utilità attesa, uno per ogni giocatore (player p) allo stato di gioco s
  - $< U(p_1, s), U(p_2, s), \ldots, U(p_n, s) >$
  - Esempio: in Reversi/Othello si possono calcolare il numero di pezzi per ogni giocatore
- Algoritmo:
  - Depth-first search come Minimax
  - Fai risalire la n-tupla che massimizza $U(p_n)$ quando muove p_n

Minimax è un caso speciale di MAX^N in cui: $N = 2$ e la funzione di valutazione restituisce la tupla $< x, -x >$.

Algoritmo MAX^N



Algoritmo Paranoid

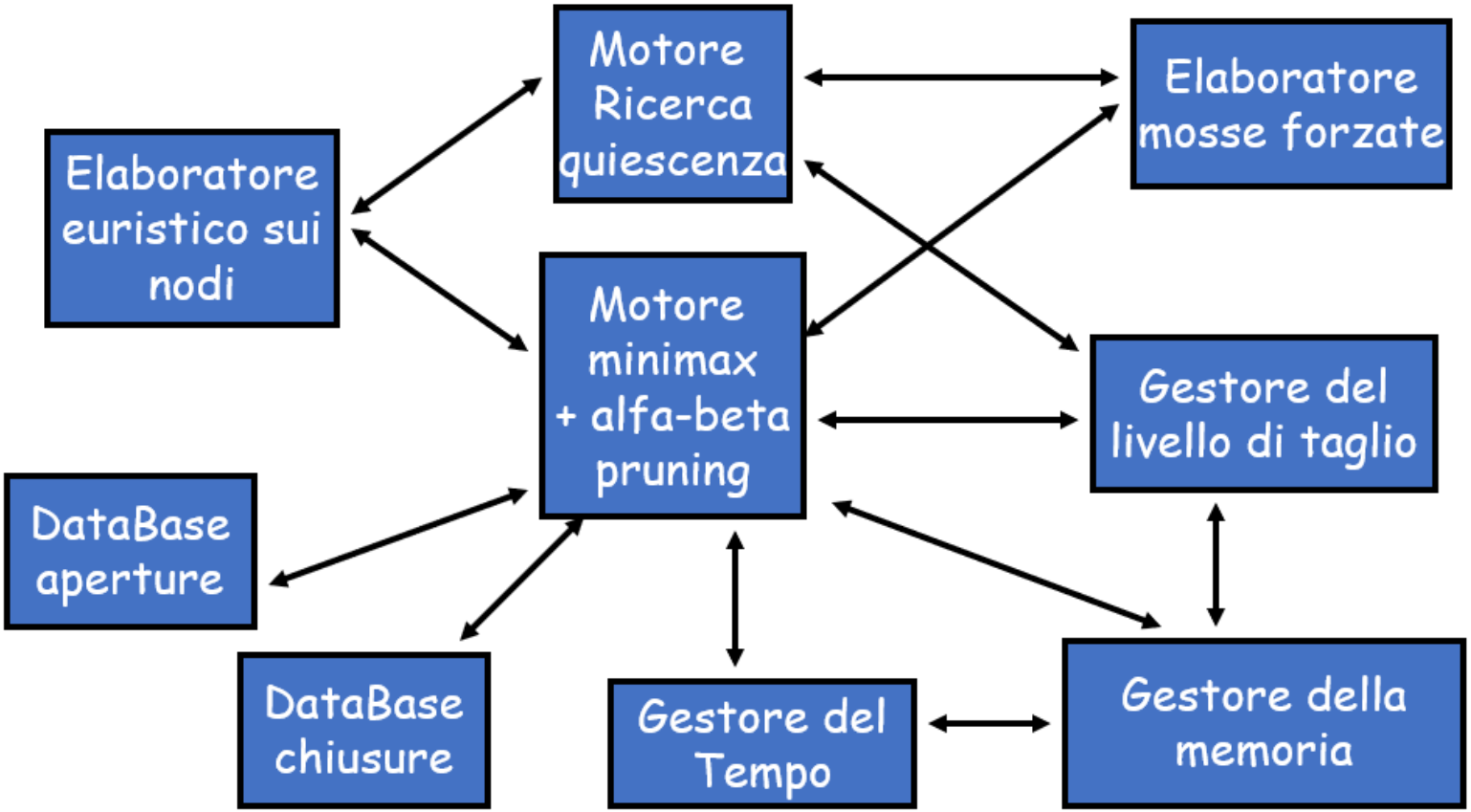
- Idea: gli altri giocatori sono come 1 solo "macro-avversario"
  - 2 giocatori: MAX (noi), MIN (avversari)
- Valutazione dei nodi dell'albero:
  - Quando tocca a MAX si massimizza l'utilità di MAX
  - Quando tocca ad 1 avversario si minimizza l'utilità di MIN

- Paranoid permette di rimuovere l'assunzione di non-accordo tra i giocatori
- Paranoid ha minori tempi di esecuzione
- Paranoid si può sposare meglio con Alfa-Beta pruning
- Paranoid non dà la garanzia di MaxN di che MAX massimizzi il suo utile finale
- Il branching factor è comunque un problema!

La vera sfida

- La vera sfida è competere con l'uomo ad armi pari.
- L'uomo non usa la ricerca come metodo principale:
  - Prima parte dai GOAL (non ben definiti)
  - A ritroso costruisce SOTTOGOAL
  - pianifica: azioni $\mapsto$ sottogoal $\mapsto$ goal
  - Usa la ricerca per raggiungere obiettivi locali
  - Ha capacità "istintive" di escludere le scelte inutili: riduce enormemente il branching factor
- Come interfacciare ragionamento goal-oriented e search?

Un giocatore di sacchi artificiale



Apprendimento per rinforzo (RL)

L'**apprendimento per rinforzo** (o reinforcement learning RL) è una tecnica di apprendimento automatico che punta a realizzare **agenti autonomi** in grado di scegliere **azioni** da compiere per il conseguimento di determinati **obiettivi** tramite interazione con l'**ambiente** in cui sono immersi.

Abbiamo quindi un agente e un ambiente dentro cui tale agente è immerso.

L'ambiente fornisce **ricompense** e un nuovo stato basato sulle azioni dell'agente.

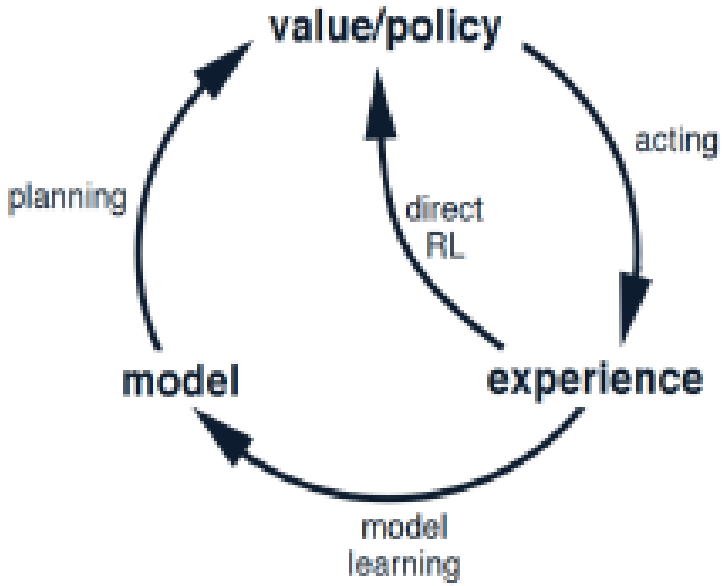
Quindi, nell'apprendimento per rinforzo, non insegniamo a un agente come dovrebbe fare qualcosa, ma gli presentiamo ricompense positive o negative in base alle sue azioni.

Apprendimento per rinforzo

Due **paradigmi** principali:

- **model-free**: imparare una funzione a valori numeri o una politica dettata dall'esperienza
- **model-based**: impara un modello dall'esperienza e poi usalo per pianificazione e recitazione.

Relazione tra leraning, planning e acting:



Model learning (ML)

Per modello intendiamo un MDP (Markov Decision Process) $< S, A, R, T, \gamma >$ parametrizzato secondo ν.

assumiamo che lo **spazio degli stati** S e lo **spazio delle azioni** A sono conosciuti.

Inoltre, assumiamo che vi sia una indipendenza condizionale tra le transazioni e le ricompense:

$$P[s_{t+1}, r_{t+1}|s_t, a_t] = P[S_{t+1}|s_t, a_t] \cdot P[R_{t+1}s_t, a_t]$$

Model learning (ML)

Imparare il modello è un processo di apprendimento supervisionato che consiste nell'apprendere

- la **funzione di ricompensa** $R(.|s,a)$ tramite regressione
- la **distribuzione di transizione** $P(.|s,a)$ come un problema di stima della densità (funzione di probabilità)

scegliamo una famiglia adatta di **modelli parametrizzati**, questi possono includere modelli di ricerca tabellare, aspettativa lineare, gaussiana lineare, processo gaussiano, rete di credenze profonde ecc.

Successivamente scegliamo una **funzione di perdita** (loss) appropriata, ad es. errore quadratico medio, divergenza KL ecc. per ottimizzare la scelta dei parametri (del modello) che minimizzano tale perdita.

Ricerca basta sulla simulazione

Dato un modello approssimato o esatto del mondo, i metodi di ricerca basata sulla simulazione cercano di identificare l'**azione migliore** da intraprendere in base alla simulazione in avanti. Viene costruito un **albero di ricerca** con lo stato corrente come radice e gli altri nodi generati utilizzando il modello acquisito.

Tali metodi possono dare grandi risparmi in quanto non è necessario risolvere il tutto MDP ma solo un MDP **subottimale** a partire dallo stato corrente.

Dopo aver acquisito tale esperienza simulata, possiamo utilizzare un approccio method-free come ad esempio i metodi **Monte-Carlo** (C).
Nello specifico, dato un modello M ed una politica di simulazione π, in un algoritmo di ricerca MC per ogni **azione** $a \in A$ simuliamo K **episodi** seguendo π.
$Q(s_t, a)$ è il valore calcolato come la **media delle ricompense** di tale simulazione. Quindi, vine scelta l'azione che **massimizza** $Q(s_t, a)$.

Monte-Carlo tree search

E' una famiglia di algoritmi basata su due **principi**:

- il vero valore di uno stato può essere stimato usando una media dei valori di una **simulazione random**
- tali valori possono essere utilizzati per **aggiustare iterativamente** le politiche di scelta in una strategia **best-first** permettendoci di concentrarci su regioni dello spazio di ricerca a rendimento/valore migliore

In ogni **nodo** memorizziamo delle **statistiche**:

- conteggio della variazione totale $N(s)$
- uno specifico conteggio per ogni coppia $N(s, a)$
- i valori stimati dal monte-Carlo $Q(s, a)$

Monte-Carlo tree search

Si costruisce **progressivamente** un albero di ricerca **parziale** utilizzando il nodo corrente come radice di un nuovo sottoalbero.

Ogni iterazione è divisa in quattro **fasi**:

- **selezione**: a partire dal nodo corrente, selezioniamo ricorsivamente i suoi figli fino ad un nodo foglia non terminale
- **espansione**: il nodo foglia non terminale è aggiunto all'albero di ricerca
- **simulazione**: lanciamo delle simulazioni da tale nodo per produrre una stima degli esiti (outcomes)
- **retropropagazione** (backdrop): i valori ottenuti dalla simulazione sono retropropragati nel percorso dalla foglia selezionata alla radice dell'albero aggiornando le statistiche dei nodi su tale percorso

Monte-Carlo tree search

Algorithm 1 General MCTS algorithm

```
1: function MCTS(s_0)
2:     Create root node v_0 corresponding to s_o
3:     while within computational budget do
4:         v_k ← TreePolicy(v_o)
5:         Δ ← Simuation(v_k)
6:         Backprop(v_k, Δ)
       return arg max_a Q(s_o, a)
```

Algorithm 2 Greedy Tree policy

```
1: function TreePolicy(v)
2:     v_next ← v
3:     if |Children(v_next)| ≠ 0 then
4:         a ← arg max_{a∈A} Q(v, a)
5:         v_next ← nextState(v, a)
6:         v_next ← TreePolicy(v_next)
       return v_next
```

Tipicamente, le varianti dell'algoritmo descritto precedente, modificano due politiche principali per migliorare l'utilizzo della memoria, per una potatura migliore o per avere statistiche più complesse:

- **tree policy** per scegliere le azioni da poter applicare in ogni nodo e quindi le loro statistiche
- **Roll out policy** per come eseguire le simulazioni nei nodi foglia

Nell'esempio, viene applicata una strategia greedy per la selezione dei nodi nello stadio iniziale e una politica random per la simulazione.

Monte-Carlo tree search

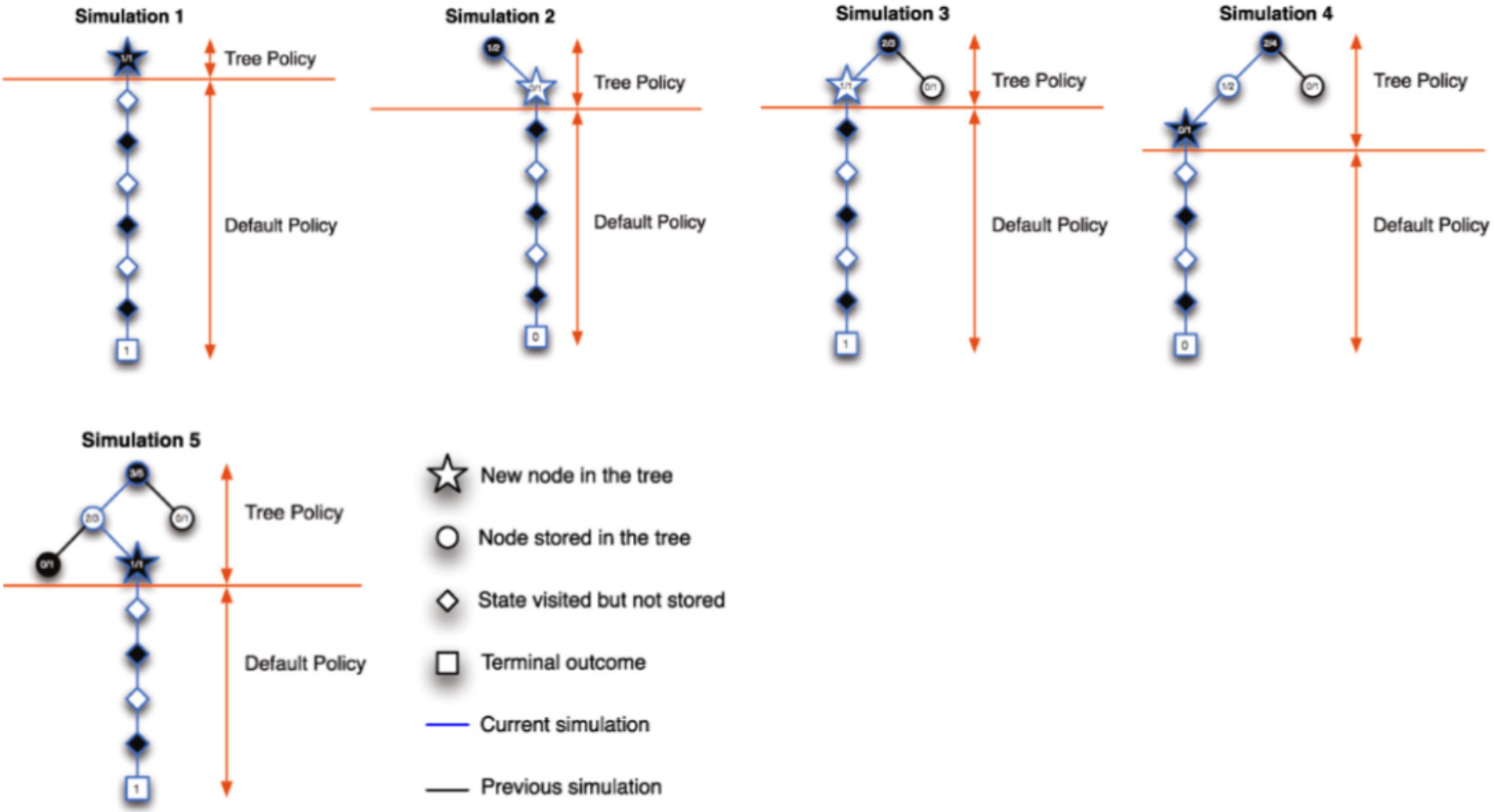
Partiamo dal nodo radice e simuliamo una traiettoria che ci dà una ricompensa di 1 in questo caso.

Usiamo quindi la politica dell'albero che seleziona in modo greedy il nodo da aggiungere all'albero e successivamente simuliamo un episodio da esso utilizzando il criterio di simulazione (default).

Questo episodio ci dà una ricompensa di 0 e aggiorniamo le statistiche nei nodi dell'albero.

Questo processo è quindi ripetuto aggiornando le statistiche e seguendo la politica specifica per la selezione del prossimo nodo dell'albero (tree policy).

Monte-Carlo tree search



Monte-Carlo tree search

Vantaggi nell'utilizzare MCTS:

- la struttura da albero è massivamente **parallelizzabile**
- sia ha una **valutazione dinamica** degli stati: risolviamo il problema dallo stato corrente in avanti
- spesso non abbiamo bisogno di conoscenza specifica del dominio: molto utilizzabile come **black-box** in quanto ha solo bisogno di **campioni** per raccogliere le statistiche
- combina efficientemente **plannign** e **sampling** per eliminare il probllema della curse of dimensionality

Upper confidence tree search

La politica greedy può portare a delle scelte molto **subottimali** che portano l'algoritmo a evitare **azioni** con un solo o pochi non significativi **risultati negativi**.
Questo fenomeno è dovuto essenzialmente alla **incertezza** circa il valore/probabilità reale di tali azioni negative.

Per risolvere tale problema si può applicare il principio di **ottimalità sotto incertezza** al MCTS usando UCB (limite superiore di confidenza).

Invece di selezionare l'azione in modo greedy, si seleziona l'azione che massimizza il **limite superiore di confidenza** il cui valore è dato da

$$Q(S,a) + \sqrt{\frac{2log \cdot N(s)}{N(s,a)}}$$

Algorithm 3 Upper Confidence Tree policy

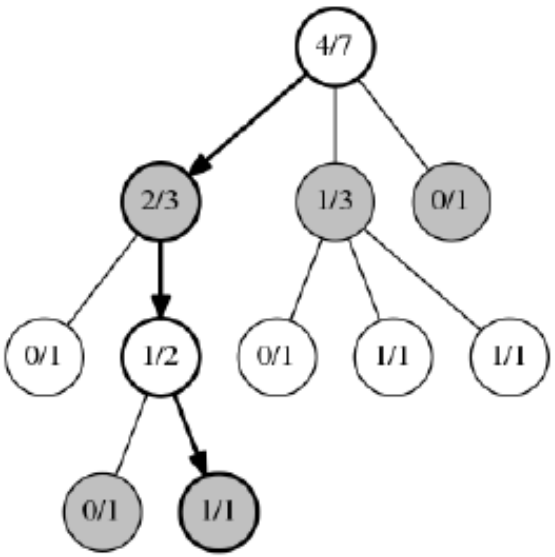
1: **function** TREEPOLICY(v)
2: $v_{next} \leftarrow v$
3: **if** $|Children(v_{next})| \neq 0$ **then**
4: $a \leftarrow arg\max_{a \in A} Q(v,a) + \sqrt{\frac{2 \log N(v)}{N(v,a)}}$
5: $v_{next} \leftarrow nextState(v,a)$
6: $v_{next} \leftarrow TreePolicy(v_{next})$
return v_{next}

GO

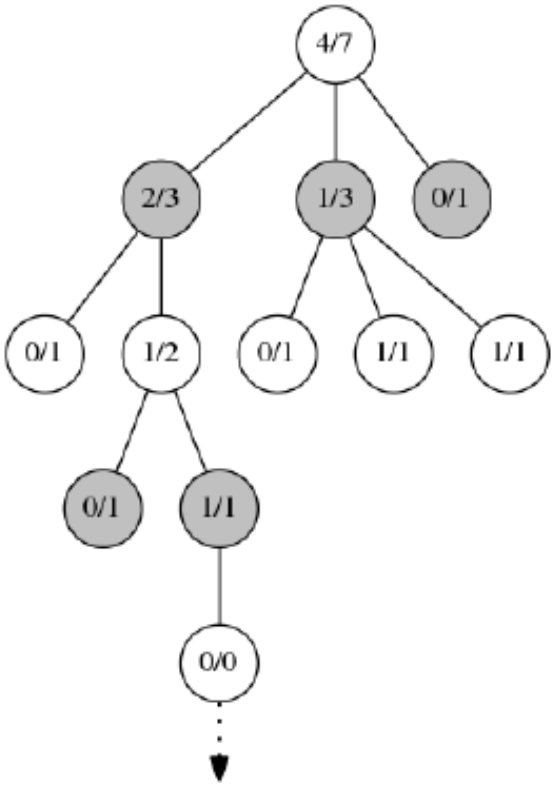
Go è un gioco a due giocatori (B/N), su una scacchiera 19×19.
Il nero e il bianco posizionano alternativamente le pietre sulla scacchiera.
Il principale l'obiettivo del gioco è circondare e conquistare il territorio.
Inoltre pietre circondate dall'avversario vengono rimossi.

La funzione di ricompensa più semplice dà una ricompensa di $+1$ se il Nero vince e 0 se il Bianco vince negli stati terminali e 0 altrove.
Di conseguenza, l'obiettivo del giocatore nero è massimizzare la ricompensa e il bianco cerca di minimizzarlo.

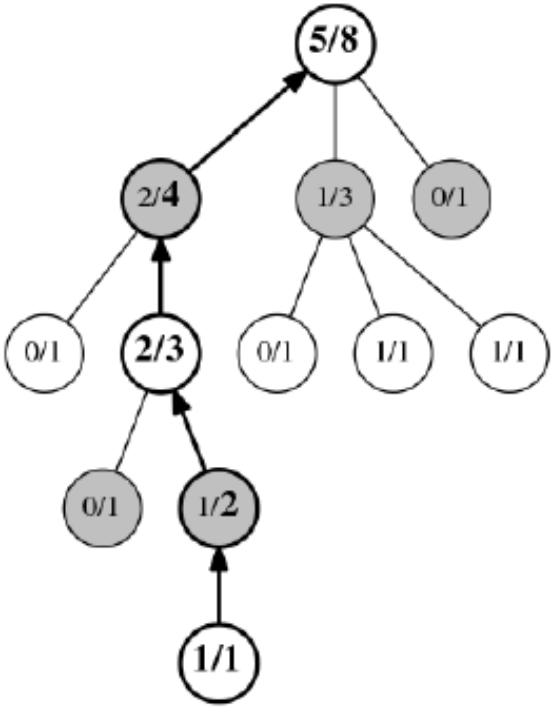
AlphaGO utilizza UCB e memorizza le probabilità di vittoria di entrambi i giocatori in ogni nodo dati dalle statistiche partite perse e partite vinte.



(a) Tree Phase



(b) Rollout Phase

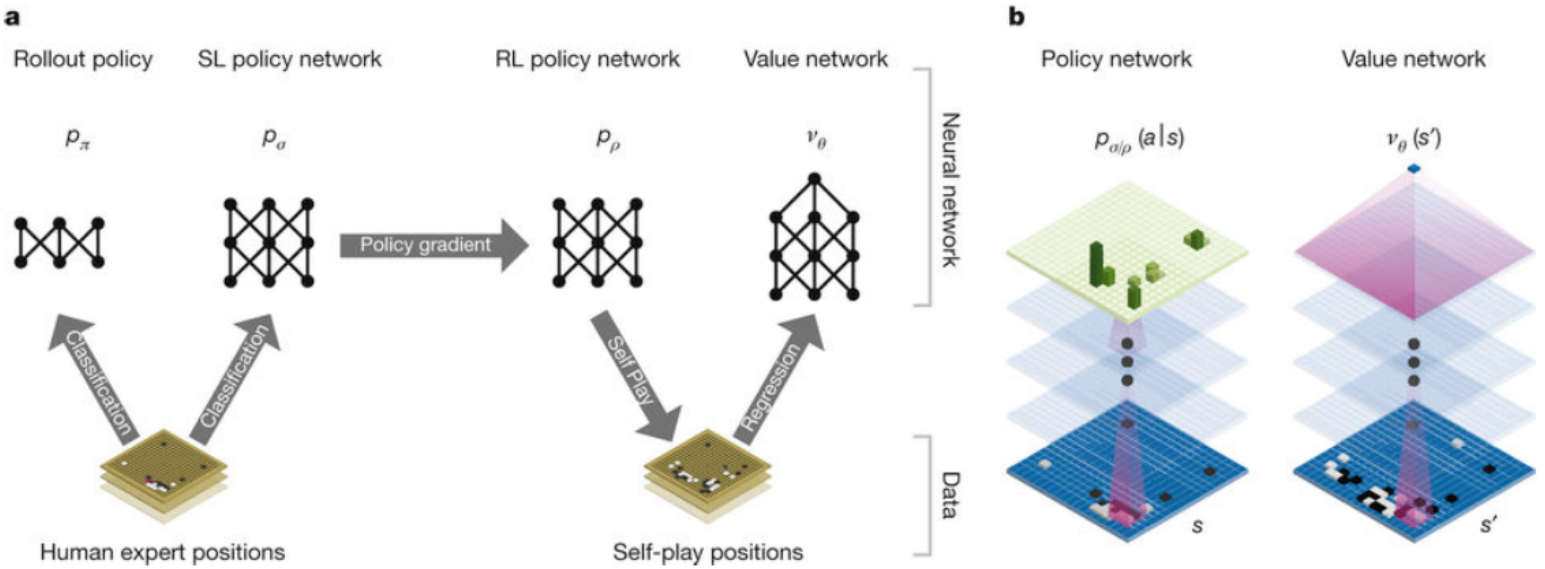


(c) Update Statistics

AlphaGO

AlphaGo [1] ha utilizzato una "deep network" di policy per la fase di roll-out, questo consente alle simulazioni di essere molto più realistiche rispetto al semplice utilizzo di rollout casuali.

Anche in un gioco complesso come Go non è fattibile simulare fino al completamento, viene utilizzato l'arresto anticipato insieme a una rete Value per ottenere la probabilità di vittoria.



Di recente è stato proposta AlphaGO Zero in cui si una una sola network e i "sample" sono ottenuti dall'algoritmo giocando con se stesso.