Università degli Studi di Salerno

Ingegneria Informatica LM-32

Secondo Anno

Progetto di agenti intelligenti per Hnefatafl–Tablut

Author:
Antonino Durazzo
Francesco de Pertis

Teacher:
Angelo Marcelli
Antonio Della Cioppa

June 2020

Indice

1 Introduzione

In questa relazione, ci si è posto l'obiettivo di sviluppare un agente capace di massimizzare le probabilità di vittoria in una partita di Tablut, una variante finlandese del classico gioco vichingo chiamato Hnefatalf.

Si è deciso di utilizzare il framework rilasciato dall'Università pubblica di Montreal (Canada) in forma di challenge per uno dei suoi corsi di Computer Science. Il framework è sviluppato in linguaggio Java e racchiude tutte le funzionalità del gioco con annessa interfaccia grafica, che permette all'utente di selezionare gli agenti di entrambe le parti, artificiali o umani che siano. Tale framework è disponibile su sito github.com al seguente url: `https://github.com/kiankd/comp424`.

In particolare, il nostro lavoro è consistito nell'implementare un package aggiuntivo contente le classi necessarie al funzionamento dell'agente.

Dopo questa breve introduzione, verranno spiegate le regole del gioco, seguite dalla progettazione dell'agente e dalle euristiche utilizzate. Infine, verranno mostrati i risultati delle prestazioni riscontrate nei test, i suoi possibili sviluppi futuri e le conclusioni alle quali si è giunti.

2 Gioco, Regole e Proprietà

Il Tablut è una variante del gioco Vichingo Hnefatafl di cui esistono variati diverse delle regole del gioco. In questo lavoro seguiremo le regole presentate nelle successive sottosezioni.

2.1 Terminologia e Configurazione iniziale

Il gioco è composto da due giocatori, gli *Swedes* (bianchi) e i *Muscovites* (neri), e si gioca su una tavola 9x9 la cui configurazione iniziale è presentata nella Figura 1.

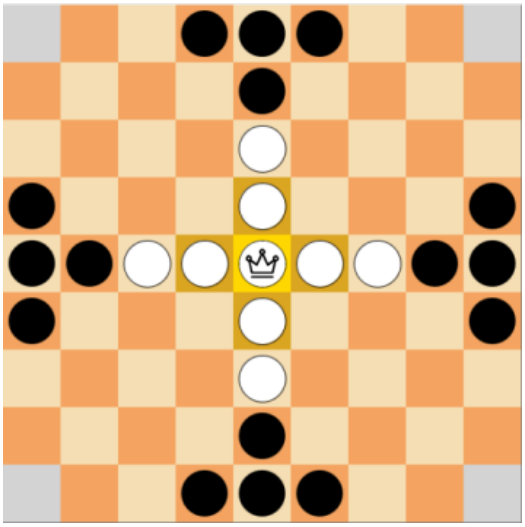

Figura 1: Configurazione iniziale. La cella centrale, colorata in giallo, è chiamata *royal citadel*, o *castle*, o *throne* in cui risiederà il *King* degli *Swedes*. Le celle, colorate in grigio, sono chiamate *escape cells*. Su ciascun lato, sono presenti 4 gruppi di 4 celle disposti a forma di T che sono chiamati *citadels* o *camps* in cui risiederanno i *Muscovites*. Gli *Swedes* risiederanno allineati lungo ciascun lato del *King*. Il Numero di pedine, a disposizione per ogni squadra, sarà quello in figura.

Il *King* degli *Swedes* è l'unica pedina a essere marcata e le restanti pedine prendo il nome di *Soldiers*. L'obiettivo dei *Muscovites* è catturare il *King* degli *Swedes* mentre gli *Swedes* hanno come obiettivo quello di proteggere il proprio *King* e per fare ciò il re deve raggiungere una delle *escape cells*. Il giocatore che raggiunge per primo il suo obiettivo vince. Le condizioni per la terminazione del gioco saranno discusse in dettaglio nel sottoparagrafo 1.4.

2.2 Movimenti e Cattura

I due giocatori alternano i loro turni, che consistono in un singolo movimento di una pedina. Una pedina può essere spostata lungo una singola linea retta, in orizzontale o in verticale, per un qualsiasi numero di celle. Il movimento non deve passare né finire in una cella occupata da un'altra pedina e a ognuna di esse, ad eccezione del *King*, non è consentito spostarsi sul *throne* o sulle *escape cells*. Per effettuare una cattura che comporta la rimozione di una o più pedine dal gioco, un giocatore deve muovere un suo pezzo in modo da circondare un singolo pezzo avversario. Una pedina è considerata circondata secondo diversi criteri:

- Quando il *King* degli *Swedes* è nel *castle*, viene considerato circondato se ci sono pedine nemiche su tutti e quattro i suoi lati.
- Quando il *King* è adiacente al *throne*, viene considerato circondato se ci sono tre pedine nemiche su tre dei suoi lati e il castello sul quarto.
- Quando un pedina è adiacente al *throne* o alle *escape cells*, a eccezione se ci si ritrova in una delle due condizioni precedenti, essa risulta essere circondata se nella cella adiacente lungo la componente verticale o orizontale è presente una pedina nemica.
- In ogni altro caso, qualsiasi pedina è considerata circondata se ci sono due pedine nemiche su due lati opposti della sua cella, in modo che i tre pezzi siano allineati in orizzontale o in verticale.

La cattura può avvenire solo in modo attivo, il che significa che se un giocatore sposta il proprio pezzo in modo da renderlo circondato, il pezzo non viene catturato. Infine la cattura risulta valida se e solo se una singola pedina viene catturata lungo un lato ed è possibile catturare più pedine (fino a 3) con una sola mossa, se quella mossa consente di circondare più di una singola pedina su più lati.

2.3 Terminazione del gioco

Il gioco termina quando viene soddisfatta una delle seguenti condizioni:

- Il *King* degli *Swedes* raggiunge una delle *escape cells*. Vittoria assegnata ai bianchi.

- I *Swedes* catturano tutti i *Muscovites*. Vittoria assegnata ai bianchi.
- Il *King* degli *Swedes* viene catturato. Vittoria assegnata ai neri.
- Se i due giocatori raggiungono 40 mosse e nessuna delle condizioni precedenti si è verificata allora il gioco si conclude con un pareggio.

Vincoli del framework:

- Se un giocatore dopo 30 secondi non ha ancora effettuato una mossa la vittoria viene assegnata all'avversario.
- Il processo relativo al giocatore può utilizzare solamante 500 mb di memoria. Se eccede questa capacità durante l'esecuzione di una mossa la vittoria viene assagnata all'opponente.

2.4 Proprietà

In qualsiasi istante del partita i due giocatori sanno tutto sullo stato del gioco. Dato uno stato del gioco, è anche noto come ogni possibile mossa lo cambierà, poiché non ci sono componenti casuali. Infine, i due giocatori hanno posizioni di partenza, pedine e obiettivi diversi. Quindi questo è un gioco a conoscenza perfetta, deterministico e perfettamente asimmetrico.

3 Progetto dell'agente

3.1 Tipo di agente

In questo capitolo ci poniamo il traguardo di progettare un agente razionale in grado di perseguire gli obiettivi di una qualsiasi delle squadre. Pertanto il primo passo per la corretta progettazione di un agente è l'individuazione del problema da risolvere che significa caratterizzare l'ambiente in cui l'agente opera. Per fare ciò useremo la sua descrizione *PEAS (Performance, Environment, Actuators, Sensors)*.
La performace risulta essere massimizzare la propria probabilità di vittoria. I sensori e gli attuatori sono forniti direttamente da funzioni interne del programma di gioco che permettono di controllare tutte le posizioni delle pedine, sia funzioni che permettono a una pedina di eseguire una nuova azione. L'ambiente in cui l'agente si trova ad operare è:

- **Completamente osservabile**.
  I sensori dell'agente danno accesso allo stato completo dell'ambiente in ogni momento, quindi il *task environment* è completamente osservabile in virtù del fatto che i sensori rilevano tutti gli aspetti rilevanti per la scelta dell'azione.

- **Deterministico**.
  Il prossimo stato dell'ambiente è completamente determinato dallo stato corrente e dall'azione eseguita dall'agente.

- **Strategico**.
  L'ambiente risulta deterministico in tutto tranne che per le azioni degli altri agenti (avversario).

- **Sequenziale**.
  Lo stato corrente dipende dalle azioni prese in precedenza.

- **Statico**.
  L'ambiente non cambia mentre un agente sta deliberando. Se introducessimo vincoli di tempo sull'azione da effettuare per ogni squadra, allora risulterebbe che il punteggio delle prestazioni cambierebbe e l'ambiente risulterebbe *semidinamico*.

- **Discreto**.
  L'ambiente ha un numero finito di stati distinti e il comportamento

dell'agente può essere visto come una successione discreta di percezioni e azioni.

- **Multiagagente**.
  L'agente da modellare gioca contro un altro giocatore che cerca di minimizzare la propria misura di prestazione.

L'agente sviluppato è un agente basato su utilità. La scelta di realizzare un agente di questo tipo ci viene in aiuto nel realizzare un agente più funzionale al gioco infatti è in grado di decidere razionalmente verso quali obiettivi alternativi muoversi. Pertanto si è presentata la necessità dell'introduzione di una funzione di utilità poiché sia per gli *Swedes* che per i *Muscovites* si deve cercare di raggiungere il proprio obiettivo, pur ritrovandosi in situazioni in cui bisogna cercare di proteggere il re per gli *Swedes*. *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 quanto sarebbe contento l'agente se riuscisse a raggiungerli. Nella terminologia corrente si dice che uno stato del mondo preferibile a un altro ha, per l'agente, maggiore utilità. La funzione di utilità assegna a uno stato un numero reale che quantifica il grado di contentezza a esso associato* [2]. Una misura di utilità è molto vantaggiosa in questo caso, poiché più obiettivi possono essere raggiunti, come massimizzare la propria utilità di vittoria e salvaguardare la sicurezza del re, per esempio quando e come muovere il re può significare essere catturati dal nemico. L'agente possiede una funzione di utilità di cui cerca di massimizzare il valore, la quale verrà accuratamente approfondita nella sezione 4.2.

3.2 Scelta dell'algoritmo

Per affrontare il problema della ricerca in uno spazio degli stati che rappresenta un mondo dinamico nel senso che il mondo sarà modificato a seguito di azioni compiute da altri agenti intelligenti con una razionalità pari all'agente che stiamo implementando e che tenderanno a modificare lo stato del mondo per conseguire obiettivi che si intedono essere contrastanti con quelli dell'agente che stiamo implementando e considerando l'assenza di elementi di causalità nel gioco, la scelta dell'agoritmo è ricaduta sulla variante $\alpha - \beta$ *pruning* dell'algoritmo decisionale *MiniMax* con un *cutoff test* in profondità per poi stimare l'utilità di vittoria dei nuovi stati terminali del taglio effettuato. La scelta di effetturare anche un *cutoff test* profondità verrà discussa nella sottosezione 4.1.1
Di seguito è mostrato uno pseudocodice dell'algoritmo sopra esplicato.

Algorithm 1 $\alpha - \beta$ pruning algorithm

```
function ALPHA-BETA-SEARCH(gameState)
    for move in tablutLegalMoves do
        cloneGameState ← gameState.update(move)
        utility ← minimax(cloneGameState, depth,α,β,false)
        if utility > maxUtility then
            maxUtility ← utility
            bestMove ← move
     return bestMove

function MINIMAX(gameState, depth,α,β,maximizingPlayer)
    if Cut-Test( depth) then
        return Eval(gameState)
    if maximizingPlayer then
        best ← −∞

        for move in tablutLegalMoves do
            cloneGameState ← gameState.update(move)
            utility ← MINIMAX(cloneGameState, depth-1,α,β,false)
            best ← max(best, utility)
            α ← max(α, best)
            if β <= α then break
    else
        best ← +∞

        for move in tablutLegalMoves do
            cloneGameState ← gameState.update(move)
            utility ← MINIMAX(cloneGameState, depth-1,α,β,true)
            best ← min(best, utility)
            β ← min(β, best)
            if β <= α then break
    return best
```

4 Euristiche

4.1 Meta-euristiche

4.1.1 Taglio della ricerca

Un limite degli algoritmi di ricerca con avversari è che per ogni mossa da effettuare, per conoscere la contromossa giusta bisognerebbe espandere l'albero di ricerca fino ad arrivare a fine partita, cosa impossibile in un gioco come il Tablut, a meno di non possedere un Titan. Secondo l'articolo "An Upper Bound on the Complexity of Tablut" [Andrea Galassi & co.][1] lo spazio degli stati sarebbe addirittura $1.4 \cdot 10^{27}$.

Nine Men's Morris	English Draughts	Tablut	Othello
$3 \cdot 10^{11}$	$5 \cdot 10^{20}$	$1.4 \cdot 10^{27}$	10^{28}

International Draughts	Chess	Go
10^{30}	10^{50}	$2 \cdot 10^{170}$

Tabella 1: Upper Bounds Complexity

L'approssimazione più naturale da effettuare con un algoritmo di questo tipo è un taglio della ricerca ad un certo livello, applicando poi una funzione di valutazione ad ogni foglia. Ma se la ricerca deve essere interrotta in stati non terminali, l'algoritmo sarà necessariamente incerto sui risultati finali di quegli stati. Questo tipo di incertezza è indotta da limitazioni computazionali, piuttosto che informative. Infatti considerando la limitazione dei 30_s per eseguire ogni mossa siamo stati costretti a effetturare un taglio fino a profondità 2.

4.1.2 Strategia aggressiva

Abbiamo pensato di sviluppare una metaeuristica che permetta una progressiva modifica dei valori dei pesi della nostra funzione di valutazione in base al numero del turno corrente. Tale caratteristica serve ad aumentare l'aggressività in termini di cattura per l'agente quando la partita sembra destinata al pareggio. I valori dei pesi delle euristiche sono stati scelti tramite ricerca per tentativi al fine di ottimizzare il comportamento dell'agente e massimizzare le sue probabilità di vittoria.

Algorithm 2 Strategia Aggressiva Swedes

procedure EVALUATESWEDES(*gameState*)
    ...
    ...
    $f_1 \leftarrow$ **oddsOfPieces**(gameState)
    $f_2 \leftarrow$ **minimizeKingDistance**(gameState)
    $f_3 \leftarrow$ **piecesAroundKing**(gameState)
    $f_4 \leftarrow$ **weightKing**(gameState)
    if $turn \leq 15$ **then**
        $utility \leftarrow f_1 + f_2 + f_3 + f_4$
    else if $turn \leq 30$ **then**
        $utility \leftarrow f_1 + 2 \cdot f_2 + f_3 + f_4$
    else
        $utility \leftarrow f_1 + 4 \cdot f_2 + f_3 + f_4$
    return *utility*

Algorithm 3 Strategia Aggressiva Muscovites

procedure EVALUATEMUSCOVITES(*gameState*)
    ...
    ...
    $f_1 \leftarrow$ **oddsOfPieces**(gameState)
    $f_2 \leftarrow$ **minimizeKingDistance**(gameState)
    $f_3 \leftarrow$ **piecesAroundKing**(gameState)
    $f_4 \leftarrow$ **weightKing**(gameState)
    if $turn \leq 15$ **then**
        $utility \leftarrow -(f_1 + f_2 + f_3 + f_4)$
    else if $turn \leq 30$ **then**
        $utility \leftarrow -(2 \cdot f_1 + f_2 + 2 \cdot f_3 + f_4)$
    else
        $utility \leftarrow -(4 \cdot f_1 + f_2 + 4 \cdot f_3 + f_4)$
    return *utility*

4.2 Funzione di valutazione

La funzione di valutazione realizzata è stata strutturata come una serie di euristiche per definire un buon comportamento per i movimenti delle singole pedine. Essa utilizza l'attuale stato di gioco per valutare l'utilità di una singola mossa, restituendola al livello min o max che l'ha chiamata, come mostrato dallo pseudocodice dell'algoritmo $\alpha - \beta$ *pruning* nella sezione 3.2.

Tale funzione è stata implementata all'interno del file Heuristic.java e denominata evaluation.
Nel caso in cui venga raggiunto uno stato terminale, si deduce l'esito della partita. Se la partita è stata vinta, viene ritornato il valore massimo rappresentabile tramite tipo intero, cioè $Integer.MAX\_VALUE$, se è stata persa, il valore minimo rappresentabile tramite tipo intero, cioè $Integer.MIN\_VALUE$ e 0 in caso di pareggio. In tal modo, l'agente comanderà sempre le mosse che lo portano alla vittoria con una mossa. Nel caso in cui non venga raggiunto uno stato terminale si procede con una stima dell'utilità, definita anche valutazione euristica. Tale valutazione viene calcolata distintamente tra Swedes e Muscovites. Nel caso in cui l'agente comandi le mosse eseguite dagli Swedes, la funzione di valutazione controlla se dallo stato corrente sia possibile vincere la partita in due mosse con probabilità 1, vale a dire indipendentemente dalle mosse che può effettuare l'avversario. In tal caso, il valore restituito

Algorithm 4 Evaluation function

procedure EVALUATION(*gameState*, *agentPlayer*)
inputs: game, player

if *winner* == *agentPlayer* **then**
return ∞
else if *winner* == $1 - agentPlayer$ **then**
return $-\infty$
else
return 0
if *agentPlayer* == *Swedes* **then**
utility $\leftarrow$ **evaluateSwedes**(gameState)
else if *agentPlayer* == *Muscovites* **then**
utility $\leftarrow$ **evaluateMuscovites**(gameState)
return *utility*

è $Integer.MAX\_VALUE$ – 10000. In tal modo, gli Swedes effettueranno una o due mosse che portino alla vittoria se tali mosse sono lecite dallo stato corrente.
Se non è possibile vincere con probabilità 1, allora lo stato verrà valutato tramite la chiamata alla funzione evaluateSwedes. Tale funzione restituisce la somma pesata di 4 funzioni euristiche:

- **oddsOfPieces**.
  Tale funzione prende in ingresso lo stato corrente e restituisce un valore che rappresenta il vantaggio o lo svantaggio in termini di pezzi con l'avversario. Tale euristica cerca implicitamente di forzare l'agente a catturare più pedine avversarie possibili.

Algorithm 5 Eursitica 1, function

procedure ODDSOFPIECES(*gameState*)
utility $\leftarrow (2 \cdot numberPiecesSwedes\text{-}2) - numberPiecesMuscovites$
return *utility*

- **minimizeKingDistance**.
  Tale funzione prende in ingresso lo stato corrente e restituisce un valore che rappresenta la distanza di Manhattan del Re dall'angolo più vicino.

Algorithm 6 Eursitica 2, function

```
procedure WEIGHTKING(gameState)
    utility ← 4 − minManatthanDistance(gameState)
    return utility
```

- **weightKing**.
  Tale funzione prende in ingresso lo stato corrente e restituisce un valore che rappresenta la vulnerabilità del Re, infatti esso è relativamente sicuro nel suo *throne* ma una volta sul campo, diventa vulnerabile. Anche se viene catturato allo stesso modo di un *soldier*, la sua vulnerabilità è maggiore in quanto la sua perdita termina il gioco.
  Pertanto la vulnerabilità è stata pensata equivalente al numero di pedine nere necessarie alla cattura del Re in base alla posizione di quest'ultimo. Come esplicitato nelle regole del gioco al capitolo 2, il Re necessita di essere accerchiato da 4 pedine nere quando esso è situato all'interno del suo trono, da 3 quando è situato in una casella adiacente al trono e da 2 nelle altre zone del campo.

Algorithm 7 Eursitica 3, function

```
procedure MINIMIZEKINGDISTANCE(gameState)
    if kingPosition(gameState) == centre then
        utility ← 4
    else if kingPosition(gameState) == neighboursCentre then
        utility ← 3
    else
        utility ← 2
    return utility
```

- **piecesAroundKing**.
  Tale funzione prende in ingresso lo stato corrente e restituisce un valore che rappresenta lo stato di protezione del Re. Essa analizza le caselle adiacenti al Re e restituisce un valore in base al contenuto di tali caselle. Più il Re è accerchiato da pedine bianche, più il Re risulta protetto e viceversa.

Algorithm 8 Eursitica 4, function

procedure MINIMIZEKINGDISTANCE($gameState$)
  $utility \leftarrow swedesAroundKing - muscovitesAroundKing$
  return *utility*

Pertanto il valore ritornato dalla funzione di valutazione sarà pari a:

$$Eval(s) = \sum_{i=1}^{4} w_i \cdot f_i(s) \tag{1}$$

Dove le f_i rappresentano i valori restituiti dalle funzioni di valutazioni euristiche, mentre w_i il peso assegnato differentemente ad ogni funzione. Tale peso serve ad aumentare l'importanza che una funzione ha rispetto alle altre. Nel caso in cui l'agente comandi le mosse eseguite dai Muscovites, viene invocata la funzione evaluateMuscovites che restituisce la somma delle le stesse funzioni euristiche ma con segno opposto. Tale scelta è dettata dalla deduzione che un'utilità per un agente risulta essere opposta per l'avversario.

5 Risultati

Dopo la progettazione e l'implementazione dell'agente, si è pensato a come misurare le performance ottenute dalla sua esecuzione. Si è deciso di misurare le prestazioni dell'agente in termini di percentuale di vittorie, numero medio di mosse per partita, catture effettuate e catture subite. I test iniziali sono stati effettuati impostando come avversario l'agente *GreedyTablutPlayer*, già implementato all'interno del framework utilizzato. Vista la natura deterministica di entrambi gli agenti, si è pervenuti al medesimo risultato per ogni loro esecuzione. Pertanto è stato appositamente sviluppato, un agente che ad ogni turno può effettuare mosse casuali con una probabilità fissata chiamato *RandomGreedyTablutPlayer*. Tale probabilità è stata impostata al 20% in modo da garantire con elevata probabilità almeno una mossa casuale ogni partita. Pertanto i test condotti vedono variare l'utilizzo di tali agenti: i primi vedono l'agente sviluppato comandare le mosse degli *Swedes*, sfidando i *Muscovites* comandati dal *RandomGreedyTablutPlayer*. Poi si è testato il comportamento degli agenti a parti invertite e infine è stata testata una partita con entrambe le parti comandata dall'agente sviluppato.

5.1 RandomGreedyTablutPlayer vs AlfabetaPlayer

Sono state effettuate 3400 partite impostando l'agente sviluppato a comando degli *Swede*s che sfida *RandomGreedyTablutPlayer* a comando dei *Muscovites*. I risultati ottenuti sono i seguenti:

- Percentuale partite vinte: 99.94%
- Percentuale partite perse: 0.03%
- Percentuale partite pareggiate: 0.03%
- Media mosse totali effettuate per partita: 20.92
- Media catture effettuate: 1.78
- Media catture subite: 2.13

5.2 AlfabetaPlayer vs RandomGreedyTablutPlayer

Sono state effettuate 3400 partite impostando l'agente sviluppato a comando degli Muscovites che sfida RandomGreedyTablutPlayer a comando dei Swedes. I risultati ottenuti sono i seguenti:

- Percentuale partite vinte: 89.53%
- Percentuale partite perse: 10.44%
- Percentuale partite pareggiate: 0.03%
- Media mosse totali effettuate per partita: 12.81
- Media catture effettuate: 4.86
- Media catture subite: 2.78

5.3 AlfabetaPlayer vs AlfabetaPlayer

Vista la mancanza di variabili aleatorie all'interno dell'agente sviluppato, la partita che vede tale agente impostato al comando di entrambe le parti vede sempre lo stesso esito raggiunto dalla stessa consecuzione di stati. L'esito è la vittoria per gli *Swedes* con 24 mosse effettuate totali, 5 catture da parte dei Muscovites e 3 da parte degli *Swedes*. Tale risultato è giustificato dall'asimmetria dello stato iniziale.

6 Sviluppi futuri

Tra possibili sviluppi futuri suggeriamo la necessità di effettuare dei test mirati a verificare se la funzione di valutazione ordini gli stati terminali allo stesso modo della vera funzione di utilità. Cio deve essere fatto siccome in caso contrario, un agente che la utilizza, come nel nostro caso, può selezionare mosse non ottimali anche se può vedere in anticipo fino alla fine del gioco.

È possibile migliore i risultati ottenuti cercando di trovare dei pesi ottimi per la funzione di valutazione messa in gioco. Attualmente i pesi ottenuti sono frutto di prove finalizzate a massimizzare la percentuale di vittorie, facendo variare un singolo peso per volta. Il prossimo step sarà quindi di utilizzare un algoritmo di ricerca locale, ad esempio *Random Mutation Hill Climbing*, per trovare gli iperparametri ottimi per la funzione di valutazione.

È possibile, inoltre, cercare di effettuare un analisi delle performance dell'agente contro altri agenti più complessi.

Infine creare un ambiente di competizione analogo a quello a cui ci siamo ispirati per realizzare il progetto tentando quindi di riuscire ad analizzare i risultati degli scontri con agenti implementati da altri studenti del corso per lo stesso gioco.

7 Conclusioni

In conclusione abbiamo sviluppato un controllore basato su utilità, implementato un programma agente che utilizza un $\alpha - \beta$ *pruning*, previsto una funzione di utilità per permettere all'agente di scegliere autonomamente l'obiettivo più conveniente. I risultati ottenuti rispecchiamo ciò che ci eravamo prefissati di vedere che risulta essere una percentuale di vittoria più elevata per il team degli *Swedes*, infatti ,contestualizzando allo specifico caso di test, supponiamo che ciò sia dovuto a un euristica per i *Swedes* più raffinata. Si riservano le migliorie descritte nella sezione 6 per il prossimo lavoro.

Elenco delle figure

Elenco delle tabelle

Riferimenti bibliografici

[1] Andrea Galassi. «An Upper Bound on the Complexity of Tablut». In: (2009). URL: http://ai.unibo.it/sites/ai.unibo.it/files/Complexity_of_Tablut_0.pdf.

[2] Peter Norvig Stuart Russel. *Intelligenza Artificiale, un approccio moderno*. Pearson Education Italia S.r.l, 2005.