**CITS3001 Project, 2016: Ammar Abu Shamleh, 21521274**

**1.0 Introduction**

The purpose of this project was to implement an AI player for an abstract board game known as Pylos. Pylos is a pyramid building board game, involving the placement of spheres by two opponent players. The goal of each player is either to preserve their spheres until the opponent runs out, or to ensure they place the last sphere atop the pyramid. Either of these conditions will result in a victory.

Successful completion of this project required implementation of AI game search theory, including *minimax* and *alpha-beta pruning,* primarily. The driving factor in producing a strong, intelligent AI, however, is the logic within the AI’s evaluation function.

The design and structure of the supplied program, along with any relevant experimentation, decisions, and results, will all be provided and explained in the coming sections of this report.

**2.0 Design and Structure**

The approach to the design of this program has been very modular, with the program being segregated into 4 main logical components:

1. An internal game state representation *(Pylos.java)*
2. A driver class, which sets up and facilitates games *(Driver.java)*
3. An internal representation of legitimate game moves *(PylosMove.java)*
4. An AI move calculating class *(PylosAI.java)*

**2.1 Game state representation**

The primary function of the *Pylos* class is to offer an internal representation of the game’s state, and implement the mechanics and rules of the game. This class has no AI logic within it, and simply implements the core mechanics of the game, and provides public methods to alter a given *Pylos* object (i.e. change a game state, or apply a move).

The game board is represented in 4 2-dimensional arrays of integers, each array representing one tier/level of the game board. As moves are made, these arrays are modified to reflect the moves. The *Pylos* class defines integer constants that represent *white, black* and *empty* (as 1, 2 and -1, respectively). These constants are used to record a player’s ownership of a position on the board, by storing the appropriate number in the appropriate element of the array(s). Here, ‘ownership’ simply refers to which colour sphere occupies a specific position.

The class implements methods to make requested moves on the game state, so long as they are legal. These moves are communicated as strings and player numbers (i.e. “a4” and *WHITE* given to a place method would implement the move of placing a white sphere at position *a4*). Such methods are used by the driver function to update the game state as moves are made. The class contains a standard method that determines if the game state represents a terminated game; this is used by both the *Driver* class (for ending the game appropriately) and the *PylosAI* class, for recognizing terminal states.

However, the class also implements validity enforcement methods which, given a move, don’t perform the move, but evaluate whether the move is legitimate or not, given the current game state. These methods are used extensively by the AI’s *actions* method, which must calculate a list of all possible moves by a specified player, in a specified game state.

Finally, the class also contains a method *applyMove,* which applies a move using not a string, but an object of type *PylosMove*. This class is integral to the operation of the AI.

**2.2 Driver class**

The primary function of the driver is simply to run and update the game to facilitate a match between an AI and a human. It simply has a human enter the move in string format, applies it to the game state, and then uses the *PylosAI* class to calculate the AI’s next move. The game itself is run using a standard while loop, terminating when the game is complete. Additionally, there are alternate driver classes in place (*AltDriver* and *HumanDriver*), which facilitate matches between AI vs. AI, and Human vs. Human, respectively.

**2.3 Game move representation**

The primary function of this class is to offer an object type for representing game moves, which underpins the operation of the AI. A move is either of type *PLACE* or *RAISE*, both of which are simply integer constants.

If the move type is *PLACE*, the coordinates are specified using 2 variables:

1. An integer specifying what level the sphere is to be placed
2. An array of 2 integers, specifying the coordinates the sphere is to be placed at

A move type of *RAISE* necessitates one more piece of information:

1. An array of integers specifying the location of the sphere to be raised (null if the move is of type *PLACE*)

(Note: The same variables used for *PLACE* moves are now used to specify where the raised sphere should be placed)

Finally, there is a Boolean determining whether or not the move follows up with a removal of sphere(s). Correspondingly, there is an integer specifying whether 1 or 2 spheres will be removed, and there are 2 arrays of integers that specify the position of the sphere(s) to be removed (both null if no sphere is being removed, or is null if only sphere is being removed)

It is the job of the *applyMove()* method to translate this logic into a viable, usable format within the *Pylos* class, and apply the move specified by a given *PylosMove* object to the game state.

It is the job of the *actions()* method to utilize this logic to construct an appropriate set of *PylosMove* objects that represent all legal moves in a given state, and store them in an array list, which is returned by the method.

* 1. **AI Move class**

The final, and most integral class, is the *PylosAI* class, which is a static class that uses game search logic to calculate an intelligent move, given a game state (i.e. *Pylos* object). The AI class is underpinned by three vital methods: *actions(), evaluate() and alphaBetaSearch().* Each of these methods has an important responsibility within the operation of the AI, which will be discussed further in the analysis of the AI. Briefly, however:

* The method *actions(Pylos state, int player)* returns an array list of *PylosMove* objects*,* which represent all the moves that can legally be made by the specified player in the specified game state. This method underpins the operation of *alphaBetaSearch()*
* The method *evaluate(Pylos state, int player)* is the cornerstone of the AI, and the performance of the AI depends critically on the quality of this function. The function takes a specified game state, and computes an evaluation of the state, for the specified player. A good state for *WHITE* player would be positive (thus *WHITE* would aim to maximise its value in game search), while a good state for *BLACK* player would be negative (thus *BLACK* would aim to minimize value in game search).
* The method *alphaBetaSearch(Pylos state, int player)* simply performs a standard recursive alphaBetaSearch to compute the best move with respect to the evaluation function in use. It proceeds up to a specified depth limit, before returning an evaluation of the state under examination once that depth limit is reached

**3.0 AI Analysis**

The *PylosAI* class is broken into 3 main components:

1. Game search logic
2. Possible move calculation
3. Evaluation function
   1. **Game search logic**

The primary function here is simply to perform the standard game search algorithms to compute a good move, starting from a specified game state. This work is all done inside the *alphaBetaSearch(), minValueAB() and maxValueAB()* methods. With the work in *actions()* and *evaluate()* being logically distinct, and being performed separately, the logic of the game search can be discussed independently.

The game search logic within *PylosAI* is simply a standard implementation of alpha beta game search, effectively running a recursive minimax algorithm on the Pylos game tree, and reducing computation time by pruning nodes that are immediately identified as being worse than what has already been seen. As the branching factor and depth of Pylos are far too large to expand nodes all the way to terminal nodes, a cut off depth is used to return moves within a tractable amount of time. This depth depends on the complexity of the evaluation function in use

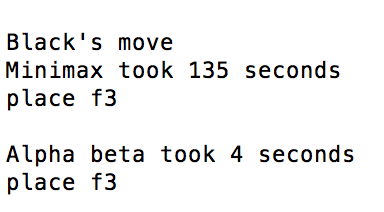
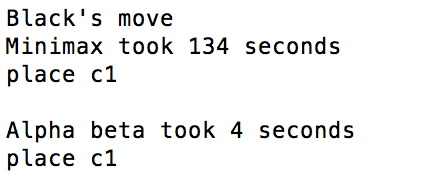
*alphaBetaSearch()* begins by computing all moves that can be performed from the current game state by the specified player, using the *actions()* method. For each action, it computes the resulting state, and then runs either *maxValueAB()* or *minValueAB(),* with the resulting state and opponent’s player number being provided as arguments (the function selected depends on whether *alphaBetaSearch()* is being run for *BLACK* or *WHITE*). Taking *WHITE* as an example (i.e. AI aims to maximise utility), the concept is to pick the action from *original state* that leads to the *result state* whose minimum value (branching down from *result state)* is the maximum of all *result states* branching down from the *original state*.

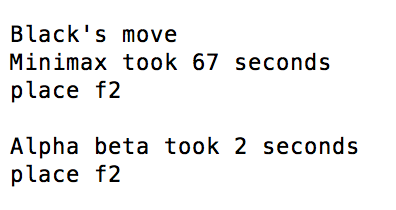
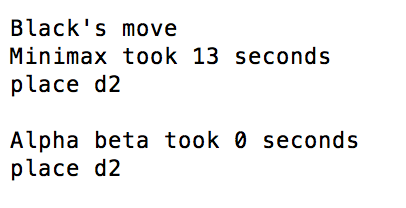
Effectively, it is just a standard game search algorithm, with node pruning for efficiency. Node pruning is done by maintaining the values *alpha* and *beta* and comparing against them at every point in the search.

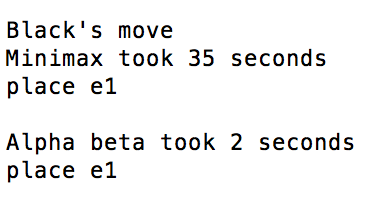
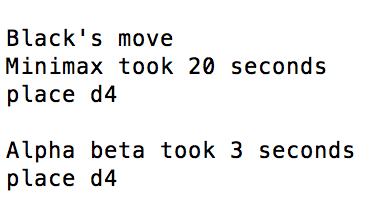
The functions *maxValueAB() and minValueAB()* call each other, until either a terminal state, or the cut off depth, have been reached. At this point, either the state’s utility (if it’s a terminal state) or its evaluation (if the cut off depth has been reached) is returned to the calling function. The recursion unwinds, and the results percolate back up the game tree, until they reach the top (the *original state* on which *alphaBetaSearch()* was called). *alphaBetaSearch()* will then pick the best predicted move, based on these values.

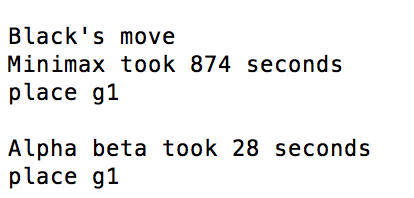
*Alpha* and *beta* are updated as the function proceeds, and are passed between *maxVaueAB() and minValueAB()* as arguments. The current depth of the search is also passed between the functions as arguments, with each function incrementing it as it passes it along. Once the depth hits the depth limit, whichever of those 2 functions is running will simply return an evaluation

The AI originally ran a basic minimax algorithm, without pruning sub-trees evolving from actions that wouldn’t be chosen. A sample match was run between two AI systems (two different evaluation functions), with each AI calculating its moves twice: once using minimax, and once using alpha beta. The moves calculated were always the same (such is the point of alpha beta). The time taken for the calculation of each move was recorded, and an average produced at the end for alpha beta, and another average for minimax. The results revealed that alpha beta was consistently faster than minimax, sometimes by a substantial margin. Following are some screen-caps of the simulation match that illustrate the difference in computation time between the 2 algorithms:









The averages produced at the end of this game revealed that alpha beta drastically reduced computation time for the AI running the more advanced evaluation function.

The AI running the simple evaluation function averaged **371ms** for *minimax,* and **365ms** for *alpha beta*

Conversely, the AI running the more complex height-based evaluation function averaged **45.2 seconds** for *minimax*, and **1.69 seconds** for *alpha beta*

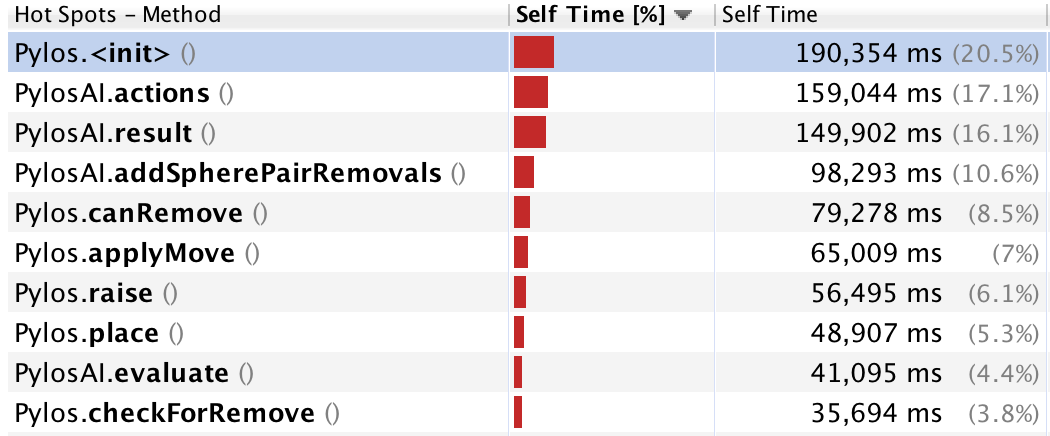
The time improvement from using alpha beta pruning here is clear to see, and, keeping time constant, allows for a better performing AI, due to the ability to look further ahead in the same amount of time.

* 1. **Possible move calculation**

The primary function of this portion of the AI class is to return all actions possible by a given player (i.e. either *WHITE* or *BLACK*) in a specified game state. Given the nature of the game, the process involves laborious calculations on the board arrays, and checks at every position on the board for the validity of a move. Any legal *PLACE* or *RAISE* moves found must then be checked to see if they require a follow up of removing 1 or 2 spheres. If they do, all configurations of placing/raising a sphere, and then following up with 1 sphere removals, and all configurations of placing/raising a sphere, and then following up with 2 sphere removals, must be considered.

The method operates by creating an array list of *PylosMove* objects, and adding all legal moves to the list, before returning it.

While simplistic in terms of design, the method performs a huge amount of computation, owing the large number of possible moves that can exist in a game state, and all the removal follow-ups that can be part of a move. As such, much of the AI’s time is spent within the *actions()* method.



The above analysis of the program’s CPU usage breakdown reveals that **17.1%** of the CPU’s time is spent within the *actions()* method, with a further **10.6%** being spent inside *addSpherePairRemovals()*, which a helper method used exclusively by the *actions()* method. Furthermore, *checkForRemove()* and *canRemove()*, two helper methods primarily utilised by *actions(),* also consume **3.8%** and **8.5%** of the CPU’s time, respectively. Conversely, the *evaluate()* method, which is the cornerstone of the AI’s performance, consumes only **4.4%** of the CPU’s time. This reveals that a primary avenue for improving the efficiency, and thus performance, of the AI, is to work on optimizing the logic within the *actions(),* and its helper methods. By reducing the time spent within *actions(),* which is called numerous times at every level of the game tree, more time can be allotted to progressing to deeper depth limits, or computing a more complex evaluation function.

* 1. **Evaluation function**

The evaluation function is the logical cornerstone of the AI. Due to the huge branching factor and depth of Pylos, it is impossible to progress all the way down to terminal nodes during a game search within a realistic amount of time. As such, a reasonable depth limit must be in place, which demands an evaluation function to evaluate a non-terminal node and return its estimated utility.

In my experimentation, I found that even minor changes to the weightings given to each attribute within the evaluation function, and the addition of new attributes, can significantly change the play-style and tactics of the AI. Designing the evaluation required an analysis of quality vs. Efficiency. A more complex function demands more computation, which in turn, reduces the amount of time that can be spent in other methods, and possibly reduces the depth limit to which the search can progress.

**4.0 Evaluation functions**

The factors considered within the evaluation function, and the weighting given to each, will mould the play-style and goals adopted by the AI in playing Pylos. In an attempt to find an effective evaluation function, I experimented with several different types of functions, each with different tactics in mind. Through my experimentation, I found that providing an evaluation function a clear and specific tactic produces a better AI player than attempting to write an evaluation that considers everything.

Providing such a goal can be done by:

* Only considering the factors relevant to that tactic, and ignoring all else, **or**
* Through providing different factors distinctly different weightings (through multiplicative constants), depending on the desired play-style

My experimentation, which will be detailed below, led me to the conclusion that an ideal evaluation function is one that considers multiple factors, but provides each factor distinctly different weightings to create a specific tactic and play-style that suits the mechanics of the game. However, even a simple function that considers only a few simple factors can be effective, so long as those factors are chosen well.

**4.1 Different function attempts**

I experimented primarily with 3 different types of functions:

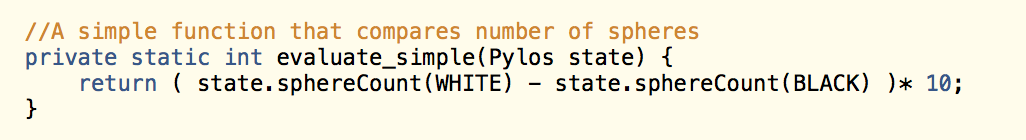
1. A simple function that just considers the number of spheres owned by each player
2. A blocking function, that simply attempts to minimize the number of spheres owned by the opponent
3. A height based function, that tried to prioritize placing spheres at higher levels

Each function had its strengths and weaknesses, and the performance and play-style of each varied significantly with the weightings given to each factor.

**4.1.1 Simple function**

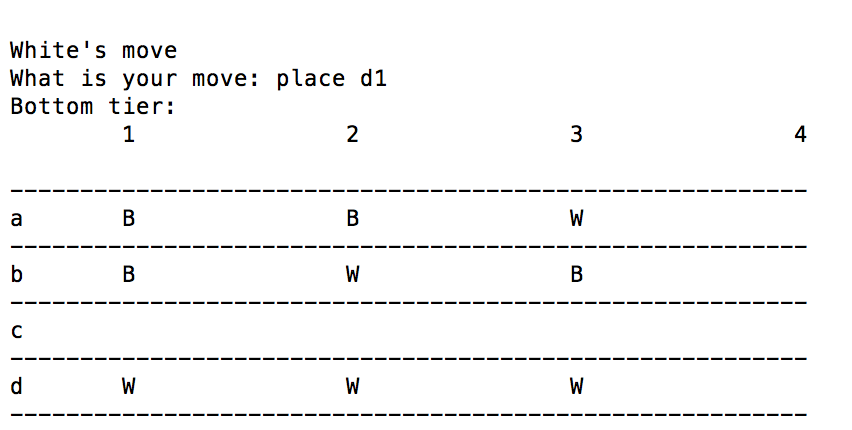
This function is (as the name suggests) a simple evaluation function that considers only the number of spheres on the board for each player. The controllable attributes here are the weightings that can be applied to each player’s number of spheres.

For example, an initial, naïve approach, might be to produce a function such as:



This function simply returns a positive number if *WHITE* owns more spheres than *BLACK*, and a negative number if the opposite is true (remembering that *BLACK* aims to minimize utility). This is simply evenly weighting the importance of preserving the player’s spheres, and the importance of draining the opponent’s spheres.

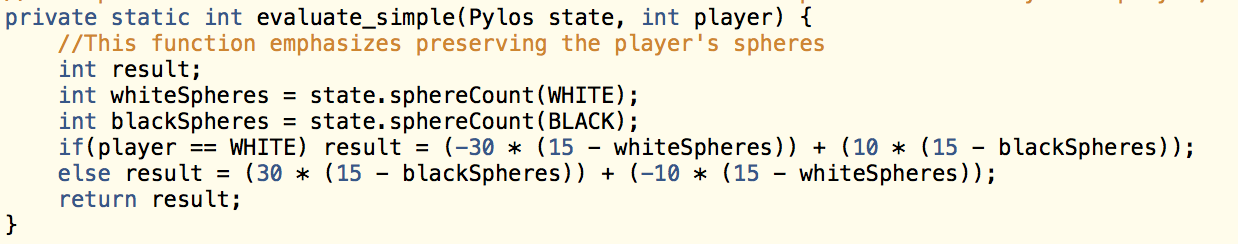
The function does play the game satisfactorily, however many moves lack intelligence and foresight, and the balancing of priorities leaves the AI making awkward decisions at times. For example, considering the following board, in which *WHITE* (human player)has just made a move:



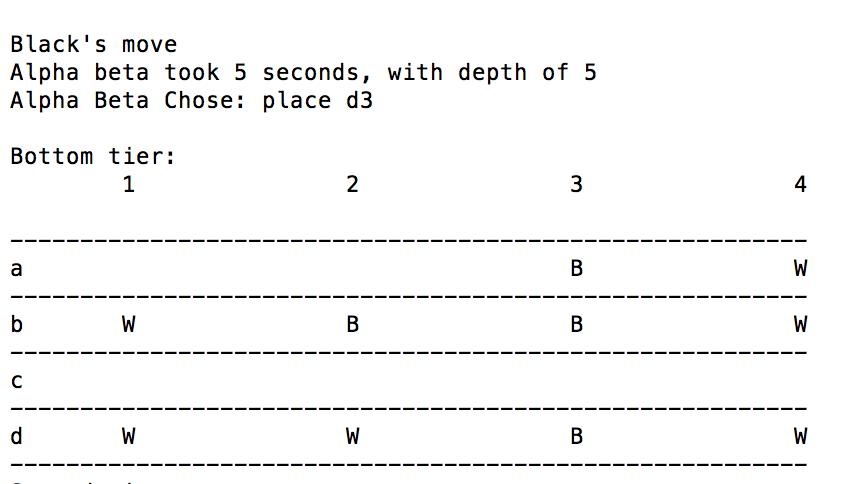
The best move for the AI (who is playing *BLACK*) to make would be to place a sphere at position *d4*, so as to block *WHITE* from creating a line and obtaining a sphere advantage. However, the AI instead responded by placing a sphere at *a4,* which immediately leaves *WHITE* with an opportunity to remove 1, or even 2 spheres.

The primary failure of this function is the even weighting of the two priorities (preserving spheres and draining opponent of spheres). The AI doesn’t clearly target one goal, and thus fails to sufficiently counter common tactics.

This is an example of the generality of the function failing it. If the function is tweaked, so as to emphasize preservation of the player’s spheres:

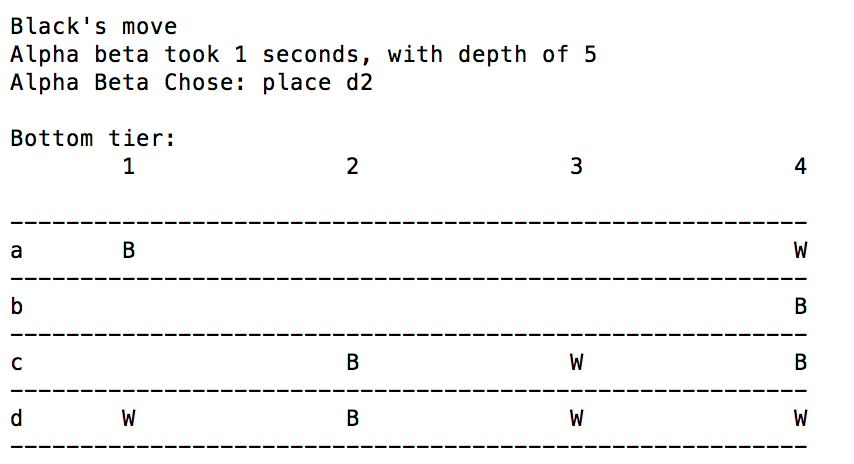


Then the AI plays with the clear goal of creating squares and lines to remove spheres, and aggressively tries to set up as many as possible. While this makes the system predictable, the 5-ply look ahead of the system confers a natural advantage over most human players, who mostly consider the immediate situation; thus situations such as the following become relatively common:



In the above situation, the AI (playing as *BLACK)* has managed to set up a situation in which it is guaranteed to form a square or line. No matter what move is made by *WHITE*, the AI will be able to follow up with a removal, as it simultaneously built up multiple squares and lines, instead of focussing on one at a time. This simple example demonstrates how developing an evaluation function with a specific tactic in mind carries through clearly to the moves made by the AI, and tends to result in better, more ‘intelligent’ performance.

Conversely, if the values in the above function were inverted, such that the AI placed a weight of 30 on minimizing opponent’s sphere store, and a weight of 10 on preserving its own spheres, the AI then aggressively tries to thwart any squares or lines being formed by the opponent.

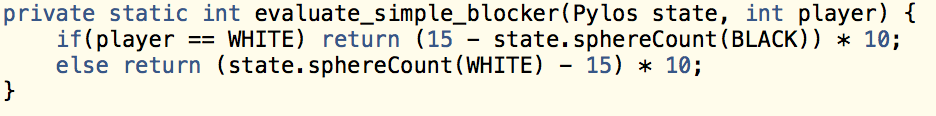


In the above example, the human (as *WHITE*) has attempted to create square/line by beginning around the point *d4*, and slowly building around it along the *4* column, the *d* row, and the square *c3, d3, c4, d4.* However, the AI has aggressively blocked the entire manoeuvre, forbidding the *WHITE* player from removing any spheres.

Thus despite being a simple function, there is significant room for tactical and varied play through experimental adjustments to attribute weightings.

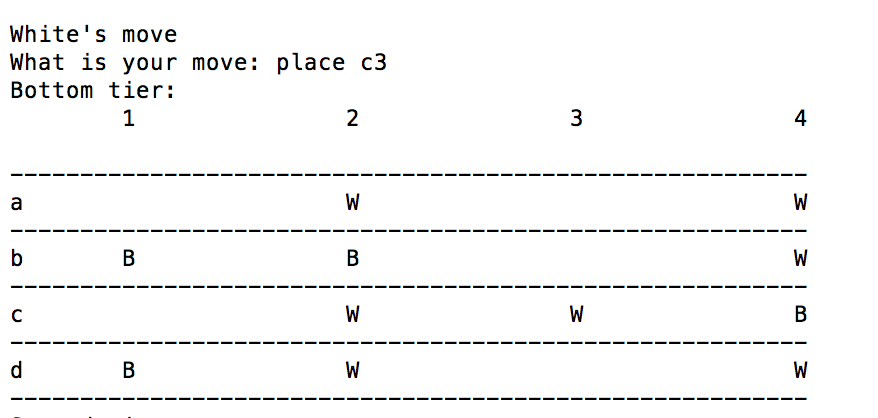
**4.1.2 Blocker function**

The blocker function is a very simple, targeted function that simply aims to maximize the number of opponent’s spheres on the board.

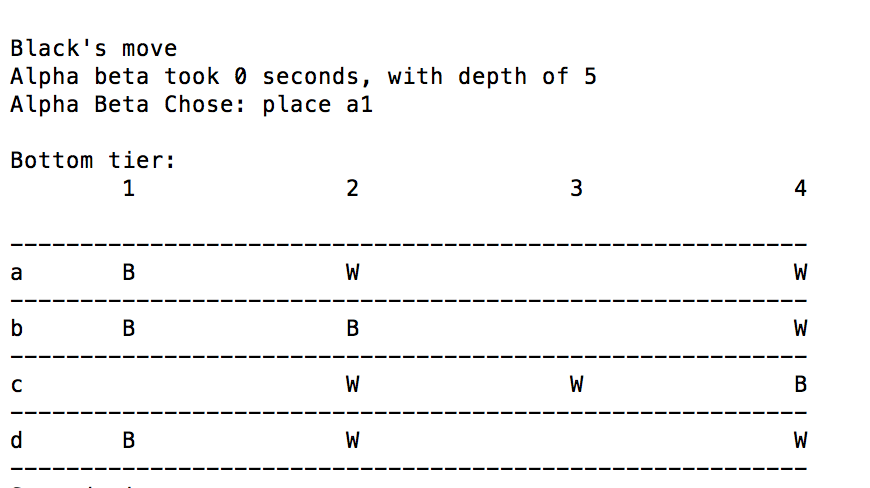


It should be an overwhelmingly single minded function that aims to block all squares and lines, however it plays quite differently to what would be expected.

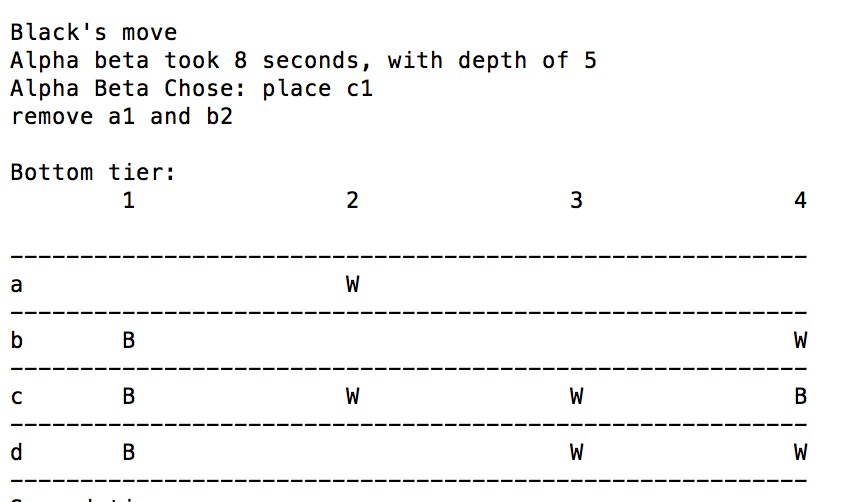
In the following screenshot, the blocker would be expected to follow up by placing a sphere at *d3*, to as to block the human *white* player from recovering their spheres.



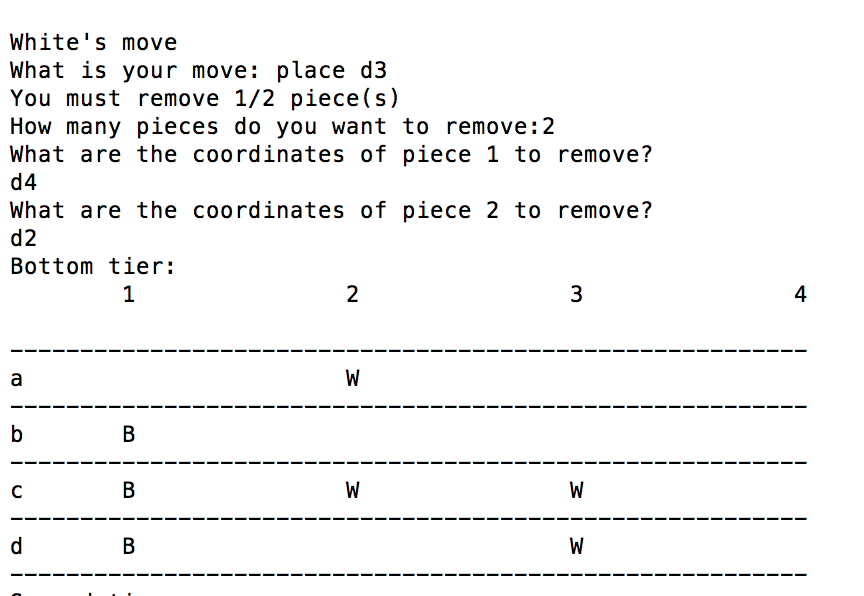
However, it instead follows up with a placement at *a1*, aiming for a line of its own.

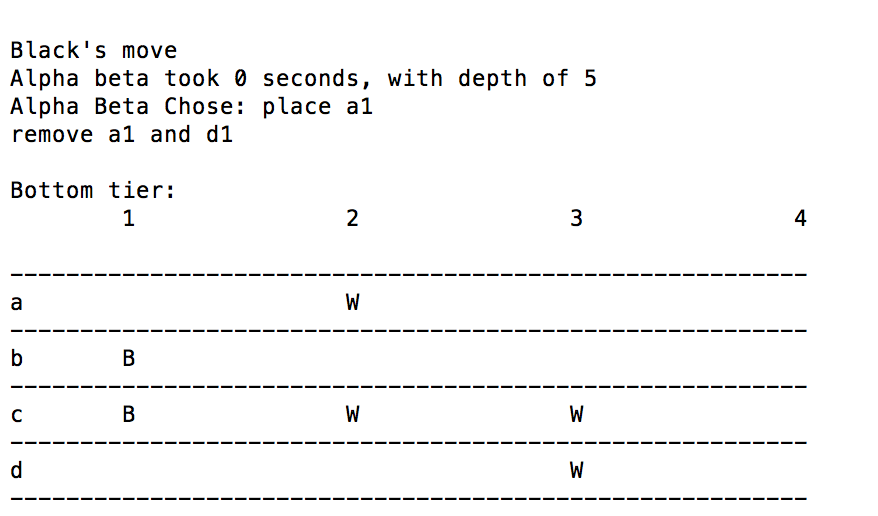


Upon further play against the blocker, there seem to be several factors working here that cause such behaviour. The blocker seems to be averse to creating squares, so as to avoid raise moves by the opponent, which conserve the opponent’s spheres. Whenever it forms a square/line, it always follows up with removals that remove as many squares as possible that would have allowed access to higher tiers. This is observed in the below example, where it forms a line, and breaks 2 squares.

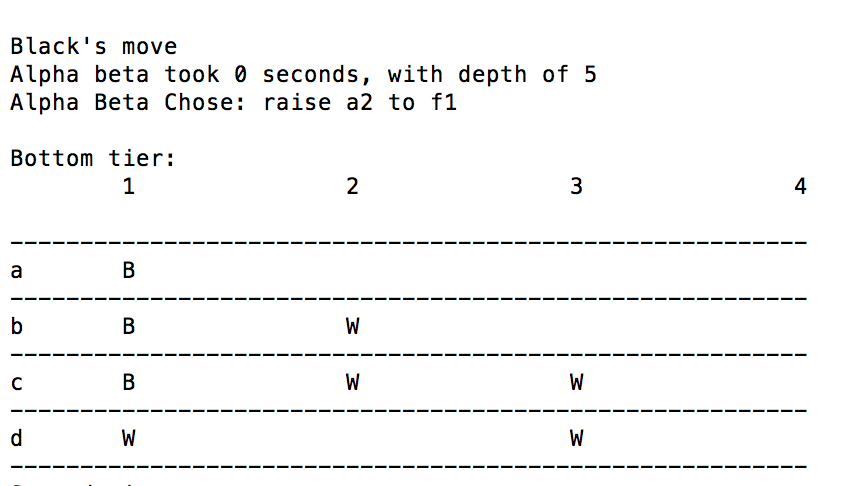
****

Furthermore, the look ahead of 5 on the blocker probably means that its moves are all heading in directions so as to just minimize the number of spheres at higher levels. It thus allows removals by the opponent, but only so it can itself follow up with removals of its own.



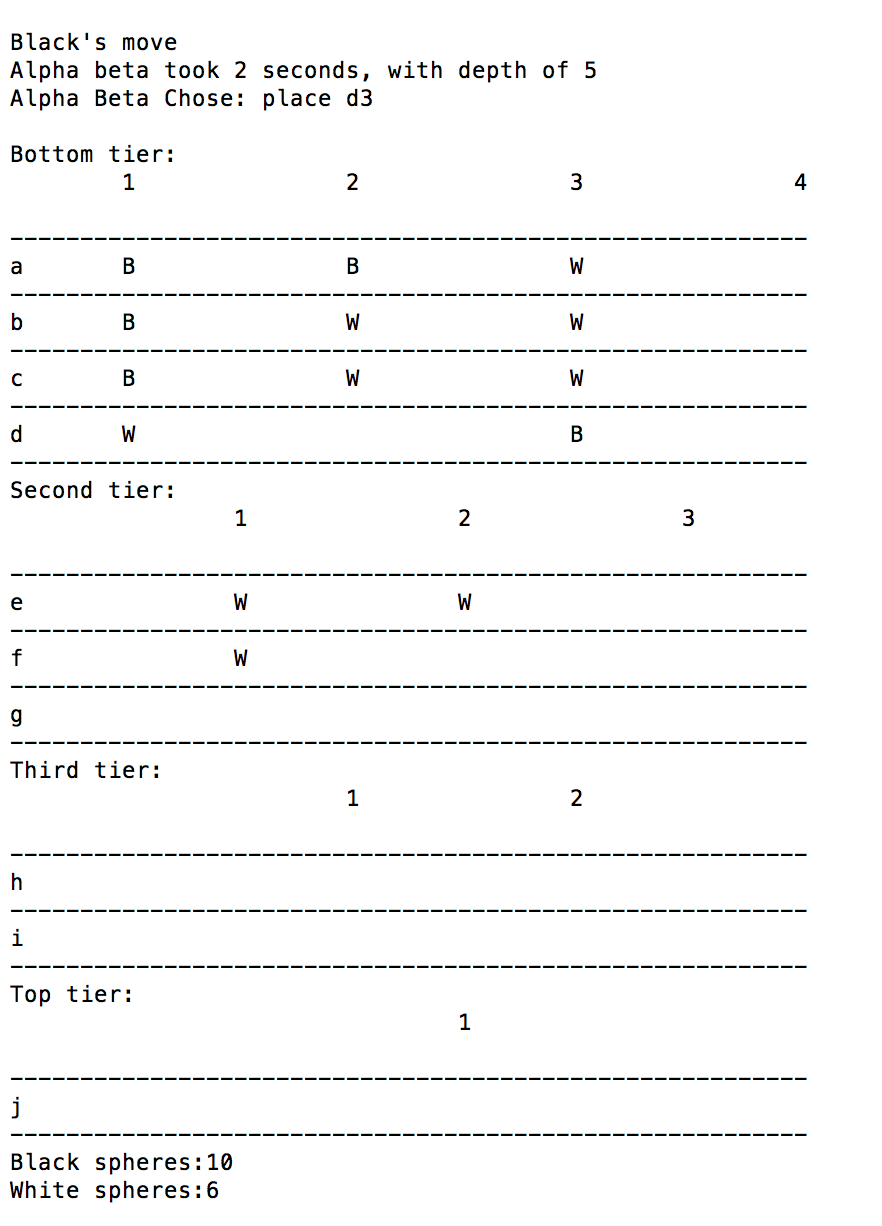
****

The blocker will also (surprisingly, perhaps), perform raise moves, despite having no inclination to prioritise height, and having no consideration of its own sphere preservation



This is likely a move made simply to prevent the opponent from preserving their spheres on the next turn through a raise.

This all demonstrates that evaluation functions do not necessarily reflect themselves obviously in the AI’s gameplay, thanks to the look-ahead of the AI allowing it to perceive the situation several moves into the future, and act according to that, rather than the immediate situation. Were the depth limit for the AI reduced to 2, this blocker would single-mindedly and very explicitly block the opponent from creating any squares or lines. More look-ahead, however, allows it to approach its goal much more subtly. In this case, the blocker is very gradually locking the opponent’s spheres on lower levels, where they’ll later be impossible to remove, and preventing the opponent from playing a game of attrition with *raise* moves.

To illustrate this, much further down in this game, the board looked like:

With the blocker *(Black)*, in this situation, having successfully obtained a sphere advantage of 4 over the opponent.

Thus despite its simplicity, the blocker function works surprisingly well. In the end, as will be shown in later experimental data, its simplicity does leave it lacking compared to other, more refined evaluation functions

**4.1.3 Height prioritised placer, opponent blocker function**

**4.2 Experimental data analysis**

**4.3 Final function selected**