CITS3001 – Project Report

Pylos AI with Minimax

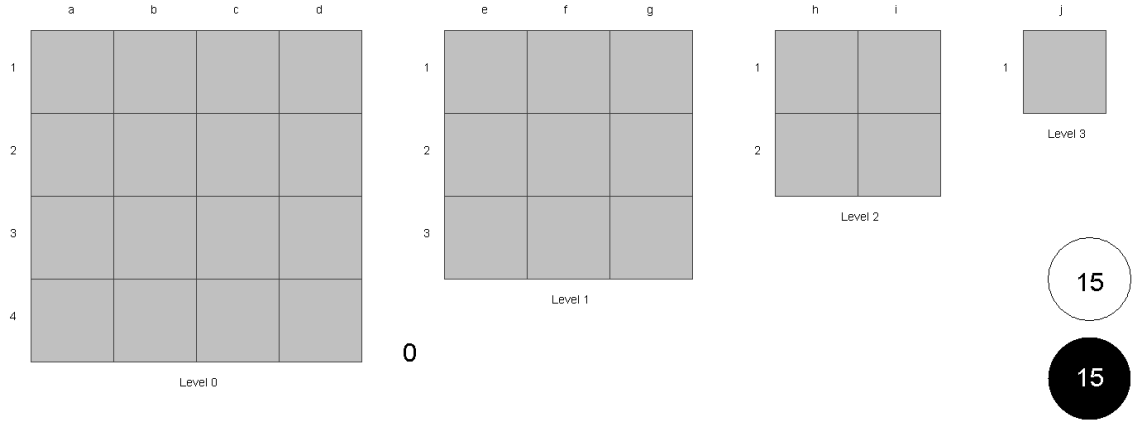
Thomas Ankers – 21490093

Jason Ankers – 21493118

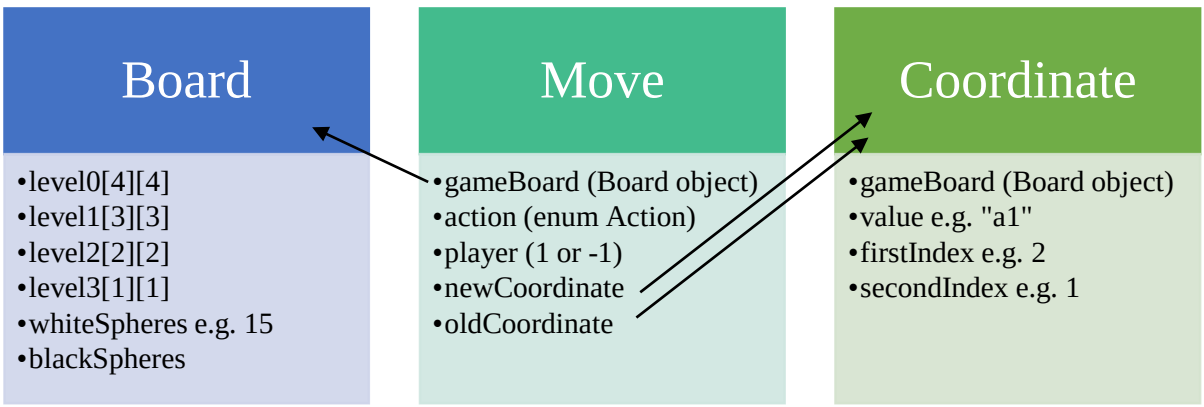
Outline

Our program demonstrates the basic functionality of the board game Pylos. We opted to illustrate all levels of the board separately and keep a visual representation of spheres remaining for each player. White represents the human player, while black represents the AI. White always makes the first move. We have mapped out each board position by labelling the sides of each level with their respective numbers and letters.

Our game is written in Java. Our AI uses the minimax algorithm combined with alpha beta pruning to generate a quick and (depending on depth) intelligent move. A tree is generated recursively through an initial alphaBeta call on a current board state.



Game Structure & Implementation



Our game operates on ‘Board’, ‘Move’ and ‘Coordinate’ objects. An initial empty board is generated when the program is first run. If the game is not in a finished state, it sits idle while waiting for a move from the current player. I the game is in a finished state, the winner function is called and the winning player is simply printed out. Players are stored as integers with 1 representing white and -1 representing black. When a board is created, each level is represented by a 2D integer array.

A player has two possible commands to issue the game if it’s their turn; *place* and *promote*. The game internally stores three actions; *place, promote,* and *remove*. If a player inputs “place a1” for example, a new coordinate is made holding the value a1. If the coordinate is valid (i.e. the coordinate actually exists), a Move object is made from the new coordinate. A coordinate object contains the level on the board (0-3), as well as the associated index for the corresponding level array. The game then checks if the move is valid. In the case of a *place* move it checks whether the cell is free and whether it has a base of four spheres on the level below to support it. In the case of *promote* or *remove*, the sphere must not have spheres resting above it. After a move is deemed valid, the move is executed on the board and the player’s spheres are adjusted. If the move was a success, the current player is swapped, and the AI move begins.

During the creation of moves, we also check to see if a line or square of the same colour is made. If so, the user is asked to specify a number of spheres to remove (0, 1, 2). Similar to placing, when removing spheres, a new Move object is made – only this time passing *remove* instead of *place* as the parameter. The object checks if the requested spheres to be removed are valid (belong to the player, not already underneath a sphere etc.) and then removes them.

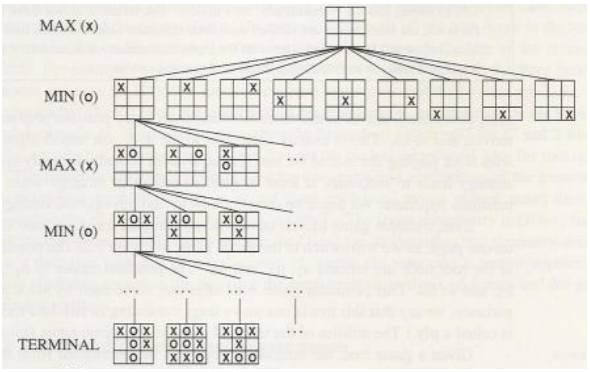
When it’s the AI’s turn to move, alphaBeta() is called on the current board state, passing the player BLACK as a parameter. A new board state is set based on the return value of the alphaBeta function. Also passed to alphaBeta is a specified depth. Depth represents the cutoff value to which the tree should be traversed. If a depth of 3 for example is given, the game will analyse 3 moves in the future when generating a move. The Board object returned from the alphaBeta function simply overwrites the previous Board that was present.

```
private void AIMove(AI p) {
    Board.enableStdout(false);
    gameBoard.setBoard(p.bestBoard(gameBoard, 3, currentPlayer));
    currentPlayer = -currentPlayer;
    gameBoard.repaint();
    Board.enableStdout(true);
}
```

The alphaBeta function explores all possible board states from a given board through a call to getPossibleBoards(), which returns every valid board state that can be returned from every possible move at a given instance in the game. A score associated with each move is calculated from the getBoardScore() function with a player passed as the parameter. Two variables alpha and beta are updated throughout the course of the alphaBeta function. Alpha represents the best already explored board state along the path to the root for the AI player, and beta represents the same but for the human player.

AI Overview & Justification

Choosing a suitable algorithm came down to the nature of the way Pylos is played. It's a fairly simple game, and since it's possible that a board can simply be evaluated at any given instance in time, with no predicting or guessing at play, any sort of search algorithm could do the job. It seemed feasible that for Pylos, a 'intelligent' move can simply be made by looking 2-4 moves in the future and choosing whatever move generates the highest score. The intelligence would be determined by the strength of the evaluation function, getBoardScore in our case, combined with the number of moves opted to look at in the future.



Evaluation function in Tic Tac Toe

Many examples use tic tac toe to demonstrate the use of an evaluation function. 1 would represent a win, -1 a loss, and 0 a draw. This always works since a game of tic tac toe is over in a couple of moves and you always generate some outcome for a player. In the case of Pylos however, it's not entirely practical to play out an entire game for every move searched. Instead, a board state needs to be analyzed, and a score generated for whatever player is making the move. Since an evaluation function doesn't predict or look in the future, it simply looks at the current state of the board and generates a score, the evaluation function for Pylos is primarily looking for lines or squares of the same colour formed.

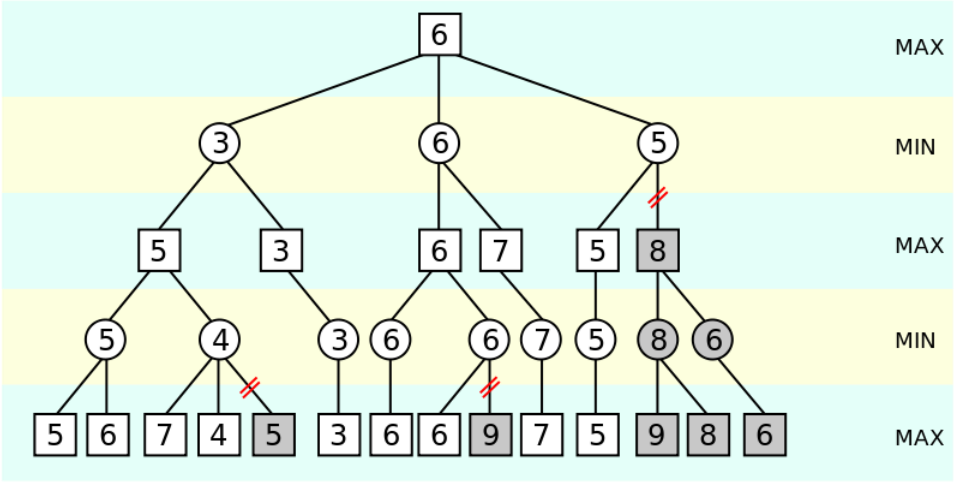
In our case, the board states returned from getPossibleBoard() take into account removals after lines or squares have been formed. As a result, our evaluation function simply looks for the board states which results the least number of spheres on the board for the given player. Further improvement would be taking into account the state of the board for the opposing player, deducting points off the score if the board state favours their next move. For example, forming a square made up of both white and black spheres would put the next player in a position to simply raise a sphere instead of having to place a new one.

Minimax Algorithm

We ultimately chose to go with the minimax algorithm since it was the only suitable option. The key to the algorithm is that it explores the back and forth moves between the AI and human by creating a tree of board states and performing a depth first (limited by our depth variable) search on the entire tree. Our search function returns the score of the board if it reaches a leaf node (or the game is finished), otherwise it will continue to recursively explore the tree until a leaf node is found. The AI class assigns the variable MAX to the AI (black) and MIN to the human player (white).

Pylos however starts with a large branching factor for the first level of the game, with 15 possible placements for the first move made. It peaks in complexity as deletion and promotion moves become available before reducing significantly after the first level has been filled. Given the fact the minimax explores every node of the tree; a lot of work has to be done calculating the optimal move.

Alpha Beta Pruning

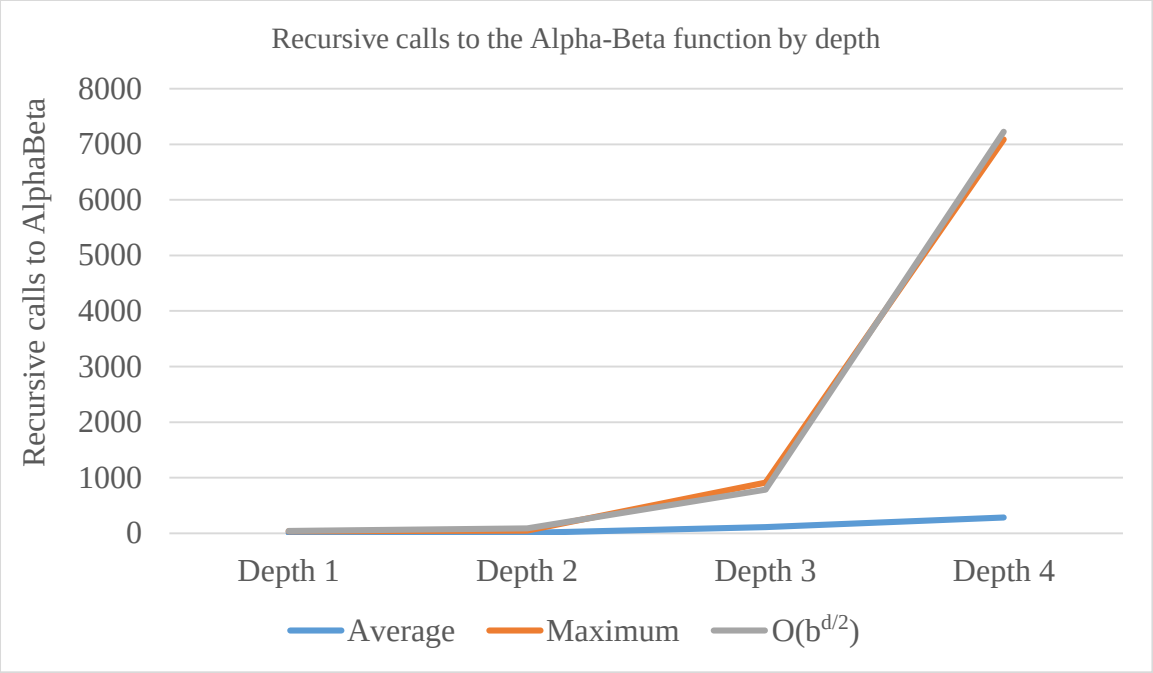


Alpha- Beta pruning example

The solution is provided by Alpha Beta pruning, an extension on the minimax algorithm that cuts off branches/nodes not worth exploring. As discussed earlier, we store variables *alpha* and *beta* in our alphaBeta function, holding the values of the best already explored board states for the respective players. If a board state is found to be worse than a previously examined state, the function cuts off evaluation any further into that region of the tree. Consequently, branches of the tree that have no influence on the final decision are not explored, heavily reducing the number of board states that need to be analyzed, especially early in the game.

Performance

The performance of the Minimax function with alpha beta pruning depends on the maximum depth set during its initial run. We experimented with several depths to determine the point where the AI was performing sufficiently and within time constraints. This turned out to be a depth of 3. The chart below illustrates the results of these tests.



The average number of calls to the Alpha-Beta function appears to increase linearly with depth, however the maximum calls appears to increase much quicker. At a depth of 4 the AI would perform well in the early stages of the game however as the complexity of moves increased it slowed down far below the 5 second limit. This is due to the number of deletion moves available. When determining the best deletion move the AI must check every single

possible deletion and then in the case of two deletions, check every single possible deletion after each the first deletion.

The best case for the alpha-beta function is when the child nodes of max are searched in a highest value first order and the child nodes of min are searched in a lowest value first order. The time complexity for this is $O(b^{d/2})$. The worst case for the function is when the opposite is true, max's children are searched in order of lowest first and min's children highest first. This produces the same result as minimax as every node is visited up to depth d. The time complexity in this case is $O(b^{d})$.

Looking at the results of our tests it appears that it is much closer to the ideal situation of $O(b^{d/2})$. A plot of this with the worst case branching factor of 85 can be seen in grey above. At this branching factor the number of recursive calls jumps from below 1000 to over 7000 calls with a depth increase from 3 to 4 making it perform very poorly as described. It does however perform far better than the Minimax function which has a worst case time complexity of $O(b^{d})$. With the worst case branching factor of 85, the Minimax function with a depth of 2 would produce over 7000 recursive calls.

Analysis

The decision to use the Minimax function with alpha-beta pruning turned out well however a number of improvements could be made to improve the efficiency of the AI. It would've been nice to be able to use a depth of 4 in our algorithm instead of 3. This would allow the AI to look further ahead and produce more tactical moves. In order to achieve this, we could have sorted each of the child states in largest-first order when it is max's turn and do the opposite when it is mins turn. Whether or not this would have a significant difference it's difficult to tell, reducing the branching factor would have a much greater effect.

The branching factor could be reduced by pre-checking each state before it is added to our possibleStates list. It is possible to achieve the same board state from multiple different moves. For example, the board state produced after a promotion can be replicate in a place-then-delete set of moves. It makes no difference to the AI how the board state is achieved just the final board state and corresponding score. This would require some processing as the list would need to be iterated over before every check. It would however likely produce a net gain in performance due to significant reduction in children for every board state. The AI would no longer waste time analysing several identical boards.

Lastly, improvements could be made to our evaluation function getBoardScore. This function was difficult to get right and is fairly basic in how it evaluates the boards. The ability to prevent the opposing player from building lines and squares is a major factor in winning the game. This is an area that could be improved as well as adding more randomness to the AI's moves. Currently the AI is rather predictable and will always start off with the same moves.