**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)*

**2.1.1 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.

**2.1.2 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.

**2.1.3 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 raise (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 (null if no sphere, or if only sphere, is being remove)

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