COMP30260 "Artificial Intelligence for Games and Puzzles"
Assignment 2
Eoghan Cunningham – 16441162.

Program Design:
The source code submitted is in six Java classes. The *Move* and *Player* class are concerned solely with managing user input and maintaining and updating the position of players. All of the game specifics and rules of the Severn Bore game are implemented in the *Board* class. Finally the *AlphaBeta* class contains the implementation of the minimax alpha-beta search with a killer move heuristic while the *AlphaBetaMove* class is essentially a structure for storing moves alongside evaluations and altering move strings to match the desired format.

Representing and printing positions:
The choice of a five-by-five game board allowed for the use of integer bit-boards, as integers in Java have 32bit representation. The game operates with three such bit-boards; one for player one, one for player two and another for the depth charges. The *drawBoard* method in the *Board* class displays the board taking each of these integers as input.

Interpret user input:
User input is received in the *getMove* method of the *Player* class. It is then handled in by the *Move* class which throws exceptions for invalid syntax.

Move validity:
The *possibleMoves* method returns a string of all possible moves that can be made by a particular player. It uses bitwise operations to cycle through and enumerate the position of each surfer. Java's Integer.*numberOfTrailingZero* allows you to turn a bit-board representation of a position into a integer (in this case, 0-24).
The *validMoves* method in the board class takes such a position and a bit board representing all the occupied cells on the board. Using the occupancy bit-board, bit masks for each row, column, diagonal and anti-diagonal and the **(O ^ (O – 2s))** algorithm used in many chess engines it produces a bit-board representing all of the valid positions accessible from the input position. It was necessary to logically AND this bit-board with the inverse of the occupancy bit-board as (of course) chess allows players to capture pieces while occupied squares are strictly off limits in Severn Bore.
The *possibleMoves* method uses the resulting bit-board to produce a string of all of the valid moves. In each case of a valid move it calls the *validMoves* method again from the resulting position to find all of the valid throws. It can be seen in the code that the move of leaving a square and placing a charge on it is added manually to the possible moves string. B1 A1 B1 is an example of such a move.
The possible moves method was understandably vital for the minimax alpha beta search but was also very useful for checking move validity. If the user input move was contained in the resulting string, the move was allowed.

End of game:
The methods described above make end of game detection very simple. If the *possibleMoves* method returns an empty string for a particular player, that player has lost.

Static evaluation function:
Bit masks for the corners and edges of the board made counting such patterns relatively simple. The method *adjacentTo* returns a bit-board where every bit that is set is a bit adjacent to the position indicated in the input. This was used as a bit mask to count surfers adjacent to surfers and chargers adjacent to surfers. Further, if the *validMoves* bit-board of one surfer ANDed with the *adjacentTo* bit-board of his partner is zero then they cannot immediately move next to eachother.

Alpha-Beta with a Killer Move and History Heuristic:
I chose a minimax style alpha beta search to allow all calls to the static evaluation function to be the same, (always evaluating player one's position).

The killer move heuristic works by maintaining a two-dimensional array of killer moves. Essentially there is a list of killer moves for each depth and when a move causes a cut off each element of the array is shifted and the move is added to the first position. The possible moves generated by the minimax is then searched and any of the killer moves and these are promoted to the front of the list. The length of this array (the number of killer moves maintained for each ply) is variable. I did not notice considerable improvement in performance however by maintaining three or four killer moves over two.

The heart of the history heuristic is a hash table with moves as keys and the number of times that moves caused a cut off as their value. The possible moves are scanned and any moves found in the history hash table are promoted. By counting the number of promotions, the front of the moves string (containing only history moves) can be separated and sorted, leaving the moves that caused the most cut offs to expanded first. It can be seen from the results below that this was very effective.

Experiments and Results:

I recorded nine games against the program. I lost them all. The settings for the games were as shown in the table below. It can be seen from the table below that use of the history heuristic brings about a massive decrease in the total number of static evaluations and resulted in a considerable increase in the speed of decisions (which is quite slow at the beginning of game with depth limit of 4).

I should be noted that while the program chose the same move regardless of the heuristic choice, it did not choose the same move for different depth limits. This means that all games played at a given depth are identical but not **all** games played were identical.

Depth:	Total Static Evaluations:		
	No Move Ordering:	Killer Move :	History:
2	7,079	6,680	2,965
3	379,668	282,802	42,391
4	4,800,000	3,900,000	328,893