1. Introduction

This section of the report discusses the software implementation of this project.

The team identified that a need exists in this project for software to control the hardware components, implement the game logic of tac tic toe and artificial intelligence to play tic tac toe while running on the available computation and memory resources of the Lego EV3 brick.

2. System Overview (Objectives)

The software implementation goals for this project can be broken down into three main sections: hardware control, game implementation and game play decision.

2.1 Hardware Control

The main functions the software must perform in hardware control are:

— Localization

— Observe the physical state of the game

— Allow the program to make physical moves on the game board

2.2 Game Implementation

The implementation of the game and game logic are:

— Keep track of game state

— Ensure cheating is not allowed

— Display the state of the game

2.3 Game play

— Decide to optimal move to make based on current game state and return instructions to hardware control to execute the move

3. Design considerations, constraints and assumptions

Based on the data gathered by the team in the ME 101 sensor lab, the team expected (some) inaccuracies from the Lego sensors. The software design should assume some sensor inaccuracies. Since sensor inaccuracies are inevitable, the software should try to minimize the effects of sensor error and be able to work with some level of error to ensure that it can perform in multiple scenarios. To combat sensor inaccuracy, the team ensure to test different possible sensor readings as well as resetting the sensor values frequently to prevent error buildup.

The team decided that the entire program should run on the Lego EV3 brick without any additional computation done off-board. This limited the computation and memory resources available to run the program. Based on released specifications, the EV3 has a TI Sitara AM1808 (ARM926EJ-S core) @300 MHz processor and main memory of 64 MB RAM with MB Flash. This is considerably less processing power and less memory compared to any personal computer released in 2018.

Based on the engineering specifications, the design of the program should insure that the bot never loses against the human player.

4. Design Process

4.1 Data structure Design

The data structure used to keep track of the game state is a 2D array with a dimension of 3 by 3 in a struct. Since the game of tic tac toe is played on a 3 by 3 grid, the most natural representation for the game state was to use a 3 by 3 array. Based on the team’s research, it was determined that passing arrays into functions in RobotC is not supported and a struct was needed to work around this limitation.

Due to the physical orientation of the board and EV3 display, the internal representation of the board has flipped row and column indices. To account for this difference, the scanning function flips the indices of the scanned cells. This insures that the internal storage of the game state is in the correct orientation.

Some variables were initialized as bytes to limit the memory used. During initial testing, the program often crashed due to memory issues and initializing values that did not need the storage size of int as bytes seemed to help with the memory issues.

4.2. Algorithm Design

Minimax with Alpha-Beta pruning was selected as the algorithm to implement the artificial intelligence. Minimax is an artificial intelligence algorithm that can be applied to two player games such as tic-tac-toe, checkers or chess. These games can be classified as zero-sun games since one player wins (+1) and the other player loses (-1) or the game results in a tie.

In a nutshell, minimax searches recursively the best move that leads the maximizing player to win or not lose. Note that game search when using regular minimax is essentially a depth first search. The algorithm simulates the game by employing a maximizer (the player) and minimizer (the opponent). The maximizer attempts to reach a state that gives the highest possible score while the minimizer will make moves that minimize the score. It considers the current state of the game and the available moves for that state and then recursively simulates that game tree until it finds a terminal state (win, draw or lose). The terminal state is then assigned a score by a static evaluator function which evaluates the terminal state and determines if that state is a win, loss or tie and returns a score. For tic tac toe, the static evaluator is hard coded to recognize one of the 8 possible winning positions (3 horizontal, 3 vertical and 2 diagonal) and a win returns a score of 10, loss a score of -10 and tie is a score of 0.

After the first implementation of normal Minimax, it is found that the Lego EV3 does not have enough memory to run all the evaluations and recursive calls that are required by minimax. The program repeatedly crashed due to memory issues. Therefore, it was decided to optimize the algorithm with alpha-beta pruning. Alpha-Beta is a optimization method for the minimax algorithm that can be used to decrease the number of nodes that are evaluated by the minimax algorithm. Alpha-beta are numbers that represent the “fail-safe” option (worst case scenario) of the maximizing player and the minimizing player respectively. As minimax progresses down the game tree, the alpha value is updated at maximizing nodes and the beta values are updated at the minimizing nodes. The optimizing happens when alpha is greater than beta. This represents that the maximizer’s score is higher than the minimizer’s score. Since the minimizer doesn’t want this to happen and will play to avoid this in reality, the subtree of that move does not need to be considered and is pruned. This saves computation both on minimax evaluations and leaf node static evaluations.

Static Evaluation

It is important to note that while in some minimax examples, the leaf node scores are just numbers, in reality, those numbers are obtained by a static evaluator function. In complex games such as chess, this static evaluator can take significant resources to run and any optimization that can reduce the number of static evaluations is a worthwhile effort.

4.3 User Interface Design

The user interface in this project was printed to the EV3 brick’s display and also included a touch sensor. The game board and any relevant prompts that informed the player on how to play the game was printed to the screen. The touch sensor was used to indicate when the user make a move and also to inform the robot that the board was reset if the user wanted to player additional games.

The team experienced some issues printing out certain messages on the screen. It is unclear what the cause of the issues was but is most likely a hardware bug or a memory problem.

5. Testing

Since RobotC is a C-based language, it was decided to implement the minimax algorithm in C for development and testing purposes before moving it to RobotC for the final implementation. After debugging of the algorithm itself was complete, it was tested on the EV3 with a display interface.

After the physical robot was built, the team wrote multiple test functions to test individual mechanical functionalities of the robot. This included separate functions to test the accuracy of the ball dispenser, board and the arm. There is also a separate function that was used to dispense a ball into each board cell to ensure that the robot could accurately place a move in each cell on the board. The team also did some experiments to determine the different values that the colour sensor might read in normal lighting conditions when scanning the green balls, yellow balls or an empty cell to provide some level of redundancy. Then the scanning functionality’s accuracy was validated since the accurate scanning of the board is vital for the robot to play the game.

6. Conclusion (Software Recommendations)

Given more time, it could implement a emergency stop button. Based on the team’s understanding, this feature would have be a lower level implementation since concurrency cannot be directly implemented in RobotC.

It is also possible to implement a timer for resetting the arm and board. This could account for the possibility that the arm or board were possibly missing. If touch sensor is not triggered in 5 seconds, assume that the arm or board is not connected or not attached and stops the motors to save power before stoping the program.

References

https://en.wikipedia.org/wiki/Lego\_Mindstorms\_EV3