AI Project 2
Gerald Hoxha
Bradley Gonthier
Tyler Oliver
Nicholas Van Beek

AI Project 2

## Strategies:

Since there are only a handful of different heuristics that can be used to evaluate the state of Ultimate Tic Tac Toe along with the constraint of using only Minimax + Alpha Beta, the best strategy is to look ahead as many levels as possible. A UTTT AI that can look further ahead has a significant advantage over other AI. A decent heuristic will also be very advantageous.

The Heuristic

A single function evaluates the board and return a score for that board. Points are given to the following patterns:

- 1000 points if a player won 3 big spaces, meaning they won the game
- 100 points of a player has one two big spaces in a row
- 10 points for each big space that a player has won
- 1 point for each group of 2 connected small spaces that a player has.

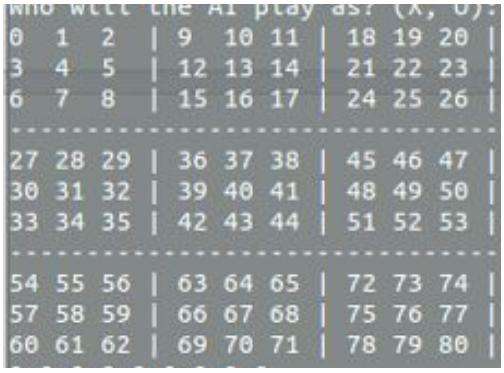
Efficiency

Since our group believed that the strongest AI is the one can look ahead the most levels, efficiency became a big focus. We wrote the entire project in C, giving us a huge boost in performance. We tested a recursive function in javascript versus a recursive function in C and found that C completed calculations nearly twice as fast.

We also designed the game board and workflow with efficiency in mind. Our board consists of two arrays one called subBoard and another called superBoard. Each time a move is made, the doMove function checks to see if a sub-board has been won by a player. If it has, the corresponding subBoard is marked on the superBoard. By doing this, we can prevent the heuristic from checking all of the spaces each time it runs. We can also evaluate a game win by just checking the superBoard, not the entire subBoard.

We also changed the way we created the game board from a 2D array to a single dimension array with 81 values. We did that because it is much more efficient and intuitive to evaluate

different states of the board. Look at a screenshot of the board bellow. If wanted to test to see if a subboard 2 was one diagonally, we could start with 9, check the space, then do 9 + 4, check the space, and do 9 + 4 + 4 and check the space again. If all of those spaces match, then it is a win.




We also tried to minimize the amount of memcopy, memory usage and variable instantiations. This saves precious CPU cycles

Multithreading

With efficiency in mind, the other strategy we used was to multithread the application. Each branch coming off of the root node is sent to a thread. We are seeing speedups linear to the number of cores available on the laptop. So a dual core CPU calculates moves about two times faster than a single core. And quad/octo core CPUs provide about 4/8 times speed ups. This allows us to look maybe 1 or 2 levels further ahead while still having reasonable processing time. Putting the board a 32 core Amazon EC2 cloud computer resulted in speedups of roughly 16 times, as each core is a hyperthread, meaning each core is essentially the equivalent to half a normal desktop computer core. We also tried our hand at GPU acceleration, but did not have enough time.

## The Three AI Types:

Dumb

- Looks 2 moves ahead
- No alpha beta pruning
- Standard heuristic mentioned above

Smarter

- Looks 3 moves ahead
- Alpha beta pruning
- Standard heuristic mentioned above

Smartest

- Looks 7, 8 or even 9 moves ahead
- Depends on speed of the computer as to how fast it is

- Standard heuristic mentioned above
- Some variable depth depending on weather the next move is searching the whole board or just a sub board
- Adjust depth based on amount of spaces left, since with less spaces we can look further ahead.

**Final Notes/TLDR:**
Heuristic is important, but not that important. We try to look as far ahead as possible through efficient coding and some multithreading.

**Compiling and Running**

gcc utttSmartest.c thpool.c -lpthread
./a.out