# Project Proposal

CS5500

Sam Rands and Zion Steiner

April 13, 2021

## Emails


- Sam Rands: samuel.rands@aggiemail.usu.edu
- Zion Steiner: zion.steiner@aggiemail.usu.edu


## Description

For our project, we wrote a parallelized version of the minimax algorithm. Minimax is an algorithm for finding the move that will result in the best worst-case end-of-game result in a perfect information game. Let's define each of those terms.

A perfect information game is one where all players know the whole game state. There is no chance involved (no dice rolling), and players cannot hide information from each other (players cannot hide hand of cards).

A best worst-case result is the best end-of-game outcome (winning, highest score) you can achieve assuming your opponent makes optimal moves each turn. The figure below illustrates this process.

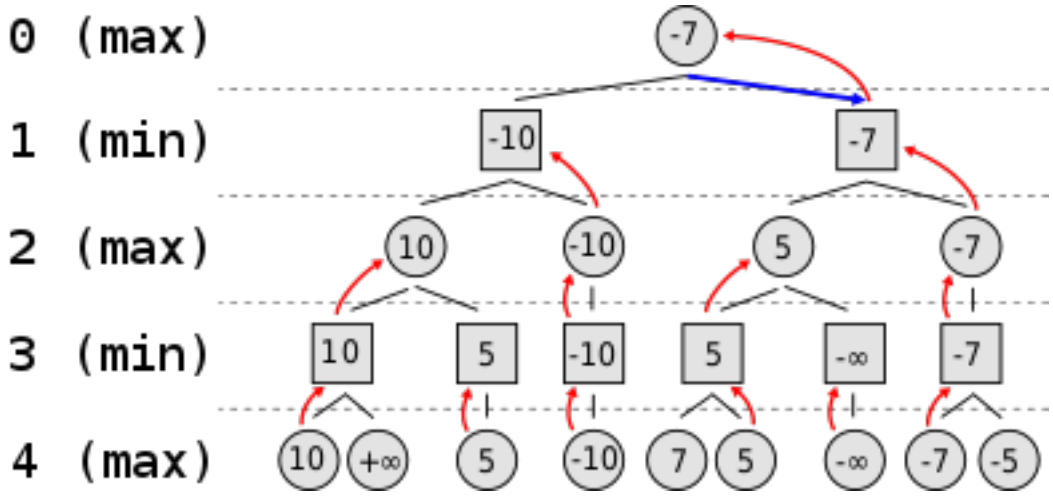


Figure 1: Minimax tree

Our implementation is game-agnostic, so it may be used for any perfect information game. As part of this, we wrote a simple interface that allows minimax to communicate with the game (to access potential moves or a move's value).

Minimax finds the best move by simulating all possible games. Obviously, this is not feasible for even simple ones like mancala, let alone complex games like chess. Chess alone has $10^{120}$ [1] games. Instead, we can find the local best worst-case result $n$ moves into the future. This allows for quick enough algorithm decisions for use as an AI agent in video games.

We parallelized this algorithm by simulating each game subtree with a new process. If a game had 5 initial moves, the leader process distributed the 5 moves between $n$ follower processes. Each process recursively found the best subtree solutions and communicated its best solution back to the leader process. The leader process then compared the moves returned by the follower processes and selected the optimal move.

To test the proof of concept, we wrote a Tic-Tac-Toe game model to interact with the minimax engine. The game model can be configured to use human or minimax players.

# Code

The basic flow of the program is as follows:

1. Init game object
2. Game object inits human player
3. Game inits minimax engine and uses it to init engine player
4. While the game is not over:
   (a) Get currPlayer move. For human player this displays a prompt to the console and receives console input. For the engine player, engine player calls and returns `Minimax.getBestMove()`.
   (b) Update game state
   (c) Switch currPlayer

The main game loop is set up and run in `main.cpp`. When a Minimax instance is created, it uses `MPI_Comm_spawn` to spawn $n$ additional follower processes which run `minimax_worker.cpp`. We decided to use the process spawning and separate cpp file solution because whoever is writing `main.cpp` and their game model should not be expected to know anything about the internals of the minimax implementation. This approach encapsulates this logic well.

When `minimax.getBestMove()` is called, the leader process gets the possible moves from the game model and delegates the computation of their game tree to the follower processes. Each process then reports their best local move back to the leader process, who compares the set of local moves to find the global best move.

## Compile and Run Commands

```
mkdir build && cd build
cmake ../src/
make
mpiexec -np 1 --oversubscribe tictactoe
```

### Listing

Our code is separated into many files, so there is no easy way to include the source code in this document in a readable fashion. However, the source code is included with the project submission and is viewable at https://github.com/zionsteiner/whatsthemove.

For brevity,

```
int main(int argc, char** argv)
{
    TicTacToe game(false, false);
    game.play();
}
```

## Tests

Because the best-worst case outcome for a Tic-Tac-Toe game is a tie, we can verify the implementation works correctly by running a Tic-Tac-Toe game with two minimax players. The outcome of this game results in a tie, meaning our code is sound. Because minimax is a deterministic algorithm, we only have to run this test once to be sure. The output of this test is included below.

## Conclusion

We learned a lot from this project. One large hurdle was serializing classes. We needed to serialize classes to send game states and moves between processes.

We also had to solve the problem of encapsulating engine library logic that uses multiprocessing and hiding it from the library user. We solved most parts of this problem, but there are still several parts to solve. In the end, our goal is to have the algorithm abstracted from the game model implementation such that any user can simply import the Minimax engine and write their own game logic. They should not have to change any library code or know anything about how minimax is implemented, nor that it uses multiprocessing.

Although we have a functional algorithm for Tic-Tac-Toe, there's still plenty of work to do on the project. Here's some future work we have planned:

- Identify and plug memory leaks
- Develop mechanism for registering new game models to existing Minimax source code. This is important for its use as a developer-friendly library.
- Python wrapper
- Minimax optimizations: parallelized alternative to alpha-beta pruning

## References

[1] Claude E. Shannon. Programming a computer playing chess. *Philosophical Magazine*, Ser.7, 41(312), 1959.