CMPUT 655 - Assignment 1 - Team Name: TheSolvers
Taylor Folkersen, folkerse
Zahra Bashir, zbashir1
Fatemeh Tavakoli, tavakol1

We have two acceptable solvers. One of them is easier to implement but slower for large boards (especially in worst-case scenarios). The other one is faster and more complicated which is our main solver. Our finalized code and results are all based on the main solver.

The Main Solver: Iterative Deepening + Heuristic Evaluation + Transposition Table

Our solver uses an iterative deepening approach, with a heuristic score for non-terminal leaf nodes. The heuristic score of a node is the number of moves the opponent with respect to that node has, times -1. Our iterative deepening starts with a maximum depth of 1 and increases by 1 each iteration. We observed that for larger max depths which don't include a proof or disproof of the root, the solver would needlessly evaluate all of the nodes up to the current max depth, which was very wasteful. To address this, the solver will only visit up to a certain number of leaf nodes at its current max depth, before stopping the search and increasing the max depth. The maximum number of paths allowed to reach the max depth also increases with each increase to max depth, by 100, and after 150 increments, this limit on the maximum number of leaf nodes is removed, and the search is allowed to complete uninterrupted. The transposition table stores the heuristic value, or actual outcome of a node, if known, as well as the best move for the player at that node, according to the heuristic. The heuristic aims to maximize the current player's moves at the next state considered. The transposition table has a fixed amount of space, and only one element per hash. States closer to the root node get priority in the table. This lets the search follow a (hopefully) good path, according to the heuristic, from the root. This combination of limited depth and limited paths to leaf nodes allows for quick and increasingly accurate updates to the heuristic value of states. The transposition table is somewhat limited due to storing each tile of the board as a byte, to verify hits/misses for table lookups. We could achieve better performance by packing 4 board tiles into 1 byte, decreasing the size of each table entry and allowing for more entries, but for some unknown reason, doing this packing changes the behaviour of the program resulting in worse performance.

The other one: Negamax + Transposition Table

The transposition table we used here is simple (it only saves player and outcome). Because of using TT, it is faster than simple Negamax and Minimax. It usually works good because clobber is not a very complex game, and in the most of the cases that we tried, the move ordering was not bad.

Other solvers:

- Minimax: We chose Negamax over minimax because Negamax is in a form that is more understandable by machines while minimax is more understandable by humans. In minimax, we had 2 functions and this made the process a little bit slower but not noticeable.
- Simple Iterative Deepening: First, we implemented this algorithm, but then it was improved using a suitable heuristic and TT which happened to be our main solver.
- Proof Number Search: We did not use PNS since our solver was already fast enough, and also we thought that in clobber, we cannot truely get advantage of the PNS features.

How to run: (mentioned in results.log as well)

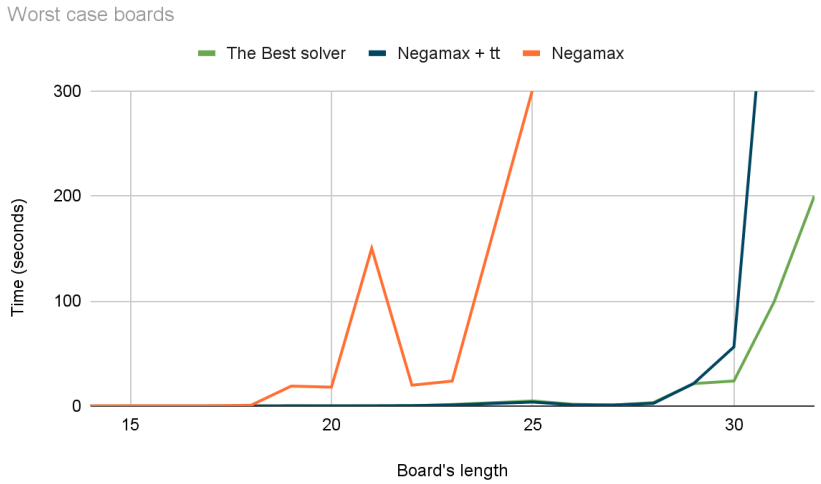
Our test file is written in python, and it will use a text file including test cases named "tests_small.txt". We tested several other test cases by just changing the file name. You can run make to compile the cpp code and then using "Python3 TheSolvers.py" you can see the results.
If the process exceeds the given time limit, it will return '?' as the result, and prints the number of nodes.

Experiments/Evaluations:

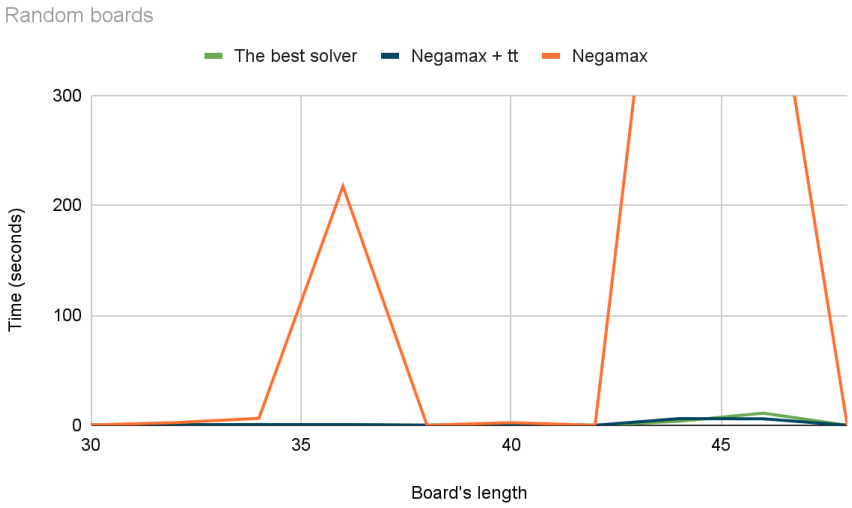
Chart 1: This chart is dedicated to running different solvers for the worst test cases.
***The worst-case*:** The format of the worst-case test is as follows: BWBWB....BW, because it has a large branching factor. When the branching factor is large, the tree becomes larger, with more branches and nodes, thus, searching through the tree to the terminals nodes would be time-consuming.
As you can see, the best solver performs better than all of the other solvers specially when the length increases. For boards with the length of less than 27, our best solver and Negamax+tt solve the board in less than 10 seconds (usually even less than 5s).
For the smaller lengths, usually both the best solver and Negamax+tt perform closely, but for boards longer than 29, our best solver considerably is faster than Negamax+tt .

Chart 2: The results on randomly generated boards depend both on the complexity and the length of that board, but here we only plotted the time based on the length. That is why we see many fluctuations in the time. But, usually for longer boards, the run time increases. This chart shows the comparison between our best solver, Negamax+tt, and Simple Negamax. Originally we also had included results from Minimax, but it was very close to the Negamax, so we removed it. Although the simple Negamax performs very badly on some random boards, our best solver and Negamax+tt solve them very fast that even are not noticeable in the chart.

Results on the Lab machine (ohaton):
For more details on the exact numbers please visit the links. The simple Minimax and Negamax results were the same so we only drew one of them.



Link: Line chart 1 -Results for worst case boards



Link : Line chart 2 - Random Boards