CMPUT 655 - Assignment 2 - Team Name: TheSolvers
Taylor Folkersen, folkerse
Zahra Bashir, zbashir1
Fatemeh Tavakoli, tavakol1

Method:
In this assignment, we tried to improve our previous solver using Combinatorial Game Theory Techniques. It is to be noted that our best solver (in assignment1) was an Iterative deepening search with a transposition table that stores heuristics, best moves, results, etc.
We use an endgame database that saves the outcome (L, R, P, N) of all boards up to 18 tiles. The creation of this endgame database takes about 25 minutes. A database of up to 16 tiles takes about 25 seconds to make.

Our search steps and techniques are as follows:
First, we simplify the state by finding subgames and removing pairs of reverse subgames, and any remaining zero subgames.
After simplifying the state, we sort the subgames in decreasing order of the number of tiles. We look for each subgame's outcomes in the database. Then we consider some special cases:

- If all the subgames' outcomes are L (Black wins), we return the state's result as Black and also save it in our transposition table.
- If all the subgames' outcomes are R (White wins), we return the state's result as White and also save it in our transposition table.
- If we only have one subgame and its outcome is N (First Player Wins), we return the current player as a winner and also save it in our transposition table.
- However, if none of the above happens: if one of the subgame's outcomes is L (or R) and the current player color is Black (or White), we delete this subgame from the state (substitute it with dot) and search on this the new board. If our solver can solve this new board and its result is the same as that removed subgame (L or R), we can conclude that the whole result of this game is L (or R), and return it.
    - Why do we do this? If we remove one subgame, the rest of the board is usually faster to search, and if it has a determined result, we will get the whole game's result much faster. We added this check as we observed that this is a common case in clobber.

Other Improvements:
If the above optimizations don't solve a state, then the search will play moves from this state in the following order: the previously stored best move according to the heuristic evaluation function (the negative of the opponent's number of moves), moves that are in a subgame having an opposing outcome class (R if the state's player is black, L if the state's player is white), and then every other move, in random order. This move randomization helps break up subgames that are larger than those in the database, by removing the bias towards picking moves on one side of the search tree, leading to faster search. As with our previous solver, the search uses iterative

deepening, with a limit on the number of searches that are allowed to reach terminal nodes (nodes at the current maximum depth, or actual terminal nodes). As the maximum depth increases, more searches are allowed to reach these terminal nodes, and after some number of iterations, search is allowed to complete uninterrupted, with unlimited depth.

Experiments:

Our new combinatorial approach is much faster than the previous one. In most of our test cases including very large boards (even for $(WB)^n$ boards), we observed considerable improvements. We have two experimental settings: in the first one, we create 4 groups of 150 randomly generated boards. We compute (on the Ohaton server) the overall time taken (in seconds) to solve all these boards and report the average time over all these groups. As the previous solver is not that fast compared to this one, in order to have equal comparison criteria, we limit the size of the randomly generated boards from 2 tile boards to 40.

Solver	Group1	Group2	Group3	Group4	Average
Full board	35.67	16.79	43.58	61.77	39.46
Combinatorial	0.49	0.57	0.63	0.47	0.54

Table1: Results on groups of 150 boards. The Average column is the average time (seconds) it takes to solve a group of 150 boards.

$(WB)^n$ cases:

In another setting, we consider the special case of $(WB)^n$ boards with different lengths to compare our full board solver vs our combinatorial solver. We have computed solvers' runtime for up to 48 tile boards (you can see more runtime results by clicking on the chart).

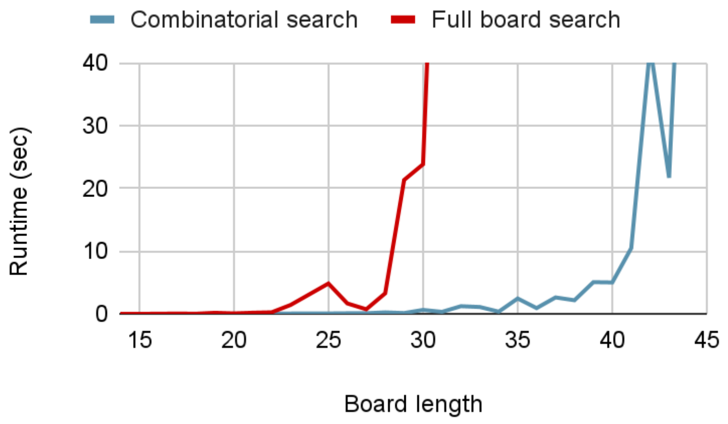

Chart1: Runtime results for worst-case boards in the form of $(WB)^n$