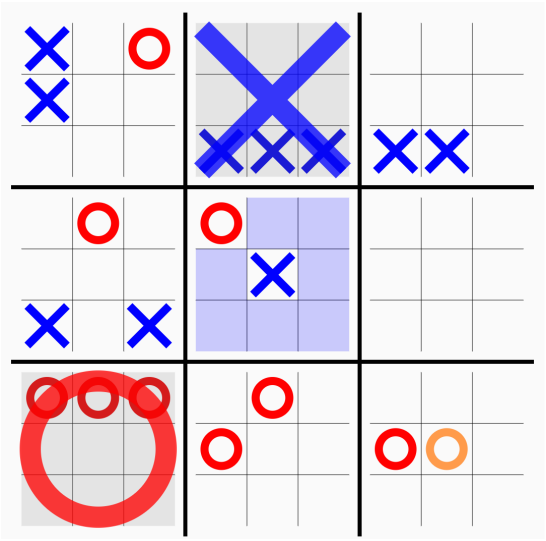
UTTTSolver: Ultimate Tic-Tac-Toe Solver Report

Fernando Macchiavello Cauvi (FM2758), Hasan Alqassab (HA2613)

Overview and Game Rules:

Ultimate Tic-Tac-Toe (UTTT) is a variant of the popular game Tic-Tac-Toe. UTTT is set up by having a 3x3 Tic-Tac-Toe board with each cell containing an interior Tic-Tac-Toe board:



“Just like in regular tic-tac-toe, the two players (X and O) take turns, starting with X. The game starts with X playing wherever they want in any of the 81 empty spots. Next the opponent plays, however they are forced to play in the small board indicated by the relative location of the previous move. For example, if X plays in the top right square of a small (3 × 3) board, then O has to play in the small board located at the top right of the larger board. Playing any of the available spots decides in which small board the next player plays.

If a move is played so that it is to win a small board by the rules of normal tic-tac-toe, then the entire small board is marked as won by the player in the larger board. Once a small board is won by a player or it is filled completely, no more moves may be played in that board. If a player is sent to such a board, then that player may play in any other board. Game play ends when either a player wins the larger board or there are no legal moves remaining, in which case the game is a draw.” (Orlin, Ben (June 1, 2013). "Ultimate Tic-Tac-Toe". *Math with Bad Drawings*.)

Implementation:

In this two-player zero-sum game, the player and the AI agent are in direct competition, each striving to win the game. The AI agent aims to maximize the objective function, while the player aims to minimize it, having no shared interests between them. Typically, such games are effectively solved using either the Minimax algorithm or the Monte Carlo Tree Search (MCTS) algorithm.

Monte Carlo Tree Search (MCTS):

MCTS is a more modern approach that uses randomness to simulate game outcomes. It builds a search tree by randomly sampling moves in the game, then uses the results of these simulations to estimate the most promising moves. MCTS is particularly effective in games with large search spaces where traditional exhaustive search methods like Minimax are impractical. It balances between exploring new, potentially promising moves and exploiting known, successful strategies.

In our first attempt to build the AI agent using MCTS, we found that it was challenging to implement it in Haskell, particularly because of the immutability nature of Haskell. We found that updating each node of the tree after simulation was very inefficient because we are left with two options: keeping Haskell's immutability and rewriting a large chunk of the tree after each simulation or using a mutable data structure and taking away the pure functionality that Haskell thrives on, especially in a parallel setting.

We then proceeded to implement the AI agent using the minimax algorithm.

Minimax Algorithm with alpha beta pruning:

This approach involves envisioning each move as a pathway to a new set of possibilities, much like branches sprouting from a tree. Each branch, or node, in this metaphorical tree represents a potential future state of the game. The real challenge for any player, or in our case, the AI, is to assess these states and determine which ones are most advantageous.

To navigate this tree effectively, the AI uses a method called heuristic evaluation. It's a bit like assigning a score to each potential outcome, with higher scores indicating scenarios more favorable to the AI. This scoring system is essential for the AI to make informed decisions, particularly in games where the range of possible moves is vast.

In games with an incredibly large number of potential moves, such as Ultimate Tic-Tac-Toe, examining every possible outcome is not feasible. So, instead of trying to analyze the entire tree, the AI limits its exploration to a certain depth. Beyond this point, it relies on evaluating the game states it has reached. This evaluation is crucial as it guides the AI in choosing the most promising path forward.

An important aspect of this evaluation function is its balanced nature. It assigns positive values if the AI is in a winning position and negative values if it's losing, with the magnitude indicating the strength of either position. The values are relative and used mainly for comparison purposes.

To enhance the efficiency of this strategy, especially in games with numerous possibilities such as this game, a technique known as Alpha-Beta pruning comes into play. This method cleverly cuts down the number of scenarios the AI needs to consider, streamlining the decision process without missing out on the best moves. It's a crucial tool for managing the computational demands of complex games.

Heuristic Evaluation in Ultimate Tic-Tac-Toe:

The game's AI utilizes a sophisticated heuristic system to evaluate both its and the player's positions on each of the smaller, interior boards. The scoring system assigns points as follows:

- Horizontal or Vertical Alignment (2 in a row): Achieving this alignment scores 6 points.
- Diagonal Alignment (2 in a row): This alignment is slightly more valuable, earning 7 points.
- Blocked 2 in a Row: A strategic position where an opponent's 2 in a row is blocked scores 9 points.
- Straight 3 in a Row: Completing this sequence garners 12 points, reflecting the significance of winning an interior board.
- Positional Play: Emphasis is placed on occupying center and corner squares, as these positions are often strategically advantageous. They help the AI avoid falling for traps.

The sum of these values from each interior board is calculated to determine the overall board value.

When considering the main board, a similar point system is applied. However, the stakes are

higher here, with each point multiplied by 150. The ultimate goal of winning or losing the game carries the highest weight, being assigned a value of 1000 points. This significant score ensures that the primary objective—winning the game—supersedes all other tactical considerations.

This heuristic approach that we used allows the AI to make calculated decisions, prioritizing moves that strategically advance its position while countering the player's moves.

Parallelization Design:

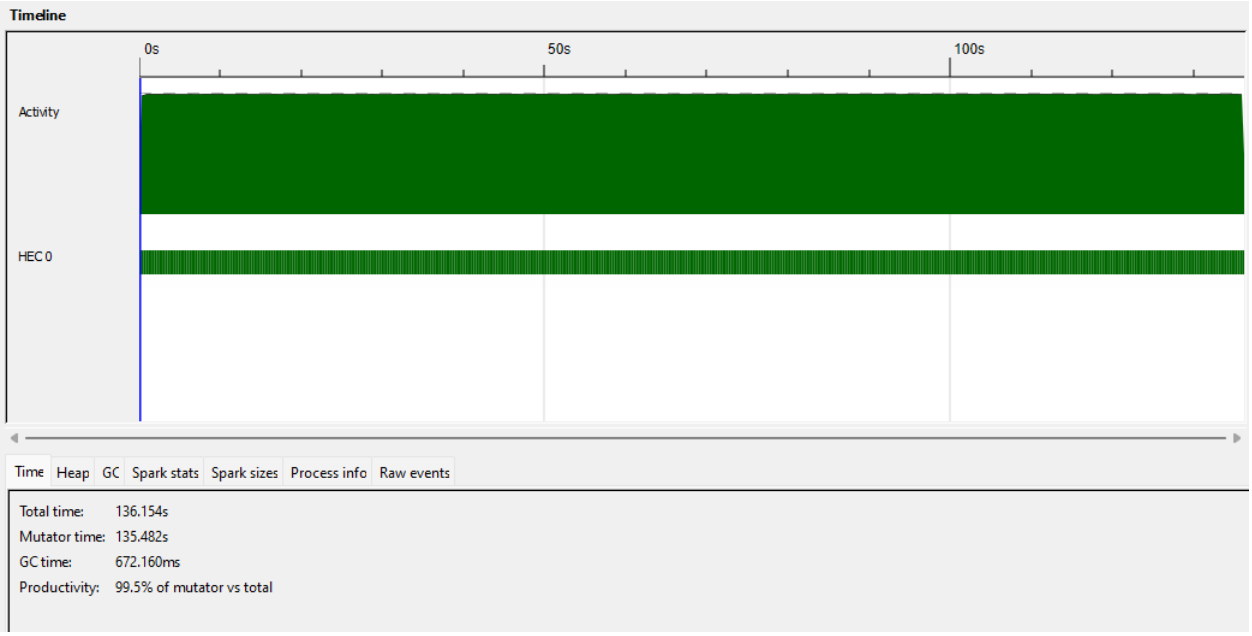
In our implementation of the AI for Ultimate Tic-Tac-Toe, parallelization is only applied to the first move choice (depth of 1). The remaining calculations and simulations are done sequentially for each individual thread. While this method isn't theoretically optimal in terms of full parallelization, it strikes a practical balance between computational efficiency and resource management, considering the limited availability of CPU cores in our machine and how many possible moves there will be on an average board state.

As the AI traverses deeper into the minimax tree, the benefits of parallelization start to diminish. At these deeper levels, the introduction of more parallel processes wouldn’t increase the speed, but rather add to overhead to the program. This diminishing return from parallelization at deeper levels is a key consideration in our parallelization design.

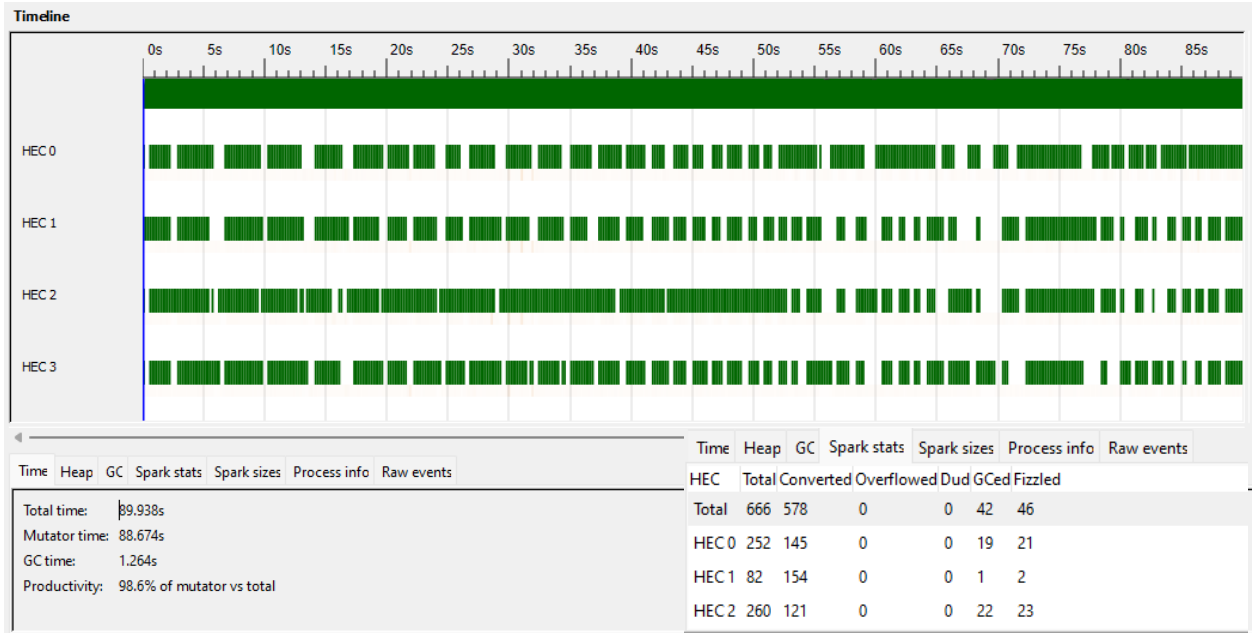
We utilized 'parMap', capitalizing on the fact that each branch of the tree operates independently and requires substantial computational resources. We also utilized 'deepseq' to force the evaluation of each branch in the attempt of reducing possible 'thunk' (delayed computations) in each branch of the tree.

Parallelization Results:

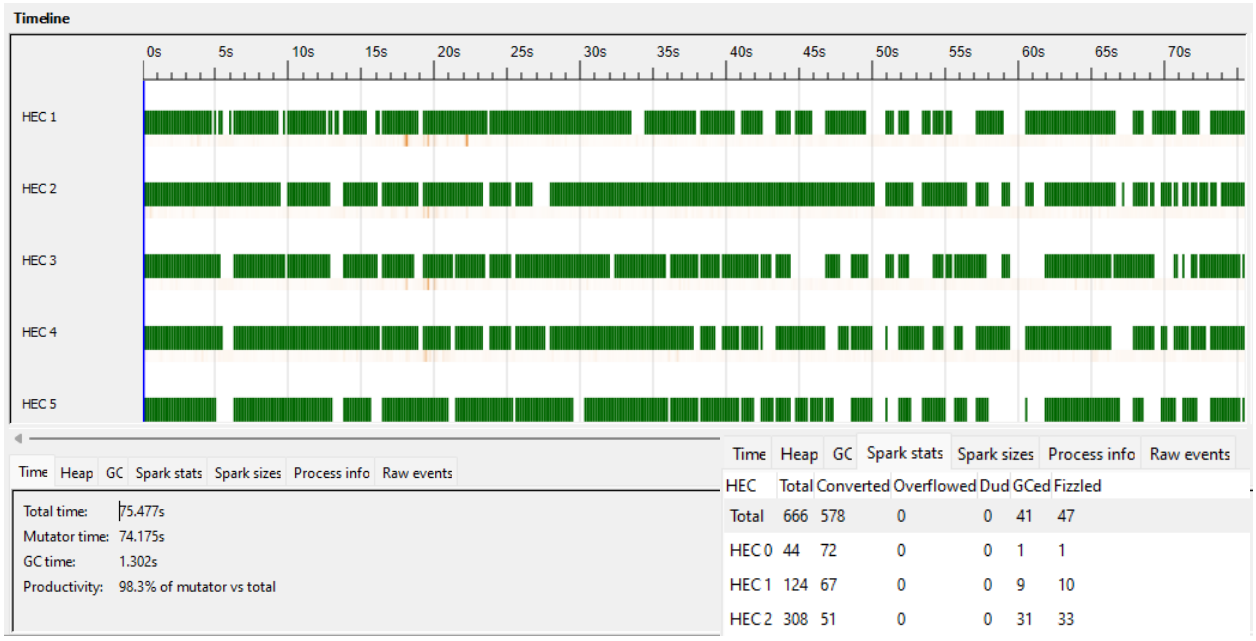
Sequential:

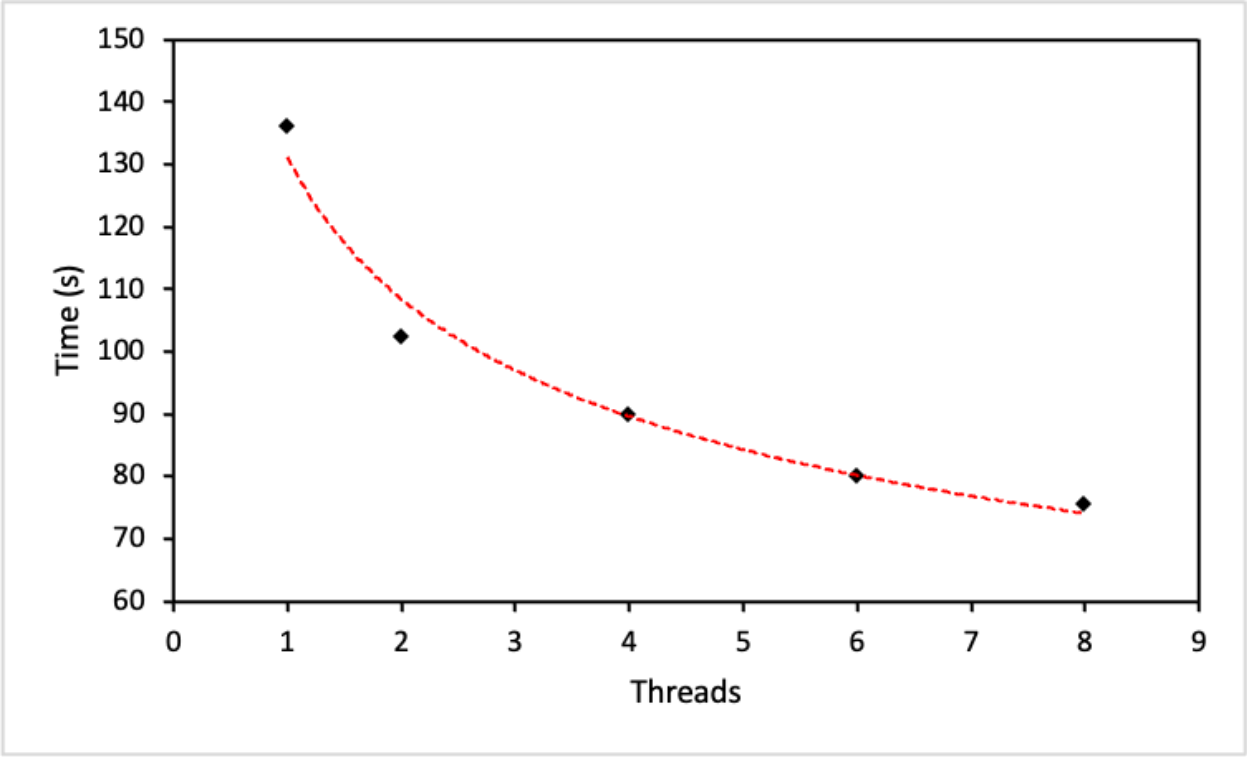


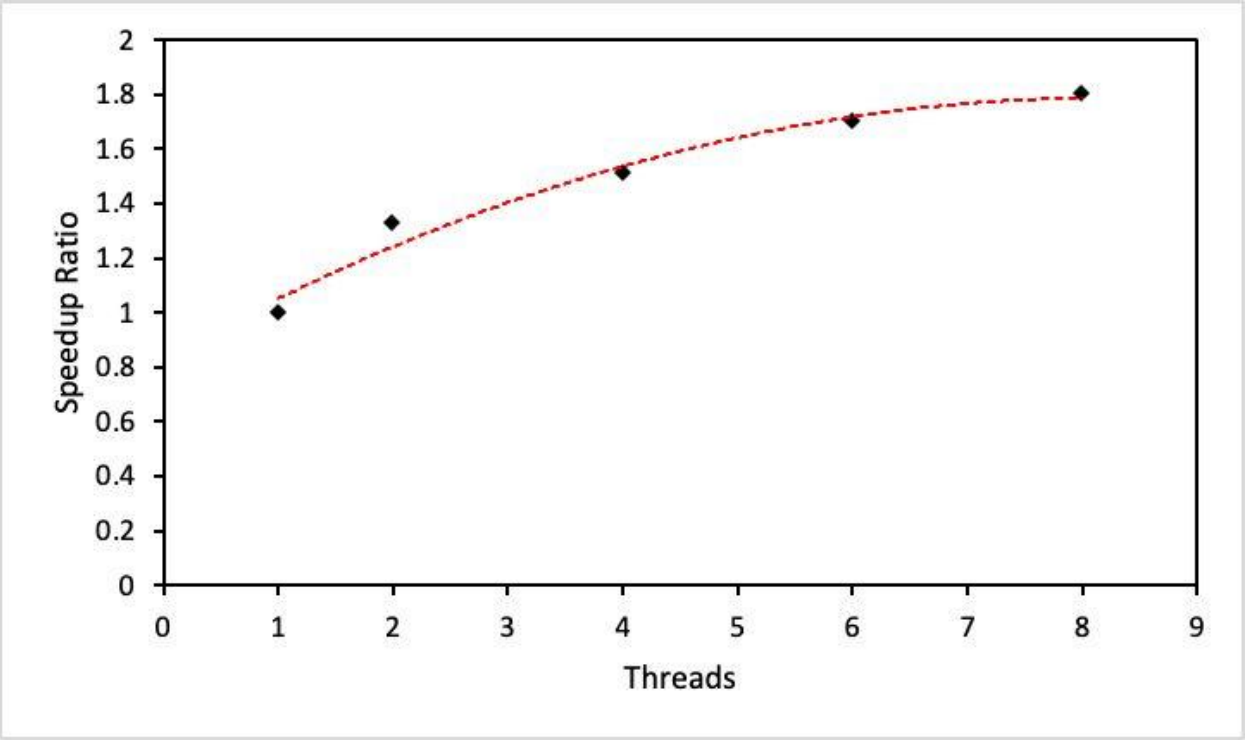
4 threads:



8 threads:


Timeline
0s 5s 10s 15s 20s 25s 30s 35s 40s 45s 50s 55s 60s 65s 70s
HEC 1
HEC 2
HEC 3
HEC 4
HEC 5
Time Heap GC Spark stats Spark sizes Process info Raw events
Total time: 75.477s
Mutator time: 74.175s
GC time: 1.302s
Productivity: 98.3% of mutator vs total
Time Heap GC Spark stats Spark sizes Process info Raw events
HEC Total Converted Overflowed Dud GCed Fizzled
Total 666 578 0 0 41 47
HEC 0 44 72 0 0 1 1
HEC 1 124 67 0 0 9 10
HEC 2 308 51 0 0 31 33


150
140
130
120
110
100
90
80
70
60
Time (s)
0 1 2 3 4 5 6 7 8 9
Threads



The graphs presented offer a clear visualization of the time taken for two AI agents to complete a game of Ultimate Tic-Tac-Toe against each other, varying with the number of parallel threads (at a depth of 6). The data shows that the addition of more threads initially results in a significant reduction in game duration (game duration is a measure of how fast does that AI produce a move because the number of moves in a game is constant across all numbers of threads). This effect is most pronounced when the number of threads increases from one to four, illustrating a sharp decrease in the time taken for the AI to make a move

However, beyond the point of four threads, the marginal time saved begins to taper off. This diminishing return is attributed to the lower branching factor and reduced depth encountered towards the end of the game. Furthermore, as the number of threads continues to increase, an overhead associated with managing these threads becomes apparent. This overhead, a consequence of juggling multiple threads, can lead to a noticeable increase in the total time required for the AI agents to play the game. This observation underscores the balance between parallel processing efficiency and the practical limits of thread management in complex AI applications like Ultimate Tic-Tac-Toe.

Acknowledgements:

UTTT Heuristics adapted from https://github.com/zesardine/UltimateTicTacToeAI