Algorithms for Solving Battleship Solitaire Puzzles

Stephen Stefanidis - 7140030
Brock University

April 2025

1 Problem Description

For the final project in 4P03, I decided to look at three different algorithms to solve the Battleship Solitaire puzzle. I implemented a **backtracking** algorithm, a **backtracking with forward checking** algorithm and a **genetic algorithm**.

In my proposal, I only mentioned two algorithms but while solving some puzzles by hand, I was inspired to incorporate a more advanced version of backtrack that would attempt to solve the puzzle like a human first (more on that later). My experiments consisted of making my own battleship solitaire solver/gui with Pygame and finding puzzles using the website https://www.puzzle-battleships.com/ to try 7 puzzles from each category listed on the sidebar (6x6,8x8,10x10,15x15 with easy and hard for each size). In the end, the 15x15 ended up taking too long to solve so I stuck with just three puzzles for those two categories.

1.1 Battleship Solitaire

Battleship Solitaire is a puzzle based on the classic multiplayer game Battleship. The goal is to place a given number of ships on the board and to make sure the constraints given on each column and row are met. Also, the ships should not touch other ships horizontally, vertically or diagonally. One thing I neglected to mention in my proposal is that in most

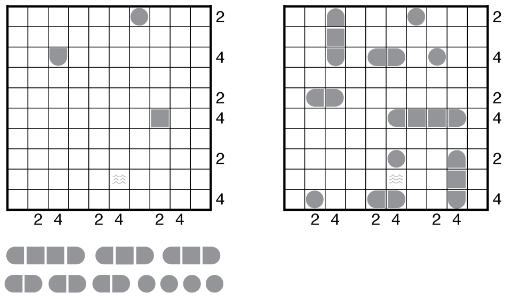


Figure 1: Example Puzzle
https://www.gmpuzzles.com/blog/battleships-rules-and-info/

puzzles some hint pieces will already be placed in order to make the puzzle easier for humans to solve. If an end-piece of a ship is a hint piece (e.g. a rounded side), it's safe to assume that you can place at least one ship next to it.

2 Backtracking

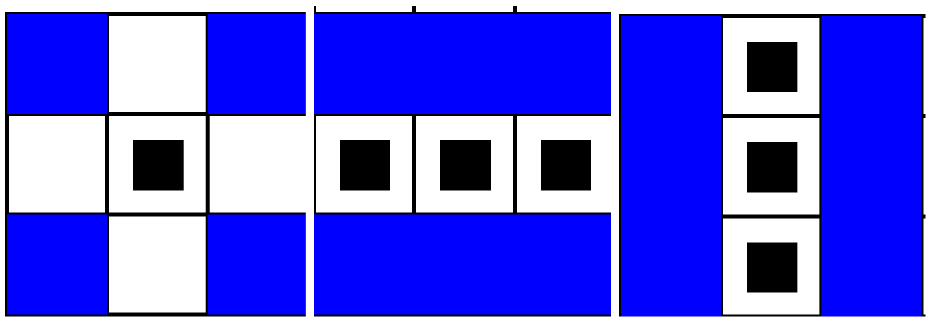
The first algorithm I implemented was a naive backtracking algorithm. This performs much like a sudoku solving algorithm, but while that one places numbers from 1-9 on the board, this program will place either a ship piece or water. Without discussing the specifics of the validity checks yet, the general pseudocode is this:

```
def naive_backtracking():
    def solve(row, col):
        if board_is_filled():
            if is_solved(board, row_numbers, col_numbers):
                print('Solved!')
                return True, board
            return False, None
        next_row = if (col == size - 1): row + 1
                   else               : row
        next_col = if (col == size - 1): 0
                   else               : col + 1
        if (board[row][col] is a hint piece):
            return solve(next_row, next_col)    # skip over this cell
        for cell in ("s", "w"):                 # try both ship and water
            if is_valid_move(row, col, cell, row_numbers, col_numbers):
                board[row][col] = cell
                if solve(next_row, next_col)[0]: # this means propagate the
                    return True, board     # success down the recursion stack
                board[row][col] = "e"      # backtracking step
        return False, None
    result = solve(0, 0)
    return result
```

2.1 Validity Check

This was the approach I took with my backtracking algorithm, but defining the is_valid_move function ended up being more interesting than I thought. At first, I thought the only thing to check for was if the row/column constraints would be broken by placing the ship, but when I finished the algorithm, it ended up being more complicated. The final solution checks for:

- Placing Ships
  - If Row/Column number is 0: do not place
  - If a ship is above and it is either
    - A known horizontal ship
    - A single ship
    - A hint bottom end-piece for a vertical ship
    - Do not place
  - (similarly for ships being below, left and right of the cell)
  - If any ship is diagonal: do not place
  - If the row or column has too many ships already: do not place
  - If this is the last placable cell of the row, and the number of ships is two less than the desired number: do not place
  - If the board has an invalid ship configuration (e.g. a 6-length ship or too many of one type): do not place
- Placing Water
  - If this is the last placable cell of the row, and the number of ships less than the desired number: do not place
  - If a hint end-piece asks for a ship in this cell: do not place
- Middle Ship Logic
  - One last optimization is that middleship (MS) hint pieces are actually more revealing then they might seem.
  - While the four corners around a MS will be dealt with by the diagonal rule above, only two possibilities exist for a middle ship.
  - Using this, I could add another validity check for water and ships to make sure that only these possibilities can happen



2.2 Backtracking Final Thoughts

Overall, the backtracking algorithm is simple to program and is complete so it will always arrive at the correct answer assuming the puzzle is possible and you don't ask it to prune a valid 'path'. As the graph shows below, the time to solve increases drastically as the board size increases, with 15x15 easy puzzles being unusually difficult compared to the 15x15 hard ones. Though I believe this to just be unlucky puzzle picking on my part.

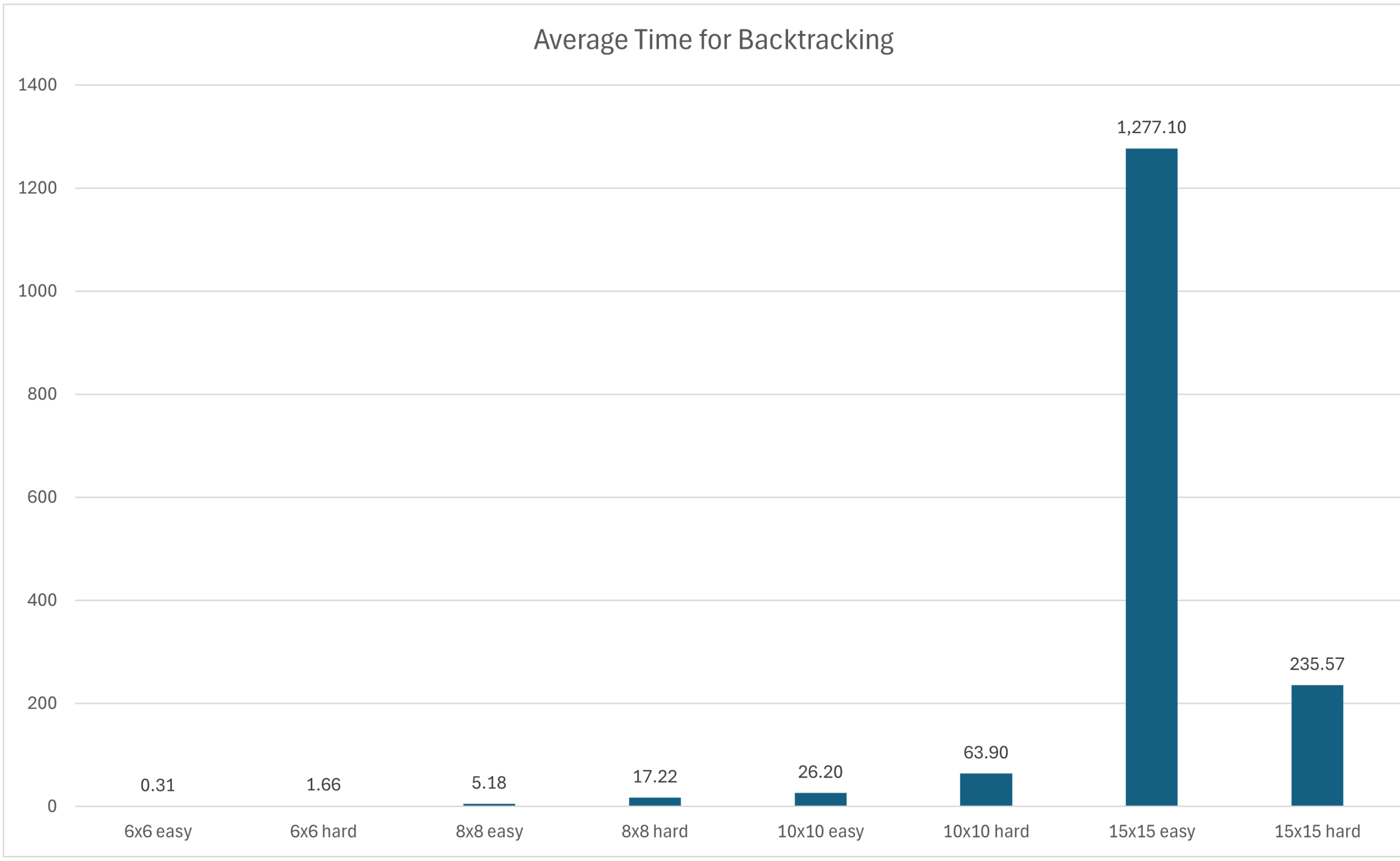


Figure 2: Backtracking Time Over Difficulty Graph

3 Backtracking with Forward Checking (A.K.A. Backtracking Plus)

This algorithm is an extension of backtracking that makes some educated guesses or guaranteed placements before continuing with the normal backtracking algorithm described above. When a human solves battleship solitaire puzzles, they do not randomly place ships and water and reverse mistakes like a computer might, instead they use the information given to solve the puzzle little by little.

3.1 Algorithm Procedure

This is the process that the algorithm takes to make guaranteed placements.

1. Go cell by cell and look for a ship, for each ship, fill in guaranteed water around the ship. If a hint end-piece is present you can also place a ship next to it
2. Go through the rows and columns, seeing if any row or column can be completely filled with ships or water.

After the program goes through rows and columns without making a change, it will 'lock' in the pieces placed and use backtracking to solve the unknown cells.

3.2 Forward Checking Final Thoughts

This algorithm ended up being the best by far in speed. Rarely, normal backtracking was a bit faster than this version (especially true in 6x6 puzzles), due to the normal backtracking not needing to iterate through rows and columns at the start. In the long run, this algorithm is far superior over naive backtracking. Since it is only an optimization on top of the backtracking algorithm, it is also complete. It is also super satisfying to watch for some puzzles because it will just instantly place a lot of the pieces down that normal backtracking would struggle with.

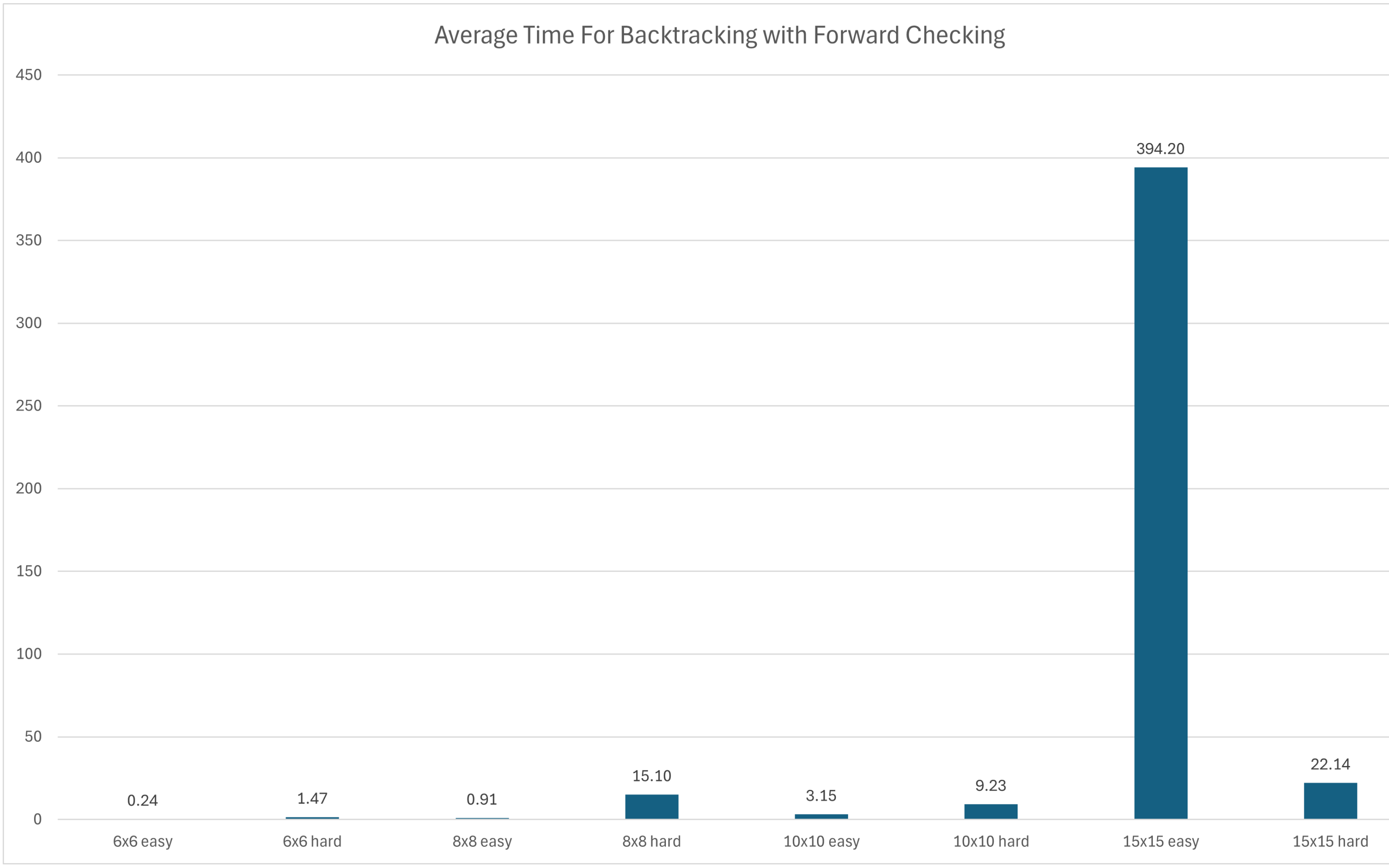


Figure 3: Backtracking Plus Time Over Difficulty Graph

4 Genetic Algorithm (GA)

Because most of my attempts using the GA led to premature convergence, I ran each board 5 times with the GA or until it solved the puzzle.

4.1 Chromosome Representation

To represent the chromosomes for this algorithm, I used a list of ships with a length, orientation, and coordinates attached to each ship class.

4.2 Selection Method

I used the 4-tournament selection method where four chromosomes are randomly sampled from the previous generation and fight to the death (evaluated by their fitness value) to decide which one moves on. This is repeated until the next generation is the same size as the previous one. I also implemented elitism where the top 5% solutions based off fitness are brought to the next generation without crossover or mutation.

4.3 Fitness

I used a sum of errors fitness where a lower fitness score is better, and a correct solution is 0. The penalties are:

- Overlapping : 10 points per ship
- Touching : 100 points per ship
- Breaking a hint: 1000 points per hint broken
- Row or column doesn't match the number: 50 points per row/column

I went with these values because I really didn't want to see solutions that ignore the hint pieces or have touching ships so I prioritized punishing those mistakes.

4.4 Crossover

I used one point crossover where a random pivot point from 1 ... n (where n is the number of ships in a chromosome) is chosen and it will swap children genes after the pivot point. The crossover rate was 100% because this is what worked for solving 6x6 puzzles

4.5 Mutation

I used a mutation where all of the ships are randomly moved around. This contradicts my presentation where I wanted to pick one ship to move because I found not enough variance was being introduced and solutions were converging far too quickly. My mutation rate was 5%.

4.6 Premature Convergence

In order to try to stop premature convergence, I used a large population size of 2500 and a very aggressive mutation function but unfortunately this did very little to improve the results. I also implemented a check to end the execution early if the best solution has not changed within 10 generations.

4.7 Genetic Algorithm Final Thoughts

In the end, the Genetic Algorithm did not end up working too well and I would not recommend using my implementation of a Genetic Algorithm to solve any battleship solitaire puzzle. Even on puzzles with no hint pieces the GA failed to converge to a solution. The only puzzles it solved were 6x6 and it did not succeed every time and took way longer than the backtracking algorithms. The only time I would use a GA for a similar problem is if all row/column constraints are not necessary and you only need to place the ships so they're not touching as most final solutions could at least get that much right.

Most unrelated problems where GA's can work nicely are ones where a 'close enough' solution can be easily modified to be correct or where the constraints are 'soft' such as course scheduling. Battleship Solitaire is not a great application because even solutions that seem really close can probably not be easily modified to find the correct solution. Note that in the graph below, the time is the total for 5 attempts (unless one of the attempts solved the puzzle) and that the algorithm was only able to solve 6x6 puzzles.

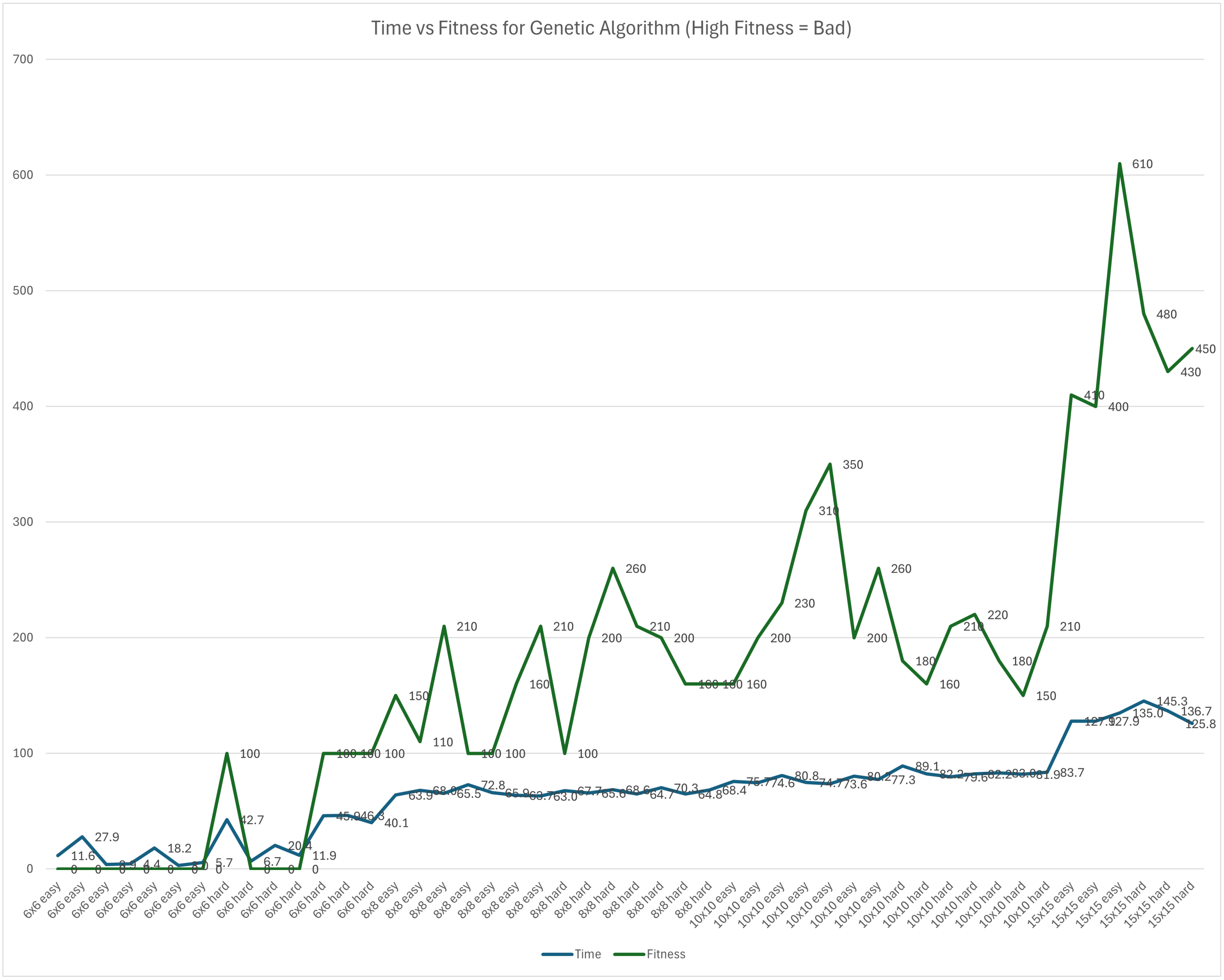


Figure 4: Genetic Algorithm Time Over Difficulty Graph

5 Conclusion

In conclusion, I would recommend using the version of backtracking with forward checking for 99% of Battleship Solitaire problems under and including 15x15 grids. The 1% of puzzles that were technically solved faster by naive backtracking were so easy that the overhead of forward checking caused the algorithm to lose.

One puzzle in [15x15 easy] (3.bs if you're curious) took naive backtracking 40 minutes and improved backtracking 20 minutes to solve which was a notable outlier. The next longest puzzle was 16 minutes for basic backtracking which took only 3 seconds for the improved algorithm.

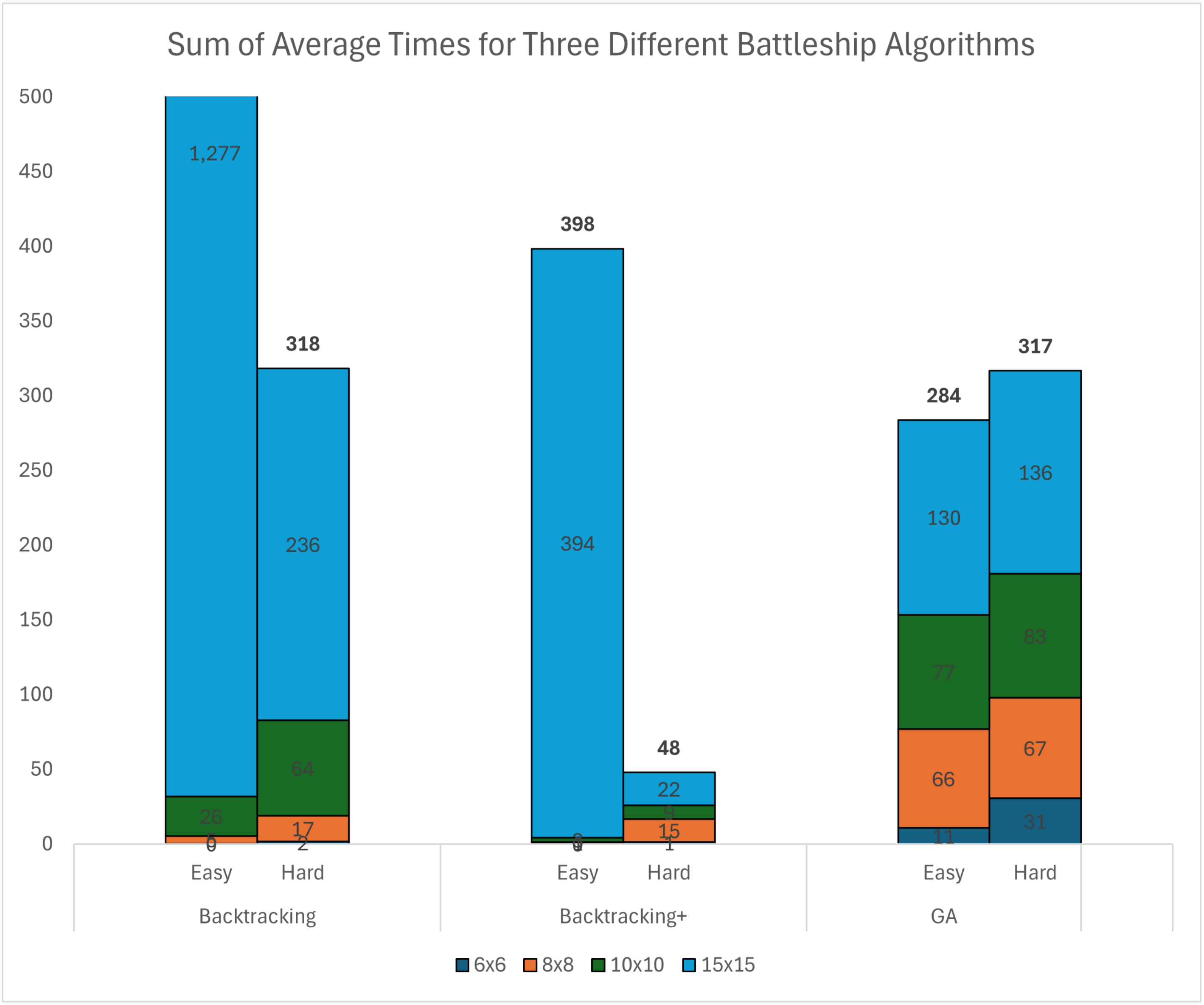


Figure 5: Overall Comparison By Time

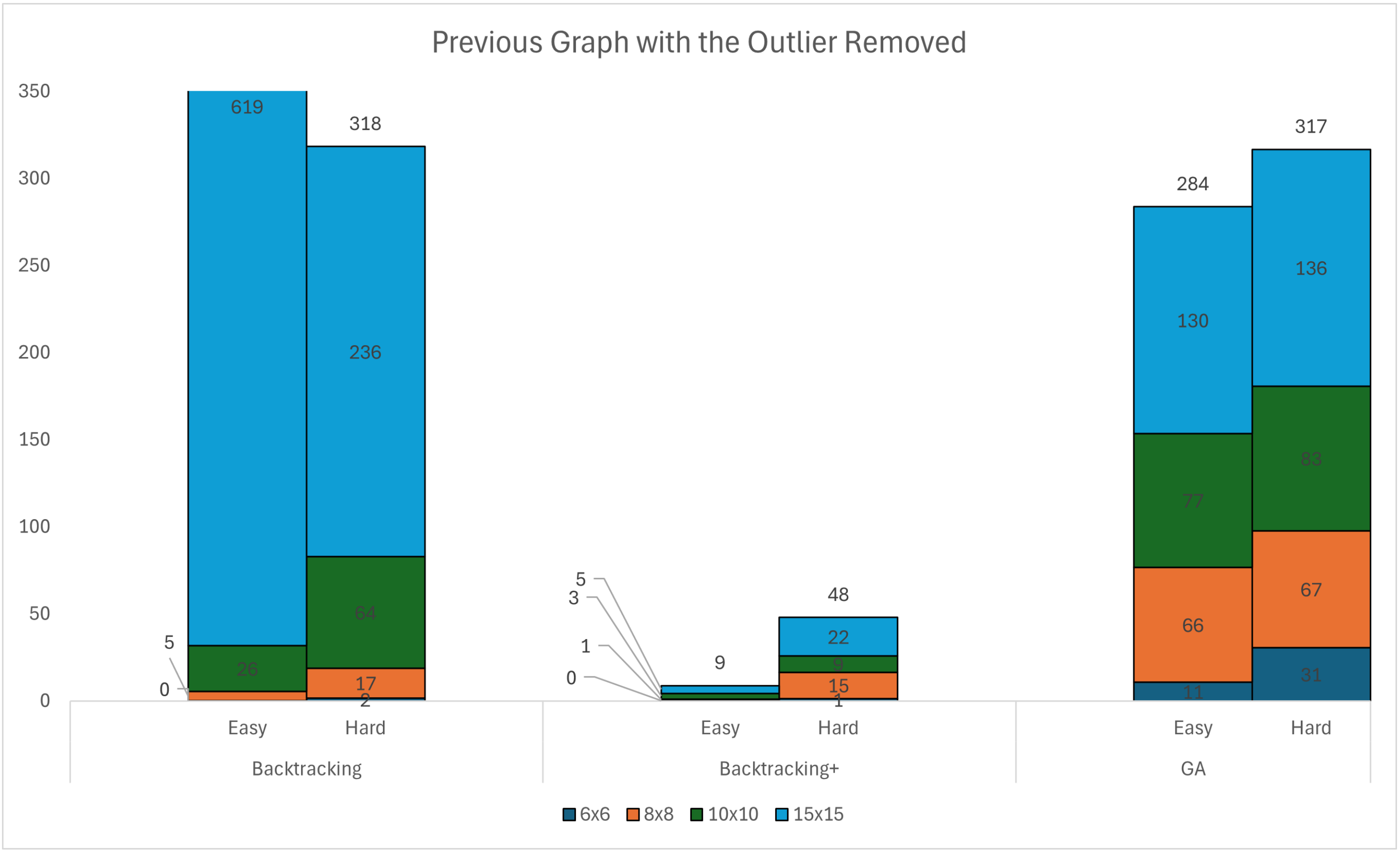


Figure 6: Overall Comparison By Time (without Outlier)

6 Appendix

6.1 How to run the program

To try the program, run main.py. Do not run genetic_algorithm.py as that is a module used by main.py. Click "Select Level" and select a valid ".bs" file (short for battleship). These are just JSON text files with board size, hint locations and row/column numbers.

Once loaded in you can solve the puzzle manually or click one of the green buttons to select a solver method. The red outlined button on the bottom left is the active selection, you can click on the active button to enter erase mode which will not allow you to erase hint pieces. **The checkboxes above these buttons has no clicking functionality**, it is only meant to automatically check off ships once completed on the board. Numbers will outline in red if the row/column has too many ships but NOT if there isn't enough ships. This was done on purpose to not make manually solving it too annoying.

During automatic solving, **user input will be disabled** but there is a bug where if you accidentally click reset during execution the board will instantly reset as soon as solving is finished. During genetic algorithms, the genetic_algorithms.py script will execute 5 times and lock input the whole time. The only way to know if the next attempt has started inside the GUI is that the checkboxes will briefly disappear. Apologies for my game being a little rough around the edges but I prioritized the implementation of algorithms over usability of the tool.

During execution of GAs, you can also look at the console to see the generation count and average fitness of the current generation.