COMP20003, Algorithms and Data Structures, Semester 2, 2024
Assignment 3 – Report

Kerui Huang – 1463475 - keruih@student.unimelb.edu.au

Executive summary

This report focuses on the evaluation of a puzzle-solving algorithm designed to solve grid-based "Chessformer" puzzles by generating and expanding nodes. The primary goal is to investigate the performance of the solver by examining both unoptimized and optimized versions. To achieve this, the report analyses execution time and the number of generated nodes, comparing cases with and without optimization across different puzzle configurations.

Introduction

Algorithm used

The unoptimized version of this implementation of a Chessformer solver is based on Dijkstra's search algorithm. The optimized version builds upon this by incorporating duplication detection with hash tables, which should enhance efficiency by preventing redundant state explorations.

The objective of the solver involves navigating a player to traverse from an initial state to a final state where all target pieces on the board are captured, in the shortest path. In the context of Chessformer, the grid represents the graph, with each state (configuration of the board) serving as a vertex, and each valid move representing an edge between vertices.

Algorithm 1 AI Chessformer Algorithm

```
1:  procedure FindSolution(start, showSolution)
2:      n ← CreateInitNode(start)
3:      Enqueue(n)
4:      exploredTable ← create empty array
5:      while queue ≠ empty do
6:          n ← queue.Dequeue
7:          exploredNodes ← exploredNodes + 1
8:          exploredTable ← record n in array
9:          if WinningCondition(n) then                    ▷ Found a solution
10:             solution ← SaveSolution(n)
11:             solutionSize ← n.depth
12:             break
13:         end if
14:         for each move action a ∈ {KnightMoves} ∪ {QueenMoves} do
15:             playerMoved ← ApplyAction(n, newNode, a)   ▷ Create Child newNode
16:             generatedNodes ← generatedNodes + 1
17:             if playerMoved is false then
18:                 Free(newNode)
19:                 continue
20:             end if
21:             queue.Enqueue(newNode)
22:         end for
23:     end while
24: end procedure
```

Algorithm 2 AI Chessformer Algorithm - optimised

```
1:  procedure FindSolution(start, showSolution)
2:      n ← CreateInitNode(start)
3:      enqueue(n)
4:      exploredTable ← create empty array
5:      hashTable ← initialise Hash Table for duplicate detection
6:      while queue ≠ empty do
7:          n ← queue.dequeue
8:          exploredNodes ← exploredNodes + 1
9:          exploredTable ← record n in array
10:         if WinningCondition(n) then                    ▷ Found a solution
11:             solution ← SaveSolution(n)
12:             solutionSize ← n.depth
13:             break
14:         end if
15:         for each move action a ∈ {KnightMoves} ∪ {QueenMoves} do
16:             playerMoved ← ApplyAction(n, newNode, a)   ▷ Create Child newNode
17:             generatedNodes ← generatedNodes + 1
18:             if playerMoved is false then
19:                 Free(newNode)
20:                 continue
21:             end if
22:             flatMap ← FlattenMap(newNode)              ▷ Converts the map into a 1D map
23:             if flatMap is a duplicate then             ▷ Checks for duplicates in Hash Table
24:                 duplicatedNodes ← duplicatedNodes + 1
25:                 Free(newNode)
26:                 continue
27:             end if
28:             hashTable ← InsertHashTable(flatMap)
29:             queue.Enqueue(newNode)
30:         end for
31:     end while
32: end procedure
```

Figure 1: Pseudocode of the search algorithms

Hypothesis

While the optimized version of the solver requires additional computation for managing the hash table, it is expected to outperform the unoptimized solver, particularly in larger or more complex puzzles where more states are generated. This is because the likelihood of encountering duplicate states increases, making the optimization more beneficial.

In theory, the time complexity of Dijkstra's algorithm for finding the shortest path is O((V + E) log V), where **V** represents the number of vertices (possible states of the puzzle), and **E** represents the number of edges (possible moves between states) in the graph. In here, the number of possible states V depends on the number of possible cells the player and pieces can occupy, while E can generally be affected by the total number of the cells.

The elimination of duplicate states would realistically reduce the number of operations overall, however, theoretically speaking, the worst-case scenario for both versions is the same (where every possible configuration of the game is a vertex, and every legal move is an edge). Hence, despite the optimisation, the time complexity should remain the same.

Methodology

Metrics Measured

In this report, two key metrics are measured to evaluate the solver's performance:

- **Number of Generated States:** This metric represents the total number of game states/nodes explored by the solver during its search for a solution. It helps assess the efficiency of the algorithm, with fewer generated states indicating a more optimized search process.

- **Solution Time:** This metric tracks the total time taken by the solver to find a solution, measured from the start of the search to when the shortest path solution is identified. It reflects the overall performance and speed of the solver, with faster solution times being more desirable.

Test Data

The test data used in this report are the "capability" puzzles provided by the COMP20003 Team. There are 12 puzzles in total with varying complexity and size. For the analysis, the independent variables used will be the number of cells in the grid, the number of ground cells and the number of initial cells with capturable pieces, all of which can affect the number of total states and legal moves of the puzzle.

Results and Discussion

This section presents the results of the performance evaluation of the Chessformer solvers, focusing on the comparison between the unoptimized and optimized versions of the algorithm. (*Note: the recorded metrics are solely from own implementation of the algorithms).

Capability No.	number of cells in the grid	number of ground cells	number of capturable pieces	Unoptimised		Optimised	
				number of generated states	solution time (s)	number of generated states	solution time (s)
1	2	2	1	2	0.000007	2	0.000009
2	3	2	1	3	0.000007	2	0.000009
3	2	2	1	2	0.000007	2	0.000009
4	4	2	1	2	0.000012	2	0.000012
5	34	11	4	17524	0.633005	112	0.002747
6	34	11	3	2593	0.010739	54	0.000539
7	8	3	1	15	0.000049	4	0.000024
8	8	3	1	13	0.000039	4	0.000017
9	8	6	1	28	0.00019	7	0.000128
10	9	9	2	87	0.000091	22	0.000069
11	19	10	1	131	0.000111	11	0.000193
12	34	17	1	6367	0.122602	21	0.000788

Figure 2: Table reporting execution time and number of generated nodes for all puzzles

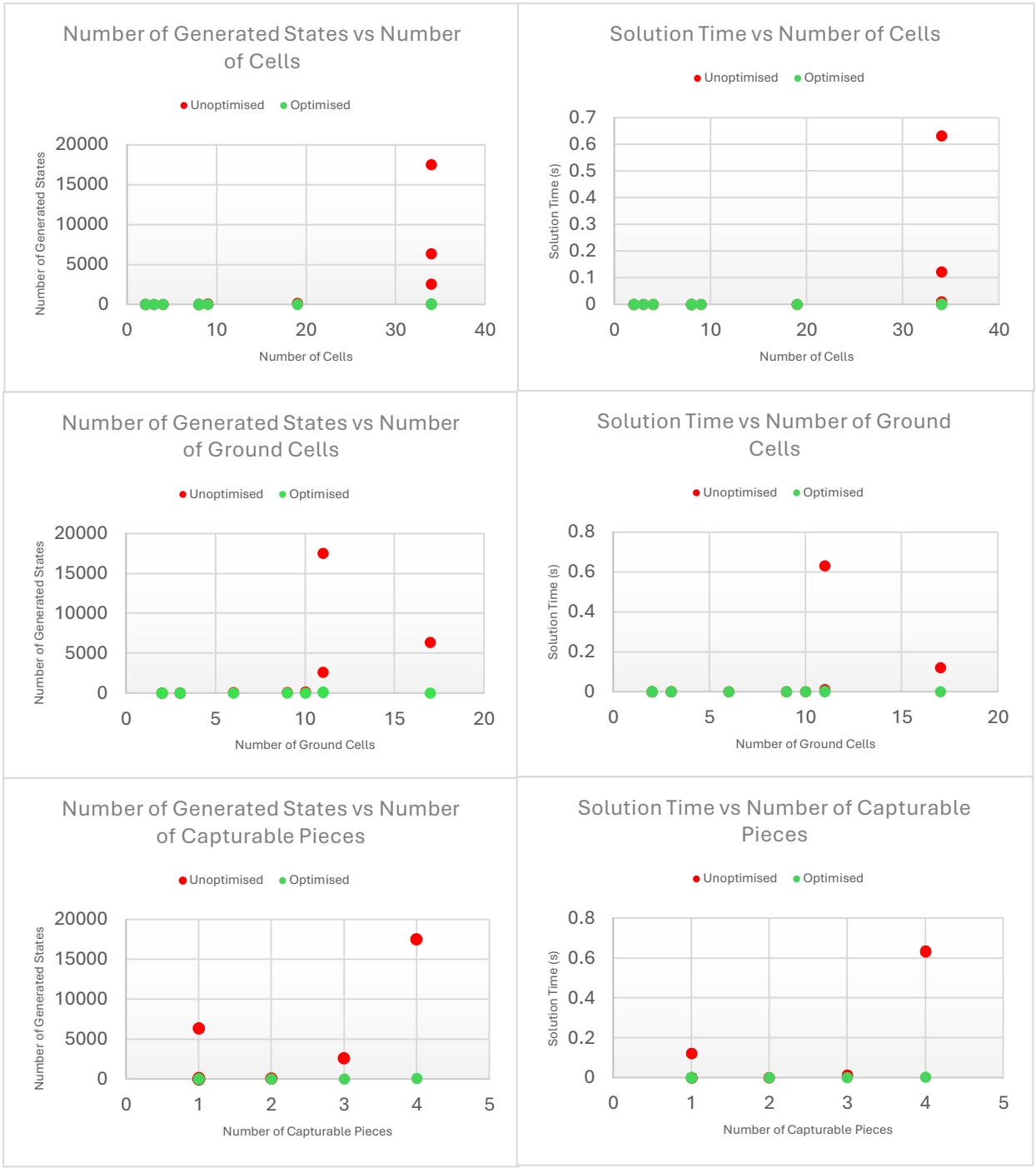

Figure 3: Plots for execution time and number of generated nodes for all puzzles

*Due to overlaps, some unoptimised points are not visible

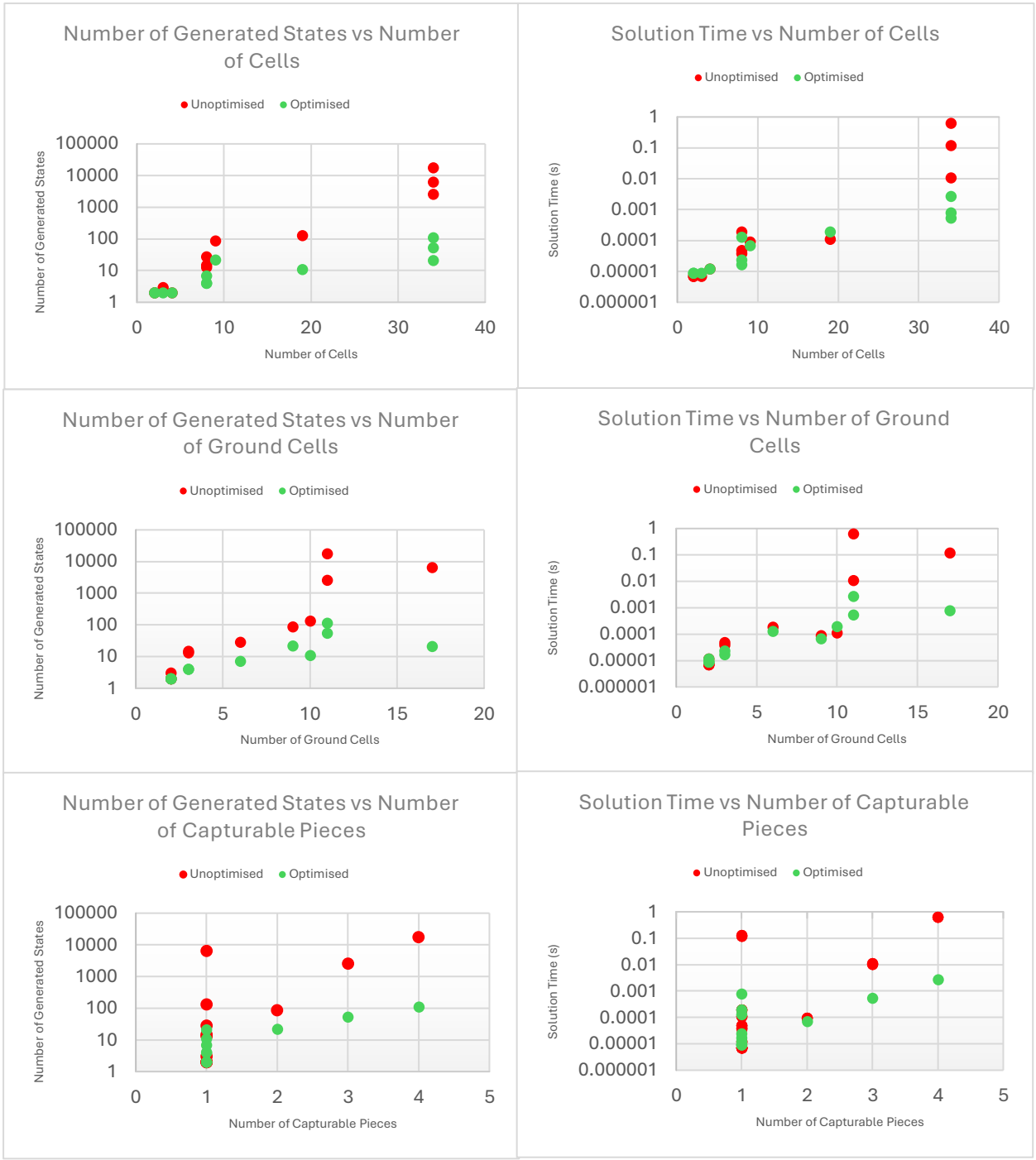

Figure 4: Plots for execution time and number of generated nodes using log10 scale

As shown in Figure 3 and 4, while the unoptimised solver produces comparable results for lower numbers of ground cells and total cells, in both solution time and number of generated states, the unoptimised solver's generated states and solution time dramatically increase as the number of ground cells and total cells increases. The difference between the two solvers is also shown to become larger with increased cells through the log scale. This can be explained by the avoidance of duplicated states by the optimised solver, the appearance of which tends to be more probable with more (ground) cells in total, since having more ground cells corresponds to more possible total states (V) and having more total cells tends to correspond to more possible legal moves.

A drop in solution time and generated states can be observed for both solvers with 17 ground cells, which contrasts the general increasing trend. Although this particular puzzle has the most ground cells, it also only has 1 piece to be captured, which can potentially be the explanation. Furthermore, a similar instance deviates from the general increasing pattern in plots for the number of capturable pieces, having solution times and generated states above expected values with 1 capturable piece. This can be explained by the same reasoning used above. Despite having only 1 capturable piece, this puzzle has the greatest number of ground cells and total cells across the test data.

Using the log scale, it is shown that the solution time and state generation for both solvers exhibit a superlinear growth with the problem size. In general, it appears that all three independent variables can influence the solution time and the states generated in the same way, where increase in one variable tends to induce increase in the other. The duplication detection used in the optimised version effectively mitigates that influence, reducing the solution time, and the states generated significantly.

An important consideration is that, due to the relatively small test dataset, the confidence in the results remains somewhat limited. A larger dataset would be needed for a more definitive conclusion and evidence about the effects of the optimisation and their growth rates.

Conclusion

In conclusion, the results demonstrate that the optimized solver, with duplication detection, significantly reduces both the solution time and the number of generated states compared to the unoptimized version, particularly in puzzles with larger grids and more complex configurations (more enemy pieces or ground cells). Given the result data, the hypothesis was largely supported. However, the specific complexity discussed in the hypothesis could not be conclusively demonstrated. While the results indicate clear benefits of optimization, the limited size of the test dataset means that definitive conclusions cannot be drawn without more systematic testing on a larger set of puzzles.

References

geeksforgeeks (2024). *Time and Space Complexity of Dijkstra's Algorithm*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/time-and-space-complexity-of-dijkstras-algorithm/.

itch.io. (2021). *Chessformer*. [online] Available at: https://rob1221.itch.io/chessformer [Accessed 12 Oct. 2024].

omar khaled abdelaziz abdelnabi (2016). *Shortest Path Algorithms Tutorials & Notes | Algorithms | HackerEarth*. [online] HackerEarth. Available at: https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/.

Reddit.com. (2024). *Time complexity of Dijkstra algorithm*. [online] Available at: https://www.reddit.com/r/algorithms/comments/o4rye3/time_complexity_of_dijkstra_algorithm/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1&utm_content=share_button [Accessed 12 Oct. 2024].

W3schools (n.d.). *DSA Dijkstra's Algorithm*. [online] www.w3schools.com. Available at: https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php.