COMP20003: Assignment 3 Report

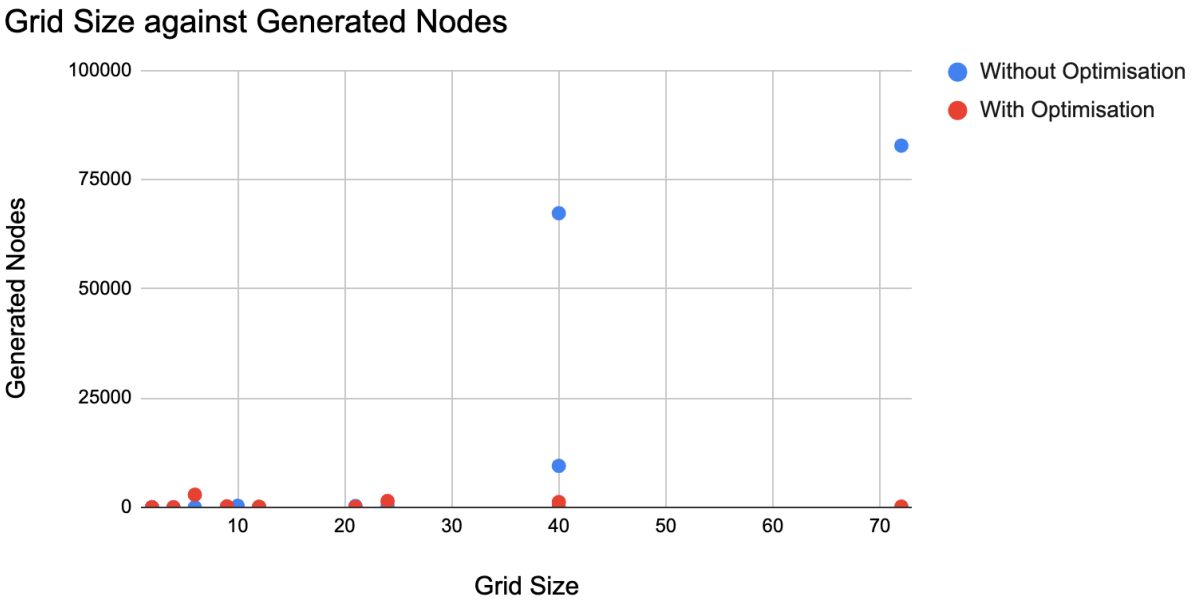
1473948 – Chao Rebecca Wei

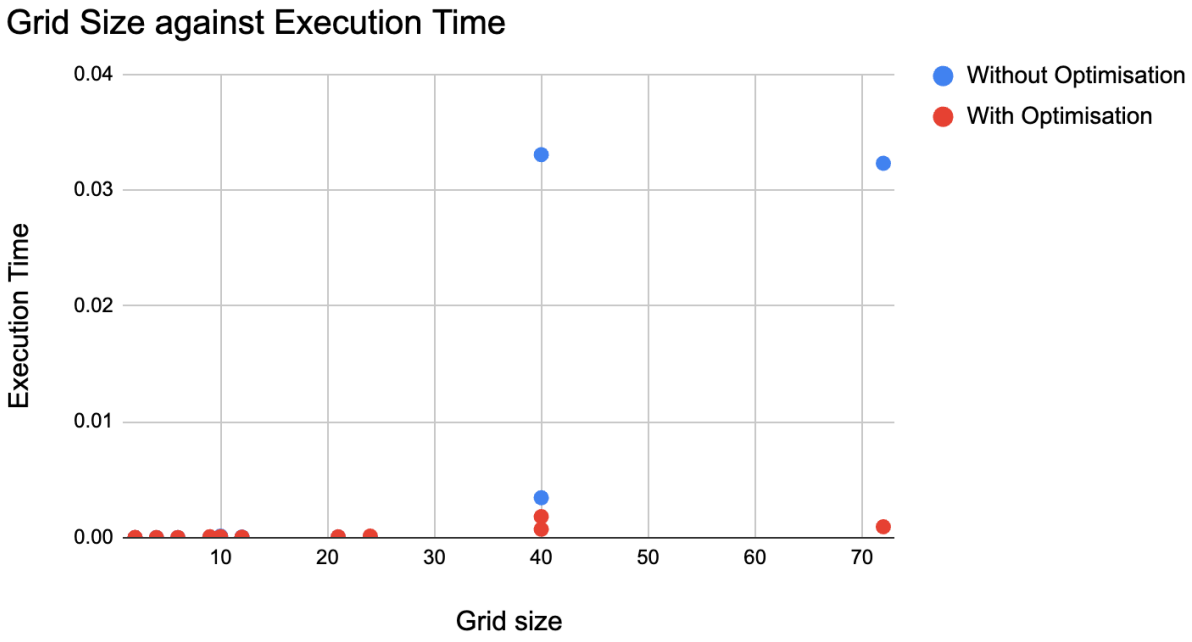
Experimentation section of Assignment 3

Table

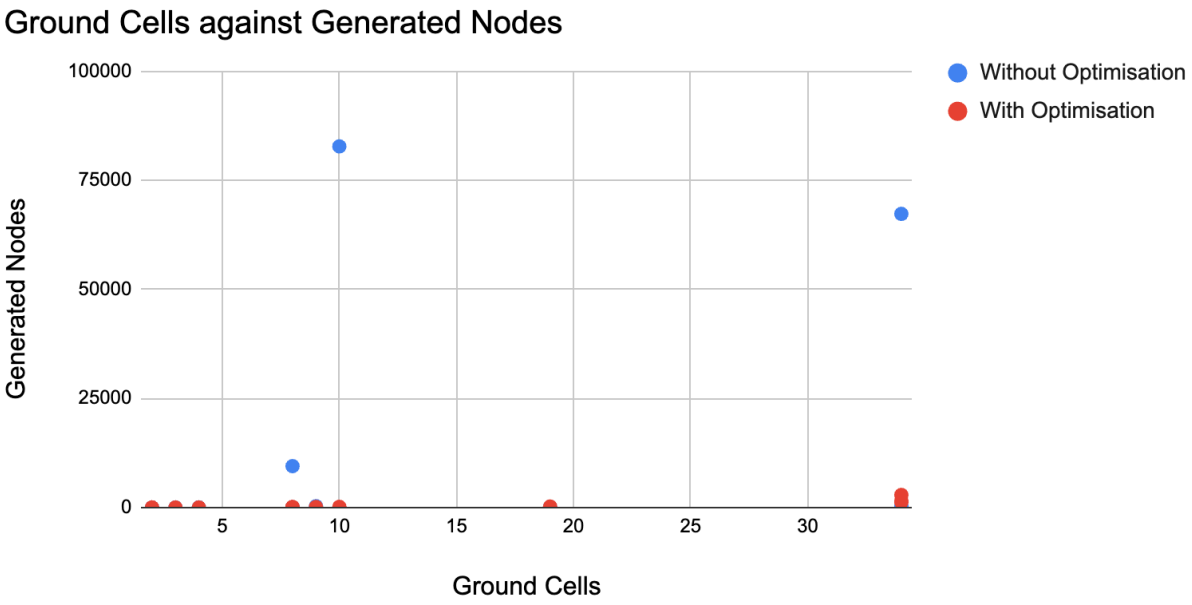
Puzzle Name	Without Optimisations		With Optimisations	
	Generated Nodes	Execution time	Generated Nodes	Execution time
Capability 1	2	0.000005	2	0.000012
Capability 2	4	0.000006	4	0.000014
Capability 3	2	0.000005	2	0.000012
Capability 4	6	0.000005	6	0.000013
Capability 5	67320	0.033068	2880	0.001817
Capability 6	9480	0.003446	1200	0.000726
Capability 7	48	0.000044	36	0.000029
Capability 8	36	0.000023	24	0.000027
Capability 9	294	0.000056	126	0.000076
Capability 10	117	0.000045	117	0.000081
Capability 11	288	0.000049	216	0.000134
Capability 12	82800	0.032320	1440	0.000939
Test Map 1	340	0.000133	150	0.000086

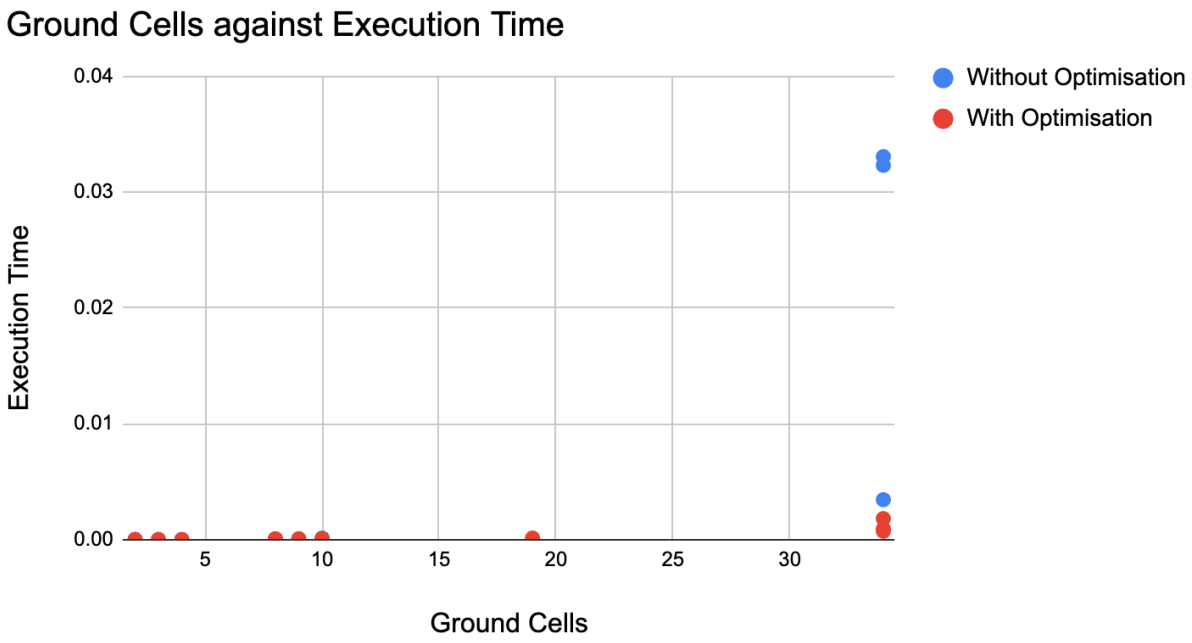
Figures



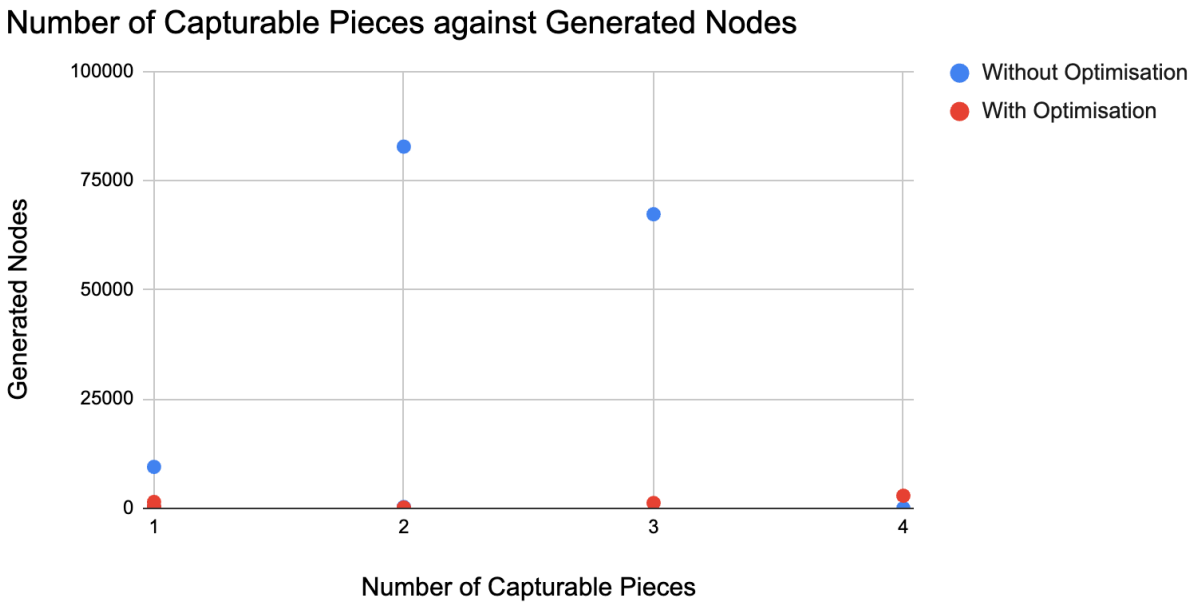


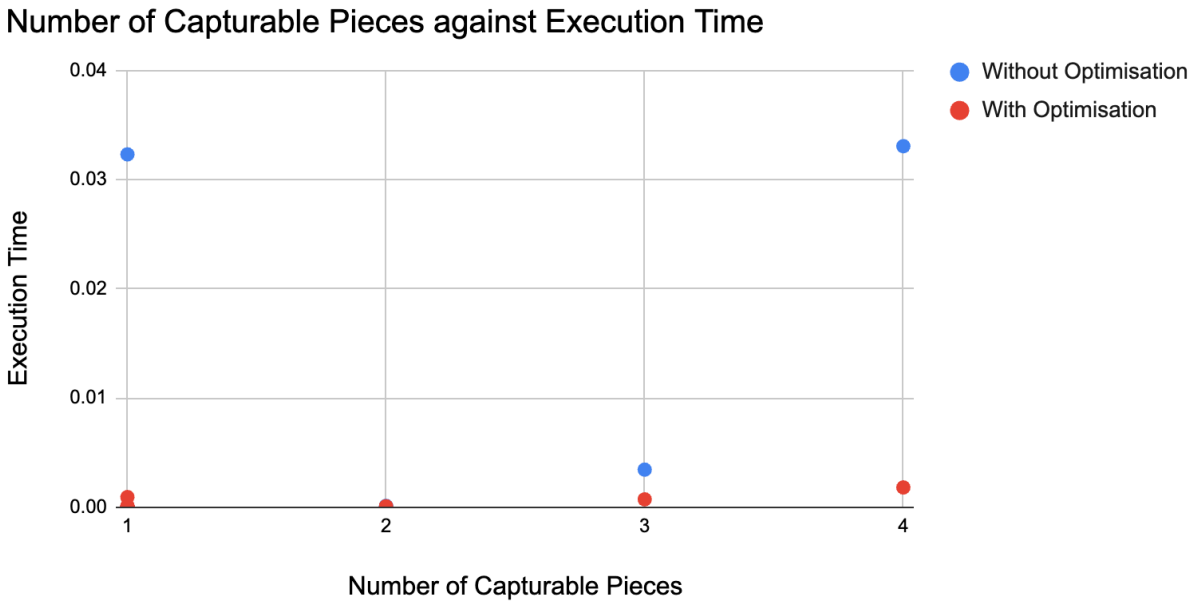
Grid Size is defined as the total number of cells within the grid, excluding the border.





Ground Cells are defined as the Grid Size, minus any spaces that the playable piece cannot move into as it is a wall piece.





Number of Capturable Pieces simply refers to the total number of “$” pieces within the map.

Analysis

Which complexity growth does your data appear to show?

The factors which are examined for their effect on the time complexity of the algorithm are:
n, the number of grid cells
g, the number of ground cells
p, the number of capturable pieces in the puzzle

For the unoptimised algorithm, the time complexity appears to be **$O(2^g * p)$**.

In theory, this would be because the unoptimised solution explores all possible states before all capturable pieces are captured, leading to a complexity proportional to all of the possible configurations of ground cells with respect to the capturable pieces. The term **2^g** is used to represent the binary decision being made (move or don't move) for g possible positions, while the term **p** is used to represent the additional paths needed to be explored to capture these pieces.

This is loosely reflected within the data provided above. Puzzles with very few ground cells and capturable pieces (such as Capabilities 1 - 3) have very few generated nodes and short execution times, but when the puzzles increase in size and capturable pieces (such as Capability 5,6 and 12), the number of generated nodes increase drastically. This increase aligns with the exponential growth described within the proposed **$O(2^g * p)$** complexity.

For the optimised algorithm, the time complexity appears to be **$O(g^2 * p)$**.

In theory, this would be because the optimised solution prevents redundant exploration of the same ground cells, leading to a complexity proportional to the possible *unique* configurations of ground cells with respect to the capturable pieces. The term **g^2** is used to reflect the total number of unique configurations of the player's position and a capturable piece's position, while the term **p** is used to represent the additional paths needed to be explored to capture these pieces.

This is feasible when compared to the results where for Capability 5, generated nodes decrease from 67,320 to 2,880, a reduction by a factor of ~23, while for Capability 12, the generated nodes decrease by a factor of ~57. This suggests that there is quadratic rather than exponential growth in the number of generated nodes.

What's the computation benefit of the optimisation, does it decrease the growth rate?

Yes, it does. When applied to more complex puzzles, optimisation (through implementing a duplicate state detection mechanism with a hash table) significantly mitigates the growth rate for generated nodes and execution time.

Empirically, this is evident by examining the more complex puzzles, Capability 12 and Capability 5, which have the biggest grid sizes (72 and 40 respectively). The difference between optimised and non-optimised performance differs by orders of magnitude for both execution time and generated nodes:
For Capability 12, the number of generated nodes decreases from 82,800 to 1,440, and the execution time decreases from 0.032320 seconds to 0.000939 seconds.
For Capability 5, the number of generated nodes decreases from 67,320 to 2,880, and the execution time decreases from 0.033068 seconds to 0.001817 seconds.

This is consistent with theory as the non-optimised solver would explore many redundant states due to revisiting the same configurations again. With the optimisation, the solver avoids explored previously visited states, decreasing the search space and leading to fewer generated nodes and faster solution times. As mentioned in the earlier question, without optimisation, the time complexity grows exponentially while with the optimisation implementation, the time complexity grows at a quadratic rate.

Were any default arguments within the code changed? If so, discuss their impact.
1 change was made to default arguments within the code:
The function "winning_state" was altered to also take state_s as a parameter. This allowed for the current state to be checked to see if the puzzle had been completed.