ASSIGNMENT 3 ALGORITHMS AND DATA STRUCTURES: EXPERIMENTATION

Claire Tosolini: CTOSOLINI 1271302

Introduction

In this assignment, an incomplete 'Flow Free' solver (adapted from mzucker: https://mzucker.github.io/2016/08/28/flow-solver.html) was provided. The 'Flow Free' puzzle presents as an NP-complete problem, and hence the best existing algorithms run with exponential time complexity. I implemented the AI Solver using Dijkstra's Algorithm, and then optimised to some degree through the addition of Dead-End Detection.

Results and Analysis

Table 1: comparison of 'Flow Free' AI solver with and without optimisation

Puzzle	Number of Free Spaces in Initial Grid	Solution Time Without Dead-End Detection	Number of Generated States Without Dead-End Detection	Solution Time With Dead-End Detection	Number of Generated States With Dead-End Detection
Regular 5x5	15	0.000	18	0.000	17
Regular 6x6	24	0.000	283	0.000	156
Regular 7x7	37	0.002	3317	0.002	644
Regular 8x8	52	0.391	409726	0.070	15407
Regular 9x9	63	0.412	587332	0.080	29546

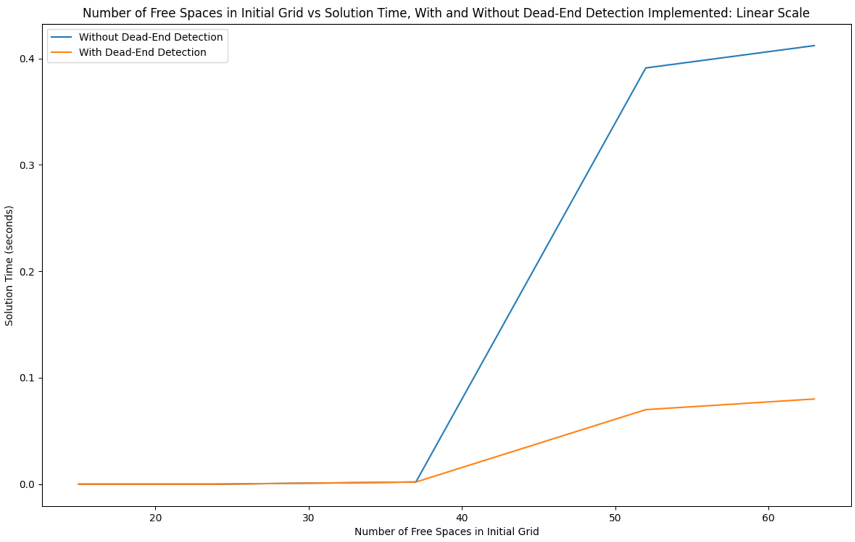

Figure 1: comparison - solution time

Figure 1 Analysis

- Results <u>with</u> Dead-End Detection implementation demonstrated a slower rate of growth of Solution Time compared to <u>without</u> (see Figure 1 opposite).
- Both implementations show <u>logarithmic complexities</u> (Figures 1.1, 1.2), however the logarithmic equation corresponding to Dead-End Detection has a slower rate of growth.

The Dead-End Detection optimisation decreased the growth rate with respect to Solution Time.

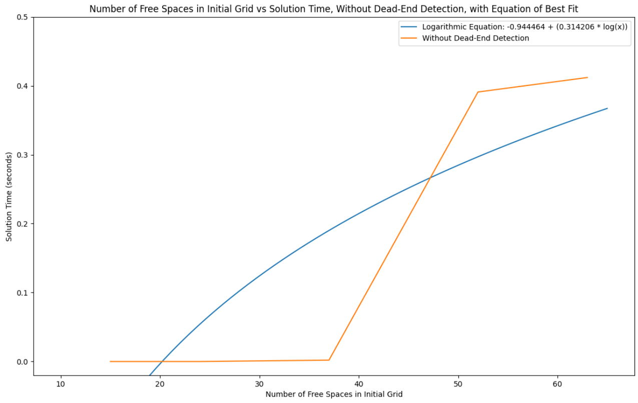

Figure 1.1: logarithmic growth without optimisation

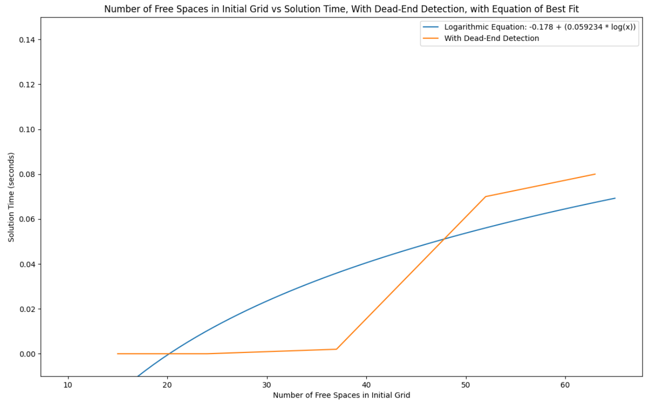

Figure 1.2: logarithmic growth with optimisation

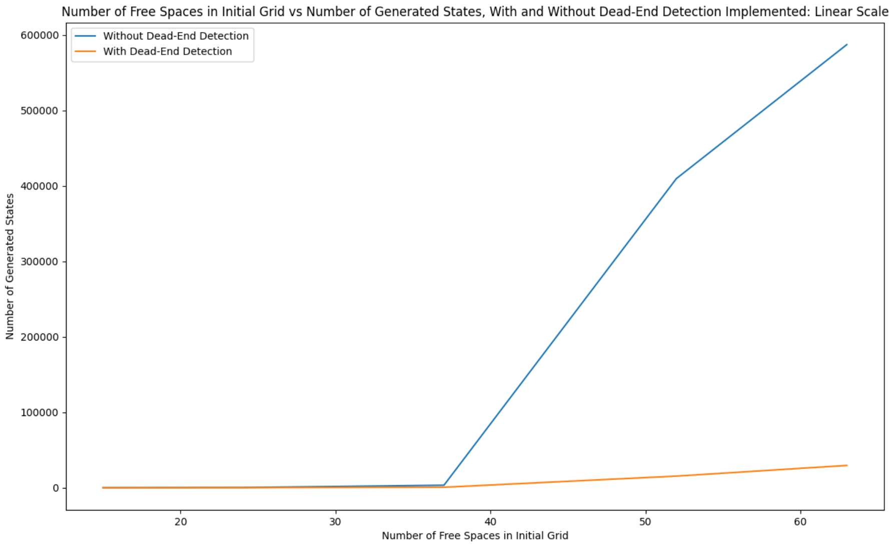

Figure 2: comparison - generated states

Figure 2 Analysis

- Results <u>with</u> Dead-End Detection implementation demonstrated a significantly slower rate of growth of Number of Generated States compared to <u>without</u> (see Figure 2 opposite).
- Both implementations show <u>exponential complexities</u> (Figures 2.1, 2.2), however the exponential equation corresponding to Dead-End Detection has a slower rate of growth.

The Dead-End Detection optimisation also decreased the growth rate with respect to Number of Generated States.

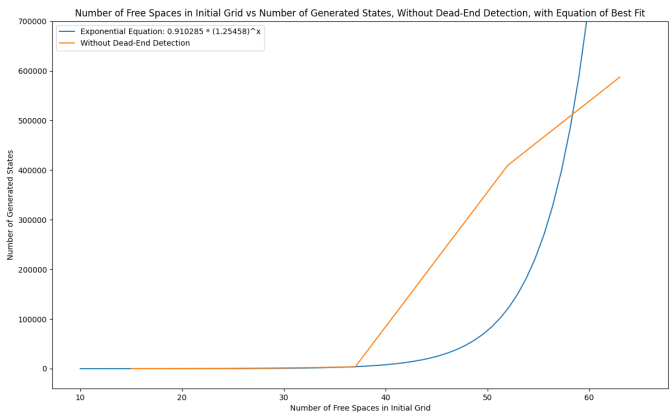

Figure 2.1: exponential growth without optimisation

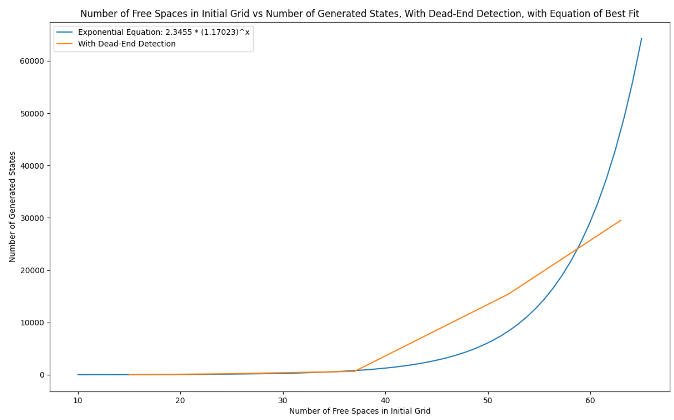

Figure 2.2: exponential growth with optimisation

Discussion of Optimisations

The Dead-End Detection optimisation was achieved through the 12-cell cross technique.

An <u>empty cell</u> is defined as a cell within the game board that contains no initial or goal position, nor a current path in progress. A <u>free cell</u> is defined as either an empty or goal position. A <u>dead-end</u> is defined as an empty cell with less than two free neighbouring cells.

For each game move, the twelve surrounding cells were checked for dead-ends. The existence of a dead-end resulted in removal of the corresponding game state.

Since this optimisation technique removed unsolvable game states and therefore any further states evolving from them, less states overall were generated. This corresponds directly to the decrease in Number of Generated Nodes. Additionally, since the time taken by the program to reach the solution state is directly related to the number of intermediate game states, the reduction in the number of generated nodes also reduced the Solution Time.

Conclusion

The AI Solver implementation, using Dijkstra, was functional but not optimal. Implementing the Dead-End Detection optimisation introduced a <u>significant computational benefit</u>, with a resultant decreased growth rate with respect to both Solution Time and Number of Generated States.

Further optimisation is possible through improvement of the existing dead-end detection, the use of alternative graph algorithms in the AI solver (e.g. A*), and further analysis of individual game boards.