Informed Search Numberlink Puzzle Solver

**Authors:** Cole Pierce and Kenny Song

**Roles:**

Coordinator: Kenny

Recorder: Kenny

Checker: Cole

**Abstract:**

The numberlink puzzle involves an m x m grid with n pairs of elements placed in the grid. The goal is to connect the elements with a line that does not overlap another line. A popular version of this game was the mobile game Flow. There are many approaches to solving a number link puzzle. We analyzed the informed search strategies of Greedy Best First search and A\* Search. Because of the memory issues associated with A\* as the board size , we also implemented an improvement on A\*: Iterative deepening. The g(s) is the total length of each line and the heuristic we chose for our h(s) was the total manhattan distance from the current end of each line to the goal of where the line is trying to go. We compared the search strategies by generating random puzzles and aggregating the statistics on different size grids with a timeout to signify the puzzle was unsolvable or not solved fast enough. We found that A\* performed the best on the smaller solvable puzzles but the larger puzzles caused memory issues as anticipated. Iterative deepening A\* was an improvement for puzzles with a simple solution, but faced the same problem with puzzles that required filling up the entire board to solve. Greedy best first search remained consistent in being able to solve easier puzzles but struggled to find the solution for the more complicated solvable puzzles before timing out.

**Hypothesis:**

Greedy best first search and A\* search will be an improvement to uninformed search strategies based on the number of nodes expanded, the depth of the solution, and the time it takes to solve the puzzle. A\* will run out of memory on the larger puzzles, but iterative deepening A\* will help with that problem.

**Motivation:**

Solving puzzles is something almost every programmer enjoys doing. We were interested in the first three project choices but chose this one because we were familiar with the informed search strategies and wanted to learn how to implement them in F# sharp with a custom s (state). We also felt this would build off well from the previous programming assignments, could be done mostly with our knowledge of F#, and would not require additional libraries than the basic ones we already knew.

**Methods:**

The three search strategies used were: Greedy best first, A\* , and iterative deepening A\*. Each state/node was represented by a grid of characters visualizing the positions each different line currently occupied in the puzzle, and a map of lines where each line knew it’s current end of the line, where it was trying to go, and the length of the line. Greedy best first search simply chose to expand the node with the lowest heuristic which in our case was the total manhattan distance from the end of each line to the goal of each line. A\* chose to expand the node with the lowest total length of each line (g(s)) + the total manhattan distance of each line (h(s)). Because a line can only expand up, down, left, or right, the Manhattan distance is an admissible heuristic because it will never overestimate the actual cost. Iterative deepening improved on A\* by limiting the depth of the search to prevent running out of memory. We improved on iterative deepening A\* by instead of starting at 1 as the depth we began at the lowest possible depth the solution could be at (manhattan distance). All three of these methods utilized a priority queue to choose which node to expand next based on the search strategy. The prioritization of the queue was done by implementing comparable to correctly order each state according to the search strategy. As our understanding of the puzzle increased we realized that a lot of the randomly generated puzzles tend to have several paths of reaching the solution. This was different from typical solvable puzzles that generally have one solution that utilizes the entire grid to solve. Our implementation of the board state already prevented returning to previous states but it did not prevent converging at the same state from different paths. We utilized a set of the board states already gone to which accounted for repeat states with different prior paths. This improvement prevented unnecessarily repeatedly duplicating search space and improved the solve time.

We also wrote a Python script to generate the random puzzles according to the specification provided. All of the validation was done internally and did not

**Results:**

Below are the generated results. I had to go back and find the unsolvable puzzle for both greedy and A\* by finding the puzzle that returned ‘-69’ before the 100 second timeout. 11x11 puzzles were the only ones that we ran into this issue. The evidence can be seen through the text file ElevenByElevenResults.txt. It is on line 5. If you would like to generate this please change the filePath on line 79 to “AI-NumberLink-Project/testCases.txt”.

Part 1-- Solvable Greedy: 11 Unsolvable: 1 Timed Out: 238

Part 1-- Solvable A\*: 9 Unsolvable: 1 Timed Out: 240

Part 2--Greedy Best Result 8x8 Average Expanded Nodes: 185  Average Cost: 31 Average Branching: 22 Average running time: 36.4286

Part 2--Greedy Best Result 9x9 Average Expanded Nodes: 40  Average Cost: 26 Average Branching: 23 Average running time: 11.2331

Part 2--Greedy Best Result 10x10 Average Expanded Nodes: 479  Average Cost: 37 Average Branching: 32 Average running time: 28.46235

Part 2--Greedy Best Result 11x11 Average Expanded Nodes: -69  Average Cost: 27 Average Branching: 36 Average running time: 0.0705

I had the code return -69 when no solution was found before the timeout of 100 seconds occured. Therefore, there was **no solutions** found for 11 by 11 puzzles.

Part 2--Greedy Best Result 12x12 Average Expanded Nodes: 0  Average Cost: 0 Average Branching: 0 Average running time: 0

This means that no solutions were found for any of the 12x12s because of the timeout.

Part 2--A\* 8x8 Average Expanded Nodes: 417  Average Cost: 29 Average Branching: 20 Average running time: 18.4035

Part 2--A\* 9x9 Average Expanded Nodes: 131  Average Cost: 26 Average Branching: 23 Average running time: 18.21554

Part 2--A\* 10x10 Average Expanded Nodes: 8710  Average Cost: 46 Average Branching: 32 Average running time: 541.29615

Part 2--A\* 11x11 Average Expanded Nodes: -69  Average Cost: 27 Average Branching: 36 Average running time: 0.0662

I had the code return -69 when no solution was found before the timeout of 100 seconds occured. Therefore, there was **no solutions** found for 11 by 11 puzzles.

Part 2--A\* 12x12 Average Expanded Nodes: 0  Average Cost: 0 Average Branching: 0 Average running time: 0

This means that no solutions were found for any of the 12x12s because of the timeout.

The maximum depth for iterative deepening A\* was around 30. This number varied based on the initial size of the puzzle and the amount of different starting pairs. The less pairs, the fewer states were required to keep in memory and fewer computation time was required.

**Discussion:**

Before even running all of the tests a big improvement we made from analyzing some of the smaller puzzles was the duplication issue discussed in the methods section. This discovery vastly improved the amount of solvable puzzles that were solved during testing. From the results you can see that generating random puzzles usually results in unsolvable puzzles. Because of this the amount of results we had were quite limiting. For initial validation we mostly tested with a few solvable puzzles. However, these puzzles were inputted manually which did not scale to the amount of puzzles needed for testing and gathering results. Our results show that although a vast improvement to uninformed search methods these informed search strategies can be further improved on for solving numberlink puzzles. The manhattan distance heuristic works very well for simpler puzzles, but especially as the size and complexity of the solution increases, the manhattan distance heuristic proves less useful. Our results for the iterative deepening A\* show that although an improvement to some puzzles it was not as much of an improvement as we had anticipated. Because a lot of puzzles are solved taking up close to the entire grid, iteratively increasing the depth 1 at a time did not end up limiting the amount of memory required. To further improve A\* we may instead start at the average solve depth for that size of puzzle instead at the minimum solvable depth for the puzzle. If we were to continue this project we would probably use a combination of the manhattan distance, the number of directions a line can expand in, and an inner to outer approach to better decide which nodes to expand first. Having a combination of heuristics that updated throughout the problem would improve the reliance on just the manhattan distance which often times traps other lines or starts off going blindly into a dead end.

**Acknowledgements:**

Priority Queue implementation from tjaskula: <https://gist.github.com/tjaskula/18a51f84ef06b4819b0952bb7e691828>

F# documentation:

<https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/>