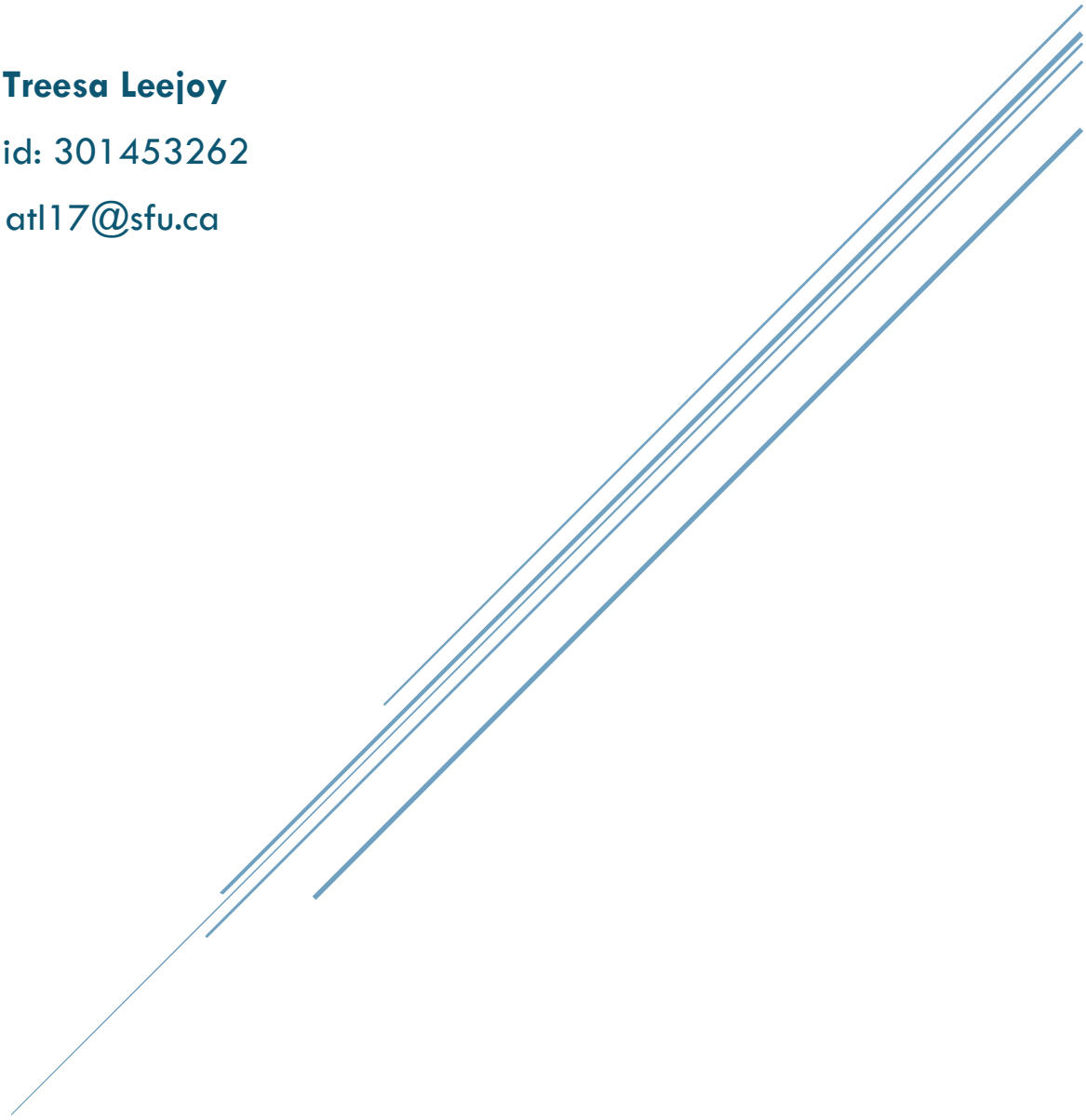
NxN-puzzle

Finding a solver for sliding puzzle

Aleena Treesa Leejoy

Student id: 301453262

Mail id: atl17@sfu.ca



Simon Fraser University

CMPT 225: Data Structures and Programming

Abstract

This project aims to develop a solver for the NxN puzzle using a variety of algorithms, heuristics, and techniques. The solver will be programmed in Java and optimized for solving the puzzle in the least possible moves. The project involves testing the solver on various NxN puzzle configurations, ranging from 3x3 to 9x9. During the process I have tried many algorithms and heuristics to develop this solver. The project will contribute to the development of more efficient and effective solvers for the NxN puzzle, which has implications for various industries, including gaming, logistics, and transportation.

The Goal

The objective of this project was to develop a program capable of solving an NxN sliding puzzle. Various techniques were considered, including using a search algorithm to solve the entire board, breaking the board into smaller sections by rows and columns to make it more manageable, or moving the tiles individually into their correct positions through iterative means.

The Search Algorithm I Chose

I opted to employ an A* search algorithm to tackle the NxN puzzle boards because of its capability to determine the shortest possible route to solving the puzzle while utilizing time and memory more efficiently than DFS and BFS. DFS explores all child nodes before backtracking, whereas BFS explores every node to a certain depth, rendering them less effective. Furthermore, A* can solve larger puzzles at a faster rate with the aid of a suitable heuristic. A* is more straightforward to implement, enabling us to concentrate on refining heuristics and optimization techniques.

Using of BFS

For solving the puzzle I used the BFS algorithm at first. The program accepts two arguments, an input file, and an output file. The input file contains the initial state of the puzzle, and the output file contains the solution to the puzzle.

What is BFS

BFS stands for Breadth-First Search, which is a graph traversal algorithm used in computer science to explore all the vertices in a graph or search tree in breadth-first order. The algorithm starts at the root node and explores all the nodes at the current depth level before proceeding to the next level. It proceeds in a systematic manner, visiting all nodes at each depth level before moving to the next level.

To implement the BFS algorithm, a queue is used to keep track of the visited nodes. The algorithm starts by placing the root node in the queue, and then repeatedly dequeuing the first node in the queue and enqueuing all of its unvisited neighbors. This process continues until the queue is empty, at which point all the nodes in the graph or search tree have been visited.

BFS is often used to solve problems such as finding the shortest path between two nodes in an unweighted graph or determining if a graph is connected. It is a complete and optimal algorithm, meaning that it will always find a solution if one exists and the solution it finds will be the shortest possible path.

However, BFS has a high memory requirement, as it needs to store all the visited nodes in memory. It can also be slow for larger graphs, as it explores all the nodes at each level before moving to the next level.

Was it Solvable?

I was able to solve all the 3x3 board using BFS.

Limitations

The program had certain limitations. One issue was that when attempting to solve larger puzzles, an out of memory error could occur due to the use of HashMap to store previous states. Furthermore, even when attempting to solve a 4x4 puzzle, BFS could not always provide a solution if the puzzle was too complex, leading to a java heap space out of memory error.

Using Manhattan distance and Hamming distance

For solving the puzzle I used the Manhattan distance and Hamming distance later. By using Manhattan and Hamming I was able to solve puzzles till 7x7.

What is Manhattan and Hamming and how it works

Hamming distance and Manhattan distance are both used in computer science to measure the distance between two points in a graph, particularly in the context of solving problems such as the sliding puzzle.

Hamming distance is the number of positions at which the corresponding symbols in two strings or sequences are different. In the context of a sliding puzzle, the Hamming distance between two puzzle states is the number of tiles that are in the incorrect position. It essentially measures how close a given puzzle state is to the goal state.

Manhattan distance, on the other hand, measures the distance between two points by adding up the absolute differences between their x and y coordinates. In a sliding puzzle, the Manhattan distance is the sum of the distances between each tile and its goal position, measured in terms of the number of rows and columns between the two positions. This heuristic is often more accurate than the Hamming distance in predicting the number of moves required to reach the goal state.

Both Hamming and Manhattan distance are used as heuristic functions in search algorithms such as A* to guide the search towards the goal state. By estimating the number of moves required to reach the goal state, these heuristic functions enable the algorithm to explore the most promising paths first, leading to a more efficient and effective solution.

Limitations

While both the Manhattan and Hamming distance heuristics were successful in solving puzzles up to 7x7 in size, they were unable to solve puzzles of size 8x8 or 9x9. This suggests that for larger puzzles, more sophisticated approaches may be required. As the size of the puzzle increases, the search space grows exponentially, and it becomes increasingly difficult to find an optimal solution in a reasonable amount of time.

Using of Euclidean distance

For solving puzzles with bigger dimensions I used Euclidean distance.

What is Euclidean distance and how it works

Euclidean distance is a distance metric used to measure the distance between two points in a Euclidean space. It is based on the Pythagorean theorem, which states that in a right triangle,

the square of the length of the hypotenuse (the side opposite the right angle) is equal to the sum of the squares of the other two sides.

To calculate the Euclidean distance between two points (x1, y1) and (x2, y2), we first calculate the difference between the x-coordinates and the y-coordinates, respectively. We then square each of these differences, add them together, and take the square root of the sum to obtain the Euclidean distance between the two points.

Limitations

I wasn't able to solve all the bigger dimension puzzles.

Final Data structure

2D array : to store each piece of a board, eventually representing each board.

Hashmap : used as a closed set, keeps the predecessor of each board and their score value.

Linkedlist : a doubly linked list used to store the path of the solution.

Arraylist : used to store adjacent board states and valid movement options.

PriorityQueue : used in the form of a minheap as an open set. A custom way of sorting is used and the items in the lowest score are removed.

Enum : used in organizing and passing the heuristic type

Tested case and number of moves

board 26, 29, 33, 34 and 40 took long to solve (around 30-40 seconds).

For the following if you test it with different heuristics it will have different number of moves. If I use Euclidean distance to solve boards with smaller dimensions it will take more moves to solve them but not all of them like board2.txt (4 moves).

Test Results		
Board Name	Is solved	Moves
board1.txt	TRUE	14
board2.txt	TRUE	4
board3.txt	TRUE	83
board01.txt	TRUE	5
board02.txt	TRUE	4

board03.txt	TRUE	30
board04.txt	TRUE	3
board05.ttx	TRUE	67
board06.txt	TRUE	22
board07.txt	TRUE	4
board08.txt	TRUE	97
board09.txt	TRUE	73
board10.txt	TRUE	84
board11.txt	TRUE	62
board12.txt	TRUE	147
board13.txt	TRUE	186
board14.txt	TRUE	112
board15.txt	TRUE	432
board16.txt	TRUE	530
board17.txt	TRUE	433
board18.txt	TRUE	392
board19.txt	TRUE	384
board20.txt	TRUE	731
board21.txt	TRUE	784
board22.txt	TRUE	594
board23.txt	TRUE	745
board24.txt	TRUE	1147
board25.txt	TRUE	621
borad26.txt	TRUE	2287
board27.txt	TRUE	1478
board28.txt	TRUE	1534
board29.txt	TRUE	1538
board30.txt	TRUE	1418
board31.txt	TRUE	1167
board32.txt	TRUE	1649
board33.txt	TRUE	1932
board34.txt	TRUE	2251
board35.txt	FALSE	N/A
board36.txt	FALSE	N/A
board37.txt	FALSE	N/A
board38.txt	FALSE	N/A
board39.txt	FALSE	N/A
board40.txt	TRUE	2494

Conclusion

By using A* algorithm with using right heuristics we can solve NxN puzzle.

References

1. Wikipedia contributors. (2022, April 12). Breadth-first search. In Wikipedia, The Free Encyclopedia. Retrieved 09:06, April 15, 2023, from https://en.wikipedia.org/wiki/Breadth-first_search

2. Wikipedia contributors. (2022, March 29). Euclidean distance. In Wikipedia, The Free Encyclopedia. Retrieved 09:09, April 15, 2023, from https://en.wikipedia.org/wiki/Euclidean_distance

3. Wikipedia contributors. (2022, March 29). Manhattan distance. In Wikipedia, The Free Encyclopedia. Retrieved 09:11, April 15, 2023, from https://en.wikipedia.org/wiki/Manhattan_distance