Informatics Institute of Technology

Algorithms: Theory, Design, and Implementation

Coursework 2 – Report - Sliding Puzzles

Module	:	5SENG003C
Name	:	Shakinya Manivannan
Group	:	SE - H
Student ID	:	<20222163> / <w1985734>

1. Introduction

A well-known issue in computer science and recreational mathematics is the sliding puzzle game. We examine the application of a sliding problem solver program in this study. When a player must move tiles to attain a particular configuration in a sliding puzzle, the program's goal is to determine the quickest path to solve the puzzle.

2. Choice of Data Structure and Algorithm

- **Data Structure : 2D Arrays**

The sliding puzzle solver program uses a 2D array to represent the puzzle grid, with each cell representing a specific position on the grid. This structure simplifies tile access, swapping, checking neighbouring cells, and verifying the puzzle's configuration. Direct mapping of indices to grid coordinates simplifies puzzle manipulation and state tracking, enhancing the algorithm's efficiency.

- **Data Structure : Queue**

The queue data structure, which is part of the 2D array, is crucial in managing potential moves during the solving process. It operates on a first-in-first-out (FIFO) basis, ensuring positions are explored layer by layer according to the Breadth-First Search (BFS) algorithm. The queue facilitates this systematic traversal strategy, contributing to the efficiency of BFS-based algorithms by ensuring structured and memory-efficient exploration. The queue efficiently manages the exploration of possible moves, guiding the algorithm towards identifying the shortest path to solve the sliding puzzle.

- **Algorithm : Breadth-First Search (BFS)**

The sliding puzzle solver program uses a modified Breadth-First Search (BFS) algorithm to efficiently find the shortest path to the goal state. The algorithm starts by initializing a queue with the puzzle's starting configuration and iteratively dequeues configurations, exploring each configuration's adjacent states. It evaluates each potential move's proximity to the goal state and its contribution to the shortest path. The modified BFS algorithm optimizes the exploration process by avoiding redundant paths and prioritizing configurations with the most direct route to the goal. This approach enables the solver to navigate complex puzzle scenarios and deliver optimal solutions within a reasonable timeframe.

3. Run of Algorithm on a Small Benchmark Example

Benchmark Example :

```
. . . . . . . 0S .
. . . 00. . . 0.
. . 0. 0. . . . .
. . 0. 0F0. . .
0. . . . . . . . .
0. . 0. . . . . .
. 0. . . . . 0. .
. . 0. . 0. . . .
. . . . . . . . . .
. . . . 0. . . . .
```

```
Finding the shortest distance..

Total distance: 44 | START:  (9, 1)
RIGHT (10, 1)
DOWN (10, 10)
LEFT (6, 10)
UP (6, 9)
LEFT (1, 9)
DOWN (1, 10)
RIGHT (4, 10)
UP (4, 7)
LEFT (3, 7)
UP (3, 5)
LEFT (2, 5)
UP (2, 1)
RIGHT (7, 1)
DOWN (7, 3)
LEFT (6, 3)
DOWN (6, 4)


Time elapsed : 31 milliseconds
No of lines : 10
```

Performance Analysis

A. Empirical Study

The study focuses on the execution time of a sliding puzzle solver program across various puzzle configurations, analysing its efficiency and scalability. It compares the solver's performance with puzzles of varying sizes, such as 3x3, 4x4, and 5x5, to assess how it scales with the size of the puzzle grid. This helps identify trends or patterns in execution time as puzzle size increases. Observable patterns in the solver's execution time, such as linear or exponential growth with increasing puzzle size, fluctuations in performance under certain conditions, or consistent performance across different puzzle configurations, provide valuable insights into the solver's behaviour and potential optimization areas. Understanding these patterns can provide valuable insights into the solver's behaviour and potential areas for optimization.

B. Theoretical Considerations

The number of vertices (cells) in BFS is V, while the number of edges (possible motions) is E. The temporal complexity of BFS is therefore **O(V + E)**. BFS has linear time complexity, or O(N), in the setting of sliding puzzles, where N is the total number of cells in the maze. The effectiveness and scalability of the method are better understood thanks to this theoretical examination.

C. Order-of-Growth Classification (Big-O notation)

As the size of the input (maze) increases, BFS's performance also increases linearly, as seen by its linear time complexity, O(N). In bigger mazes where the number of cells rises proportionately, BFS's efficiency in completing sliding puzzle games is highlighted by this categorization.

4. Conclusion

In conclusion, the program that solves sliding puzzles successfully illustrates how data structures and algorithms may be used to tackle a common issue. The BFS-based strategy that was selected effectively determines the shortest path to solve the challenge. According to the performance analysis, [recap conclusions and observations]. In general, the software offers a strong solution for effectively resolving sliding puzzles.