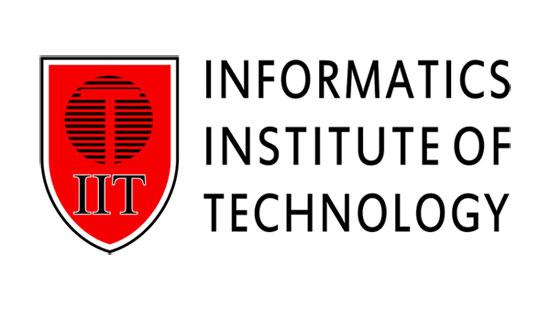
**Logo, company name

Description automatically generated**

Informatics Institute of Technology

affiliated with

University of Westminster

Algorithms: Theory, Design and Implementation

5SENG003C.2

Coursework I: Report

Rusini Thara Gunarathne

UoW No. – 18099940

IIT No. – 20200205

Tutorial Group G

Algorithmic Approach a) A short explanation of your choice of data structure and algorithm.

The Dijkstra Algorithm was chosen to be utilized in the algorithmic implementation of the ice sliding puzzle. Dijkstra's method is an iterative algorithmic approach which finds the shortest path from one starting node to all other nodes in a graph.

Some of the most used path finding algorithms for similar problems such as this are A\* and Breadth-First Search (BFS). BFS is very much similar to Dijkstra.

**Why Dijkstra is better than A\* and Breadth First Search (BFS)?**

Both BFS and Dijkstra belong to the SSSP (Single Source Shortest Path) class of algorithms, which address the issue of determining the shortest path from a starting node to all other nodes in the graph. Following the completion of the algorithm, the shortest paths from the source node to all other nodes in the graph will be found.

A\* algorithm is relatively faster than Dijkstra since it is an improvised version of Dijkstra, but if you have many target nodes and you don't know which one is closest to the main one, A\* is not very optimal. This is because it needs to be run several times (once per target node) to get to all of them.

Dijkstra is an uninformed algorithm. This means that it does not need to know the target node beforehand. For this reason, it's optimal in cases where you don't have any prior knowledge of the graph when you cannot estimate the distance between each node and the target. Since Dijkstra picks edges with the smallest cost at each step it usually covers a large area of the graph. This is especially useful when you have multiple target nodes, but you don't know which one is the closest.

**Why Priority Queue?**

because in a min heap priority queue, find min should be an O(1) operation. because dijkstra’s algorithm is a greedy algorithm, it always takes the edge with the lowest edge cost, thus, instead of having to iterate through the entire list of adjacent nodes for that vertex and comparing their edge costs, you simply perform an O(1) find-min

The priority queue selects the next vertex so as to (eventually) ensure shortest paths in a weighted graph. If you use a FIFO queue instead, you will not be able to account for arbitrary edge weights. This will essentially be breadth-first search which only guarantees finding shortest paths in unweighted graphs. The use of a priority queue comes at the cost of greater runtime - usually by a log factor.

The chosen data structure was Priority Queue. Since we want the path with smallest weight, we will have a minimum priority queue.

In Dijkstra's algorithm, the important step is selecting an unexplored vertex *v*such that there is an edge *(u, v)*in the graph, where *u*is an already explored vertex, and *d'(v) = dist(u)*+*cost(u, v)* is minimum. Here, *dist(u)* is the length of the already found shortest path from the source vertex *s*to vertex *u*, and *cost(u, v)* is the weight of the edge from *u*to *v*.

If we store the unexplored vertices in a simple array/linked list, we would have to iterate over the whole list each time to find the desired vertex *v,*with minimum *d'(v)*(*d'(v) = dist(u) + cost(u, v)*).

If we store the vertices in a priority queue with *d'(v)* as the key for each vertex, we can get the vertex with minimum *d'(v)* by using the Extract-Min operation. If we use a binary (min-)heap, the asymptotic complexity of Extract-Min operation will be ***O(log n)***.

Examples of running the Algorithm on Benchmark examples b) A run of your algorithm on a small benchmark example. This should include the supporting information as described in Task 4.

As benchmark examples the text files maze10\_3 and maze 20\_3 was chosen. And the following number of steps for the shortest path from S to F were generated for each of puzzles.

maze10\_3.txt Steps for the solution

A screenshot of a computer

Description automatically generated

maze20\_3.txt Steps for the solution

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidenceA screenshot of a computer

Description automatically generated

1. Empirical Analysis of Algorithm Performance - c) A performance analysis of your algorithmic design and implementation. This can be based either on an empirical study, e.g., doubling hypothesis, or on purely theoretical considerations, as discussed in the lectures and tutorials. It should include a suggested order-of-growth classification (Big-O notation).

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **File name** | **No. of elements** | **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Average** | **Change of Ratio** |
| puzzle\_21.txt | 441 | 0.071 | 0.069 | 0.062 | 0.063 | 0.059 | 0.0648 | 1.614198 |
| puzzle\_42.txt | 1764 | 0.165 | 0.112 | 0.08 | 0.086 | 0.08 | 0.1046 | 1.397706 |
| puzzle\_84.txt | 7056 | 0.11 | 0.134 | 0.187 | 0.177 | 0.123 | 0.1462 | 1.826265 |
| puzzle\_168.txt | 28224 | 0.248 | 0.299 | 0.323 | 0.233 | 0.232 | 0.267 | 1.719101 |
| puzzle\_336.txt | 112896 | 0.339 | 0.558 | 0.522 | 0.501 | 0.375 | 0.459 | 1.934205 |
| puzzle\_672.txt | 451584 | 0.86 | 0.856 | 0.865 | 1.06 | 0.798 | 0.8878 | 3.224825 |
| puzzle\_1344.txt | 1806336 | 2.857 | 2.78 | 2.946 | 2.8 | 2.932 | 2.863 | 4.098708 |
| puzzle\_2688.txt | 7225344 | 11.565 | 12.044 | 11.216 | 12.066 | 11.782 | 11.7346 | 0 |

Big-O notation is the ideal approach to represent the speed or complexity of a particular algorithm since it focuses on the worst-case scenario. Following the doubling hypothesis from the preceding chart, the run time in Big-O notation is O(N^2).