**Task 5 (30 marks).** Write a brief report (no more than 3 A4 pages) containing the following:

1. a) A short explanation of your choice of data structure and algorithm.
2. b) A run of your algorithm on a small benchmark example. This should include the supporting information as described in Task 4.
3. 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).
4. ---------------------------------------------------------------------------------------------------------------------------------------

Algorithmic Approach

Dijkstra Algorithm was used in the implementation

For similar problems where path finding algorithms involve some of the most used algorithms are A\* and BFS

The same puzzle can be solved using A\* and Breadth First Search (BFS).

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

Both BFS and Dijkstra fall under SSSP (Single Source Shortest Path) algorithms where it solves the problem of finding the shortest path from a starting node (source) to all other nodes inside the graph. After the algorithm ends, we’ll have the shortest paths from the source node to all other nodes in the graph.

A\* algorithm is relatively faster than 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?

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)***.

After selecting vertex *v*, and updating *dist(v)*, we may find that we have a shorter path to another unexplored vertex *w*through *v*, i.e. a path with cost *d'(w) = dist(v) + cost(v, w)*, which is less than the cost of the existing path to *w*.

This vertex will be in the priority queue, and its key value will need to be changed to the new value using the Decrease-Key operation, which will decrease the key of a certain element, and bubble it up if necessary to ensure the min-heap property. A binary (min-)heap will support doing Decrease-Key operation with ***O(log n)*** complexity.

Again, if we had used a simple list, this step would have required us to iterate over the whole list to update the key of a vertex.