1. **A high-level description of the overall solution strategy chosen for the shortest**

**path problem**

The C++ program initially reads the number of vertices and edges from the text file provided from the console. Then the program reads the edges and its weights, and generate an adjacency matrix to store weights between every 2 vertices where there is an edge. All other edges are assigned infinity where there is no edge.

This program uses A\* algorithm which uses heuristics to find the shortest path in the graph. The heuristic estimates the cost of the cheapest path from each vertex to the target which is done by calculating Euclidean distance for each vertex to the target vertex.

Once heuristics are calculated for each vertex, A\* algorithm is called. This algorithm is implemented, firstly by creating a distance array which would be holding the shortest distance from source to each vertex. Also there would be a parent array which would keep track of the previous node of each vertex in the shortest path. Initially distance array is assigned infinity except the source vertex which would be zero. The index of the source vertex in the distance array as well as weight stored in dist[source] is inserted into heap. In insert heap function, heuristic value of inserted vertex is added up with the original weight and kept in the heap.

For each vertex u extracted from the heap, the program would check if there is any path between vertex u and vertex v. If a path exists between u and v, it would update the distance of vertex v with the following statement.

dist[v] = dist[u] + graph[u][v]

Once the distance is updated, its index and value is again inserted into heap. The algorithm would keep on inserting and extracting weights until it reaches the target vertex and displays the shortest path.

A function which finds out the vertices in the shortest path, recursively goes through each of the element in the parent array and stores the vertices in a path array. Later on each vertex from this path is printed on the screen.

1. **A high-level description of the design and overall solution strategy you have**

**developed for the second shortest path problem**

The second shortest path is calculated by keeping each edge in the shortest path as infinity and call A\* algorithm one by one. Each shortest distance, its path and length at every iteration is inserted into the heap. The root node in the heap will provide the second shortest path. The reason for choosing heap in finding out the second shortest path is because the complexity of using heap in extracting the minimum value is O (log n). Other option would be to sort the array first and find out the shortest distance from that array which will be inefficient.

1. **A list of all of the data structures used, where they are used and the reasons for**

**their choice**

**Linear arrays**

**Where used: -**

* For storing vertices and edges, weights associated with each edge and coordinates for each vertex.
* For storing adjacency matrix.
* For storing parent of each vertex in the shortest path.
* For storing vertex nodes traversed in the shortest path.
* For storing distance of each node from the source.

**Reasons: -**

* Easy to store and access element in array.
* Both store and access takes constant time.
* Arrays take linear (O(n)) space in the number of elements n that they hold.

**Binary Heaps (Min Heap)**

**Where used: -**

* Storing index and vertex weights.
* Storing all shortest paths and distances while finding out shortest of second shortest path.

**Reasons: -**

* Weights can be inserted and extracted from the heap in O (log n) time.
* Shortest path can be inserted and extracted from the heap in O (log n) time.

1. **A list of any algorithms used, where they are used and why they are used**

**A\* Algorithm**: - A\* star algorithm is a variation of Dijkstra algorithm which uses heuristics to guide its search in finding out the shortest path between a source vertex and a destination vertex in an undirected or directed weighted graph having no negative weight cycles. The complexity for finding out the shortest path, if we use a binary heap as a priority queue is O(|E|+|V|log|V|).

**Binary Heaps (Min Heap):** - Priority queues can be efficiently implemented using Binary Heap because it supports insertion, deletion in O (log n) time.

Binary Heaps were used for implementing priority queues for storing index and weights and it is being used in A\* Algorithm for accessing the vertex having minimum cost. If we don’t implement priority queue in A\* Algorithm, the program would have to do a linear search in the distance array to find out the vertex which has the minimum weight and hasn’t been visited before.

**Questions**

1. **If we require that the second shortest path be longer than the shortest path?**

The proposed solution is correct in finding the longer second shortest path. It’s because the second shortest path would be strictly longer than the first shortest path every time when the algorithm finds out the second shortest path. In our case the weight associated with each edge is greater than the Euclidean distance between its start and end vertices, determined by their coordinates. Because of this reason, A\* star algorithm finds out the best possible solution.

So the path which requires less vertices to reach the destination node, will be detected first by this algorithm. It could be possible the second shortest path is of the same length as the first shortest path but it can’t be less than the first shortest path.

1. **If the graph contains cycles?**

The proposed solution will work fine if there are positive cycles in the graph. A\* algorithm doesn’t allow negative weight cycles because the shortest path in such graphs are undefined. The total weight of the shortest becomes lower each time when the cycle is traversed. If the graph has negative weights, we can use Bellman-Ford algorithm to find out the shortest path.

1. **If the graph is undirected?**

The proposed solution will work fine for undirected graph also. Since the undirected graph is basically same as a directed graph with bidirectional connections between two vertices. As explained earlier, A\* algorithm works on positively weighted graphs. So even If the graph happens to have any cycle it would still be able to find the shortest path, provided the cycle does not have any negative weight.

In A\* algorithm, whether the graph is directed or undirected, it doesn’t matter. It has to simply refer the adjacent vertex in the adjacency matrix to find the shortest distance value.