TRAFFIC LIGHT OPTIMIZATION USING GREEDY ALGORITHMS (DIJKSTRA'S ALGORITHM)

DESIGN AND ANALYSIS OF ALGORITHMS – 18CSC204J

MINI-PROJECT – CT4

Department and Branch: NWC, CSE

Register Numbers:

RA2111029010017, RA2111029010033, RA2111029010009, RA2111029010043

Section: Q2

CONTENTS

CONTRIBUTION TABLE

Members	Contributions
Anjana	Problem statement, definition and explanation, conclusion.
Gayathri	Design technique used and its explanation.
Meenakshi	Algorithm for the problem, and its explanation.
Shambhavi	Time complexity analysis.

TRAFFIC LIGHT OPTIMIZATION

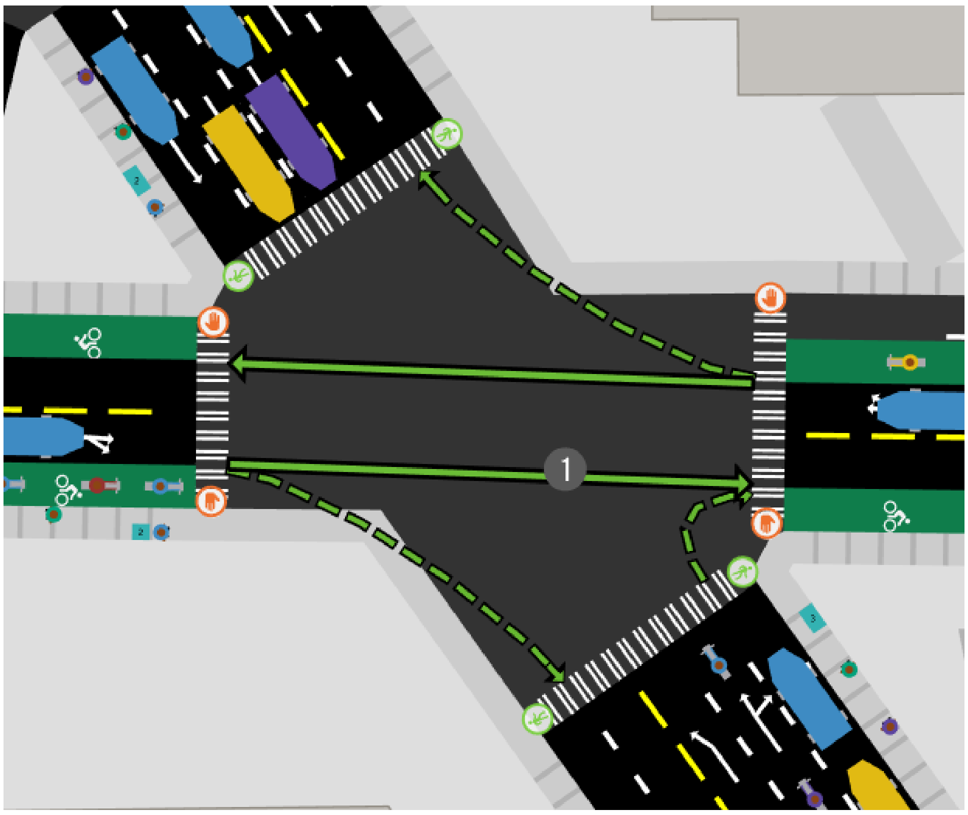
PROBLEM STATEMENT AND PROBLEM DEFINITION

One of the major issues in today's world is the growth in traffic on roads caused by vehicles. Many cities are impacted by traffic congestion and increased emissions from fuel use, which have a negative impact on the economy, the environment, and the overall quality of life. This calls for innovative solutions for road traffic management. The concepts of intelligent traffic systems began to pique interests at the turn of the twentieth century. Various approaches such as inductive loop detection, infrared sensors, video data processing, and others can be applied for traffic control. Software-based implementations of certain algorithms are applicable in line with the above approaches as well. *This report will explore the optimisation design technique of a greedy algorithm, with the objective of focussing on reducing traffic congestion and crossing time, thus enabling the smooth flow of traffic on a road network.*

In general, high crossing times and traffic congestions on roads are a result of the distance between two or more consecutive traffic lights stationed at their respective locations. When this distance is too large, issues such as increased delay, low capacity of the road network, increased risks of accidents due to insufficient gaps in speed of travel, lower efficiency of the road network as a result of lack of coordination in timing when signals work together. To explain this further, the following example is discussed:

.

In this case, a four-way junction is taken into consideration. In order to ensure that the vehicles do not bump into each other and follow their routes (as marked by the numbers), the presence of the 4 traffic lights enhances this process and enables the smooth flow of traffic. Although this is a simplified case, the significance of focussing on this approach comes to light when there are large inflows/outflows of traffic in real life. Four-way intersections can even get more complex, as depicted in the image below:



In this scenario, there are multiple curvatures on each side that would need to be travelled through by the vehicles on these roads. Larger vehicles would need information ahead of time, as to whether they need to stop, or can continue/start to move ahead on their paths. Traffic lights are of great significance in such scenarios, and their placements should be as near as possible to ensure clarity of directions for the proper flow of traffic in those road networks.

Based on these factors, it is evident that finding optimal distances between traffic lights such that they work efficiently in unison and do not interfere with each other would be a suitable solution. Thus, we proceed with the optimal solution in finding the appropriate least distance between two or more traffic lights that are fixed at their respective positions.

DESIGN TECHNIQUE USED

Discussion of Dijkstra's Algorithm for Traffic Light Optimization: Effectiveness and Limitations

The design technique used as a solution to the traffic light optimization problem discussed in this report is that of the Greedy Algorithm, specifically, the Dijkstra's algorithm. A Greedy algorithm is an algorithmic paradigm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Using Dijkstra's Algorithm for Traffic Light Optimization:

To use Dijkstra's algorithm for traffic light optimisation, we need to represent the road network with traffic lights as nodes and road segments as edges. Each traffic light node is assigned a weight that corresponds to the typical length of time that vehicles spend waiting at that intersection, and each road segment's length is used to determine the weight of the corresponding edge.

The algorithm starts with the current traffic light and calculates the total waiting time for vehicles along the path by adding the weights of the nodes (traffic lights) and edges (road segments). It then determines the shortest path to each adjacent traffic light and updates the traffic light states to give the traffic lights along the path priority.

By repeating this process for every traffic light in the network, we can optimize the traffic lights to reduce overall waiting times and facilitate a smooth traffic flow through the road network. This approach can be particularly useful in heavily congested areas and can help reduce travel times and fuel consumption, as well as improve safety and reduce emissions.

Effectiveness of Dijkstra's Algorithm for Traffic Light Optimization:

Dijkstra's algorithm is a potent tool for improving traffic lights in a weighted graph. Dijkstra's algorithm can help find the shortest path between two adjacent traffic lights while minimizing the total waiting time of vehicles at all intersections along the path by considering the weight of the edges and selecting the locally optimal choice at each stage.

Limitations of Dijkstra's Algorithm:

It's important to note that while Dijkstra's algorithm can be an effective solution to traffic light optimization, it may not always provide the optimal solution in all situations. This is because there may be cases where the locally optimal choice at each stage may not lead to the best overall solution.

ALGORITHM FOR THE PROBLEM, EXPLANATION OF THE ALGORITHM

The step-by-step explanation of the greedy algorithm – Dijkstra's algorithm and its implementation in determining the shortest distance between two or more traffic lights, by taking the road network as a graph, is given below:

1.Initialize the graph with nodes and edges and assign a weight to each edge.

2.Assign a tentative distance value to each node: set it to zero for the start node and to infinity for all other nodes.

3.Create a set of unvisited nodes and add all nodes to it.

4.Create an empty set of visited nodes.

5.While the set of unvisited nodes is not empty:

- **a**. Select the unvisited node with the smallest tentative distance.
- **b**. For each neighbour of the selected node:
  - **I.** Calculate the tentative distance to that neighbour by adding the weight of the connecting edge to the tentative distance of the selected node.
  - **II**. If the tentative distance is smaller than the current distance of the neighbour node, update the neighbour node's distance to the tentative distance.
- **c.** Add the selected node to the set of visited nodes and remove it from the set of unvisited nodes.
- **d.** If the destination node has been visited, stop the search and return the shortest path and distance. Otherwise, continue to step 5.

6.If the destination node has not been visited, there is no path from the start node to the destination node.

An example of an algorithm that can be expanded upon here in line with the traffic light optimization problem is the algorithm used in the N-Queens problem, which involves finding and placing “Queens” on a chessboard such that, no two queens interfere with each other vertically, horizontally and diagonally. The N-Queens problem is primarily best solved through the backtracking approach. While there might not always be a single algorithm that will always be a fit in the solution of a problem, combining multiple approaches can lead to even more optimal resources when implemented appropriately. This is of relevance in the case of traffic light optimization, as backtracking can be applied to determine whether the route depicted from the output of Dijkstra's algorithm's implementation is the shortest out of all combinations, thus enabling even more accurate results as outputs of the implementation in route planning for traffic light optimization.

IMPLEMENTATION

The implementation, along with documented explanations of each line for the algorithm, as a program in Python is as given below:

```
import heapq
def dijkstra_shortest_path(graph, start, end):
# Initialize distances to all nodes as infinity
distances = {vertex: float('inf') for vertex in graph}
# Distance to the starting node is 0
distances[start] = 0

# Create a priority queue to keep track of the unvisited nodes
priority_queue = [(0, start)]

while priority_queue:
# Get the node with the smallest tentative distance
current_distance, current_vertex = heapq.heappop(priority_queue)

# If the current node is the destination, we are done
if current_vertex == end:
# Return the shortest distance and path
```

```
shortest_distance = current_distance
shortest_path = []
while current_vertex != start:
shortest_path.append(current_vertex)
current_vertex = previous_vertices[current_vertex]
shortest_path.append(start)
shortest_path.reverse()
return shortest_distance, shortest_path
# If the current node hasn't been visited yet, add it to visited nodes
if distances[current_vertex] < current_distance:
continue
# Check all neighbors of the current node
for neighbor, weight in graph[current_vertex].items():
distance = current_distance + weight
if distance < distances[neighbor]:
# Update the tentative distance of the neighbor
distances[neighbor] = distance
# Record the path by saving the previous vertex
previous_vertices[neighbor] = current_vertex
# Add the neighbor to the priority queue
heapq.heappush(priority_queue, (distance, neighbor))
# If the priority queue is empty and we haven't reached the destination, there's no path
return float('inf'), []
```

The graph input is a dictionary of nodes, edges, and their weights, with start and end inputs representing starting and ending nodes. This can be understood further with a sample input and output as shown below:

Sample Input:

```
# If the nodes and weights of the graph are given as:
graph = {'A': {'B': 1, 'C': 2},
        'B': {'D': 3},
        'C': {'D': 1}, 'D': {}}
start = 'A'
end = 'D'
print(shortest_path(graph, start, end))
```

Sample Output:

```
(3, ['A', 'B', 'D'])
```

From the above, It can be inferred that the shortest distance in the input graph between the beginning node "A" and the destination node "D" may be calculated from the output shown above as being 3. This distance is determined by summing the edge weights along the (A-B, B-D) edges, which have weights of 1 and 3, respectively, along the shortest path from 'A' to 'D'.

In addition, it can be observed that ['A', 'B', 'D'] is the shortest route from 'A' to 'D. This path is made up of a series of nodes that travels the shortest distance possible between the starting and ending nodes.

COMPLEXITY ANALYSIS

The time complexity of Dijkstra’s Algorithm is O(V^2) using the adjacency matrix representation of graph. The time complexity can be reduced to O((V+E)logV) using adjacency list representation of graph, where E is the number of edges in the graph and V is the number of vertices in the graph.

Time complexity of Dijkstra's algorithm is

O(V^2) where V is the number of vertices in the graph.

It can be explained as below:

First thing we need to do is find the unvisited vertex with the smallest path. For that we require

O(V) time as we need check all the vertices.

Now for each vertex selected as above, we need to relax its neighbours which means to update each neighbour's path to the smaller value between its current path or to the newly found. The time required to relax one neighbour comes out to be of order of O(1) (constant time).

For each vertex we need to relax all its neighbours, and a vertex can have at most V-1 neighbours, so the time required to update all neighbours of a vertex comes out to be [O(V) * O(1)] = O(V)

So now following the above conditions, we get:

Time for visiting all vertices =O(V)

Time required for processing one vertex=O(V)

Time required for visiting and processing all the vertices

=O(V)∗O(V)=O(V^2)

So, the time complexity of Dijkstra's algorithm using adjacency matrix representation comes out to be O(V^2).

Worst Case: O(V^2)

Average Case: O(V^2)

Best Case: O((V+E)logV); where E is the number of edges in the graph and V is the number of vertices in the graph.

CONCLUSION

Thus, through a thorough analysis of the greedy algorithm - Dijkstra's, the factors of its application in the problem of traffic light optimization addressed through finding the optimal least distance between two or more traffic lights, was made possible through investigating its nature in terms of the steps involved in the algorithm, as well as its time complexity in solving this problem [O(V^2) in its worst and average case scenarios, and O((V+E)logV) in the best case, where V --> vertices indicating the distances, and E --> Edges or Nodes, representing the traffic lights]. Not only is it important to ensure the appropriateness of the proximity of traffic lights based on the factors of waiting time/delays and speed limits, but also traffic volume and optimal green light for each direction, analysing the traffic patterns and types of vehicles over time, safety, cost effectiveness and considerations of future growth trends in traffic as well, all of which would provide more efficient outcomes as per the needs of the situation being handled. Potential extensions of research along the line of traffic light optimization include expanding upon TSCIP (intersection traffic signal control problem), which considers factors such as real-time strategies, signal timing constraints, rapid developments in traffic systems with the goal of maximizing traffic flow, utilization of ITS (Intelligence Transport Systems) in the application of computing, information, and communications technologies to the real-time management of vehicles and networks, combinations with other algorithms such as backtracking, and working further with sensors and other relevant systems to obtain desired results.

REFERENCES

- https://www.sciencedirect.com/science/article/pii/S1877042818300387

- https://etrr.springeropen.com/articles/10.1186/s12544-020-00439-1

- https://etrr.springeropen.com/articles/10.1186/s12544-020-00440-8

- From Dijkstra's algorithm to route planning (incl. bidirectional search & transit node routing)

- https://www.scaler.com/topics/data-structures/dijkstra-algorithm/

-https://www.simplilearn.com/tutorials/data-structure-tutorial/greedy-algorithm

- https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/

https://www.researchgate.net/publication/323928657_Reducing_a_congestion_with_introduce_the_greedy_algorithm_on_traffic_light_control