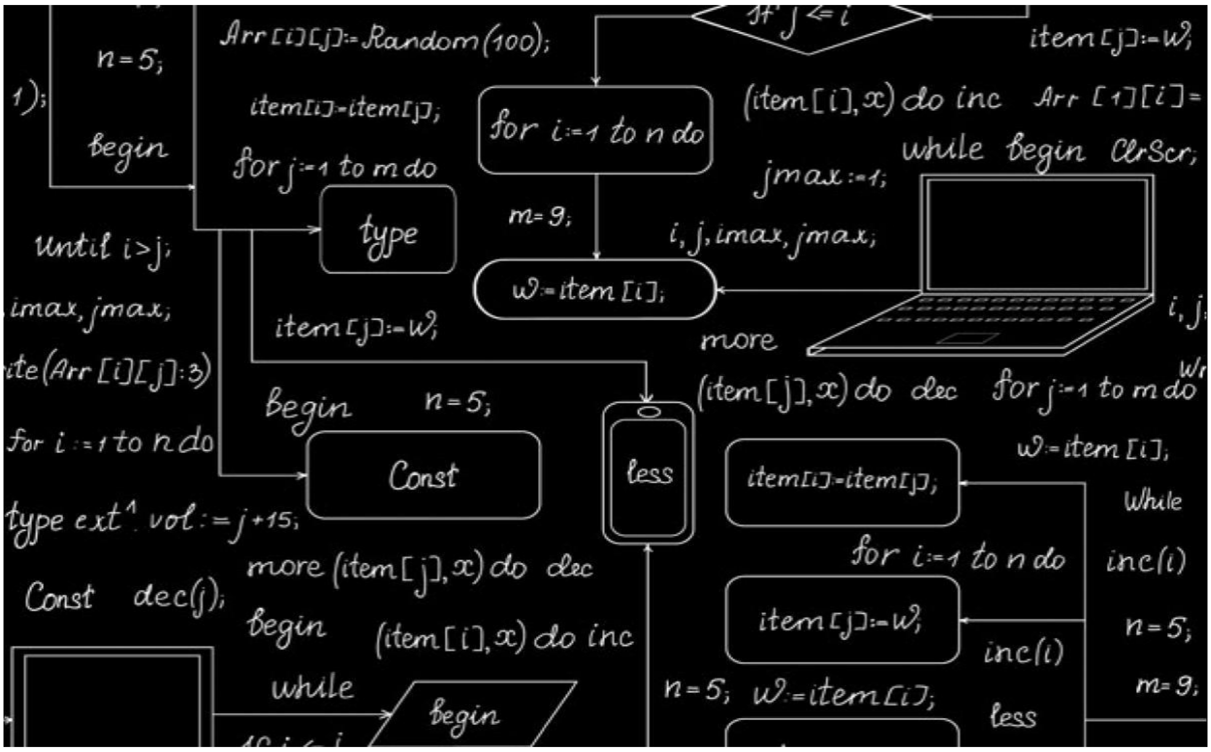
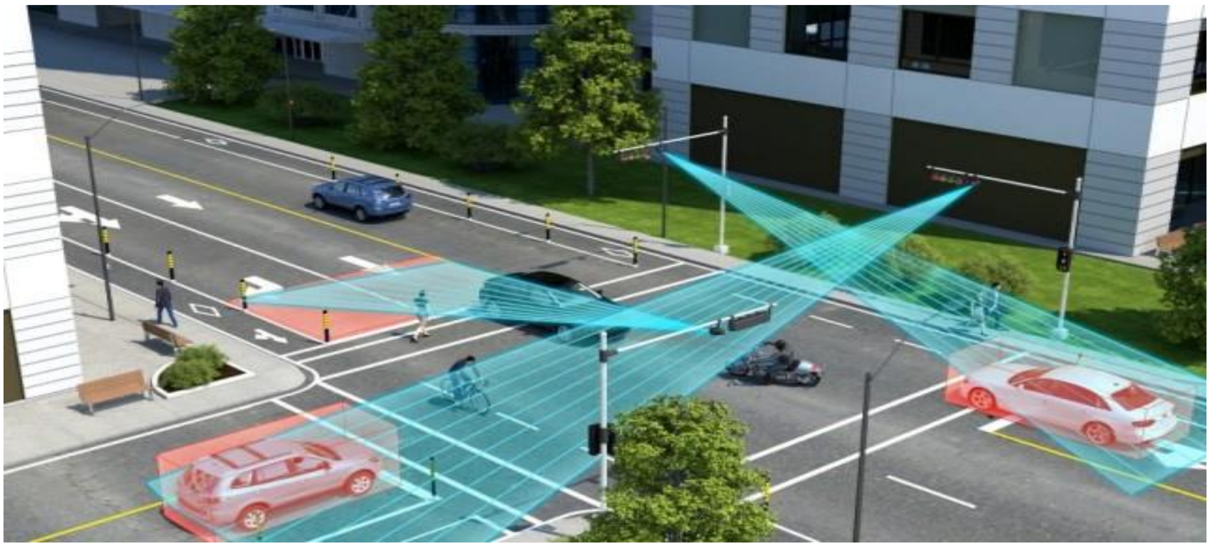


Algorithm Analysis and Design Project

Traffic Management System



by
Aaryan Sharma (2020115008)
aaryan.sharma@research.iiit.ac.in

Introduction:

With the continuously developing infrastructure, rapid globalization and increasing population, quick mobility is one of the primary needs. We see people using different modes of transports to move from one place to another. Sometimes, the route is as easy as going to a friend's house two streets away or using airways to move to a different state or country. But in everyday life, the situation is a bit complex and tricky. And with the increasing number of people and therefore increasing number of vehicles, it is becoming more and more essential to have a traffic management system to ensure the smooth functioning of a region.

What happens if we don't have a proper system for movement from one place to another? A system is necessary for smooth functioning, and a lack of a system would lead to unfavourable situations such as accidents. Moreover, the infrastructure requirement is proportional to the population growth, whereas the population is growing much faster than the infrastructure currently, which increases the risk of such accidents.

Another aspect we can consider is the increased emissions and greenhouse effect due to the extensive use of a large number of vehicles. Therefore, lack of a system leads to unfortunate economic losses and health hazards.

In this project, I will try to develop an algorithm that can be used to develop a traffic management system that ensures the smooth functioning of vehicles and saves one's time and fuel, thereby having positive impacts on society and the environment.

One of the most basic approaches to reducing traffic is reducing the number of cars, which can be done quickly if every person reaches their destination as soon as possible. It can be achieved if every person has knowledge of the shortest path available to them.

In this project, we imagined the city as a Graph, with the roads acting as the edges and the junction of roads as the vertices.

Firstly what is a graph?
A Graph G is a non-linear data structure consisting of a finite number of nodes (or vertices) V and edges E (arcs or lines) connecting a pair of nodes in a graph. It can also be said as

$$G = (V,E)$$

Graphs could be interpreted to represent networks, including the path of cities, telephone networks, or social networks. Each node itself is a structure that can be used to store information about any data. Say location, name etc. For example, every account on LinkedIn is a unique node. Similarly, every place holds a unique identity, which can be used to identify it as a node.

On the other hand, the edges in a graph sometimes have a weight, which indicates a strength (or any other attribute) between two nodes. For example, the nodes represent airports for an airport network, edges represent flights between them, and weight may represent the number of flights between two airports.

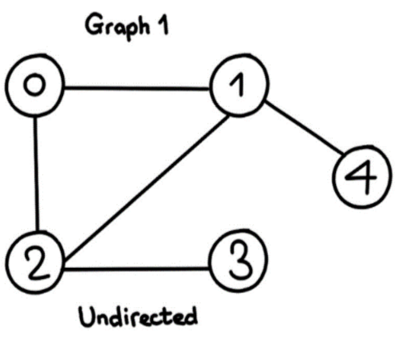
There are two types of graphs:

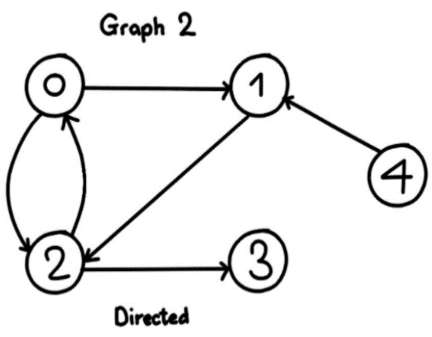
1.) Undirected Graphs

- In an **Undirected graph,** the edges represent a two-way relationship that does not have a direction, i.e., edge traversing in both directions is possible.

2.) Directed Graphs

- In a **Directed graph,** the edges represent a one-way relationship in a definite direction. In a directed graph, one can move in only one direction along an edge.



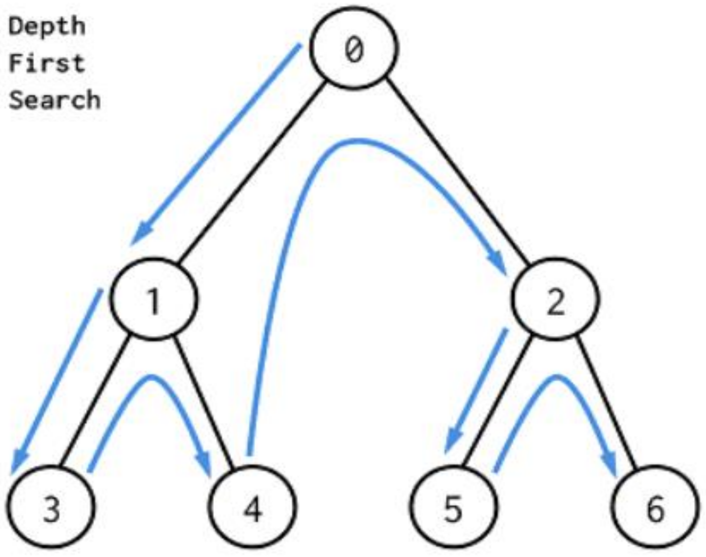


From the knowledge we have attained so far, we imagine our city as a directed graph with edges representing the roads, nodes representing the junctions and weights representing the distance between two nodes of the graph. The lesser the weight, the shorter will be our path. Now, we will have to figure out how we will move from one place to another, i.e. one node to another, in the most efficient way.

Let's start with discussing the naïve approaches and problems with them:

1.) Depth-First Search (DFS)

- DFS is an algorithm used to traverse or search in tree or graph data structure. It consists of choosing a root node (starting point) and traversing as far as possible along every branch before backtracking.
- The basic idea is to start a root or an arbitrary node, mark it, move to the next unmarked adjacent mode, and continue to explore every unmarked node. Then again, backtracking and checking for other unmarked nodes and finally printing the nodes in the desired path.
- We have to create a Boolean visited array to mark visited vertices while implementing the DFS approach as the graph may contain a cycle, and the algorithm will keep visiting the same vertex again and again.
- The algorithm can easily be simulated by a recursive call or an iterative approach involving a stack.
- The stack may contain the same vertex twice, so we need to check the visited set before printing. At each vertex, the function calls for every unvisited vertex in the adjacency list and at every instance, we dig one level deeper.
- Time Complexity is O(V+E), where V and E represent the number of vertices and edges in the graph.



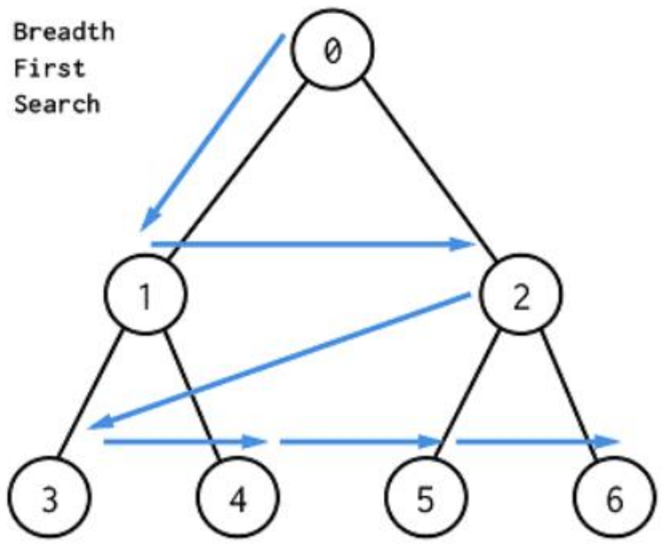
Pseudocode for DFS

```
DFS($G$)
1  for each vertex $u \in G.V$
2      $u.color$ = WHITE
3      $u.\pi$ = NIL
4  $time = 0$
5  for each vertex $u \in G.V$
6      if $u.color$ == WHITE
7          DFS-VISIT($G, u$)

DFS-VISIT($G, u$)
1   $time = time + 1$            // white vertex $u$ has just been discovered
2   $u.d = time$
3   $u.color$ = GRAY
4   for each $v \in G.Adj[u]$    // explore edge $(u, v)$
5       if $v.color$ == WHITE
6           $v.\pi = u$
7           DFS-VISIT($G, v$)
8   $u.color$ = BLACK            // blacken $u$; it is finished
9   $time = time + 1$
10  $u.f = time$
```

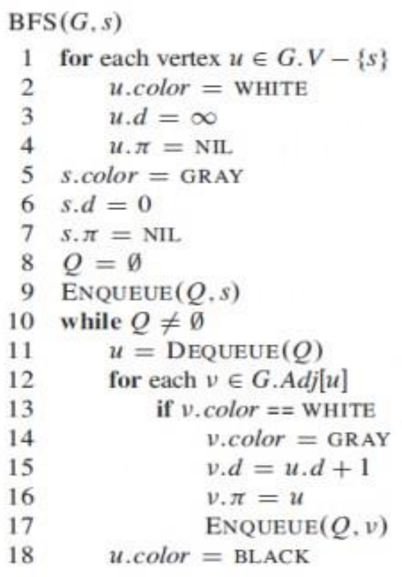
Source: Cormen, T. H., & Cormen, T. H. (2001). *Introduction to algorithms*. Cambridge, Mass: MIT Press.

2.) Breadth-First Search (BFS)

- BFS is an algorithm used to search in a tree or graph that works because vertices of the upcoming levels are queued up for checking at every level. Unlike the DFS algorithm, it ventures into its neighbourhood before entering the depth.
- While defining the BFS for the shortest path between the source node to any particular node, the algorithm ensures that the distance we find to each vertex is the minimum distance from each vertex IF the edges have the same weight.
- A BFS algorithm uses a queue for the implementation. Each successive vertex connected to the current one is inserted into the queue at each vertex. It can be observed that points are inserted into the queue so that the nearest points are inserted into the queue first. Each vertex is visited once without any repetitions.
- However, the major disadvantage of BFS is that it is valid only for unweighted graphs. For a source S and destination D, let an intermediate node M exist, and the weight of the edges be S-> M, M->D, and S->D be 1,2,10 respectively. According to BFS, the answer would be S->D as it is only one edge long, whereas the part S-> M->D is two edges long, but weight is only 3. So, here BFS fails to provide the best possible solution.
- Time Complexity is O(V+E), where V and E represent the number of vertices and edges in the graph.



Pseudocode for BFS



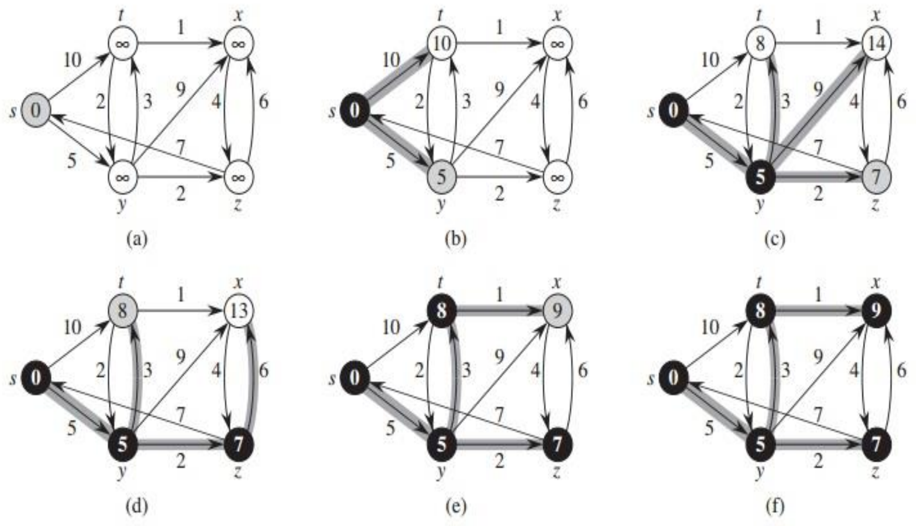
```
BFS($G, s$)
for each vertex $u \in G.V - \{s\}$
    $u.color$ = WHITE
    $u.d = \infty$
    $u.\pi$ = NIL
$s.color$ = GRAY
$s.d = 0$
$s.\pi$ = NIL
$Q = \emptyset$
ENQUEUE($Q, s$)
while $Q \neq \emptyset$
    $u$ = DEQUEUE($Q$)
    for each $v \in G.Adj[u]$
        if $v.color$ == WHITE
            $v.color$ = GRAY
            $v.d = u.d + 1$
            $v.\pi = u$
            ENQUEUE($Q, v$)
    $u.color$ = BLACK
```

Source: Cormen, T. H., & Cormen, T. H. (2001). *Introduction to algorithms*. Cambridge, Mass: MIT Press.

The major problem of the BFS search was that it couldn't be employed in unweighted graphs or graphs where edges have unequal weight as BFS focuses on the minimum number of edges. And After studying the DFS and BFS approach, we would study the most practical algorithm used to find the shortest path between nodes of a weighted graph, i.e. closest to the real-world application.

3.) Dijkstra's Algorithm:

- Suppose we replace the implementation of a queue in a BFS algorithm with a data structure that provides us with the shortest node at every point. This can be done by employing a priority queue or a heap, which provides the smallest element in O(logN) time.
- A minheap is a complete binary tree for which both children are greater than or equal to the parent at each node.
- In Dijkstra's algorithm, we use a minheap. For a graph with equal weights, Dijkstra would look like BFS. By the nature of Dijkstra, it doesn't visit every node; instead, the distance to each node is always minimum from the centre.
- For Dijkstra's algorithm in a graph, we start by identifying an initial node, and the source node is assigned distance 0, whereas all other nodes are considered to be at infinite distance.
- Create a set sptSet (shortest path tree set) that monitors the vertices included in the shortest path tree. Initially, it is an empty set, whereas we add vertices when we move forward, and a visited node is never added to the set of unvisited nodes ever again.
- If we reach the destination or the smallest distance among the unvisited nodes is infinity, the algorithm finishes.



Source: Cormen, T. H., & Cormen, T. H. (2001). *Introduction to algorithms*. Cambridge, Mass: MIT Press.

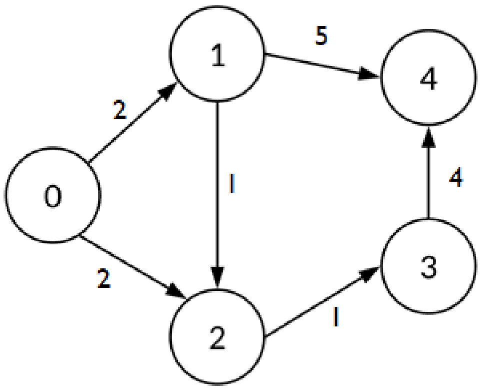
Pseudocode for Dijkstra's Algorithm

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:             // Initialization
        dist[v] ← INFINITY                  // Unknown distance from source to v
        prev[v] ← UNDEFINED                 // Previous node in optimal path from source
        add v to Q                          // All nodes initially in Q (unvisited nodes)

    dist[source] ← 0                        // Distance from source to source

    while Q is not empty:
        u ← vertex in Q with min dist[u]    // Source node will be selected first
        remove u from Q

        for each neighbor v of u:           // where v is still in Q.
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:               // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

However, the Dijkstra algorithm is considered one of the best algorithms to find the shortest path between two desirable nodes in a weighted graph, but can we conclude that opting for the shortest path is the most efficient way of managing the traffic.

NO.

Let us consider the following graph, where the weight represents the length of the edge:



Let us assume that a person named Vivek wants to go from node 0 to 2. According to Dijkstra, the most efficient path would be 0->2.

But If 0->2 is a busy single lane road with too many traffic signals and too many plotholes, with a traffic jam. And on the other hand, 0->1->2 is a four-lane road with minimum traffic. Then the second path is advisable and eco-friendly. How?

The shortest path method is ideal for someone if they are the only one travelling in the whole city, which is impossible in the real world. At any given number of times, several vehicles are already present on the roads. Firstly it would save one's time from the jam and avoid the exhaust from vehicles. Moreover, despite covering a long distance, it would be economical. For this, we need to understand the working of a car engine. An average Indian car works most efficiently at 40-60 kmph and mileage up to 25 kmpl. However, a traffic jam won't allow one to go above 10 kmph, which would allow a maximum mileage of 10-12 kmpl. So, opting for the first route costs about 3.3 times more time and 1.6 times more fuel. So, we can conclude that the shortest path is not always the ideal path.

So, here to make a more efficient system, we should consider the concept of flow. Or we can say the capacity of a road. As we know, the capacity of any road is limited. For smooth transportation, we can not allow all the cars to pass through the same road; instead, they should be diverted among roads, in such a way that none of the roads has to reach more than its total capacity, which may lead to accidents or traffic jams.

So, to develop an efficient system, apart from knowing the most minor paths, maximizing the flow of vehicles is equally important.

The algorithm revolving around maximization of the flow comes from the Ford-Fulkerson Method.

Ford Fulkerson Algorithm for maximizing flow:

Every graph represents a network, and each edge in the network has its capacity. Between any two nodes named source 's' and sink 't', we can find the maximum possible flow between these nodes by following two constraints:

1. Flow on any edge can never exceed the capacity of the edge.
2. Incoming flow for any node is equal to outgoing flow except for source and sink.

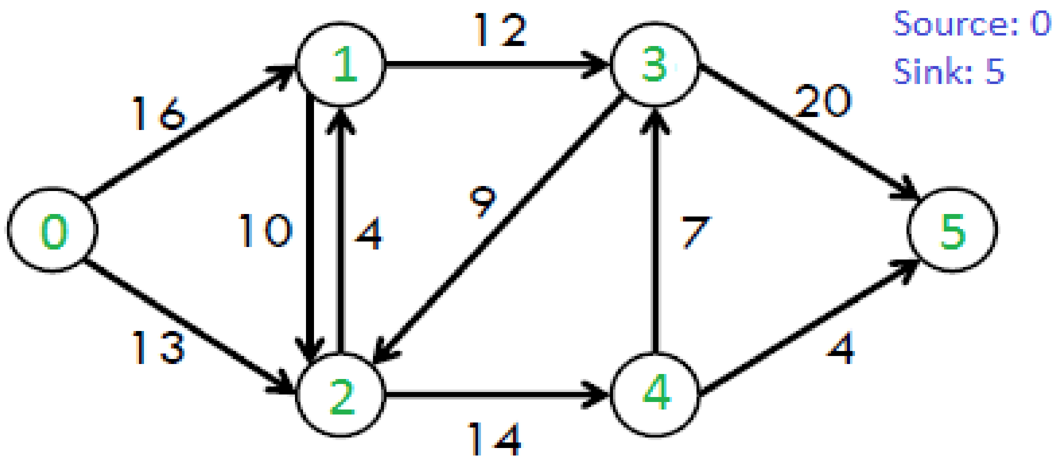
Algorithm:

- Initiate with the flow as 0.
- While augmenting the path from source to sink, add this path flow to flow.
- Return the flow.

Pseudocode for FFA:

```
1  for each edge (u, v) ∈ E [G]
2      do f [u, v] = 0
3         f [v, u] = 0
4  while there exists a path p from s to t in the
   residual network G_f
5      do c_f (p) = min {c_f (u, v) | (u, v) is in p}
6         for each edge (u, v) in p
7             do f [u, v] = f [u, v] + c_f (p)
8                f [v, u] = - f [u, v]
```

Problem on Ford Fulkerson Maximum flow:



In the given problem, the maximum flow is 23, which means that at any given time, only 23 vehicles can go from 0 to 5. However, this leads us to the dilemma of which vehicle should opt for which route.

This can be done by prioritizing the vehicles. We all agree that emergency vehicles such as ambulances, fire vans, police cars should be given maximum priority. Similarly, we can also prioritize the roads based on their situation, flow, traffic signals, length of the route, and other criteria.

Based on the Ford Fulkerson Algorithm (FFA), we developed a PRAFT: Prioritized Routing Assistant for Flow of Traffic algorithm, with the help of the Uniform Cost Search (UCS) algorithm.

PRAFT: Prioritized Routing Assistant for Flow of Traffic

Similar to FFA, PRAFT works on the directed graphs with a source and a sink while aiming to maximize the flow of vehicles while assigning them different routes based on their priorities. And the priority is directly proportional to the quality of roads. For example, higher priority vehicles get a higher quality of roads. The priority of vehicles can be assigned by human knowledge, where the quality of roads can be decided by a function, based on various factors such as flow, traffic signals, length of the route, expected time etc. Moreover, we can try to prioritize these factors as much as we can. For example, someone would prefer less time in the morning while rushing to the office, unlike the scenic views while returning.

Notations-

- G(V, E) - Graph with V vertices and E edges
- G_f(V, E) - Residual Graph with V vertices and E edges
- c(u,v) - Capacity of the edge from u to v
- c_f(u,v) - Capacity of the edge from u to v in the residual graph
- f(u,v) - Flow of edge from u to v
- s - Source node
- t - Sink Node
- T(s,t) - Total Flow from s to t
- pf - Priority Function

Pseudo Code for PRAFT

- **Input -** The input consists of a network G=(V, E) with flow capacity c for every edge in the network and a priority function pf for priority values of the edges, a source node s and a sink node t.
- **Output -** The algorithm finds the maximum flow from source s to sink t and obtain paths in order of priority.
- **Initialization -** For all edges in G, initialize $f(u,v) \rightarrow 0$ and $T(s,t) \rightarrow 0$
- **Iteration -**
    - While there exists a path p from s to t in G_f found using UCS(which uses priority queue) such that $c_f(u,v) > 0$ for all edges $(u,v) \in p$ and sum of pf of all $(u,v) \in p$ is lowest across all possible paths which UCS ensures.
        - Find $c_f(p) = \min \{ c_f(u,v) : (u,v) \in p \}$
        - For each edge $(u,v) \in p$:
            - $f(u,v) \leftarrow f(u,v) + c_f(p)$ (send flow along the path)
            - $f(v,u) \leftarrow f(v,u) - c_f(p)$ (the flow might be returned later)

```
        ▪ End For
        ▪ Store p and corresponding $c_f(p)$
        ▪ $T(s,t) \leftarrow T(s,t) + c_f(p)$
    o End While
```

Uniform Cost Search (UCS)

- A vital step in FFA is the identification of augmenting path in each iteration, which can be obtained using BFS (or DFS, depending upon the graph).
- The Uniform Cost Search (UCS) algorithm is also a variant of Dijkstra's algorithm, where rather than inserting all vertices in the priority queue, and we insert the source node only, then one by one as per the need.
- At every step, we check if the item is in the priority queue, whereas, If it is there, we perform the decrease key, else insert the item.
- Since UCS follows the path cost; therefore, our priority function, rather than path length only, at every step, the successive node n to be expanded so that the cost remains lower, i.e. the sum of edge cost from the source to node n.

Pseudocode for UCS Algorithm

```
• Initialization - vis(u)$\leftarrow$0 for all vertices u in G and cost(u)$\leftarrow\infty$ for all vertices u in G other than s and cost(s)$\leftarrow$0. Initialize an empty priority queue of pairs q, which will form a minheap using the first element of the pair by inserting (0,s).
• Iteration -
    o While q is non-empty:
        ▪ The top element of q is (val,u)
        ▪ If vis(u) is 0:
            ▪ For every edge from u to a vertex v in $G_f$ such that vis(v) is 0 :
                ▪ cost(v) $\leftarrow$ min {cost(v) , val + pf value of edge (u,v)}
            ▪ End For
            ▪ vis(u) = 1
        ▪ q.pop
        ▪ Store p and corresponding $c_f(p)$
• Return - The value of cost(t), i.e. the minimum cost to reach the target in $G_f$.
```

Conclusion:

Throughout the discourse of the project, we discussed various methods to traverse from one node to another. Every algorithm has its perks and disadvantages. We concluded that going absolutely for the shortest path or maximum flow can never lead to a well-functioning traffic system.

However, using the Prioritized Routing Assistant for Flow of Traffic, it becomes easier to generate a priority function as per our needs and call of the situation. As the PRAFT is based on FFA, I have written the code for the FFA algorithm and PRAFT algorithm, and upon running both for the same graphs, the results are the same as desired. However, the PRAFT algorithm acts as an updated FFA, where we used Dijkstra to get the most desirable path and system to manage the traffic of any city.

For the real-life implementation of the same, we can further require lots of data for a city, which is available in government offices, or we can add other aspects in priority function too.

PLEASE READ THE README.txt FILE.

References:

- Cormen, T. H., & Cormen, T. H. (2001). *Introduction to algorithms*. Cambridge, Mass: MIT Press.
- Behavioural analysis and prioritized routing of vehicular traffic during an emergency evacuation, G. Gupta, 2011