PROJECT

On

Simple Navigator for LPU

System is expected to give the comfortable route to reach any destination from any place inside the LPU-Campus. (for Individual / for Car)

By

Tutun Mandal

11801600 (A23)

K18KH

School of Computer Science and Engineering

Lovely Professional University

Phagwara, Punjab (India)

Git Hub Link:-

https://github.com/tutun2711/AI-project.git

CONTENTS:-

Abstract:-

We were given “SIMPLE NAVIGATION SYSTEM for LPU”as our project, System is expected to give the comfortable route to reach any destination from any place inside the LPU-Campus. (for Individual / for Car)

This didn’t turned out to be the easiest.

We were really interested to learn something new throught this project. We tried a lot to cover the whole campus, however there a a few loop holes in this which we gracefully accept.

We used the “Djikstra's Shortest Path Algorithm”to do so, I personally find it best way to find the best path for navigation.

Djikstra’s algorithm is a path-finding algorithm, similar to routing and navigation.

We will be using it to find the shortest path between two nodes in a graph.

It fans away from the starting node by visiting the next node of the lowest weight and continues to do so until the next node of the lowest weight is the end node.

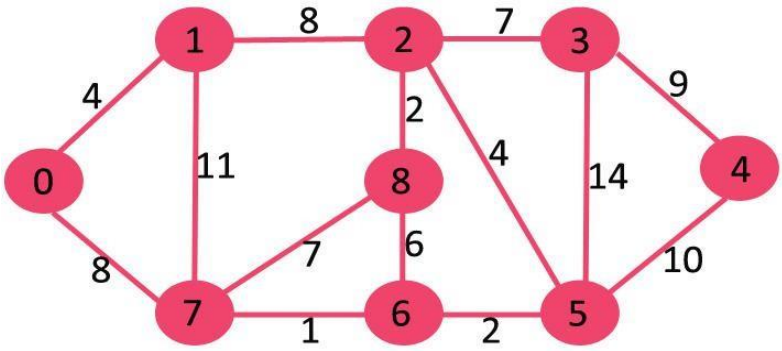
Introduction:-

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.
Algorithm
1) Create a set superSet (shortest path tree set) that keeps track of vertices included that in shortest path tree, i.e., whose shortest distance from source is calculated and kept aside. Initially, this set is null.
2) Assign a cost/ distance value to all edges in the input graph. Initialize all cost/ distance values as INFINITITY. Assign distance value as 0 for the source vertex so that it is always first.
3) While superSet doesn’t include all vertices.
….**a)** Pick a vertex that is not present in superSet and has shortest cost value.
….**b)** Include it to superSet.
….**c)** Update cost/distance value of all nearby vertices of the choosen one. To update

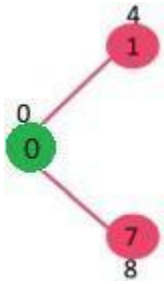
the distance values, loop through all adjacent vertices. For every nearby vertex v, if sum of distance value of u (from source) and weight of edge u-v, is smaller than the distance value of v, then update the cost value of v.

Let us understand the method by the following example:

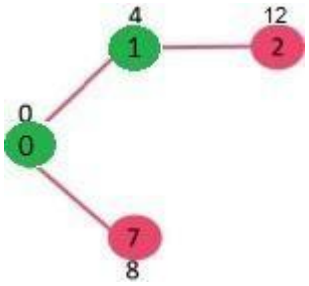


The set *superSet* is initially vacant and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinity. Now pick the vertex with shortest distance value. The vertex 0 is picked, include it in superSet. So superSet becomes {0}. After including 0 to superSet update cost values of its nearby vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

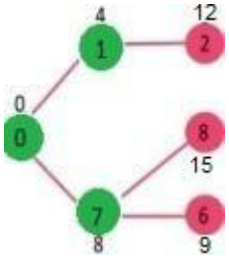
Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



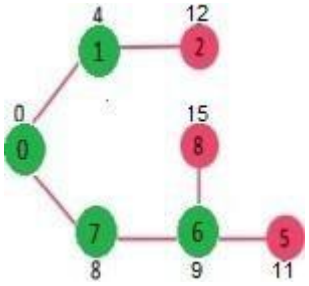
Now choose the vertex with shortest distance value and not already included in SPT (not in superSet). The vertex 1 is choose and added to superSet. So superSet is now {0, 1}. Update the cost values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



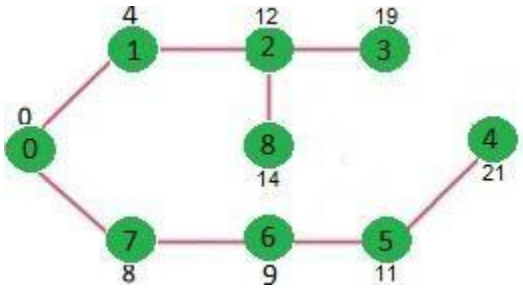
Choose the vertex with shortest distance value again and do the same



Then repeat the same again:



Finally, we get the following Shortest path tree.

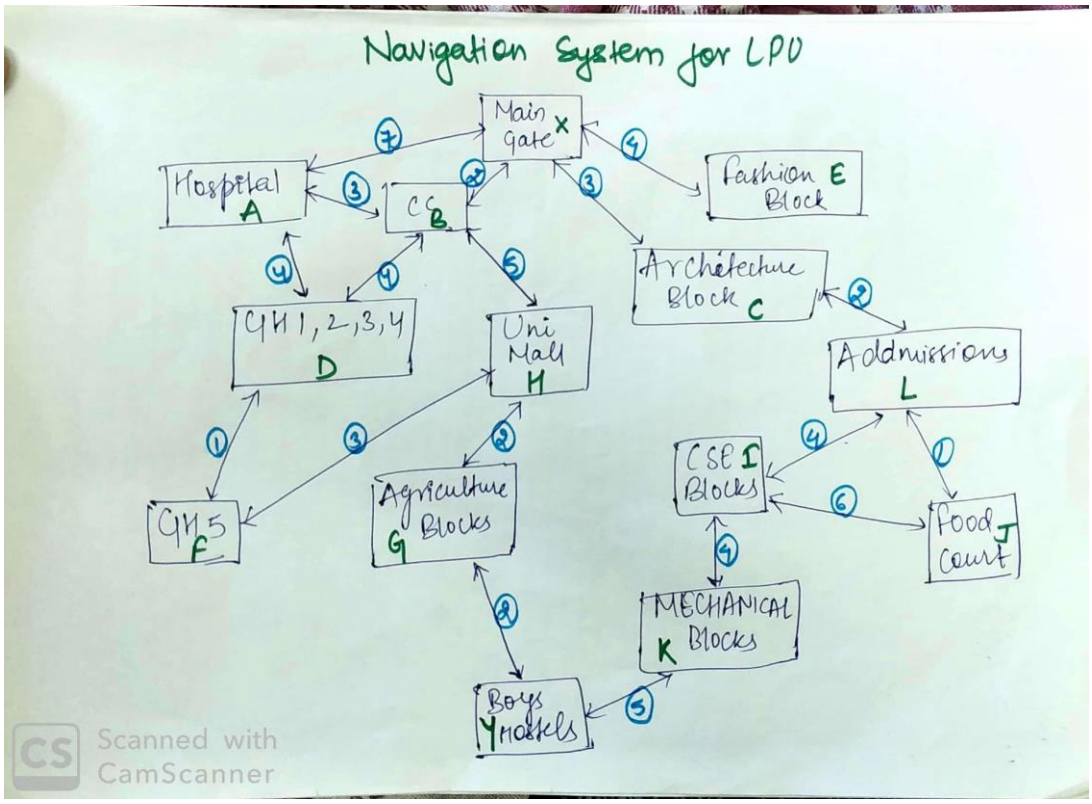


Objectives:-

Navigation systems that we use now-a-days have a lot variety of useful features and give drivers an upper hand . For example, most devices let us to locate restaurants, railway stations, petrol pumps, temples, hotels, and various attractions on our way. It is comforting to know that if you are running low on fuel or need to find a room for the night; your GPS navigation system can lead you in the right direction.
Once your stop is complete, the system will make sure we have our way back to our preferred route. Consumers can keep data stored concerning previously used routes and addresses in their GPS units. This convenience saves drivers time and effort when trying to located directions to places they have visited in the past.
GPS navigation systems offer a variety of benefits to drivers who travel long distances as well as those who stay close to home.

The purpose of a navigation system is to help everyone who uses it to find their way from a starting point to an ending destination. To utilize the system, drivers need to provide start point and also the point where they would like to go, which is destination. The system uses shortest path algorithm and determine to quickest route to your final destination.Then the shortest directions are provided to the user to go through it. Eliminate the fear of getting lost or the trouble of having to stop for directions along the way. If a driver should happen to make an incorrect turn, the navigation system can recalculate the directions to get the driver back on the correct route. It also eliminates the need for using maps or the Internet to plan routes before the start of a trip.

Methodology:-



This is the map and notations I am using for My LPU Navigation system.
The notations and distance is the same.

Notations are as follows:
A- Main Gate
B- CC
C- Architecture block
D- GH-1,2,3,4
E- Fashion Block
F- GH-5
G- Agriculture Block
H- Uni-Mall
I- CSE Block
J- Food Court
K- Mechanical Block
L- Admissions
X- Main Gate
Y- BH

Distance between them is as follows:-

('X', 'A', 7),
('X', 'B', 2),
('X', 'C', 3),
('X', 'E', 4),
('A', 'B', 3),
('A', 'D', 4),
('B', 'D', 4),
('B', 'H', 5),
('C', 'L', 2),
('D', 'F', 1),
('F', 'H', 3),
('G', 'H', 2),
('G', 'Y', 2),
('I', 'J', 6),
('I', 'K', 4),
('I', 'L', 4),
('J', 'L', 1),
('K', 'Y', 5)

Now apply Djikstra's algorithm:-

MY CODE:

In [1]:

```
from collections import defaultdict
class Graph():
```

```
    def_init_(self):
        """ self.edges is a dict of all possible next nodes
e.g. {'X': ['A', 'B', 'C', 'E'], ...} self.weights has all the
weights between two nodes,     with the two nodes as a tuple as the
key e.g. {('X', 'A'): 7, ('X', 'B'): 2, ...} """
        self.edges = defaultdict(list)
        self.weights = {}

    def add_edge(self, from_node, to_node, weight):
        # Note: assumes edges are bi-directional
        self.edges[from_node].append(to_node)
        self.edges[to_node].append(from_node)
        self.weights[(from_node, to_node)] = weight
        self.weights[(to_node, from_node)] = weight
```

In [2]:

```
graph = Graph()
```

In [3]:

```
edges = [
    ('X', 'A', 7),
    ('X', 'B', 2),
    ('X', 'C', 3),
    ('X', 'E', 4),
    ('A', 'B', 3),
    ('A', 'D', 4),
    ('B', 'D', 4),
    ('B', 'H', 5),
    ('C', 'L', 2),
    ('D', 'F', 1),
    ('F', 'H', 3),
    ('G', 'H', 2),
    ('G', 'Y', 2),
    ('I', 'J', 6),
    ('I', 'K', 4),
    ('I', 'L', 4),
    ('J', 'L', 1),
    ('K', 'Y', 5),]

for edge in edges:
    graph.add_edge(*edge)
```

Now we need to implement our algorithm.

At our starting node (X), we have the following choice:

- Visit A next at a cost of 7
- Visit B next at a cost of 2
- Visit C next at a cost of 3
- Visit E next at a cost of 4

We choose the lowest cost option, to visit node B at a cost of 2.
We then have the following options:

- Visit A from X at a cost of 7
- Visit A from B at a cost of (2 + 3) = 5
- Visit D from B at a cost of (2 + 4) = 6
- Visit H from B at a cost of (2 + 5) = 7
- Visit C from X at a cost of 3
- Visit E from X at a cost of 4

The next lowest cost item is visiting C from X, so we try that and then we are left with the above options, as well as:

- Visit L from C at a cost of (3 + 2) = 5

Next we would visit E from X as the next lowest cost is 4.

For each destination node that we visit, we note the possible next destinations and the total weight to visit that destination. If a destination is one we have seen before and the weight to visit is lower than it was previously, this new weight will take its place. For example

- Visiting A from X is a cost of 7
- But visiting A from X via B is a cost of 5
- Therefore we note that the shortest route to X is via B

We only need to keep a note of the previous destination node and the total weight to get there.

We continue evaluating until the destination node weight is the lowest total weight of all possible options.

In this trivial case it is easy to work out that the shortest path will be:

From main gate to boy's hostel:

X -> B -> H -> G -> Y

For a total weight of 11.

In this case, we will end up with a note of:

- The shortest path to Y being via G at a weight of 11

- The shortest path to G is via H at a weight of 9
- The shortest path to H is via B at weight of 7
- The shortest path to B is directly from X at weight of 2

And we can work backwards through this path to get all the nodes on the shortest path from X to Y.

Once we have reached our destination, we continue searching until all possible paths are greater than 11; at that point we are certain that the shortest path is 11.

In [4]:

```
def dijsktra(graph, initial, end):
    # shortest paths is a dict of nodes
    # whose value is a tuple of (previous node, weight)
    shortest_paths = {initial: (None, 0)}
    current_node = initial
    visited = set()

    while current_node != end:
        visited.add(current_node)
        destinations = graph.edges[current_node]
        weight_to_current_node = shortest_paths[current_node][1]

        for next_node in destinations:
            weight = graph.weights[(current_node, next_node)] +
weight_to_current_node
            if next_node not in shortest_paths:
                shortest_paths[next_node] = (current_node, weight)
            else:
                current_shortest_weight = shortest_paths[next_node][1]
                if current_shortest_weight > weight:
                    shortest_paths[next_node] = (current_node, weight)

        next_destinations = {node: shortest_paths[node] for node in
shortest_paths if node not in visited}
        if not next_destinations:
            return "Route Not Possible"
        # next node is the destination with the lowest weight
        current_node = min(next_destinations, key=lambda k:
next_destinations[k][1])

    # Work back through destinations in shortest path
    path = []
    while current_node is not None:
        path.append(current_node)
```

```
        next_node = shortest_paths[current_node][0]
        current_node = next_node
    # Reverse path
    path = path[::-1]
    return path
```

Results:-

In [5]:

```
dijsktra(graph, 'X', 'Y')
```

OUTPUT will be as follows:

Out[5]:

```
['X', 'B', 'H', 'G', 'Y']
```

So there we have it, confirmation that the shortest path from X to Y is:

X -> B -> H -> G -> Y

```
In [9]: dijsktra(graph, 'X', 'Y')
Out[9]: ['X', 'B', 'H', 'G', 'Y']
```

TRY For A -> K

```
In [10]: dijsktra(graph, 'A', 'K')
Out[10]: ['A', 'B', 'H', 'G', 'Y', 'K']
```

Testing:-

1. **X -> A**

```
In [11]: dijsktra(graph, 'X', 'A')
Out[11]: ['X', 'B', 'A']
```

2. **X -> D**

```
In [12]: dijsktra(graph, 'X', 'D')
Out[12]: ['X', 'B', 'D']
```

```
In [ ]:
```

3. **X -> H**

```
In [13]: dijsktra(graph, 'X', 'H')
Out[13]: ['X', 'B', 'H']
```

```
In [ ]:
```

4. **X -> L**

```
In [14]: dijsktra(graph, 'X', 'L')
Out[14]: ['X', 'C', 'L']
```

```
In [ ]:
```

5. **B ->I**

```
In [15]: dijsktra(graph, 'B', 'I')
Out[15]: ['B', 'X', 'C', 'L', 'I']
```

```
In [ ]:
```

6. **G -> E**

```
In [16]: dijsktra(graph, 'G', 'E')
Out[16]: ['G', 'H', 'B', 'X', 'E']

In [ ]:
```

7. K -> D

```
In [17]: dijsktra(graph, 'K', 'D')
Out[17]: ['K', 'Y', 'G', 'H', 'F', 'D']

In [ ]:
```

8. D -> J

```
In [18]: dijsktra(graph, 'D', 'J')
Out[18]: ['D', 'B', 'X', 'C', 'L', 'J']

In [ ]:
```

9. J -> J

```
In [19]: dijsktra(graph, 'J', 'J')
Out[19]: ['J']
```

10. F -> Y

```
In [20]: dijsktra(graph, 'F', 'Y')
Out[20]: ['F', 'H', 'G', 'Y']
```

Conclusion:-

Some ancient ways to navigate where:
1. Most people navigated using stars and the sun
2.Sticks that showed currents of water.
3.Vikings knew about North America about 400 years before Cristopher Columbus.
4.Early navigation was probably a big mistake in the seas and oceans.

But now we have devices to navigate us through though times, its easy and in our palms.. In our simplest gadgets.
Technology is for sure making our life simpler.

Future Scope:-

Practical Navigation and more GUI (Graphical User interface) mode can be added to UMS or LPU touch app.
So Navigation in our university campus is pretty helpful, specially for freshers and their parents who are unaware about the best ways to reach particular blocks.

More specific things can be added by seniors and make it a professionally polished AI program in LPU.

The government has been talking about this for quite some time and GPS has been successfully adopted in some areas or institutions. This is a an opportunity. We are consistently engaging with all the leading transport companies and related government institutions across India.