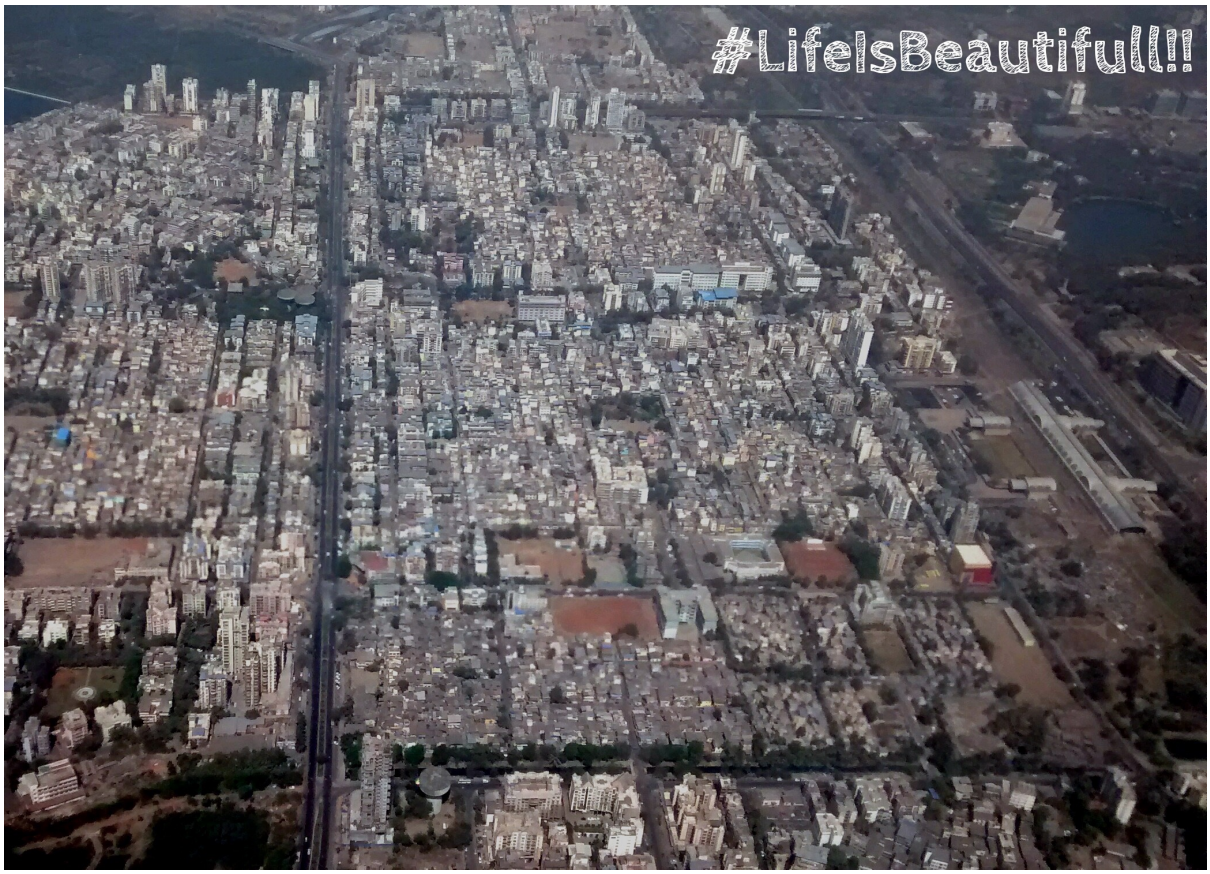
Traffic Simulation



Data Structures Mini Project

Sumedh Arani, Tarun Nayani, Vidyasagar G

3 Sem

Traffic Simulation

Introduction

In our project, we've made an attempt to simulate the traffic situation and eventually help the user by finding him the shortest path to his destination and also evaluate the fastest path to his destination.

As easy it may look from the outside, we've tried to implement a dynamically changing attributes of the graph. To emulate a traffic environment wherein the traffic on each road keeps changing in real time, we've randomised all the traffic weights of each edge on after every traversal. So the user can get a real time shortest path after he traverses every node. The user may or may not follow the shortest path, so in case the user diverts from the path we've shown to him, no troubles at all!! We find the shortest and fastest path for you back again.

Project Statement

To calculate the shortest path in a weighted graph whose weights keep changing in a induced dynamic environment and also the source from which the shortest path is to be calculated keeps changing as per the whims and fancies of the user using the program.

Project Description

To emulate a basic maps model, we've used several data structures like

- Adjacency lists to represent graphs
- Arrays
- Heaps
- Priority Queue

with tweaks here and there to accomplish the ultimate purpose of this project. The user inputs his source and destination through a command line interface o which the destination remains innocuous and the source can be inputted by the user as he wishes.

After this program calculates the shortest path and the fastest path using two weights assigned to each edge of which one keeps changing dynamically after the user traverses at least one edge using Dijkstra's algorithm implemented using a priority queue resulting in improved efficiency over a one implemented using queues.

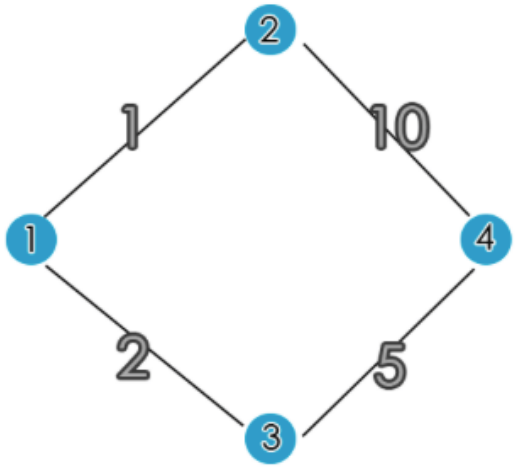
A beautiful interface has been written for creating graphics for this project. This interface written by one of the team member provides an easy way to implement any graphic visuals representing in accordance with the graph written using adjacency lists. It uses a buffer file to process the data dynamically and it flawlessly works in cohesion with the back end implementation to achieve the desired results. This eliminates any latency that may have occurred due to different running time of the front end and the back end application. The back end program writes onto a buffer file which is then further read and represented with visuals.

This concept is essentially how Google maps provides you directions and we run our program with essentially having two separate weights to calculate the shortest and fastest path. The two attributes we've used in our design is distance

which remains constant and other is the traffic mode on the basis of which the speed is calculated. If the traffic is high, the denominator increases and hence the speed decreases which is then selected as the weight for the graph to find shortest path. There are many thousands of vertices and edges, and when you ask for directions you typically want the shortest or least expensive route to and from your destinations.

Dijkstra's Algorithm

Dijkstra's algorithm is used to find the shortest path from an origin vertex to a destination vertex of a weighted graph. We've implemented it using a priority queue to improve the overall efficiency of our program.



For the purposes of this post, the blue circles represent "nodes" or "vertices" and the black lines are "edges" or "node paths". Each edge has a cost associated with it. For -this- image, the number in each node in this image is simply a label for the node, not the individual node cost.

Our problem is to find the most cost efficient route from Node1 to Node4. The numbers on the node paths represent the "cost" of going between nodes. The

shortest path from Node1 to Node4 is to take Node1 to Node3 to Node4, as that is the path where the least cost is incurred.

Specifically, the cost to go from Node1 to Node3 is (2), plus the cost of Node3 to Node4 (5) is 7 (2 + 5).
Now, we can see that the alternative (Node1 to Node2 to Node4) is much more costly (it costs 11, versus our 7).
An important note - greedy algorithms aren't really effective here. A greedy algorithm would basically find the cheapest local costs as it traverses the graph with the hopes that it would be globally optimum when it's done. Meaning, a greedy algorithm would basically just take the first low value it sees. In this case, the lower value is 1 but the next value is 10. If we were to simply just apply a greedy algorithm, we end up taking the more costly from Node1 to Node4. Figuring out the best path to take with this graph is pretty easy for us to do mentally, as if you can add small numbers you can figure out the best path to take. For a small graph like the previous, it's quite easy.

Dijkstra's algorithm is an algorithm that will determine the best route to take, given a number of vertices (nodes) and edges (node paths). So, if we have a graph, if we follow Dijkstra's algorithm we can efficiently figure out the shortest route no matter how large the graph is.

Dijkstra's algorithm provides for us the shortest path from NodeA to NodeB.

How it works

First we'll describe Dijkstra's algorithm in a few steps, and then expound on them further:

Step 0.

Temporarily assign C(A) = 0 and C(x) = infinity for all other x.

C(A) means the Cost of A

C(x) means the current cost of getting to node x

Step 1.

Find the node x with the smallest temporary value of c(x).

If there are no temporary nodes or if c(x) = infinity, then stop.

Node x is now labeled as permanent. Node x is now labeled as the current node.

C(x) and parent of x will not change again.

Step 2.

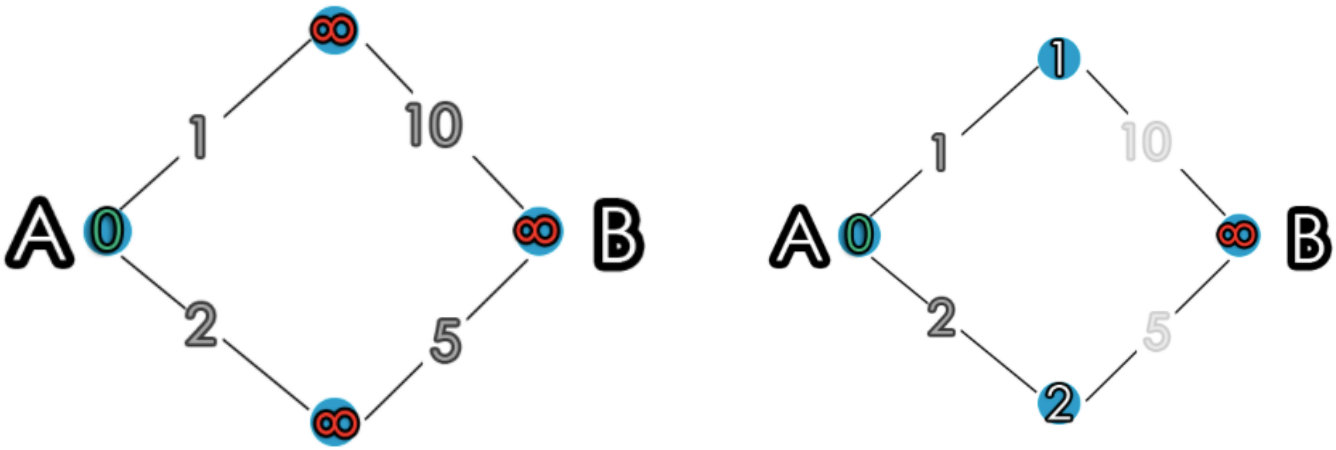
For each temporary node labeled vertex y adjacent to x, make the following comparison:

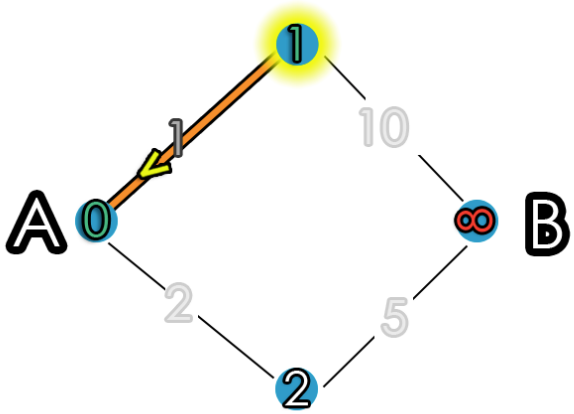
if c(x) + Wxy < c(y), then c(y) is changed to c(x) + Wxy

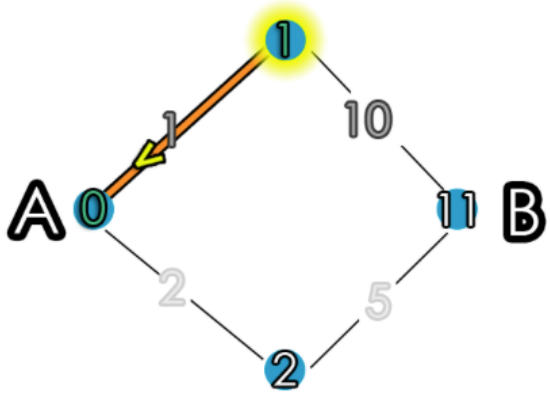
assign y to have parent x

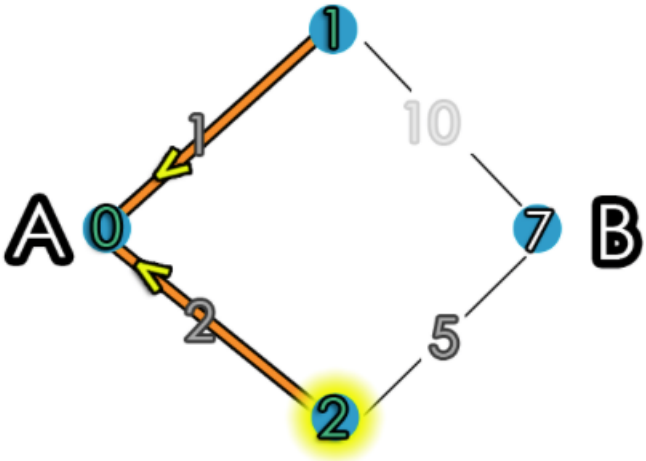
Step 3.

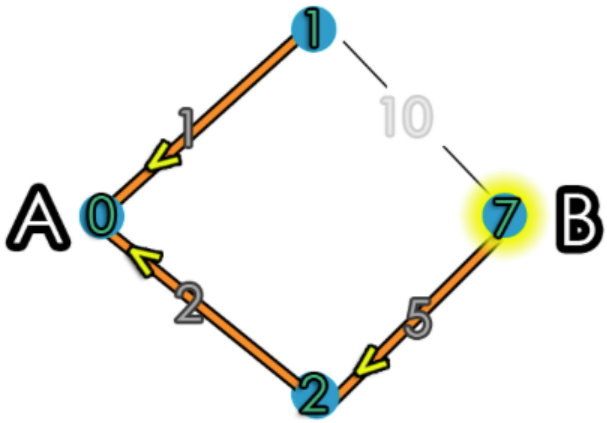
Return to step 1.











Efficiency

- Extract Minimum using priority queue: O(log V)
- To build the minimum heap: O(V)
- Total running time: O(E log V) i.e all vertices are reachable from the source

Front End

The front-end and the back-end are two different programs which are bridged by a file. Data from the back-end is dumped into the file. The file buffers all the data which comes from the back-end so that the front end accesses the file with ease and smooth file operations.

The idea behind keeping the back-end as a separate program is to make its development simpler, independent and less buggy.

Front end uses the “awt” package for all the drawings on the canvas. Front end is a generic program which draws any object (that’s specified in the function that reads the file) on the canvas.

For traffic simulation, lines, circles and characters (only) are needed to display the graph/map data on the canvas. Thus, the read file function supports only line, circles and character drawing.

About “mapGraphics.java”

Main loop:

An infinite while loop which reads the buffered data from the file again and again and searches for new data/ new object to be drawn. It also maintains file line number of the previously read line.

Text file:

It contains data about the object and its attributes.

For eg:

LINE_020_030_100_200_RED

The above line will draw a line between (20,30) and (100,200) in red.

Object codes:

1. CHAR_*_x_y_COLOR // will print the specified char
2. RECT_x1_y1_x2_y2_COLOR // will draw a rectangle
3. CIRL_x_y_radius_COLOR // circle
4. LINE_x1_y1_x2_y2_COLOR // line

Where all the number must be in 3 digit format.

Rectangular coordinate system:
Computer graphics usually has its origin at the top left corner of the screen. The program automatically converts the rectangular coordinates system data to computer coordinates system data.

COLOR	CODES
Blue	BLU
Black	BLK
Red	RED
Yellow	YLW
Green	GRN
Cyan	CYN
Dark Gray	DGR
Gray	GRY
Light Gray	LGR
Magenta	MGT
Orange	ORG
Pink	PNK
White	WHT

Color codes:

Default path color: LIGHT_GRAY

Shortest path: MAGENTA

Fastest path: GREEN

Low traffic: GRAY

Moderate traffic: YELLOW

Heavy traffic: RED

Nodes: Black

Vehicle: BLUE

About interfaces in “trafficSimulation.java” (back-end)

All the functions take the data and creates an encoded string which contains data at specific locations which gives various attributes to the front-end program.

Function “circle”:

Prototype: public static void circle(int x, int y, int r, String col);

Parameters “x” and “y” specifies the location of the circle.

Parameter “r” specifies radius of the circle.

Parameter “col” specifies color for the fill.

Function “line”:

Prototype: public static void line(int x1, int y1, int x2, int y2, String col);

Parameters “x1” & “y1” specifies the start of the line.

Parameters “x2” & “y2” specifies the end location of the line.

Parameter “col” specifies the color.

(Line function creates many parallel line to the actual line for extra visibility)

Function “character”:
Prototype: public static void character(char ch_dash, int x, int y, String col);
Parameter “ch” contains the character to be printed on the canvas.
Parameter “x” & “y” specifies the location of the character on the canvas.
Parameter “col” specifies the color of the character to be displayed on the canvas.

All the function described above (circle, line, character) encodes the object data in the string. Then this string is appended into the file using file operations.

Thank You