Representing Maps with Graphs

Data Structures 3460:316
05/06/2016

Tyler Harbert (tgh10)

Abstract - The objective of this project is to use a directed, weighted Graph to represent a road map and use Dijkstra's algorithm to find the shortest path between two locations. The edges of the Graph will use the distance between vertices, the speed limit and traffic information to determine the edge weight. Overall this research proves that a directed, weighted graph is an effective method to determine the shortest path on a road map.

I. INTRODUCTION

The idea of this project is to show how a directional, weighted graph can be used to find the shortest path between two locations on a map. This system should be able to accept the distance between points, the speed limit between points, and traffic information in order to accurately tell a user the shortest path to their destination. The reason that a directional graph is the appropriate structure for this problem is because traffic can often vary in different directions so this can give the most accurate shortest path estimate to a user. An example of this would be if there is an accident southbound on a highway, then users traveling northbound would not want this to be taken into account when determining their shortest path. This could also affect a single user if they intend to return to their original starting point after reaching their original destination then they may take a different path on both trips.

The data structure used to realize the directional, weighted graph is an adjacency list. The reason that this was chosen was because the graph of road connections is very sparse, so this data structure will have a much better data consumption than an adjacency matrix. There is an in-depth analysis of these space savings in the 'Space Complexity' section of part II of this research. This list will use inheritance to form weighted edges from unweighted edges and to form a graph from an unweighted graph. This would allow for future applications to use the unweighted versions as well as the weighted versions.

The algorithm chosen to locate the shortest path between two points on the map is a modified version of Dijkstra's Algorithm. This algorithm works in the same way by updating the weights of all adjacent vertices and always visits the unvisited vertex with the least weight next. One difference with this implementation is that the algorithm terminates after the specified destination vertex is visited instead of visiting all other vertices. Another difference is that tracks the path from the starting node to the destination so that the exact path can be returned to the user.

The hypothesis for this project is that a directional graph can be used to accurately locate the shortest path on a map by using the distance, the speed limit, and traffic information.

II. WORKING PROCEDURE(S) AND COMPLEXITY

Working Procedures [1]

path.hpp:

- *Path* : A class that is used to track the path from the source to all visited vertices. When Dijkstra's Algorithm is complete this path can be traversed backwards to realize the shortest path to the destination.
    - *add* (*source, destination*): Sets the value of the destination index in the array to the source index.
    - *get (destination)*: Starting at the destination index in the array, this visits the index saved in the array until a -1 is found. These are pushed into a vector, reversed, and returned.

node.hpp:

- *Node* : Base class for vertices and edges. Contains a *Node* pointer to the next node.
- *Vertex* : Inherits from *Node*.
- *uEdge* : Inherits from *Node* and represents an unweighted edge. Contains an integer member that represents its vertex index
- *Edge* : Inherits from *uEdge*. Contains a template type member which represents the weight of an edge.

uadjlist.hpp:

- *uGraph* : Represents an unweighted graph. Contains an integer member to hold its size and a *Vertex* pointer array to represent all vertices.
    - *add_edge (vertex1, vertex2, directional = true)*: Adds an unweighted edge from vertex1 to vertex2. If directional is specified as false then an unweighted edge is also added from vertex2 to vertex1.

adjlist.hpp:

- *Graph* : Inherits from *uGraph* and represents a weighted graph. This class accepts a template type to represent edge weights.

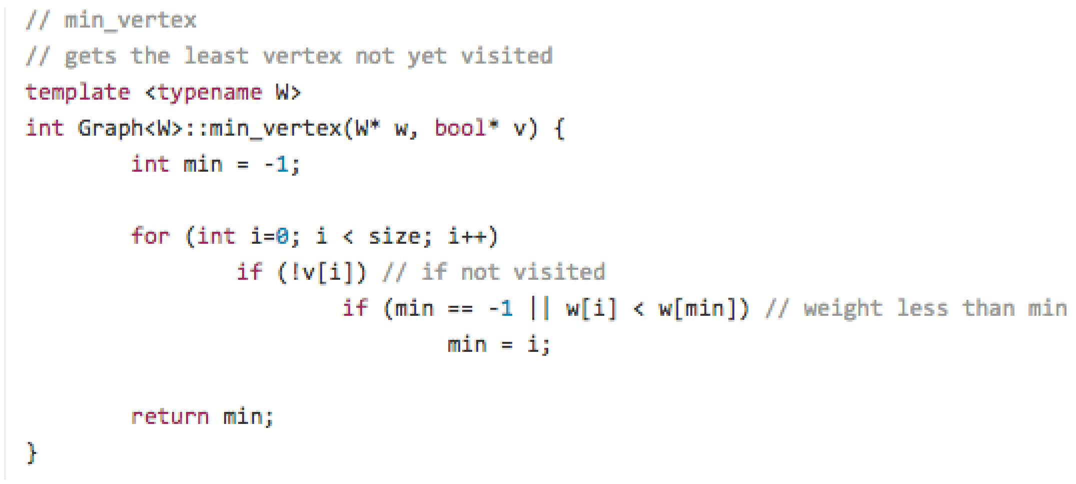
- *add_edge (vertex1, vertex2, weight, directional = true)*: Like *uGraph*, this function adds either one or two edges depending on whether directional is true or false. In addition to this functionality, a template typed weight is added to these edges.

- *dijkstra(source, destination)*: Uses Dijkstra's algorithm to find the shortest path from the source to the destination vertex. This function uses a *Path* object to track the path between vertices. After the shortest path is found this is traversed backwards and the shortest path is returned in a standard library vector [2].

- *min_vertex(visited[], weight[])*: This is a private member function used to determine the unvisited node with the lowest weight.

Time Complexity

This implementation of Dijkstra's Algorithm uses loops to locate the unvisited vertex with the least weight. The algorithm that accomplishes this is shown in *Figure 1* below.



```
// min_vertex
// gets the least vertex not yet visited
template <typename W>
int Graph<W>::min_vertex(W* w, bool* v) {
        int min = -1;

        for (int i=0; i < size; i++)
                if (!v[i]) // if not visited
                        if (min == -1 || w[i] < w[min]) // weight less than min
                                min = i;

        return min;
}
```

Figure 1: Algorithm to find the unvisited vertex with the least weight [1]

Shaffer explains in his textbook that this will have a worst case time of O(V^2) because this scan is run V times. Other implementations can use a minheap to locate the vertex with the least weight and reduce this time to O(V log(V)) [3].

Space Complexity

Because most graphs representing a road map are sparse we chose to use an adjacency list to represent this structure as opposed to an adjacency matrix. As an example we will compare a nondirectional graph with 100 vertices and 1000 edges using each method. The operating system is 64-bit so pointers are 8 bytes, the weight if represented by a double is 8 bytes, and the vertex index is represented by an integer of 4 bytes.

$$S = V * ptr + E\,(w + ptr + vi)$$

Equation 1: Space complexity of adjacency list

$$S = V^2 * w$$

Equation 2: Space complexity of adjacency matrix

Using the equations we find that a graph of the size specified would create a 80,000 byte adjacency matrix and a 20,800 byte adjacency list. This arbitrary example shows that much data is wasted using the adjacency matrix if the graph is sparse.

III. EXPERIMENT RESULTS AND MEASURES

In order to test Dijkstra's Algorithm the first task was to implement one of the Open DSA examples to insure that the correct path is found. *Figure 2* below contains a directional graph that searches the shortest path from vertex A to all other vertices using Dijkstra's Algorithm.

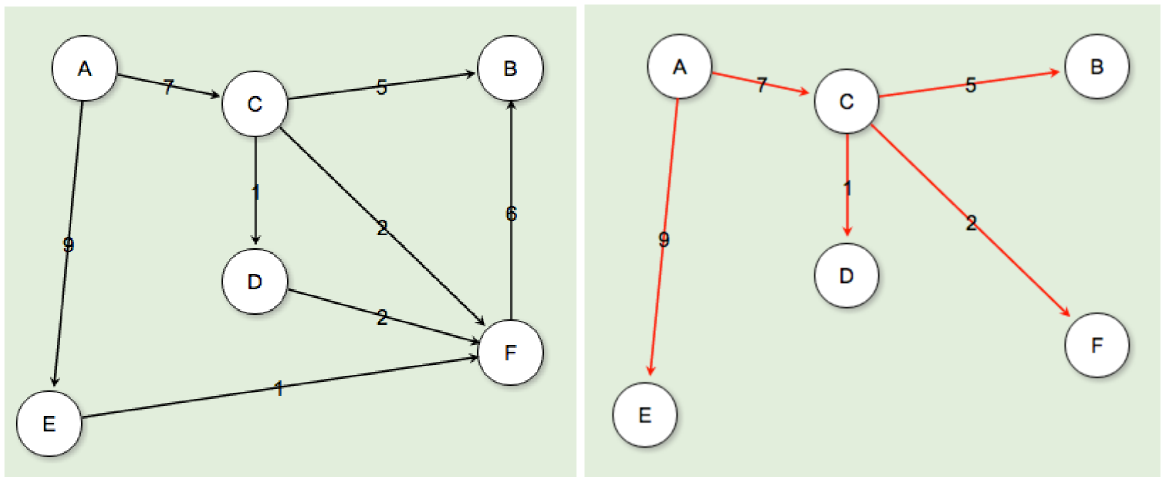
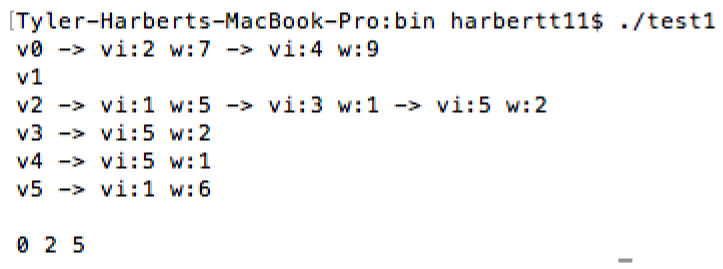


Figure 2: A directional weighted graph before and after dijkstra's algorithm

In order to test this the same graph is created using a *Graph* with a template weight of type int. The nodes A - F are represented by the values 0 - 5 respectively. The result in *Figure 3* is achieved by running the first test with 0 (A) as the source and 5 (F) as the destination.



```
[Tyler-Harberts-MacBook-Pro:bin harbertt11$ ./test1
v0 -> vi:2 w:7 -> vi:4 w:9
v1
v2 -> vi:1 w:5 -> vi:3 w:1 -> vi:5 w:2
v3 -> vi:5 w:2
v4 -> vi:5 w:1
v5 -> vi:1 w:6

0 2 5
```

Figure 3: The print out of the adjacency list and the shortest path from v0 to v5

Once the structure and algorithm were verified to be working properly a hypothetical map and scenario was built. The problem that is being shown is what is the fastest way to go from our data structures class in the Arts and Sciences building to Chipotle on Exchange Street. The map in *Figure 4* shows the streets around campus and the distances between vertices have been determined using Google Maps measurement feature [4]. The blue lines represent non directional connections, which are represented by two directional connections, where an estimated speed limit is included near each distance measurement.

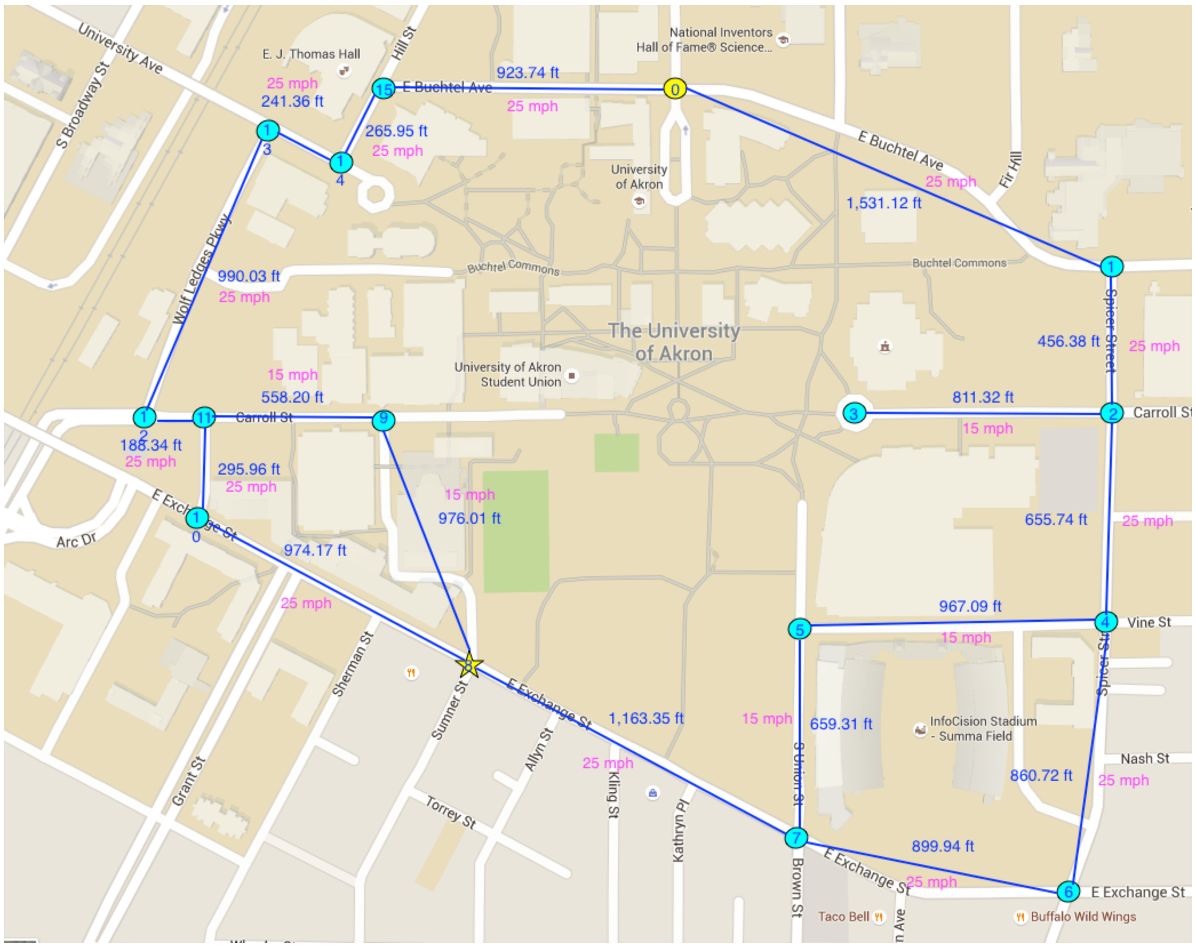
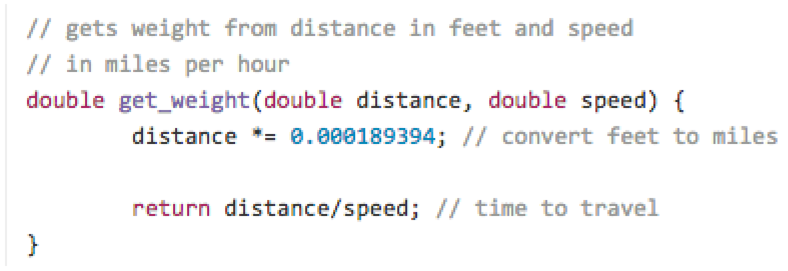


Figure 4: A map around The University of Akron with distance measurements in feet

and estimated speed limits in miles per hour

In order to determine the weight of a connection the function in *Figure 5* is used to determine the time that it would take to travel each path.

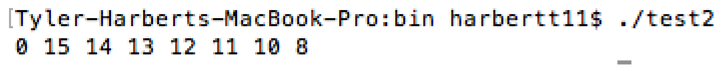


```
// gets weight from distance in feet and speed
// in miles per hour
double get_weight(double distance, double speed) {
        distance *= 0.000189394; // convert feet to miles

        return distance/speed; // time to travel
}
```

Figure 5: The time it takes to travel a distance in feet at the specified speed limit

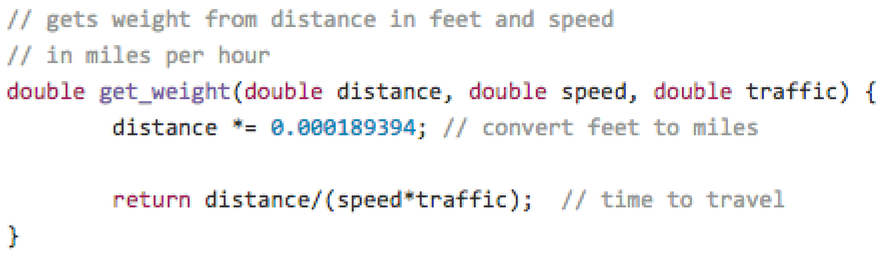
The result of running Dijkstra's Algorithm with the values specified in the image from vertex 0 at the Arts and Sciences building to vertex 8 at Chipotle is shown in *Figure 6* below. It is clear from this image the fastest path is to travel west from the starting position and then south to Exchange Street.



```
[Tyler-Harberts-MacBook-Pro:bin harbertt11$ ./test2
 0 15 14 13 12 11 10 8
```

Figure 6: Result on running Dijkstra's Algorithm on vertices 0-to-8 in the graph in *Figure 4*

The next step in the process was to develop a test in which traffic information in a specific direction reflects the shortest path. The first step was to modify the get_weight function to accept traffic information. This will be a value between 0 and 1 that represents a percentage of the speed limit that can be achieved due to traffic. The modified function is included in *Figure 7* below.



```
// gets weight from distance in feet and speed
// in miles per hour
double get_weight(double distance, double speed, double traffic) {
        distance *= 0.000189394; // convert feet to miles

        return distance/(speed*traffic);  // time to travel
}
```

Figure 7: The time it takes to travel a distance in feet at a percentage of the specified speed limit

After updating this function the map is updated for a hypothetical scenario. If a concert at E. J. Thomas Hall were to let out at the time someone desired to go to Chipotle we can make up a factor of 25% of the effective speed limit will be achievable on the surrounding roads leading away from the theatre. These same roads will not have this delay while heading in the direction of the theatre. *Figure 8* shows this added traffic metric in red added to the original map.

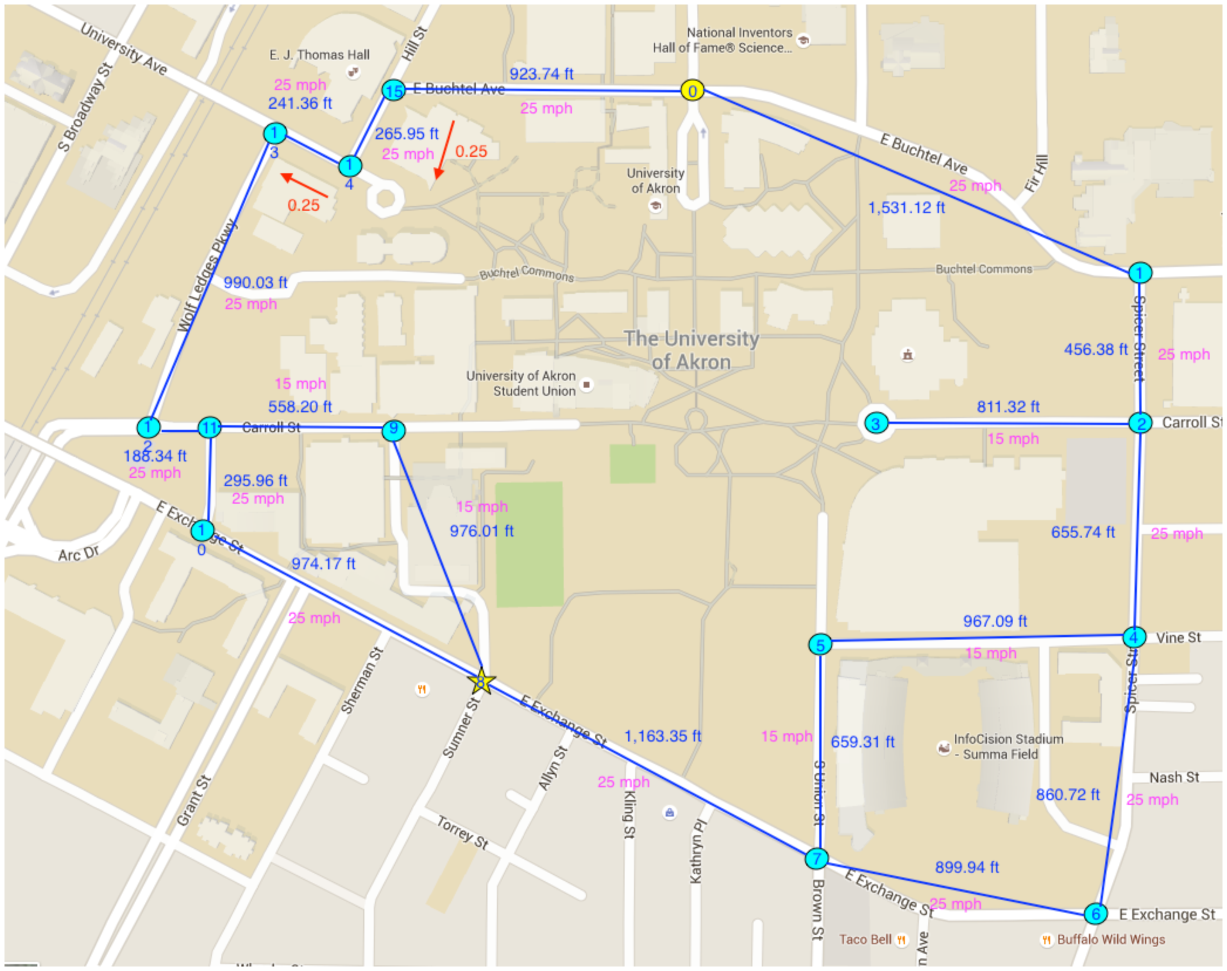
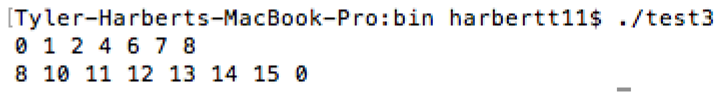


Figure 8: The map from *Figure 4* with an added slowdown by E. J. Thomas Hall

The output of running Dijkstra's Algorithm again from vertex 0 to vertex 8 and then from vertex 8 back to vertex 0 is shown in *Figure 9* below.



```
Tyler-Harberts-MacBook-Pro:bin harbertt11$ ./test3
 0 1 2 4 6 7 8
 8 10 11 12 13 14 15 0
```

Figure 9: Running Dijkstra's algorithm on vertices 0-to-8 and 8-to-0 on the Graph in *Figure 8*

The first output decided that the shortest path to Chipotle was now on the east side of campus because traffic made the weight on the west side of campus larger. Additionally, the algorithm still knew that the shortest path back to the Arts and Sciences building was on the west side of campus because the traffic was only traveling southbound.

IV. CONCLUSION

Overall the hypothesis that a directed, weighted graph can be used to find the shortest path on a road map using distance, speed limits and traffic information was found to be true. Finding the shortest path on an example map was done with Dijkstra's Algorithm and it was proven to adjust paths due to traffic in either direction. A future implementation of this system would benefit from using a minheap to find unvisited vertex with the least weight because it reduces the time complexity from $O(V^2)$ to $O(V \log(V))$. Also, a way that a GPS could adjust for traffic in a real world scenario is it could use the time in between GPS pings of several users to find what effective amount of the speed limit is being achieved on different roads and use that to adjust weights accordingly.

V. REFERENCE

[1] T. Harbert, ‘Project 5’, 2016. [Online]. Available: https://github.com/tyharbert/DataStructures/tree/master/Project%205. [Accessed: 8 May 2016].

[2] ODSA Project Contributors, ‘Shortest-Paths Problems’, 2016. [Online]. Available: https://canvas.instructure.com/courses/995719/assignments/4287854. [Accessed: 8 May 2016].

[3] C. A. Shaffer. “Single-Source Shortest Paths” in *Data Structures and Algorithm Analysis*, Update 3.2.0.6. Clifford A. Shaffer, 2009-2012, pp. 400 - 402

[4] maps.google.com, 2016. [Online]. Available: https://www.google.com/maps/@41.0745693,-81.5118198,15z. [Accessed: 9 May 2016]