Sebastian Arango

Kyle Sturmer

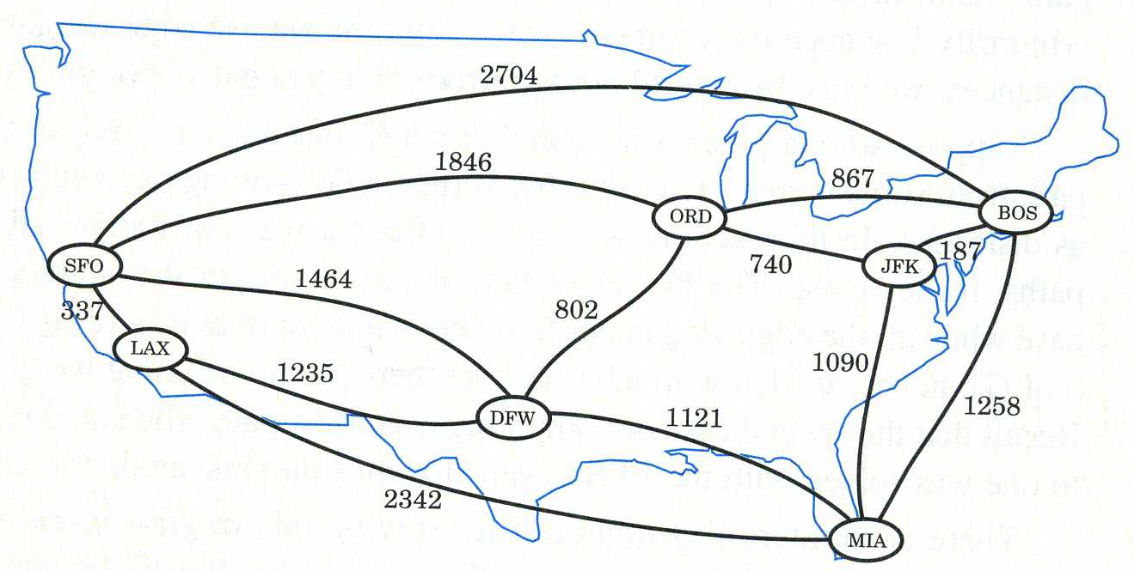
12/14/15

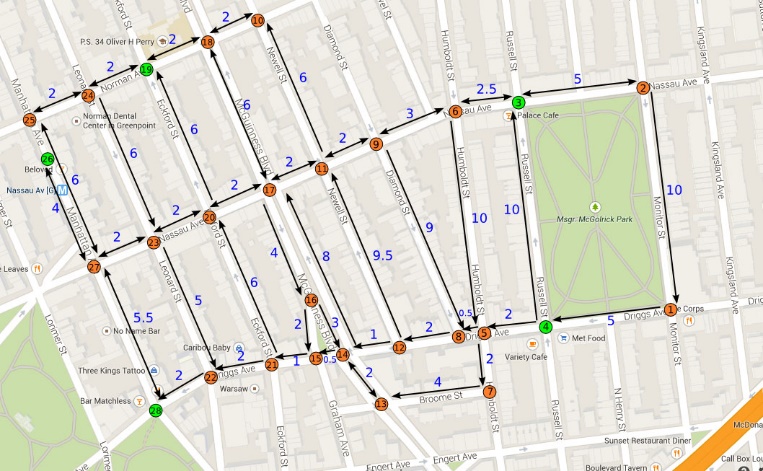
Professor Zlatareva

**Dijkstra’s Algorithm**

Our application will find the shortest distance from a fixed starting point to all major airports by using Dijkstra’s algorithm. We chose a graph of travel by commercial airplanes. We wanted to figure out what would be the shortest distance for a plane to go from San Francisco International Airport to a select few major airports. For example, would it take less miles from SFO airport to MIA airport if you were to stop at LAX or DFW airport first? For commercial airline companies, it is crucial to save as much money as you can. By using Dijkstra’s algorithm it will find the shortest path from SFO any other major airports, ultimately saving them money by using less gas.

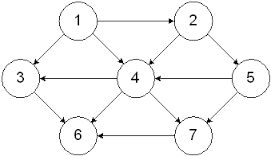
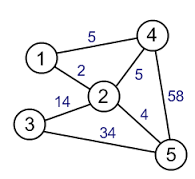
For our tests, we made SFO the starting vertex. In all the cases where there was a direct path from San Francisco Airport to another airport, that path was found to have the lowest weight, meaning it would always be better to go straight there than to stop at another airport if you wanted to save as much gas as possible. If the flight were to go to MIA airport, stopping at DFW would be the most cost efficient route because traveling to LAX would take more miles, meaning more money for the company to spend on gas. If the flight were to go to JFK, stopping at ORD would be the best path.



Another implementations for this algorithm would be a GPS. Someone traveling through a city who does not know the best way to travel might want to use a GPS run by Dijkstra’s algorithm because it also takes into consideration if the roads are bidirectional or directional. Some roads, such as in the picture, can only be traveled on in one direction, so using Dijkstra’s algorithm will implement these facts. As stated early, Dijkstra’s algorithm is very useful because it can be ran on a graph with different types of edges. The algorithm will convert the map of the city into a graph so that a shortest path can be found based on a person’s current position and an end destination.

One last great implementation of Dijkstra’s algorithm would be OSPF, which is used as protocols aimed at traffic moving around large system networking. The algorithm finds the shortest path between routers to be able to exchange information for the best path to a network of given destination. The fewer routers have to work, the less money is needed to be spent to exchange information.

Dijkstra’s algorithm is a graph search algorithm that will find the shortest path between nodes in a graph. The computer scientist Edsger Dijkstra formulated Dijkstra’s Algorithm in 1956. Dijkstra’s algorithm is known as a single-source shortest path (SSSP). A SSSP algorithm calculates the length of the shortest path from the source to each of the remaining nodes in the graph. Dijkstra’s algorithm runs on a weighted graph and with nonnegative weights. The algorithm exists in many different variations and implementations. The original algorithm that Dijkstra created found the shortest path between two nodes. However, the most common implementation finds the shortest path from the initial node to each node in the graph.



A graph will have a few main components which are vertices and edges. A graph will have multiple vertices that contain data within the graph. The vertices are connected to each other by edges. Edges can have either have a weight or no weight value at all, which means that an edge could be defined by an arbitrary value or have no value at all. A graph with weighted edges is defined as a weighted graph (on right) and an unweighted graph (left) has no values assigned to its edges. Graph on the left can also be considered directed, because the vertices’ edges can only travel in the direction that the arrow is pointing. The graph on the right is considered undirected, because the edges connecting the vertices are bidirectional, or do not specify the direction that you can traverse the graph.

|  |
| --- |
| Psuedo code |
|  | **for each** vertex v in Graph: | // Initialization |
|  | dist[v] := infinity | // initial distance from source to vertex v is set to infinite |
|  | previous[v] := undefined | // Previous node in optimal path from source |
|  | dist[source] := 0 | // Distance from source to source |
|  | Q := the set of all nodes in Graph | // all nodes in the graph are unoptimized - thus are in Q |
|  | **while** Q **is not** empty: | // main loop |
|  | u := node in Q with smallest dist[ ] |  |
|  | remove u from Q |  |
|  | **for each** neighbor v of u: | // where v has not yet been removed from Q. |
|  | alt := dist[u] + dist\_between(u, v) |  |
|  | **if** alt < dist[v] | // Relax (u,v) |
|  | dist[v] := alt |  |
|  | previous[v] := u |  |
|  | **return** previous[ ] |  |

For Dijkstra’s algorithm to work, you must set the value of the starting node equal to zero and the weight to every other node to infinity. While there are still edges that have not been calculated, find the distances between all of the vertices. The distance to each vertex will be updated, so by the time all of the distances have been calculated there should be no more vertices that are set to infinity. The shortest path from point *a* to *b* will be found by using the relaxation method. The relaxation method will be called numerous times to compare weights.

If D[u] + w((u,z)) < D[z] then

D[z] <- D[u] + w((u,z))

In this example, v is the starting vertex. If the distance from v to u plus the weight from u to z is less than the direct weight from v to z, then the distance to z will not be the shortest edge path of D[z], but will become the distance from v to u plus the weight of u to z. The relaxation method got its name from an example using a spring. When you stretch out a string as far as you can, that is equivalent to setting the distance from the starting vertex to all others vertices to infinity initially. When you find a shorter path from two vertices, you relax the string, or decrease its size.

The simplest implementation of Dijkstra’s algorithm would be to use an ordinary linked list or array, for storing the set of unvisited vertices. To find the adjacent vertex with the minimum distance you would use a simple linear search through the set of all the unvisited vertices. For sparse graphs, Dijkstra’s algorithm can be implemented more efficiently by storing the graph into an adjacent list, and using a min-priority queue to find the minimum distance vertices.

As previously stated, for sparse graphs, Dijkstra’s algorithm can be implemented more efficiently, by storing the graph into an adjacency list. This is because an adjacency list has many advantages that Dijkstra’s algorithm can benefit from. The advantage of using an adjacency list for Dijkstra’s algorithm is that it’s easy to find nodes that are adjacent to each other. Therefore, theoretically an adjacency list should work best, since from every vertex, the algorithm scans all its neighbors. Other advantages are that it has a fast iteration over all edges and an adjacency list uses memory in proportion to the number of vertices, which will save memory compared to an adjacency matrix. In addition, to using an adjacency list best way to find the lowest cost path to an adjacent node would be to use a priority queue.

Using a priority queue will be more efficient than using a normal traversal to find the next lowest cost path. There are many different ways of implementing a priority queue; using a self-balancing binary search tree, binary heap, pairing heap, or a Fibonacci heap for the priority queue. The binary heap will keep track of which vertex has the lowest distance. With a binary heap implementation to the priority queue the time complexity will be

with E & V as the number of edges and vertices in the graph respectively. This algorithm has a different time complexity from graph search algorithm because the binary heap adds and removes elements in time, whereas graph algorithm’s data structures add and remove elements in constant time. Now with the Fibonacci heap implemented to a priority queue, it improves the efficiency of the algorithm to run at ). Implementing Dijkstra’s algorithm with a Fibonacci heap is the fastest implementation known. The main disadvantage that Dijkstra’s algorithm has is that it will fail in cases where the edges are negative numbers. Another weakness is that Dijkstra’s algorithm does a blind search causes the algorithm to consume a lot of time and waste of resources. Overall, the advantage of this algorithm is that it’s an efficient algorithm and with a fast run time of ) when using a min-priority queue with a Fibonacci heap.

|  |
| --- |
| public class Dijkstra  {  public static int [] dijkstra (WeightedGraph G, int s)  {  final int [] dist = new int [G.size()];  final int [] pred = new int [G.size()];  final boolean [] visited = new boolean [G.size()];    for (int i=0; i < dist.length; i++)  {  dist[i] = Integer.MAX\_VALUE;  }  dist[s] = 0;  for (int i=0; i < dist.length; i++)  {  final int next = minVertex(dist, visited);  visited[next] = true;    final int [] n = G.neighbors(next);  for (int j=0; j < n.length; j++) {  final int v = n[j];  final int d = dist[next] + G.getWeight(next,v);  if (dist[v] > d)  {  dist[v] = d;  pred[v] = next;  }  }  }  return pred; // (ignore pred[s] == 0!)  }  private static int minVertex (int [] dist, boolean [] v)  {  int x = Integer.MAX\_VALUE;  int y = -1; // graph not connected, or no unvisited vertices  for (int i=0; i < dist.length; i++)  {  if (!v[i] && dist[i]<x)  {  y=i;  x=dist[i];  }  }  return y;  }  } |

|  |
| --- |
| import java.util.Scanner;  public class WeightedGraph  {  private int [][] edges; // adjacency matrix  private Object [] labels;    public WeightedGraph (int n)  {  edges = new int [n][n];  labels = new Object[n];  }  public int size() { return labels.length; }  public void setLabel (int vertex, Object label)  {  labels[vertex] = label;  }    public Object getLabel(int vertex)  { return labels[vertex]; }    public void addEdge(int source, int target, int w)  { edges[source][target] = w; }    public boolean isEdge(int source, int target)  { return edges[source][target] > 0; }    public void removeEdge(int source, int target)  { edges[source][target] = 0; }    public int getWeight(int source, int target)  { return edges[source][target]; }  public int [] neighbors(int vertex)  {  int count = 0;  for (int i=0; i < edges[vertex].length; i++)  {  if (edges[vertex][i] > 0) count++;  }    final int[]answer= new int[count];  count = 0;    for (int i=0; i < edges[vertex].length; i++)  {  if (edges[vertex][i] > 0) answer[count++] = i;  }    return answer;  }  public void print ()  {  for (int j=0; j < edges.length; j++)  {  System.out.print(labels[j] + ": ");  for (int i=0; i < edges[j].length; i++) {  if (edges[j][i] > 0) System.out.print(labels[i] + ":" + edges[j][i] + " ");  }  System.out.println ();  }  }  public static void main (String args[])  {  Scanner scan = new Scanner(System.in);  int e;  final WeightedGraph t = new WeightedGraph (7);  t.setLabel (0, "SFO");  t.setLabel (1, "ORD");  t.setLabel (2, "DFW");  t.setLabel (3, "LAX");  t.setLabel (4, "MIA");  t.setLabel (5, "JFK");  t.setLabel (6, "BOS");    t.addEdge (0,6, 2704); // SFO to...  t.addEdge (0,1, 1846);  t.addEdge (0,2, 1464);  t.addEdge (0,3, 337);    t.addEdge (1,6, 867); // ORD to...  t.addEdge (1,5, 740);    t.addEdge (5,6, 187); // JFK to..    t.addEdge (3,2, 1235); //LAX to..  t.addEdge (3,4, 2342);    t.addEdge (2,1, 802); // DFW to..  t.addEdge (2,4, 1121);    t.addEdge (4,6, 1258); // MIA to  t.addEdge (4,5, 1090);    System.out.println();  System.out.println("(1)ORD, (2)DFW, (3)LAX, (4)MIA, (5)JFK, (6)BOS ");  System.out.print("Enter number corresponding to destination: ");  e = scan.nextInt();    final int [] pred = Dijkstra.dijkstra (t, 0);  System.out.print("The Shortest path to your destination will be: ");  Dijkstra.printPath (t, pred, 0, e);    }  } |

In the Dijkstra class will contain the algorithm, with will find the shortest path from the initial node to all other nodes. The algorithm’s parameters will be a *WeightedGraph* object and an integer *s* that will be the source vertex from which the algorithm will initially start at. This algorithm will be using several arrays to solve the shortest path problem. The arrays are

(DIST): will be the distance array that will store the shortest known distance from the initial node.

(PRED): will be the path array; in the end will show the shortest path from the initial node

(VISITED): the visited array; shows the set of nodes that have been visited. All nodes will be set initially to false.

The minVertex method has the parameters of the distance array and the visited array; this method will return the index value of the node that has been visited. First the algorithm will set all of the tentative distance values to the nodes as infinity. Then the for-loop will find the first adjacent node with the shortest path from the initial node, by using the path array. Then the algorithm will go into the nested for loop with is the relaxation method is used to check if there is another path to a node that has a lower cost, and it will update the path and distance arrays. If not, then it will continue the algorithm.

The algorithm will be using a matrix in the *WeightedGraph* class to represent the graph. In this class the objects are label to make it easier to set each node a name and weighted edges between two nodes. Then in the main method of the *WeightedGraph* class we have inputted the data corresponding to the major airports and their distance of their edges. It will ask the user to input their destination they would like to go from SFO airport and will display the shortest path.

In our application, we used arrays for our algorithm a nested for loop which makes our application run at ; compared to using a priority used to help reduced time. In addition, we used an adjacency matrix to represent our graph which is not very memory efficient compared to an adjacency list. Overall, Dijkstra’s algorithm is a very popular and efficient algorithm that finds the shortest path/distance from the initial node to every node in a graph and is still used for a vast amount of applications today.

Work Sited

* http://www.gitta.info/Accessibiliti/en/html/Dijkstra\_learningObject1.html
* http://searchenterprisewan.techtarget.com/definition/OSPF
* Michael T. Goodrich and Roberto Tomassia “Data Structures and Algorithms in JAVA”, John Wiley & Sons, Inc. (fourth edition)
* https://www.cs.auckland.ac.nz/software/AlgAnim/dijkstra.html
* http://www.cs.nyu.edu/courses/summer07/G22.2340-001/Presentations/Puthuparampil.pdf
* http://cs.fit.edu/~ryan/java/programs/graph/WeightedGraph-java.html
* http://cs.fit.edu/~ryan/java/programs/graph/Dijkstra-java.html