Jason James

Professor Eyles

Data Structures Research Project

11/26/18

Dijkstra's Algorithm

There are many modern algorithms that drive the technology and decisions of our evolving world, today. One of the types of these many algorithms are "Shortest Path Finding Algorithms". There are even a great number of shortest-path-finding algorithms. One of the more well-known shortest-path-finding algorithms is "Dijkstra's Shortest-Path-Finding Algorithm". Dijkstra's Algorithm is an algorithm that will take a set of nodes or locations and determine the shortest path from a point of origin to a final destination. The algorithm was founded by Edsger W. Dijkstra in 1956 and has since then been used in many practical applications.

To demonstrate how Dijkstra's Algorithm works, I have included a C++ Project which includes the functionality and execution of Dijkstra's Shortest Path Algorithm. The program consists of two classes.

The first class is the Node class (Figure 1). The Node class contains 4 field variables, which is the nodeName, numOfEdges, dDistance, and *dPrevPathNode. The nodeName variable is a character value that stores the character label of the node. The numOfEdges variable is an integer value which stores the number of edges touching a node. The dDistance variable is an integer variable that contains the placeholder/temporary value of the distance from the start node,

which is changed and utilized in the method that calculates the Dijkstra's Algorithm. The *dPrevPathNode is a Node pointer value which is used to point at the previous node which is also cleared, altered, and utilized during the execution of Dijkstra's Algorithm. The Node class also consists of a constructor, with a single character label argument, and getter and setter methods for all four field variables.

Figure 1 - Node Class Header File

```
class Node
{
public:
    Node(char label);
    char getName(); //Returns nodeName
    int getEdgeCount(); //Returns numOfEdges
    int getDijkstraDistance(); //Returns dDistance
    Node* getPreviousNode(); //Returns dPrevPathNode
    void setEdgeCount(int edgeCount); //Sets EdgeCount
    void setDijkstraDistance(int distance); //Sets dDistance
    void setPreviousNode(Node *dPrevPathNode); //Sets dPrevPathNode


private:
    char nodeName; //Label name for the node
    int numOfEdges; //Number of edges touching a node
    int dDistance; //The placeholder distance value that is modified via Dijkstra's Algorithm
    Node *dPrevPathNode; //The placeholder value for shortest previous node modified in Dijkstra's Algorithm
};
```

The second class is the Edge class (Figure 2). The Edge class contains 3 field variables, which is *point1, *point2, and weight. The *point1 variable is a Node pointer value which points to a Node object that the Edge object is connected to. The *point2 variable is also a Node pointer value which serves the same purpose as *point1 and has no importance or precedence over or under *point1. The weight variable is an integer value that stores the numerical edge weight value of the Edge object. This weight value can be used as either the distance, time, cost, etc. between two nodes. The Node class contains a constructor that consists of three arguments, which are the three field variables (*point1, *point2, and weight), and getter and setter methods for all three of these field variables.

Figure 2 - Edge Class Header File

```
class Edge
{

private:
    Node *point1; //connected point 1
    Node *point2; //connected point 2
    int weight; //The numerical edge weight value

public:
    Edge(Node *point1, Node *point2, int weight); //Node Constructor
    int getWeight(); //Returns weight
    void setWeight(int weight); //Sets weight
    Node *getPoint1(); //Gets point1
    void setPoint1(Node *point1); //Sets point1
    Node *getPoint2(); //Gets point2
    void setPoint2(Node *point2); //Sets point2
};
```

The primary driver program/file in this project is called “main.cpp”. The driver program begins with setting up the graph demo that will be referred to for the duration of this report. The demo graph for this report will consist of 6 nodes and will be connected by 7 edges. The nodes and edges will be created manually (no random graph generation). This process can be seen below in Figure 3 and Figure 4.

Figure 3 - Creating graph

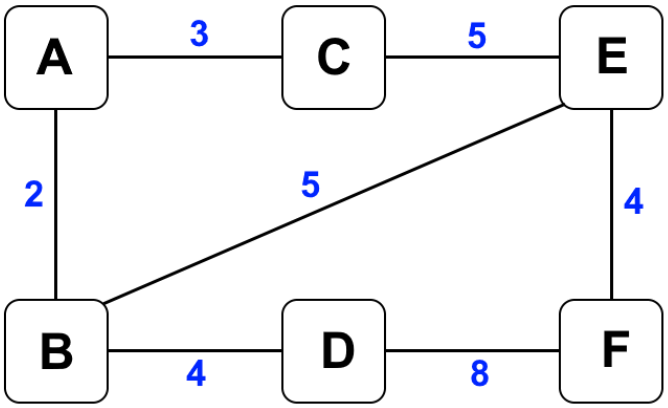
```
const int nodeCount = 6; //Can be used for various arguments
const int edgeCount = 7; //Can be used for various arguments

////////////////////////////////////////
////////// Creating the graph //////////
////////////////////////////////////////

//Manually create nodes
Node *A = new Node('A');
Node *B = new Node('B');
Node *C = new Node('C');
Node *D = new Node('D');
Node *E = new Node('E');
Node *F = new Node('F');

//Manually create edges
Edge *edgeAB = new Edge(A, B, 2);
Edge *edgeAC = new Edge(A, C, 3);
Edge *edgeBD = new Edge(B, D, 4);
Edge *edgeBE = new Edge(B, E, 5);
Edge *edgeCE = new Edge(C, E, 5);
Edge *edgeDF = new Edge(D, F, 8);
Edge *edgeEF = new Edge(E, F, 4);
```

Figure 4 - Visual Representation of Graph



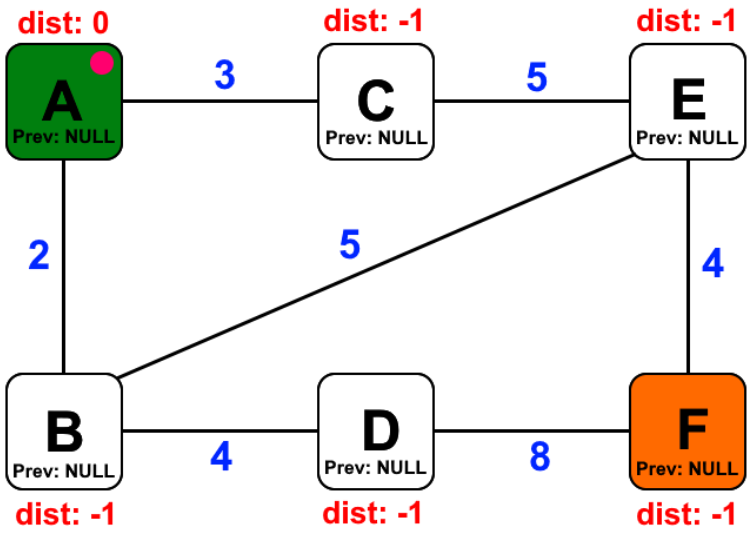
The code from Figure 3 will setup all the necessary data for a graph that can be utilized by Dijkstra's Algorithm. This graph data is then ready to be "packaged" into a Node and Edge pointer array to be sent to the findDijkstrasShortestPath method as an argument. The findDijkstrasShortestPath method requires 6 arguments, which is *nodeList[], *edgeList[], *startNode, *endNode, nodeCount, and edgeCount. The *nodeList[] and *edgeList[] variables, as previously mentioned, are the node and edge pointer arrays which compose the demo graph. The *startNode and *endNode variables are the origin and final locations in the shortest path that is trying to be calculated. The nodeCount and edgeCount are both integer values that represent the size of the *nodeList[] and *edgeList[] arrays, primarily for later iterating through these array at multiples times.

The beginning of the findDijkstrasShortestPath method begins with initializing the dijkstraDistance and previousNode values of all nodes back to their intended default values by iterating through all nodes in nodeList via for loop. This is done in the event that the method has been used at least once before during execution. At this point all nodes should have "-1" as their

dijkstraDistance and should have null as their previousNode values. The reason why the dijkstraDistance values must equal "-1" is because in theory all values' distances are assumed to be positive infinity until they can be determined, individually. Since there is no real way of representing infinity in C++, "-1" is used as a marker to indicate that it should be assumed that it is infinite distance and the node has node been reached yet.

A new Node pointer value, named *currentNode, is initialized at the address of *startNode and the dijkstraDistance for this value is set to zero. This is done to keep track of which node on the graph needs to look for its adjacent/connected nodes. Also, since the currentNode at this time is the startNode, it should be noted that the distance from itself is zero. After this a new Node pointer array, named visitedNodes, of size nodeCount, is initialized with no values. An integer, named visitedNodesCount is also initialized at zero, which will be used to keep track of the used size of the visitedNodes array. At this point the graph should visually look as shown in Figure 5, below. Note: the pink circle indicates that the node is the currentNode.

Figure 5 - Graph Initialized for Dijkstra's Algorithm



Upon setting up the graph to a default position, the first phase of the Dijkstra's Algorithm is ready to begin. The first phase, is the phase that will obtain and update the dijkstraDistance and previousNode values of every node in the nodeList[] array. To do this, the program enters a while loop that loops until the visitedNodesCount equals the nodeCount, or in other words: until every node has been marked as visited. For each iteration of this while loop it will loop through every node in nodeList and see if it is connected, and then if it is not a visited node and is not the same as the currentNode. If all of these conditions are met, it will then loop through all edges in edgeList[] and try to find an edge that contains both the node within the for loop and the currentNode. If the "loop edge's" weight is lower than an integer, called leastWeight, or if leastWeight is still at its initialized value of "-2" (infinity), it will then store a variable of that Edge object pointer and weight of that edge in the variable leastWeight. After the "edgeList[] loop" has finished, the program will see whether or not the node within the "node loop" has a previousNode, or in other words: not equal to null, and if true, will set the previousNode of the "loop node" to the currentNode. The program will then see if the "loop node's" dijkstraDistance is still listed at "-1", and if true, it will set the dijkstraDistance of the node to the currentNode's dijkstraDistance plus the weight from the stored "loop edge". This will then conclude the "node loop". The currentNode will then be marked as a visited node, by adding it to the visitedNodes[] array and increase the visitedNodesCount by 1. The "Phase 1" portion of the algorithm will then conclude by assigning the next currentNode, by finding a node that is not visited and does not have a dijkstraDistance value of "-1". Below, Figures 6 and 7 show the states of the graph when currentNode = A and F, respectively. Note that red "X" marks visited nodes.

Figure 6 - Graph at currentNode = A

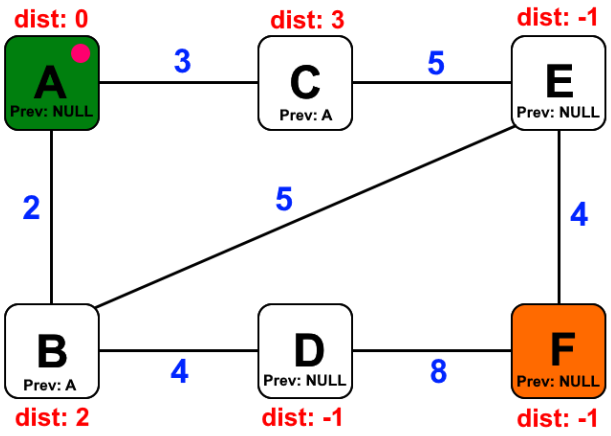
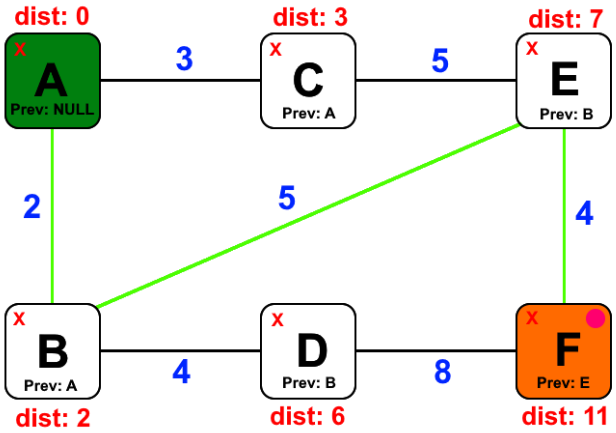


Figure 7 - Graph at currentNode = F



Technically, Dijkstra's Algorithm is officially completed after "Phase 1", because Dijkstra's Algorithm, technically finds the shortest path from the startNode to all nodes in the connected graph. However, in most practical situations most people do not want this. They will instead want just the shortest path between the startNode and a specified node, such as endNode. To do this we enter "Phase 2" which utilizes the previousNode values by creating a new Node pointer, called "*reverse", which is initialized at *endNode. Then the program must assess how many edges there are in between the startNode and the endNode, by looping from endNode to its previousNode, until it reaches the startNode, all while increasing an integer size variable. Once

the size is determined, an array that will store the shortest path between the nodes in order will be created. For the array to obtain all of these nodes in order, a similar for loop to the one that obtain the size will be created which will store the endNode at the “size -1” position and will again go “backwards” while placing the nodes until it reaches index = zero. At this point a for loop can be made that prints the shortest path between the two points as seen in Figure 8, below.

Figure 8 - Final Output

```
Shortest Path from A to F:  A B E F

::Node Data::
A::: Edges: 2 :: dijkstraDistance: 0
B::: Edges: 3 :: dijkstraDistance: 2
C::: Edges: 2 :: dijkstraDistance: 3
D::: Edges: 2 :: dijkstraDistance: 6
E::: Edges: 3 :: dijkstraDistance: 7
F::: Edges: 2 :: dijkstraDistance: 11
```

Despite the fact that Dijkstra’s Shortest Path Algorithm can be incredibly useful, it does have a fundamental flaw. This flaw is that Dijkstra’s Shortest Path Algorithm does not have a sense of direction. As stated my Dr. Mike Pound (University of Nottingham), in the example of a travel time weight between nodes on a road system, there could be a traffic jam on a motorway, which would increase the edge weights in that direction, and would direct someone’s GPS on the opposite direction of that motorway where there are lower edge weights. This is flawed “Because Dijkstra doesn’t build any idea of the direction you’re traveling, it’s going to look up the [lowest cost edges] first” (Pound & Haran). Or in other words, even though an individual wants to travel to his/her destination, which is East along a motorway, which contains a traffic jam, Dijkstra’s Algorithm will search for routes to the West along the motorway, until it figures out that the edge weights are no longer cheaper than going East through the traffic jam. This goes back to an earlier point that was made about how Dijkstra’s Algorithm’s primary goal is to find the shortest

path between a starting node and every node on a graph. Therefore, if Dijkstra's Algorithm is implemented in a shortest distance route finding system, it should use a hybrid of Dijkstra's Algorithm, as well as using some form of heuristic method that takes into account the general direction that the endNode is located, relative to the startNode and the currentNode.

As mentioned before, Dijkstra's Shortest Path Algorithm was founded in 1956, by Edsger W. Dijkstra, which was "developed in 20 minutes while Dijkstra was relaxing on a café terrace with his fiancée" (Richards). Dijkstra had a fond love of computers and wanted more people to share this by trying to reason why computers are so useful. Dijkstra accomplished this with his ARMAC computing machine, that he created, and "to show the power of the ARMAC, Dijkstra implemented one of his most famous algorithms, the shortest path algorithm" (Thissen).

Dijkstra's Shortest Path Algorithm has had a wide variety of actual practical applications, since its inception in 1956. One of these applications is road construction. In one study, Dijkstra's Shortest Path Algorithm was used to determine the cheapest road construction path through a forest in Iran, in which "Dijkstra's algorithm that consisted of nodes and links was used to optimize the road path in a broadleaved forest. The lower the cost, the greater the chance that the link will get routed" (Parsakhoo). Therefore, a thick area of the forest would have a higher edge weight, and a sparse clearing would have a lower edge weight. Dijkstra's Shortest Path Algorithm has also been commonly used in network routing protocols. Network routing protocols use shortest path algorithms, such as Dijkstra's Algorithm, to find the shortest path to send a pack from a transmitter to a receiver within the lowest amount of time. In this case, the routers are the nodes, and the physical connections are the edges. Within network routing protocols, Dijkstra's Algorithm is considered a Link-State Algorithm, which is a type of shortest-

path algorithm where “the network topology and all link costs are known” (Fink). Therefore, the nodes/routers do not need to store all adjacent node costs.

Dijkstra’s Shortest Path Algorithm is an incredibly flexible algorithm that has and will continue to be used in such a wide variety of applications. It has helped society with road cost, network routing, transportation cost, GPS routing, and much more. It gained the interest of the ARMAC machine’s capabilities to compute shortest paths between locations. Dijkstra’s Algorithm, while it does have a fundamental flaw, can be avoided by making slight modifications to adapt to the individual situation, and is a very intelligent and useful shortest-path algorithm.

Works Cited

Fink, James. "Network Routing." *Gettysburg College*, Gettysburg College, 2001, cs.gettysburg.edu/~jfink/courses/cs322slides/3-19.pdf. Accessed 26 Nov. 2018.

Parsakhoo, A., and M. Jajouzadeh. “Determining an Optimal Path for Forest Road Construction Using Dijkstra’s Algorithm.” Journal of Forest Science, vol. 62, no. 6, pp. 264–268.EBSCOhost, doi:10.17221/9/2016-JFS. Accessed 13 Oct. 2018. https://www.agriculturejournals.cz/publicFiles/9_2016-JFS.pdf

Pound, Mike, and Brady Haran. “Dijkstra's Algorithm - Computerphile.” *YouTube*, Computerphile, 4 Jan. 2017, www.youtube.com/watch?v=GazC3A4OQTE.

Richards, Hamilton. "Edsger W. Dijkstra." *A.M. Turing Award Laureate*, ACM, 2012, amturing.acm.org/award_winners/dijkstra_1053701.cfm. Accessed 26 Nov. 2018.

Thissen, W.P.C., et al. "ARMAC." *Unsung Heroes in Dutch Computing History*, 2007, www-set.win.tue.nl/UnsungHeroes/machines/armac.html. Accessed 26 Nov. 2018.