Graph Theory – Dijkstra's Algorithm

JANUARY 11, 2015MAY 18, 2015 / VAMSI SANGAM

Hello people…! In this post I will talk about one of the fastest single source shortest path algorithms, which is, the Dijkstra's Algorithm. The Dijkstra's Algorithm works on a weighted graph with non-negative edge weights and ultimately gives a Shortest Path Tree. It is a Greedy Algorithm, which sort of… mimics the working of Breadth First Search and Depth First Search. It is used in a number of day-to-day scenarios. It is used in network routing, to calculate the path from a network device A and B in a network which would have the maximum bandwidth. It could also be used by the GPS in a car to calculate the shortest path between two locations. The Dijkstra's Algorithm can be modified to solve a lot of real world problems. So let's get started…!

The Dijkstra's Algorithm starts with a source vertex '**s**' and explores the whole graph. We will use the following elements to compute the shortest paths –

- Priority Queue **Q**, implemented by a Min Binary Heap using C++ STL Vector.
- Another set **D**, which keeps the record of the shortest paths from starting vertex **s**. Implemented using C++ STL Vector.

Just like the other graph search algorithms, Dijkstra's Algorithm is best understood by listing out the algorithm in a step-by-step process –

- The Initialisation –

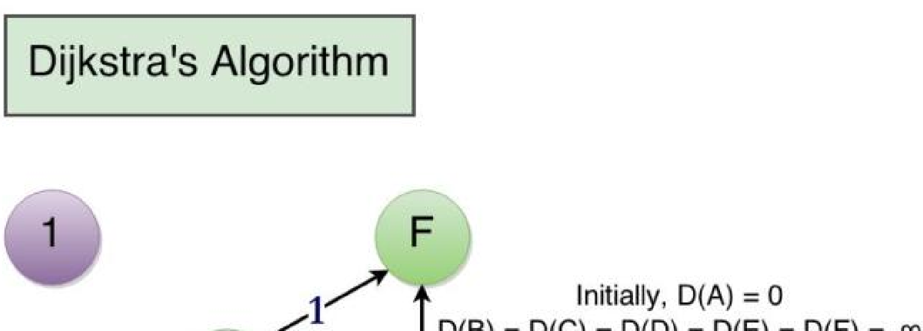
1. **D**(**s**), which is the shortest distance to **s** is set to 0. It is obvious as distance between source to itself is 0.
2. For all the other vertices **V**, **D**(**V**) is set to infinity as we do not have a path yet to them, so we simply say that the distance to them is infinity.
3. The Priority Queue **Q**, is constructed which is initially holds all the vertices of the Graph. Each vertex **V** will have the priority **D**(**V**).

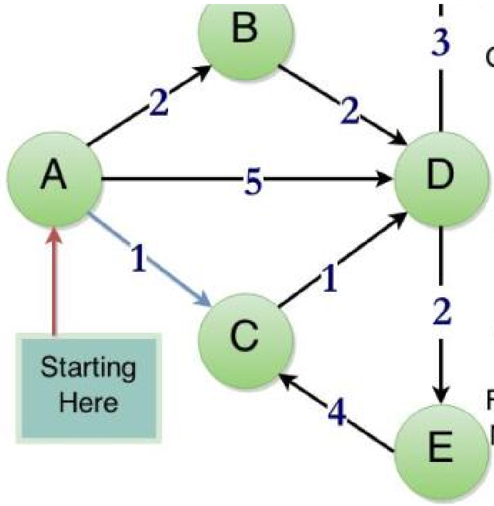
- The Algorithm –

1. Now, pick up the first (or the minimum) element from the Priority Queue **Q** (which removes it from **Q**). For the first time, this operation would obviously give **s**.
2. For all the vertices adjacent to **s**, i.e., for all vertices in **adjacencyMatrix**[**s**], check if the edge from **s** → **v** gives a shorter path. This is done by checking the following condition –
   if, **D**(**s**) + (weight of edge **s** → **v**) < **D**(**v**), we found a new shorter route, so update **D**(**v**)
   D(**v**) = **D**(**s**) + (weight of edge **s** → **v**)

3. Now pick the next element from **Q**, and repeat the process until there are elements left in **Q**.

It might look like a long and cumbersome process, but this is actually a very smart technique. It's okay if you don't understand it in the first reading. Give another 3-4 readings and try to picture what is happening to the graph when you implement the algorithm, in your head. After you feel you have got a hang of the algorithm, look at the sketch below for complete understanding.

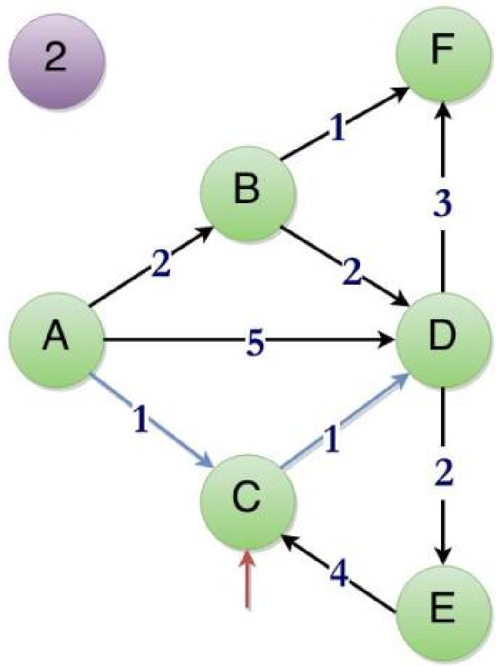




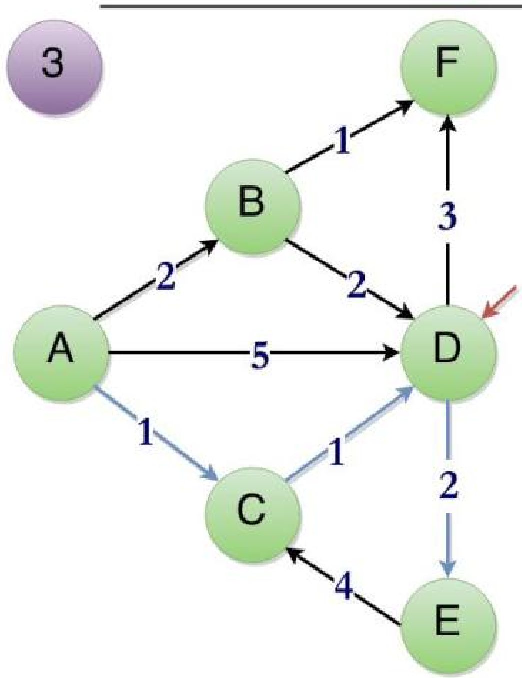
So, D = {0, ∞, ∞, ∞, ∞, ∞}
Q = {A(0), B(∞), C(∞), D(∞), E(∞), F(∞)}
The priorities in Q, are given in paranthesis

Now, extract the minimum element from Q.

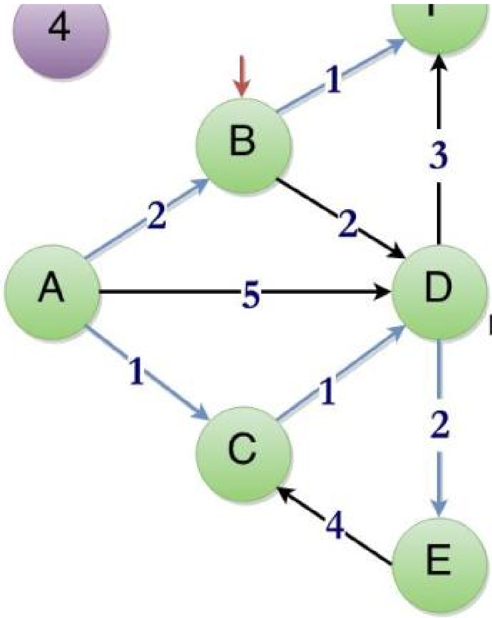
Q = {B(2), C(1), D(5), E(∞), F(∞)}
D = {0, 2, 1, 5, ∞, ∞}
Finished with A, extract-min from Q
Minimum element in Queue is C(1)



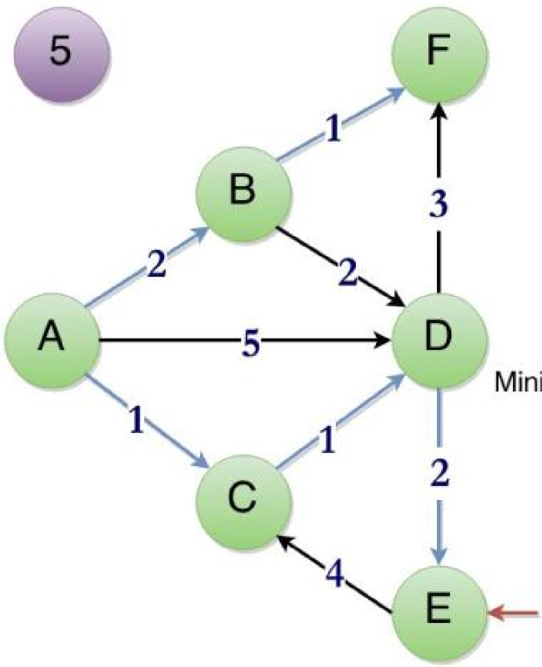
Q = {B(2), D(2), E(∞), F(∞)}
D = {0, 2, 1, 2, ∞, ∞}
Finished with C, extract-min from Q
In Q, B(2) and D(2) are minimum, anyone can be picked, we'll go for D(2).



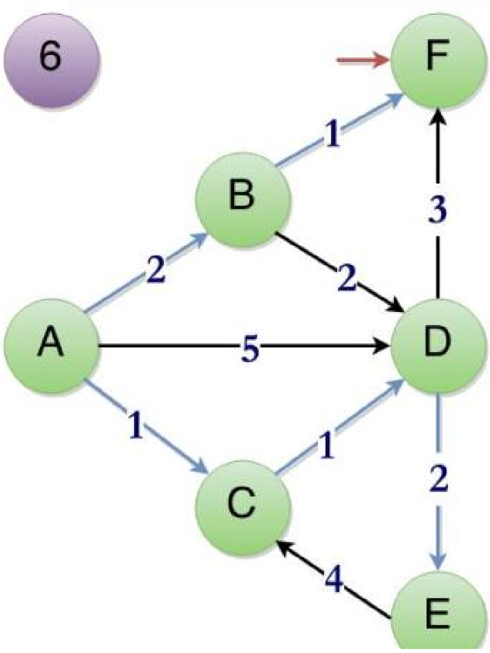
Q = {B(2), E(4), F(5)}
D = {0, 2, 1, 2, 4, 5}
Minimum Element in Queue is B(2)



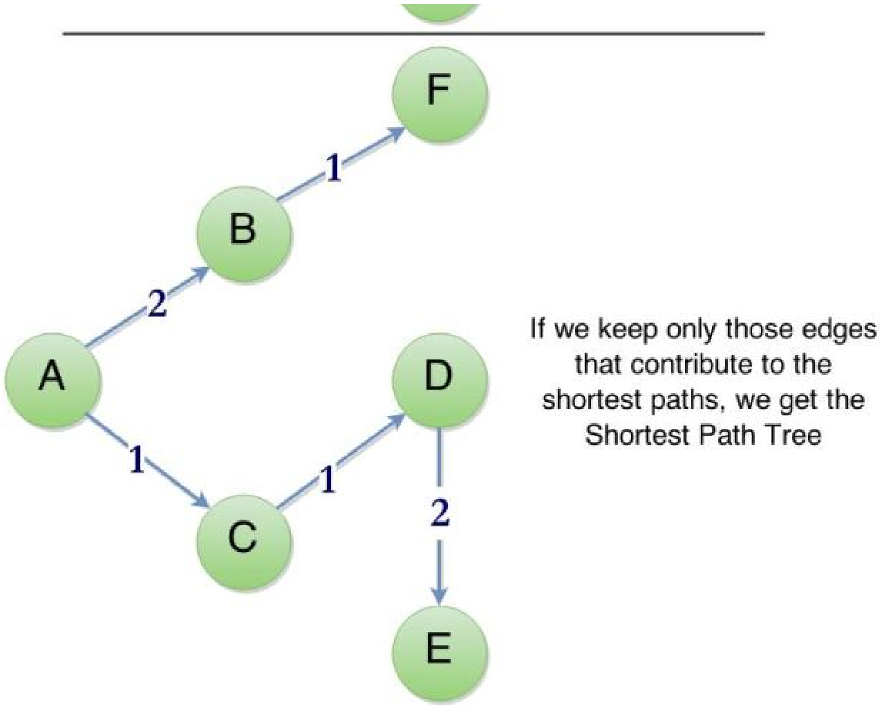
Q = {E(4), F(5)}
D = {0, 2, 1, 2, 4, 3}
Minimum Element in Queue is E(4)



Q = {F(5)}
D = {0, 2, 1, 2, 4, 3}
Minimum Element in Queue is F(5)



Q = {}
D = {0, 2, 1, 2, 4, 3}
Nothing left in Queue,
Dijkstra is done...!



(https://theoryofprogramming.files.wordpress.com/2015/01/dik1.jpg)

The Dijkstra's Algorithm is a little tricky. Many don't understand it in the first attempt. In reference to the diagram above, I will give a step-by-step explanation for each graph marked with the number on top in purple.

1. Firstly, initialize your components, the shortest distances array **D**, the priority queue **Q**, and starting vertex **s**. The distance from source to itself is zero. So, **D**(**s**) = 0, and the rest of the array is ∞ . The set of vertices **V** are inserted into the priority queue **Q**, with a priority **D**(**V**). Now, we start our algorithm by extracting (hence removing it from the priority queue) the minimum element from the priority queue. The minimum element in the priority queue will definitely be **s** (which is A here). Look at all the adjacent vertices of A. Vertices B, C, D are adjacent to A. We can go to B travelling the edge of weight 2, to C travelling an edge of weight 1, to D travelling an edge of weight 5. The values of **D**(B), **D**(C), **D**(D) are ∞ . We have found a new way of reaching them in 2, 1, 5 units respectively, which is less than ∞ , hence a shorter path. This is what the if-condition mentioned above does. So, we update the values of **D**(B), **D**(C), **D**(D) and the priorities of B, C, D, in the priority queue. With this we have finished processing the Vertex A.
2. Now, the process continues to its next iteration and we

extract the minimum element from the priority queue. The minimum element would be Vertex C which would be having a priority of 1. Now, look at all the adjacent vertices to C. There's Vertex D. From C, the it would take 1 unit of distance to reach D. But to reach C in prior, you need 1 more unit of distance. So, if you go to D, via C, the total distance would be 2 units, which is less than the current value of shortest distance discovered to D, **D**(D) = 5. So, we reduce the value of **D**(D) to 2. This reduction is also called as "Relaxation". With that we're done with Vertex C.

3. Now, the process continues to its next iteration and we extract the minimum element from the priority queue. Now, there are two minimum elements, B and D. You can go for anyone, it doesn't matter. For now, we will go for Vertex D. From Vertex D, you can go to Vertex E, and Vertex F, with a total distance of 2 + 2 {**D**(D) + (weight of D → E)}, and 2 + 3. Which is less than ∞ , so **D**(E) becomes 4 and **D**(F) becomes 5. We're done with Vertex D.
4. Now, the process continues to its next iteration and we extract the minimum element from the priority queue. The minimum element in the priority queue is vertex B. From vertex B, you can reach vertex F in 2 + 1 units of distance, which is less than the current value of **D**(F), 5. So, we relax **D**(F) to 3. From vertex B, you can reach vertex D in 2 + 2 units of distance, which is more than the current value of **D**(D), 2. This route is not considered as it is clearly proven to be a longer route. With that we're done with vertex B.
5. Now, the process continues to its next iteration and we extract the minimum element from the priority queue. The minimum element in the priority queue is vertex E. From vertex E, you can reach vertex C in 4 + 4 units of distance, which is more than the current value of **D**(C), 1. This route is not considered as it is clearly proven to be a longer route. With that we're done with vertex E.
6. Now, the process continues to its next iteration and we extract the minimum element from the priority queue. The minimum element in the priority queue is vertex F. You cannot go to any other vertex from vertex F, so, we're done with vertex F.
7. With the removal of vertex F, our priority queue becomes empty. So, our algorithm is done…! You can simply return the array **D** to output the shortest paths.

Having got an idea about the overall working of the Dijkstra's Algorithm, it's time to look at the pseudo-code –

```
dijsktra(G, S)
    D(S) = 0
    Q = G(V)

    while (Q != NULL)
        u = extractMin(Q)
        for all V in adjacencyList[u]
            if (D(u) + weight of edge < D(V))
                D(V) = D(u) + weight of edge
                decreasePriority(Q, V)
```

In the pseudo-code, **G** is the input graph and **S** is the starting vertex. I hope you understand the pseudo-code. If you don’t, feel free to comment your doubts. Now, before we code Dijkstra’s Algorithm, we must first prepare a tool, which is the Priority Queue.

The Priority Queue

The Priority Queue is implemented by a number of data structures such as the Binary Heap, Binomial Heap, Fibonacci Heap, etc. The priority queue in my code is implemented by a Binary Heap. If you are not aware about the Binary Heap, you can refer to my post on Binary Heaps (https://theoryofprogramming.wordpress.com/2014/12/28/binary-heaps/). But the implementation that we will do here is different from that in my post regarding the Binary Heap. The difference is that, here we will implement the Binary Heap using C++ STL Vector, not an array. This is because a heap is meant to grow, not remain of a fixed size, so we are using a data structure that can grow, a vector. Also, when we use a vector we can apply the same traversing techniques that we used in the case of an array. And, every element of our vector will be a Pair of two integers from the utility header file. The two integers represent vertex and weight, which is actually the shortest distances and this weight property will function as the priority to the elements. So, the vertices are placed in the priority queue based on their weight property. We will have to code the operations based on this weight property. Now the functionalities that we need from our priority queue are –

- **Insert** – We will insert |**V**| elements into the Priority Queue.
- **Extract Min** – We return the top-most element from the

Binary Heap and delete it. Finally we make the neccessary

- **Decrease Priority** – We decrease the priority of an element in the priority queue when we find a shorter path, as known as Relaxation.

If you know the working of the Binary Heap and have a little knowledge about the STL Library, you can code the Priority Queue in about 2-5 hours. You can keep referring to the internet of various functions and the syntaxes and any other doubts you have. Typing the code doesn't take long, but debugging it and making it work takes the most time. That's how you learn coding. Try your best, work on it for hours, if you don't get it, take a break for 10-20 minutes… Come back and check my code below and try to figure out how close you were to getting it perfect…!

```
/*
 * Priority Queue implemented
 * by a Binary Heap using
 * C++ STL Vector, for
 * Dijkstra's Algorithm
 *
 * Authored by,
 * Vamsi Sangam
 */

#include <cstdio>
#include <vector>
#include <utility>

using namespace std;

// Inserts an element into the Queue
void enqueue(vector< pair<int, int> > * pric
{
    (*priorityQueue).push_back(*entry);

    int i = (*priorityQueue).size() - 1;
    pair<int, int> temp;

    while (i > 0) {
        if ((*priorityQueue)[(i - 1) / 2].se
            temp = (*priorityQueue)[(i - 1)
            (*priorityQueue)[(i - 1) / 2] =
            (*priorityQueue)[i] = temp;

            i = (i - 1) / 2;
        } else {
            break;
        }
    }
```

```
}

// Iterates over the Queue to return the inc
int findByKey(vector< pair<int, int> > * pri
{
    int i;

    for (i = 0; i < (*priorityQueue).size();
        if ((*priorityQueue)[i].first == ent
            break;
        }
    }

    if (i != (*priorityQueue).size()) {
        return i;
    } else {
        return -1;
    }
}

// Decreases the priority of the element anc
void decreasePriority(vector< pair<int, int>
{
    (*priorityQueue)[index].second = newWeig

    int i = index;
    pair<int, int> temp;

    while (i > 0) {
        if ((*priorityQueue)[(i - 1) / 2].se
            temp = (*priorityQueue)[(i - 1)
            (*priorityQueue)[(i - 1) / 2] =
            (*priorityQueue)[i] = temp;

            i = (i - 1) / 2;
        } else {
            break;
        }
    }
}

// Returns the minimum element, deletes it a
pair<int, int> extractMin(vector< pair<int,
{
    pair<int, int> min = (*priorityQueue)[0]
    pair<int, int> temp;

    // Swap first and last
    temp = (*priorityQueue)[0];
    (*priorityQueue)[0] = (*priorityQueue)[(
    (*priorityQueue)[(*priorityQueue).size()

    (*priorityQueue).pop_back();

```

```
    int i = 0;
    pair<int, int> parent;
    pair<int, int> rightChild;
    pair<int, int> leftChild;

    while (i < (*priorityQueue).size()) {
        parent = (*priorityQueue)[i];
        printf("Currently at - (%d, %d)\n",

        if (2 * (i + 1) < (*priorityQueue).s
            // both children exist
            rightChild = (*priorityQueue)[2
            leftChild = (*priorityQueue)[2 *

            if (parent.second < leftChild.se
                break;
            } else {
                if (leftChild.second < right
                    temp = (*priorityQueue)|
                    (*priorityQueue)[2 * (i
                    (*priorityQueue)[i] = te

                    i = 2 * (i + 1) - 1;
                } else {
                    temp = (*priorityQueue)|
                    (*priorityQueue)[2 * (i
                    (*priorityQueue)[i] = te

                    i = 2 * (i + 1);
                }
            }
        } else if ((2 * (i + 1)) >= (*priori
            // only left child exists
            leftChild = (*priorityQueue)[2 *

            if (leftChild.second < parent.se
                temp = (*priorityQueue)[2 *
                (*priorityQueue)[2 * (i + 1)
                (*priorityQueue)[i] = temp;
            }

            break;
        } else {
            // no more children exist
            break;
        }
    }

    return min;
}

int main()
{
    int n;
```

```

    printf("Enter the size -\n");
    scanf("%d", &n);

    int vertex, weight, i;
    vector< pair<int, int> > priorityQueue;
    pair<int, int> entry;

    for (i = 0; i < n; ++i) {
        scanf("%d%d", &vertex, &weight);
        entry = make_pair(vertex, weight);
        enqueue(&priorityQueue, &entry);
    }

    printf("\n\nThe Priority Queue (Interpre

    vector< pair<int, int> >::iterator itr =

    while (itr != priorityQueue.end()) {
        printf("(%d, %d) ", (*itr).first, (*
        ++itr;
    }
    printf("\n");

    pair<int, int> min = extractMin(&priorit
    printf("\n\nExtract Min returned = (%d,
    itr = priorityQueue.begin();

    while (itr != priorityQueue.end()) {
        printf("(%d, %d) ", (*itr).first, (*
        ++itr;
    }
    printf("\n");

    decreasePriority(&priorityQueue, priorit
    printf("\n\ndecreasePriority() used, The
    itr = priorityQueue.begin();

    while (itr != priorityQueue.end()) {
        printf("(%d, %d) ", (*itr).first, (*
        ++itr;
    }
    printf("\n");

    return 0;
}
```

There are many other functionalities that a Priority Queue can give. But for now, we'll need only these.

Joining the pieces

After you have your tool ready, you are all good to code Dijkstra's Algorithm. Coding the Dijkstra's Algorithm is easy but can go really weird. This is because you need to handle and co-ordinate many data structures at once. You'll have to manage the adjacency list, the priority queue, the shortest distance array, and most importantly your loops…! You will surely end up spending more time in debugging the code, which is perfectly the right way of doing it. All throughout the coding, keep revising the step-by-step process explained above. If you don't get it you, don't fret, I put my code below. Before you look at my code, I would like to mention a few things –

- We cannot have infinity in programming, so the shortest distances are initialised to the highest possible value in integer range, present in a macro, INT_MAX, in the header file climits.
- The header file utility must be included to use pairs.

```
/*
 * Dijkstra's Algorithm in C++
 * using Binary Heap as Priority
 * Queue implemented using
 * C++ STL Vector
 *
 * Authored by,
 * Vamsi Sangam
 */

#include <cstdio>
#include <cstdlib>
#include <climits>
#include <vector>
#include <utility>

using namespace std;

// Our Vertex for Graph
struct node {
    int vertex, weight;
    struct node * next;
};

// To construct our Adjacency List
// Follows Head Insertion to give O(1) inser
struct node * addEdge(struct node * head, i
```

```
{
    struct node * p = (struct node *) calloc

    p->vertex = vertex;
    p->weight = weight;
    p->next = head;

    return p;
}

// Adds vertices to the Priority Queue, Vert
// as pairs of vertex number and its shortes
// This is logically a Binary Heap Insertion
void enqueue(vector< pair<int, int> > * prio
{
    (*priorityQueue).push_back(*entry);

    int i = (*priorityQueue).size() - 1;
    pair<int, int> temp;

    while (i > 0) {
        // Checking the priority of the pare
        if ((*priorityQueue)[(i - 1) / 2].se
            temp = (*priorityQueue)[(i - 1)
            (*priorityQueue)[(i - 1) / 2] =
            (*priorityQueue)[i] = temp;

            i = (i - 1) / 2;
        } else {
            break;
        }
    }
}

// Finds for a Vertex in the Priority Queue
// returns its index as in its vector implem
int findByKey(vector< pair<int, int> > * pri
{
    int i;

    // Linear Search
    for (i = 0; i < (*priorityQueue).size();
        if ((*priorityQueue)[i].first == ent
            break;
        }
    }

    if (i != (*priorityQueue).size()) {
        return i;
    } else {
        return -1;
    }
}
```

```
// Decreases the priority of a given entry i
// Priority Queue who's location is given by
// to 'newWeight' and re-arranges the Binary
void decreasePriority(vector< pair<int, int>
{
    // Decreasing Priority
    (*priorityQueue)[index].second = newWeig

    int i = index;
    pair<int, int> temp;

    // Adjusting the Binary Heap, similar re
    while (i > 0) {
        if ((*priorityQueue)[(i - 1) / 2].se
            temp = (*priorityQueue)[(i - 1)
            (*priorityQueue)[(i - 1) / 2] =
            (*priorityQueue)[i] = temp;

            i = (i - 1) / 2;
        } else {
            break;
        }
    }
}

// Picks up the minimum element of the Prior
// the Binary Heap and finally returns the M
// Functionally resembles Delete operation i
// returns the deleted element which is the
pair<int, int> extractMin(vector< pair<int,
{
    pair<int, int> min = (*priorityQueue)[0]
    pair<int, int> temp;

    // Swap first and last elements
    temp = (*priorityQueue)[0];
    (*priorityQueue)[0] = (*priorityQueue)[(
    (*priorityQueue)[(*priorityQueue).size()

    (*priorityQueue).pop_back();

    int i = 0;
    pair<int, int> parent;            // The
    pair<int, int> rightChild;        // are
    pair<int, int> leftChild;     // the if

    while (i < (*priorityQueue).size()) {
        parent = (*priorityQueue)[i];

        if (2 * (i + 1) < (*priorityQueue).s
            // both children exist
            rightChild = (*priorityQueue)[2
            leftChild = (*priorityQueue)[2 *

```

```
            if (parent.second < leftChild.se
                // Parent has lesser priorit
                break;
            } else {
                if (leftChild.second < right
                    // Left-child has a less
                    temp = (*priorityQueue)|
                    (*priorityQueue)[2 * (i
                    (*priorityQueue)[i] = te

                    i = 2 * (i + 1) - 1;
                } else {
                    // Right-child has a les
                    temp = (*priorityQueue)|
                    (*priorityQueue)[2 * (i
                    (*priorityQueue)[i] = te

                    i = 2 * (i + 1);
                }
            }
        } else if ((2 * (i + 1)) >= (*prior
            // only left child exists
            leftChild = (*priorityQueue)[2 *

            if (leftChild.second < parent.se
                // Left-child has a lesser p
                temp = (*priorityQueue)[2 *
                (*priorityQueue)[2 * (i + 1)
                (*priorityQueue)[i] = temp;
            }

            break;
        } else {
            // no more children exist
            break;
        }
    }

    return min;
}

// The Dijkstra's Algorithm sub-routine whic
// number of vertices, a starting vertex, an
// input and computest the shortest paths an
void dijkstra(struct node * adjacencyList[],
{
    int i;

    // Initially no routes to vertices are k
    // here, we initialize to a very high in
    for (i = 0; i < vertices; ++i) {
        shortestDistances[i] = INT_MAX;
    }

```

```
    // Setting distance to source to zero
    shortestDistances[startVertex] = 0;

    struct node * trav;
    vector< pair<int, int> > priorityQueue;
    pair<int, int> min;

    // Making a the vertex that corresponds
    // 'startVertex' which will have a prior
    // and we begin to intialise the Priorit
    pair<int, int> entry = make_pair(startVe
    enqueue(&priorityQueue, &entry);

    // Initialising Priority Queue
    for (i = 1; i <= vertices; ++i) {
        if (i == startVertex) {
            continue;
        } else {
            // Priorities are set to a high
            entry = make_pair(i, INT_MAX);
            enqueue(&priorityQueue, &entry);
        }
    }

    // We have the tools ready..! Let's rol
    while (priorityQueue.size() != 0) {
        min = extractMin(&priorityQueue);

        trav = adjacencyList[min.first];
        while (trav != NULL) {
            if (shortestDistances[trav->vert
                // We have discovered a new
                // Make the neccesary adjust
                entry = make_pair(trav->vert

                int index = findByKey(&prior

                decreasePriority(&priorityQu
                shortestDistances[trav->vert
            }

            trav = trav->next;
        }
    }
}

int main()
{
    int vertices, edges, i, j, v1, v2, w;

    printf("Enter the Number of Vertices -\r
    scanf("%d", &vertices);

    printf("Enter the Number of Edges -\n");
```

```
    scanf("%d", &edges);

    struct node * adjacencyList[vertices + 1
    //Size is made (vertices + 1) to use the
    //array as 1-indexed, for simplicity

    //Must initialize your array
    for (i = 0; i <= vertices; ++i) {
        adjacencyList[i] = NULL;
    }

    printf("\n");

    for (i = 1; i <= edges; ++i) {
        scanf("%d%d%d", &v1, &v2, &w);
        adjacencyList[v1] = addEdge(adjacenc
    }

    //Printing Adjacency List
    printf("\nAdjacency List -\n\n");
    for (i = 1; i <= vertices; ++i) {
        printf("adjacencyList[%d] -> ", i);

        struct node * temp = adjacencyList[i

        while (temp != NULL) {
            printf("%d(%d) -> ", temp->verte
            temp = temp->next;
        }

        printf("NULL\n");
    }

    int startVertex;

    printf("Choose a Starting Vertex -\n");
    scanf("%d", &startVertex);

    int shortestDistances[vertices + 1];

    dijkstra(adjacencyList, vertices, startV

    printf("\n\nDijkstra's Algorithm Used -
    for (i = 1; i <= vertices; ++i) {
        printf("%d ", *(shortestDistances +
    }
    printf("\n");

    return 0;
}
```

This is the Dijkstra's Algorithm. The code is well commented with explanation. If you don't understand anything or if you have any doubts. Feel free to comment them. Now talking about the complexity of Dijkstra's Algorithm. We perform |V| enqueue operations into the priority queue, which take O(log N), here, N is |V|, so this takes O(|V| log |V|). And at most |E| decrease priority operations which will take O(|V|) time. The extract-min is also called |V| times which will take O(|V| log |V|) time. So, the overall complexity of Dijkstra's Algorithm we wrote is O(|V| log |V| + |E||V|). Dijkstra's Algorithm can be improved by using a Fibonacci Heap as a Priority Queue, where the complexity reduces to O(|V| log |V| + |E|). But the Fibonacci Heap is an incredibly advanced and difficult data structure to code. We'll talk about that implementation later.

I really hope my post has helped you in understanding the Dijkstra's Algorithm. If it did, let me know by commenting. I tried my best to keep it as simple as possible. If you have any doubts, you can comment them too and I will surely reply to them. This algorithm is a particularly tough one. So, good luck… Keep practicing and… Happy Coding…! 🙂

You may also like –

- [Dynamic Programming – Introduction and Fibonacci Numbers (https://theoryofprogramming.wordpress.com/2015/03/02/dynamic-programming-introduction-and-fibonacci-numbers/)](https://theoryofprogramming.wordpress.com/2015/03/02/dynamic-programming-introduction-and-fibonacci-numbers/)
- [Binary Indexed Tree or Fenwick Tree (https://theoryofprogramming.wordpress.com/2014/12/24/binary-indexed-tree-or-fenwick-tree/)](https://theoryofprogramming.wordpress.com/2014/12/24/binary-indexed-tree-or-fenwick-tree/)
- [Number Theory – Modular Arithmetic (https://theoryofprogramming.wordpress.com/2014/12/24/number-theory-modular-arithmetic/)](https://theoryofprogramming.wordpress.com/2014/12/24/number-theory-modular-arithmetic/)
- [Graph Theory – Breadth First Search (https://theoryofprogramming.wordpress.com/2014/12/25/graph-theory-breadth-first-search/)](https://theoryofprogramming.wordpress.com/2014/12/25/graph-theory-breadth-first-search/)
- [Quick Sort (https://theoryofprogramming.wordpress.com/2015/01/20/quick-sort/)](https://theoryofprogramming.wordpress.com/2015/01/20/quick-sort/)

More in the Navigation Page ! → (https://theoryofprogramming.wordpress.com/navigation-page/)

Graph Theory

ADJACENCY LIST **ALGORITHMS** **BINARY HEAPS** **GRAPH THEORY** **PRIORITY QUEUE** **SHORTEST PATH**