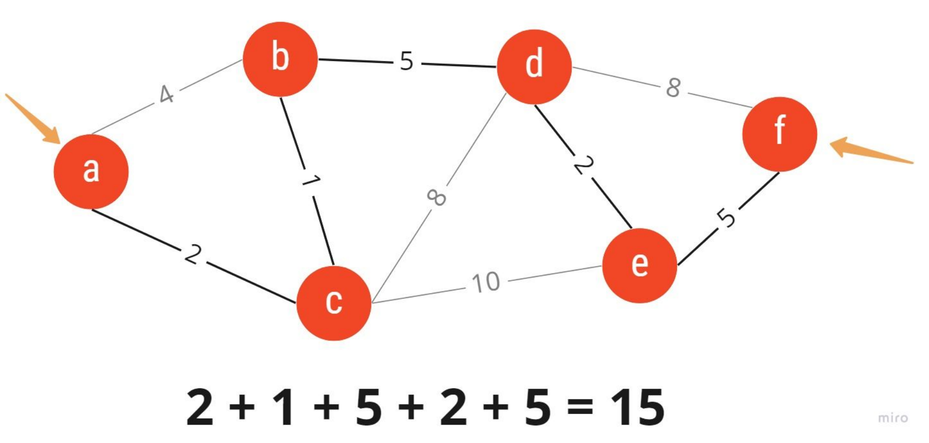
Dijkstra’s Algorithm

Caleb Fernandes
Jadon Orr
Luke Shaw

Introduction

Shortest-path finding algorithm designed for a weighted graph. The algorithm will take a start vertex and end vertex as inputs and output the length of the shortest path (summation of the weights along the path) and the actual path.



Shortest Path:

A → C → B → D → E → F

Distance of Shortest Path: 15

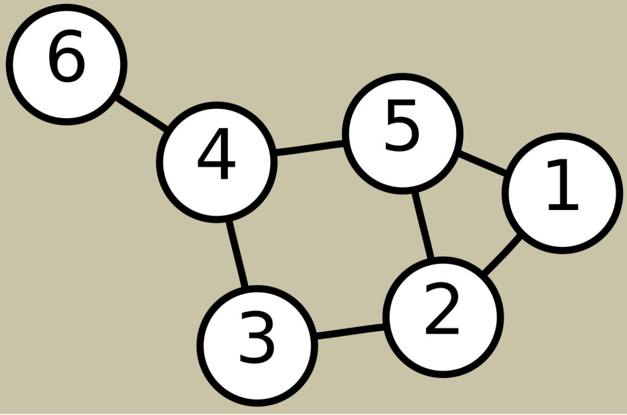
Implementation Components

The algorithm will contain a Vertex, Edge, Array-based Priority Queue, and Graph ADT. The Graph will contain an Adjacency List which will store a Vertex object and then the Edges.

```
class Graph : public GraphBase
{
public:
    Graph();
    ~Graph();

    void addVertex(std::string label) override;
    void removeVertex(std::string label) override;
    void addEdge(std::string label1, std::string label2, unsigned long weight) override;
    void removeEdge(std::string label1, std::string label2) override;
    unsigned long shortestPath(std::string startLabel, std::string endLabel, std::vector<std::string> &path) override;
    void printGraph();
    Vertex *getVertex(const string &label);

    vector<Vertex> V_list;
    vector<Map<string, unsigned long>> distance; // this will conain elements that look like (Vertex, distance)
    vector<Map<string, string>> parents;         // this will contain (Vertex, Vertex)
};
```

Vertex Class

A vertex is defined as a singular node which is present within a graph. A vertex is almost always connected to another vertex via an edge. In the instance below, the circled numbers are considered vertices.




```cpp
class Vertex
{
public:
    Vertex(const string &name);
    void printVertex();

    string label;
    vector<Edge> EdgeList;
};
```

```cpp
Vertex::Vertex(const string &name)
{
    label = name;
}

void Vertex::printVertex()
{
    cout << this->label << ": { ";
    for (auto e : this->EdgeList)
    {
        e.printEdge();
        cout << " ";
    }
    cout << "}" << endl;
}
```

```
void Graph::addVertex(std::string label)
{
    // will check to make sure that you cannot add duplicate vertex
    for (const Vertex &v : V_list)
        if (v.label == label)
            return;
    // adds vertex to the V_list
    V_list.push_back(Vertex(label));
};
```

```cpp
void Graph::removeVertex(std::string label)
{
    // remove from v_list
    for (auto v = V_list.begin(); v != V_list.end();)
    {
        if (v->label == label)
        {
            v = V_list.erase(v); // erase returns the next valid iterator
        }
        else
        {
            ++v;
        }
    }
    // remove from all edge lists
    // A simplier way to do this would be to use the label.edgeList to create a li
    for (Vertex &v : V_list)
    {
        for (auto e = v.EdgeList.begin(); e != v.EdgeList.end();)
        {
            if (e->start == label || e->end == label)
            {
                e = v.EdgeList.erase(e); // erase returns the next valid iterator
            }
            else
            {
                ++e;
            }
        }
    }
};
```

Edge Class

An edge is represents a connection between two vertices in a graph, specifying the actual distance or weight between these two vertices.

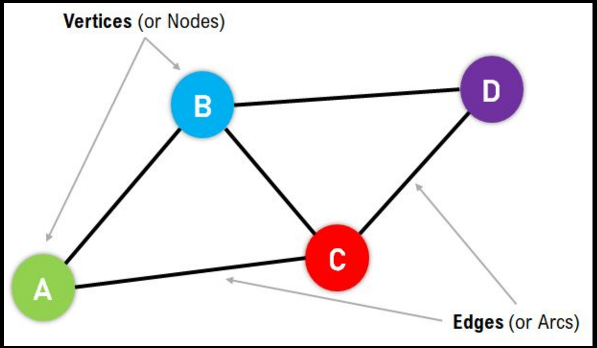
It can be defined as:

edge(vertex A, vertex B, distance x);

In this instance, an undirected node is created meaning there is no fixed direction; traversal is possible from A → B and B → A.

```cpp
class Edge
{
public:
    Edge(const string &x, const string &y, const unsigned long &z);
    void printEdge();
    string start;
    string end;
    unsigned long distance;
};
```

```cpp
Edge::Edge(const string &x, const string &y, const unsigned long &z)
{
    start = x;
    end = y;
    distance = z;
}
```



addEdge( )

1. Checks if two vertices are the same then it will skip adding an edge.
2. Find the two vertices in the vertex list and store them in v1 and v2.
3. Ensure the vertices are not null.
4. Check if the connection between the two vertices exist if it does skip adding the edge.
5. Add the edge into the edge list in both directions using the same the weight.

```cpp
void Graph::addEdge(std::string label1, std::string label2, unsigned long weight)
{
    if (label1 == label2)
        return;

    Vertex *v1 = nullptr;
    Vertex *v2 = nullptr;

    // Find the two vertices
    for (auto &v : V_list)
    {
        if (v.label == label1)
            v1 = &v;
        else if (v.label == label2)
            v2 = &v;
    }

    // Ensure both vertices exist
    if (!v1 || !v2)
        return;

    // Check if edge already exists
    for (const auto &e : v1->EdgeList)
    {
        if ((e.start == label1 && e.end == label2) || (e.start == label2 && e.end == label1))
            return;
    }

    // Add edge to both vertices
    v1->EdgeList.push_back(Edge(label1, label2, weight));
    v2->EdgeList.push_back(Edge(label2, label1, weight));
}
```

removeEdge()

1. Iterate through every vertex in the vertex list.
2. If the label given in the arguments being the edge removed matches a label in the list, continue.
3. Iterate over the edge list and find the edge that matches the labels.
4. Remove the edge at the iterator and exit the function.

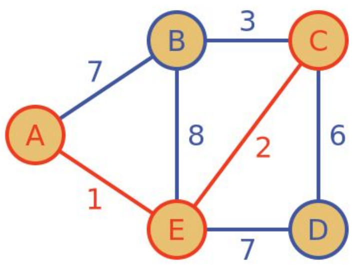
```cpp
void Graph::removeEdge(std::string label1, std::string label2)
{
    for (Vertex &v : V_list)
    {
        if (v.label == label1 || v.label == label2)
        {
            for (auto e = v.EdgeList.begin(); e != v.EdgeList.end();)
            {
                if ((e->start == label1 && e->end == label2) || (e->start == label2 && e->end == label1))
                {
                    e = v.EdgeList.erase(e); // erase returns the next valid iterator
                }
                else
                {
                    ++e;
                }
            }
        }
    }
}
```

ShortestPath()

Vis: if the shortest distance to the vertex has already been found

Dist: contains the shortest distance to that vertex

Parent/Prev: contains where the vertex came from



	Vis	Dist	Prev
A	1	0	-
B	0	6	C
C	1	3	E
D	0	8	E
E	1	1	A

```
//-----INITIALIZATION OF VECTORS----
PQ Q;
distance.clear(); // clearing the distance vector;
parents.clear();  // clear distance vector
vector<std::string> visited;
// initialize the distance vector
//  Init distances
for (auto &v : V_list)
{
    unsigned long dist = (v.label == startLabel) ? 0 : ULONG_MAX;
    distance.push_back(Map<string, unsigned long>(v.label, dist));
    if (v.label == startLabel)
    {
        Q.enQ(Map<Vertex *, unsigned long>(&v, 0));
        parents.push_back(Map<string, string>(v.label, "\0")); // no parent for start
    }
}
```

Main Loop

1. Iterate through Priority Queue (PQ) until empty
2. Iterate through each Neighbor of Vertex popped from PQ
3. If vertex is already visited: do nothing and continue
4. Find the distance to the Neighbor by adding the distance stored in the (Vertex, Distance) pair in the PQ (remember this distance is the shortest distance to get to the vertex we have found already) to the weight of the Edge as TotalDistance (TotalDistance = Distance + EdgeWeight)
5. If TotalDistance < Distance[Neighbor] (this is the value stored in the distance array at the Neighbor entry)
   a. Update Distance[Neighbor] with the new distance
   b. Update Parent of Neighbor with Vertex
   c. Add (Neighbor, TotalDistance) to PQ
6. Else: do nothing and continue

```cpp
//-----MAIN LOOP OF CODE----
while (!Q.empty())
{
    auto node = Q.min();
    Q.deQ();

    Vertex *v = node.first;
    unsigned long dist = node.second;

    // Check if already visited
    bool alreadyVisited = false;
    for (const auto &label : visited)
    {
        if (label == v->label)
        {
            alreadyVisited = true;
            break;
        }
    }
    if (alreadyVisited)
        continue;
    visited.push_back(v->label);
```

Main Loop cont…

```cpp
// create the distance metric
for (auto &edge : v->EdgeList)
{
    // check the total distance of the node against the distance in  distance vector
    string n = edge.end;
    unsigned long l = edge.distance;
    unsigned long total_dist = dist + l;

    // iterating through distance vector
    for (auto &d : distance)
    {
        //checking to see if entry at totalDist < Distance[Neighbor]
        if (d.first == n && total_dist < d.second)
        {
            //updating Distance[Neighbor]
            d.second = total_dist;
            Q.enQ(Map<Vertex *, unsigned long>(getVertex(n), total_dist));

            // Update or add parent
            bool in_parents = false;
            for (auto &p : parents)
            {
                if (p.first == n)
                {
                    p.second = v->label;
                    in_parents = true;
                    break;
                }
            }
            //if the node has not been added to parent vector already
            if (!in_parents)
            {
                parents.push_back(Map<string, string>(n, v->label));
            }
        }
    }
```

Reconstruct Path

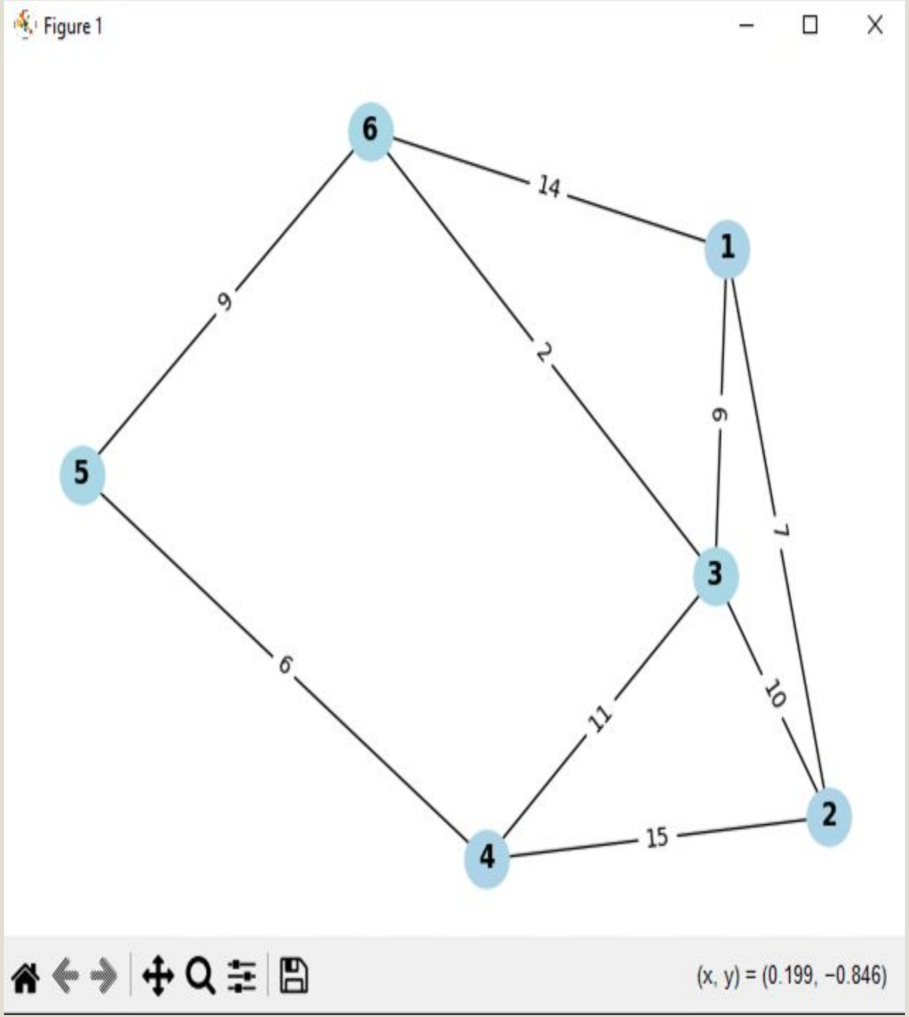
```cpp
    //---RECONSTRUCT PATH---
    path.clear();
    string current = endLabel;
    while (current != "\0")
    {
        path.insert(path.begin(), current);
        bool found = false;
        for (auto &p : parents)
        {
            if (p.first == current)
            {
                current = p.second;
                found = true;
                break;
            }
        }
        if (!found)
            break; // no path
    }

    //---GET FINAL DISTANCE
    for (auto &d : distance)
    {
        if (d.first == endLabel)
            return d.second == ULONG_MAX ? 0 : d.second;
    }

    // incorrect path
    return 0;
};
```

Demo Time

Case 1: Simple Numbers
- a. Basic Run of Algorithm
- b. Same Graph but after Vertex 3 is removed
- c. Same Graph but after Edge (5,6,9) is removed

Case 2: Building Path
Case 3: Another Building Path

```
TEST CASE 1:

Graph:
1: { (1, 2, 7) (1, 3, 9) (1, 6, 14) }
2: { (2, 1, 7) (2, 3, 10) (2, 4, 15) }
3: { (3, 1, 9) (3, 2, 10) (3, 4, 11) (3, 6, 2) }
4: { (4, 2, 15) (4, 3, 11) (4, 5, 6) }
5: { (5, 4, 6) (5, 6, 9) }
6: { (6, 1, 14) (6, 3, 2) (6, 5, 9) }

Basic Test Run
Shortest Path Length: 20
Shortest Path: 1 3 6 5

Removing Vertex 3
Graph:
1: { (1, 2, 7) (1, 6, 14) }
2: { (2, 1, 7) (2, 4, 15) }
4: { (4, 2, 15) (4, 5, 6) }
5: { (5, 4, 6) (5, 6, 9) }
6: { (6, 1, 14) (6, 5, 9) }
Shortest Path Length: 23
Shortest Path: 1 6 5

Removing Edge ('5', '6', 9)
Graph:
1: { (1, 2, 7) (1, 6, 14) }
2: { (2, 1, 7) (2, 4, 15) }
4: { (4, 2, 15) (4, 5, 6) }
5: { (5, 4, 6) }
6: { (6, 1, 14) }
Shortest Path Length: 28
Shortest Path: 1 2 4 5

----------------------------
```




```
TEST CASE 1:

Graph:
1: { (1, 2, 7) (1, 3, 9) (1, 6, 14) }
2: { (2, 1, 7) (2, 3, 10) (2, 4, 15) }
3: { (3, 1, 9) (3, 2, 10) (3, 4, 11) (3, 6, 2) }
4: { (4, 2, 15) (4, 3, 11) (4, 5, 6) }
5: { (5, 4, 6) (5, 6, 9) }
6: { (6, 1, 14) (6, 3, 2) (6, 5, 9) }

Basic Test Run
Shortest Path Length: 20
Shortest Path: 1 3 6 5

Removing Vertex 3
Graph:
1: { (1, 2, 7) (1, 6, 14) }
2: { (2, 1, 7) (2, 4, 15) }
4: { (4, 2, 15) (4, 5, 6) }
5: { (5, 4, 6) (5, 6, 9) }
6: { (6, 1, 14) (6, 5, 9) }
Shortest Path Length: 23
Shortest Path: 1 6 5

Removing Edge ('5', '6', 9)
Graph:
1: { (1, 2, 7) (1, 6, 14) }
2: { (2, 1, 7) (2, 4, 15) }
4: { (4, 2, 15) (4, 5, 6) }
5: { (5, 4, 6) }
6: { (6, 1, 14) }
Shortest Path Length: 28
Shortest Path: 1 2 4 5

____________________________
```