Dijkstra's Algorithm

Dean Worrels, Jason Maya, Yahya Khlifa, Sujal Kumar Shah

April 20th 2025

What is the assignment

The goal of the project is to implement a Dijkstra's algorithm in C++ which finds the shortest path between two vertices in an undirected weighted graph, with the limitations of not using external non-standard C++ libraries and only using String and Vector classes from standard libraries.

The project also requires implementing methods of dynamic modification to the graph, such as adding or removing vertices and edges.

What is Dijkstra's Algorithm

Dijkstra’s is an algorithm that finds the shortest path between nodes in a graph. Starting from a source node, it repeatedly visits the nearest unvisited node and updates its neighbors’ distances, eventually calculating the minimum distance to all other nodes.

Why use Dijkstra’s Algorithm

- Solves the single-source shortest path problem efficiently
- Designed for graphs with non-negative edge weights
- Greedy approach ensures optimal local decisions lead to optimal global paths
- Efficient when combined with heaps

Real world applications:

  - GPS navigation systems
  - Robotics and drone path planning
  - Game AI for NPC pathfinding

HeapQueue.hpp

We borrowed this hpp from PP3 for its implementation of a priority queue

```
public:

VectorCompleteTree() : V(1) {}

int size() const { return V.size() - 1; }

Position left(const Position &p) { return pos(2 * idx(p)); }

Position right(const Position &p) { return pos(2 * idx(p) + 1); }

Position parent(const Position &p) { return pos(idx(p) / 2); }

bool hasLeft(const Position &p) const { return 2 * idx(p) <= size(); }

bool hasRight(const Position &p) const { return 2 * idx(p) + 1 <= size();
}

bool isRoot(const Position &p) const { return idx(p) == 1; }

Position root() { return pos(1); }

Position last() { return pos(size()); }

void addLast(const E &e) { V.push_back(e); }

void removeLast() { V.pop_back(); }

void swap(const Position &p, const Position &q)
```

HeapQueue.hpp

continued

```
class HeapQueue
{
public:
  int size() const;
  bool empty() const;
  void insert(const E &e);
  const E &min();
  void removeMin();
```

Graph.hpp

Functions

```
class Graph : public GraphBase {
public:
 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;
```

Graph.cpp

From shortestPath function, this is the loop that implements Dijkstra's algorithm

```
//Loop for Dijkstra's algorithm
    for (size_t i = 0; i < vertices.size(); ++i)
    {
        //Find the vertex with the smallest distance
        unsigned long minDist = MAX_WEIGHT;
        int u = -1;

        for (size_t j = 0; j < dist.size(); ++j)
        {
            if (!visited[j] && dist[j].first < minDist)
            {
                minDist = dist[j].first;
                u = j;
            }
        }

        if (u == -1) break; //All reachable vertices have been
visited

        visited[u] = true;
        std::string currLabel = dist[u].second;
        int vertexIndex = findVertexIndex(currLabel);

        //Update distances for neighboring vertices
        for (const auto& edge : vertices[vertexIndex].adj)
        {
            int v = findVertexIndex(edge.label);

            if (visited[v])
            continue;   //Skip visited vertices

            unsigned long alt = dist[u].first + edge.weight;
            if (alt < dist[v].first)
            {
                dist[v].first = alt;
                prev[v] = currLabel;
```

main.cpp

Where we put our test cases

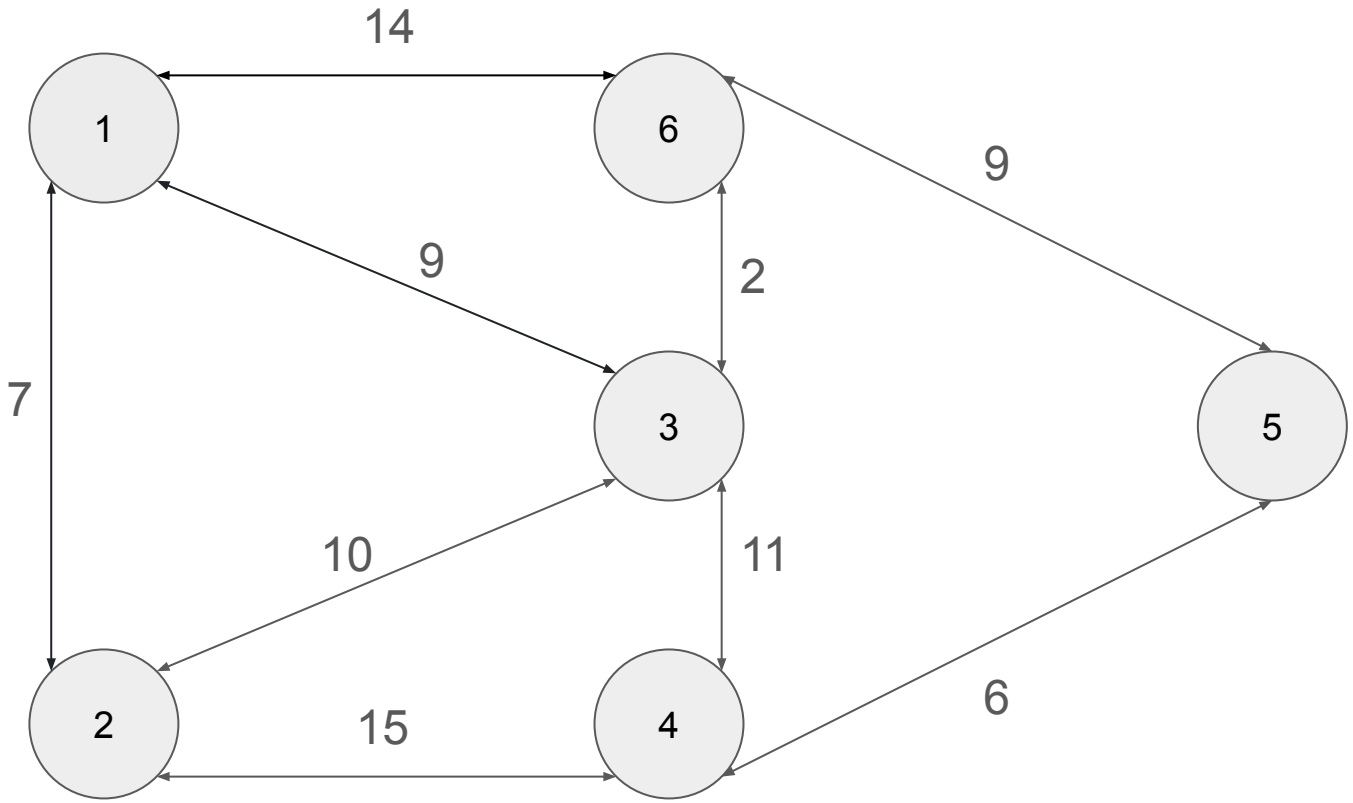
```
void runGraph1() {
    Graph g;
    std::vector<std::string> vertices1 = { "1", "2", "3",
"4", "5", "6" };
    std::vector<EdgeStruct> edges1 = {
        {"1", "2", 7}, {"1", "3", 9}, {"1", "6", 14}, {"2"
"3", 10},
        {"2", "4", 15}, {"3", "4", 11}, {"3", "6", 2},
{"4", "5", 6}, {"5", "6", 9}
    };

    for (const auto& v : vertices1) g.addVertex(v);
    for (const auto& edge : edges1) g.addEdge(edge.from,
edge.to, edge.weight);

    std::vector<std::string> path;
    unsigned long dist = g.shortestPath("1", "5", path);

    std::cout << "Graph 1 shortest path from 1 to 5: \n";
    std::cout << "Distance: " << dist << "\nPath: ";
    for (const auto& node : path) std::cout << node << " "
    std::cout << "\n\n";
}
```

Demo: Graph 1 (Shortest path 1 to 5)

Visited:
{}

Unvisited:
{1,2,3,4,5,6}

Node	Distance	Previous
1	0	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	

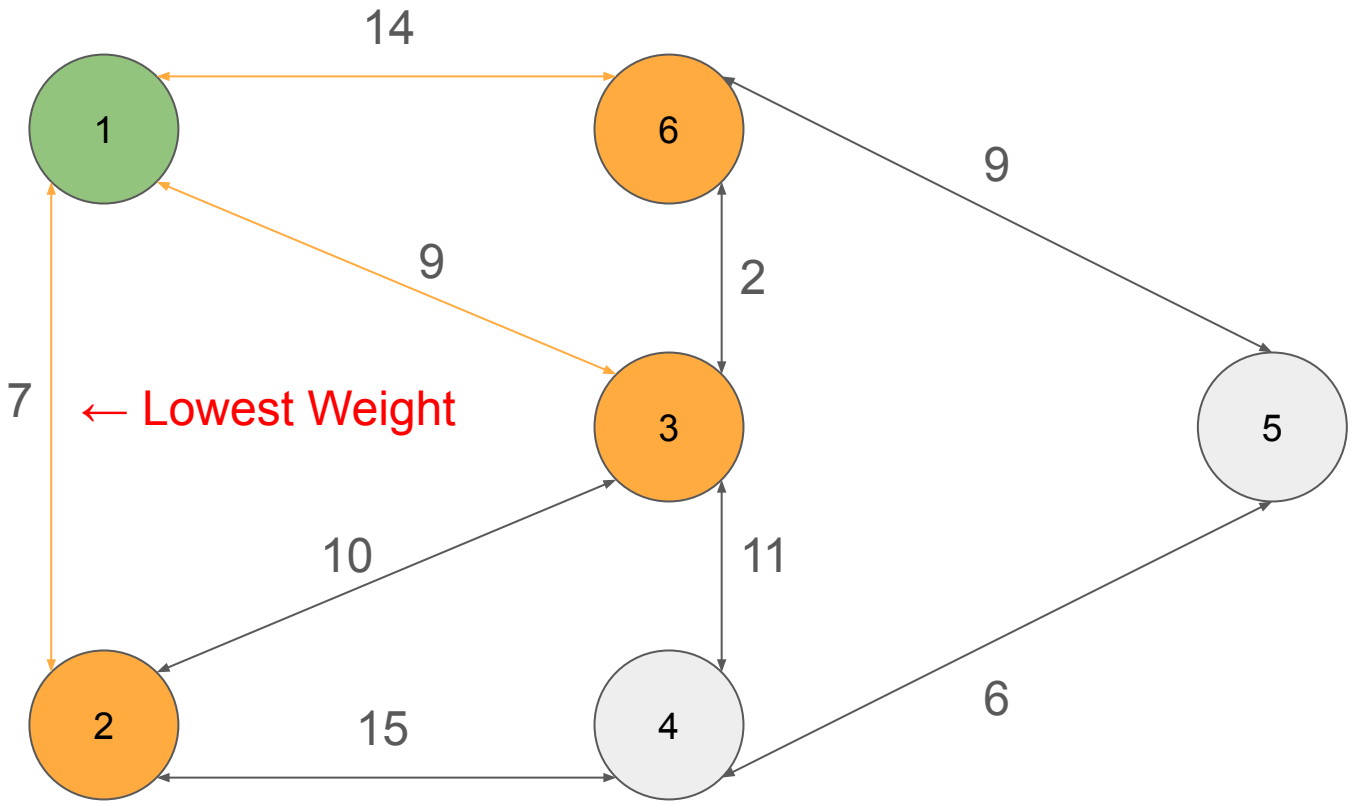


Demo: Graph 1 (Shortest path 1 to 5)

Visited:
{}

Unvisited:
{1,2,3,4,5,6}

Node	Distance	Previous
1	0	
2	**7**	**1**
3	**9**	**1**
4	∞	
5	∞	
6	**14**	**1**

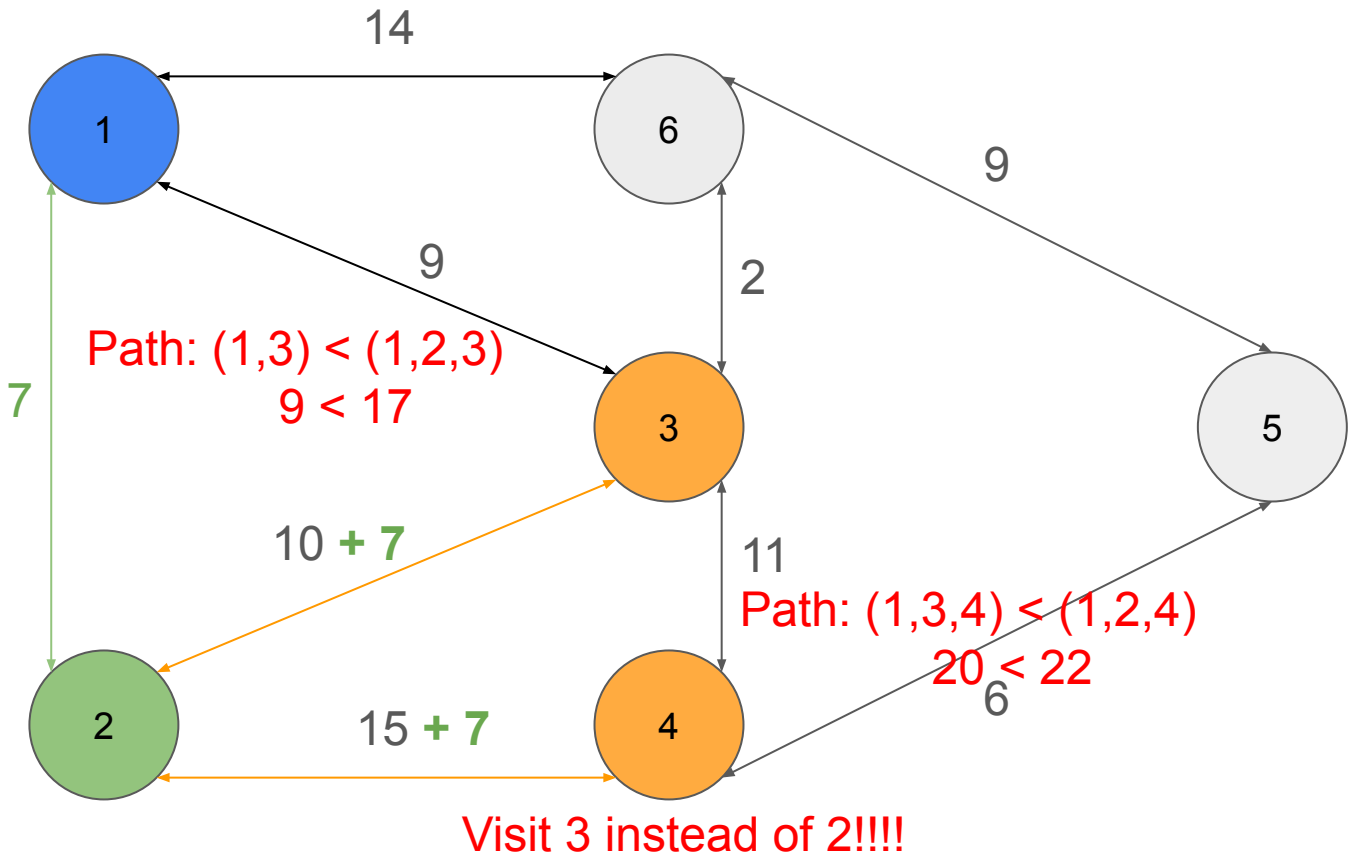


Demo: Graph 1 (Shortest path 1 to 5)

Visited:
{1}

Unvisited:
{2,3,4,5,6}

Node	Distance	Previous
1	0	
2	7	1
3	9	1
4	**22**	**2**
5	∞	
6	14	1

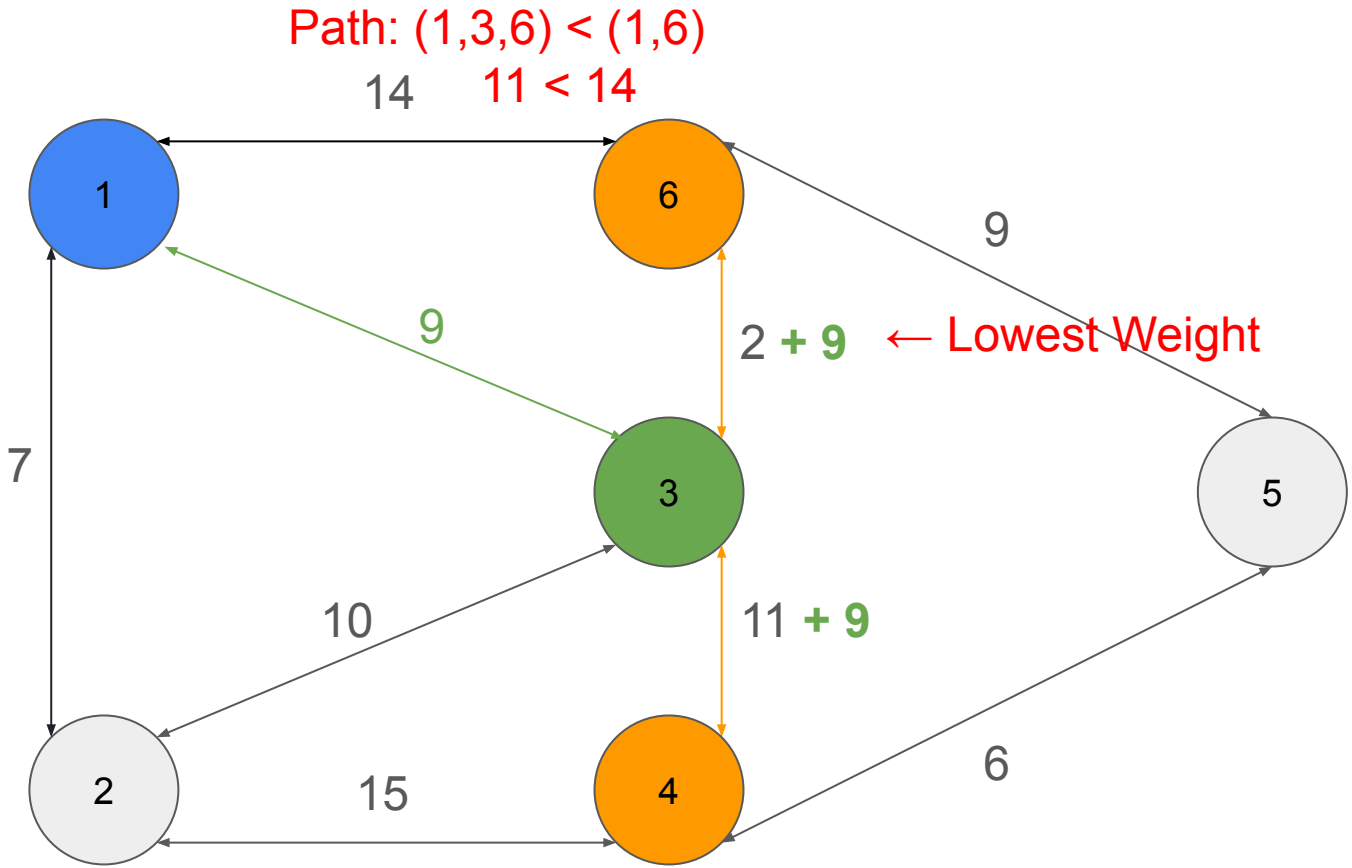


Demo: Graph 1 (Shortest path 1 to 5)

Visited:
{1}

Unvisited:
{2,3,4,5,6}

Node	Distance	Previous
1	0	
2	7	1
3	9	1
4	**20**	**3**
5	∞	
6	**11**	**3**



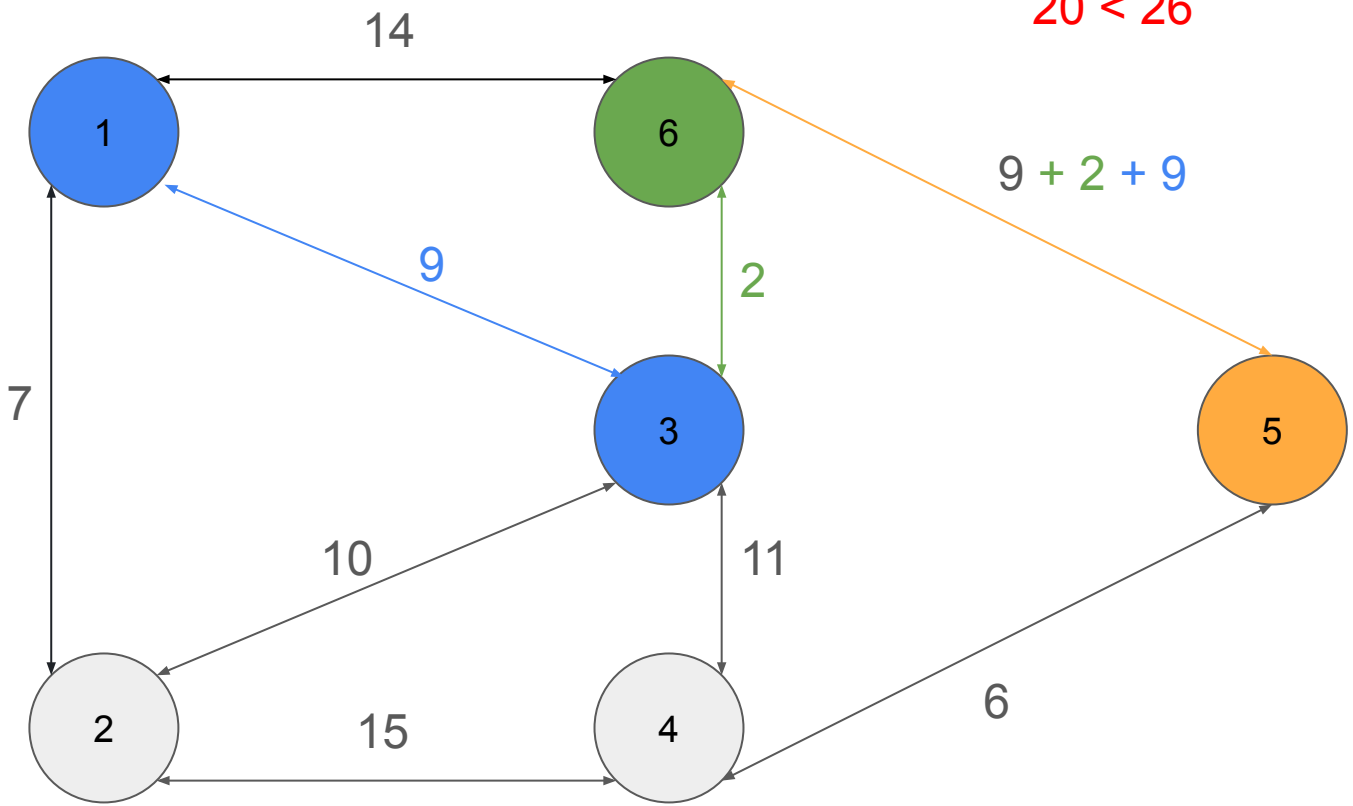
Demo: Graph 1 (Shortest path 1 to 5)

Path: (1,3,6,5) < (1,3,4,5)
20 < 26

Visited:
{1,**3**}

Unvisited:
{2,4,5,6}

Node	Distance	Previous
1	0	
2	7	1
3	9	1
4	20	3
5	**20**	**6**
6	11	3

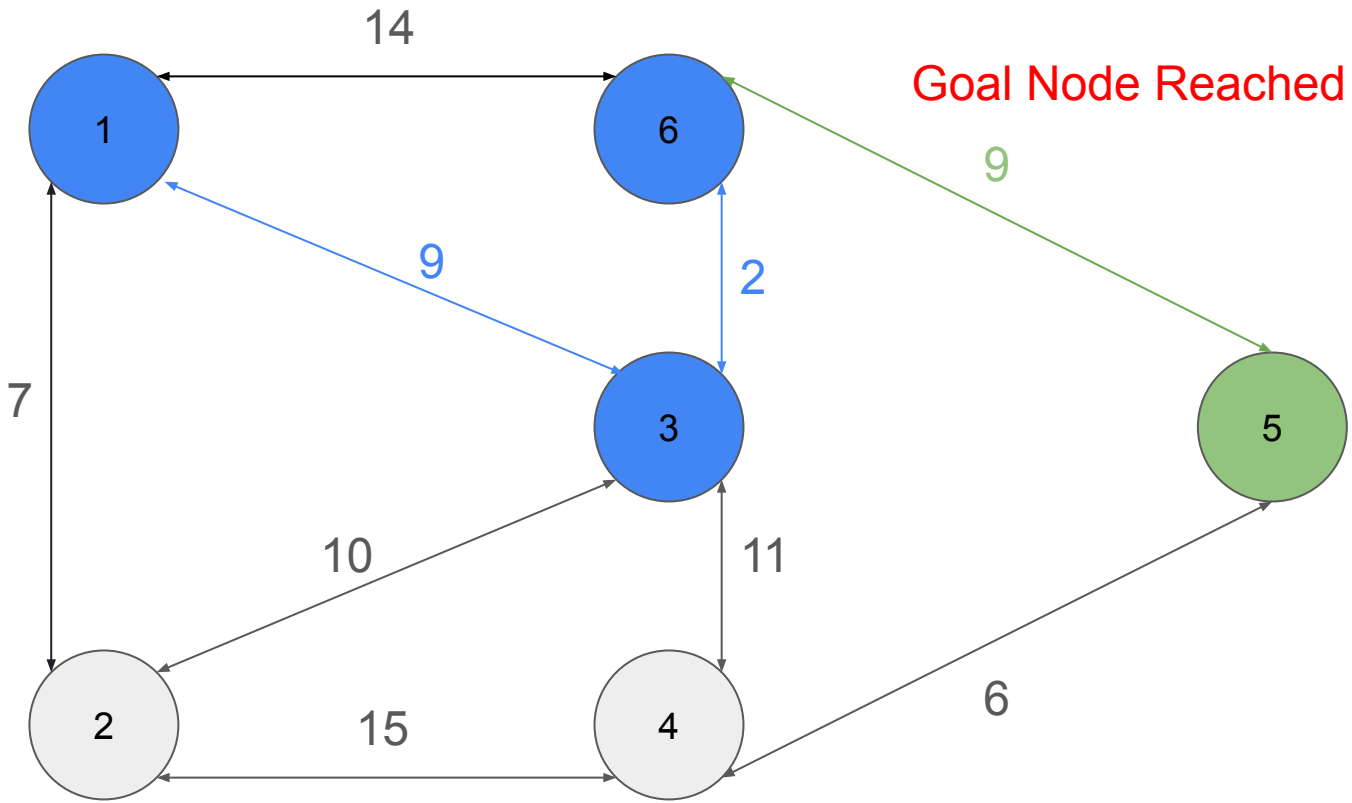


Demo: Graph 1 (Shortest path 1 to 5)

Visited:
{1,3,**6**}

Unvisited:
{2,4,5}

Node	Distance	Previous
1	0	
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3

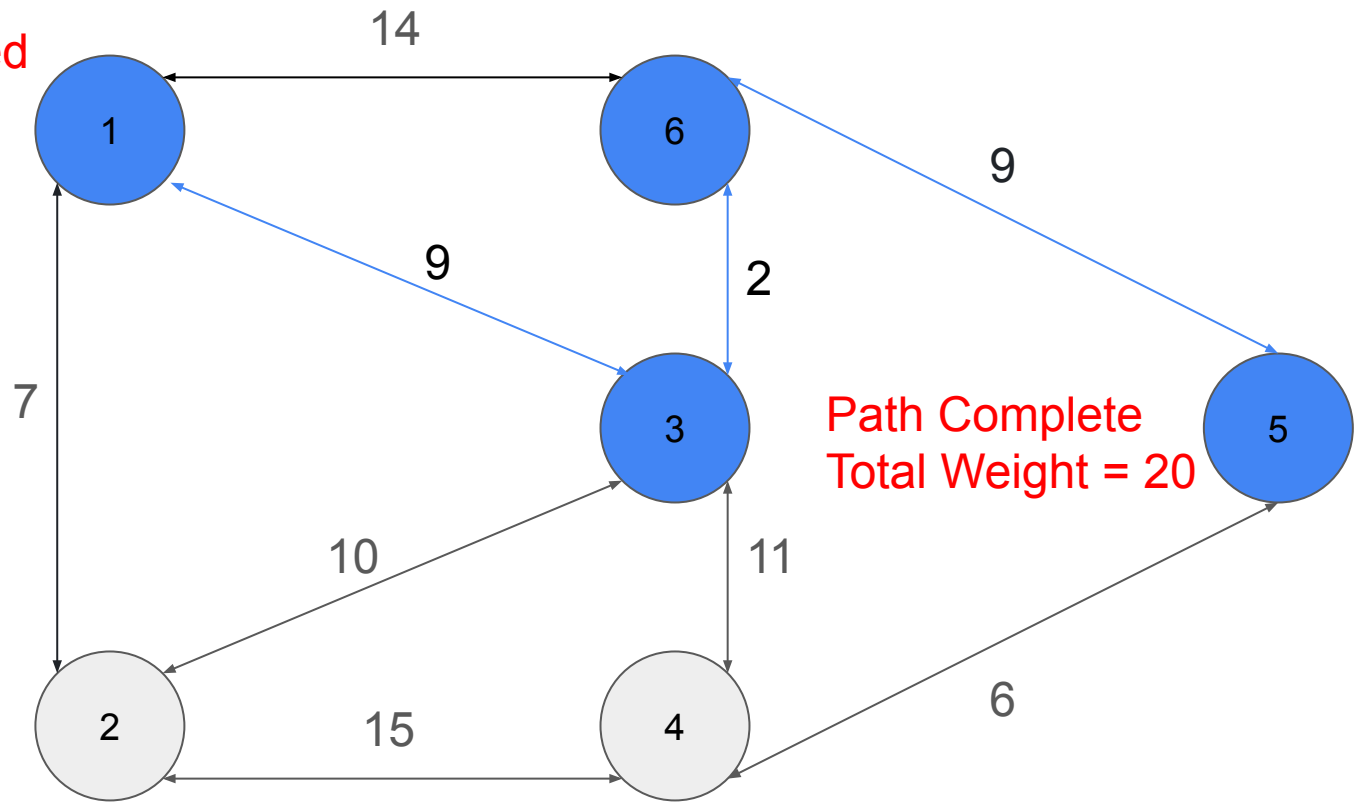


Demo: Graph 1 (Shortest path 1 to 5)

Visited:
{1,3,6,**5**}← Path Used

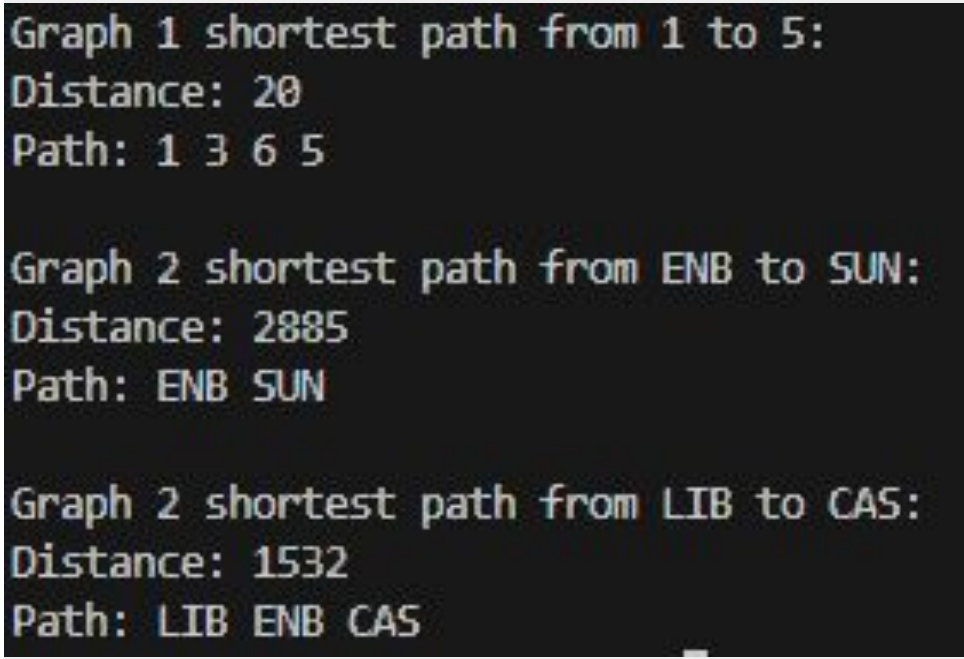
Unvisited:
{2,4}

Node	Distance	Previous
1	0	
2	7	1
3	9	1
4	20	3
5	20	6
6	11	3



Output of Code

The output of out code matches the demo for Graph 1



Thank you!

ANY QUESTIONS?