Learning Objectives

Learners will be able to…

- **Define the shortest path in a graph (weighted and unweighted)**
- **Implement shortest path with BFS, Dijkstra’s algorithm, and Bellman-Ford algorithm**
- **Analyze complexity and performance of the shortest path algorithms**

Shortest Path Fundamentals in Graph Theory

What is a Shortest Path?

Understanding the fundamentals of shortest paths in graph theory is crucial for solving a wide range of problems in computer science and related fields. This section introduces the basic concepts and principles underlying the shortest path problem in graphs.

In graph theory, the shortest path between two vertices (or nodes) is the path that has the least total weight (or cost, length, etc.) compared to all other possible paths between these vertices. The weight of a path is typically the sum of the weights of its constituent edges.

Key Concepts

- **Weighted vs. Unweighted Graphs**:
  - **Unweighted Graphs**: The shortest path is the path with the fewest edges.
  - **Weighted Graphs**: Each edge has a weight, and the shortest path minimizes the total weight of the edges.
- **Directed vs. Undirected Graphs**:
  - The path's direction matters in directed graphs, while any path is viable in undirected graphs as long as it connects the start and end vertices.
- **Positive vs. Negative Weights**:
  - Most shortest path algorithms assume positive weights, but some can handle negative weights as well.
- **Single-Source vs. All-Pairs Shortest Path**:
  - **Single-Source**: Finding the shortest paths from a single source vertex to all other vertices in the graph.
  - **All-Pairs**: Finding the shortest paths between all pairs of vertices in the graph.

Applications

- **Network Routing**: Optimizing the paths that data packets take across a network.
- **GIS and Map Services**: Calculating the quickest or shortest route for navigation and travel planning.
- **Resource Allocation and Logistics**: Determining optimal paths for transportation and distribution in supply chain management.
- **Social Network Analysis**: Identifying the shortest path can help

understand relationships and influences within a social network.

Basic Approach: Brute-Force Method

A brute-force method to find the shortest path involves checking all possible paths between the source and destination vertices and selecting the path with the minimum total weight or edge count. While this method is straightforward, it is highly inefficient, especially for large graphs, as the number of possible paths can grow exponentially.

Preliminary Algorithm Concepts

- **Relaxation**: A key concept in shortest path algorithms where an estimated shortest path is gradually replaced by more accurate values until the shortest path is found.
- **Edge Relaxation**: Involves checking if the known distance to a vertex can be improved by taking a new edge.
- **Path Reconstruction**: Keeping track of the sequence of vertices and edges that make up the shortest path, often using predecessor pointers.

Challenges and Considerations

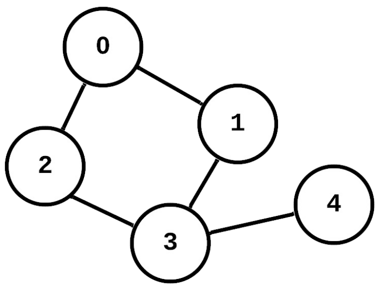
- **Negative Weight Cycles**: The presence of negative weight cycles in a graph can make it impossible to find a shortest path, as the total weight can be indefinitely reduced by repeatedly traversing the cycle.
- **Algorithm Selection**: The choice of algorithm depends on the graph's characteristics (e.g., weighted/unweighted, directed/undirected) and the problem's requirements (e.g., single-source vs. all-pairs).

Understanding these fundamentals sets a strong foundation for moving into more advanced shortest path algorithms like Dijkstra's and Bellman-Ford. These algorithms build upon these basic principles and offer more efficient solutions for different types of graphs and problem scenarios.

Basic Shortest Path

Shortest Path: Undirected and Unweighted

Finding the shortest path in an undirected, unweighted graph can be efficiently achieved using Breadth-First Search (BFS). BFS is particularly suitable for this task because it explores the graph level by level, ensuring that the first time a node is reached is via the shortest path. We are going to assume the following graph for this example:



Let's break down the implementation of this approach. We are going to use a few different data structures to help with this. First, use a queue to store the vertices in each level. We are then also going to keep track of the immediate predecessor in a map. The third data structure is a set which keeps track of the visited vertices. Finally, we will reconstruct the shortest path.

Defining the Graph Class

We will start by creating a basic unweighted and undirected `Graph` class that uses an adjacency list for storing the graph's structure. The adjacency list is a map where each key is a vertex, and the value is a list of adjacent vertices. We need the `addVertex` and `addEdge` methods. We are not going to add any other graph operations for this example.

```
#include <iostream>
#include <unordered_map>
#include <list>
#include <queue>
#include <unordered_set>
#include <algorithm>
#include <vector>

class Graph {
private:
    std::unordered_map<int, std::list<int>> adjList;

public:
    // Method to add a vertex
    void addVertex(int vertex) {
        adjList[vertex];  // Ensures the vertex is added
    }

    // Method to add an edge
    void addEdge(int vertex1, int vertex2) {
        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }

    // Other methods...
};
```

Implementing BFS for Shortest Path

To find the shortest path, we'll implement a `BFS` that not only traverses the graph but also keeps track of each vertex's predecessor. This information allows us to reconstruct the shortest path once the destination is reached. Start by creating the `findShortestPath` method. This public method takes two integers representing the starting and ending `vertices` in a graph. The method returns a vector of integers. Create the variable queue which is a queue used for `BFS` traversal. The predecessors variable is an `unordered_map` that stores the immediate predecessor of a vertex. Finally, visited is a set that contains all of the `vertices` visited. Add the starting vertex to each of these three data structures.

```
    std::vector<int> findShortestPath(int startVertex, int
        endVertex) {
        std::queue<int> queue;
        std::unordered_map<int, int> predecessors;
        std::unordered_set<int> visited;

        queue.push(startVertex);
        visited.insert(startVertex);
        predecessors[startVertex] = -1;  // Start vertex has no
        predecessor
```

Next, use a while loop to iterate over the `vertices` in the queue. Set `currentVertex` to the first vertex out of the queue. If the current vertex happens to be the ending vertex, return a call to the `constructPath` method and apply it to the ending vertex and the map of predecessors.

```
        while (!queue.empty()) {
            int currentVertex = queue.front();
            queue.pop();

            if (currentVertex == endVertex) {
                return constructPath(endVertex, predecessors);
        // Construct and return the path
            }
```

Otherwise, find the `list` of neighbors for a given `vertex`. Iterate over the list of neighbors. If we have not already `visited` the neighbor, add it to `queue`, `visited`, and `predecessors`. If the program is still running after the conclusion of the for and while loops, return an `empty` vector. This represents that a shortest path does not exist between the `two` vertices.

```
            for (int neighbor : adjList[currentVertex]) {
                if (visited.find(neighbor) == visited.end()) {
                    queue.push(neighbor);
                    visited.insert(neighbor);
                    predecessors[neighbor] = currentVertex;
                }
            }
        }
        return {};
    }
```

We also need to create the private helper method `constructPath`. This method takes an integer representing the ending `index` and a `map` of the predecessors. The method returns a vector of integers representing the path of `vertices` in the shortest path. Start by creating a vector of integers called path. Iterate over the `map` of predecessors, adding the `vertices` in the map to the path list. Reverse the list to get the correct order and return path.

```
private:
    std::vector<int> constructPath(int endVertex,
        std::unordered_map<int, int>& predecessors) {
        std::vector<int> path;
        for (int vertex = endVertex; vertex != -1; vertex =
        predecessors[vertex]) {
            path.push_back(vertex);
        }
        std::reverse(path.begin(), path.end()); // Reverse to
        get the correct order
        return path;
    }
```

Testing the Code

To test our code, create the main method. Instantiate a Graph object and add the vertices `0`, `1`, `2`, `3`, and `4`. Then add edges between `0` and `1`, `0` and `2`, `1` and `3`, `2` and `3`, and `3` and `4`. Then calculate the shortest distance between vertex `0` and vertex `4`.

```cpp
int main() {
    Graph graph;

    // Add vertices
    graph.addVertex(0);
    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);
    graph.addVertex(4);

    // Add edges
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 3);
    graph.addEdge(3, 4);

    // Find the shortest path from 0 to 4
    std::vector<int> shortestPath = graph.findShortestPath(0,
        4);
    std::cout << "Shortest path from 0 to 4: ";
    for (int vertex : shortestPath) {
        std::cout << vertex << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

▼ **Code**

> Your code should look like this:
>
> ```cpp
> #include <iostream>
> #include <unordered_map>
> #include <list>
> #include <queue>
> #include <unordered_set>
> #include <algorithm>
> #include <vector>
>
> class Graph {
> private:
>     std::unordered_map<int, std::list<int>> adjList;
>
> public:
>     void addVertex(int vertex) {
>         adjList[vertex];
> ```

```
    }

    void addEdge(int vertex1, int vertex2) {
        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1);
    }

    std::vector<int> findShortestPath(int startVertex, int
        endVertex) {
        std::queue<int> queue;
        std::unordered_map<int, int> predecessors;
        std::unordered_set<int> visited;

        queue.push(startVertex);
        visited.insert(startVertex);
        predecessors[startVertex] = -1;

        while (!queue.empty()) {
            int currentVertex = queue.front();
            queue.pop();

            if (currentVertex == endVertex) {
                return constructPath(endVertex,
        predecessors);
            }

            for (int neighbor : adjList[currentVertex]) {
                if (visited.find(neighbor) ==
        visited.end()) {
                    queue.push(neighbor);
                    visited.insert(neighbor);
                    predecessors[neighbor] = currentVertex;
                }
            }
        }
        return {};
    }

private:
    std::vector<int> constructPath(int endVertex,
        std::unordered_map<int, int>& predecessors) {
        std::vector<int> path;
        for (int vertex = endVertex; vertex != -1; vertex =
        predecessors[vertex]) {
            path.push_back(vertex);
        }
        std::reverse(path.begin(), path.end());
        return path;
    }
};

int main() {
    Graph graph;

    graph.addVertex(0);
    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);
    graph.addVertex(4);
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 3);
    graph.addEdge(3, 4);

    std::vector<int> shortestPath =
        graph.findShortestPath(0, 4);
    std::cout << "Shortest path from 0 to 4: ";
    for (int vertex : shortestPath) {
        std::cout << vertex << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

You should see the following output:

```
Shortest path from 0 to 4: 0 1 3 4
```

Optimizing BSF Shortest Path

Optimizing Shortest Path in Undirected Graphs

While the Breadth-First Search (BFS) approach is effective for unweighted graphs, its efficiency can be significantly enhanced for graphs with a large number of vertices and edges. In this section, we delve into an optimized method for implementing the shortest path algorithm in undirected graphs, utilizing priority queues to streamline the order of vertex processing.

Leveraging Priority Queues for BFS

Incorporating a priority queue into the BFS process allows us to prioritize processing the vertex nearest to the source. This strategy is particularly advantageous in weighted graphs, ensuring a more efficient pathfinding process. Even in unweighted graphs, employing a priority queue can optimize BFS by minimizing the number of vertices processed.

We will modify our `Graph` class to integrate a priority queue into the `BFS` routine. Although our graph remains unweighted, this adjustment lays the groundwork for handling more intricate weighted graphs.

Instead of implementing the `queue` with a standard queue, we are going to replace it with a `priority_queue` and a custom comparator. You only need to change this one line of code in the editor. Everything else can remain unchanged.

```
// BFS using a priority queue
std::vector<int> findShortestPath(int startVertex, int
    endVertex) {
    std::priority_queue<int, std::vector<int>,
    std::greater<int>> queue;
    std::unordered_map<int, int> predecessors;
    std::unordered_set<int> visited;

    // The rest of the code remains unchanged
}
```

▼ **Code**

> Your code should look like this:
>
> ```
> #include <iostream>
> #include <unordered_map>
> #include <list>
> ```

```cpp
#include <queue>
#include <unordered_set>
#include <algorithm>
#include <vector>

class Graph {
private:
    std::unordered_map<int, std::list<int>> adjList;

public:
    // Method to add a vertex
    void addVertex(int vertex) {
        adjList[vertex];  // Ensures the vertex is added
    }

    // Method to add an edge
    void addEdge(int vertex1, int vertex2) {
        adjList[vertex1].push_back(vertex2);
        adjList[vertex2].push_back(vertex1); // Since the
        graph is undirected
    }

    // BFS using a priority queue
    std::vector<int> findShortestPath(int startVertex, int
        endVertex) {
        std::priority_queue<int, std::vector<int>,
        std::greater<int>> queue;
        std::unordered_map<int, int> predecessors;
        std::unordered_set<int> visited;

        queue.push(startVertex);
        visited.insert(startVertex);
        predecessors[startVertex] = -1;

        while (!queue.empty()) {
            int currentVertex = queue.top();
            queue.pop();

            if (currentVertex == endVertex) {
                return constructPath(endVertex,
        predecessors); // Construct and return the path
            }

            for (int neighbor : adjList[currentVertex]) {
                if (visited.find(neighbor) ==
        visited.end()) {
                    queue.push(neighbor);
                    visited.insert(neighbor);
                    predecessors[neighbor] = currentVertex;
                }
            }
        }
        return {}; // Return empty vector if no path is
        found
    }

private:
    // Helper method to backtrack and construct the path
    std::vector<int> constructPath(int endVertex,
        std::unordered_map<int, int>& predecessors) {
        std::vector<int> path;
        for (int vertex = endVertex; vertex != -1; vertex =
        predecessors[vertex]) {
            path.push_back(vertex);
        }
        std::reverse(path.begin(), path.end()); // Reverse
        to get the correct order
        return path;
    }
};

int main() {
    Graph graph;

    // Add vertices
    graph.addVertex(0);
    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);
    graph.addVertex(4);

    // Add edges
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 3);
    graph.addEdge(3, 4);

    // Find the shortest path from 0 to 4
    std::vector<int> shortestPath =
        graph.findShortestPath(0, 4);
    std::cout << "Shortest path from 0 to 4: ";
    for (int vertex : shortestPath) {
        std::cout << vertex << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

```
Shortest path from 0 to 4: 0 1 3 4
```

The `priority_queue` used here is a `min-heap` because of the `std::greater<int>` comparator. It ensures that the vertex with the smallest value is processed first. Processing `vertices` with the smallest values first streamlines the search for the shortest path.

Using a priority queue in `BFS` for shortest path discovery serves as a precursor to more sophisticated and efficient algorithms like Dijkstra's algorithm. This approach is especially beneficial in weighted graphs, where selecting the next vertex based on the shortest known distance to the source guarantees both accuracy and efficiency in the algorithm.

Dijkstra’s Algorithm

Navigating Weighted Graphs

In weighted graphs, the shortest path between two vertices is determined not by the number of edges, but by the minimal sum of the edge weights. To discover the shortest path in such graphs, we adopt a strategy that accounts for the weights of the edges.

For this section, let’s focus on Dijkstra’s algorithm, an efficient technique for finding the shortest path in graphs with non-negative edge weights. The algorithm was conceived by computer scientist Edsger W. Dijkstra in 1956 and remains a popular solution for solving shortest path problems in computing. This algorithm is a cornerstone in graph theory and computer science for solving single-source shortest path problems efficiently.

What is Dijkstra’s Algorithm?

Dijkstra’s algorithm finds the shortest path from a starting node to all other nodes in a graph with **non-negative edge weights**. The essence of the algorithm is to repeatedly select the nearest vertex that has not been processed yet and update the path lengths to its neighbors if a shorter path is found. The algorithm finishes when all nodes have been processed.

How Does it Work?

The algorithm maintains a set of nodes whose shortest distance from the start node is already known and a set of nodes whose shortest distance is not yet determined. Initially, the distance to the start node is zero, and the distances to all other nodes are set to infinity. The algorithm proceeds by selecting the unvisited node with the smallest distance, calculating the distance through it to each unvisited neighbor, and updating the neighbor’s distance if it is smaller. This process repeats until all nodes have been visited.

Key Concepts

Initialization: Set the initial node distance to zero and all others to infinity.

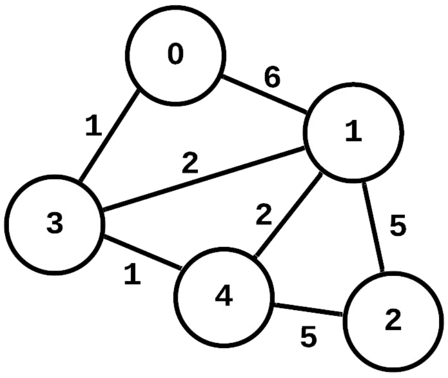
Selection: At each step, select the unvisited node with the smallest known distance.

Relaxation: Update the neighboring nodes’ distances if a shorter path is found through the selected node.

Termination: The algorithm terminates when every node has been visited.

Implementing Dijkstra’s Algorithm

We are going to use the weighted, undirected graph below for our example. In addition, we are going to use an adjacency matrix to represent the edges and weights between vertices.



Start by creating the Node class. This class will help us manage the priority queue. The Node class has two integer attributes, vertex and weight. The class implements the operator< to order the Node objects by their weight attribute.

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>

class Node {
public:
    int vertex, weight;

    Node(int v, int w) : vertex(v), weight(w) {}

    bool operator<(const Node& other) const {
        return weight > other.weight; // Min-heap priority queue
    }
};
```

We are going to start with a helper function. printSolution takes a vector of integers representing the shortest distances between the source vertex and the other vertices. Iterate over the vector, printing the index and the element. The variable i represents the vertex in the graph, and distances[i] represents the distance from that vertex to the source vertex.

```
void printSolution(const std::vector<int>& distances) {
    std::cout << "Vertex\t| Distance from Source" << std::endl;
    for (int i = 0; i < distances.size(); i++) {
        std::cout << i << "\t|\t" << (distances[i] == INT_MAX ?
        "INF" : std::to_string(distances[i])) << std::endl;
    }
}
```

The `dijkstra` function contains the actual algorithm to find the shortest path. Create a priority queue to store nodes. We also need a vector for distances and a vector for visited nodes.

Fill the `distances` vector with the largest value possible for an integer. Then set the distance for the source vertex to `0`. Finally, create a `Node` object with the source vertex and a weight of `0`, and add this to the priority queue.

```
void dijkstra(const std::vector<std::vector<int>>& graph, int
        source) {
    std::priority_queue<Node> queue;
    std::vector<int> distances(graph.size(), INT_MAX);
    std::vector<bool> visited(graph.size(), false);

    distances[source] = 0;
    queue.push(Node(source, 0));
```

The algorithm continues by iterating as long as the queue is not empty. Extract the `Node` object from the queue and get the `vertex` attribute. Check to see if the current vertex has already been visited by the algorithm. If not, mark it as visited.

```
    while (!queue.empty()) {
        Node current = queue.top();
        queue.pop();
        int vertex = current.vertex;

        if (visited[vertex]) continue;
        visited[vertex] = true;
```

Create a for loop that iterates for the length of the graph. The expression `graph[vertex][i]` represents the weighted value (position in the adjacency matrix) between the current vertex (`vertex`) and the vertex `i`. Check if the next position in the adjacency matrix is a neighbor (`graph[vertex][i] > 0`) and that vertex `i` has not been visited. If so, calculate a new distance to reach the neighbor. If the new distance is less than the distance currently stored in `distances[i]`, replace the old distance with the new distance. Then add a `Node` object with the vertex `i` and the weight `distances[i]` to the priority queue. After the for and while loops have ended, call the `printSolution` method and pass it the `distances` vector.

```
        for (int i = 0; i < graph.size(); i++) {
            if (graph[vertex][i] > 0 && !visited[i]) {
                int newDist = distances[vertex] + graph[vertex]
[i];
                if (newDist < distances[i]) {
                    distances[i] = newDist;
                    queue.push(Node(i, distances[i]));
                }
            }
        }
    }

    printSolution(distances);
}
```

Testing the Code

To test the algorithm, create the `main` method. Create the `source` variable and set it to `0`. Create a 2D vector of integers representing the adjacency matrix of the graph. Pass the source vertex and adjacency matrix to the `dijkstra` function.

```
int main() {
    int source = 0;
    std::vector<std::vector<int>> graph = {
        {0, 6, 0, 1, 0},
        {6, 0, 5, 2, 2},
        {0, 5, 0, 0, 5},
        {1, 2, 0, 0, 1},
        {0, 2, 5, 1, 0}
    };

    dijkstra(graph, source); // 0 is the source node
    return 0;
}
```

▼ **Code**

> Your code should look like this:
>
> ```
> #include <iostream>
> #include <vector>
> #include <queue>
> #include <climits>
>
> class Node {
> public:
>     int vertex, weight;
>
>     Node(int v, int w) : vertex(v), weight(w) {}
>
>     bool operator<(const Node& other) const {
> ```

```
        return weight > other.weight; // Min-heap priority
        queue
    }
};

void printSolution(const std::vector<int>& distances) {
    std::cout << "Vertex\t| Distance from Source" <<
        std::endl;
    for (int i = 0; i < distances.size(); i++) {
        std::cout << i << "\t|\t" << (distances[i] ==
        INT_MAX ? "INF" : std::to_string(distances[i])) <<
        std::endl;
    }
}

void dijkstra(const std::vector<std::vector<int>>& graph,
        int source) {
    std::priority_queue<Node> queue;
    std::vector<int> distances(graph.size(), INT_MAX);
    std::vector<bool> visited(graph.size(), false);

    distances[source] = 0;
    queue.push(Node(source, 0));

    while (!queue.empty()) {
        Node current = queue.top();
        queue.pop();
        int vertex = current.vertex;

        if (visited[vertex]) continue;
        visited[vertex] = true;

        for (int i = 0; i < graph.size(); i++) {
            if (graph[vertex][i] > 0 && !visited[i]) {
                int newDist = distances[vertex] +
        graph[vertex][i];
                if (newDist < distances[i]) {
                    distances[i] = newDist;
                    queue.push(Node(i, distances[i]));
                }
            }
        }
    }

    printSolution(distances);
}

int main() {
    int source = 0;
    std::vector<std::vector<int>> graph = {
        {0, 6, 0, 1, 0},
        {6, 0, 5, 2, 2},
        {0, 5, 0, 0, 5},
        {1, 2, 0, 0, 1},
        {0, 2, 5, 1, 0}
    };

    dijkstra(graph, source); // 0 is the source node
    return 0;
```

```
}
```

You should see the following output:

```
Vertex  | Distance from Source
0       |       0
1       |       3
2       |       7
3       |       1
4       |       2
```

Dijkstra’s algorithm is efficient for finding the shortest path in graphs with non-negative weights. The use of a priority queue makes selecting the minimum distance more efficient.

Understanding the algorithm’s operation, including initialization, selection, relaxation, and termination, is crucial for implementation and modification for specific needs.

Dijkstra’s algorithm plays a vital role in various applications, from network routing protocols to mapping and navigation services, showcasing its versatility and importance in computer science and operations research.

Complexity of Dijkstra’s Algorithm

Analyzing the Complexity of Dijkstra’s Algorithm

Dijkstra’s algorithm is a powerful tool for finding the shortest path in a graph. However, understanding its computational complexity is crucial for evaluating its efficiency and suitability for various applications. The time complexity of Dijkstra’s algorithm depends on how it is implemented, particularly in terms of the data structures used for storing the graph and managing the priority queue.

Basic Implementation

In a basic implementation where the graph is stored as an adjacency matrix and a simple linear array is used to find the vertex with the minimum distance, the time complexity is $O(V^2)$, where V is the number of vertices in the graph. This is because, for each vertex, the algorithm scans all vertices to find the one with the minimum distance that has not been visited yet.

Priority Queue Implementation

A more efficient implementation uses a priority queue to manage the vertices, reducing the time it takes to find the vertex with the minimum distance. The time complexity in this case depends on the type of priority queue used:

- **Binary Heap**: With a binary heap, the time complexity is $O((V + E) logV)$, where E is the number of edges in the graph. This is because each vertex is inserted and extracted from the priority queue once, which takes $O(logV)$ time, and each edge’s relaxation operation takes $O(logV)$ time.
- **Fibonacci Heap**: Using a Fibonacci heap further improves the time complexity to $O(V logV + E)$. This is because the amortized time for decrease-key operations in a Fibonacci heap is $O(1)$, which is ideal for the edge relaxation process.

Space Complexity

The space complexity of Dijkstra’s algorithm is $O(V)$ for storing the distance array and $O(V)$ for the priority queue, resulting in a total space complexity of $O(V)$.

Key Points to Remember

- The time complexity of Dijkstra's algorithm varies depending on the implementation, with $O(V^2)$ for a basic implementation and $O((V + E) log V)$ or $O(V log V + E)$ for implementations using binary or Fibonacci heaps, respectively.
- The space complexity of the algorithm is $O(V)$, accounting for the storage of the distance array and the priority queue.
- Choosing the right data structure for the priority queue is crucial for optimizing the algorithm's performance, especially for graphs with a large number of vertices and edges.

Understanding the computational complexity of Dijkstra's algorithm is essential for assessing its performance and selecting the most appropriate implementation for a given problem.

Bellman-Ford Algorithm

Handling Graphs with Negative Weights

The Bellman-Ford algorithm is another important algorithm for finding the shortest path in a graph. Unlike Dijkstra’s algorithm, which is limited to graphs with non-negative edge weights, the Bellman-Ford algorithm can handle graphs with negative edge weights, making it more versatile in certain situations.

What is the Bellman-Ford Algorithm?

The Bellman-Ford algorithm calculates the shortest paths from a single source vertex to all other vertices in a weighted graph. It can detect and report the presence of negative cycles in the graph, which are cycles with a total negative edge weight. This is a crucial feature, as negative cycles can lead to infinitely decreasing path lengths, making the shortest path undefined. In Dijkstra’s algorithm, when you encounter a node that you have already seen and its weight is already higher than previously seen, you could just stop and move on to the next node. Because of the added negative weight, the shortest path in Bellman-Ford could still be present.

How Does it Work?

The algorithm works by iteratively relaxing the edges of the graph. It repeatedly updates the distance to each vertex from the source vertex, based on the edge weights. The key idea is that after $V - 1$ iterations, where V is the number of vertices in the graph, the algorithm will have found the shortest paths to all vertices, unless there is a negative cycle.

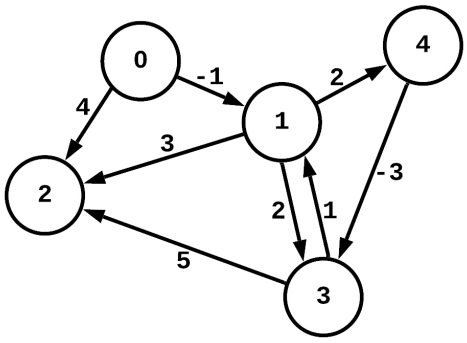
Key Concepts

Relaxation: The process of updating the distance to a vertex if a shorter path is found through an edge.

Negative Cycle Detection: The algorithm can detect negative cycles by checking for further distance improvements after $V - 1$ iterations. If any distances can be reduced, a negative cycle exists in the graph.

Implementing the Bellman-Ford Algorithm

We are going to use the weighted, directed graph below for our example. In addition, we are going to define the Edge class that contains the direction and weight of a connection between vertices.



Start by creating the `Edge` class, which has three integer attributes: `source`, `destination`, and `weight`. In the constructor, set each of the attributes with the correct argument. Direction goes from the `source` vertex to the `destination` vertex.

```
#include <iostream>
#include <limits>
#include <cstring> // for std::fill
#include <climits>

class Edge {
public:
    int source, destination, weight;

    Edge(int s, int d, int w) : source(s), destination(d),
        weight(w) {}
};
```

The `printSolution` helper function that takes a pointer to an array of integers representing the shortest distances between the source vertex and the other vertices. Iterate over the array, printing the index and the element. The variable `i` represents the vertex in the graph and `distances[i]` represents the distance from that vertex to the source vertex.

```
void printSolution(int* distances, int vertices) {
    std::cout << "Vertex\t| Distance from Source" << std::endl;
    for (int i = 0; i < vertices; i++) {
        std::cout << i << "\t|\t" << (distances[i] == INT_MAX ?
        "INF" : std::to_string(distances[i])) << std::endl;
    }
}
```

Next, start the `bellmanFord` function which takes a pointer to an array of `Edge` objects, the number of vertices, and the source vertex. Create an array of integers called `distances`. Set its length to the number of vertices. This array contains the shortest distance between each vertex and the source. Fill the array with the largest integer value. Then set the distance for the source vertex to `0`.

```
void bellmanFord(Edge* edges, int edgeCount, int vertices, int
        source) {
    int* distances = new int[vertices];
    std::fill(distances, distances + vertices, INT_MAX);
    distances[source] = 0;
```

Then create a for loop that starts iterating at `1` instead of `0`. Keep iterating as long as the iteration variable is less than the number of vertices. Create a second loop that iterates over the array of `Edge` objects. Check to see if the distance for the source vertex in the edge is not the maximum value. Also check if the distance for the source vertex plus the weight of the edge is less than the distance of the destination vertex. If so, update the distance for the destination vertex to be the source distance plus the edge weight.

```
    for (int i = 1; i < vertices; i++) {
        for (int j = 0; j < edgeCount; j++) {
            Edge edge = edges[j];
            if (distances[edge.source] != INT_MAX &&
                distances[edge.source] + edge.weight <
        distances[edge.destination]) {
                distances[edge.destination] =
        distances[edge.source] + edge.weight;
            }
        }
    }
```

We also need to check for a negative cycle. Iterate over the array of Edge objects. Check if the distance for the source vertex is not equal to the largest integer value. Also check if the distance of the source vertex plus the edge weight is less than the distance of the destination vertex. If both of these things are true, print a message that the graph contains a negative cycle and exit the method. If there are no negative cycles, call the `printSolution` method and pass it the array of distances.

```
    // Check for negative cycles
    for (int j = 0; j < edgeCount; j++) {
        Edge edge = edges[j];
        if (distances[edge.source] != INT_MAX &&
            distances[edge.source] + edge.weight <
        distances[edge.destination]) {
            std::cout << "Graph contains a negative cycle." <<
        std::endl;
            delete[] distances;
            return;
        }
    }

    printSolution(distances, vertices);
    delete[] distances;
}
```

Testing the Code

To test the code, create the main method. We are going to create a graph with five vertices and a source of vertex `0`. Create the array `edges` that contains eight `Edge` objects. Then pass this array to the `bellmanFord` function.

```
int main() {
    int vertices = 5;
    int edgeCount = 8;
    int source = 0;
    Edge edges[] = {
        Edge(0, 1, -1),
        Edge(0, 2, 4),
        Edge(1, 2, 3),
        Edge(1, 3, 2),
        Edge(1, 4, 2),
        Edge(3, 2, 5),
        Edge(3, 1, 1),
        Edge(4, 3, -3)
    };

    bellmanFord(edges, edgeCount, vertices, source);
    return 0;
}
```

▼ **Code**

> Your code should look like this:
>
> ```
> #include <iostream>
> #include <limits>
> #include <cstring> // for std::fill
>
> class Edge {
> public:
>     int source, destination, weight;
>
>     Edge(int s, int d, int w) : source(s), destination(d),
>         weight(w) {}
> };
>
> void printSolution(int* distances, int vertices) {
>     std::cout << "Vertex\t| Distance from Source" <<
>         std::endl;
>     for (int i = 0; i < vertices; i++) {
>         std::cout << i << "\t|\t" << (distances[i] ==
>         std::numeric_limits<int>::max() ? "INF" :
>         std::to_string(distances[i])) << std::endl;
>     }
> }
>
> void bellmanFord(Edge* edges, int edgeCount, int vertices,
>         int source) {
>     int* distances = new int[vertices];
> ```

```
    std::fill(distances, distances + vertices,
        std::numeric_limits<int>::max());
    distances[source] = 0;

    for (int i = 1; i < vertices; i++) {
        for (int j = 0; j < edgeCount; j++) {
            Edge edge = edges[j];
            if (distances[edge.source] !=
        std::numeric_limits<int>::max() &&
                distances[edge.source] + edge.weight <
        distances[edge.destination]) {
                distances[edge.destination] =
        distances[edge.source] + edge.weight;
            }
        }
    }

    // Check for negative cycles
    for (int j = 0; j < edgeCount; j++) {
        Edge edge = edges[j];
        if (distances[edge.source] !=
        std::numeric_limits<int>::max() &&
            distances[edge.source] + edge.weight <
        distances[edge.destination]) {
            std::cout << "Graph contains a negative cycle."
        << std::endl;
            delete[] distances;
            return;
        }
    }

    printSolution(distances, vertices);
    delete[] distances;
}

int main() {
    int vertices = 5;
    int edgeCount = 8;
    int source = 0;
    Edge edges[] = {
        Edge(0, 1, -1),
        Edge(0, 2, 4),
        Edge(1, 2, 3),
        Edge(1, 3, 2),
        Edge(1, 4, 2),
        Edge(3, 2, 5),
        Edge(3, 1, 1),
        Edge(4, 3, -3)
    };

    bellmanFord(edges, edgeCount, vertices, source);
    return 0;

}
```

You should see the following output:

```
Vertex    | Distance from Source
0         |       0
1         |       -1
2         |       2
3         |       -2
4         |       1
```

Key Points to Remember

- The Bellman-Ford algorithm can handle graphs with negative edge weights and can detect negative cycles.
- The time complexity of the algorithm is $O(V * E)$, where V is the number of vertices and E is the number of edges in the graph.
- The space complexity is $O(V)$ for storing the distance array.
- The algorithm is particularly useful for applications that require the handling of negative edge weights or the detection of negative cycles, such as certain types of network routing protocols or financial models.

Understanding the Bellman-Ford algorithm and its characteristics is essential for addressing a wide range of shortest path problems, especially in scenarios where negative edge weights are involved or when detecting negative cycles is important.

Formative Assessment 1

Formative Assessment 2