Learning Objectives

Learners will be able to…

- **Define a spanning tree and minimum spanning tree**
- **Implement the Kruskal's and Prim's algorithms in C++**
- **Analyze these minimum spanning tree algorithms**

Introduction to Spanning Trees

Spanning Trees

Spanning trees are a fundamental concept in graph theory with significant applications in computer science, especially in the fields of network design and optimization. A spanning tree of a graph is a tree that connects all the vertices together without any cycles and includes all the vertices of the graph.

Definition of a Spanning Tree

A spanning tree is defined for an undirected graph. It is a subgraph that includes all the vertices of the original graph. The key characteristics of a spanning tree are:

- It is a tree, which means it is a connected graph with no cycles.
- It spans all the vertices of the graph, meaning every vertex in the original graph is included in the spanning tree. If a vertex is missed, then it is not a spanning tree.
- It has exactly $V - 1$ edges, where V is the number of vertices in the graph. This is the minimum number of edges needed to connect all the vertices without forming a cycle.

Why is it Called a Spanning Tree?

Remember when we said “Every tree is a graph, but not every graph is a tree”. The term “spanning tree” comes from the fact that this tree “spans” all the vertices of the graph. In other words, it covers all the vertices, ensuring that there is a path between every pair of vertices, just like a tree in a forest connects all its branches and leaves. The spanning tree is a minimalistic structure that achieves this connectivity without any redundant paths, which would form cycles.

Importance in Undirected Graphs

Spanning trees are particularly important in the context of undirected graphs. In an undirected graph, edges have no direction, meaning the graph represents a two-way relationship between vertices. Spanning trees help in organizing these undirected graphs in a way that maintains connectivity while eliminating any cycles, making the structure simpler and easier to analyze.

In summary, spanning trees are a crucial concept in graph theory, providing a simplified, cycle-free structure that ensures connectivity among all vertices of an undirected graph. Their properties and

applications make them an essential topic of study in computer science and related fields.

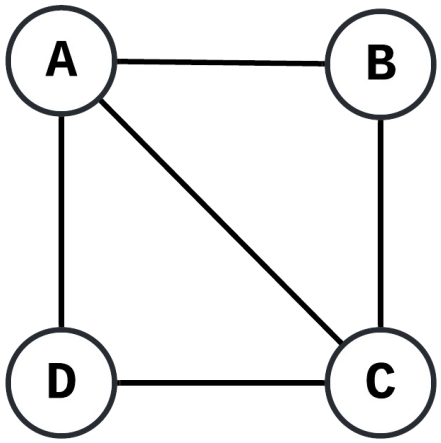
Spanning Tree Examples

Visualizing Spanning Trees

To understand spanning trees better, let's consider a simple undirected graph and explore its possible spanning trees. This visual example will help illustrate how a spanning tree connects all the vertices of a graph without forming any cycles.

Original Graph

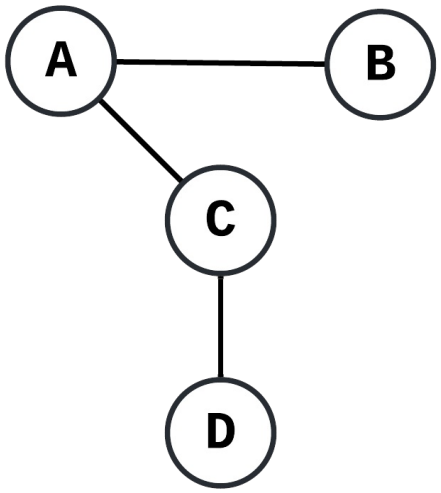
Consider the following undirected graph with vertices A, B, C, and D, and edges AB, BC, CD, DA, and AC:



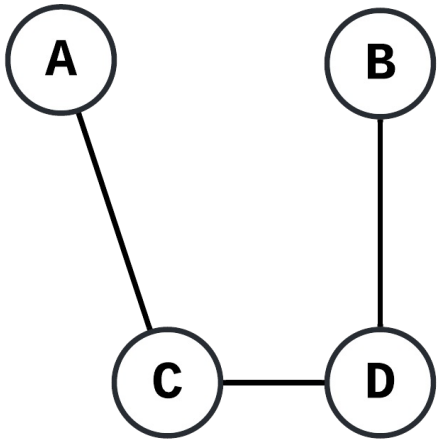
Possible Spanning Trees

From this graph, we can derive several spanning trees by ensuring that all vertices are connected without any cycles. Here are three possible spanning trees:

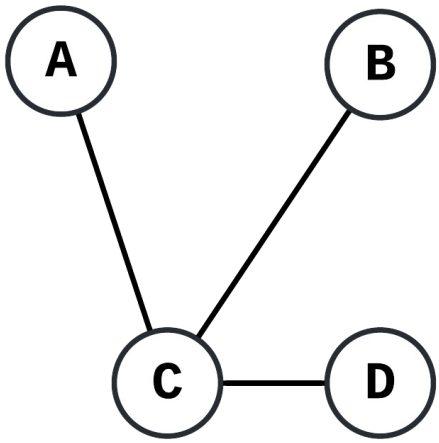
1. **Spanning Tree 1**:
   Edges: AB, AC, CD



2. **Spanning Tree 2**:
   Edges: AC, BD, CD



3. **Spanning Tree 3**:
   Edges: AC, BC, CD



Observations

- Each spanning tree includes all the vertices (A, B, C, D).
- Each spanning tree has exactly $V - 1$ edges, where V is the number of vertices. In this case, each tree has 3 edges.
- There are no cycles in any of the spanning trees.

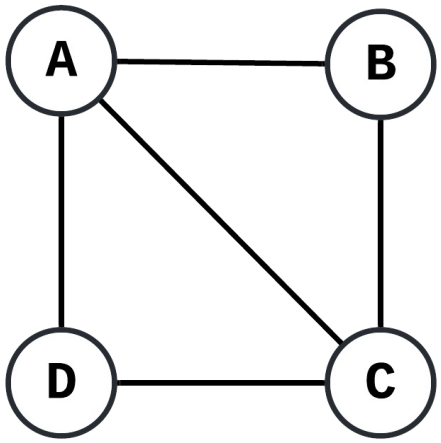
Applications of Spanning Trees

Visualizing Spanning Trees

To understand spanning trees better, let's consider a simple undirected graph and explore its possible spanning trees. This visual example will help illustrate how a spanning tree connects all the vertices of a graph without forming any cycles.

Original Graph

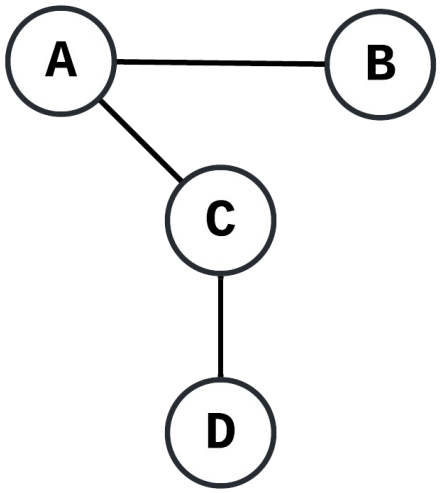
Consider the following undirected graph with vertices A, B, C, and D, and edges AB, BC, CD, DA, and AC:



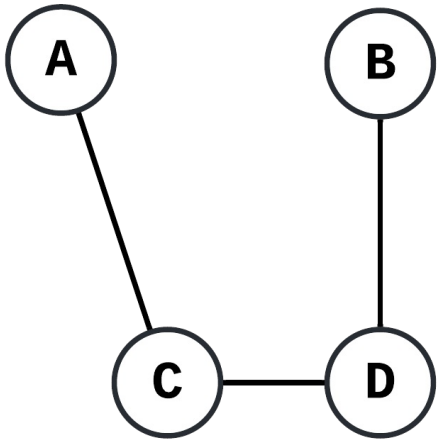
Possible Spanning Trees

From this graph, we can derive several spanning trees by ensuring that all vertices are connected without any cycles. Here are three possible spanning trees:

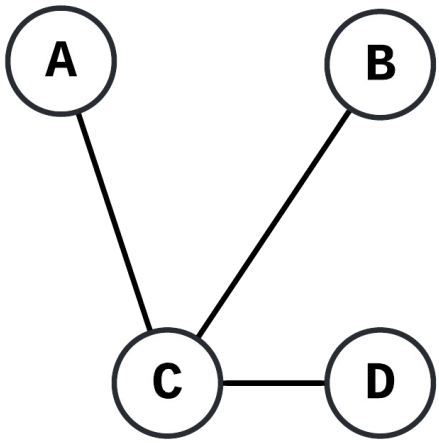
1. **Spanning Tree 1**:
   Edges: AB, AC, CD



2. **Spanning Tree 2**:
   Edges: AC, BD, CD



3. **Spanning Tree 3**:
   Edges: AC, BC, CD



Observations

- Each spanning tree includes all the vertices (A, B, C, D).
- Each spanning tree has exactly $V - 1$ edges, where V is the number of vertices. In this case, each tree has 3 edges.
- There are no cycles in any of the spanning trees.

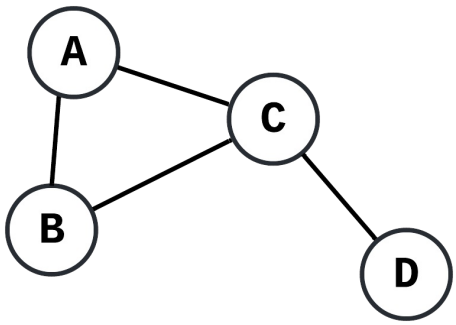
Kruskal's Algorithm

Finding a Spanning Tree

In this section, we will demonstrate the process of finding a spanning tree and introduce Kruskal's algorithm, a popular method for finding the minimum spanning tree (MST). Let's consider a simple undirected graph and demonstrate how to find a spanning tree:

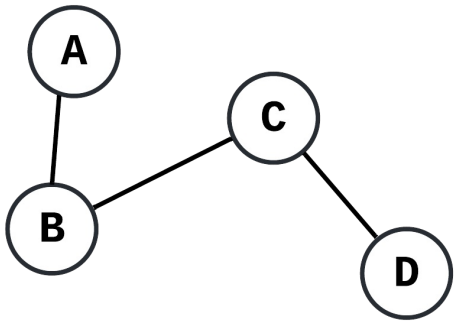
- **Start with an empty set of edges.**
- **Add edges one by one, ensuring that no cycles are formed.**
- **Continue until all vertices are connected.**

For example, assume we are working with the following graph:



- Start with an empty set.
- Add edge `AB`.
- Add edge `BC` (adding `AC` would form a cycle with `AB`).
- Add edge `CD`.

Now, all vertices are connected without any cycles, so we have a spanning tree.



Kruskal's Algorithm

Kruskal's algorithm is a popular method for finding the minimum spanning tree (MST) of a graph. It works as follows:

1. **Sort all the edges in ascending order by their weight.**
2. **Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If not, add it to the spanning tree.**
3. **Repeat step 2 until there are $V - 1$ edges in the spanning tree, where V is the number of vertices.**

Kruskal's algorithm uses a disjoint-set data structure to keep track of which vertices are in which components (i.e., connected subgraphs). This helps in checking for cycles efficiently.

▼ **Click here to see more on disjoint sets**

> In mathematics, a set is a collection of distinct objects, considered as an object in its own right. Sets are fundamental objects in mathematics and are used to define more complex mathematical structures like numbers, functions, and relations. The objects in a set are called its elements or members. Sets are typically denoted by curly braces `{}`, and the elements are listed inside the braces, separated by commas.
>
> In the context of Kruskal's algorithm, "disjoint" refers to the disjoint-set data structure (also known as a union-find data structure) used to keep track of which vertices are in which components (i.e., connected subgraphs) of the graph. The term "disjoint" here indicates that the sets being managed by the data structure are disjoint, meaning they do not overlap.
>
> The disjoint-set data structure supports two main operations:
>
> 1. `find`: Given a vertex, this operation returns the representative (root) of the set to which the vertex belongs. It is used to determine if two vertices belong to the same component (set).
> 2. `union`: Given two vertices, this operation merges the sets to which the vertices belong into a single set. It is used to join two components when adding an edge to the spanning tree, ensuring that no cycles are formed.
>
> In Kruskal's algorithm, the disjoint-set data structure is used to efficiently check for cycles in the graph. Before adding an edge to the spanning tree, the algorithm uses the `find` operation to determine if the edge's endpoints are already in the same component (i.e., would form a cycle). If they are not in the same component, the edge is added to the spanning tree by using the `union` operation to merge the components.

Start by creating the `Graph` class. We also need the nested class `Edge` that implements comparison. This will allow us to sort `Edge` objects.

```
#include <iostream>
#include <algorithm>

class Graph {
public:
    class Edge {
    public:
        int src, dest, weight;

        bool operator<(const Edge& other) const {
            return weight < other.weight;
        }
    };
};
```

In the `Graph` class, create integer attributes for the number of vertices and edges. We also need an array of `Edge` objects to represent the edges in the graph. In the constructor, there are two integers used to set the number of vertices and edges. Instantiate an array of `Edge` objects.

```
public:
    int numVertices, numEdges;
    Edge* edges;

    Graph(int v, int e) {
        numVertices = v;
        numEdges = e;
        edges = new Edge[numEdges];
    }

    ~Graph() {
        delete[] edges;
    }
```

The `find` method is used to determine the root of the set for a given vertex. This method takes an array of integers and an integer, and it returns an integer. If the value of `disjoint[i]` is `-1`, return `i`, otherwise recursively call the `find` method.

```
    int find(int disjoint[], int i) {
        if (disjoint[i] == -1)
            return i;
        return find(disjoint, disjoint[i]);
    }
```

The `unionSets` method takes an array of integers and two integers representing vertices in the graph. Calculate the root vertex for the given integers. Then update the array of integers.

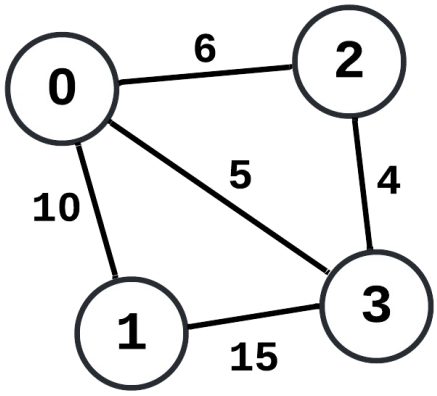
```
    void unionSets(int disjoint[], int x, int y) {
        int xset = find(disjoint, x);
        int yset = find(disjoint, y);
        disjoint[xset] = yset;
    }
```

Now we are ready for Kruskal's algorithm itself. This method returns a dynamic array of `Edge` objects. Start by making an empty array of `Edge` objects that represent the minimal spanning tree. Next, create an array that represents a disjoint set data structure.

```
    Edge* kruskalMST() {
        Edge* result = new Edge[numVertices - 1];
        int resultIndex = 0;
        int sortedIndex = 0;

        std::sort(edges, edges + numEdges);

        int* disjoint = new int[numVertices];
        std::fill(disjoint, disjoint + numVertices, -1);

        while (resultIndex < numVertices - 1) {
            Edge nextEdge = edges[sortedIndex++];
            int x = find(disjoint, nextEdge.src);
            int y = find(disjoint, nextEdge.dest);

            if (x != y) {
                result[resultIndex++] = nextEdge;
                unionSets(disjoint, x, y);
            }
        }

        delete[] disjoint;
        return result;
    }
```

Testing the Code

To test our code, create the `main` method. Create a `Graph` object with 4 vertices and 5 edges. The first edge goes from vertex O to vertex 1 and has a weight of 10. The second edge goes from vertex 0 to vertex 2 and has a weight of 6. The third edge goes from vertex 0 to vertex 3 and has a weight of 5. The fourth edges goes from vertex 1 to vertex 3 and has a weight of 15. The fifth and final edge goes from vertex 2 to vertex 3 and has a weight of 4.




```cpp
int main() {
    int numVertices = 4;
    int numEdges = 5;
    Graph graph(numVertices, numEdges);

    graph.edges[0].src = 0;
    graph.edges[0].dest = 1;
    graph.edges[0].weight = 10;

    graph.edges[1].src = 0;
    graph.edges[1].dest = 2;
    graph.edges[1].weight = 6;

    graph.edges[2].src = 0;
    graph.edges[2].dest = 3;
    graph.edges[2].weight = 5;

    graph.edges[3].src = 1;
    graph.edges[3].dest = 3;
    graph.edges[3].weight = 15;

    graph.edges[4].src = 2;
    graph.edges[4].dest = 3;
    graph.edges[4].weight = 4;

    Graph::Edge* result = graph.kruskalMST();

    std::cout << "Edges in the minimum spanning tree:\n";
    for (int i = 0; i < numVertices - 1; ++i) {
        std::cout << "Edge from vertex " << result[i].src
                  << " to vertex " << result[i].dest
                  << " with weight " << result[i].weight <<
        std::endl;
    }

    delete[] result;
    return 0;
}
```

▼ **Code**

> Your code should look like this:

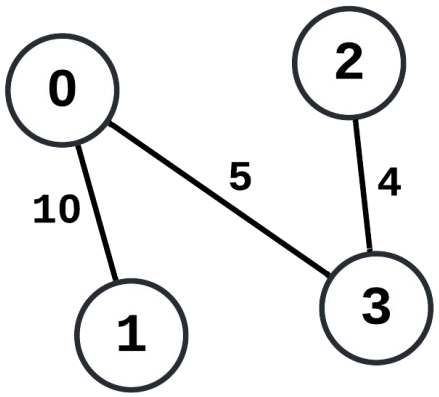
```
#include <iostream>
#include <algorithm>

class Graph {
public:
    class Edge {
    public:
        int src, dest, weight;

        bool operator<(const Edge& other) const {
            return weight < other.weight;
        }
    };

public:
    int numVertices, numEdges;
    Edge* edges;

    Graph(int v, int e) {
        numVertices = v;
        numEdges = e;
        edges = new Edge[numEdges];
    }

    ~Graph() {
        delete[] edges;
    }

    int find(int disjoint[], int i) {
        if (disjoint[i] == -1)
            return i;
        return find(disjoint, disjoint[i]);
    }

    void unionSets(int disjoint[], int x, int y) {
        int xset = find(disjoint, x);
        int yset = find(disjoint, y);
        disjoint[xset] = yset;
    }

    Edge* kruskalMST() {
        Edge* result = new Edge[numVertices - 1];
        int resultIndex = 0;
        int sortedIndex = 0;

        std::sort(edges, edges + numEdges);

        int* disjoint = new int[numVertices];
        std::fill(disjoint, disjoint + numVertices, -1);

        while (resultIndex < numVertices - 1) {
            Edge nextEdge = edges[sortedIndex++];
            int x = find(disjoint, nextEdge.src);
```

```
            int y = find(disjoint, nextEdge.dest);

            if (x != y) {
                result[resultIndex++] = nextEdge;
                unionSets(disjoint, x, y);
            }
        }

        delete[] disjoint;
        return result;
    }
};

int main() {
    int numVertices = 4;
    int numEdges = 5;
    Graph graph(numVertices, numEdges);

    graph.edges[0].src = 0;
    graph.edges[0].dest = 1;
    graph.edges[0].weight = 10;

    graph.edges[1].src = 0;
    graph.edges[1].dest = 2;
    graph.edges[1].weight = 6;

    graph.edges[2].src = 0;
    graph.edges[2].dest = 3;
    graph.edges[2].weight = 5;

    graph.edges[3].src = 1;
    graph.edges[3].dest = 3;
    graph.edges[3].weight = 15;

    graph.edges[4].src = 2;
    graph.edges[4].dest = 3;
    graph.edges[4].weight = 4;

    Graph::Edge* result = graph.kruskalMST();

    std::cout << "Edges in the minimum spanning tree:\n";
    for (int i = 0; i < numVertices - 1; ++i) {
        std::cout << "Edge from vertex " << result[i].src
                  << " to vertex " << result[i].dest
                  << " with weight " << result[i].weight <<
        std::endl;
    }

    delete[] result;
    return 0;
}
```

You should see the following output:

```
Edges in the minimum spanning tree:
Edge from vertex 2 to vertex 3 with weight 4
Edge from vertex 0 to vertex 3 with weight 5
Edge from vertex 0 to vertex 1 with weight 10
```

If you were to draw out this graph, it would look something like:



In this implementation, the `Graph` class represents an undirected graph with a specified number of vertices and edges. The `kruskalMST` method sorts the edges by weight and then iteratively adds the smallest edge to the spanning tree, provided it doesn't create a cycle. The cycle check is performed using the disjoint set data structure.

Breaking Down Kruskal's Algorithm

Disjoint Set

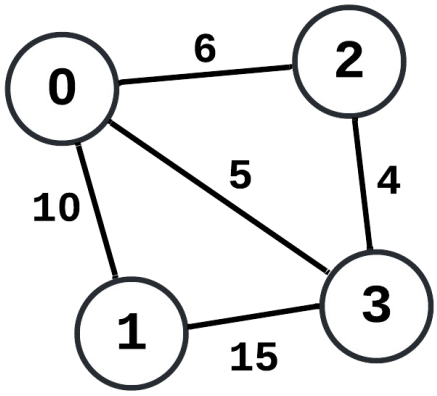
The key to understanding Kruskal's algorithm rests on disjoint sets. A disjoint set (sometimes called union-find) is a data structure that keeps track of non-overlapping (disjoint) sets. In our code, this is done with the disjoint array. We initiate this array with `-1` for each element. This means that each vertex represents its own set. As the algorithm advances, the elements in disjoint will change.

The `find` method takes a vertex (`i`) and follows the chain of parent pointers (recursive calls) until it reaches the root (`-1`) and returns `i`. The `union` method then sets the root of one set (`yset`) as the parent of the root of the other set (`xset`), effectively merging the two sets.

```
int find(int disjoint[], int i) {
    if (disjoint[i] == -1) {
        return i;
    }
    return find(disjoint, disjoint[i]);
}

void unionSets(int disjoint[], int x, int y) {
    int xset = find(disjoint, x);
    int yset = find(disjoint, y);
    disjoint[xset] = yset;
}
```

Walking Through Kruskal's Algorithm

Now that we have a better understanding of a disjoint set data structure, let's walk through Kruskal's algorithm using the same example graph from the previous page:



Step 1

The disjoint array is initiated with all elements set to `-1` as each vertex is the root of its own set. The array of edges is sorted in ascending order by weight.

```
disjoint: [-1, -1, -1, -1]
sorted edges: (2, 3, 4), (0, 3, 5), (0, 2, 6), (0, 1, 10), (1,
        3, 15)
```

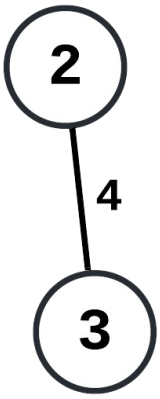
Step 2

- Start with the edge `(2, 3, 4)`.
- Finding the root of `2` returns `2` since `disjoint[2]` is `-1`, so `x = 2`.
- Finding the root of `3` returns `3` since `disjoint[3]` is `-1`, so `y = 3`.
- Since `x` is not equal to `y` (the edge will not cause a cycle), add the edge to the result and call `unionSets(disjoint, 2, 3)`.

The value of `disjoint` is now:

```
[-1, -1, 3, -1]
```

- The minimal spanning tree looks like:



Step 3

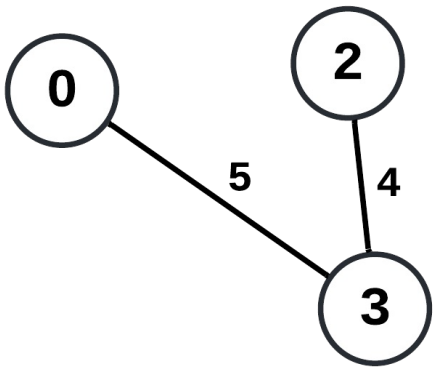
- Process the edge `(0, 3, 5)`.
- Finding the root of `0` returns `0` since `disjoint[0]` is `-1`, so `x = 0`.
- Finding the root of `3` returns `3` since `disjoint[3]` is `-1`, so `y = 3`.

Since `x` is not equal to `y` (the edge will not cause a cycle), add the `edge` to the result and call `unionSets(disjoint, 0, 3)`.

The value of `disjoint` is now:

```
[3, -1, 3, -1]
```

- The minimal spanning tree looks like:



Step 4

- Process the edge `(0, 2, 6)`.

- Finding the root of `0` returns `3`. Since `disjoint[0]` is not `-1` you look up `disjoint[3]` which is `-1`, so `x = 3`.
- Finding the root of `2` returns `3`. Since `disjoint[2]` is not `-1` you look up `disjoint[3]` which is `-1`, so `y = 3`.

Since `x` is equal to `y`, do not add the edge to the result as this would cause a cycle. You do not perform a union.

The value of `disjoint` remains:

```
[3, -1, 3, -1]
```

The minimal spanning tree looks like:

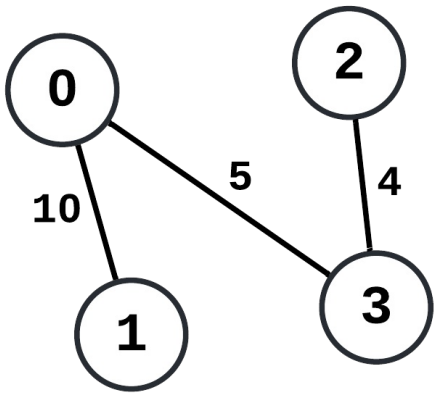
Step 5

- Process the edge `(0, 1, 10)`.
- Finding the root of `0` returns `3` since `disjoint[0]` is `3`. Next you look up `disjoint[3]` which is `-1`, so `x = 3`.
- Finding the root of `1` returns `1` since `disjoint[1]` is `-1`, so `y = 1`.
- Since `x` is not equal to `y` (the edge will not cause a cycle), add the edge to the result and call `unionSets(disjoint, 3, 1)`.

The value of `disjoint` is now:

```
[3, -1, 3, 1]
```

- The minimal spanning tree looks like:



Wrapping-Up

The final state of the `disjoint` array means that vertices 0 and 2 are in the same set whose root is vertex 3. Vertex 1 is the root of its own set, and vertex 3 is in a set whose root is vertex 1.

To really understand Kruskal's algorithm, you need to understand the interplay between a disjoint data set, the `find` method, and the `union` method. The disjoint set and the `find` method are used to calculate the values `x` and `y`, which are then used to determine if an edge is added (or not) to the minimal spanning tree. The `union` method is then used to update the elements in the disjoint set.

Deep Dive: Kruskal's Algorithm

Understanding Kruskal's Algorithm

To ensure a thorough understanding of Kruskal's algorithm, it's important to delve into its mechanics and examine how it efficiently finds the minimum spanning tree (MST) of a graph. This deeper understanding will equip you with the ability to apply this algorithm to complex problems and appreciate its elegance and efficiency.

Step-by-Step Walkthrough

Let's break down Kruskal's algorithm into its core steps and examine each in detail:

1. **Sort Edges by Weight**: Begin by sorting all edges in the graph in ascending order of their weights. This step ensures that when we start adding edges to the MST, we always consider the smallest edge first, adhering to the greedy approach.

2. **Initialize Disjoint Sets**: Initialize a disjoint-set data structure for all vertices in the graph. Initially, each vertex is in its own set, representing that no vertices are connected yet.

3. **Iterate Over Sorted Edges**: Go through the sorted edges one by one. For each edge, check if the endpoints of the edge belong to different sets in the disjoint-set data structure.

    - If they belong to different sets, it means adding this edge will not create a cycle. Add the edge to the MST and union the two sets in the disjoint-set data structure.
    - If they belong to the same set, ignore this edge as adding it would create a cycle.

4. **Repeat Until MST is Complete**: Continue this process until you have added $V - 1$ edges to the MST, where V is the number of vertices in the graph.

Understanding the Disjoint-Set Data Structure

A key component of Kruskal's algorithm is the disjoint-set data structure, also known as a union-find data structure. It is crucial for efficiently managing the sets of vertices and checking for cycles. Here's how it works:

- **Find**: Determine which set a particular element belongs to. This can be used for checking if two vertices belong to the same set.
- **Union**: Combine two sets into a single set. This is used when an edge is

added to the MST, connecting two previously unconnected components.

The disjoint-set data structure is optimized with techniques like union by rank and path compression, making the operations nearly constant time.

Visualizing Kruskal's Algorithm

Visualizing the algorithm can greatly aid in understanding its process. Here's a simple way to visualize it:

1. Draw the graph with all its vertices and edges.
2. Highlight the edges in the order they are added to the MST.
3. Observe how the sets in the disjoint-set data structure evolve as edges are added.

Analyzing Kruskal's Algorithm

Analyzing Kruskal's Algorithm

Understanding the complexity and potential optimizations of Kruskal's algorithm is crucial for assessing its efficiency and suitability for different applications. Let's delve into these aspects to gain a deeper understanding.

Time Complexity Analysis

The time complexity of Kruskal's algorithm is primarily determined by the sorting of edges and the operations on the disjoint-set data structure. Here's a breakdown:

1. **Sorting Edges**: Sorting all the edges of the graph takes $O(ElogE)$ time, where E is the number of edges.
2. **Disjoint-Set Operations**: Each edge requires two find operations and at most one union operation. With path compression and union by rank, these operations can be performed in nearly constant time, $O(\alpha(V))$, where α is the inverse Ackermann function, which grows very slowly and can be considered almost constant for all practical purposes and V is the number of vertices.
3. **Overall Complexity**: Combining the sorting and disjoint-set operations, the total time complexity is $O(ElogE + E\alpha(V))$, which simplifies to $O(ElogE)$ since $\alpha(V)$ is nearly constant.

Space Complexity Analysis

The space complexity of Kruskal's algorithm is primarily determined by the storage of edges and the disjoint-set data structure:

1. **Edge Storage**: Storing all the edges requires $O(E)$ space.
2. **Disjoint-Set Data Structure**: The disjoint-set data structure requires $O(V)$ space, where V is the number of vertices.
3. **Overall Complexity**: The total space complexity is $O(E + V)$.

Optimizations

1. **Edge Sorting**: If the edges are already sorted or can be sorted in linear time (e.g., using counting sort for small integer weights), the time complexity can be reduced.
2. **Disjoint-Set Implementation**: Implementing the disjoint-set data structure with path compression and union by rank ensures that the find and union operations are nearly constant time.
3. **Early Termination**: The algorithm can be terminated as soon as $V - 1$

edges are added to the MST, which might lead to a slight improvement in runtime for dense graphs.

Practical Considerations

- **Sparse vs. Dense Graphs**: Kruskal's algorithm is generally more efficient for sparse graphs. For dense graphs, Prim's algorithm might be a better choice.
- **Edge Weight Distribution**: The efficiency of sorting can vary depending on the distribution of edge weights. For certain distributions, specialized sorting algorithms can be used to improve efficiency.
- **Parallelization**: Some parts of the algorithm, such as edge sorting, can be parallelized to improve performance on multi-core systems.

By understanding the complexity and optimizations of Kruskal's algorithm, you can make informed decisions about its application and ensure efficient implementation in solving minimum spanning tree problems.

Prim's Algorithm

An Alternative Approach for Finding MSTs

While Kruskal's algorithm is a powerful tool for finding the minimum spanning tree (MST) of a graph, another equally important algorithm in the realm of graph theory is Prim's algorithm. Prim's algorithm offers an alternative approach to finding the MST, particularly well-suited for dense graphs.

Prim's algorithm, like Kruskal's, is a greedy algorithm that builds the MST one edge at a time. However, while Kruskal's algorithm sorts all edges and adds them one by one, Prim's algorithm grows the MST from a single starting vertex by adding the cheapest edge from the tree to a vertex not yet in the tree.

Steps of Prim's Algorithm

1. **Initialization**: Start with a vertex, marking it as part of the MST. Initialize a priority queue with all edges from this vertex, sorted by their weights.
2. **Iterate Until the MST is Complete**: Repeat the following steps until the MST includes all vertices:
    - Extract the minimum weight edge from the priority queue.
    - If the edge connects to a vertex not already in the MST, add the vertex to the MST and add all its edges to the priority queue.
    - If the edge connects to a vertex already in the MST, discard the edge.
3. **Finalize the MST**: Once all vertices are included in the MST, the algorithm terminates, and the MST is complete.

Implementing Prim's Algorithm

Start by creating the `Graph` class. Before adding attributes and methods for the `Graph` class, we need to define the `Edge` class. It has two integer attributes for the destination and weight. The constructor takes two integers and sets them to their respective attribute. The attributes need getter methods so we can access them later on. Lastly, override the << operator so that when we print an `Edge` object, we see the vertex and weight attributes with spacing between them.

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <limits>
#include <functional>

class Graph {
private:
    class Edge {
        public:
            int vertex, weight;

            Edge(int vertex, int weight) {
                this->vertex = vertex;
                this->weight = weight;
            }

            friend std::ostream& operator<<(std::ostream& os,
        const Edge& e) {
                os << e.vertex << " \t" << e.weight;
                return os;
            }
    };
```

The `Graph` class has the integer attribute `numVertices` representing the number of vertices in the graph. It also has the `adjList` attribute which is an adjacency list. The adjacency list uses an integer representing a vertex and a vector of `Edge` objects representing neighbors. The private helper method `addVertex` is used by the constructor to add a vertex to the graph. The list of neighbors for the given vertex is empty.

```
    int numVertices;
    std::map<int, std::vector<Edge>> adjList;

    void addVertex(int vertex) {
        adjList[vertex] = std::vector<Edge>();
    }
```

The `Graph` constructor takes an integer representing the number of vertices for the graph. Iterate the same number of times as vertices in the graph. Using the `addVertex` method, add a vertex to the graph with each iteration.

```
public:
    Graph(int vertices) {
        numVertices = vertices;
        for (int i = 0; i < numVertices; i++) {
            addVertex(i);
        }
    }
```

The public `addEdge` method adds an edge to the graph. It takes integers representing the first vertex, the second vertex, and the weight. Since this is an undirected graph, make sure that the edges are bidirectional between the first and second vertices.

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

We are now ready to implement Prim's algorithm. Start by creating the `primMST` method. It has no parameters and returns a `Graph` object. We need three arrays. The first array, `inMST`, is a boolean array used to keep track of vertices in the MST. The `minWeights` array is used to keep track of the minimum weight for an edge. The `edges` array is used to keep track of edges in the minimum spanning tree. Fill the `minWeights` array with the largest possible integer, and fill edges with `-1`.

```
    Graph primMST() {
        std::vector<bool> inMST(numVertices, false);
        std::vector<int> minWeights(numVertices,
        std::numeric_limits<int>::max());
        std::vector<int> edges(numVertices, -1);
        minWeights[0] = 0;
```

We are going to use a priority queue to hold `Edge` objects. We also need to be able to sort the items in the priority queue by edge weight. Set the first element in `minWeights` to `0`, and add an edge to the priority `queue`. The edge has a `destination` of `0` and a weight of `0`. Iterate as long as there are elements in the priority queue. Remove the next element in the queue, get the vertex from the `Edge` object, and set it as the current vertex. Mark the current vertex as being in the MST.

```
        auto cmp = [](const Edge& left, const Edge& right) {
            return left.weight > right.weight;
        };
        std::priority_queue<Edge, std::vector<Edge>,
        decltype(cmp)> pq(cmp);

        pq.push(Edge(0, 0));

        while (!pq.empty()) {
            int current = pq.top().vertex;
            pq.pop();
            inMST[current] = true;
```

Iterate over the neighbors for the current vertex. If the neighbor is not already in the MST and the weight of the neighbor is less than the current minimum weight, the neighbor also belongs in the MST. Update `minWeights`

with the neighbor's weight. Set `edges[neighbor]` to the current vertex. Then add the neighbor to the priority queue.

```
            for (const Edge& edge : adjList[current]) {
                int neighbor = edge.vertex;
                int neighborWeight = edge.weight;

                if (!inMST[neighbor] && neighborWeight <
        minWeights[neighbor]) {
                    minWeights[neighbor] = neighborWeight;
                    edges[neighbor] = current;
                    pq.push(Edge(neighbor, neighborWeight));
                }
            }
        }
```

The `minimum` spanning tree can be determined through a combination of the `edges` and `minWeights` arrays. We are going to encapsulate the MST in a `Graph` object so it is a bit easier to understand. Create a `Graph` object for the MST. Have a for loop start with the index `1` and run the same number of times as there are vertices in the graph. For each iteration, add an edge to the MST. The first vertex is located at `edges[i]`, the second vertex is `i`, and the weight is `minWeights[i]`.

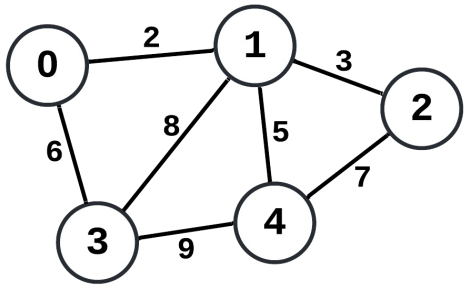
```
        Graph mst(numVertices);
        for (int i = 1; i < numVertices; i++) {
            mst.addEdge(edges[i], i, minWeights[i]);
        }

        return mst;
    }
```

We need to be able to print the minimum spanning tree to see its contents. Create the `printGraph` method which has no parameters. Iterate over the entries in the map. For each entry, iterate over the `Edge` objects. Get the key from the map entry. If the key is less than vertex in the edge, print the complete edge. If we do not use this conditional, we will print the edge `1 - 3` as well as `3 - 1`. We know this is an undirected graph, so we don't need to see the same edge twice.

```
    void printGraph() const {
        for (const auto& entry : adjList) {
            for (const Edge& edge : entry.second) {
                int source = entry.first;
                if (source < edge.vertex) {
                    std::cout << source << " - " << edge <<
        std::endl;
                }
            }
        }
    }
};
```

Testing the Code

To test the algorithm, create the `main` function. In it, instantiate a `Graph` object with five vertices. Add edges from vertex 0 to vertex 1 with a weight of 2, an edge between vertex 0 and vertex 3 with a weight of 6, an edge between vertex 1 and vertex 2 with a weight of 3, an edge between vertex 1 and vertex 3 with a weight of 8, an edge between vertex 1 and vertex 4 with a weight of 5, an edge between vertex 2 and vertex 4 with a weight of 7, and an edge between vertex 3 and vertex 4 with a weight of 9. Call `primMST` on the graph to generate the minimal spanning tree. Then print the tree.



```
int main() {
    Graph graph(5);

    graph.addEdge(0, 1, 2);
    graph.addEdge(0, 3, 6);
    graph.addEdge(1, 2, 3);
    graph.addEdge(1, 3, 8);
    graph.addEdge(1, 4, 5);
    graph.addEdge(2, 4, 7);
    graph.addEdge(3, 4, 9);

    Graph mst = graph.primMST();
    std::cout << "Edge\tWeight" << std::endl;
    mst.printGraph();

    return 0;
}
```

▼ **Code**

> Your code should look like this:

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <limits>
#include <functional>

class Graph {
private:
    class Edge {
        public:
            int vertex, weight;

            Edge(int vertex, int weight) {
                this->vertex = vertex;
                this->weight = weight;
            }

            friend std::ostream& operator<<(std::ostream&
        os, const Edge& e) {
                os << e.vertex << " \t" << e.weight;
                return os;
            }
    };

    int numVertices;
    std::map<int, std::vector<Edge>> adjList;

    void addVertex(int vertex) {
        adjList[vertex] = std::vector<Edge>();
    }

public:
```

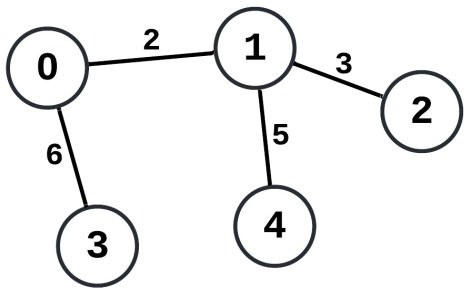
```
    Graph(int vertices) {
        numVertices = vertices;
        for (int i = 0; i < numVertices; i++) {
            addVertex(i);
        }
    }

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

    Graph primMST() {
        std::vector<bool> inMST(numVertices, false);
        std::vector<int> minWeights(numVertices,
        std::numeric_limits<int>::max());
        std::vector<int> edges(numVertices, -1);
        minWeights[0] = 0;

        auto cmp = [](const Edge& left, const Edge& right)
        {
            return left.weight > right.weight;
        };
        std::priority_queue<Edge, std::vector<Edge>,
        decltype(cmp)> pq(cmp);

        pq.push(Edge(0, 0));

        while (!pq.empty()) {
            int current = pq.top().vertex;
            pq.pop();
            inMST[current] = true;

            for (const Edge& edge : adjList[current]) {
                int neighbor = edge.vertex;
                int neighborWeight = edge.weight;

                if (!inMST[neighbor] && neighborWeight <
        minWeights[neighbor]) {
                    minWeights[neighbor] = neighborWeight;
                    edges[neighbor] = current;
                    pq.push(Edge(neighbor,
        neighborWeight));
                }
            }
        }

        Graph mst(numVertices);
        for (int i = 1; i < numVertices; i++) {
            mst.addEdge(edges[i], i, minWeights[i]);
        }

        return mst;
    }

    void printGraph() const {
        for (const auto& entry : adjList) {
            for (const Edge& edge : entry.second) {
                int source = entry.first;
```

```
                if (source < edge.vertex) {
                    std::cout << source << " - " << edge <<
        std::endl;
                }
            }
        }
    }
};

int main() {
    Graph graph(5);

    graph.addEdge(0, 1, 2);
    graph.addEdge(0, 3, 6);
    graph.addEdge(1, 2, 3);
    graph.addEdge(1, 3, 8);
    graph.addEdge(1, 4, 5);
    graph.addEdge(2, 4, 7);
    graph.addEdge(3, 4, 9);

    Graph mst = graph.primMST();
    std::cout << "Edge\tWeight" << std::endl;
    mst.printGraph();

    return 0;
}
```

You should see the following output:

```
Edge    Weight
0 - 1   2
0 - 3   6
1 - 2   3
1 - 4   5
```

If you were to draw out this graph, it would look something like:



In this implementation, the `Graph` class represents an undirected graph with a specified number of vertices and edges. The `primMST` method constructs the MST using Prim's algorithm, starting from vertex 0. The

priority queue pq is used to manage the edges based on their weights, ensuring that the smallest edge is always selected next.

Comparing Kruskal's and Prim's Algorithms

Comparing Kruskal's and Prim's Algorithms

Kruskal's and Prim's algorithms are two popular methods for finding the minimum spanning tree (MST) of a graph. While they both serve the same purpose, they have different approaches and characteristics. Understanding the differences between these algorithms is crucial for choosing the right one for a given problem.

Prim's Algorithm Overview

Prim's algorithm starts with a single vertex and grows the MST by adding the smallest edge that connects the tree to a vertex not yet in the tree. It continues this process until all vertices are included in the MST.

Key Differences

1. **Approach**:
    - **Kruskal's Algorithm**: Works by sorting all the edges and adding them to the MST one by one, ensuring no cycles are formed.
    - **Prim's Algorithm**: Starts with a single vertex and expands the MST by adding the smallest edge that connects the tree to a new vertex.
2. **Data Structures**:
    - **Kruskal's Algorithm**: Uses a disjoint-set data structure to manage the sets of vertices and check for cycles.
    - **Prim's Algorithm**: Uses a priority queue to select the next smallest edge to add to the MST.
3. **Suitability**:
    - **Kruskal's Algorithm**: Generally more efficient for sparse graphs, where the number of edges is much less than the square of the number of vertices.
    - **Prim's Algorithm**: Typically better for dense graphs, where the number of edges is close to the square of the number of vertices.
4. **Time Complexity**:
    - **Kruskal's Algorithm**: $O(ElogE)$, where E is the number of edges.
    - **Prim's Algorithm**: $O(V^2)$ with a simple priority queue, or $O(E + VlogV)$ with a binary heap or Fibonacci heap, where V is the number of vertices.

Choosing Between Kruskal's and Prim's

- **Graph Density**: For sparse graphs, Kruskal's algorithm is often preferred due to its efficiency in handling a smaller number of edges. For dense graphs, Prim's algorithm is usually more efficient.
- **Data Structure Overhead**: Kruskal's algorithm has a lower overhead since it primarily relies on sorting and disjoint-set operations. Prim's algorithm requires a more complex priority queue, which can have higher overhead, especially for dense graphs.
- **Implementation Complexity**: Prim's algorithm can be simpler to implement, especially if the graph is represented using an adjacency list.

Both Kruskal's and Prim's algorithms are powerful tools for finding the minimum spanning tree of a graph. The choice between them depends on the characteristics of the graph and the specific requirements of the problem at hand. Understanding their differences and strengths allows you to select the most appropriate algorithm for efficiently solving MST problems.

Formative Assessment 1

Formative Assessment 2