Contents

1 Kruskal's Algorithm for Minimum Spanning Tree (MST) 2
1.1 Data Structure Definitions 2
1.2 Code 2
1.3 Step 1: Initialize the Kruskal's MST Structure 3
1.4 Step 2: Initialize the Components Array 3
1.5 Step 3: Process Edges from the Min-Heap . . . 3
1.6 Step 4: Check for Cycles (Disjoint Set Union) . 3
1.7 Step 5: Merge the Components 3
1.8 Step 6: Add the Edge to the MST 3
1.9 Summary 4

2 Prim's Algorithm (non-POT implementation 5
2.1 Data Structure Definitions 5
2.2 Code 5
2.3 Step 1: Initialize the Prim's MST Structure . . 6
2.4 Step 2: Initialize the Visited Set 6
2.5 Step 3: Process Edges to Find the Minimum Edge 6
2.6 Step 4: Check Each Edge for Minimum Weight 6
2.7 Step 5: Add the Minimum Edge to the MST . 6
2.8 Step 6: Mark the Vertex as Visited and Add Edge to MST 7
2.9 Step 7: Return the Minimum Spanning Tree . . 7
2.10 Summary 7

1 Kruskal's Algorithm for Minimum Spanning Tree (MST)

The function `kruskAlgo` implements Kruskal's Algorithm to find the Minimum Spanning Tree (MST) of a graph. It uses a Min-Heap to store edges sorted by weight, and progressively selects edges to form the MST while ensuring no cycles are created. Below is a detailed explanation of the function:

1.1 Data Structure Definitions

```
typedef int graphType[MAX][MAX];
typedef int set[MAX];

typedef struct {
 int u, v;
 int weight;
}edgetype;

typedef struct {
 edgetype heap[HEAPMAX];
 int count;
} minHeap;

typedef struct {
    edgetype edges[MAX];
 int minCost;
 int count;
}kruskMST;
```

1.2 Code

```
kruskMST kruskAlgo(minHeap* heap)
{
    kruskMST k = {.minCost = 0, .count = 0};

    int comp[6];

    int i;
    for(i = 0; i < MAX; i++){
        comp[i] = i;
    }

    while(heap->count > 0){
        edgetype min = deleteMin(heap);

        if(comp[min.u] != comp[min.v]){
            int change = comp[min.u];

            for(i = 0; i < MAX; i++){
                if(comp[i] == change){
                    comp[i] = comp[min.v];
                }
            }

            k.edges[k.count++] = min;
            k.minCost += min.weight;
        }
    }

    return k;
}
```

1.3 Step 1: Initialize the Kruskal's MST Structure

We begin by initializing a structure to store the MST and its associated cost:

```
kruskMST k = {.minCost = 0, .count = 0};
```

Here: - `k.minCost` is set to 0, and it will hold the total cost of the MST once it is calculated. - `k.count` starts at 0 and will keep track of the number of edges added to the MST.

1.4 Step 2: Initialize the Components Array

Next, we initialize an array `comp` where each vertex is initially its own parent:

```
int comp[6];
for(i = 0; i < MAX; i++) {
    comp[i] = i;
}
```

The array `comp[i]` represents the parent of vertex i. Initially, each vertex is its own parent, indicating that all vertices are in separate connected components.

1.5 Step 3: Process Edges from the Min-Heap

We then enter a loop where we process edges from the Min-Heap. As long as there are edges left in the heap, we select the edge with the smallest weight:

```
while(heap->count > 0) {
    edgetype min = deleteMin(heap);
```

Here, the `deleteMin(heap)` function removes and returns the edge with the smallest weight from the heap.

1.6 Step 4: Check for Cycles (Disjoint Set Union)

Before adding the selected edge to the MST, we check if it would form a cycle. This is done by comparing the connected components (parents) of the two vertices of the edge:

```
if(comp[min.u] != comp[min.v]) {
    int change = comp[min.u];
```

If the two vertices u and v are in different components (i.e., `comp[min.u] != comp[min.v]`), then adding the edge does not form a cycle, and we can safely add it to the MST.

1.7 Step 5: Merge the Components

Once we confirm that the edge does not form a cycle, we merge the two components by updating the parent of all vertices in the same component as `min.u`:

```
for(int i = 0; i < MAX; i++) {
    if(comp[i] == change) {
        comp[i] = comp[min.v];
    }
}
```

This loop ensures that all vertices in the same connected component as u now belong to the component of v, effectively merging the two components.

1.8 Step 6: Add the Edge to the MST

Now that the edge is safe to add to the MST, we do so by updating the MST structure:

```
k.edges[k.count++] = min;
k.minCost += min.weight;
```

Here: - `k.edges[k.count++]` adds the edge `min` to the MST. - `k.minCost` is updated by adding the weight of the edge `min.weight`.

Step 7: Return the Minimum Spanning Tree

Finally, the function returns the MST stored in the `k` structure:

```
return k;
```

1.9 Summary

In summary, Kruskal's algorithm processes edges in increasing order of weight, checking whether each edge connects two different components. If it does, the edge is added to the MST, and the two components are merged. This process continues until the MST contains $|V| - 1$ edges, where $|V|$ is the number of vertices.

The time complexity of this implementation is $O(E \log E)$, where E is the number of edges, because of the heap operations and sorting of the edges

2 Prim's Algorithm (non-POT implementation

Prim's Algorithm is a greedy algorithm that finds the minimum spanning tree (MST) for a weighted, connected, and undirected graph. The algorithm starts from an arbitrary vertex and iteratively adds the smallest edge connecting a visited vertex to an unvisited vertex, ensuring that the resulting tree has the minimum possible weight.

2.1 Data Structure Definitions

```
typedef int graphType[MAX][MAX];
typedef int set[MAX];

typedef struct {
 int u, v;
 int weight;
}edgetype;

typedef struct {
    edgetype edges[MAX];
    int edgeCount;
 int minCost;
}primMST;
```

2.2 Code

```
primMST primAlgo(graphType graph, int startVertex)
{
  //Write code here

 primMST p = {.edgeCount = 0, .minCost = 0};
 set visited = {};
 visited[startVertex] = 1;

 int i, j;

 while(p.edgeCount < MAX - 1){

  edgetype minEdge = {.weight = INFINITY};

  for(i = 0; i < MAX; i++){
   if(visited[i] == 1){

    for(j = 0; j < MAX; j++){
     if(visited[j] == 0 && graph[i][j] < minEdge.weight){
      edgetype temp = {i, j, graph[i][j]};
      minEdge = temp;
     }
    }
   }
  }

  p.minCost += minEdge.weight;

  if(minEdge.weight != INFINITY){
   p.edges[p.edgeCount++] = minEdge;
   visited[minEdge.v] = 1;
  } else {
   printf("Graph is not connected");
  }
 }
 return p;
}
```

2.3 Step 1: Initialize the Prim's MST Structure

We begin by initializing a structure to store the MST and its associated cost:

```
primMST p = {.edgeCount = 0, .minCost = 0};
```

Here: - `p.edgeCount` is set to 0 and will keep track of the number of edges added to the MST. - `p.minCost` is also initialized to 0, and it will store the total weight of the MST once it is computed.

2.4 Step 2: Initialize the Visited Set

Next, we initialize a set `visited` to track which vertices have been added to the MST:

```
set visited = {};
visited[startVertex] = 1;
```

Here: - `visited[startVertex] = 1;` marks the starting vertex as visited, indicating the starting point for the MST. - The set `visited` will be used to check which vertices have already been added to the MST.

2.5 Step 3: Process Edges to Find the Minimum Edge

We then enter a loop that continues until the MST contains $V - 1$ edges (where V is the number of vertices in the graph). In each iteration, we find the smallest edge that connects a visited vertex to an unvisited vertex:

```
while(p.edgeCount < MAX - 1) {
    edgetype minEdge = {.weight = INFINITY};
```

Here: - We initialize `minEdge` with an infinite weight to ensure that any valid edge found will be smaller. - The loop continues until we have added $V - 1$ edges to the MST.

2.6 Step 4: Check Each Edge for Minimum Weight

Inside the loop, we iterate through all vertices to find the smallest edge connecting a visited vertex to an unvisited one:

```
for(i = 0; i < MAX; i++) {
    if(visited[i] == 1) {
        for(j = 0; j < MAX; j++) {
            if(visited[j] == 0 && graph[i][j] < minEdge.weight) {
                edgetype temp = {i, j, graph[i][j]};
                minEdge = temp;
            }
        }
    }
}
```

Here: - For each visited vertex i, we check all unvisited vertices j. - If the weight of the edge between i and j is smaller than the current `minEdge`, we update `minEdge` to store this new minimum edge.

2.7 Step 5: Add the Minimum Edge to the MST

Once the minimum edge is found, we update the MST structure and the total cost:

```
p.minCost += minEdge.weight;
```

Here: - We add the weight of the selected `minEdge` to `p.minCost`, which tracks the total cost of the MST.

2.8 Step 6: Mark the Vertex as Visited and Add Edge to MST

Now that we have selected the minimum edge, we mark the destination vertex of the edge as visited and add the edge to the MST:

```
if(minEdge.weight != INFINITY) {
    p.edges[p.edgeCount++] = minEdge;
    visited[minEdge.v] = 1;
} else {
    printf("Graph is not connected");
}
```

Here: - The edge `minEdge` is added to the MST in `p.edges[p.edgeCount++]`. - The destination vertex `minEdge.v` is marked as visited. - If no valid edge is found, it indicates that the graph is disconnected, and we print an error message.

2.9 Step 7: Return the Minimum Spanning Tree

Finally, the function returns the MST stored in the structure `p`:

```
return p;
```

2.10 Summary

In summary, Prim's Algorithm finds the minimum spanning tree (MST) by iteratively selecting the smallest edge that connects a visited vertex to an unvisited vertex. The process continues until all vertices are included in the MST. The total cost of the MST is tracked as the sum of the weights of the selected edges.

The time complexity of this implementation is $O(V^2)$, where V is the number of vertices, due to the nested loops. This can be improved to $O(E \log V)$ by using a priority queue to select the minimum edge more efficiently.