Indian Institute of Information Technology Sonepat

Design & Analysis Of Algorithms (CSL-302)

Submitted To

Mr. Md Arquam

Submitted By

Shivansh Joshi

Branch-CS

Roll No.: 12011042

PRACTICAL-8

AIM

Write a program to implement Prim's Algorithm for 100 nodes using heap and priority queue.

CODE

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a node in adjacency list
struct AdjListNode {
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList {
    struct AdjListNode* head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph {
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
{
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct
AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
```

```
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists.  Size of array will be V
    graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
    // Add an edge from src to dest.  A new node is added to the adjacency
    // list of src.  The node is added at the beginning
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode {
    int v;
    int key;
};

// Structure to represent a min heap
struct MinHeap {
    int size; // Number of heap nodes present currently
    int capacity; // Capacity of min heap
    int* pos; // This is needed for decreaseKey()
    struct MinHeapNode** array;
```

```
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int key)
{
    struct MinHeapNode* minHeapNode = (struct MinHeapNode*)malloc(sizeof(struct
MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->key = key;
    return minHeapNode;
}

// A utilit function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
    minHeap->pos = (int*)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(capacity * sizeof(struct
MinHeapNode*));
    return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size && minHeap->array[left]->key <
minHeap->array[smallest]->key)
        smallest = left;
```

```
    if (right < minHeap->size && minHeap->array[right]->key <
minHeap->array[smallest]->key)
        smallest = right;

    if (smallest != idx) {
        // The nodes to be swapped in min heap
        struct MinHeapNode* smallestNode = minHeap->array[smallest];
        struct MinHeapNode* idxNode = minHeap->array[idx];

        // Swap positions
        minHeap->pos[smallestNode->v] = idx;
        minHeap->pos[idxNode->v] = smallest;

        // Swap nodes
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size - 1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
```

```
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decrease key value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int key)
{
    // Get the index of v in  heap array
    int i = minHeap->pos[v];

    // Get the node and update its key value
    minHeap->array[i]->key = key;

    // Travel up while the complete tree is not hepified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key) {
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;
        minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
        i = (i - 1) / 2;
    }
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
int isInMinHeap(struct MinHeap* minHeap, int v)
{
    if (minHeap->pos[v] < minHeap->size)
        return 1;
    return 0;
}

// A utility function used to print the constructed MST
void printArr(int arr[], int n)
{
    for (int i = 1; i < n; ++i)
        printf("%d - %d\n", arr[i], i);
}
```

```
// The main function that constructs Minimum Spanning Tree (MST)
// using Prim's algorithm
void PrimMST(struct Graph* graph)
{
    int V = graph->V; // Get the number of vertices in graph
    int parent[V]; // Array to store constructed MST
    int key[V]; // Key values used to pick minimum weight edge in cut

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices. Key value of
    // all vertices (except 0th vertex) is initially infinite
    for (int v = 1; v < V; ++v) {
        parent[v] = -1;
        key[v] = INT_MAX;
        minHeap->array[v] = newMinHeapNode(v, key[v]);
        minHeap->pos[v] = v;
    }

    // Make key value of 0th vertex as 0 so that it
    // is extracted first
    key[0] = 0;
    minHeap->array[0] = newMinHeapNode(0, key[0]);
    minHeap->pos[0] = 0;

    // Initially size of min heap is equal to V
    minHeap->size = V;

    // In the following loop, min heap contains all nodes
    // not yet added to MST.
    while (!isEmpty(minHeap)) {
        // Extract the vertex with minimum key value
        struct MinHeapNode* minHeapNode = extractMin(minHeap);
        int u = minHeapNode->v; // Store the extracted vertex number

        // Traverse through all adjacent vertices of u (the extracted
        // vertex) and update their key values
        struct AdjListNode* pCrawl = graph->array[u].head;
        while (pCrawl != NULL) {
            int v = pCrawl->dest;

            // If v is not yet included in MST and weight of u-v is
            // less than key value of v, then update key value and
            // parent of v
```

```
            if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v]) {
                key[v] = pCrawl->weight;
                parent[v] = u;
                decreaseKey(minHeap, v, key[v]);
            }
            pCrawl = pCrawl->next;
        }
    }

    // print edges of MST
    printArr(parent, V);
}

// Driver program to test above functions
int main()
{
    // Let us create the graph given in above fugure
    int V = 100;
    struct Graph* graph = createGraph(V);
    for(int a =0;a<99;a++)
    {
        for(int b=a+1;b<100;b++)
        {
            addEdge(graph, a, b, (rand()%100+1));
        }
    }
    printf("Edges for Prim's minimum spanning tree:\n");
    PrimMST(graph);

    return 0;
}
```

OUTPUT

```
Edges for Prim's minimum spanning tree:
8 - 1
57 - 2
31 - 3
75 - 4
45 - 5
16 - 6
73 - 7
85 - 8
4 - 9
58 - 10
94 - 11
38 - 12
78 - 13
16 - 14
69 - 15
64 - 16
40 - 17
84 - 18
97 - 19
89 - 20
35 - 21
64 - 22
52 - 23
80 - 24
18 - 25
95 - 26
54 - 27
11 - 28
74 - 29
84 - 30
26 - 31
0 - 32
20 - 33
85 - 34
20 - 35
86 - 36
81 - 37
75 - 38
```

```
26 - 31
0 - 32
20 - 33
85 - 34
20 - 35
86 - 36
81 - 37
75 - 38
53 - 39
28 - 40
35 - 41
44 - 42
65 - 43
82 - 44
17 - 45
57 - 46
61 - 47
27 - 48
95 - 49
56 - 50
87 - 51
79 - 52
61 - 53
60 - 54
47 - 55
44 - 56
8 - 57
82 - 58
72 - 59
20 - 60
58 - 61
24 - 62
61 - 63
60 - 64
71 - 65
4 - 66
77 - 67
51 - 68
58 - 69
```

```
71 - 65
4 - 66
77 - 67
51 - 68
58 - 69
1 - 70
15 - 71
28 - 72
63 - 73
44 - 74
89 - 75
2 - 76
72 - 77
61 - 78
20 - 79
60 - 80
32 - 81
40 - 82
62 - 83
79 - 84
35 - 85
77 - 86
17 - 87
90 - 88
81 - 89
81 - 90
16 - 91
15 - 92
57 - 93
60 - 94
45 - 95
4 - 96
41 - 97
87 - 98
23 - 99


...Program finished with exit code 0
Press ENTER to exit console.
```

ANALYSIS

The adjacency list representation of a graph allows the whole to be traversed in O(V+E) time using Breath First Search(BFS), as opposed to $O(V^2)$ time for adjacency matrix.

The methodology of this algorithm is to traverse all the vertices of the graph using BFS and use a Min Heap to store the vertices not yet included in the Minimum Spanning Tree. Min Heap is used as a priority queue to get the minimum weight edge.

Time complexity for extracting minimum weight edge from the Min Heap is O(lgV).

The PrimMST function calculates the minimum weight path through the graph scanning all the possible ways the spanning tree can be built. It runs for a total of V+E times. At each run it finds the minimum weighted edge connecting the concerned vertices of the graph at that point of execution. Min weight edge is extracted using the Min Heap.

Therefore, Prim's algorithms takes time $O(E + V) \times O(lgV) = O((E + V)lgV) = O(ElgV)$ as for a connected graph $V = O(E)$.

However, this time can be reduced t $O(E + VlgV)$ using fibonacci heap.

Therefore,

Time complexity of Prim's algorithm using binary heap $O(ElgV)$.