EECE 7205-Assignment 4

Sreejith Sreekumar: 001277209

November 9, 2018

1 Qn1 - V1. Prims Algorithm: Adjacency Matrix Implementation

1.1 Code

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


int get_min_key(int key[], int priority_queue[], int vertex_count){

  int min = 9999, min_index;

  for (int v = 0; v < vertex_count; v++) {
     if (priority_queue[v] != -1 && key[v] < min) {
      min = key[v], min_index = v;
    }
  }

  return min_index;

}


void print_min_span_tree(int parent[],
                         int vertex_count,
                         int **graph) {

  std::cout<<"Edges\n";
  std::cout<<"Vertex1"<<"\t"<<"Vertex2"<<"\n";

  for (int i = 1; i <  vertex_count; i++) {
    std::cout<<parent[i]<<"\t"<<i<<"\t"<<"\n";
  }

}
```

```
void prims_min_span_tree(int **graph, int vertex_count){

  /**
   * Parent's and keys
   */
  int parent[vertex_count];
  int key[vertex_count];

  /**
   *  Using an array for the priority queue
   */
  int priority_queue[vertex_count];

  /**
   * Initially set all keys to infinity - A large value instead
   * All values in priority_queue are initially filled with vertex ids
   * when used up we set it to -1
   */
  for (int i = 0; i < vertex_count; i++) {
    key[i] = 9999;
    priority_queue[i] = i;
  }

  /**
   * Setting first node to have a key 0 and
   * no parent
   */
  key[0] = 0;
  parent[0] = -1;

  int count=0;
  while(count < vertex_count-1){

    /**
     * Get element in the priority queue with minimum key
     * We are using u - set priority_queue to -1
     */
    int u = get_min_key(key, priority_queue, vertex_count);
    priority_queue[u] = -1;


    for (int v = 0; v < vertex_count; v++)

      /**
       * Conditions:
       * (i) graph[u][v] has a non zero value if u and v are adjacent
       * (ii) v has to be present in the priority queue
```

```
        * (iii) weight(u,v) has to be smaller than the key[v]
        */
       if (graph[u][v] && priority_queue[v] != -1 && graph[u][v] < key[v]) {
         parent[v] = u, key[v] = graph[u][v];
       }

     count++;
   }

   print_min_span_tree(parent, vertex_count, graph);
}



int main() {

   int vertices;

   std::cout<<"Enter the number of vertices: ";
   std::cin>>vertices;

   int **_graph = new int*[vertices];

   for(int i=0; i<vertices; i++){
     _graph[i] = new int[vertices];
   }


   for(int i=0; i<vertices; i++){
     std::cout<<"Enter the connections for vertex "<<i<<" (Please separate weights by a space): ";
     for(int j=0; j<vertices; j++){
       std::cin>>_graph[i][j];
     }
   }

   prims_min_span_tree(_graph, vertices);

}
```

1.2 Output

```
~/code/explore-algorithms-cpp/hw4 $ g++ prims_matrix.cpp
~/code/explore-algorithms-cpp/hw4 $ ./a.out
Enter the number of vertices: 5
Enter the connections for vertex 0 (Please separate weights by a space): 0 2 0 6 0
Enter the connections for vertex 1 (Please separate weights by a space): 2 0 3 8 5
Enter the connections for vertex 2 (Please separate weights by a space): 0 3 0 0 7
Enter the connections for vertex 3 (Please separate weights by a space): 6 8 0 0 9
```

```
Enter the connections for vertex 4 (Please separate weights by a space): 0 5 9 7 0

Edges
Vertex1         Vertex2
0         1
1         2
0         3
1         4
```

2 Qn1-V2. Prims Algorithm: Adjacency List Implementation

2.1 Code

```
#include <vector>
#include <iostream>
#include <bits/stdc++.h>

using namespace std;


/**
 *  Class for the graph of which
 *  Prim's would be a function
 */
class Graph {

  int vertices;

public:

  /**
   *  Adjacency list as a vector of vector for storing paths.
   *  The i'th element in the outside vector represents the path's from vertex i
   *
   *  The vector holds a vector of 'pair' of  values.
   *
   *  Every 'pair' indicates an edge
   *  First elment of the pair: Destination vertex
   *  Second element of the pair: Weight of the edge
   */
  std::vector<std::vector<std::pair<int, int>>> adjacencyList;


  /**
   * Constructor and functions to be defined
   */
  Graph(int vertices);
```

```
  void create_edge(int u, int v, int weight);

  void prims_min_span_tree();

};



Graph::Graph(int vertex_count) {

  this->vertices = vertex_count;

  /**
   * Intializing the adjacency list.
   * We have 'n' elements in the list where n is the number of vertices.
   * At this point we have only vertices and no edges defined.
   */
  for(int i=0; i<vertex_count; i++){
    std::vector<std::pair<int, int>> p;
    adjacencyList.push_back(p);
  }

}


void Graph::create_edge(int u, int v, int weight){

  /**
   *  Creating an edge - when we do this we create a connection from
   *  u to v and one from v to u - Hence two pushbacks at two indices in the
   *  adjacency list.
   */
  std::pair<int, int> adjListNode = std::make_pair(v, weight);
  this->adjacencyList[u].push_back(adjListNode);
  this->adjacencyList[v].push_back(adjListNode);

}

void Graph::prims_min_span_tree(){

  /**
   *  Using the Built-in heap from the STL
   *  as a priority queue utility
   */
  priority_queue<pair<int, int>,
                 vector<pair<int, int>>,
                 greater<pair<int, int>>> priorityQueue;
```

```
int src = 0;
/**
 * Initialize the keys for all vertices with a very large value
 * to represent infinity
 */
vector<int> key(this->vertices, 9999);
/**
 *  Set all parents to -1 initially
 */
vector<int> parent(this->vertices, -1);

/**
 * Track the vertices we've taken in the Minimal Spanning Tree.
 * When ith element is added, we set the i'th element of the
 * vector to True.
 */
vector<bool> hasAdded(this->vertices, false);

/**
 * Adding to the priority queue
 */
priorityQueue.push(make_pair(0, src));
key[src] = 0;

while (!priorityQueue.empty())
  {
    int u = priorityQueue.top().second;
    priorityQueue.pop();

    hasAdded[u] = true;

    /**
     * Vector iterator to iterate through the connected nodes
     */
    vector<pair<int, int>>::iterator i;
    /**
     * Adjacency List (vector) pointing to the u-th vertex
     */
    std::vector<std::pair<int, int>> l = this->adjacencyList[u];

    for(int i=0; i<l.size(); i++)
      {
        pair<int, int> _edge = l[i];

        /**
         * Since destination and weight are our first and second elements in
         * the 'pair'.
         */
```

```
            int v = std::get<0>(_edge);
            int weight = std::get<1>(_edge);

            if (hasAdded[v] == false && key[v] > weight)
              {
                key[v] = weight;
                priorityQueue.push(make_pair(key[v], v));
                parent[v] = u;
              }
          }
      }

  /**
   * Finish - Print the vertices
   */
  cout << "Source" << "\t" << "Destination"<<"\n";
  for (int i = 1; i < this->vertices; ++i)
    cout << parent[i] << "\t" << i<<"\n";
}


int main(){

  int vertices;
  int edge_count;
  std::cout<<"Enter the number of vertices: ";
  std::cin>>vertices;
  std::cout<<"Enter the number of edges: ";
  std::cin>>edge_count;

  Graph  g(vertices);
  int a[3];
  for(int i=0; i<edge_count; i++){
    std::cout<<"Enter edge "<<i<<" in the format <source destination weight>: ";
    for(int j=0; j<3; j++){
      std::cin>>a[j];
    }
    g.create_edge(a[0],a[1],a[2]);
  }

  g.prims_min_span_tree();

}
```

2.2 Output

```
~/code/explore-algorithms-cpp/hw4 $ g++ prims_list.cpp
~/code/explore-algorithms-cpp/hw4 $ ./a.out
```

```
Enter the number of vertices: 5
Enter the number of edges: 7
Enter edge 0 in the format <source destination weight>: 0 1 2
Enter edge 1 in the format <source destination weight>: 1 2 3
Enter edge 2 in the format <source destination weight>: 2 4 7
Enter edge 3 in the format <source destination weight>: 1 4 5
Enter edge 4 in the format <source destination weight>: 1 3 8
Enter edge 5 in the format <source destination weight>: 3 4 9
Enter edge 6 in the format <source destination weight>: 0 3 6
Source        Destination
0        1
1        2
0        3
1        4
```