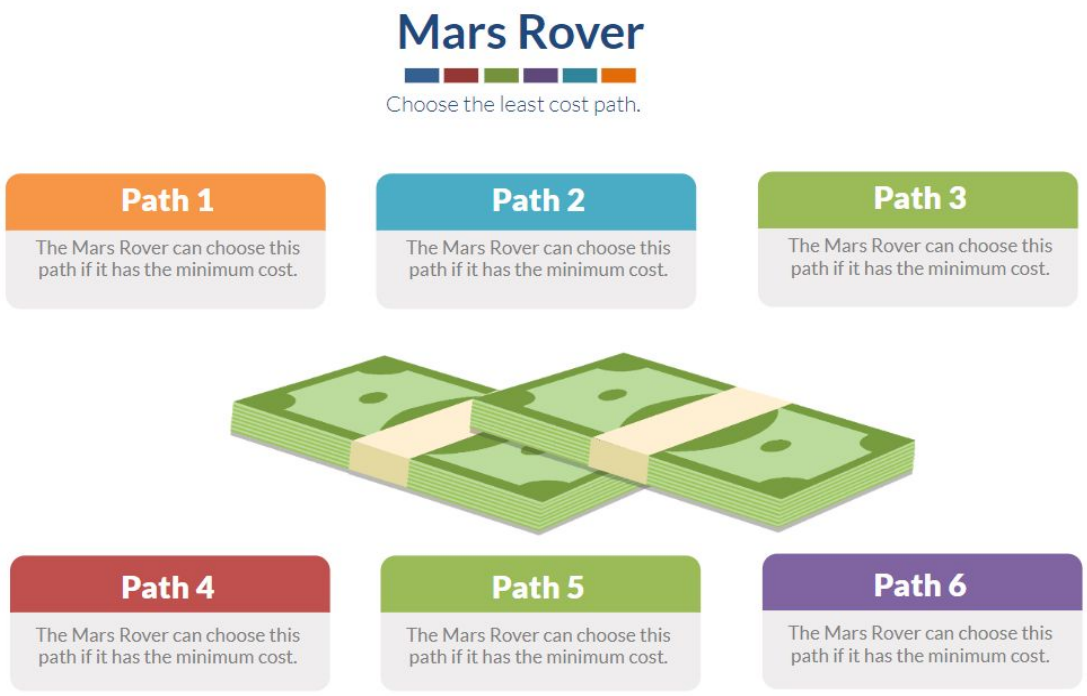
Prim’s Algorithm

Prim’s Algorithm can be used to find the cost of a particular path of the Mars Rover. If multiple paths of the same length exist, that is the shortest path out of all possible paths the Rover can take, then the Rover can choose the path that has the minimum cost using this algorithm.

The time complexity of this path is O(VlogV+ElogV).



Program:

//Header Section

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

using namespace std;

//Class

class Edge
```

```
{
    public:
    int nbr;
    int wt;
};

vector<vector<Edge>> graph;

//Functions

void addedge ( int v1, int v2, int wt)
{
    Edge e1;
    e1.nbr = v2;
    e1.wt = wt;
    graph[v1].push_back(e1);

    Edge e2;
    e2.nbr = v1;
    e2.wt = wt;
    graph[v2].push_back(e2);
}

void addedge(vector<vector<Edge>>& g, int v1, int v2, int wt)
{
    Edge e1;
    e1.nbr = v2;
    e1.wt = wt;
    g[v1].push_back(e1);

    Edge e2;
    e2.nbr = v1;
    e2.wt = wt;
    g[v2].push_back(e2);
}

//Class

class Ppair
{
    public:
    int v;
    int av;
```

```
    int c;

    Ppair ( int v, int av, int c)
    {
        this -> v = v;
        this -> av = av;
        this -> c = c;
    }

    bool operator<(const Ppair& other) const
    {
        return this -> c < other.c;
    }

    bool operator>(const Ppair& other) const
    {
        return this -> c > other.c;
    }
};

//Functions

void display ( vector<vector<Edge>>& g)
{
    for(int v = 0 ;  v < g.size() ; v++)
    {
        cout<< v << " -> ";
         for(int n=0; n < g[v].size() ; n++)
         {
             Edge ne = g[v][n];
             cout<< " [ "<<ne.nbr <<","<< ne.wt <<" ]";
         }
        cout<<"."<<endl;
    }
}

int counter =0;

void prims ()
{
    vector<vector<Edge>> mst (graph.size() , vector<Edge>());
    priority_queue< Ppair , vector<Ppair> , greater<Ppair>> pq;
    vector<bool> visited ( graph.size() , false);
```

```
    Ppair rtp (0 , -1 , 0);
    pq.push(rtp);

    while( pq.size() > 0 )
    {
      Ppair rem = pq.top();
      pq.pop();

      if(visited[rem.v] == true )
      {
        continue;
      }

      visited[rem.v] = true;


      if(rem.av  !=  -1 )
      {
        addedge(mst , rem.av, rem.v, rem.c);
      }

      for ( int n = 0 ; n < graph[rem.v].size() ; n++ )
      {
          Edge ne = graph[rem.v][n];
          if( visited[ne.nbr] == false)
          {
              Ppair np (ne.nbr , rem.v , ne.wt);
              pq.push(np);
          }
      }
    }

    display ( mst );
}

//Main Function

int main( int argc, char** argv)
{
    graph.push_back(vector<Edge>());
    graph.push_back(vector<Edge>());
    graph.push_back(vector<Edge>());
```

```
    graph.push_back(vector<Edge>());
    graph.push_back(vector<Edge>());
    graph.push_back(vector<Edge>());
    graph.push_back(vector<Edge>());

    addedge(0,1,20);
    addedge(1,2,10);
    addedge(2,3,20);
    addedge(0,3,40);
    addedge(3,4,2);
    addedge(4,5,3);
    addedge(5,6,3);
    addedge(4,6,8);

    display ( graph );

    cout << endl;

    prims ();
}
```