

1 Problem Formulation

The problem can be modeled as the shortest-path problem in the graph theory. The objective of the problem is to find a path between two nodes in a weighted digraph such that no other path has a lower total weight. The nodes represent (x,y) coordinates of the locations in the map. Some of neighbor nodes are connected by edges if the rover can move between them. This information is obtained from the file 'overrides.data'. In addition, each edge has a weight that measures the cost of travelling between two connected nodes. The weights are computed using the constraints imposed on the rover's speed. The problem states that the rover's speed is subject to the direction of its movement and the change in elevation. The elevation of the terrain is read from the file 'elevation.data'.

Let $G=(V,E)$ be a digraph with node set V and edge set E . The edges are represented as ordered pairs to indicate the direction, i.e., $E \subseteq \{(u, v) \mid (u, v) \in V^2 \wedge u \neq v\}$, where (u,v) is the directed edge from a node u to another node v . Each edge is associated to a weight. We assume that G has no negative weight cycles¹ to find shortest paths. Therefore, a positive real-valued function, i.e., $f : E \rightarrow \mathbb{R}^+$ is defined to weight edges as follows:

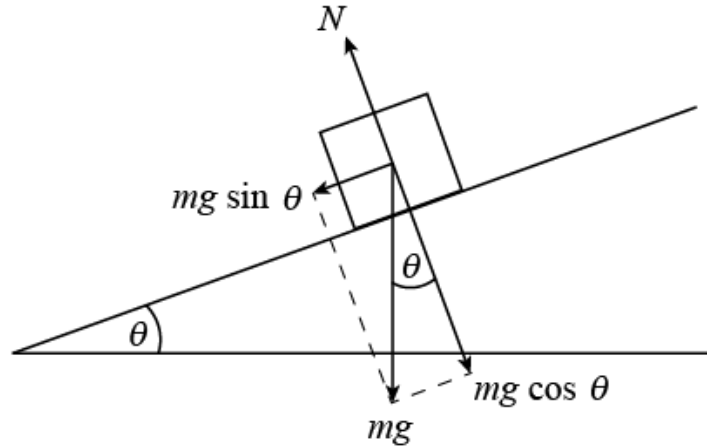


Figure 1: Acceleration of Gravity

Fig. 1 depicts the gravitational force and its components. The parallel component of the gravitational force ($mg \sin \Theta$) accelerates or decelerates the rover, depending on the direction of the slope. This force is zero when the rover moves on a plane surface. There are two unknowns: First, the amount of gravitational acceleration, i.e., g must be defined as a design parameter in island time. Since elevation is unitless, the slope angle Θ cannot be computed. However, if we know the rover's climbing limits, we can compute that angle. Let Φ be the maximum slope that the rover can climb up/down safely. In a worst-

¹ A *negative weight cycle* is a directed cycle whose total weight is negative. The concept of a shortest path is meaningless if there is a negative cycle. This is because that the shortest path will contain infinite number of loops around the negative weight cycle.

case scenario, assume that the max slope angle is Φ and it is represented as 255 in the file 'elevation.data'. Then, the slope angle is computed as

$$\Theta = \frac{\Phi}{255} \Delta h,$$

where Δh is the absolute change in elevation. Applying the Newton's second law, i.e., $ma = mg \sin \Theta$ gives the acceleration $a = g \sin \Theta$. When the rover climbs up/down a hill, the distance equation is written as follows:

$$\mathcal{G}t \pm \frac{1}{2}at^2 = \ell \sec \Theta,$$

where \mathcal{G} is the rover's speed (i.e., 1 cell/island second) and ℓ is the distance travelled (1 cell for straight movement or $\sqrt{2}$ cells for diagonal movement). The sign between the two terms on the left hand side is determined by the direction of the slope. The function f solves the quadratic equation above for t given that $t \in \mathbb{R}^+$.

The shortest path from a source node s is a directed subgraph of G , i.e., $G' = (V', E')$ rooted at s such that

- $V' \subseteq V$ is the set of nodes reachable from s in G
- $G' \subseteq G$ forms a rooted tree with root s
- For all $v \in V'$, the unique, simple path from s to v in G' is the shortest path from s to v in G .

2 Data Structure

We can define a class to represent directed edges as:

```
#ifndef DIRECTEDGE_H_
#define DIRECTEDGE_H_

class DirectedEdge{

    int node0;           // node number of the head node
    int node1;           // node number of the tail node

    double weight;       // weight of the ordered edge

public:
    DirectedEdge(int n0, int n1, double w): node0(n0), node1(n1), weight(w) {}

    // returns the head node of the ordered edge
    int from();

    // returns the tail node of the ordered edge
    int to();

    // returns the weight of the ordered edge
```

```

    double getWeight();
};

#endif /* DIRECTEDGE_H_ */

```

Table 1: DirectedEdge.h

```

#include "DirectedEdge.h"

int DirectedEdge::from() { return node0; };

int DirectedEdge::to() { return node1; };

double DirectedEdge::getWeight() {return weight; }

```

Table 2: DirectedEdge.cpp

Using the DirectedEdge we can define the following class to represent a weighted digraph:

```

#include <vector>
#include <map>
#include <string>
#include <fstream>
#include <sstream>
#include "DirectedEdge.h"

using std::vector;
using std::map;
using std::string;
using std::ifstream;

#ifndef DIGRAPH_H_
#define DIGRAPH_H_

class Digraph{

    int numV;        // number of nodes in this digraph
    int numE;        // number of edges in this digraph

    // adjacency list to represent the weighted digraph
    map<int,vector<DirectedEdge> > adjlist;

public:
    /*
     * Initializes a weighted digraph from the specified input file:
     * The first line is the number of nodes numV,
     * the next line is the number of edges numE,
     * the following lines are the ordered pairs
     * of nodes and their weights with each entry
     * separated by whitespace., i.e., u, v, w for
     * a edge from node u to node v with weight w.
     *
     * throws runtime exception  if the input file cannot be cannot be opened.
     */

```

```

    Digraph(string);

    /*
     * Adds a directed edge to the digraph.
     *
     * throws argument exception if the tail and head nodes are not in the range.
     * throws argument exception if the number of vertices or edges is negative.
     */
    void addEdge(DirectedEdge);

    /*
     * Returns the number of nodes in this weighted digraph.
     */
    int getV();

    /*
     * Return the number of edges in this weighted digraph.
     */
    int getE();

    /*
     * Return the directed edges of a node
     */
    vector<DirectedEdge> getEdges(int);

};

#endif /* DIGRAPH_H_ */

```

Table 3: Digraph.h

```

#include "Digraph.h"

Digraph::Digraph(string filename): numV(0), numE(0)
{
    string line;        // line
    int linenum = 0;    // line number

    int n0, n1;        // head and tail nodes
    double w;           // weight of the edge

    // open the file for reading
    ifstream infile (filename.c_str());

    // check if the file is open
    if ( infile.is_open() )
    {
        // Read the next line from File until it reaches the end.
        while (std::getline(infile, line))
        {
            // string stream object to tokenize each lines
            std::istringstream iss(line);

```

```
        // read the number of nodes
        if (linenum == 0){
            iss >> numV;

            if (numV < 0)
                throw std::invalid_argument("number of edges
cannot be negative");
        }

        // read the number of edges
        else if (linenum == 1) {
            iss >> numE;

            if (numE < 0)
                throw std::invalid_argument("number of nodes
cannot be negative");
        }

        // read the ordered pairs and weight to add a directed edge
        else
        {
            iss >> n0 >> n1 >> w;

            addEdge(DirectedEdge(n0, n1, w));

        }

        ++linenum;
    }

    infile.close();
}
else
    throw std::runtime_error("could not the open file");
}

void Digraph::addEdge(DirectedEdge e)
{
    int n0 = e.from();
    int n1 = e.to();
    double w = e.getWeight();

    // validate the directed edge e
    if (n0 == n1)
        throw std::invalid_argument("nodes cannot be same");

    else if (n0 < 0 || n0 >= numV)
        throw std::invalid_argument("invalid head node" );

    else if (n1 < 0 || n1 >= numV)
        throw std::invalid_argument("invalid tail node");

    else if (w < 0 )
```

```

        throw std::invalid_argument("negative edge weight");

        // add the edge e from node n0 to node n1
        adjlist[n0].push_back(e);
    }

    int Digraph::getV() { return numV; }
    int Digraph::getE() { return numE; }

    vector<DirectedEdge> Digraph::getEdges(int u)
    {
        return adjlist[u];
    }

```

Table 4: Digraph.cpp

3 Single-Source Shortest Path Algorithm

There are two variants of the shortest-path problem. The problem at hand is the single-source shortest path problem since the source nodes are known. In the single-source shortest path, we want to compute the distance from a given source node to every other reachable nodes. There are many algorithms to solve this problem, but the implementations of the most shortest-path algorithms are based on the relaxation technique, which repeatedly decreases an upper bound on the total weight of the shortest path until the upper bound equivalent to the shortest-path weight.

For each node $v \in V$, we compute the upper-bound on the total weight of the shortest path from source node s to v , i.e., $d[v]$. We call $d[v]$ the shortest path estimate. In addition, we define predecessor of each node $\pi[v]$ that is another node if there exists a directed edge between them or NULL, otherwise. After the initialization of the predecessors and the shortest-path estimates according to

$$\pi[v] = -\infty \quad \forall v \in V,$$

$$d[v] = \begin{cases} 0 & \text{if } v = s, \\ \infty & \text{if } v \in V - \{s\}, \end{cases}$$

The relaxing an edge (u, v) means to test whether we can improve the shortest path to v found so far by going through u and if so, updating $d[v]$ and $\pi[v]$, i.e.,

$$d[v] > d[u] + w_{(u,v)}$$

where $w_{(u,v)}$ is the weight of the directed edge from u to v .

A relaxation step may decrease the value of the shortest-path estimate $d[v]$ and update v 's predecessor $\pi[v]$:

if $d[v] > d[u] + w_{(u,v)}$
 then $d[v] = d[u] + w_{(u,v)}$ and $\pi[v] = u$.

If the graph consists of nonnegative weights, we can employ Dijkstra's algorithm to solve the single-source shortest path problem. Dijkstra's algorithm maintains a set S of nodes whose shortest-path weights from the source node have already been determined. The algorithm repeatedly selects a node $u \in V - S$ with the minimum shortest-path estimate, insert u into S and relaxes all edges leaving u . Since the algorithm always selects the closest node in $V - S$ to insert it into set S , it is called as the greedy strategy.

```
#ifndef DIJKSTRASP_H_
#define DIJKSTRASP_H_

#include "Digraph.h"
#include <set>
#include <limits>

using std::multiset;

class DijkstraSP{

    vector<double> distTo;    // distTo[v] = distance of shortest s->v path
    vector<int> edgeTo;      // edgeTo[v] = last edge on shortest s->v path
    // min-priority queue in the form (weight, node)
    multiset<std::pair<double, int> > pq;
    vector<bool> visNodes;   // vector of visited nodes
    const double inf;       // positive infinity

public:

    /* Solves the single-source shortest paths problem for a weighted digraphs
     * with positive weights using the Dijkstra's shortest path algorithm. This
     * implementation of the algorithm uses a binary min-heap.
     */
    DijkstraSP(Digraph, int);

    /*
     * Relax the distances of the connected nodes to the popped node in case of
     * current node distance + edge weight < next node distance, then push the
     * new node with its new distance to min-priority queue.
     */
    void relax(DirectedEdge);

    /*
     * Returns true if there is a path from the source s to node v
     */
    bool hasPathTo(int);

    /*
     * Returns a shortest path (node numbers) from the source s to node v
     */
}
```

```

    */
    vector<int> pathTo(int);

    // throw invalid argument exception unless 0 <= v < V
    void validateVertex(int);

};

#endif

```

Table 5: DijkstraSP.h

```

#include "DijkstraSP.h"
#include <iostream>

DijkstraSP::DijkstraSP(Digraph G, int s)
    : distTo(G.getV()),
      edgeTo(G.getV(), -1),
      visNodes(G.getV(), false),
      inf(std::numeric_limits<double>::infinity())
{

    // set all d[v] = infinity except that d[s] = 0
    vector<double>::iterator it;
    for (it = distTo.begin(); it != distTo.end(); ++it)
    {
        *it = inf;
    }
    distTo[s] = 0;

    // relax vertices in order of distance from s
    pq.insert(std::make_pair( distTo[s], s));

    while (!pq.empty())
    {
        // pop the node with the minimum weight
        std::pair<int , int> p = *pq.begin();

        // remove the element with minimum weight
        pq.erase(pq.begin());

        for (DirectedEdge & e : G.getEdges(p.second))
            relax(e);

        // set visit flag true for the selected node
        visNodes[p.second] = true;
    }
}

void DijkstraSP::relax(DirectedEdge edge)
{
    // head and tail nodes
    int n0, n1;

```



```
// weight of edge
double w;

n0 = edge.from();
n1 = edge.to();
w = edge.getWeight();

// check node n1 visited before
if (!visNodes[n1])
{
    // check the relaxation condition
    if (distTo[n1] > distTo[n0] + w)
    {
        // update the weight of path to s
        distTo[n1] = distTo[n0] + w;

        // update the predecessor of n1
        edgeTo[n1] = n0;
    }
    // insert the next vertex with the updated distance
    pq.insert(std::make_pair(distTo[n1], n1));
}
}

void DijkstraSP::validateVertex(int v)
{
    int V = distTo.size();

    if (v < 0 || v >= V)
        throw std::invalid_argument("invalid node");
}

bool DijkstraSP::hasPathTo(int v)
{
    validateVertex(v);

    return distTo[v] < inf ? true : false;
}

vector<int> DijkstraSP::pathTo(int v)
{
    // shortest path
    vector<int> path;

    if (!hasPathTo(v))
        return path;

    path.push_back(v);
}
```

```

    for (int e = edgeTo[v]; e != -1; e = edgeTo[e])
    {
        path.push_back(e);
    }

    return path;
}

```

Table 6: DijkstraSP.cpp

The time complexity of the Dijkstra's algorithm depends how the closest node is selected from the set $V - S$. Let $|V|$ and $|E|$ be the number of nodes and edges in our weighted digraph, respectively. The simplest implementation can use an unsorted array. Since insert and extract operations takes linear time, i.e., $O(|V|)$, its worst time complexity to visit all nodes is given by $O(|V|^2)$ time. However, the more efficient data structures like multiset class can reduce the complexity. The insert and extract operations of the multiset takes $O(\log|V|)$ and thus the worst time complexity reduces to $O((|V|+|E|)\log|V|)$.

4 Conclusion and Examples

The problem defined in Bachelor task can be solved using the Dijkstra's algorithm. Dijkstra's shortest path class (DijkstraSP) takes a digraph with positive edge weights, and a source node as arguments. The digraph object is instantiated by reading line by line the number of nodes, the number of edges, and the tuples of (head node, tail node, weight) in a text file. Therefore, the first step solving to the problem is to generate that text file using the given data files and the function $f : E \rightarrow \mathbb{R}^+$ defined in section 1. After solving the shortest paths to all nodes in the digraph, the public member function hasPathTo check if a target node t is reachable from source node s . The another member function pathTo returns the shortest path found from s to t . For the given problem, we need to call DijkstraSP object twice for the paths: *i*) the rover's node to the bachelor's node, and *ii*) the bachelor's node to the weddings node, respectively. Note that the nodes can be numbered by row-by-row ordering cells of 2048×2048 data matrices.

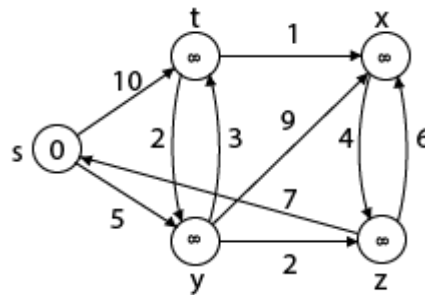
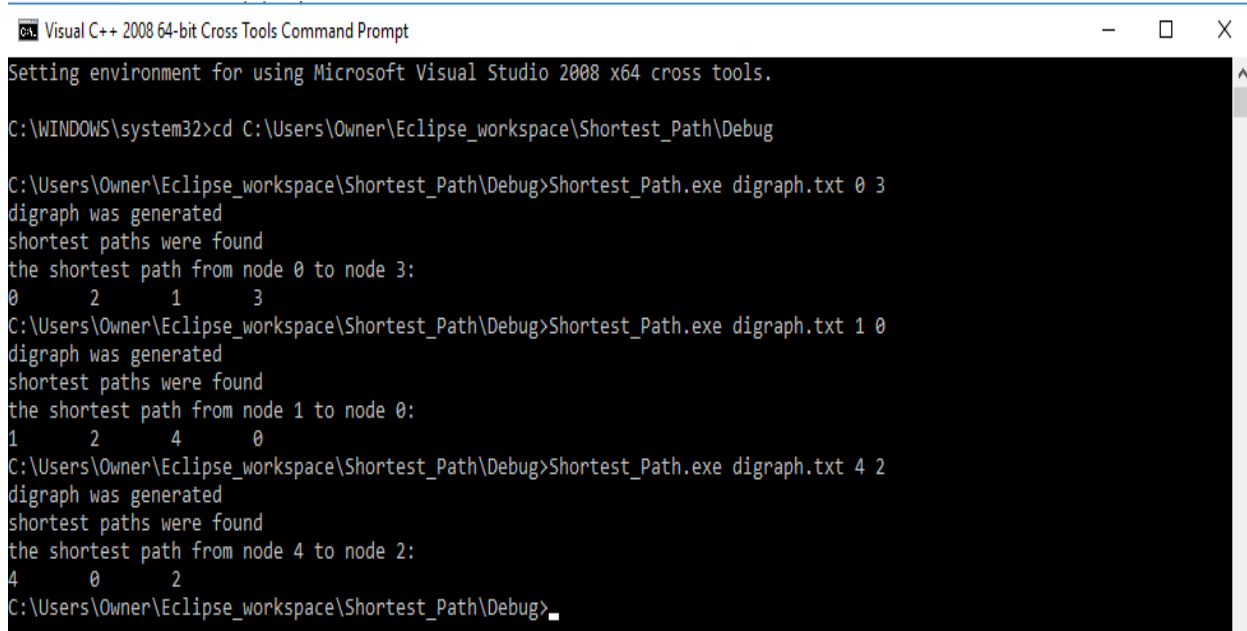


Figure 2: Digraph of nodes s , t , y , x and z with their positive edge weights. The nodes are numbered as follows: $s=0$, $t=1$, $y=2$, $x=3$ and $z=4$.

To validate the proposed Dijkstra's algorithm, we design a unit test. The problem is defined to find the shortest paths for the digraph in Fig. 2. Initially, the distances to the source node s from all nodes are set to infinity except the source node whose distance is set to zero. Note that when the .exe file is called with arguments as text file, source node, target node in the command prompt, the user can set the source and target nodes. Otherwise, the Test.cpp runs the algorithm for default values. Fig. 3 shows some results for the weighted digraph given in Fig. 2.



```
Visual C++ 2008 64-bit Cross Tools Command Prompt
Setting environment for using Microsoft Visual Studio 2008 x64 cross tools.

C:\WINDOWS\system32>cd C:\Users\Owner\Eclipse_workspace\Shortest_Path\Debug

C:\Users\Owner\Eclipse_workspace\Shortest_Path\Debug>Shortest_Path.exe digraph.txt 0 3
digraph was generated
shortest paths were found
the shortest path from node 0 to node 3:
0      2      1      3
C:\Users\Owner\Eclipse_workspace\Shortest_Path\Debug>Shortest_Path.exe digraph.txt 1 0
digraph was generated
shortest paths were found
the shortest path from node 1 to node 0:
1      2      4      0
C:\Users\Owner\Eclipse_workspace\Shortest_Path\Debug>Shortest_Path.exe digraph.txt 4 2
digraph was generated
shortest paths were found
the shortest path from node 4 to node 2:
4      0      2
C:\Users\Owner\Eclipse_workspace\Shortest_Path\Debug>
```

Figure 3: The results of some unit tests.