

KIT205 Data Structures and Algorithms Assignment 3

Due 28th May, 11:55pm

Introduction

You are a member of a team developing a mars rover. A Mars rover does not need to be completely autonomous, but due to the time delay it does help if the rover can be at least semi-autonomous. Due to the extreme cost of getting a rover to Mars, the programming needs to be extremely risk averse.

Your job is to develop the path-finding system and it is crucially important that you minimise risk. The biggest risk is tipping over (due to weight restrictions, rovers cannot usually right themselves).

Given a digital elevation map of the terrain, you must identify the path between two points that minimises risk.

Assignment Specification

For this assignment, you will be given a digital elevation map (DEM) and an equation for calculating the risk in moving from one elevation to another adjacent elevation. The risk is based on the slope (different in elevation). To simplify the task you consider only the discrete positions and elevations represented by the DEM and will consider only moves in the four cardinal directions (NSEW).

You will be provided with a Visual Studio Project that contains some code to get you started.

```
int** make_dem(int size, int roughness);
```

*This function generates a 2D array of ints representing the heights of a square area of the terrain. The size should be one more than a power of 2 (e.g. 9, 17, 33, 65) and roughness values around 4*size work well. You will also notice that the code sets a random seed - this can be set to a fixed value for testing purposes (to ensure that you always get the same map).*

```
int risk_func(int diff);
```

This function is used to calculate the risk in moving between two adjacent points in the DEM with the given height difference.

```
void print_2D(int** array2D, int size);  
void print_2D_ascii(int** array2D, int size);
```

These two functions can be used for printing DEMs and markers. These functions take a DEM where heights are expected to be in the range 0-99. Any negative value will be printed as a marker. You should use these functions to visualise your solution (setting the height of positions that are on your path to -1). You can use either print function. The first function gives more detail, but the second is a bit easier to interpret. (NOTE: these functions print with the y axis going across the page and the x axis going down)

Your first task is to use these functions to generate an adjacency list graph representation of the terrain. Start by generating a DEM, then use the risk function to build the weighted graph. The vertex number of a given x,y position in the map should be calculated as $x \cdot \text{size} + y$.

So, for example, the following DEM:

```
49 51 51 56 57  
46 47 50 52 54  
45 47 48 50 49
```

15 17 18 30 17
46 46 47 47 47
45 45 45 45 47

Would result in an edge from vertex 0 to vertex 5 (in bold) with weight calculated using risk_func(46-49).

You must use the following data structures to represent the graph as an adjacency list:


```
typedef struct edge{
    int to_vertex;
    int weight;
} Edge;

typedef struct edgeNode{
    Edge edge;
    struct edgeNode *next;
} *EdgeNodePtr;

typedef struct edgeList{
    EdgeNodePtr head;
} EdgeList;

typedef struct graph{
    int V;
    EdgeList *edges;
} Graph;
```

Part A

For part A you will use risk_func to generate the adjacency list for the DEM. You should then implement Dijkstra's algorithm based on the pseudocode on Wikipedia (https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm 

While a priority-queue implementation would be best for our scenario, you should instead use a simple search to find the nearest vertex.

Your submission must contain a main function with the following structure:

- generate a DEM
- convert it to an adjacency list, G, using risk_func
- calculate the shortest paths from vertex a given vertex by calling a function of the form:
 - `dijkstra(&G, start, distance, previous);`
 - Where G is the graph, *distance* is an array that will be used to store the distances calculated, and *previous* will be used to store the path information
- reconstruct the path the start vertex to the given destination vertex using:
 - `get_path(end, previous, &path);`
 - Where *end* is the destination vertex, *previous* is the array modified by the *dijkstra* function, and *path* will be a linked list of vertices representing the shortest path from *start* to *end*.
- create a copy of the DEM with the heights changed to -1 for any vertex in *path*
- print the DEM to show the path

Part B

For part B you will write code to allow a greater degree of autonomy. Instead of finding a single path, you will be given a list of points of interest. You must then write code to find a path of minimum risk that visits all of these points starting

with the first vertex (but not necessarily in the order given after that).

If you attempt this part, you should add the following code to your main function *after* the code for Part A (Part B should NOT be in a separate project). The new code should:

- construct a linked list of points of interest
 - You can make this an EdgeList, ignoring the edge weights, or use a separate linked list of ints
 - Make it easy for the marker to modify the points of interest
- calculate the optimal path for the points in the list (based on the same DEM as for part A)
 - You may use your previous Dijkstra code to construct a new graph containing only the points of interest with the edge weights representing the calculated shortest path distances.
 - You may represent this new graph as an adjacency *matrix*
 - You may use other data structures that you implement
- create a copy of the DEM with the heights changed to -1 for any vertex in *path*
- print the path

Note: this is a version of the travelling salesman problem, which is NP-hard (see [week 12 lecture](#)). This means that it will take a **very** long time to run for more than a few points of interest. Your code only needs to work for a few points, so you do not have to use any clever techniques or approximation techniques to find the optimal solution - just do a brute-force search.

Assignment Submission

Assignments will be submitted via MyLO (an Assignment 2 dropbox will be created). You should use the following procedure to prepare your submission:

- Make sure that your project has been thoroughly tested using the School's lab computers
- Choose "Clean Solution" from the "Build" menu in Visual Studio. This step is very important as it ensures that the version that the marker runs will be the same as the version that you believe the marker is running.
- Quit Visual Studio and zip your entire project folder
- Upload a copy of the zip file to the MyLO dropbox

History tells us that mistakes frequently happen when following this process, so you should then:

- Unzip the folder to a new location
- Open the project and confirm that it still compiles and runs as expected
 - If not, repeat the process from the start

Learning Outcomes and Assessment

This assignment is worth 15% of your overall mark for KIT205. Your submission will be assessed by examining your code and by running additional tests in Visual Studio. If your submission does not run in Visual Studio 2019 on Windows, it cannot be assessed.

Grades will be assigned in accordance with the [Assignment 3 Rubric](#)

The assignment contributes to the assessment of learning outcomes:

- LO1 Transform a real-world problem into a simple abstract form that is suitable for efficient computation**
- LO2 Implement common data structures and algorithms using a common programming language**
- LO4 Use common algorithm design strategies to develop new algorithms when there are no pre-existing solutions**

