Project Report

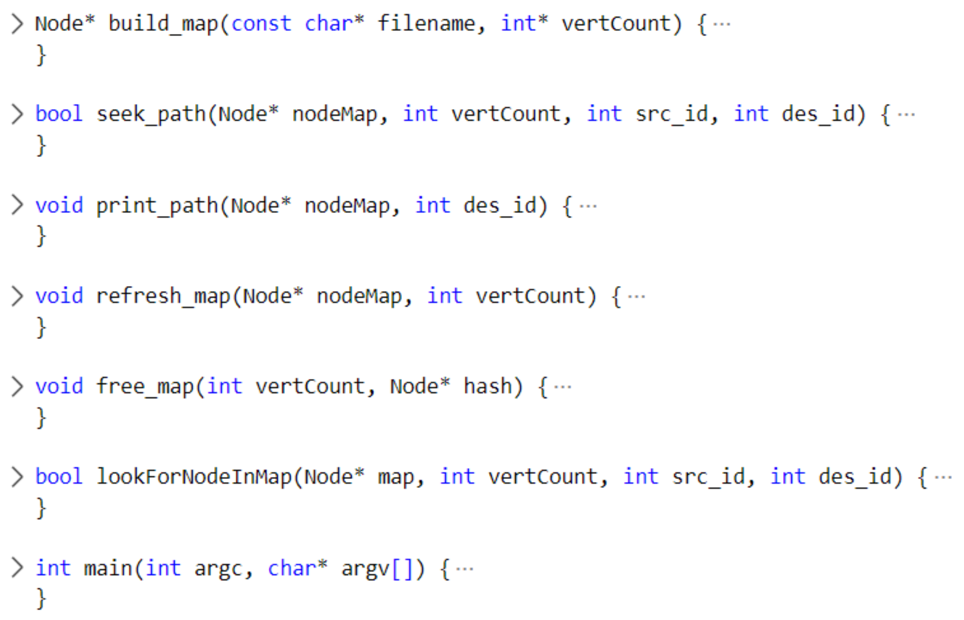
Project 3 (Map Routing), ECE368

Name: Yi En Gan (Andrew)

Login: gan35

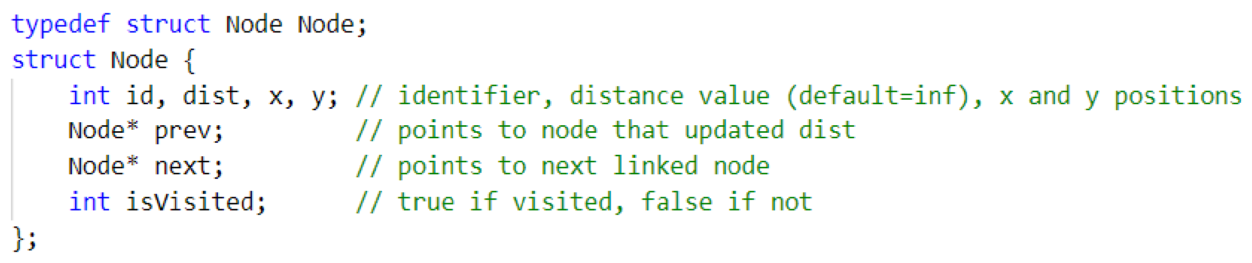
Overall approach to the problem and a summary of the solution and code.

The project utilizes Dijkstra's Algorithm to determine the shortest path from one node to another in a map structure. The pathfinding process was split into three components: building the map, finding shortest path and printing the path.



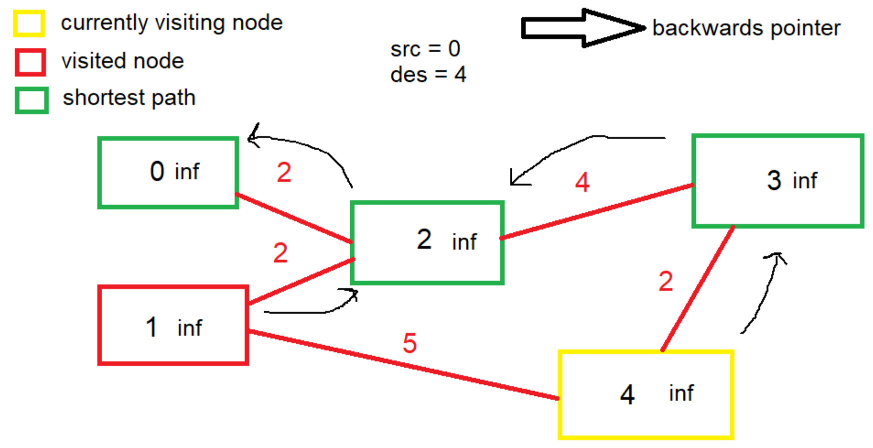
```
> Node* build_map(const char* filename, int* vertCount) {⋯
  }

> bool seek_path(Node* nodeMap, int vertCount, int src_id, int des_id) {⋯
  }

> void print_path(Node* nodeMap, int des_id) {⋯
  }

> void refresh_map(Node* nodeMap, int vertCount) {⋯
  }

> void free_map(int vertCount, Node* hash) {⋯
  }

> bool lookForNodeInMap(Node* map, int vertCount, int src_id, int des_id) {⋯
  }

> int main(int argc, char* argv[]) {⋯
  }
```

Firstly, information about the map are stored in a hash table of struct types. The hash table is a bunch of linked lists with the header node being the vertex of reference and the next nodes being the adjacent vertices.



```
typedef struct Node Node;
struct Node {
    int id, dist, x, y; // identifier, distance value (default=inf), x and y positions
    Node* prev;         // points to node that updated dist
    Node* next;         // points to next linked node
    int isVisited;      // true if visited, false if not
};
```

When a vertex is visited, all the vertices linked to it are assigned new calculated distances only if the new distance is shorter than the existing distance value. If a vertex has been visited, it is marked, and its distance is never updated again. If the distance is updated, a pointer points from vertex of updated distance to vertex of reference. This 'saves' the shortest path taken to that node.



Once the destination node has been detected, the pointer traverses backwards to the source node via the backwards pointer and the path is printed after inverting the traversed path.

Known bugs / limitations of your program / assumptions made

One unintended phenomenon is that the program may take alternative paths that are equal in distance to the expected results.
One assumption that was made is that the vertices are numbered from 0 until the total number of vertices. This required the use of linear probing or another hash table to store the hash table position to vertex id correspondence. It was ultimately decided that the program will simply look through the hash table until the id is found.

Help received

Wikipedia and GeeksForGeeks were consulted.
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Serious problems encountered

At first, it was difficult to choose what method would be best to 'store' the shortest path. It was decided that a pointer that points towards the vertex that updated its distance would be used so that the program can traverse from destination to source.

Another problem was that the distance value had to be square rooted after each calculation instead of storing it as a squared value. It was expected that this would not affect the program, but it turns out that the value may be too large to be stored as an unsigned int, causing it to overflow to a small value.

List any other comments/feedback here (e.g., whether you * enjoyed doing the exercise, it was too easy/tough, etc.).

It would be much appreciated if the assignment did not require accounting for inconveniences such as the vertex not being incrementally numbered from 0 to max. The focus of the assignment seems to be Dijkstra’s Algorithm, and creating a hash table just to locate the vertex in the map table seems like a tedious practice. This was my favorite project out of the three, and the difficulty is just right.

Example Output

Modified map5x5.txt, modified query5x5.txt

The program accounts for invalid vertices and unlinked vertices

Contents of map5x5.txt
Added node 26 but is isolated

Contents of query5x5.txt
```
5
6 23    // valid query
1 5     // valid query
2 23    // valid query
59 56   // invalid vertices
0 26    // unlinked vertices
```

Output of Program
```
7800
6 7 8 13 18 23
2900
1 0 5
8900
2 3 8 13 18 23
INF
59 56
INF
0 26
```