PART 2 AUTOROUTER

Data Structures and Algorithms

David Gray s2947315
david.gray2@griffithuni.edu.au

Table of Contents

Problem Statement

The goal of Part 2 of this assignment was to create a program to create an autorouter using recursion. A 2D grid is passed through and read and the amount of nodes must be calculated along with their type, and the location of obstacles must be recorded. Nodes of the same type must be connected without having lines cross over and without hitting end points.

User Requirements

No user requirements exist other than a computer that is able to open a '.exe' file.

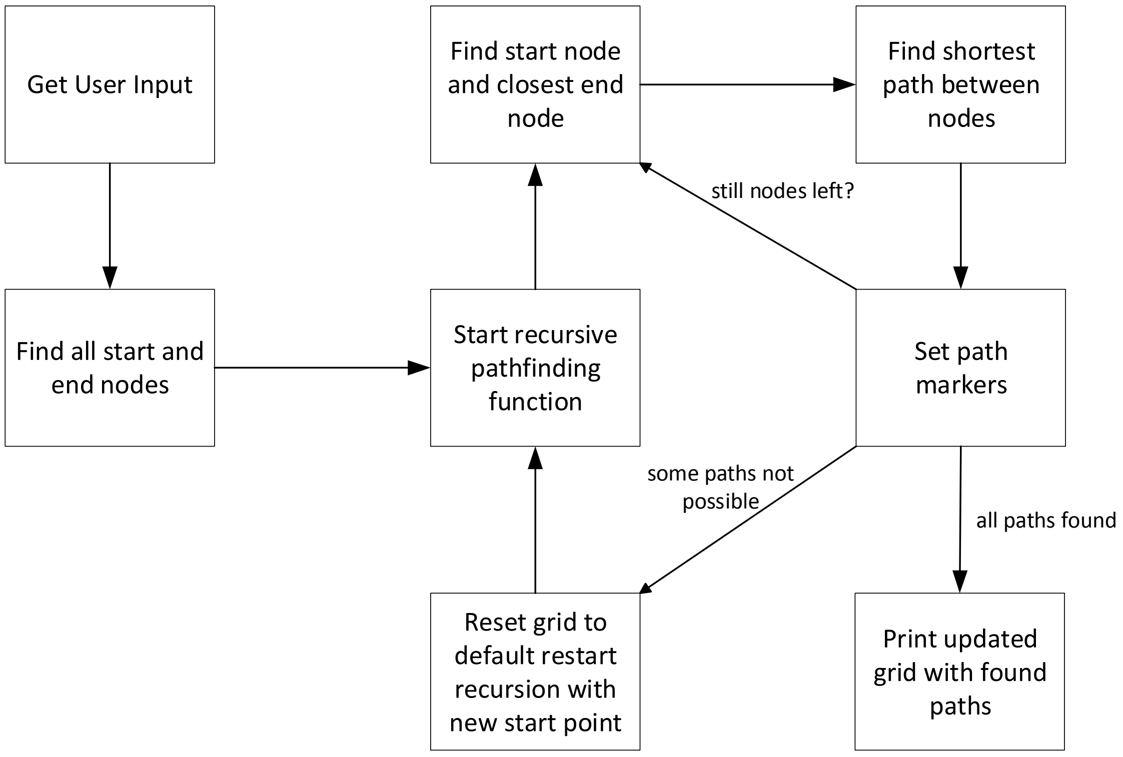
Software Requirements

The following outlines the software requirements for the program:

1. The program shall prompt the user to enter a text file name which will contain the grid that the program use.
2. The program shall find the number of node end points that need to be connected.
3. The program shall start with one node and find an end node that is of the same type and attempt to link them together, avoiding obstacles and other paths, using recursion. This process will be repeated until all nodes of the same type have been connected.
4. The program shall print out the grid showing all of the solved paths, the number of nodes connect, the number of paths, and the number of isolated nodes.

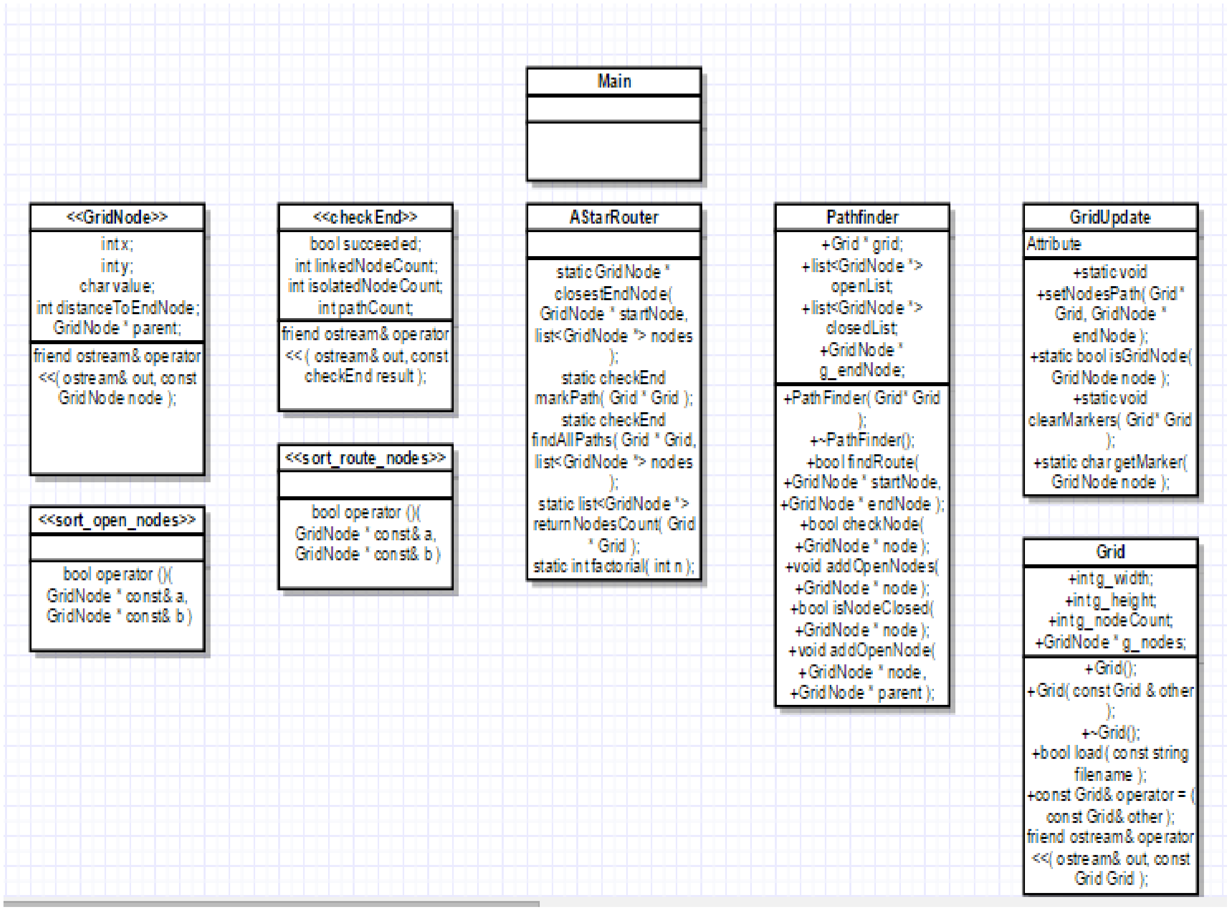
Software Design

High Level Design – Logical Block Diagram



Get User Input
Find all start and end nodes
Start recursive pathfinding function
Find start node and closest end node
Find shortest path between nodes
still nodes left?
Set path markers
some paths not possible
all paths found
Reset grid to default restart recursion with new start point
Print updated grid with found paths

Structure Chart


Main
<<GridNode>>
int x;
int y;
char value;
int distanceToEndNode;
GridNode * parent;
friend ostream& operator <<(ostream& out, const GridNode node);
<<checkEnd>>
bool succeeded;
int linkedNodeCount;
int isolatedNodeCount;
int pathCount;
friend ostream& operator << (ostream& out, const checkEnd result);
AStarRouter
static GridNode * closestEndNode(GridNode * startNode, list< GridNode *> nodes);
static checkEnd markPath(Grid * Grid);
static checkEnd findAllPaths(Grid * Grid, list< GridNode *> nodes);
static list<GridNode *> returnNodesCount(Grid * Grid);
static int factorial(int n);
Pathfinder
+Grid * grid;
+list<GridNode *> openList;
+list<GridNode *> closedList;
+GridNode * g_endNode;
+PathFinder(Grid* Grid);
+~PathFinder();
+bool findRoute(+GridNode * startNode, +GridNode * endNode);
+bool checkNode(+GridNode * node);
+void addOpenNodes(+GridNode * node);
+bool isNodeClosed(+GridNode * node);
+void addOpenNode(+GridNode * node, +GridNode * parent);
GridUpdate
Attribute
+static void +setNodesPath(Grid * Grid, GridNode * endNode);
+static bool isGridNode(GridNode node);
+static void clearMarkers(Grid* Grid);
+static char getMarker(GridNode node);
<<sort_open_nodes>>
bool operator ()(GridNode * const& a, GridNode * const& b)
<<sort_route_nodes>>
bool operator ()(GridNode * const& a, GridNode * const& b)
Grid
+int g_width;
+int g_height;
+int g_nodeCount;
+GridNode * g_nodes;
+Grid();
+Grid(const Grid & other);
+~Grid();
+bool load(const string filename);
+const Grid& operator = (const Grid& other);
friend ostream& operator <<(ostream& out, const Grid Grid);

The main functions in the software

`bool findRoute( GridNode * startNode, GridNode * endNode );`

Takes two GridNode pointers and checks to see if either a null. If so then the function returns false. From these two nodes, the distanceToEnd value is updated on every node within the grid using the Euclidean distance function basing it on the endNode x and y values and the current node being looked at x and y values. The open nodes list and closed nodes list are cleared, sets the current routes endNode, checks both the endNode and currentNode and then starts the recursion calling the checkNode function.

`static checkEnd markPath( Grid * Grid );`

Takes a Grid object and sets creates a new checkEnd object which is used to test whether or not finding all paths was successful. Using the Grid object we get the amount of total route nodes. From this the factorial of the number of route nodes is found in order to calculate the amount of possible calculations we will need to perform before we can determine if a solution can be found. A for loop is created using the permutation count from the factorial and within this loop the findAllPaths function is called which is the function used to see if all routes are found. If all routes are found then it returns the checkEnd object. If not then the grid is cleared and the next permutation is started.

`static checkEnd findAllPaths( Grid * Grid, list<GridNode *> nodes );`

Takes a Grid object and a list of GridNodes that represent the route nodes. The checkEnd result is initialized. The total number of isolated nodes is set to the amount of route nodes. A pathfinder object is then created passing through the Grid object. A Boolean is then created to mark whether or not a path is found and then a loop is created that gets the start and end nodes, increments the linked node count and decrements the isolated nodes count for each iteration through the loop where a path is found. Once all nodes have been tested then the loop breaks and the checkEnd's succeed variable is set to whether or not all paths have been found and then the checkEnd result is returned.

`static list<GridNode *> returnNodesCount( Grid * Grid );`

Takes in a Grid object as a parameter and searches through each node within the Grid and adds it to a list of route nodes if the value from the current node being checked is not equal to a space or a #. The list of the total amount of route nodes is returned.

`static GridNode *closestEndNode(GridNode *startNode, list<GridNode *> nodes );`

Takes in a GridNode object and a list of GridNodes as parameters. The object then checks to see if there is an endNode and if there is not, then it sets one. Using this endNode the distance from the endNode to the startNode is calculated using the Euclidean distance formula and then the endNode is returned. If there is already an endNode then it checks to see if there is another that is closer and if there is then that is set as the endNode and returns that instead.

`void addOpenNodes( GridNode * node );`

Takes a GridNode object as a parameter and calculates the positions directly around it and checks to see whether or not they are in the closed node list and if they are a space. If they are not a closed node and its value is a space then addOpenNode function is called passing through GridNode to add it to the open nodes list.

`bool isNodeClosed( GridNode * node );`

Takes in a GridNode object then calls the STL find searching through the closed nodes list to see if one matches the GridNode object. The result of that returns a Boolean which is then also returned.

Data structures in the software.

Standard Template Library List

The main type of data structure used in this program is a list which is taken from the Standard Template Library.

It is used to store the location and data of open, closed and route nodes that are used to calculate the potential paths in the autorouter.

```
list<GridNode *> openList;
```

This creates a list of open nodes that specifies which nodes in the grid can be used to create a path between the start and end nodes.

```
list<GridNode *> closedList;
```

This creates a list of closed nodes that specifies which nodes are not allowed to be used in a path between the start and end nodes.

```
list<GridNode *> amountOfNode
```

This creates a list of start and end nodes that are used by the AutoRouter to recursively find all possible paths between these nodes.

Arrays

The second type of data structure used within this program is arrays.

An array is used to create the buffer for the file input.

An array of grid nodes is also created in order to keep the location of every node within the grid.

Requirement Acceptance Tests

Software Requirement No	**Test**	**Implemented (Full /Partial/ None)**	**Test Results (Pass/ Fail)**	**Comments (for partial implementation or failed test results)**
1	Accept text file names as arguments from the command line	Yes	Pass	
2	Find start and end nodes in the file	Yes	Pass	
3	Uses recursion to check for the shortest possible route between nodes	Yes	Pass	
4	Check different starting points for solving the grid	Yes	Pass	
5	Prints out the grid with the solved paths, the number of connected nodes and the number of routes	Yes	Pass	

User Instructions

Open the file Pathfinder.exe
Enter text file name as a command line argument