**Assignment 2 – Pathfinding and Alpha-Beta Pruning**

**CISC 352**

**Due March 16, 2018**

**Spencer Edwards - 10129952**

**Hannah LeBlanc - 10091837**

**Pathfinding**

Input and output file names are passed along with the variable mode to the solveGrids function as arguments. Mode denotes which moves are legal for the search. When in mode A, moving up, down, left and right are legal moves. In mode B, the legal moves are the same as mode A but diagonals are also allowed. The solved graphs are written to their respective pathfinding\_a\_out.txt and pathfinding\_b\_out.txt files.

SolveGrids reads in graphs from the input files one by one. Each graph is parsed into a 2D array and is used to create a GridMap object. While we are reading in the graphs from the file we look for the start and goal nodes. When a GridMap object is instantiated, Greedy and A\* searches are used to find the path from the start node to the goal node.

The greedySearch function uses the algorithm discussed in the assignment description. A priority queue is made using a python list called frontier and the Python heapq package. The algorithm starts by pushing the start node onto the queue along with its coordinates. The start node is also added to the cameFrom dictionary with a value of None. While there are still nodes to search in the frontier priority queue, a node is popped from the queue, if the coordinates of this node are the same as the goal, the algorithm is finished searching. If the popped node is not the goal node, the neighbors function is called and the legal moves for the search are returned as a list of tuples. The neighbors are added to the frontier along with their heuristic value. Once the goal is reached, the cameFrom dictionary is used to reconstruct the path from the start node to the goal node.

The aStarSearch differs from greedySearch because each edge travelled has a cost of 1. The cost, along with the heuristic function is used to search the graph. The start node is added to the frontier priority queue and the cameFrom dictionary. aStarSearch also uses a dictionary called csf that keeps track of the cost so far for each node. While the frontier still contains nodes, nodes are popped and the neighbors of the nodes are determined using the neighbors function just like greedySearch does. For each neighbor, the cost and heuristic value are summed and added to the frontier. Eventually, when the goal is found, the greedyReconstruct function determines the path followed by the search.

We chose to use the Euclidean distance between the current node and the goal node for our heuristic function. Other heuristic functions included in our code are the manhattan distance and chebyshev distances. We included these in our code but we do not use them. Once a solution has been found, its path is added to a graph using the editGraph function which adds P’s to the graph where the search travelled in its solution. On larger graphs we would avoid using copy.deepcopy to create temporary graphs to avoid aliasing but it is sufficient for our maximum graph size. The solution graphs are written to files in solveGrids.

**Alpha-Beta Pruning**

The program as a whole is controlled by the solvePrunes function. This function reads the input file “alphabeta.txt” containing the specifically formatted sets of nodes and edges, creates a Graph class for each graph it reads in (a graph consisting of one set of nodes and one set of edges), and then write the output requested by Professor Kaos to the file “alphabeta\_out.txt”.

The Graph class is designed to setup a single graph/tree that it then traverses using the MiniMax algorithm in combination with alpha-beta pruning. This class really consists of two main parts; the parsing of the nodes and edges into their respective data structures, and the traversal of the graph.

The parseNodes function takes the string representing the set of nodes as read from the input file, and parses the string into pieces to ultimately form a Python dictionary (which are implemented more or less as hash tables). In this case, the key for a dictionary entry is the letter of the node (ie. “A” or “Abc” depending on the node), and the value for the entry is either “MIN” or “MAX”, depending on if that particular node is at a layer of the tree that is a max layer or a min layer.

The parseEdges function works very similarly to the parseNodes function. The parsing works in a very similar manner, but the dictionary is set up differently. The key for a dictionary entry is once again the letter of the node, but this time the value is actually a list of other nodes to which the key node is connected to via edges. The list of edges of a key node contains just the children of that node (in other words, the parent of the key node is not in the list).

The alphaBeta function pretty much follows the algorithm given by Professor Kaos in both the course notes and the assignment description. It is obviously tailored to the way it was implemented in our code (ie. our choice of data structures, etc), and really just differs in manner by which we must examine leaf nodes. We do not actually store the leaves as nodes, as they are instead just stored in the dictionary of edges for a “true” or “letter” node. If a node has leaves, the leaves are iterated through via a loop instead of recursively calling the alphaBeta function on them as is done for true nodes. While efficiency was not a primary concern for this assignment given that it is for the most part implemented recursively, eliminating the need to call the alphaBeta function on the leaf nodes as well as the true nodes should speed things up a bit. Other than this, the alphaBeta function follows the algorithm described in the course notes and assignment description.