Pathfinding: Technical Document
Group 4
CISC 352
March 13, 2020

Andrea Perera-Ortega (20008318)
Cameron Duffy (20024164)
Josh Ehrlich (10191667)
Leo Toueg (20062982)
Tyler Mainguy (20023142)

Introduction

This assignment has two main parts. The first is to implement two efficient pathfinding algorithms to find a path within a given input grid, and the second part is to implement an alpha-beta pruning algorithm.

The essence of pathfinding algorithms is to find the shortest route between two points. These algorithms are incredibly useful in the context of artificial intelligence as they are used to determine routes for agents to follow. This field of research is heavily influenced on Dijkstra's algorithm for finding a shortest path on a weighted graph. While this algorithm is optimal, it is inefficient compared to other algorithms. Two other more efficient search algorithms will be implemented and examined in the first part of this assignment. These algorithms are Greedy Search, which is generally efficient but not optimal, and A* which is both optimal and efficient. Greedy Search follows the cheapest heuristic cost while A* examines both the path cost and the heuristic cost.

The goal of search algorithm alpha-beta pruning is to decrease the number of nodes evaluated by the minimax algorithm in its search tree. A minimax algorithm works by recursively choosing the next move in an n-player game. This is a useful algorithm, especially in the context of two-person zero-sum games, because it allows player A to guarantee themselves a payoff regardless of player B's strategy. Player B also can guarantee themselves a payoff. Since it's a zero-sum game, each player minimizes their own maximum loss. For a game like chess, there are an extremely large number of potential moves which significantly increases the execution time of minimax. Fortunately, alpha-beta can make this search much more efficient by not evaluating moves when at least one possibility has been found that proves that the current move will be worse than a previously examined move. Alpha-beta also returns the same move that minimax would have, but also removes (prunes) branches that would never affect the final decision.

Objectives

Pathfinding

In this assignment, we will look to implement two more efficient pathfinding algorithms, Greedy Search and A*. We will be taking m x n (8<=m<=1024 and 8<=n<=1024) input grids that contain a start position ('S') for an agent, a goal position ('G'), open areas represented by '_', and barriers represented by 'X'.

The input of the algorithm will consist of several m x n grids. The algorithms will locate a path between the start and goal positions on the grid and denote it by inserting 'P' to represent each move and output the grids into a text file. The first task will be to implement Greedy and A* with the agent only moving up, down, left or right and output to "pathfinding_a_out.txt". The second task will be to repeat this, but now the agent can move diagonally in addition to its original movements and output to "pathfinding_b_out.txt". Both tasks will use the most informed distance heuristic.

Alpha-beta

The objective of alpha-beta is to optimize the search function of a tree. This is extremely important to many areas of research, business, and general software development. The overarching premise of alpha-beta pruning is to reduce the total number of nodes that are required to be analyzed, i.e. to prune the tree and find the goal node more rapidly. This is done by eliminating branches that are deemed to be redundant by searching the tree and maintaining variables that allow us to understand optimal values over time. Therefore, once we know the predicted value of a subsection, we can determine if it will be selected for and thus, eliminate further subsections so as to expedite our search.

This results in an extremely effective and efficient algorithm for tree searching. We understand that a large part of our future in computer science is building efficient search algorithms which is something that we believe to be of vital importance in our future.

Algorithms

Pathfinding

The code written to solve the path finding problem is similar for both the Greedy and A* approaches, but diverge slightly within different functions. For the sake of brevity, we will discuss the code as a whole as it applies to both algorithms similarly, but point out where it diverges to differentiate between both approaches.

Reading and processing of files (function: read_file())

Before attempting to solve the mazes, our algorithm ensures there are no existing output files in the directory so that they can be properly appended to. Assuming that there are no output files, or they've been removed by this function, the two output files that contain the input mazes are read into the program. The maze grids are represented using a list of lists. After those have been read in by a function, the mazes are passed into a function titled "maze_solve_iter()" which iteratively solves each maze from an input file. For each maze in the list of mazes in the first input file, the maze is solved twice, once with Greedy and once with A* (without diagonal capabilities). The same is repeated with the second input file, however this time there will be diagonal capabilities permitted.

Solving maze (functions: solve_maze(), get_neighbours(), show_path())

The mazes are passed into a function titled "solve_maze" which takes three parameters (the maze being solved, a flag indicating whether the solution will be found using greedy [True] or A* [False]), and a flag indicating whether diagonal moves are allowed). The maze is then searched to find the start and goal x and y positions in the maze array and those positions are saved.

A priority queue titled "frontier" is initialised to keep track of the best selection options. Additionally two dictionaries are initialised, one titled "came_from" and another titled "cost_so_far". The latter will only be used by the A* algorithm. The start node is added to the priority queue and we begin iterating until we've looked at all neighbours, or we've found a solution.

The current best option is taken from the priority queue and we check to see if it's the goal space. If it is, we break from the function and stop looking. If not, we get the legal neighbours from the current location using a function titled get_neighbours().

The get_neighbours() function takes in 5 parameters. The parameters are the current x and y coordinates, the maximum x and y coordinates of the maze to ensure we don't illegally index, and a flag (boolean) to specify whether diagonal neighbours are looked for. The get_neighbours function looks at the left neighbours if we aren't at the left wall and only appends the neighbour to a list of neighbours if it isn't a barrier of the maze. If the diagonal flag is true, meaning we can

traverse through the maze diagonally as well as up, left, right and down, then the function also looks at the top-left and bottom-left neighbours. The same process is then repeated with the right neighbours, including checking the bottom and top-right neighbours if the diagonal flag is true. The list of neighbours is returned back to the solve_maze function.

Back in the solve_maze function, each unseen neighbour is added to the priority queue. This is where the code starts to differentiate.

> For A*: if the greedy flag is false, then we find the cost of moving to the current node by adding 1 to the cost so far. If we haven't seen the node yet, or the cost to reach the node is now cheaper, then we update the cost so far dictionary with the new cost. The node is then put into the priority queue with the updated cost and the heuristic() function is used which returns the euclidean distance between the current location and the goal node.
>
> For Greedy: If the current spot hasn't been examined already, then the cost is calculated using the same heuristic function and put into the priority queue.

If a solution is found, then a function titled "show_path()" is called to update the maze with the path from the start to the goal by placing a "P" down to represent each path step.

Output maze (functions: output_maze(), write_maze())

The mazes are then written to an output file. The non-diagonal Greedy and A* mazes are written to "pathfinding-a-out.txt". The diagonal Greedy and A* mazes are written to "pathfinding-b-out.txt".

Alpha-beta

Our algorithm is broken down into four main sections: reading and processing the file, building a tree from the file, pruning the tree, and then writing the score and number of leaf nodes examined to an output file.

Reading and Processing the File (functions: readFile() and parse(input) output(graphs))

First, let's understand how we read and process the file. readFile() takes the name of an input file ("alphabeta.txt") as a parameter. It opens this file, and saves it as a list of strings, with each string representing an individual line of text from the file. This is done using the built in .readlines() method. Next, readfile() iterates through this list of lines, removing the newline character from the end of each line and separating each string into a list of nodes and a list of edges. Now, instead of a list of strings, there is a two-dimensional array. It is a list where each element consists of a list of nodes and a list of edges for a particular graph. This 2D array is returned by readFile().

Next, the output() function is called, which opens an output file titled "alphabeta_out.txt". This file takes the 2D array returned by readFile() as a parameter. It iterates through each element of the outer list, calling the parse function and passing the current element of the outer list as a parameter. The parse function separates its string parameter based on brackets and the space separating the set of nodes from the set of edges. This function returns a list of nodes and a list of edges. The output function saves the lists returned by the parse function to two seperate variables, one for each list.

Building a Tree (functions: buildTree())

The output file uses the set of nodes and set of edges to instantiate a tree object. When a tree is created, it has a root attribute, which is set to None. It also has attributes for its nodes and edges, which are set to the lists passed in to the Tree object upon instantiation. The tree also has a tree attribute, which is initialized as an empty list, and a count attribute, which is initialized as 0 and is used to count the number of the tree's leaf nodes visited during alpha-beta pruning.

The output file then calls the buildTree method on the tree it just instantiated. This method takes no parameters. It begins by creating an empty list saved to the variable node_list. Then, it loops through the values in the tree's self.nodes attribute. This attribute contains a list of lists, where each element of the outer list represents a node, and the two elements of each inner list represent the value of the node and whether that node is a min node or a max node. This is relevant for the minimax algorithm. On each iteration of this loop, a Node object is created, with the node's value and minmax attributes being set to the data in the inner list of the iterator. Each node is appended to the node_list list after it is instantiated. Once this loop is finished, node_list is

populated with a node object for each parent node in the graph. No leaf nodes are included in node_list.

Next, the builtTree method executes a nested for-loop. The outer loop iterates through each node object in node_list. The inner loop iterates through each element in the tree's self.edges attribute. This attribute contains a list of lists, where each element in the outer list represents an edge, and the two elements in the inner list represent the nodes that the edge connects. If the second element of this list is an integer, then this edge represents a connection to a leaf node, and the integer is the leaf node's value. For each iteration of the nested loop, the value attribute is compared to the value of the first element of the edge being examined. If these two values are equal, it means that this edge is connected to the current node, and that the second value of this edge element is the child node of the current node.

If the child node is not a leaf node, node_list is looped through again to find the child node. This child node object is appended to the list of the current node's children, stored in the current node's children attribute. If the child node is a leaf node, a node object is instantiated for this leaf node, with the leaf node's integer value being stored as its value attribute. Its minmax attribute is left as None, and its children attribute remains an empty list. This node is then appended to the children attribute of its parent node.

Once the nested for-loop is finished, the children attribute of each node in node_list have been populated with a list of its child nodes. node_list now represents a complete tree, and is assigned to the tree's tree attribute. The first node in this list is assigned to the tree's root attribute. This is based on the assumption that the first input set in each line of the input file will begin with the root node. This assumption is given in the assignment description.

By storing each node's children as a list of node objects, we are able to constantly expand the amount of children that each node has. This maintains the possibility of a random branching factor of the input graphs.

Alpha-Beta Pruning (functions: alpha_beta())

After the buildTree method has been called on the tree object, the alpha_beta function is called on the tree. This function takes a current node, an alpha value, and a beta value as parameters. We pass in the root node of the tree, using its self.root attribute, as the current_node parameter. Because this function is recursive, when we call it we pass in None as the alpha and beta parameters.

The alpha_beta function first looks to see if the current node passed in as a parameter is equal to the tree's root node. If it is, it sets the value of alpha to negativity infinity and the value of beta to positive infinity using the infinity value in the math module.

Then, alpha_beta() checks if the current node is a leaf node by examining the length of the list stored in the node's children attribute. This is the base case of this recursive function. If the current node is a leaf node, the tree's count attribute is incremented by one to track that another leaf node has been visited. Also, the value of this leaf node is then returned.

If the current node is not a leaf node, the value of its minmax attribute is examined to check if this is a max node or a min node.

If it is a max node, a for-loop is used to iterate through each of its child nodes. For each child node, the built-in max function is called on the current alpha value and a recursive function call of the alpha_beta function. The child node and the current alpha and beta values are passed into the recursive function call as its parameters. The value returned from the max function is saved to the alpha variable. If alpha is ever greater than or equal to beta, alpha is returned. Also, once the for-loop is finished, alpha is returned.

If the current node is a min node, a for-loop is also used to iterate through each of its child nodes. For each child node, the built-in min function is called on the current beta value and a recursive function call of alpha_beta. The child node and the current alpha and beta values are passed into the recursive function call as its parameters. The value from the min function is saved to the beta variable. If beta is ever less than or equal to alpha then beta is returned. This prunes the tree and eliminates nodes based off of the Min/Max and alpha/beta values. Beta is also returned once the for-loop finishes.

Once the alpha_beta function is finished recursively calling itself, it returns the value at the root of the tree determined by the minimax algorithm. The alpha beta function will have also updated the count attribute of the tree object, making it equal to the number of leaf nodes visited during alpha beta pruning.

Outputting to a text file (functions: output(graphs))

In each iteration of the for loop iterating through each graph in the input file, a tree is built and its alpha_beta function is called. The value at the root node obtained by the minimax algorithm using alpha beta pruning is saved to a variable named score. For each tree, its score and the number of leaf nodes visited are output to the output text file that was already opened at the start of the program.