

Assignment 2

From pathfinding to Pruning

CISC 352 | 16/03/18

Avry Harris (10189321)

Emily Bao (10103388)

Eric Lee (10106823)

Soheil Hamidi (10162318)

Siyu Teng (10122592)

# Pathfinding - Greedy

## Variables USED, and Functions Explained

|  |  |
| --- | --- |
| Local Variables in main | Purpose |
| inputFile and inputFile\_b | These reference the input files provided, pathfinding\_a.txt and pathfinding\_b.txt |
| inputList and inputList\_b | These represent the first stage of turning the text file input into a matrix we can work with. Each new line becomes its own element |
| inputLists and inputLists\_b | These are the lists of the matrices that the program will work with, find a path on and print to the output files. |
| tmpList | A temporary 1-dimensional array that is used to help create the inputLists |
| outputFile | This references both output files that will be used for this assignment, pathfinding\_a\_out.txt and pathfinding\_b\_out.txt. |

#### initializeBoard

Input: An individual m\*n matrix

What it does: This function accepts a matrix of barriers and open spaces as well as goal and start positions. It searches through the matrix, saving the starting position and goal coordinates as a tuple for future reference. It then calculates what, if any, direction that the path needs to move in to reach the goal state, ignoring barriers. “rightMove” and “downMove” variables are used to indicate the next possible movement as either -1, 1, or 0 depending on the determined moving direction as a string of 8 possible directions and NULL. It then returns the five values it has found to wherever it was called.

#### checkPosition

Input: A matrix (inputList), row, column, visitedNodes, pathStack = []

What it does: This function, as one might assume, checks the position of the coordinates given on the matrix that is provided. It makes sure that it is a) within the bounds of the matrix, b) the position is not a barrier ‘X’ and c) that the path has not already Crossed this node. Once it has matched the condition, the current square will be inserted into the beginning of the path stack and returns True. It will otherwise return False if the condition hasn’t been met.

#### checkDestination

Input: A matrix (inputList), row, column

What it does: This function is very similar to the one above but is called slightly before checkPosition. It is used to determine if, given some coordinates (row, column) and a matrix, the goal has been reached and is inbound. It returns either True or False and is called to check if the goal is within one movement of the current position.

#### greedyDiagonal

Inputs: the output of initializeBoard, inputList ( a matrix), the start position, the goal position, the values for rightMove and downMove as well as the string movingDirection

What it does: This function, starting with the start position, generates a stack for the next position to move, uses that position to check if the goal state is in sight and if not moves to the next best position Greedy algorithm style. It checks the 8 possible tiles around it and calls checkPosition to add it to the stack if it is valid. It executes in a while loop, adding ‘P’ to every position it has been to, until there are no more places for it to go or it reaches the goal. It then returns the inputList with the path leading towards the Goal.

#### greedy

Inputs: The same as greedyDiagonal

What it does: It functions the same way as greedyDiagonal, with the exception that it can only move horizontally or vertically, not diagonally. If it needs to go in the direction UP\_RIGHT for example, it will try the horizontal move, then the vertical move, adding two ‘P’s instead of one.

## Algorithm Summary

This algorithm does it’s best to find the best route at every step of the way. It doesn’t use heuristics to look ahead, it just finds the best path it can see right now, takes it and hopes it works. This algorithm is, for the most part, efficient, but not optimal.

One decision in the implementation that was used to optimize this algorithm was that a dictionary was used to store all of the nodes visited nodes so that they could be accessed in O(1) time. We also made use of stacks in storing potential paths if the current one fails, so our algorithm would retreat instead of being completely blocked.

# Pathfinding – A\*

## Variables USED, and Functions Explained

|  |  |
| --- | --- |
| Local Variables in main | Purpose |
| inputFile | This reference the input files provided. |
| inputList | These represent the first stage of turning the text file input into a matrix we can work with. Each new line becomes its own element. |
| sPosition and gPosition | These are the indices of the start and goal positions respectively. |
| frontier | This is a heapDict for storing positions for the algorithm to explore as a priority queue. |
| outputFile | This references the output file that will be used for this assignment. |

#### get\_adjacent\_cells

Input: Current Cell

What it does: This function Returns adjacent cells to a cell clockwise starting from the cell on the right.

#### get\_adjacent\_cell\_diagonal

Input: Current Cell

What it does: The same as *get\_adjacent\_cells* but for the diagonals.

#### findpath\_a

Input: None

What it does: This function uses the A\* pathfinding method, using the Manhattan distance heuristic function to find the best path it can only moving vertically or horizontally, not diagonally.

#### findpath\_b

Input: None

What it does: The same as findpath\_a but this function is able to check and move diagonally as well as horizontally and vertically.

#### draw\_path

Input: came\_from, a dictionary that stores the path that the A\* algorithm has found.

What it does: This function uses the came\_from input given to it to alter the input list, inserting the completed path found.

#### heuristic

Input: Two positions, a and b

What it does: This function finds the Manhattan distance between the two points given.

## Algorithm Summary

This algorithm is similar to the greedy algorithm above, in that it looks for the best option and takes it. They differ in the fact that A\* considers heuristics, An estimated cost from the current state to the end goal. This addition to the typical greedy algorithm results in an algorithm that is efficient, just like the Greedy algorithm, but also optimal.

# Alpha-Beta Pruning

## Variables USED, and Functions Explained

|  |  |
| --- | --- |
| Global variables | Purpose |
| traces\_enabled | This global variable instructs the program to either print off statements related to tracing through the program, helping find errors, or to not print them if it is set to false |
| leaf\_nodes\_examined | Keeps track of how many leaf nodes that the alpha-beta algorithm has examined |

#### read\_file

Input: None

What it does: This function reads the alphabeta text file into the program, splitting up the text input for each graph into lists of tuples that the program can work with. It creates two lists, one for edges and one for if the current node is a min or max node. It then calls the write\_file function with these two lists as input.

#### write\_file

Input: The list of edges and of min/max produced in read\_file

What it does: This program opens the output file for writing, then, for each graph, starts the recursive alpha\_beta algorithm and prints the result once it’s been found in the output file.

#### alpha\_beta

Input: the current node, the minmax list for each node, the list of edges, and values for alpha and beta.

What it does: Using the internal functions below this function applies the minmax algorithm with alpha beta pruning to find the score while examining as few nodes as possible.

#### is\_leaf

Input: a node

What it does: It checks if the value of the node is an integer and not a letter. If it is an integer then the program returns True.

#### is\_max\_node

Input: a node

What it does: using the getMaxNodes function, this function generates a list of all of the ‘MAX’ nodes, this is so a node can be identified as a min or max node depending on if it is found in the list max\_nodes.

#### getMaxNodes

Input: none

What it does: goes through the minmax list and appends any ‘MAX’ nodes to the list max\_nodes

#### getChildren

Input: none

What it does: generates a list of children for the current node.

## Algorithm Summary

Alpha-beta pruning, in this assignment, is used to look at an undisclosed number of trees with an undisclosed number of nodes. This is used during a two-player min-max game, where one player is trying to select the lowest possible score while the other is trying to grab the highest. This algorithm functions recursively, recusing through the tree figuring out the moves that it’s opponent would make. It can make the decisions with perfect accuracy and so this eliminates the need for the algorithm to look at large parts of the tree “pruning” off branches.

In this algorithm one way in which we sought to optimize was by using a list to store edges instead of a set because sets are not subscriptable. We also have a variable trace\_enabled which globally triggers a large amount of print statements that greatly aid when debugging and tracing the code.