COS30019 Assignment 1- Tree Based Search Report

Alex Nikolov

100 595 852

Option B: Robot Navigation

Instructions:

My program has a simple TUI interface, with commands being input in the following format:

`Main <filename> <search-method>`

`<filename>` specifies the text file specifying the problem that is to be solved, and <search-method> is the search method to be used. The options for `<search-method>` are: BFS, DFS, IDS GBFS, A*, and WA*.

Once run, the program uses the selected search method to find a solution, and prints said solution in the terminal, along with how many nodes were expanded in the process, and what file it was run on.

Introduction:

For this assignment, I chose option B, in which the objective is to implement a program where a ‘robot’ navigates a 2-dimensional grid maze, from a given starting coordinate to one of the given goal coordinates (preferably the one that is closer to reach). The size of the grid is variable, and given by a file that the program has to ingest. There are also “walls” that are specified in the problem file, which the robot may not enter and must navigate around.

The objective of this assignment is to investigate and implement different search algorithms that may be used to solve either a navigation problem such as this, or more generally, any search problem that can be represented as a graph of nodes, wherein each node is connected to at least one other node by a path that must be crossed in order to travel between them.

Uninformed Search:

The simplest search algorithms use uninformed search, and select the next node to search through using a set of generic rules depending on the algorithm that doesn’t take into account the state of the environment around it.

The issue with uninformed search is that as the search algorithm essentially goes around blindly searching without knowledge of where it’s goals lie, the time complexity of the search can be exponential (at least for algorithms that find an optimal solution). For small search problems, this is not an issue, however as the size of a search space grows, uninformed search algorithms can quickly become too slow to be useful.

Informed Search:

As the size of search problems grows, so does the time required for a search algorithm to find a solution, in the case of large search spaces (such as a 2 dimensional grid, where the search space is proportional in size to the x and y dimensions of the grid multiplied with each other), we need some sort of way to point a search algorithm in the right direction rather than wait for it to blindly stumble across a correct solution.

This is where informed, or heuristic search becomes useful. A heuristic function essentially puts a number on how far from a destination a potential search node is, which the search algorithm uses in

order to inform its decision on which node to expand next. In the case of the maze program, the most appropriate heuristic function is the Manhattan grid distance between the current location of the agent, and the nearest goal node.

Search Algorithms

Breadth-First Search (BFS)

Possibly the simplest search algorithm, BFS takes the first node, and adds all the possible nodes from which the agent could move in a FIFO (First-in First-out) buffer for expansion. As each node is popped off the FIFO, first it is compared to the destination nodes to check whether the search is complete, and if not, it's children nodes are added to the FIFO buffer for later expansion. The BFS algorithm essentially expands nodes in a concentric manner around the starting node until it either finds a solution, or runs out of space to search.

In order to make it more efficient, I have also added a list of nodes that have already been searched so it doesn't go back where it has already gone.

The primary advantages of BFS are that it is simple to implement, and it will find an optimal solution (the shortest path between the starting node and a solution node, with no movements in the wrong direction).

The main issue with BFS is that both it's space and time complexity is exponential, so as a problem grows in size, the amount of time and memory required to find a solution grows exponentially, quickly making it too slow to be useful in the real world.

Depth-First Search (DFS)

At its core, DFS is a variation on BFS, except instead of expanding nodes in the order in which they were added to the frontier, it expands the latest node that is added to the list first. All that is required to convert a BFS algorithm to a DFS one is changing the FIFO frontier buffer into a LIFO (Last-In Last-Out) buffer.

An issue with DFS is that if there is no method to check for duplicate nodes, it is likely to end up in an infinite loop, searching through the same nodes until it is killed.

DFS, unlike BFS is not an optimal algorithm, and will not always find the shortest path between the starting node and goal node.

The main advantage of DFS over BFS is that it is usually far faster, and less memory intensive when finding a solution.

Iterative-Deepening Search (IDS)

An attempt to combine the advantages of both BFS and DFS, IDS is a depth-first search, where it's maximum depth is limited, and iteratively deepened until a solution is found. Unlike DFS, it doesn't require a list of nodes that it has already visited, as the depth limit kills if it ends up in an endless loop. It also finds optimal solutions, unlike DFS. IDS is better than DFS because it's memory complexity scales linearly rather than exponentially.

I selected IDS because it is optimal, and negates the main weaknesses of BFS. Uniform-cost search would be equivalent to BFS, and would therefore be inapplicable in this case, leaving only IDS and Bidirectional Search as viable options. I selected IDS over Bi-directional primarily because of its simplicity of implementation, and also because the variable number of goal states would have made bidirectional search more like tri, or quad-directional search.

Greedy Best First Search (GBFS)

Greedy best first search is the simplest informed search algorithm for this problem. In order to implement it, the first step is to have a heuristic function, that measures the Manhattan-grid distance (as this is the minimum number of moves to get from a give node to the destination node) between each node in the frontier and it's closest goal node. GBFS's evaluation function for selecting which node to expand next is look in its frontier, and select the node with the smallest heuristic value.

GBFS is relatively simple to implement and usually quite fast, however it is not an optimal search algorithm, as it does not care about how deep the current search tree is, and can easily be fooled by a wall into taking a sub-optimal path.

A* Search

A* search is similar to GBFS, except it's evaluation function is f(n) = g(n) + h (n), meaning that it adds up the heuristic function, and cost of the path it took to get to the node being evaluated before selecting the next node for expansion.

Unlike GBFS, A* is an optimal search algorithm, as it will always choose the node with the lowest distance from the starting point plus distance from a goal node to expand next.

It's thoroughness is also a weakness however, as it will open all potential nodes. As an example, when the agent is moving diagonally through an empty space, instead of expanding nodes by order in a roughly diagonal line pointing in the direction of the nearest goal, as the evaluation function in the open space is the same, it will open them all in a manner akin to a wave-front propagating through a gap in a wall to fill the entire cavity, and find the best opening at the other end of the open space.

Weighted A* Search (WA*)

In order to address the issues discussed previously with GBFS, and A*, a possible solution is to arrive at WA* search. WA*'s evaluation function is f(n) = g(n) +W* h (n), with W being any value over 1. This evaluation function means that nodes closer to the destination are favoured over nodes further away from it.

As the heuristic function returns a value higher that is potentially higher than the actual number of steps it would take to get to the destination node, it is no longer an admissible heuristic, and WA* is not an optimal search algorithm, and will not necessarily find the shortest path to the goal unlike regular A*.

WA* is an attempt at a compromise between GBFS' fast, if at times very sub-optimal approach, and A*'s slower, but more optimal approach that doesn't get tricked by walls getting in its way.

In the case of the small 5*11 test problem given, there is no real point of using WA* over regular A*, as the solution in both cases is found in a fraction of a second, however it can make a large difference when evaluating much larger problems.

Implementation:

The core structure of my program was based on the Npuzzler example given.

Enumeration List:

- mazeCellState
- direction

Class List:

- Main
- SearchMethod
  - AStar
  - BreadthFirstSearch
  - DepthFirstSearch
  - GreedyBestFirstSearch
  - IterativeDeepeningSearch
  - WeightedAStar
- MazeState
- Solution

Enumeration Descriptions

mazeCellState:

Used to store the state of the maze state, there are three possible options; empty, wall, and agent (agent was only used for debugging purposes).

direction:

Options are up, down, left, and right. Used to communicate move options and store the moves made.

Class Descriptions

Main:

The main class is used to ingest the input file, initialise the starting and goal states, initialise the search methods, feed the start and goal states to the search method specified by the user, and print the results back to the terminal if a solution is found.

SearchMethod:

Search Method is an abstract class that serves as a generic foundation that all the other search methods extend. It contains the following data elements:

- name (String): used to find the specific search method.
- SearchedNodes (ArrayList<MazeState>): Used to store a list of nodes that have already been searched to prevent expanding nodes unnecessarily.
- FrontierNodes (LinkedList<MazeState>): Used to store a list of nodes in the frontier. A linked list is used here rather than an array list because it is possible to add and remove arbitrary elements from it as needed.
- Goal (ArrayList<MazeState>): Used to store a list of goal nodes.
- CurrentNode (MazeState): The current node being evaluated and expanded.

Methods:

- abstract Solution Solve(MazeState startingMaze, ArrayList<MazeState> goalMazes)
  - Every search algorithm overrides this method with their own search implementation.
  - Return a solution object.
- boolean isSolved(MazeState currentMaze)
  - Evaluates whether the node currently being evaluated is in the list of goal nodes.
- void AddToFrontier(ArrayList<MazeState> additions)
  - Adds Mazestates from an array list to the frontier, after making sure that they are not in the list of already searched nodes or already in the frontier.
- MazeState popLastFromFrontier()
  - Used by DFS and IDS to use the frontier linked list as a LIFO queue.
- MazeState popSmallestFCostFromFrontier()
  - Used by all informed search algorithms, this method finds the Mazestate in the frontier with the smallest Fcost, and pops it out of the frontier for evaluation.

AStar

While the frontier isn't empty, it keeps expanding the node in the frontier with the lowest FCost, calculating their children's heuristic, and adding it to the distance they have already travelled before adding them to the frontier.

BreadthFirstSearch

Pops nodes from the frontier and expands them in the same order as they are were added to the frontier.

DepthFirstSearch

Pops node from the frontier and expands them in order of newest to oldest, expanding a node-tree until it either reaches its end, or finds a solution before moving on to a different branch in the search tree.

GreedyBestFirstSearch

Pops nodes from the frontier and expands them in order from lowest to highest heuristic cost. Children of the node being currently expanded have their heuristic costs evaluated and assigned to their Fcost variable before being added to the frontier.

IterativeDeepeningSearch

Almost the same as DFS, except the maximum depth of the search tree is increased from one to the number of squares in the maze one at a time.

WeightedAStar

Similar to A*, except the evaluation function is equal to the current node cost plus the heuristic cost times 1.5. This multiplier is intended to make it favour nodes closer to a solution and waste less search time expanding nodes it doesn't need to.

MazeState

Variables:

- public static mazeCellState[][] Maze
  - Stores the maze problem specified in the problem file, this is used to determine what moves the agent can make.
- public int Gcost
  - Stores the number of moves that have been made to reach this maze state.
- public double Hcost
  - Stores the calculated heuristic cost to the nearest goal state.
- public double Fcost
  - Stores the evaluation function as calculated by the individual search method.
- private int[2] agentLocationXY
  - Stores the current location of the agent.
- public LinkedList<direction> directionList
  - Stores the list of directions that have been taken from the starting node to reach this node.

Methods:

- public void AddWall(int x, int y, int width, int height)
  - Adds a wall based on line in the input file.
- public void AddAgent(int x, int y)
  - Adds an agent at the specified location.
- public boolean CompareMazeStates(MazeState in)
  - Compares whether the input Mazestate has an agent at the same location as this instance.
- public ArrayList<direction> getPossibleMoves()
  - Determines the possible directions agent in this MazeState can move in.
- public MazeState MoveAgent(direction dir)
  - Returns a MazeState with it's agent moved in the direction specified.
- public ArrayList<MazeState> getPossibleMoveStates()
  - Returns an arraylist of MazeStates that that result if the agent in this mazestate moves in every direction where it isn't blocked by a wall.
- public void PrintMaze().
  - Prints a character based impression of the current maze state. This is only used for debugging and development purposes.
- public int GetManhattanDistance(MazeState in)
  - Determines the Manhattan grid distance between the agent in this maze and a specified MazeState.
- public int getShortestManhattanDistance(ArrayList<MazeState> inputNodes)
  - Given an array list of mazestates, the Manhattan grid distance to the closes one is returned.

Solution:

Variables:

- public int numberofNodes;
  - Stores the number of nodes that were expanded to find a solution.
- public LinkedList<direction> directionList;
  - Stored a list of the move directions that it took to reach a solution.

Acknowledgements/Resources:

Russell, SJ (Stuart J & Norvig, P 2010, *Artificial intelligence : a modern approach*, 3rd edition., Prentice Hall, Upper Saddle River, N.J.