Himath Ratnayake

s5209861 | Griffith university 2021

Intelligent Systems

Assignment 1 report – rush hour

**Problem Formulation**

**Initial State:**

The initial state is the state from which the problem-solving process starts. In the case of rush hour, the initial state is the configuration of the board as given by the input file rh.txt, with different cars obstructing the path of car “XX” to the desired exit.

**Actions:**

Given a particular state of the board, there are a variety of actions that can be executed. A single vehicle can move left/right if its orientation is horizontal, or up/down if its orientation is vertical. Thus, an action can be described as the moving of a single vehicle in a direction based on its orientation by a certain number of blocks.

**Transition Model:**

The transition model is defined as a collection of state-action pairs, which with the initial space create a “state space”. It is essentially a description of what each action does. In rush hour, the state-action pairs would be movements of various cars within the board (actions), resulting in new board configurations (states). For example, “move Car *CC* left by 3 blocks”.

* It can be summarized as: Board(Car, Direction, Move)*🡪 New State*

**Goal Test:**

The goal test is used to verify whether the given state is the goal state. In rush hour, the goal state is the board configuration in which there are no longer any obstructions between the car “XX” and the exit on the right side of the board’s third line.

**Path Cost:**

The past cost is equal to the total number of movements within the board required to reach the goal state. If the step cost were to equal the number of squares moved by a vehicle, the proposed solution may not be optimal, as step cost must be equal. For this assignment, a vehicle can move to any valid position it is able to, and regardless of how many squares it moves, the step cost will always be one if a move has taken place. By keeping the step cost equal, algorithms such as breadth-first search are able to be optimal.

**Software Design**

**Functions:**

|  |  |
| --- | --- |
| **process\_inputs** | |
| **Description** | This is an initial function with reads in the rh.txt text file, and then creates the initial states of each board from the string given. |
| **Data Types/Structures** | The main data structures utilized in this function are lists. While iterating through the file contents, it looks for key characteristics of lines which signal that there is important information present there.  **For the board:**  Each current board is stored within a 2-dimensional matrix known as currentboard, and this matrix is then added to another larger list that comprises of all 40 initial game board states. This allows for easy storage and access of each game board. To identify whether there is a board to be read in, the value of the current line’s length is checked to see if it equals 37 (36 letters and a newline character). If this is the case, that value is stored in the “current” array to be printed later and the gameboard making process begins. To make the gameboard, the 36 letters are broken up into to 6 letter substrings comprising each row and column of the 6x6 matrix. Then they are joined together to create the board matrix and finally appended into the larger initial\_states list.  **For the solutions:**  Each solution is also stored in a list of lists, with each index corresponding to a different game board’s proposed solutions. Each line in rh.txt that contains a solution starts with “Sol”, so when this value is encountered, a Boolean is used to mark that line as one with a solution, and that solution is added to a temp array, which is then appended to the proposed solutions array if it’s a single line solution. For multiline solutions, the new line character is replaced with a blank space, and the multiline values are added to a temp array. Then all values in temp for the multiline values are added into the proposed solution array for that index/board number.  The input parameters that have now been processed are now returned, including the initial states of the boards, the board string, and proposed solutions, so that they can be printed. |

|  |  |
| --- | --- |
| **expand** | |
| **Description** | This function is used to create new states from old ones. First, the get\_cars function returns all the unique cars within the current board. Then for each car, its index is found.  Before processing, car orientation is found; if the same car letter occurs for two indexes in a row index[i] == index[i+1], then this means that the car’s orientation is horizontal.  The car is moved to a new location each time, while verifying that the board’s limits are not breached. For horizontal cars, to move by 1 block to the left or right, the index (shift) must only be moved by 1. However, for vertical blocks it must be moved by 6 indices to go up, as this is the difference in how many indices there is between the current row’s piece and the same piece in the row above or below. Due to the format of the state as a list, the current methodology can be prone to making too many horizontal movements if the line above is empty (occupied with only a ‘.’). This is because the board is parsed through as a list. To ensure that horizontal movements stay within the same row, is made where it makes sure that the new location that the car moves to is within the same 6 letter substring, so that the car doesn’t move to a different row, creating an invalid action.  This same process is repeated for blocks going to the right or down, but simply the shift and also the current state is reversed, so that indexing now happens, essentially, “backwards”. This allows for the rest of the possible moves to be found.  Finally, the amount of states that are found is returned as a list of arrays, so that it can be used within algorithms. |

|  |  |
| --- | --- |
| **new\_state** | |
| **Description** | The new\_state function is used to transform the board from an initial state to a new one, based on an action that is passed in from the expand function. Each time this happens, a new\_state array is made and the contents of the board passed in is copied. The new\_state edited based on the action given, and this edited board will be appended to a set of such boards that have already been found (explored\_states).  The new\_state function takes in the current state, index, number of blocks occupied by a car (“count”), and the action.  The action is passed as a 3 letter string; The 1st letter is the car, 2nd letter is the direction and the 3rd letter is the amount the car moved by.  For horizontal cars, instances of the car on the board are replaced with a “.”, from the car’s start index to the index + count. Then, depending on whether it is left or right, the new indexes where the car moves to is calculated. The new start index will be the initial index of the car + orientation \* the amount the car must move by. Orientation will be -1 if the direction is left, so that the car moves in that direction on the board. The new end index of the car will be new start index + “count”.  For vertically oriented cars, to remove instances of the car, the initial index is saved in a temp variable. Then, the cars are replaced by an empty slot (“.”) , and then a for loop in the range “count” is iterated through, with +6 being added to the index each time to signify a new row. This also only happens when the index < 37, so that indexing does not exceed the bounds of the board.  As the index is being changed each time, the index is redeclared as the original value by setting the index = temp.  Then, to add the car to the new location, a new place is calculated in a for loop ranging from 1 to count+1.  The new place index is equal to the index + orientation \* 6i, where “i” will increase with each for loop by 6.  If the car is meant to move down, the new\_place index must be less than 37, and if the car is meant to move up, the new\_place index must be greater than 0; this will ensure any indexing errors do not occur.  Finally, the new\_state array is returned so that it can be appended to the found\_states list of arrays in the expand function. |

|  |  |
| --- | --- |
| **get\_cars** | |
| **Description** | The get\_cars function is used to scan the current board state and find the locations and names of all cars present. It is a subfunction that is called upon by the expand function.  If a letter is found and its not yet in cars, the new letter is stored within a “cars” dictionary along with its index. This in Letter:Start Index key-value pairs.  This dictionary is then returned to be used by the expand function. |

|  |  |
| --- | --- |
| **Is\_solved** | |
| **Description** | The Is\_Solved function is called by all algorithms to detect whether the current goal state passed in to the is\_solved function is equal to a goal state.  The goal car will always be present in the 3rd row of the board – this is equivalent to the 12th to 18th index of the board. This index range of the current state is turned into a substring. The goal car is always denoted by the letter ‘X’ too. The substring is traversed until ‘X’ is found, at which point the loop breaks with a variable named ‘end\_x’ used to denote the end index of ‘X’ in the substring. Then, in another loop the rest of the substring is traversed. If the rest of the letters after ‘X’ in that row are empty, then a solution has been found, and the ‘solved’ bool is returned as True. However, if another car’s letter is encountered, ‘solved’ becomes False, and the loop immediately breaks.  This function then returns the final value of ‘solved’ back to the algorithm which called it. |

|  |  |
| --- | --- |
| **output** | |
| **Description** | The output function is responsible for writing the solutions found by each function to a file. It takes in 3 arguments – the current problem, the solution found, time taken, and the algorithm. It then opens an output file known as “rhoutput.txt” in append mode and writes the results and statistics for each problem (the solution number, time taken and solution). |

|  |  |
| --- | --- |
| **heuristic** | |
| **Description** | The heuristic function is used by A\* search (as explained in depth further into the report). It returns a value equal to the number of cars blocking the exit of the goal car (“XX”). Similar to the is\_solved function, it does this by looking at index 12 to 18, which is turned into a substring where car XX is situated. From the end of the car “XX”, it checks the substring to see if there is another car present using the .isalpha function which will return true if a letter (i.e. car) is present at the current index. |

|  |  |
| --- | --- |
| **class PriorityQueue** | |
| **Description** | This class contains the following functions and is used by the A\* algorithm  **Constructor:**  Initializes an index to 0 and creates a queue object  **Empty:**  If the len of the queue is 0, then returns true. Else, it returns False  **Pop:**  Returns the value situated at the end of the queue, then pops it using the heapq.heappop function in the heapq library.  **Push:**  Pushes a priority value, index and “item” which consists of a list containing the current moves and states of the board. It then increments the index by 1. |

|  |  |
| --- | --- |
| **main** | |
| **Description** | The lists for proposed solutions, initial states, and current problem are initialsied and sent to the process\_inputs function. The output.txt file is also created, and then closed to initialize it and remove any data contained within it from previous runs of the program.  A user input is also prompted for, regarding the maximal time.  Then, each problem is iterated through, and the initial board state is printed. Then, algorithms for BFS, Iterative deepening, A Star, and Hill Climbing are run.  For each algorithm, the start time is initialized, and the algorithm is run. Then, the solution is printed out, as well as the time taken if applicable, and relevant data is sent to the output file which is called. If the algorithm fails to operate within the maximal time, a timeout exception is raised, and the function terminates, moving on to the next function or iteration. |

**Bread First Search Algorithm**

Breadth first search is a search algorithm in which a root node (in rush hour’s case, the initial board state read in from rh.txt) is expanded, then all successors of that state are next expanded, and so on for the resulting successors from that. In BFS, the shallowest unexpanded node is what is chosen for expansion, and this is done with a a FIFO (first in first out) queue structure for the frontier, which pops the oldest queue element.

A set is also used, which will store unique states of the board and ensure that the same state is not traversed multiple times.

**Pseudocode:**

Create a queue containing the start state of the board

Create a set containing all the states of the board that have already been found

While the queue length is greater than 0:

Pop FRONT element of queue into “Path” variable. This will store various sets of actions explored so far.

If the current last element of the path variable is the goal state:

If it is, return True

Get the next possible states of the last state stored in the path variable with the expand

For each of the next states, if they are not in seen states set:

Add next state to the seen states set

Add next state to the queue along with the current path so far

Return False if no solution found

**Iterative Deepening Algorithm Pseudocode:**

Iterative deepening also utilises a queue data structure, but while Breadth-first search uses a FIFO queue, depth first uses LIFO (last in first out) so it always expands the deepest node in the current search tree frontier. A set data structure comprised of lists is also still used to keep track of unique board states that have already been visited.

**Iterative\_deepening(start\_state)**

Initialise depth variable as 0

While depth variable is greater than equal to 0:

Call the DLS function, passing in the state and current depth

If the DLS function finds a solution, return True

Otherwise, increment depth

Return False if no solution found

**DLS(start\_state, depth):**

Create a queue containing the start state of the board

Create a set containing all the states of the board that have already been found

While the queue length is greater than 0:

Pop LAST element of queue into “Path” variable. This will store various sets of actions explored so far.

If the current last element of the path variable is the goal state:

If it is, return True

Get the next possible states of the last state stored in the path variable with the expand

Add next state to the queue along with the current path so far

Recursively call DLS again, while decrementing depth

If the queue’s length ends up at 0, then all states were traversed but no solution was found.

Return False if no solution found

**A\* Search Algorithm Heuristics:**

In A\* search, a heuristic is used, which ranks alternatives in expanded nodes to calculate which one is the best to follow based on the cost to get to the goal.

A priority queue class is responsible for designating various priority values to elements within the queue (the elements being various states of the board). This class is utilized in the A\* algorithm to designate various values based on the movement cost + heuristic value.

Furthermore, for A\* Search, the heuristic crafted is optimal when it is both consistent and admissible, meaning it doesn’t overestimate the cost to the goal state.

**Zero Heuristic:**

The value of the heuristic is equal to zero in all states, so since it returns the same value for each node, so the algorithm isn’t improved at all from the Breadth-First Search.

**Heuristic 1:**

Another heuristic could be the distance of the target vehicle to the exit. This heuristic is admissible, as it doesn’t overestimate the final cost to the goal; the path cost to the goal will always be greater than or equal to the steps needed to get the target vehicle to the exit.

**Heuristic 2:**

Finally, the heuristic that was ultimately chosen was counting the number of cars in the way of the target car’s path to the exit. Every car blocking the way of the target car counts as a value of one. If a car is blocking the exit, then it is assumed that at least these cars must be moved in order to reach the goal state. This heuristic is admissible, since it doesn’t overestimate the path cost to reach the goal state; since it counts the vehicles blocking the exit, when one of these cars moves away from the path, the heuristic value can be lowered when calculated again as that vehicle is no longer blocking the way of the target car. This makes it so that the heuristic value is always less than or equal to path cost needed to reach the goal state, making it admissible.

**A\* Pseudocode:**

Initialise Priority Queue, pqueue

Initialise Closed\_Set

Push the initial start state and moves, as well as priority to pqueue

While the pqueue is not empty:

Pop the moves and current state from pqueue into a variable “current”

If they are solved, print the board and return the moves

Else, for the new moves and states found in child nodes of current:

Initialise a movement cost g

Use this to calculate priority for the current state using the equation f = g(u) + h(u)

If any child nodes are not in the closed set, add them to closed set

Push the new moves, current child node and priority level calculated before into pqueue

Otherwise, continue

The key value of the current state (g, aka movement cost) will become the new g value.

**Further Explanation**

* The closed set is the set of nodes that have already been considered but not chosen
* The priority queue pqueue is the set of nodes that the next “current” node is chosen from (also sometimes referred to as the open set)

**Hill-Climbing Algorithm:**

For the Hill Climbing Algorithm, rather than reporting the sequence of moves, it will only report if a solution has been found and return the state of the algorithm.

**Steepest Ascent Greedy Hill Climbing Pseudocode:**

Set initial state as current

While solution hasn’t been found, keep iterating:

Current\_Cost 🡨 Evaluate current state using cost function

Expand the current state and find all state-action pairs possible

For each state-action pair found:

Evaluate the state using cost function

If all evaluations using cost function < Current\_Cost

Return the current state

Else

Set the lowest cost evaluation calculated as new current state

**Random Restart Hill Climbing Pseudocode:**

For a given problem in rh.txt, create a data set of successor states of this problem using expand function

When restarting, use a random state from this data set

**Pseudocode:**

Set initial state as current

While solution hasn’t been found, keep iterating:

Evaluate current state using cost function

Expand the current state and find all state-action pairs possible, as well as their costs

Pick a random state-action pair from the ones with the lowest costs

If evaluation of this state using cost function is less than that of the current state

Return the current state

Else

Set the lowest cost evaluation calculated as new current state, and continue iterating

**Experiment Results**

For breadth first search, it is complete as long as the total number of nodes is finite, and a goal state exists. The time taken is based on the number of nodes generated. The space complexity of breadth first search is much greater than that of depth first search, as more memory is needed to store all nodes that are traversed and generated, which can be a problem as the number of nodes exponentially increases with increasing depth.

When comparing the proposed solutions to the ones outputted by the BFS algorithm, it was found that on all cases the BFS algorithm reported the same number of solutions as the proposed solutions from rh.txt. This makes sense as Breadth First Search is able to find the shortest path due to it exploring all nodes available.

The initial cutoff time that was tested was 30 seconds, and this was found to be sufficient for all problems of Breadth First Search. On average, the beginner problems took 2.29 seconds, intermediate took 2.72 seconds, advanced took 3.31 seconds and expert took 2.83 seconds. This lead to an average time across all difficulties of 2.78 seconds. Notably, Breadth First Search performed better at the expert level as a whole time-wise (2.83 seconds average), which was more difficult, when compared with the advanced level (3.31 seconds average). A reason for this could be attributed to problem 25, which at 9.12 seconds computing time took the 2nd longest time across the problem sets to calculate a solution for, upwardly skewing the advanced level’s average time. The reason for this is due to significantly more nodes needing to be traversed for this particular solution, when compared to other solutions (9612 visited nodes for this problem with BFS).

In contrast, iterative deepening is a form of depth-first search where a depth-limited depth-first search is done with increasing depth limits gradually. The advantage of using iterative deepening is that, though it may get the same optimal answer as breadth-first search, much less memory is consumed. While Breadth-first search uses a FIFO queue, depth first uses LIFO (last in first out) so it always expands the deepest node in the current search tree frontier.

Initially, it was carried out with an implementation that was detailed in the pseudocode above for iterative deepening. However, this implementation proved to be extremely efficient, taking multiple hours to reach a solution for problem. However, the solution it outputted was optimal as expected (the shallowest expanded node), but it proved to be simply unfeasible for the iterative deepening to go through with all 40 problems in a realistic amount of time. The time taken would be exponential, and with repeat states not being accounted for, it meant that the algorithm was simply not efficient on its own without this check for already visited nodes. However in ideal cases it would have expanded and found the goal path at the shallowest node.

The current implementation of iterative deepening is able to find a solution the fastest on average out of all the algorithms, but it fails to find an optimal solution, is most likely due to the addition of a “visited” array much like the one in BFS. The problem is that when this is added, the depth-limited algorithm also skips these seen states after seeing them just once. This makes it so that parts of the tree where this state may reappear are blocked from forming, as the state had already been traversed at an earlier depth and can’t be reused again in the path. One way to get around this problem is by making it so that it skips seen states if the current depth the state was found in is greater or equal to the depth that state was last seen in. Essentially, if the depth was deeper at a new iteration for a particular state than it was at a previous iteration, then it would bypass being added to the visited array, and it could be used to find an optimal solution. Overall, Iterative deepening had the greatest distance from proposed solutions, being significantly less optimal, but it was also always successful at finding a goal state, albeit with a long solution.

In A\* search, the cost to reach the node (g(n)) and the cost to get from the node to the goal (h(n)) is combined. G(n) gives the path cost from the start state to node n, and h(n) estimates the cost of the cheapest path from n to the goal using a heuristic function. Hence, the node with the lowest value of g(n) + h(n) is considered the cheapest solution. A\* search was able to find the optimal solution in most cases using the priority queue and heuristic discussed in the software design section (the explanation for heuristic admissibility was also elaborated upon there). The time taken with A\* was comparable to Breadth-First search for most problems, but it took noticeably longer for problems such as number 14, which had a computing time of 69 seconds. This resulted in A\* search taking the longest on average at the intermediate level (10.42 seconds), but if problem 14 was to be removed, the intermediate level, the average time would be decreased greatly to 3.86 seconds. Similar to Breadth-first search, A\* also performed better on the harder “expert” level problems (4.24 seconds average) compared to the advanced level ones (4.63 seconds on average). However this could also be attributed to a greater variation of times in advanced level, with problem 24 taking 11.63 seconds to compute, which is much longer than any times in the expert level. As a whole, the A\* search algorithm took 7.29 seconds to compute a problem, if problem 14 was included, or 4.67 seconds on average if problem 14 was to be classed as an outlier. In both cases, it would still result in it being slower than Breadth-First search. A maximal time of 45 seconds is ultimately sufficient to get all solutions for BFS, ID, and 38/40 (a vast majority) for A\*. It is also expected that A\* should expand far fewer nodes than BFS but it was found that the amount expanded by A\* was somewhat comparable to that of BFS. This could be attributed to the heuristic not being strong enough, resulting in only very minor improvements over node expansion than BFS.

For the greedy hill climbing algorithm, it searches for an optimal solution by starting from an initial state and analysing the aspects in which it can be improved. A scoring function similar to that of a heuristic is necessary. The heuristic used was the same as that of A\* (cars blocking), but the problem was that this heuristic’s values were too simple, resulting in the algorithm getting “stuck” prematurely due to the heuristic values being the same across a multitude of boards, meaning that it was unable to output a valid solution for problems. To combat this, a stronger heuristic/cost function would have been necessary. An example could be, after finding the vehicles that block the target vehicle to the exit (which could be named as the first tier), to also find a secondary tier of vehicles which block the first tier (cars that block the target car). This secondary tier of cars must also move in order to move the cars that block the target car. It is also thus admissible and consistent as when one vehicle (either a first or second tier) stops blocking, the heuristic will then calculate a lower value compared to the state, so that path cost isn’t overestimated.

With random hill climbing however, some of these problems had more progress. Random hill climbing doesn’t guarantee to find the optimal solution, but it in most cases the solution should be fairly close to an optimal. However, in this implementation random restart was also unable to find a valid solution in a feasible maximal time, owing to the heuristic used. Of the two types of hill climbing, it was found that random hill climbing was more effective and less likely to get stuck but it was still not able to return a valid goal state.

**Conclusion**

For wide and shallow solutions breadth-first search should usually be more efficient, whereas for solutions where the depth is very high, depth-first search with iterative deepening should be better in many cases. However, the iterative deepening implementation proved to be extremely slow due to being unable to skip repeat states in an intuitive way, and so a different implementation where un-optimal answers were returned was used instead. This was faster than Breadth-First search but had much longer solution length.

A\* search performed comparatively similarly and sometimes slower than BFS, which could be owed to the heuristic used. It was expected that it should expand fewer nodes than BFS but the number expanded was generally similar to BFS, which could also be due to the heuristic not being strong enough, though the one chosen should be admissible. Steepest Ascent Hill Climbing was found to be least effective in this implementation due to the issue of being prone to getting stuck in local maximum combined with a heuristic that was not advanced enough. Due to this, it was unable to output a solution in all cases. When Random Restart Hill Climbing was tried, it was unable to restart a goal state, with it instead getting stuck at local maxima or in a loop.