Problem Formulation

1. Initial State

The initial states for each problem are the forty problem boards that are read in from the rh.txt file. These are stored into a 1D list called problemStorage and indexed according to their corresponding problem number. Each problem is then inherited into its corresponding state object as defined in the single_State class.

2. Possible actions

The possible actions that any horizontal car or truck, and any vertical car or truck can make are defined in the do_Move expand function. This function takes a node as input and appends every possible movement as a new problem state in a 1D list called generated_States. This can be expressed as:

ACTIONS(In: Initial State) = {If: can do move ←do move, generated_States ←new problem state}

The child_State within this function is then returned to the initial state. The process then repeats until all generated states for the input node have been generated.

3. Transition Model

All successfully implemented search algorithms will result in the transition model:

RESULT(In: Initial State, Go: Goal State) = In: Goal State

This means that all search algorithms are fed the initial board state. They then must find the goal state and report the sequence of moves to get there.

4. Goal Test

The goal state is defined by the goal car being in the goal state. This is the case if the goal car 'X' exists in index 17 of the problem line. This is the point right in front of the open barrier and indicates that X has a clear path to move into the goal state. This can be defined as:

{X In: Index 17 of problem}

5. Path Cost

The path cost for BFS and IDS are considered to be 0. Since there is no path cost these search algorithms are free to traverse the nodal network. This is not the case for A* and hill climbing as they depend on explicit path costing to operate. The heuristic (h) determined for A* is the number of cars between the goal car and the goal state. This is defined as the total number of blocking cars. The goal car in the goal state (index 17) needs to have a blocking cars value of 0 which is the required characteristic of the goal node. Nodes need to be searched according to their h value of blocking cars to find the node with h = 0. For the hill climbing algorithm the goal state will also have a heuristic h value of 0. This path costing is determined by the distance between the goal car and the goal state. If the goal car is in the goal state, then h = 0. Otherwise it will keep looking for local maximums, lowest value of h, and random restarting until it finds the goal node with h = 0.

Software Design

The first data structures used are the 1D lists that store the problem lines (problemStorage), and the given solution lines (solutionStorage). This stores each problem's problem line and solution line into a corresponding index in the 1D list.

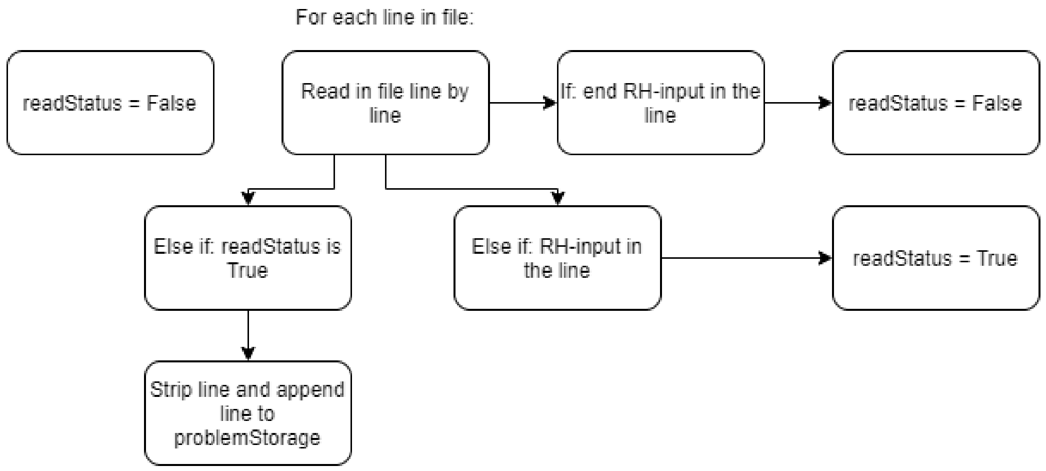

For each line in file:
readStatus = False
Read in file line by line
If: end RH-input in the line
readStatus = False
Else if: readStatus is True
Else if: RH-input in the line
readStatus = True
Strip line and append line to problemStorage

Figure 1: problemStorage Flowchart

As seen above, the initial state of the readStatus flag is set to false. The function then reads each line in rh.txt one by one between --- end RH-input --- and --- RH-input ---. When the line contains the RH-input, the readStatus flag is set to true which activates the storing of each line. These lines are appended to a 1D list called problemStorage. The readStatus flag is then set back to false when end RH-input is in line and stops storing the problem lines.

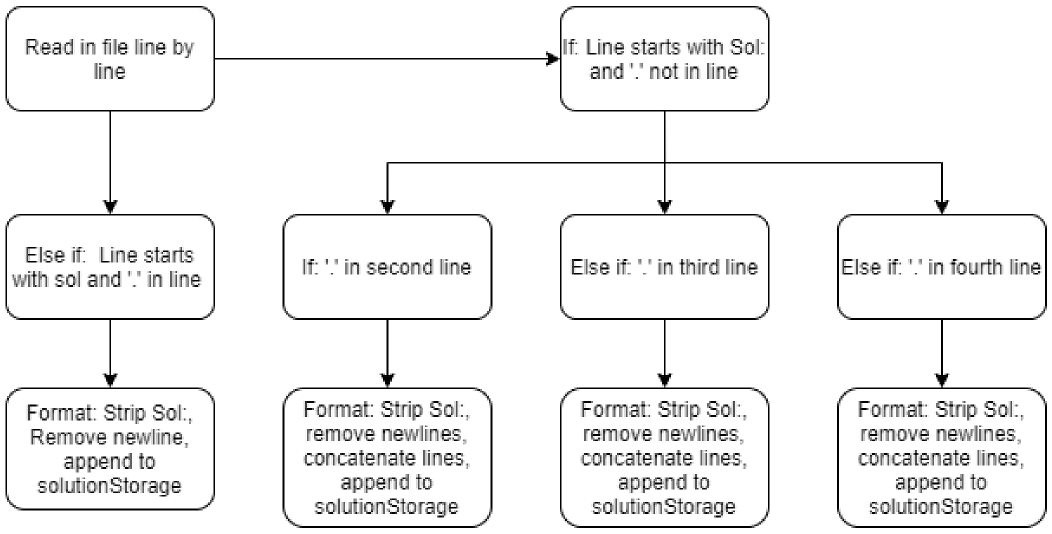

Read in file line by line
If: Line starts with Sol: and '.' not in line
Else if: Line starts with sol and '.' in line
If: '.' in second line
Else if: '.' in third line
Else if: '.' in fourth line
Format: Strip Sol:, Remove newline, append to solutionStorage
Format: Strip Sol:, remove newlines, concatenate lines, append to solutionStorage
Format: Strip Sol:, remove newlines, concatenate lines, append to solutionStorage
Format: Strip Sol:, remove newlines, concatenate lines, append to solutionStorage

Figure 2: solutionStorage Flowchart

The above represents the function for reading in each solution line. The longest solution line will extend over four lines and so the line on which the terminating character ('.') exists on must be found. Once it is found, the Sol: prefix and newlines are stripped, and these lines are concatenated if they are multi-line. The resulting string is then appended to the solutionStorage 1D list.

To the right is the UML diagram for the single_State class. The class has a single constructor function that takes a problem as input. Objects are then created by feeding the 40 problems as input to the single_State class and storing these in a 1D list called state_Objs. Each object contains its problem line and current move_Sequence performed by the expand function.

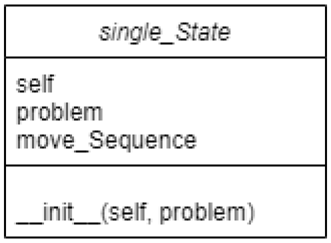


Figure 3: single_State UML Diagram

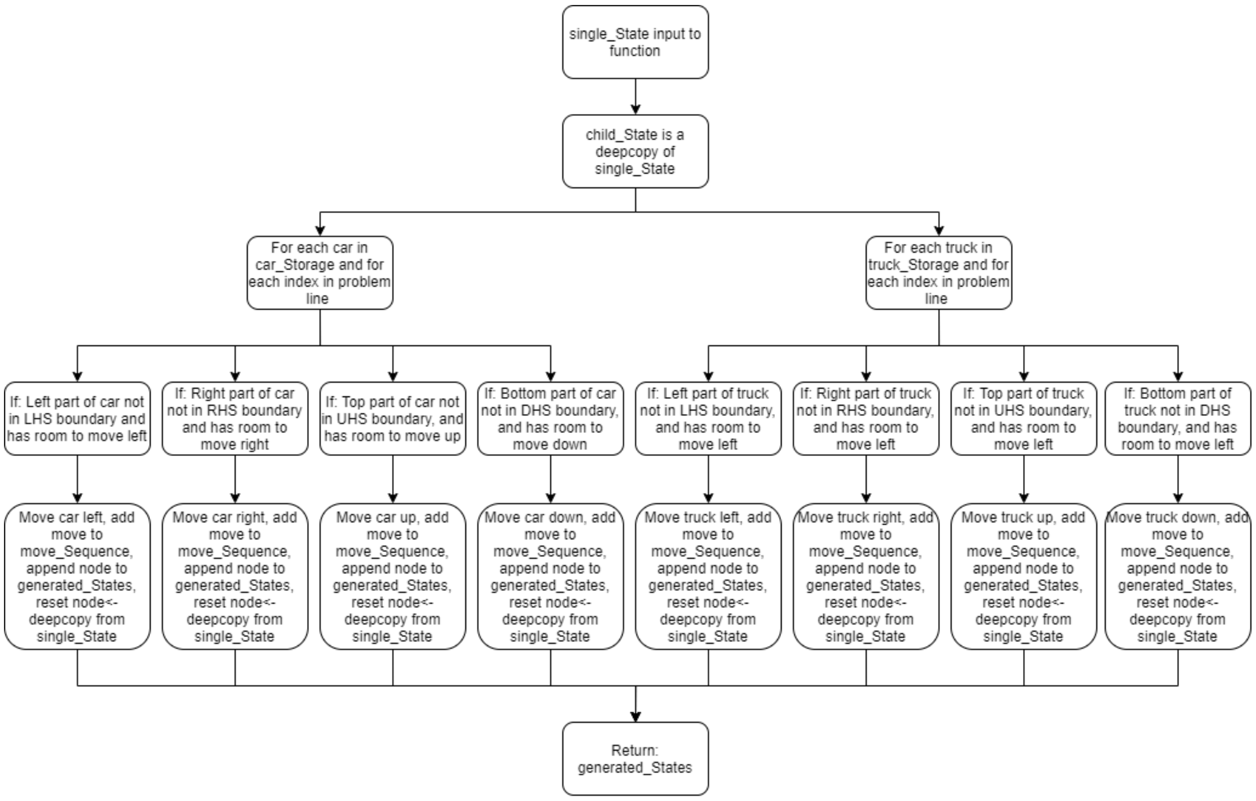

single_State input to function
child_State is a deepcopy of single_State
For each car in car_Storage and for each index in problem line
For each truck in truck_Storage and for each index in problem line
If: Left part of car not in LHS boundary and has room to move left
If: Right part of car not in RHS boundary and has room to move right
If: Top part of car not in UHS boundary, and has room to move up
If: Bottom part of car not in DHS boundary, and has room to move down
If: Left part of truck not in LHS boundary, and has room to move left
If: Right part of truck not in RHS boundary, and has room to move left
If: Top part of truck not in UHS boundary, and has room to move left
If: Bottom part of truck not in DHS boundary, and has room to move left
Move car left, add move to move_Sequence, append node to generated_States, reset node<- deepcopy from single_State
Move car right, add move to move_Sequence, append node to generated_States, reset node<- deepcopy from single_State
Move car up, add move to move_Sequence, append node to generated_States, reset node<- deepcopy from single_State
Move car down, add move to move_Sequence, append node to generated_States, reset node<- deepcopy from single_State
Move truck left, add move to move_Sequence, append node to generated_States, reset node<- deepcopy from single_State
Move truck right, add move to move_Sequence, append node to generated_States, reset node<- deepcopy from single_State
Move truck up, add move to move_Sequence, append node to generated_States, reset node<- deepcopy from single_State
Move truck down, add move to move_Sequence, append node to generated_States, reset node<- deepcopy from single_State
Return: generated_States

Figure 4: do_Move (extend) Function Flowchart

The above flowchart represent the do_Move function. This acts as the expand function to create possible states from an input node. When a node is passed to do_Move, the function will simulate possible moves that can be taken from that state and adds these new nodes to a 1D object list called generated_States. This list also includes the move sequence that was taken, e.g. if car 'X' moved to the right the move sequence would be XR. LHS, RHS, UHS, DHS boundaries represent the leftmost, rightmost, top, and bottom row. These indexes act as boundaries when the vehicle is moving in that specific direction to ensure the vehicle move does not result in an illegal state. For example, if a car was in index 0 (top left corner) and wanted to move left without a boundary condition, Python would interpret this as index[0-1] which moves half of the car into the last index (bottom right) of the board. The cars and trucks are distinguished from each other by storing them in different lists. The car_Storage contains all cars, and truck_Storage contains all trucks. This makes it easy for the program to know if a vehicle is a car or a truck and move it according to their specifications. Since the problems are stored one dimensionally, vertical and horizontal vehicles are determined by their index in the in the problem line. If the vehicle is horizontal, its char will repeat up to index+1 for a car and index+2 for a truck. If the vehicle is vertical on the other hand, the char will repeat up to index+6 for a car and index+12 for a truck. The child_State is reset to the initial state so that the program repeats with the initially supplied node.

To the right is the Queue UML class diagram. This is used for the frontier in the BFS search algorithm. The BFS frontier inherits from the queue, and then back inserts a node. The back_Insert function takes an item (node) as input and inserts this to the front of the frontier. It is called back_Insert because the BFS is a FIFO queue (first in first out) which means that as each element is added to the frontier they are being added to the end of the queue. Therefore, it is inserting nodes to the back of the frontier queue. Since python lists are not mutable, the iter function is used with a generator yield i. This means that with every object inserted into the frontier, the indexes of the list will be generated and allow the queue to be dynamic and mutable. The check_Empty function return a Boolean value based on whether the queue is empty. Lastly, the front_Remove function pops the last element. It is called front_Remove because it removes elements from the front of the queue.

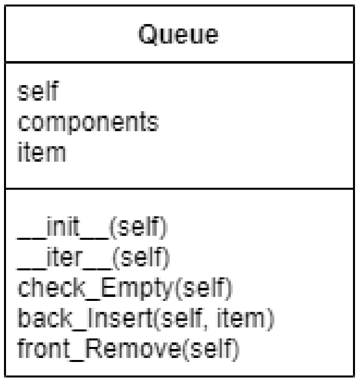


Figure 5: Queue UML Diagram

Search Algorithms

Breadth First :

The breadth-first algorithm takes advantage of using the queue class to represent the frontier in FIFO fashion (first in first out). The main block of the algorithm runs in a while loop with a time limit. The time limit is defined in the driver code as 5 minutes. The BFS reiterates this while loop until the goal state is found or until the time limit has ended. It first checks the failure state which is triggered when the frontier is empty. The frontier will only be empty if an error has occurred since nodes are inserted into the frontier at the bottom of the loop. The function then removes the node first in queue from the frontier, initially this will be the initial state. To avoid repetition, this node is appended to a list called explored_States which keeps track of each problem line that has occurred to avoid repetition. Next, the do_Move function is called into a variable called expanded_Node, which now contains all the expanded states for the current node. Two for loops are now called that iterate through these expanded nodes using a variable called child_State. If the problem attributed to the child_State has not already occurred, then it is appended to the explored_States. The goal state is defined as the car 'X' existing in index 17 of the problem line which is the point where the car can move off the screen. If the goal state has been triggered, a new move_Sequence is formulated. The new move sequence takes any duplicate moves that occur consecutively and combines this into one move. The success function is then called which prints the relevant information such as compute time, number of nodes searched (length of explored_States), depth (length of child node move sequence) and the updated move sequence. If the goal state is not triggered, the child node is added to the frontier queue and is expanded after the original frontier has finished expanding. The queue will repeat until either the goal state is found, an error has occurred, or the function has timed out. If the function has timed out, the failure time-out print function is returned.

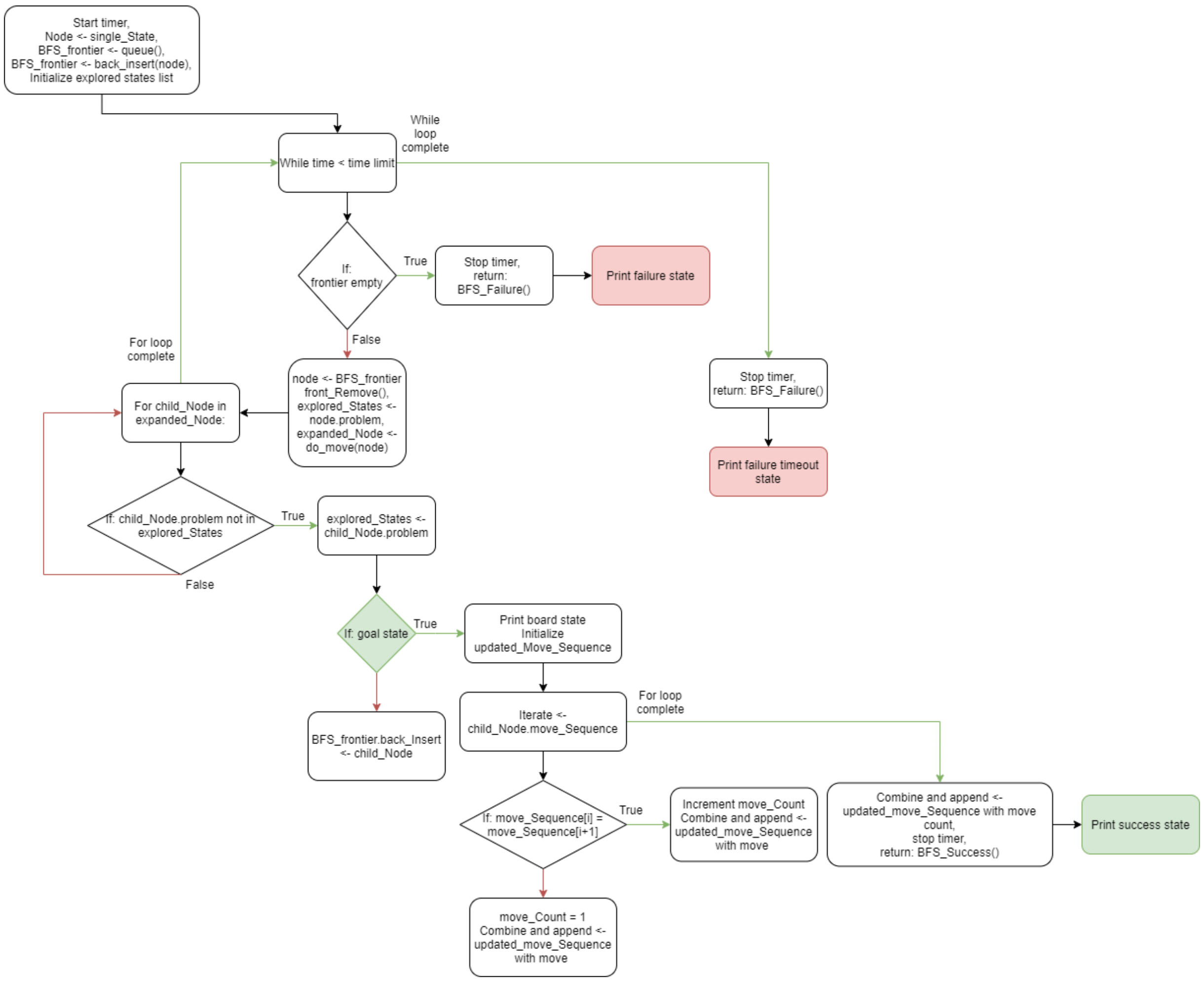

Start timer, Node <- single_State, BFS_frontier <- queue(), BFS_frontier <- back_insert(node), Initialize explored states list
While time < time limit
While loop complete
If: frontier empty
True
Stop timer, return: BFS_Failure()
Print failure state
False
node <- BFS_frontier front_Remove(), explored_States <- node.problem, expanded_Node <- do_move(node)
For child_Node in expanded_Node:
For loop complete
Stop timer, return: BFS_Failure()
Print failure timeout state
If: child_Node.problem not in explored_States
True
False
explored_States <- child_Node.problem
If: goal state
True
Print board state Initialize updated_Move_Sequence
BFS_frontier.back_Insert <- child_Node
Iterate <- child_Node.move_Sequence
For loop complete
If: move_Sequence[i] = move_Sequence[i+1]
True
Increment move_Count Combine and append <- updated_move_Sequence with move
Combine and append <- updated_move_Sequence with move count, stop timer, return: BFS_Success()
Print success state
move_Count = 1 Combine and append <- updated_move_Sequence with move

Figure 6: BFS Flowchart

Iterative Deepening:

Unlike BFS, iterative deepening does not utilize the frontier queue. By definition, IDS utilizes a LIFO stack(last in first out) which is implemented using a recursive DLS function. The driver code for the IDS takes in a maximum depth that the IDS will traverse to. The IDS function uses a for loop to iterate from 1 to the depth specified. This will call the recursive DLS function with the node (initially defined as the initial state), the limit (the index iteration between 1 and max depth) and [single.State.problem]. [single.State.problem] acts as the explored states list in the DLS function. This must be used otherwise the strings will be immutable. Calling this function returns two variables, goal_Node and cutOff_Occurred. Both are Boolean values that direct the flow of the IDS after the DLS recursion is complete. Inside the DLS function, the goal_State will be tested. If the goal_State has been found it will return the node that was the goal_Node and true for cutOff_Occurred. Else if the limit has been reduced to 0, the node will be returned with none (NULL) and the cutOff_Occurred as true. This means that no goal node was found, and cutoff occurred. If neither of these conditional statement pass the function will set cutOff_Occurred to false and expand the node. It will then iterate through each expanded node with child_Node like BFS and check if child_Node.problem already exists in the explored states. If it does not already exist, it will be appended to explored_States and the function calls itself again with limit-1. This is where the recursion occurs, and the function will keep being called until limit is reduced to 0. If the result of the recursion returns a node that is not none, then the goal_Node will be returned with cutOff_Occurred = true. Otherwise if it returns cufOff as true then cutOff_Occurred = true since the function was cut off at the limit. This process is one by one traversing down each node of the tree until the max depth has occurred. At which point it will climb back up to the unexpanded child node of the last parent node and repeat until the limit has been reached again. Unlike BFS which examines the tree row by row (iterating the search through each frontier), the IDS examines the tree almost column by column, traversing the tree down to the max depth for each expanded node one by one. If no goal state has been found the DLS function will return none for the node and the state of cutOff_Occurred. Now that we are back in the IDS function, the function will check if the goal_Node is equal to a node and not none. If it is the DLS has found the goal node and will print the success condition. Otherwise if no goal node was found and the cut off did not return true, an error has occurred somewhere, and the failure condition will be printed.

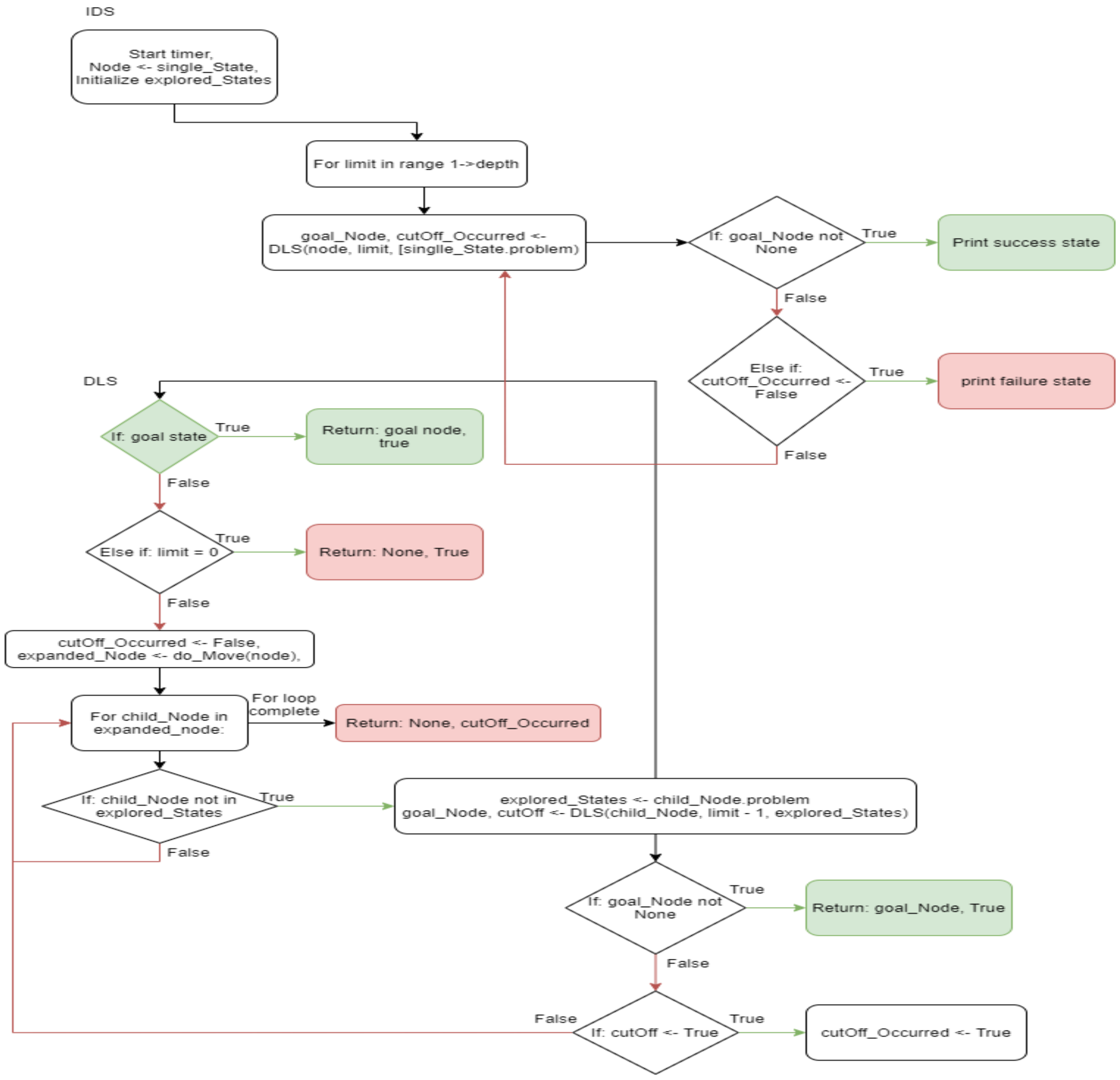

IDS
Start timer, Node <- single_State, Initialize explored_States
For limit in range 1->depth
goal_Node, cutOff_Occurred <- DLS(node, limit, [singlle_State.problem)
If: goal_Node not None
True
Print success state
False
Else if: cutOff_Occurred <- False
True
print failure state
False
DLS
If: goal state
True
Return: goal node, true
False
Else if: limit = 0
True
Return: None, True
False
cutOff_Occurred <- False, expanded_Node <- do_Move(node),
For child_Node in expanded_node:
For loop complete
Return: None, cutOff_Occurred
If: child_Node not in explored_States
True
explored_States <- child_Node.problem goal_Node, cutOff <- DLS(child_Node, limit - 1, explored_States)
False
If: goal_Node not None
True
Return: goal_Node, True
False
If: cutOff <- True
False
True
cutOff_Occurred <- True

Figure 7: IDS Flowchart

A*:

The A* search algorithm was only partially implemented and so instead of a state diagram flowchart, pseudo code will be used to outline the function of the algorithm.

```
Def A_Star_Heuristic(single_State)
For I in single_State ← block_Count ← number of cars blocking ‘X’ from goal state
Def A_Star(single_State, problem)
Node ← single_State
Explored_States ← node
Frontier ← node
While ← time() < time_Limit
Frontier ←- empty ← return failure
Expanded_Node = do_Move(node)
For child_Node in expanded_Node
If child_Node.problem not in explored_States ← explored_States.append(child_node.problem)
If ‘X’ == child.problem[17] <- return success
If A_Star_Heuristic(child_Node) < A_Star_heuristic(node) ← frontier ← child_Node
```

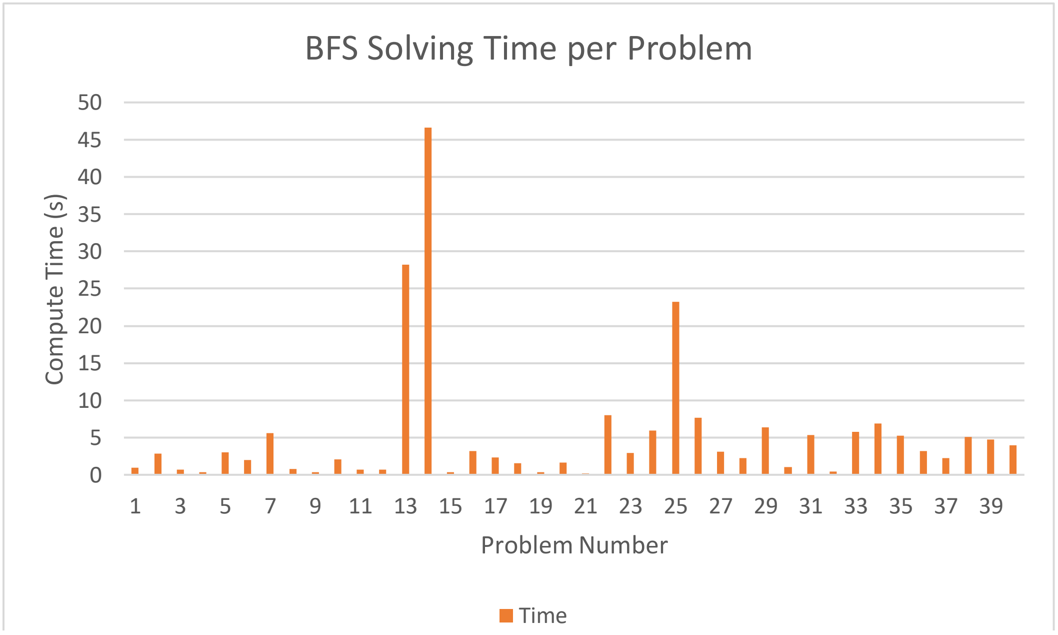
Hill-Climbing:

The Hill-Climbing algorithm was only partially implemented and so instead of a state diagram flowchart, pseudo code will be used to outline the function of the algorithm.

```
Def Hill_Climbing_Heuristic(single_State)
For I in single_State.problem ← if I = goal car ← h ← I -17
Distance from goal_State ← increment h
Def Hill_Climbing(single_State, problem) ← return local maximum
Node ← single_State
Expanded_Node ← node
Initial_State ← initial h
Loop ← time() < time_limit
For child_Node in expanded_node:
If child_Node.problem not in explored_States:
If child_Node (h) <= previous h ← return child_Node
Else ← random restart ← override initial state
```

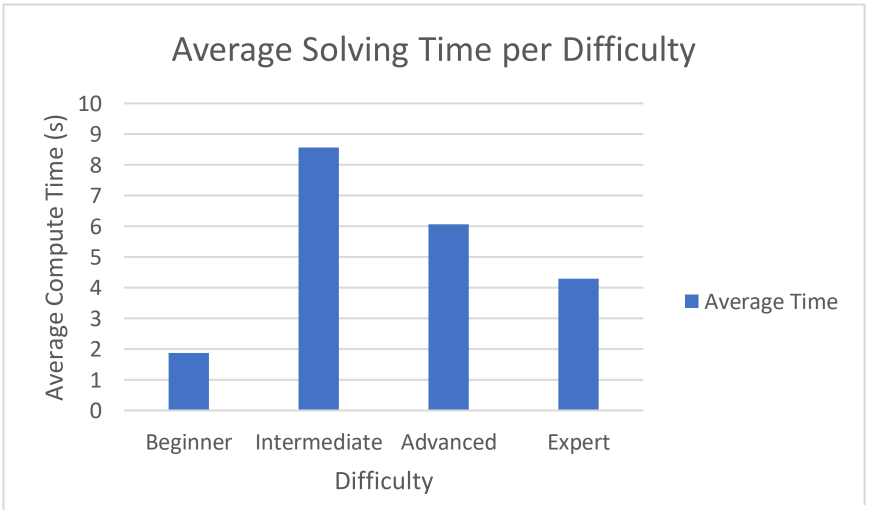
Experiments

Some considerable measurements to explore for the breadth first algorithm are the compute times per problem, average compute times per difficulty, difference in solution length between original solution and BFS solution, and average difference in solution for each difficulty. To find the compute times for each problem, a list called recorded_Times is added before the BFS function. This list is then initialized inside the BFS function as a global variable and each time is appended to the list. This time can then be plotted against the problem number and time taken to give a visualization of the compute time per problem.



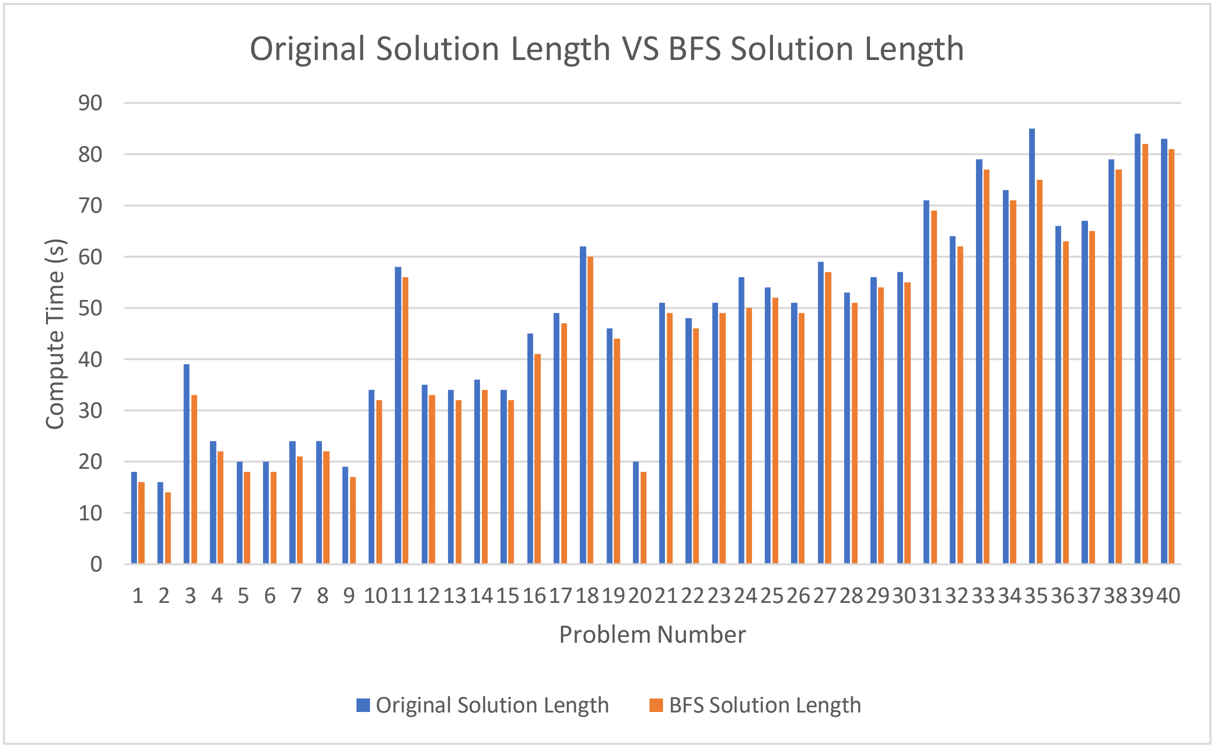
Graph 1: BFS Compute Time per Problem

As can be seen, there is an unusual spike in compute time for problem 14. This is unusual because problem 14 is of intermediate difficulty and is much longer than the computational time required for expert problems. To get a better understanding of the computing times per difficulty we can plot the average compute time for each problem grouped in their difficulties.



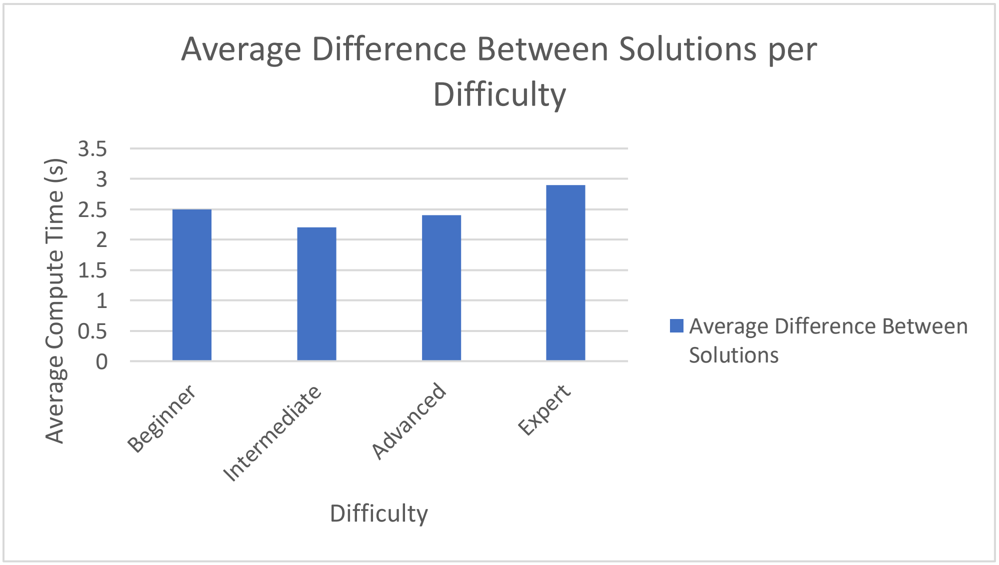
Graph 2: Average Compute Time per Difficulty

To see how efficient the BFS algorithm really is, the original solution length and BFS solution length can be compared. The BFS solution length is the BFS depth which is the length of the goal nodes move sequence. The length of the original solution can be found be found by iterating through the solutionStorage strings and summing each int. The results conclude that the BFS solution actually beats all of the original solutions.



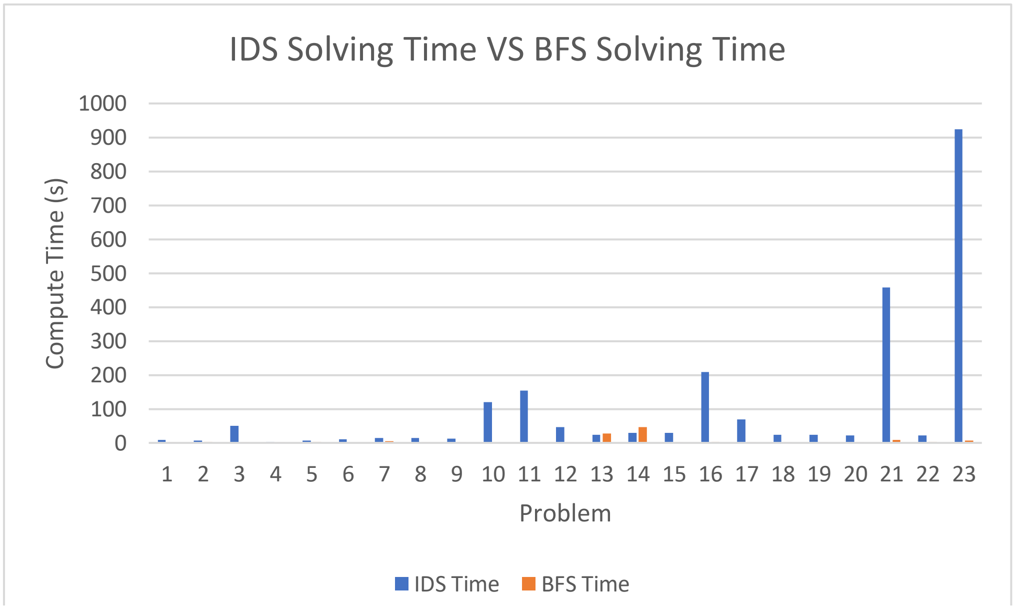
Graph 3: Original Solution Length VS BFS Solution Length

Most cases are pretty close as the most common difference between solutions is only 2 moves. This is not the case for problem 35, however. This is interesting because problem 35 is of expert difficulty and there is a solution difference of 10. The BFS solution took 75 moves and the original took 85. To get a better understanding of how difficulty effects solution length, the average solution difference per difficulty can be graphed.



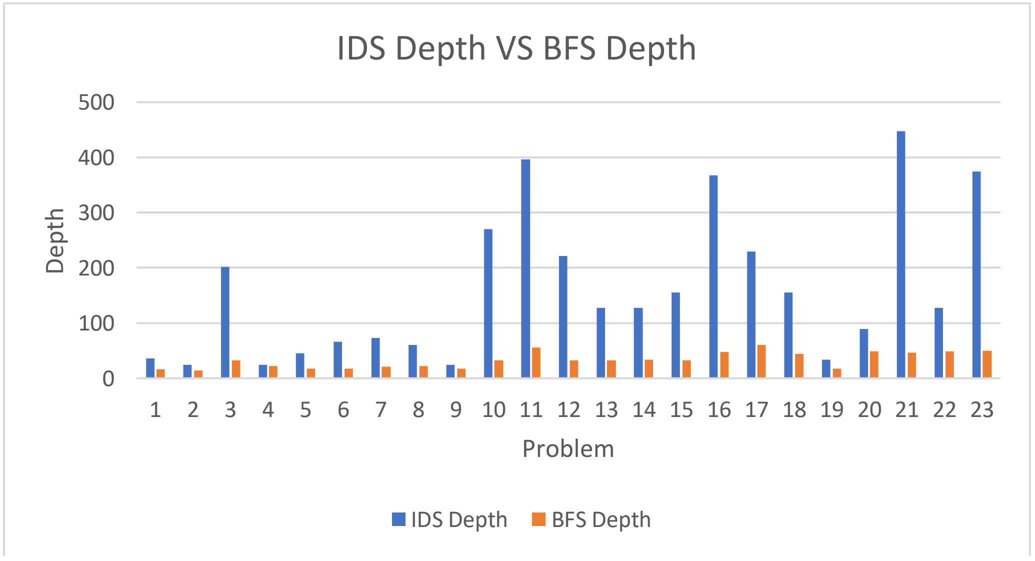
Graph 4: Average Difference Between Solutions per Difficulty

This proves that the BFS search algorithm is actually the most effective at higher difficulties with an average difference between solutions of 2.9. Just because the BFS seems to be effective, it does not mean it is the most effective search algorithm. IDS is theoretically the same time complexity and hence should yield similar results. Unfortunately, only around half of the IDS results were computable with a depth limit of 200 and 500. The difference between the implemented BFS and IDS was very noticeable in terms of computational intensity. From the results that were able to be computed, we can observe the differences in node depth, number of nodes searched, and compute time between the BFS and IDS algorithm. Note that problems 16 and 25-40 are not included in the analysis.

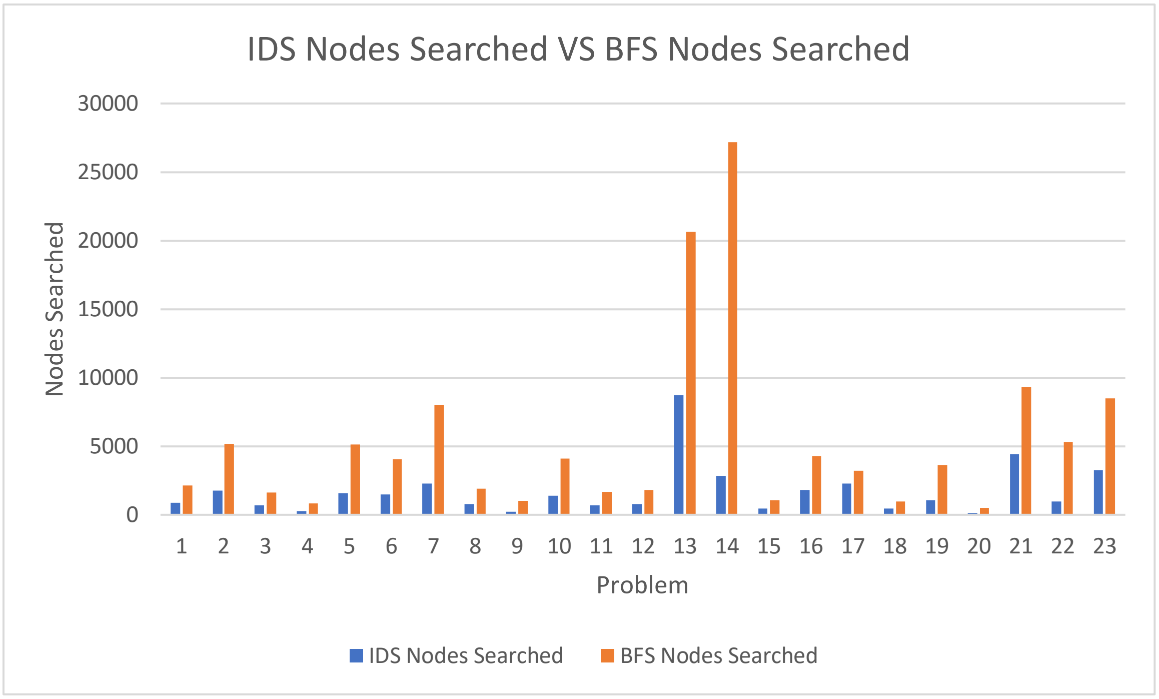


Graph 5: IDS Compute Time VS BFS Compute Time

As seen above, the IDS computational time is far higher than the BFS computational time, such that BFS computational time is hardly visible. As it takes so much longer than BFS, this can either be attributed to incorrect code or IDS becoming trapped in deep nodes that don't have a goal state. We can get a better idea if we compare the depths and total nodes searched of these two algorithms.



Graph 6: IDS Depth VS BFS Depth



Graph 7: IDS Nodes Searched VS BFS Nodes Searched

As seen from these two graphs, the IDS algorithm is searching much deeper than the BFS algorithm, but BFS is searching many more nodes. The IDS does not exceed the depth limit of 500 which means the code is working to a degree. It is interesting here how the IDS finds a goal state by searching fewer overall nodes but much deeper, and the BFS searches many more shallow nodes and is much quicker. Since the IDS is searching so deep, it may indicate that the massive time difference is a result of the recursive DLS function implemented. As the depth limit increments in the IDS function, level of recursion may approach 500. Since the time complexity for IDS is $O(b^d)$ where d = depth and b = branching factor, IDS has the potential to rapidly evolve in complexion. BFS also has time complexity $O(b^d)$ however also has space complexity $O(b^d)$, where IDS has space complexity O(bd). This gives a better understanding of the comparison between algorithms. The space complexity of BFS increases rapidly, as seen in the nodes searched graph (graph 7), whereas IDS is much smaller, where the opposite is true for depth. Since IDS is travelling much deeper, d is going to increase, dramatically increasing the time complexity power, whereas for BFS d is much smaller as it is increasing the breadth b, base number, more frequently.

Since the A* and Hill-Climbing algorithm were not implemented, it is more of a grey area as to how they would have performed in terms of BFS and IDS. First of all, this is because the time complexity for A* is complete dependent on the quality of the implemented heuristic. The best-case scenario for A* considering a perfect heuristic is a time complexity of O(d). However, the worst case is $O(b^d)$. The space complexity on the other hand is $O(b^d)$ which is the same as BFS. With the right heuristic, such as the prototyped car blocking heuristic, A* has the potential to outperform both BFS and IDS. The Hill-Climbing algorithm on the other hand is more unpredictable. This is because of the random restarts that restart the node position based on the given heuristic. Hill-Climbing also has the potential to become stuck on a shoulder close to the global maximum which may lead to only a percentage chance of computing a solution.

Conclusion

In conclusion, since only two of the algorithms were implemented in the code, only the BFS and IDS can be appropriately compared. In the experiment section, the BFS algorithm was tested in terms of solving time, average solving time per difficulty, difference between original solution length and BFS solution length, and the average difference between these two solution lengths per difficulty. The results conclude that the BFS solving time is actually pretty quick, at least when compared to the IDS results. Looking at the average time per difficulty, it was found that the algorithm actually struggled with the intermediate difficulty the most, followed by advanced, expert and then beginner. This defies my expectation that expert difficulty problems would take longer to solve. The average difference between solution lengths per difficulty show that the BFS solutions closest to the original solutions were the expert difficulty problems. This is most likely because there are more vehicles and hence more precise moves that are required. All solutions were quite close to the original solutions however, most being two moves more efficient, except for the outlier at problem 35 which reduced the original solution by 10 moves (85 down to 75). The IDS algorithm on the other hand was much slower than the BFS in terms of computation time. It also searched much deeper, and much fewer total nodes were searched. Originally this was attributed to a fault in the IDS algorithm, which may exist, but the results that were computable matched the general trend of their known time complexity and space complexity. Both BFS and IDS have a time complexity of $O(b^d)$. Since the BFS increases the breadth of the search more frequently, and the IDS increases the depth of the search more frequently, IDS is bound to have a higher computation time since it is increasing the power d in its time complexity. This is opposed to BFS increasing the base number. Vice versa, the space complexity for IDS is O(bd), whereas the space complexity for BFS is $O(b^d)$. This can be visualized in the depth graph (graph 6) and the nodes searched graph (graph 7). Since BFS has a higher space complexity, more nodes will be searched in the state space than with IDS, and since the IDS utilizes depth over breadth, it will travel much deeper than BFS.

A* and hill climbing were attempted but not implemented, but with the right heuristic, the A* search algorithm had the potential be the most time efficient algorithm. It would have had similar space complexity to IDS as well since their space complexities are the same. This means that A* basically takes the most appealing elements from BFS and IDS in terms of time and space complexity and combines them. Like mentioned earlier, Hill-Climbing is more unpredictable due to the random restarts and potential for shoulder error.