Murdoch University ICT203 S2 2024
Assignment 1 Report
Author: David Long
Student ID: 34252947

File Directory

```
├── Report.pdf
├── conda_env.yml
├── results
│   ├── frontier_sizes.png
│   ├── search_expansions.png
│   ├── search_times.png
│   └── solution_rewards.png
├── src
│   ├── analysis
│   │   ├── astar_search.py
│   │   ├── bfs.py
│   │   ├── dfs.py
│   │   ├── perform_analysis.py
│   │   ├── taxi_puzzle.py
│   │   └── ucs.py
│   ├── astar_search.py
│   ├── bfs.py
│   ├── dfs.py
│   ├── main.py
│   ├── taxi_puzzle.py
│   ├── test_action_mask.py
│   ├── test_astar_search.py
│   └── ucs.py
│
```

Sections

Introduction

This project aims to allow easy graph-search algorithm implementations for the Taxi-v3 environment provided by the Gymnasium Python module. The structure of the program allows for almost identical and efficient code implementations for 4 different search algorithms and demonstrates their solutions and performance statistics. This report explains a detailed technical description of the structure of the program and how it achieves solutions using state encapsulation with various search algorithms. The project includes an analysis directory, containing a program which runs performance analyses on each search algorithm and plots the results. This contains *modified* search algorithms which include additional code to collect the performance data (more details in Results section).

User Guide

Requirements

- conda >= 4.12.0
  https://conda.io/projects/conda/en/latest/user-guide/getting-started.html
- python >=3.10.14
  https://www.python.org/downloads/

Both must be added to the $PATH, and usable using the "**conda**" and "**python3**" commands, respectively.
To verify, you may use the commands "**conda –version**" and "**python –version**".

Steps

1. Install conda environment dependencies.

   Change directory to a1 using the command "**cd**". Output of the command "**pwd**" should be **"<path_to_a1>/a1**".
   Once the working directory is inside the a1 folder, enter the following command to install the project's python dependencies:
   conda env create –file conda_env.yml

2. Activate the conda environment.

   Once the conda environment has been created from step 1, use the following command to activate the environment:
   conda activate ict203_dl
   A successful activation will show the (ict203_dl) prefix on the command line.

3. Run the program.

   To run the python program, use either of the following commands:
   1. **python3 src/main.py**
   2. **python3 src/main.py "render"**

   The first command will run the program without any rendering.
   The "**render**" argument in the second command will trigger rendering to view the simulation visually.

   When the programming is running, it uses various search algorithms to explore the state of the current randomly generated environment. Once each search algorithm finds a solution, the program then simulates each of them. When rendering is disabled, the loop runs rapidly, continuously creating new environments, processing the search algorithms, and simulating their solutions.
   I recommend running the second command- rendering enabled to view the search algorithms' solutions closely.

   There is an optional performance analysis program included in the analysis directory. It requires an additional module which can be installed using the command:
   conda install matplotlib
   To run it, use the command: **python3 src/analysis/perform_analysis.py**

Design/Algorithm

State

To be able to process the Taxi-v3 environment, the environment state needs to be decoded. The environment state is created encoding its' variables into a single value:

(taxi_row * 5 + taxi_col) * 5 + passenger_location) * 4 + destination

The inverse of the encoding process can be viewed in taxi_puzzle.py as a function called decode_state(state).

The first step I took to designing this solution was to extract the state's individual data points, and I decided on the intuitive structure of a list containing 4 elements [taxi_col, taxi_row, passenger_location, destination]. The taxi column and row are simply the grid positions of the taxi. The passenger location member is a value that represents which of the 4 drop-off/pickup locations on the grid the passenger is currently at (index 0-3 inclusive). If the passenger is inside the taxi, the value is 4. The destination location is the same, but only 0-3 to represent the drop-off destination. The 4 drop-off/pickup grid positions are the following: [(0,0),(4,0),(0,4),(3,4)].

I created an array called state_grid_positions with these location tuples. When accessed with an index (0-3 inclusive) the result is the corresponding grid coordinates.

The goal state is a matching passenger location and destination, indicating a successful drop-off (and pickup). This task can be split up into 2 phases, where in the first phase the goal is to pick-up the passenger, and the second is to drop-off the passenger. In the code, the split goal phase logic is only present in the heuristic function present in astar_search.py (more details under the A* Search section).

TaxiPuzzle

The state gets stored inside of the class TaxiPuzzle, whose purpose is to act as a node in the search tree and be able to generate child nodes and calculate legal actions given the state. The class contains the rules and the environment data within its' methods, encapsulating the state.

TaxiPuzzle contains each variable, specifically:

- state (decoded 4 element list as mentioned above)
- parent (parent node),
- action (action that the parent node took to achieve this node's state. i.e. transition)
- path_cost (the total path cost g(x) up to this node/state)
- heuristic_function (optional heuristic function)
- evaluation_function (result of path_cost+heuristic_function, if a heuristic function is present)

The purpose of the TaxiPuzzle class is to store state data in nodes to create a search tree as shown in Figure 1. Figure 1 is an example of a state where the taxi's location is at (col, row) (0, 2), and is only able to move D, U, R. This basic example shows the transition between states using only legal actions, however in this program, the agent can perform illegal actions such as walking into walls, illegal drop-offs, and pickups. The path_cost that is transferred into child states is affected by whether the action is legal. In the case of an illegal action, the state may not change (such as walking into a wall), so the child node would have an increased cost and a matching state as the parent.

The objective of Taxi-v3 is to maximise the reward. This is achieved in the program by essentially carrying the reward as a cost (reward == -cost && cost == -reward). The search algorithms will *(excluding DFS)* achieve this goal by choosing a minimal cost path to reach the solution (thus maximising the reward).

Action Mask
The action mask for any given state can be generated using the generate_action_mask() method in TaxiPuzzle. This is a self-implementation of what the gym library's built in action mask generator is doing. It has been tested against all possible states to be 100% consistent with the built-in action mask output, provided by env.step() or env.reset(). The test code is present in test_action_mask.py and can be ran using Python3 after user guide steps 1 and 2 are complete. This testing file creates a Taxi-v3 environment and runs random actions (legal and illegal) and simulates the next step with env.step(). A self-implemented action mask is generated using the TaxiPuzzle method and compared with the action mask returned by env.step(). If there are any discrepancies the loop breaks with an error message. Upon leaving this running continuously it is found that all possible state-action masks are tested and no discrepancies were present. The purpose of the self-implemented action mask is to 'observe' the environment in a way that is 100% consistent with the env's built-in version without having to simulate the env instance. This completely decouples the TaxiPuzzle class from the env instance. In the program, the env is simulated after a solution is found to demonstrate and verify the solution against the 'actual' environment. After testing thousands of solutions using similar methods to the action mask test program, no invalid paths were found. Figure 1 demonstrates child node generated for legal actions only. However, in the actual program and environment, *noops* are permitted, where illegal actions incur costs and an unchanged state. *Noop(s) refer to an unchanged state, the result of an illegal/invalid action.*

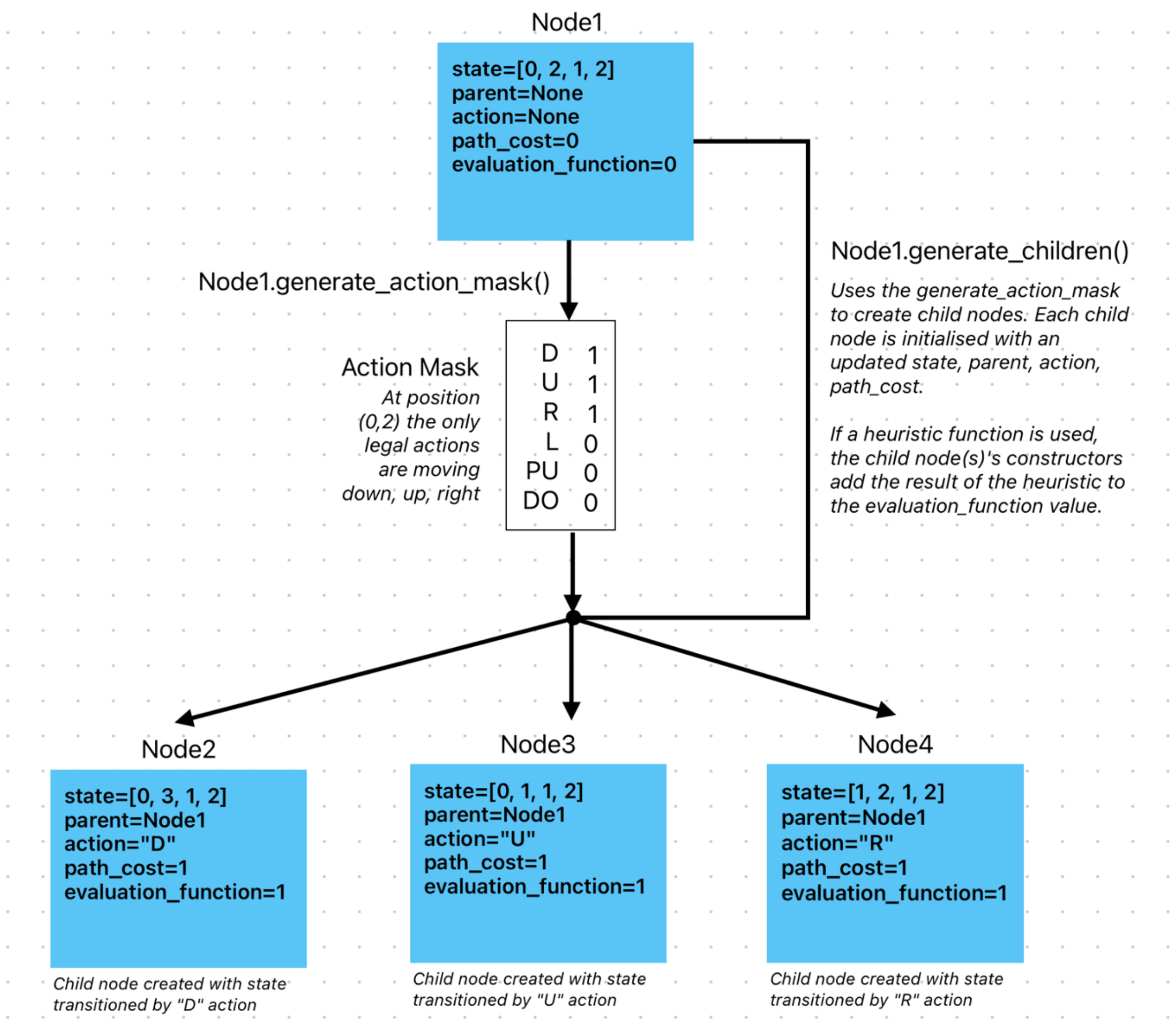
Node1
state=[0, 2, 1, 2]
parent=None
action=None
path_cost=0
evaluation_function=0
Node1.generate_action_mask()
Action Mask
At position (0,2) the only legal actions are moving down, up, right
D 1
U 1
R 1
L 0
PU 0
DO 0
Node1.generate_children()
Uses the generate_action_mask to create child nodes. Each child node is initialised with an updated state, parent, action, path_cost.
If a heuristic function is used, the child node(s)'s constructors add the result of the heuristic to the evaluation_function value.
Node2
state=[0, 3, 1, 2]
parent=Node1
action="D"
path_cost=1
evaluation_function=1
Child node created with state transitioned by "D" action
Node3
state=[0, 1, 1, 2]
parent=Node1
action="U"
path_cost=1
evaluation_function=1
Child node created with state transitioned by "U" action
Node4
state=[1, 2, 1, 2]
parent=Node1
action="R"
path_cost=1
evaluation_function=1
Child node created with state transitioned by "R" action

Figure 1: TaxiPuzzle node tree structure

Child Node Generation
Each instance of TaxiPuzzle can create child nodes for all actions at a current state using the generate_children() method. This function generates a child node for *all* actions (legal and illegal) with an updated state based on the action. The parent node carries its' path cost to the child nodes, with it increased by an amount dependant on the action and its' legality (as to match the behaviour of the Taxi-v3 environment). The action mask is used to determine the reward (-cost) to be inserted into the child node. In this function the cost is described in code as "reward" with values matching the specification described for the Taxi-v3 environment.

> *-1 per step unless other reward is triggered.*
> *+20 delivering passenger.*
> *-10 executing "pickup" and "drop-off" actions illegally.*

When the reward value is passed into the child node, *-reward* is used. This is so the search algorithms can sort the nodes based on their path_cost attribute, where higher is worse.

A* Search
For this project, I have implemented A* search, breadth-first search (BFS), depth-first search (DFS), and uniform-cost search (UCS). The reason I implemented multiple searches is because the TaxiPuzzle design made it trivial to implement these with barely any code difference in the search implementations themselves.
The A* search algorithm contains a heuristic function, whereas the others do not. This heuristic function can be used to describe the split goal phases for this problem. The heuristic function defined in astar_search.py works like so:

```
if passenger not in taxi and passenger not in destination
    return Manhattan distance from taxi position to passenger location
else if passenger in taxi
    return Manhattan distance from taxi position to drop-off destination
```

The A* search function uses this heuristic by passing the function into the TaxiPuzzle instances. The TaxiPuzzle class constructor will then use the function and add the calculated heuristic cost to its' total path cost. The search function works by storing TaxiPuzzle instances (as well as a count) in a priority queue. On each iteration it pops the next instance from the queue. The instance it pops will always have the lowest path_cost (path_cost+heuristic_cost) in the queue. It then checks if the goal has been reached at the node's current state. If not, the node is added to an explored list then its' child nodes are generated. If no child nodes are created, the loop will continue to the next iteration (the node is effectively removed from the queue with no additional nodes added after). If child nodes are created, they are added to the priority queue if they do not exist in the explored list.

The nodes are added to the priority queue with a count value. This is so the priority queue sorts nodes with equal path_cost+heuristic_cost values by their count value. The result is nodes with a lower count (nodes which were added first) are prioritised against nodes who have equal path_cost+heuristic_cost values. Every time a node is added to the queue, the count is incremented.
The first time a node is found with a state that satisfies the goal, the loop breaks. This node will have *an* optimal path since it has the lowest score in the priority queue (lower=better). 'An optimal path' is specified since there may be multiple paths with the same cost. Figure 2 presents a simple example of multiple paths with the same cost. For the Taxi-v3 environment, there are various paths between the two goal states (from initial state to pickup location, and

from pickup location to destination) which have the same step count (step cost = 1), the A* search function will find the first completed solution with one of these paths.

The chosen path may be affected by the order of which the children are generated if there are multiple equal-cost, optimal solutions. In the implemented function generate_children(), the order of the child nodes generated are in order of the actions taken prior- D, U, R, L, PU, DO. Each time a child node is added to the priority queue (in order of the actions) the count is added alongside it. It is determined that nodes which are added to the priority queue first with equal costs to other nodes are prioritised first. Therefore, the order in which child nodes are generated can affect which optimal solution is chosen.

For the nodes to be able to be sorted in the priority queue, I have added a less-than operator to the TaxiPuzzle class. This operator allows comparison between TaxiPuzzle instances, in this case the evaluation_function member (path_cost+heuristic_cost) is compared. I added this for search implementation simplicity, as otherwise the evaluation function would have to be added to the priority queue alongside the node instance.

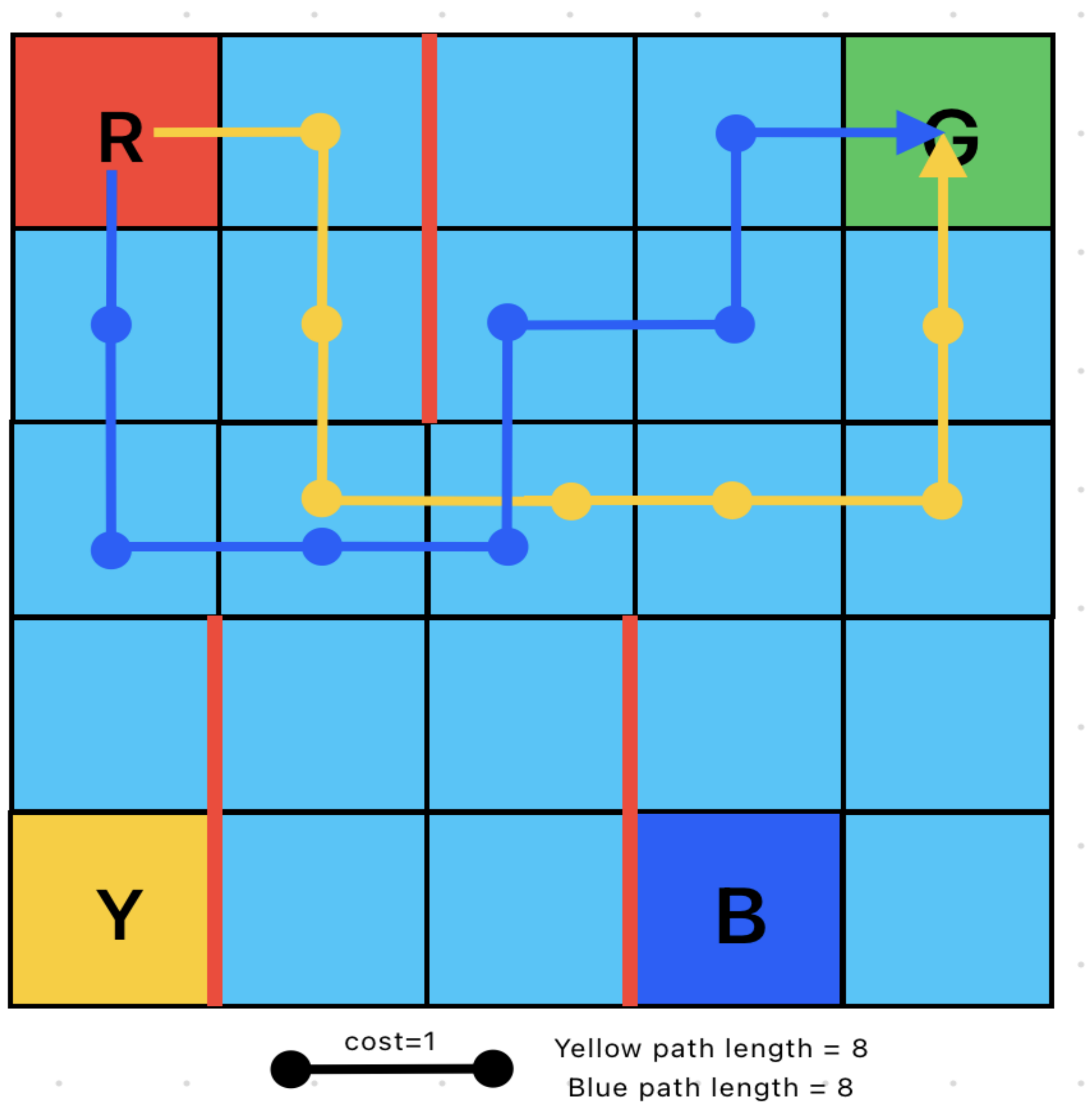

Figure 2: Multiple 'optimal' paths on an unweighted-step grid. Same layout as Taxi-v3

Other Search Algorithms
The implementation for the A* search algorithm is the *most-unique* relative to the code implementations for the other algorithms since it includes a heuristic function and a count value for each 'state' added to the priority queue. The implementations for UCS, BFS, DFS are simpler as the core loop is the same as the A* search algorithm, however instead they use a priority queue with only the node instance, a regular queue, and a stack respectively. They effectively all follow the following loop structure pseudocode (example for BFS):

```
create new queue
create TaxiPuzzle instance with initial state
create explored list
while true
        pop node1 from queue
        if node has reached goal
                return node solution
        end if

        if node not in explored
                add node to explored
        end if

        create children list

        insert node1 child nodes to children
        for child in children
                if child not in explored
                        put child queue
                endif
        end for
end while
```

The same code can be used for depth-first search, by simply replacing the queue with a stack. Uniform-cost search is also the same but uses a priority queue. When comparing the actual Python code in bfs.py, dfs.py, and ucs.py, the structure is the same and the code only changes on a few lines. The design of the TaxiPuzzle class, which encapsulates the state of the environment and creates child nodes with transitioned states based on valid actions allows for the code to be structured in this way. The methods in TaxiPuzzle are designed around the rules of the Taxi-v3 environment to be able to simulate it. The entire state of the environment is held in a 4-element array, and the state is manipulated using specified rules in the methods of TaxiPuzzle to allow this.

Gymnasium Library
The gymnasium library is used to visualise the route solutions returned by the search algorithms. It is also used for testing environment rules defined within the TaxiPuzzle class. The main loop of the program follows this structure:

reset environment → run A, UCS, DFS, BFS → for each solution: visualise it*

Each time the loop iterates, a randomised environment is created. There are a few if-statements in the main loop to ensure that the results from the solutions are consistent with the rules defined by the env. Errors are thrown if there are any discrepancies. It is structured this way to ensure that the results found by the program, actually valid.

Limitations

The class, and its' algorithms run completely decoupled from the env instance created by the following line in main.py.

env = gym.make("Taxi-v3", render_mode="human" if render else None).env

The reason I committed to this design was because I attempted to design the search algorithms around the state and action masks provided by env.step() but ran into design issues. The problem I experienced was that I wasn't able to figure out how to run env.step(action) to retrieve the action mask for child nodes in generate_children(). At this stage in the design, I had the env instance outside of the TaxiPuzzle class, but I also thought that I could potentially have an env inside each class as a member. However, I then couldn't figure out how to deep-copy the env to each child node in a non-complicated way that didn't seem like a worse solution. To solve these design issues, I decided to create a 100% consistent action mask generator, so that I would not need to simulate the env's environment to access these. The core design is completely decoupled from the env instance and does not require the gymnasium library at all to function, as it simply takes in a 4-element array representing the state and has all its' rules defined explicitly in the class.
The env instance and gymnasium library are used in this program for visual demonstration and to validate the results, after the solutions have been found. It is also used to set the initial state using a seed, with env.reset(seed=n). The initial encoded state value is retrieved from the env for the algorithms to process their solutions. Once the solutions are found, the env is simulated on the found paths for viewing and validation. Unfortunately, I could not figure out how to use the env to simulate steps for exploring, and instead it is done completely decoupled from it. I believe there is a way to use the visualised environment with gym.render() to view the search algorithms visually during their exploration, but it is not included in this implementation.

The evaluation_function function member in TaxiPuzzle is hard-coded to be computed as path_cost + heuristic_cost. This presents a limitation in that if wanting to implement a search algorithm such as greedy best-first search, whose evaluation function is only the heuristic function. In this case, a solution could be to 'drag' the evaluation function outside of the constructor and pass it into the class similarly to the heuristic function (though it would replace the heuristic function). This would allow more control over the class and allow easier implementations for other unique search algorithms.

Simulation Results and Discussions

The analysis/ directory contains a performance analysis program, which runs modified search algorithms, collects numerous data points on their performance and plots them. The search algorithms are slightly modified only including additional code to collect data. These modifications are expected to impact their performances slightly. The performance metrics are recorded for 500 different random states (likely covering most of the possible states), so the recorded metrics are expected to be accurately representative of the non-modified algorithms, but slower for the search time for example.

The performance metrics I recorded are for every time the search algorithms are ran. The search duration, total solution reward, total number of expansions and frontier size for each step are recorded by **analysis/perform_analysis.py**. Table 1 summarises the data collected for each algorithm.

	A*	UCS	DFS	BFS	Fig. n
Search Time (ms)	0.76	4.58	0.61	4.14	3
Solution Reward	7.69	7.69	0.82	7.69	4
Search Expansions	68	334.36	72.30	351.16	5
Final Frontier Size	30	424	7	896	6

Table 1: Performance results summary, all metrics are calculated means across all 500 runs.

Search Times

It is clear from Table 1 and Figure 3 that DFS has the lowest search time among all the algorithms tested with a mean recorded time of 0.61ms per run. I believe that DFS can quickly find the solution due to the nature of the exploration pattern, where it explores paths deeply until there are no further states available or the goal has been found. In the environment as shown in Figure 2 above, there are minimal dead-ends where DFS would have to branch, relative to other problems. DFS effectively simulates every action without the ability to step backward into a previous state due to keeping track of explored states. This pattern causes it to find a solution quickly. The Taxi-v3 problem has a reasonably simple environment, with minimal room to get 'lost' or stuck in search. A* search has a similar mean search time at 0.76ms. A* prioritises nodes which have a lower combined path cost and heuristic cost $f(x) = g(x) + h(x)$. The path in this environment is defined by the heuristic function. The heuristic function used in these analyses is a 2-phase Manhattan-distance metric. A* will, more-times-than-not, prioritise nodes which move it closer to the *current-goal*, as defined by the heuristic. This makes A* fast relative to UCS and BFS, as the latter explore nodes many more states, which often do not bring it closer to reaching the solution. Figure 5 and Figure 6 show metrics related to the node expansions and search behaviour.

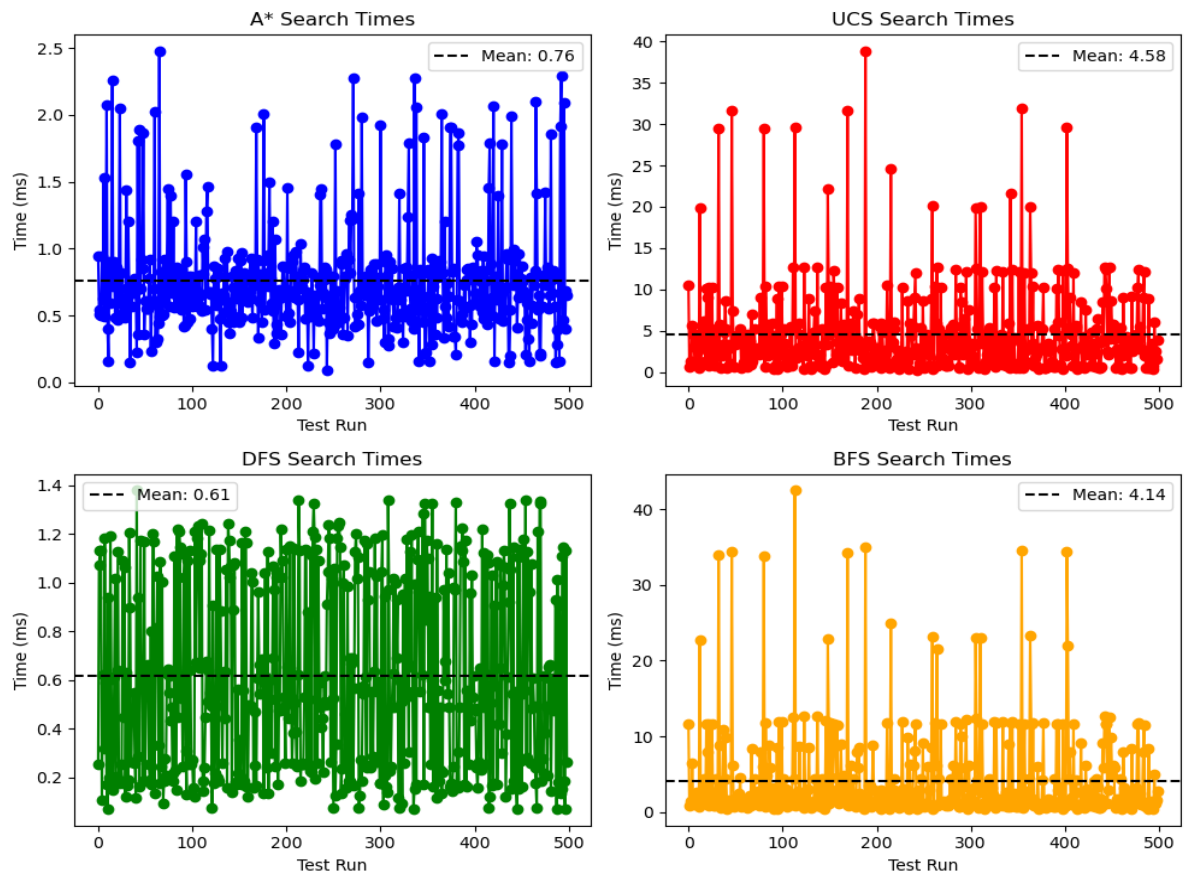

Figure 3: Search times for every run

Reward
The reward demonstrates how DFS does not find the optimal solution, compared to the other included search algorithms that do. The mean final reward across all runs is 7.69 for A*, UCS, and BFS. While the mean reward for DFS is 0.82, significantly lower. The matching results for the former search algorithms shows that across all test runs, the optimal solution was found every single time. According to the graphs shown in Figure 4 it is somewhat visible that they are identical, meaning that for every run, a same-cost solution was found. It is possible that the solutions differed, but the cost remained optimal regardless. *Interestingly, the spread across the highest and lowest reward remains mostly consistent at a difference of about 11 points.*

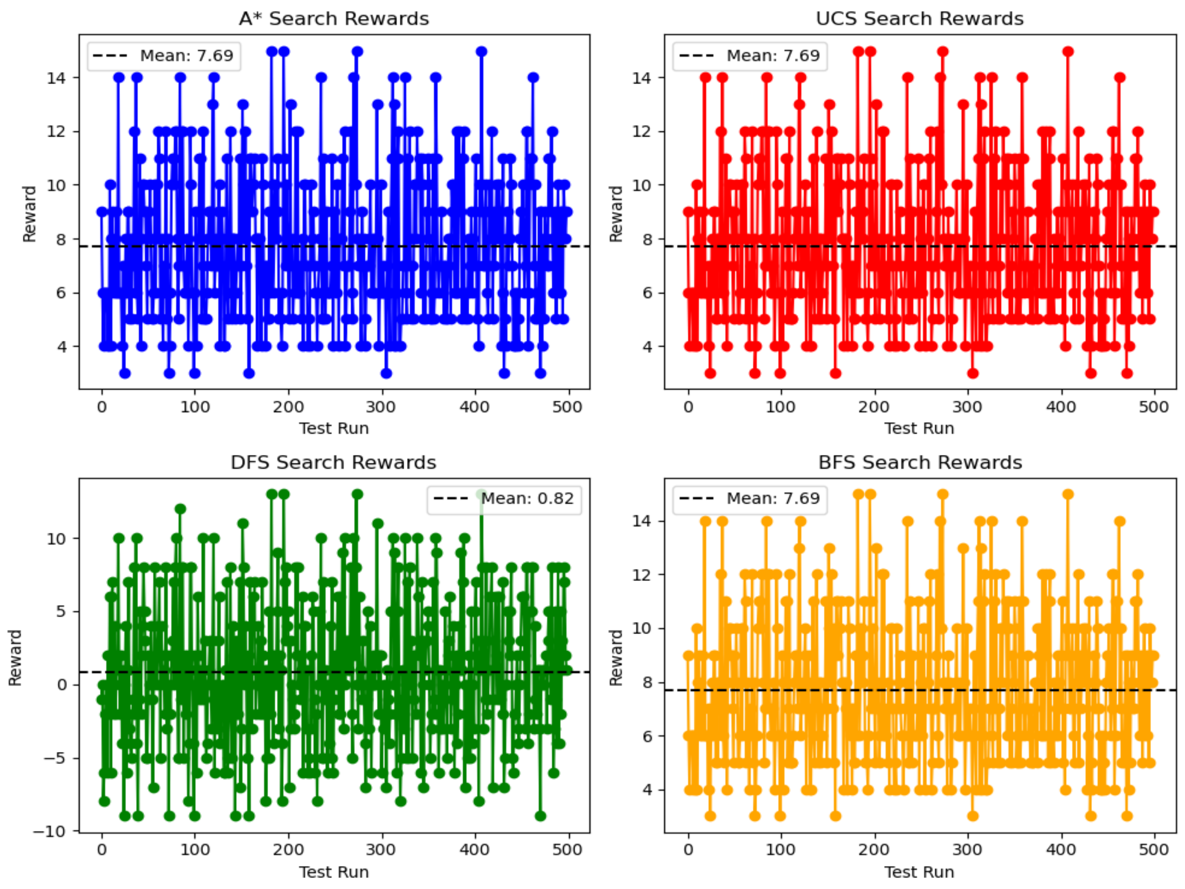

Figure 4: Final solution reward

Search Expansions
As the search algorithms iterate *'step'*, they expand their search on the next prioritised node in the data structure. For example, for every item in the priority queue, A* search expands on the next node with the lowest path cost + heuristic cost. When the search expands, it removes the node from the priority queue, then for every action at that node's state, it adds a child node (with the transitioned state and cost, etc) to priority queue. The expansion count is recorded for every single time the search algorithm removes a node from the data structure, effectively recording the number of iterations it took for a solution to be found. Figure 6 shows the number of nodes that are present in the data structure at every iteration, averaged across all runs. The results shown in Figure 5 demonstrates that A* search was able to find a solution with an average of 68 expansions. This is significantly less than UCS and BFS, but close DFS's mean expansions. The metric recorded for DFS can further explain the nature of

the environments' state tree created by it. The search algorithm was able to consistently find a solution without significant branching or backtracking due to dead ends. A* however, generally would be able to find the solution with lesser expansions relative to UCS and BFS due to the heuristic cost being considered when prioritising nodes to expand on.

BFS and UCS have significantly higher mean search expansions and frontier sizes over the course of their exploration, as shown in Figure 5 and Figure 6. This demonstrates the pattern that these search algorithms follow, where they *equally* expand their states to child states until a solution is found. The inability to expand to nodes which are already explored allows them to find a solution reasonably quick without getting stuck, and in this environment, every initial state has a possible solution. BFS, however peaks at a much higher frontier size as shown in Figure 6. The expansion rate grew exponentially for BFS, due to continuously expanding on nodes as they appear in the queue, while ignoring nodes which already exist (previous states). BFS does not prioritise nodes which are closer to the goal. All nodes are explored equally until a goal is found. A dip in the frontier size is visible in Figure 6, for UCS search. This can be described as the search removing nodes from the frontier without adding any child nodes. It is filtering out nodes which have no further states, which had the lowest path score at that point (since they were prioritised by the priority queue based on f(x)=g(x)). The frontier sizes shown in Figure 6 shows how many instances of TaxiPuzzle are stored at any given time (averaged over all runs). From this information it can be expected that BFS and UCS have the highest memory usage out of these algorithms, significantly higher than A* and DFS. DFS is generally able to find a solution from most states by searching deeply within a few search trees, without requiring storage of high number of nodes/states.

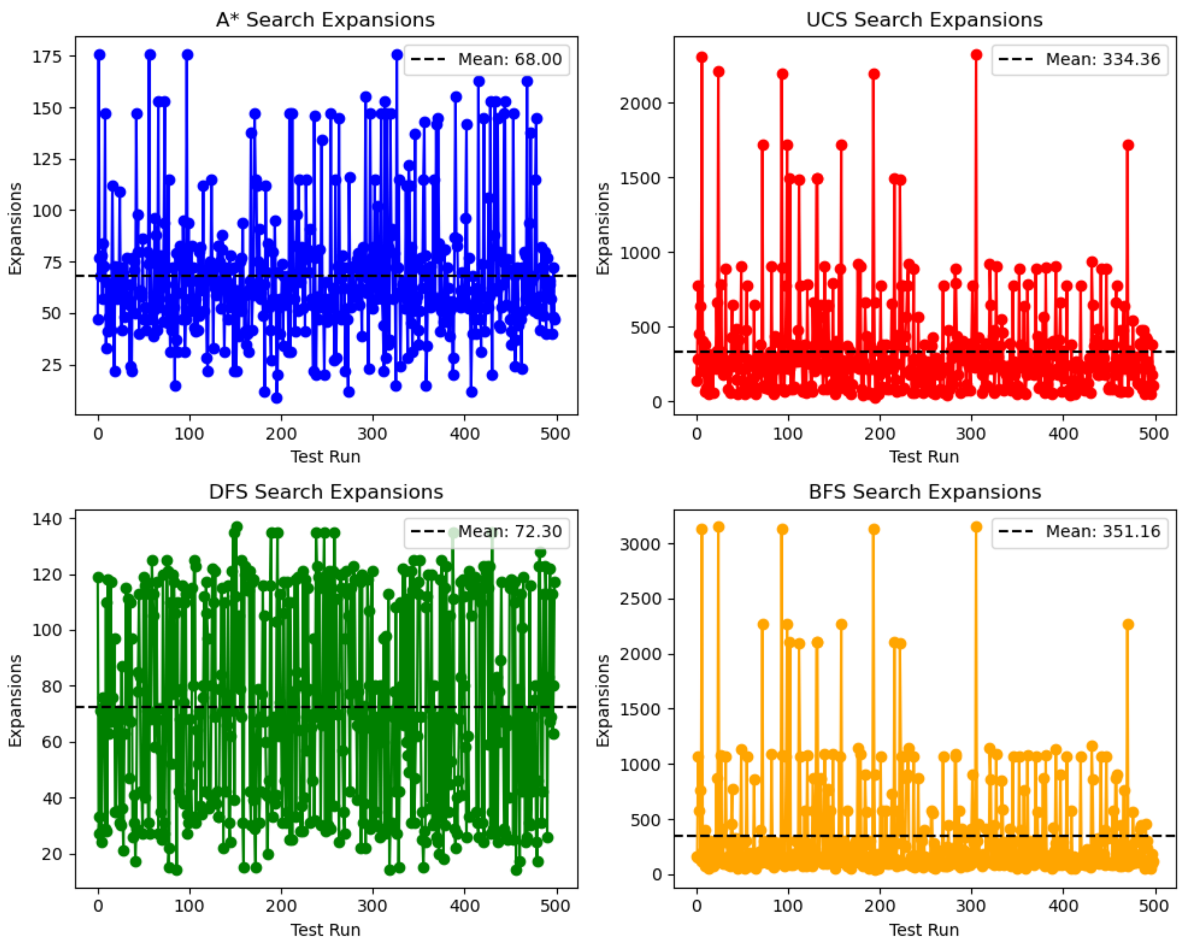

Figure 5: Number of total expansions 'iterations' for each run

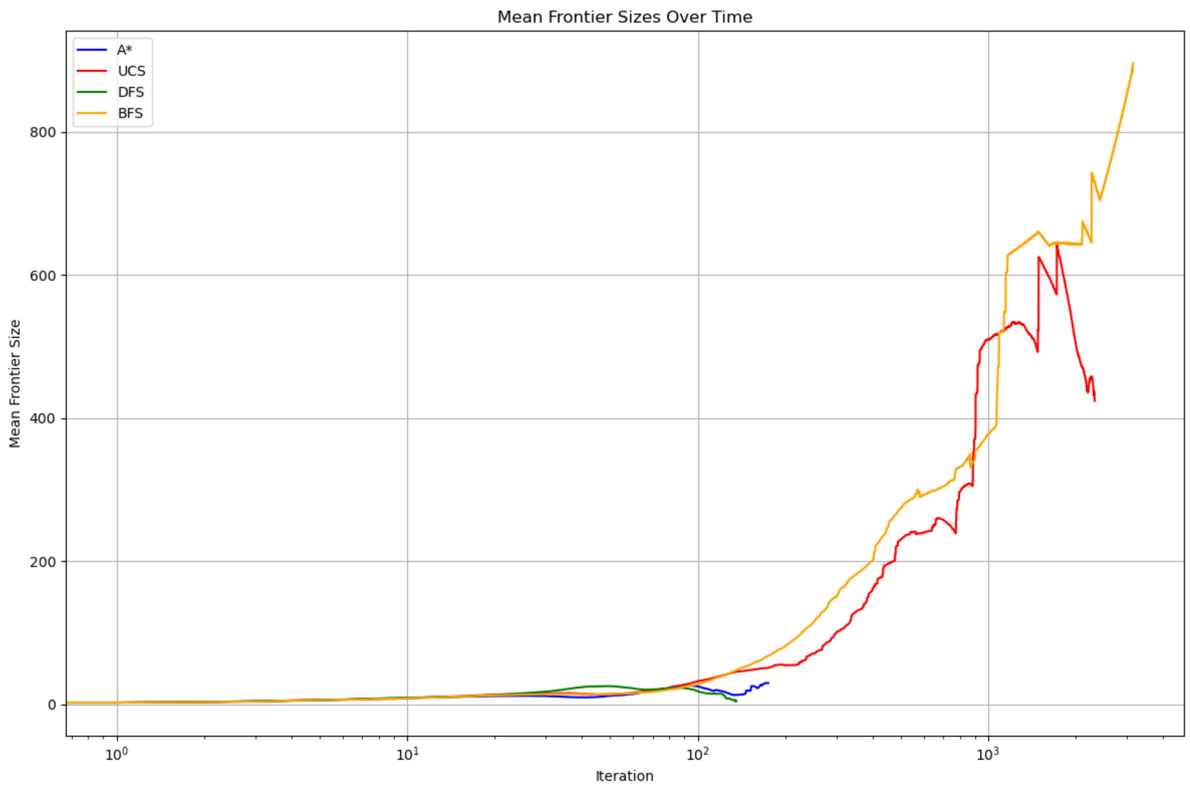

Figure 6: Mean frontier size across every run at every iteration