ARI2101

Fundamentals of Automated Planning

Assignment 2024/2025



Pawlu Spiteri

pawlu.spiteri.23@um.edu.mt

4105H

Matthew Farrugia

matthew.m.farrugia.23@um.edu.mt

Part 1:

Programming Language: Java 17.0.13

**State Implementation**

This state class encapsulates the board configuration, movement history, and heuristic information.



The board configuration is held in a one-dimensional integer array of length nine, with the integers ranging from 0 to 8, and 0 being the empty tile. To track the position of the empty tile an emptyTileIndex variable is maintained. This reduces constant looping through the board to find the empty tile, improving time to calculate possible moves and tile swapping.

Maintaining a reference to its parent state, null in the case of the initial state, enables retracing after a search concludes.

Maintaining move type, represented by a characters (1, u, r, or d), representing left, up, right, down respectively. This is used to represent the move done to reach the state from its parent state. This character is used during the plan validation step to validate a generated plan.

To support heuristic-based search algorithms, distance costs are maintained including cost to from initial state to state n (gCost), heuristic estimate from state n to goal state (hCost), and total estimated cost (fCost(), which is simply gCost + hCost).

By overriding equals and hashCode, the class ensures that comparisons and hash codes are computed based on the board configuration alone, ignoring differences in other attributes like parent reference or move done. This allows for states to be accurately identified and managed in data structures such as hash sets.

The state class also provides a children method to generate all valid successor states. Successors are generated based on the precondition: if the empty tile has an inbounds neighbour. For each valid neighbour a child state is created where the neighbouring tile and empty tile are swapped using the Swap helper function. The child state’s parent reference is set to the current state and the action performed to generate the child is recorded accordingly.



**Result Implementation**

//Runtime figures – unique states, duration

Duration of search is done within the search algorithm and does not include time taken to validate, generate and print out results.

Unique states visited is calculated by summating the number of the states in the closed hash set, and the number of states in the edge states data structure.

//Retrace path – moves, actions, boards

The retrace step traverses the path taken from the final state to the initial state, through parent references until parent is null. At each parent actions taken is incremented, the parent’s board is pushed onto a stack, and using stack’s first in last out order to reverse the plan. This correctly describes the plan as going from the initial state to the final state.

By the same logic moves are also pushed onto a stack but the last move is ignored, null.



Plan Validation Step - validity

After plan is in correct order, a plan validation step is executed to validate if a plan correctly reached the goal. This is done by peeking the initial board and simulating all the moves in order and if the board matches the goal board then the plan is marked as valid.



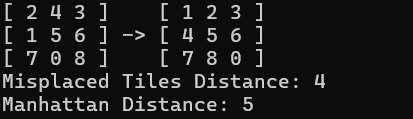
**Distance Functions Implementation**

The DistanceFunctions class contains the implementation of the manhattan distance and misplaced tiles functions.

Manhattan - Compute a lookup table for the destination tiles, mapping each tile value to its index. For every tile value in the start state, find its corresponding index in the destination state using the lookup table. Calculate the tile's 2D position by using the modulus operator to determine the x-coordinate and integer division to determine the y-coordinate for both the start and destination positions. Compute the absolute differences between the x and y coordinates, and sum these values to derive the total Manhattan distance.

Misplaced Tiles - Checks weather the tile values match at a specific index, if not increment cost.

Testing using Figure 2, gives expected heuristic values



**Search Algorithms Implementation**

- In all implementations a hash set is maintained to store explored states since its lookup and add times are both constant.

- In all implementations a hash side is maintained alongside the queue or a variation of it to speed up lookup times from O(n) to O(1). This doubles the size needed to store enqueued states but since layers grow exponentially with depth and lookups happen multiple times per state dequeue, it is a trade-off needed to drastically improve search times.

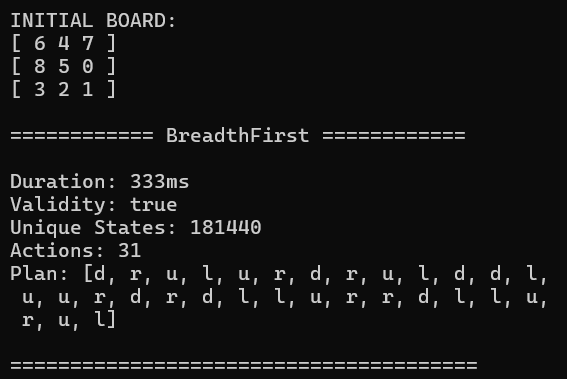
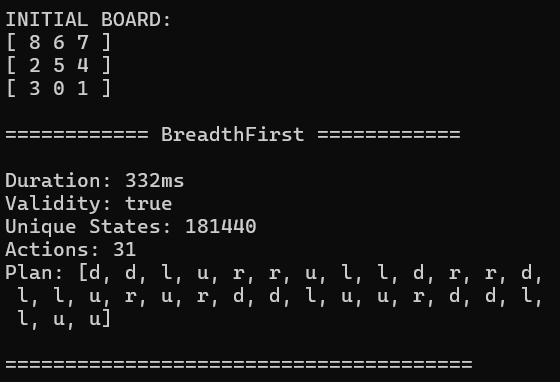
- States having boards identical to the boards of other states are considered redundant. Maintaining hash sets that store both explored and unexplored states (In the case of EHC a hash set containing the unexplored is not maintained), helps to lookup generated children states before enqueuing or pushing to avoid going in circles.

- A search will run until either the goal state is found or the data structure storing the unexplored states is empty.

**Breadth First Search Implementation**

The breadth first (BFS) search will run until either the goal state is found or the queue storing the unexplored states is empty. This implementation employs a queue to store unexplored states. This ensures that layer is exhaustively explored while enqueuing layer. Alongside the queue is a hash set used to speed up state lookups from Oto O. Since layers grow exponentially with depth and lookups happen multiple times per state dequeue, lookup speed is an important consideration.  
A hash set is maintained to store explored states since its lookup and add times are both constant.

States which have boards identical to the boards of other states are considered redundant. Maintaining hash sets that store both explored and unexplored states helps to lookup generated children states before enqueuing them to avoid going in circles. 

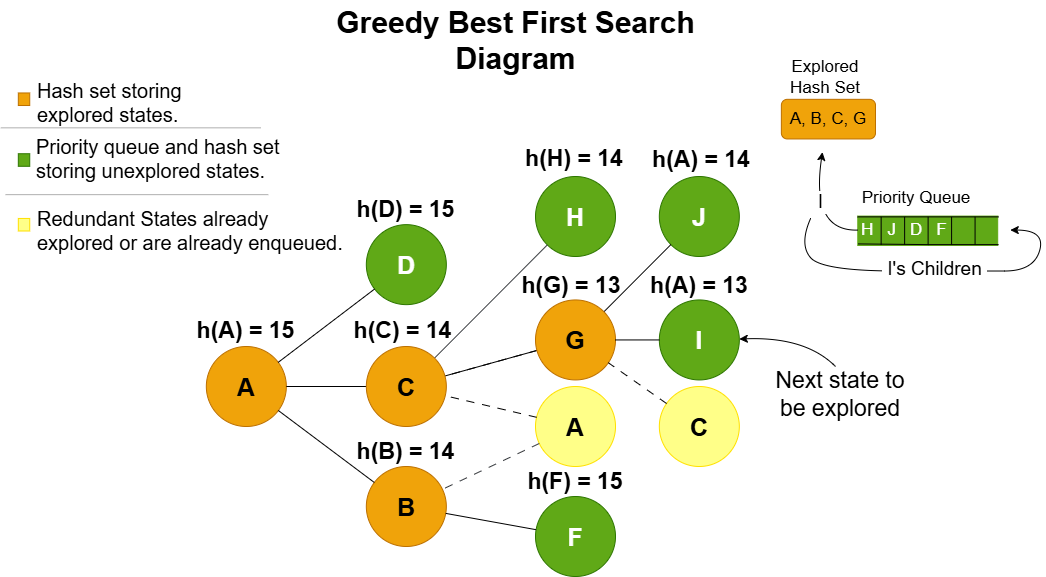


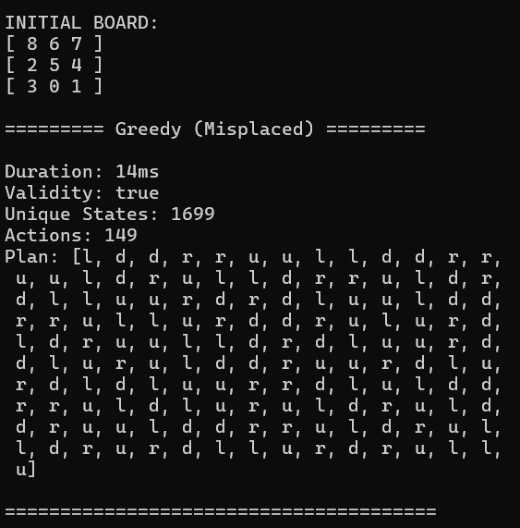
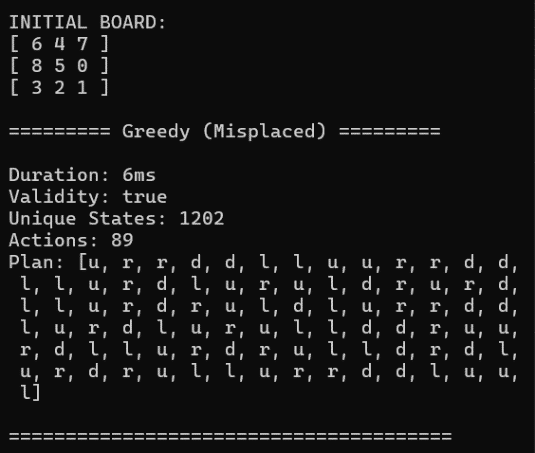
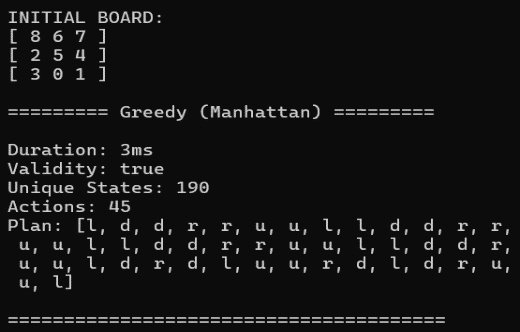
**Greedy Best First Search Implementation**

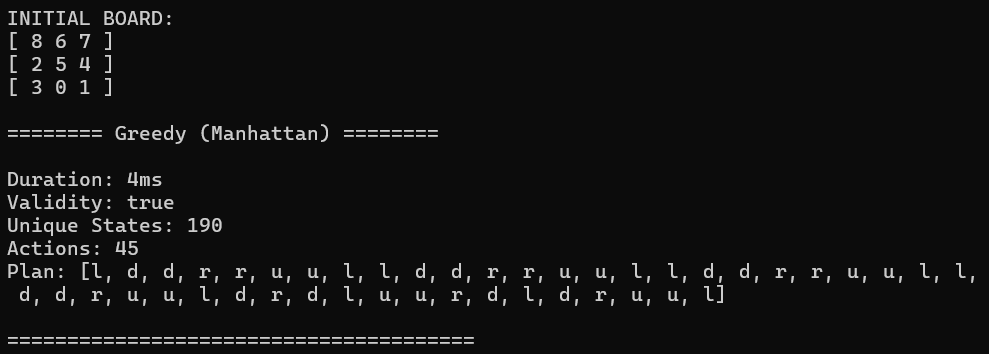
The greedy best first (GBFS) search will run until either the goal state is found or the queue storing the unexplored states is empty. This implementation employs a priority queue to store unexplored states. The priority queue enqueues according to the heuristic estimate to the goal state. It also maintains a hash set alongside the priority queue as in the previous search algorithm.

A hash set storing explored states is maintained as in the previous search algorithm.

Redundant states are handled by checking against the hash sets storing explored and unexplored states as in the previous search algorithm.

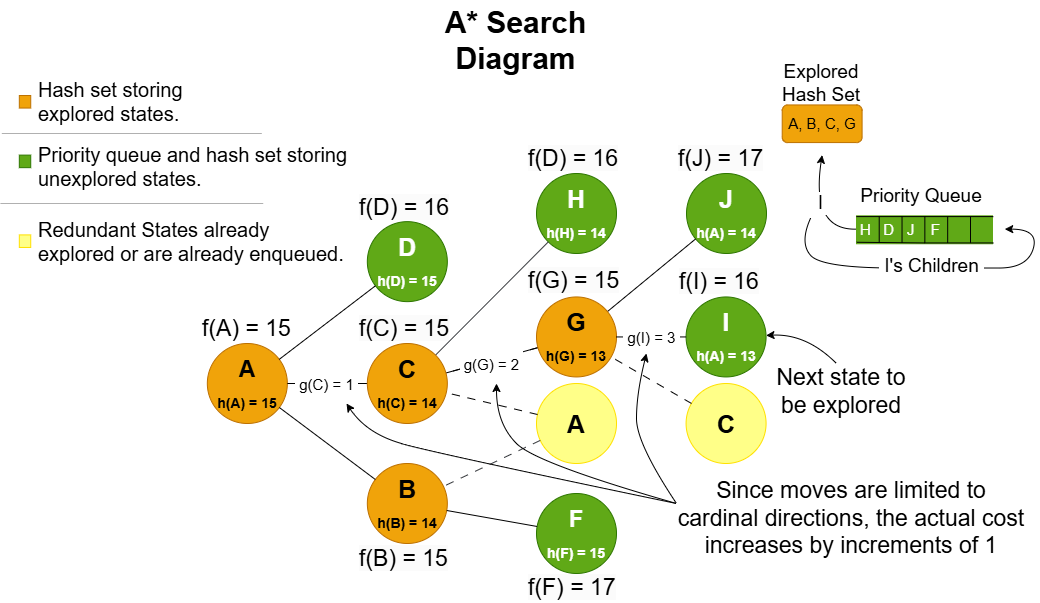
****



**A\* Search Implementation**

The A\* search implementation employs a priority queue to store unexplored states. The priority queue enqueues according to total estimated cost, and then if two states have the same total estimated cost, they are checked against their heuristic estimate to the goal state. It also maintains a hash set alongside the priority queue as in the previous search algorithms.

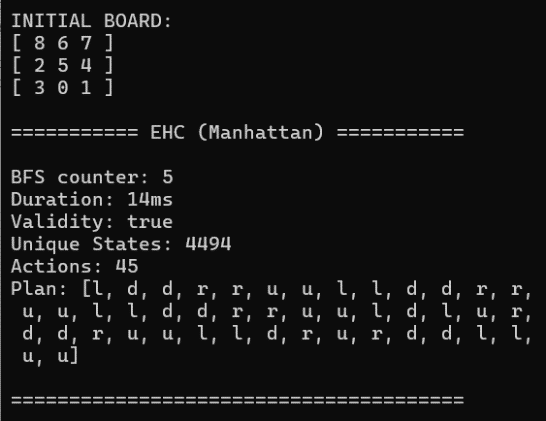
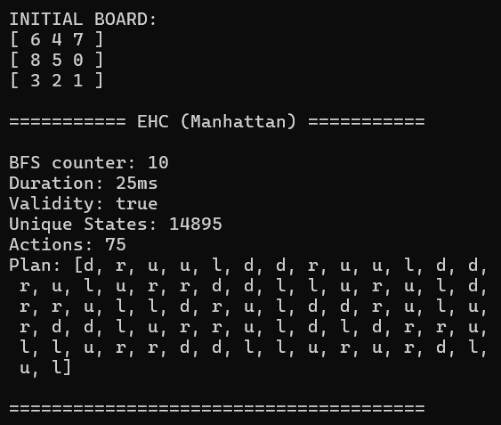
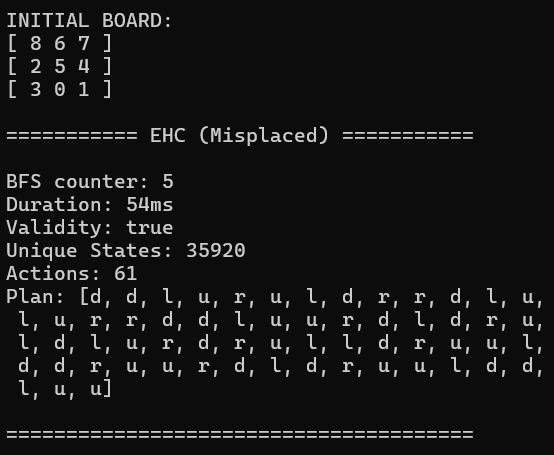
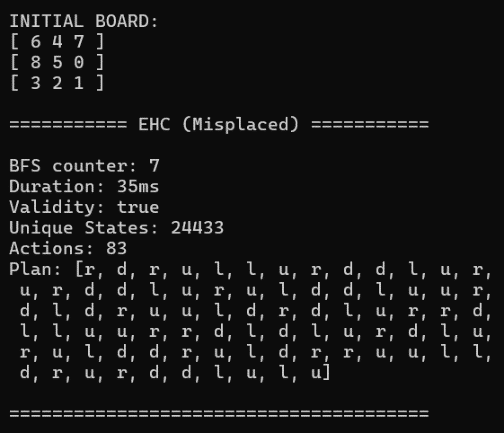
Since moves are limited to cardinal directions the distance between a state and any of its children is always 1. Therefore a child’s gCost is its parent’s gCost incremented by 1.

****



**Enforced Hill Climb Search Implementation**

The Hill climb algorithm is implemented by simply checking if a state has any children which are heuristically closer to the goal state, and if so, repeat the process with new best state. Otherwise perform BFS search until a state with a better heuristic is found, where again hill climb is executed. A hash set is maintained, storing explored states, by both searches. A queue plus a hash set are maintained for the BFS as in the actual implementation.



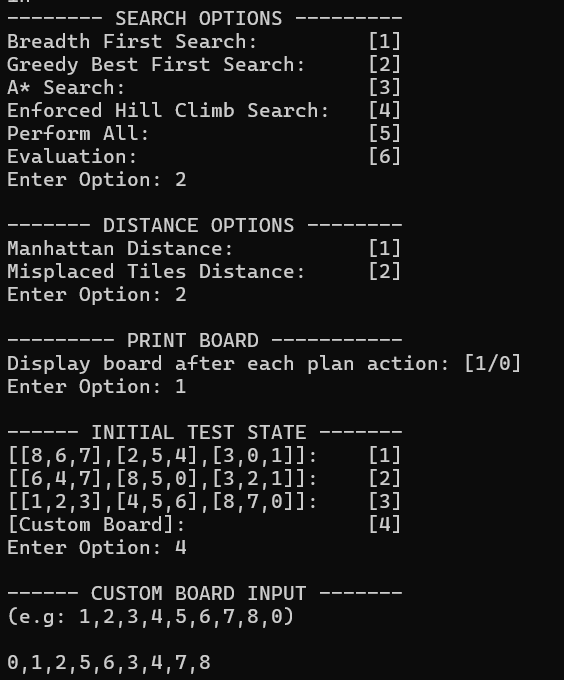
**Menu**

Search options include the four search algorithms, an option to perform all search algorithms with an additional, and an option to compute the evaluation metrics for the two initial test states. It also displays a summary and writes the summery to a csv file for further graph representation using a python script.

Distance function option is available for options [2], [3], and [4].

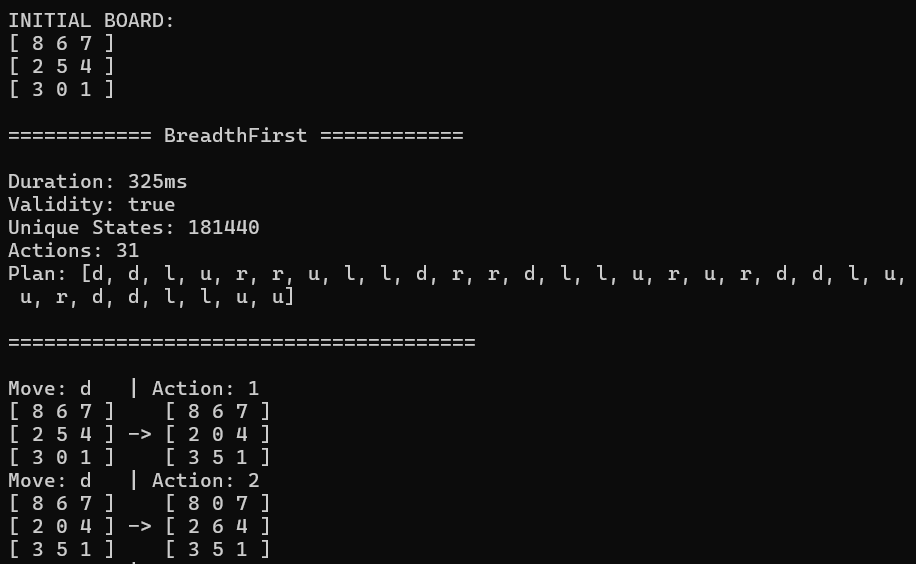
The print board option is available for each search algorithm except for option [5] & [6].

The initial state option offers three default states; Test initial state 1, Test initial state 2 and an unsolvable state, and an option to input a custom board configuration.

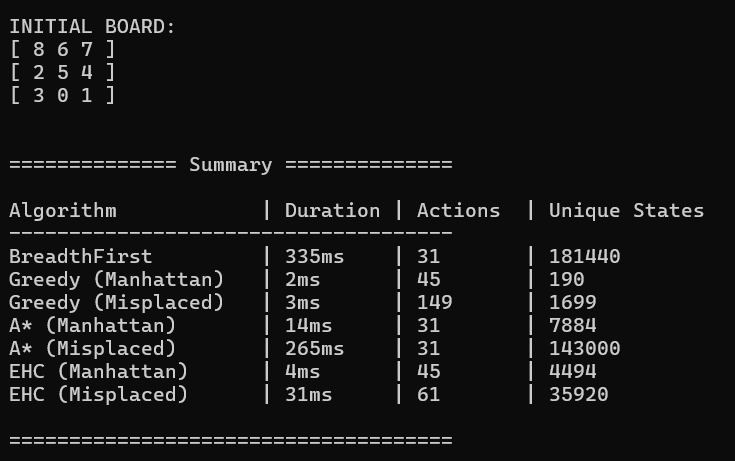


**Plan Output**

An example output including the required figures and information with the optional board at each step. The printing of boards was truncated. A summary is provided when the “Perform All” option is chosen for easy comparison of figures between different search algorithms.

****

****

** . . .**

**Evaluation**

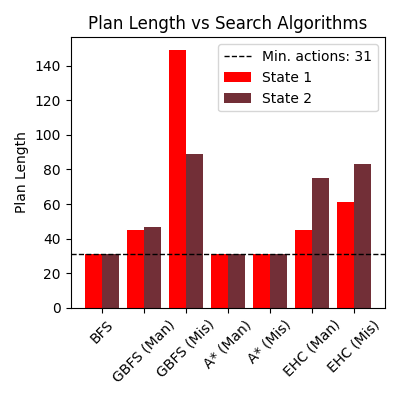
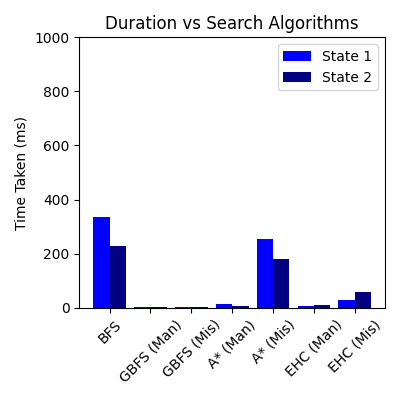
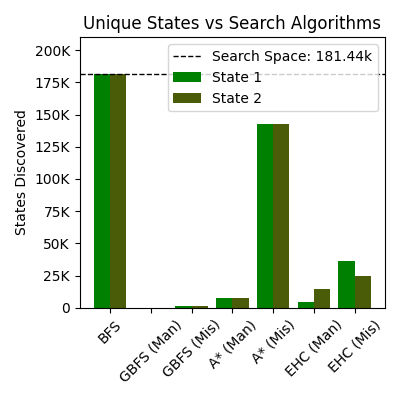
State Space is 9!/2 as inverted boards cannot be accessed. Include reference here  
A state is at most 31 moves away from an initial state. Include reference here

The breadth first search algorithm (BFS) search algorithm ensures the shortest path to the goal state by exploring all states layer by layer. However this exhaustive search has a significant drawback in that it generates all states up to the goal state’s depth. Since all states are at most 31 moves away, the entire state space is generated as seen when testing with initial test states 1 and 2 which are 31 moves away from their goal.  
Furthermore if a goal state is at layer and layer exists, layer will be partially explored. The percentage of layer to be generated grows as distance m grows. ( being the distance from the initial state of a layer).  
This makes the BFS impractical for problems with deeper state spaces or higher branching factors.

The greedy best first search algorithm (GBFS) yields very fast search times as it keeps its discovered state space small. However, GBFS exhibits a form of 'tunnel vision,' focusing solely on heuristic costs and ignoring the actual cost from the start state to a given node, unlike A\*. As a result, it does not guarantee the shortest possible plan, as it does not consider the full cost of reaching the goal.  
This makes GBFS very good at getting to a goal node quickly and when the Manhattan distance function was used was also able to keep unique states generated low but doesn’t ensure the optimal plan. The misplaced tiles heuristic though is seen to perform slightly worse in both number of unique states generated and also plan length.

A\* works very similarly logically to GBFS in that it evaluates the priority of expansion based on the heuristic estimate to the goal state from a node and the distance to that node from the start state. The only difference is that as opposed to GBFS, A\* considers the distance travelled to reach a node which enables A\* to generate an optimal solution.

The enforced hill climb search (EHC) “... is based on the commonly used hill-climbing algorithm for local search, but differs in that breadth-first search forwards from the global optimum is used to find a sequence of actions leading to a heuristically better successor if none is present in the immediate neighbourhood. The key bottleneck in using EHC is where the search heuristic cannot provide sufficient guidance to escape a plateau[1](https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume28/coles07a-html/footnode.html" \l "foot38) in a single action step, and breadth-first search is used until a suitable action sequence is found. Characteristically, EHC search consists of prolonged periods of exhaustive search, bridged by relatively quick periods of heuristic descent.” [1].



With a correlation of 0.97 between duration and unique states generated, the number of unique states generated serves as a reliable metric for evaluating the speed of an algorithm, particularly since execution durations may vary between runs.

Both Breadth-First Search (BFS) and A\* generate optimal plans. Among these, A\* (Manhattan) is the best-performing algorithm, followed by A\* (Misplaced), which shows a significantly poorer performance. A substantial 18.1-fold increase in unique states generated for both cases, leading to significantly poorer performance that approaches the performance observed with BFS when finding a plan. This consistency may suggest that the relative inefficiency of the misplaced tiles heuristic, compared to the Manhattan distance heuristic, scales predictably with plan length. It could indicate that the misplaced tiles heuristic introduces a uniform pattern of exploration that is less sensitive to variations in individual cases. Exploring why this fixed rate occurs might provide insights into the relationship between heuristic quality and state-space exploration efficiency in A\*.

BFS, being exhaustive and unguided, ranks last in terms of speed, as it generates an exceptionally large number of unique states. In fact, BFS demonstrates the slowest performance among all seven evaluated search algorithms making it the most computationally expensive.

While GBFS and EHC do not produce the optimal plan they consistently perform well in terms of speed. GBFS demonstrates the fastest performance generating by far the lowest number of unique states within the 100s range. Specifically, the GBFS (Manhattan) generated 8.9 and 8.1 times fewer states than the second best search algorithm evaluated which is GBFS (Misplaced). With A\* (Manhattan) consistently performing slower with the number of unique states generated in the 7k range. In case 1 EHC (Manhattan) performs better than A\* (Manhattan) and worse for case 2. When also comparing EHC (Manhattan) to EHC (Misplaced) the drop in performance is also less deterministic, hinting that EHC’s performance is more variable.

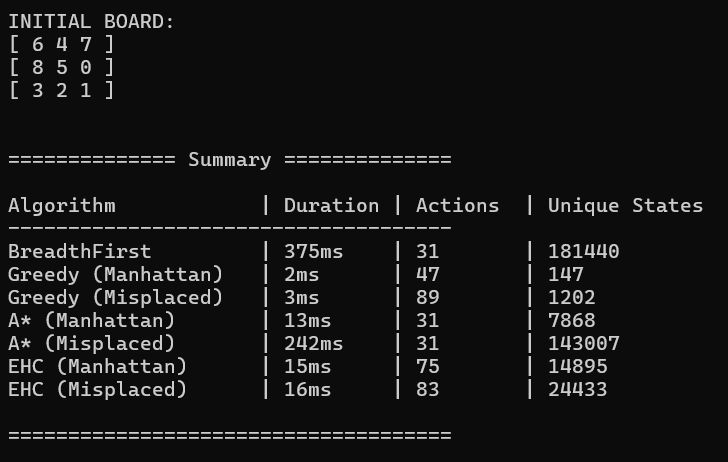
We can therefore conclude from the above that the misplaced tiles heuristic performed worse in the context of speed in comparison to the Manhattan distance heuristic in all cases where a search algorithm required a heuristic function.

For the two algorithms that do not generate optimal plans, the Manhattan distance heuristic produces consistently shorter plan lengths than the misplaced tiles heuristic. Specifically GBFS (Misplaced) performed the worst with substantially longer plans than its Manhattan counterpart, with a 3.3-fold increase Case 1 and a 1.9-fold increase for Case 2. Additionally, we observed that the plans generated with the misplaced tiles heuristic were inconsistently long with a 60 actions between the two cases. In contrast, the plans produced with the Manhattan distance heuristic were much more consistent, varying by only 2 actions between the cases.

Similarly, in the case of enforced hill climb (EHC), plan lengths increase by 1.3-fold for Case 1 and 1.1-fold for Case 2 when using the misplaced tiles heuristic. Although these increases are not as substantial, they are present nonetheless.

We can also then conclude that for the two search algorithms evaluated which do not produce the optimal plan, the plan length was consistently longer and more variable when using the Misplaced tiles heuristic.

Therefore a when considering a practical application where speed is the main concern the GBFS (Manhattan) is the best option while still maintaining consistent and slightly longer plan lengths than the optimal plan. While when the optimal plan is integral to our solution, A\* (Manhattan) performs the best while still only placing roughly around third place compared against all other search algorithms.



**Part 2: PDDL Implementation**

PDDL provides the ability of separating a planning problem into a planning domain and problem instances. This is great for representing planning problems due to generalization of a solution plan and also for easy problem instance creation. PDDL supports STRIPS (Stanford Research Institute Problem Solver) which provides a number of useful components to plan out a solution. These include:

* Predicates – facts which describe the environment of the space.
* Actions -
  + Preconditions – some actions have certain requirements to execute.
  + Add/delete effects – changing of facts when going through an action.
* Initial state - set of facts which hold for the beginning of the planning process.
* Goal state – provides the facts required to reach final state.

PDDL is useful especially in our case since our scenario is fully deterministic, meaning all tiles and positions on the grid are visible. For our project, we must present a domain about a 3 by 3 grid with numbers from 1 to 8 randomly allocated and 1 empty position.

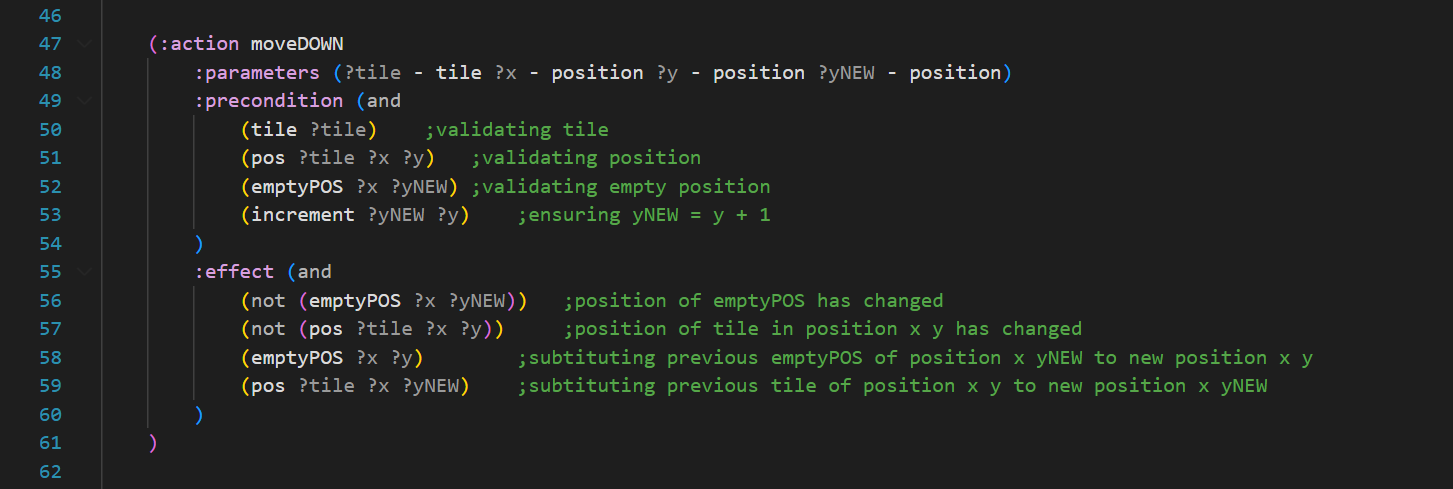
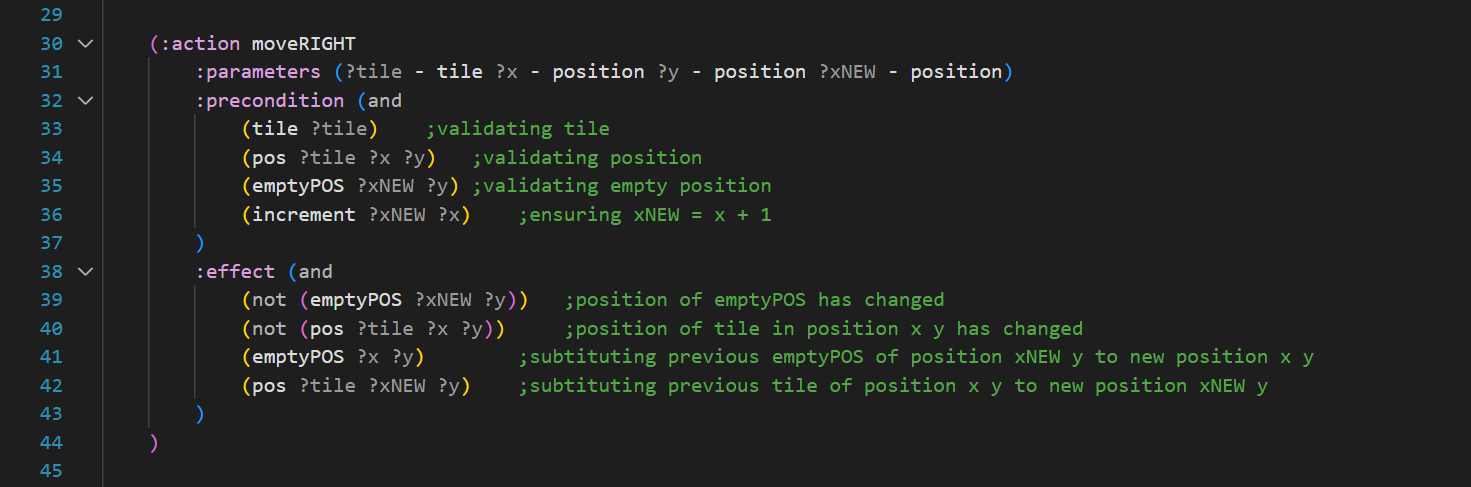
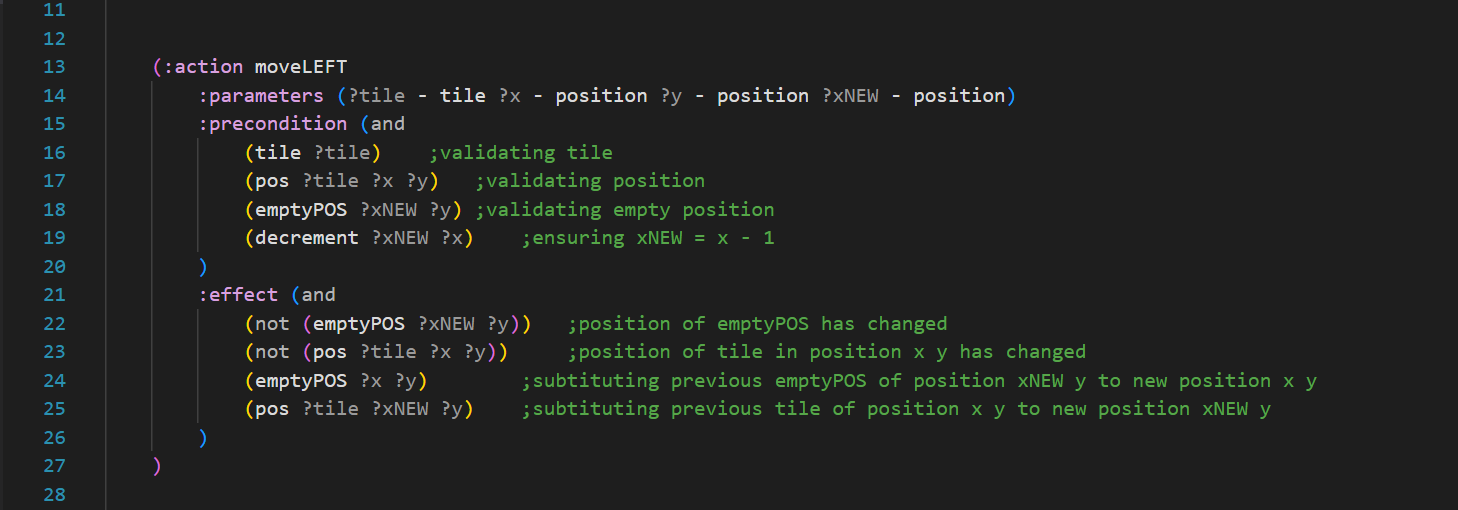
Domain file: Automated.pddl



The above represents the fundamentals aspects of the domain file;

* Requirements:
  + strips to define actions and preconditions
  + typing to introduce object types
  + equality for logical comparing
* Types:
  + Tile – represents a puzzle tile of the grid
  + Position – represents the coordinates on the grid of a specific tile or empty space.
* Predicates:
  + Tile – validates a tile variable
  + Pos – describes the coordinates of a tile
  + emptyPos - describes the current position of where there is no tile (empty space/ 0)
  + increment – to represent a +1 addition to a previous position
  + decrement – to represent a -1 subtraction to a previous position

After gathering the fundamentals of the domain, then we create the possible actions. Each action will have its own unique requirements(preconditions) and own effects.



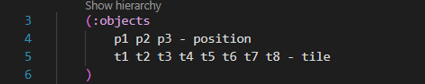
Actions:

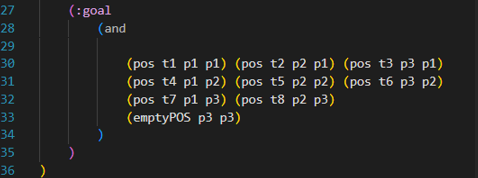
* moveLEFT:
  + preconditions – checks and validates all called parameters and ensures a decrement between variables.
  + effects – states that the previous positions are no longer valid and swaps the empty position with the tile in position x y.
* moveRIGHT:
  + preconditions – checks and validates all called parameters and ensures an increment between variables.
  + effects – states that the previous positions are no longer valid and swaps the empty position with the tile in position x y.
* moveUP:
  + preconditions – checks and validates all called parameters and ensures a decrement between variables.
  + effects – states that the previous positions are no longer valid and swaps the empty position with the tile in position x y.
* moveDOWN:
  + preconditions – checks and validates all called parameters and ensures an increment between variables.
  + effects – states that the previous positions are no longer valid and swaps the empty position with the tile in position x y.

Problem Definitions:

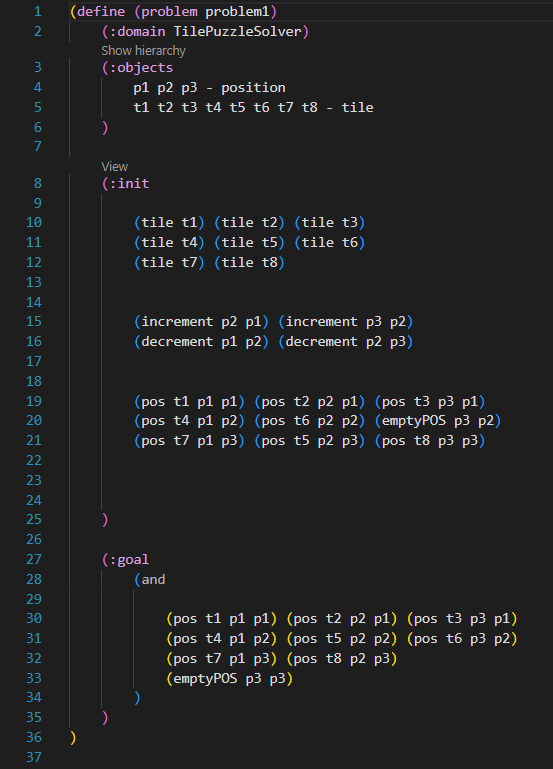
Our program ranges 6 different problem, 1-4 are solved within 2 to 3 moves while 5 and 6 are longer and have deeper state searching to solve.

Our problems are defined as following:

* object creation – initializing variables and setting their object type.
* Init – sets all puzzle tiles from t1 to t8, sets possible increments and decrements (1->2->3) and (3->2->1). Then the initial state of the puzzle board is created. Usually this is the only part of the file which differs from one problem file to another, since the object type declaration, variable creation, and goal state are kept the same.
* Goal – represents the state of facts which must be true for the algorithm to stop (Final state). The goal state is the same for every PDDL problem;



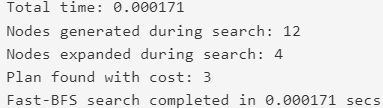
Problem 1: problem1.pddl



Initial state of problem 1: [[1,2,3,][4,6,0][7,5,8]]



Solution of problem 1:



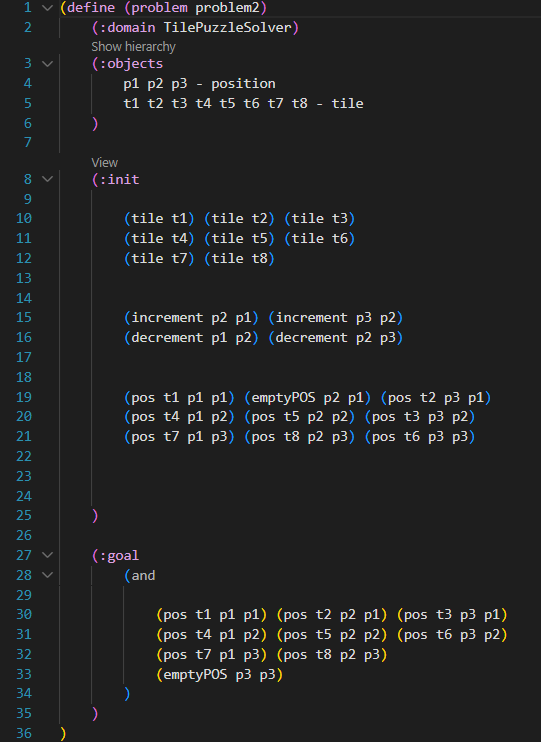
Plan length: 3

States expanded: 4

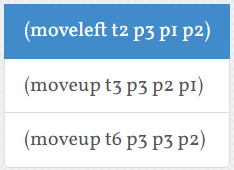
States discovered: 12

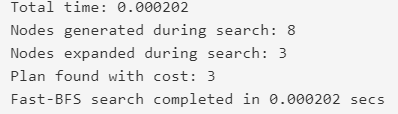
Time taken: 0.000171

Problem 2: problem2.pddl



Initial state of problem 2: [[1,0,2][4,5,3][7,8,6]]

Solution for problem 2:



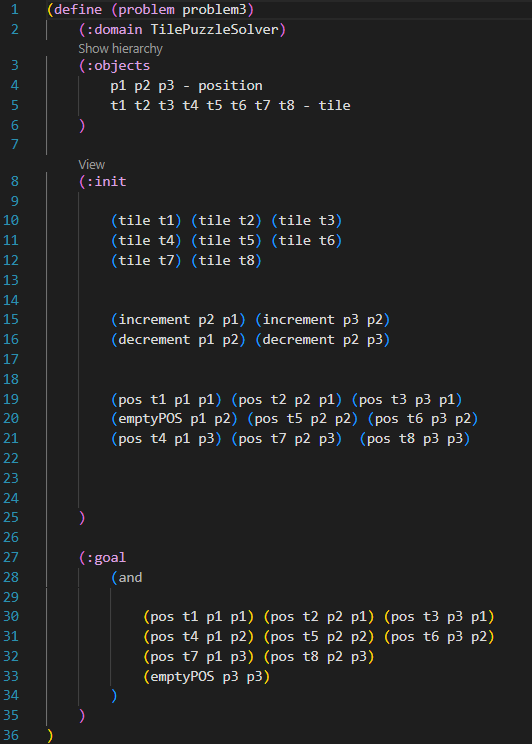
Plan length: 3

States expanded: 3

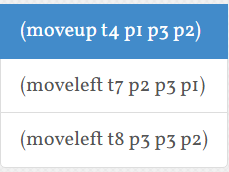
States discovered: 8

Time taken: 0.000202

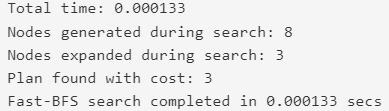
Problem 3: problem3.pddl



Initial state of problem 3: [[1,2,3][0,5,6][4,7,8]]



Solution for problem 3:



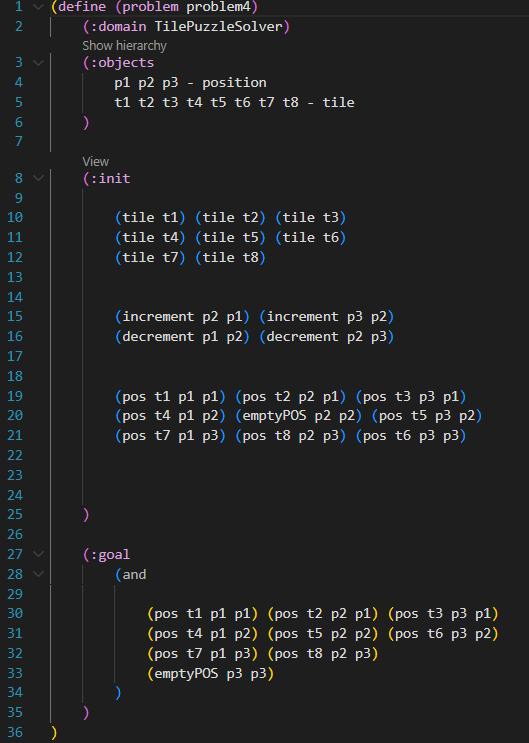
Plan length: 3

States expanded: 3

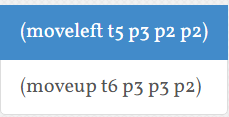
States discovered: 8

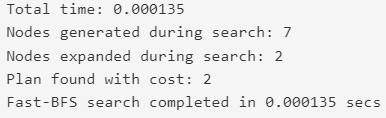
Time taken: 0.000133

Problem 4: problem4.pddl



Initial state for problem 4: [[1,2,3][4,0,5][7,8,6]]

Solution for problem 4:



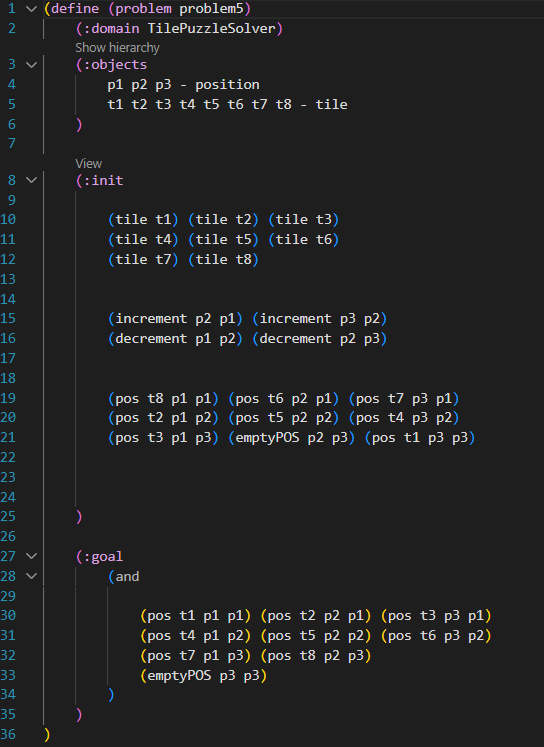
Plan length: 2

States expanded: 2

States discovered: 7

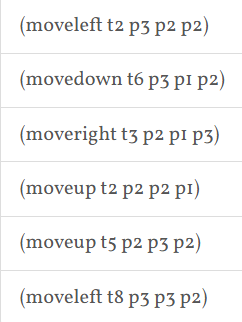
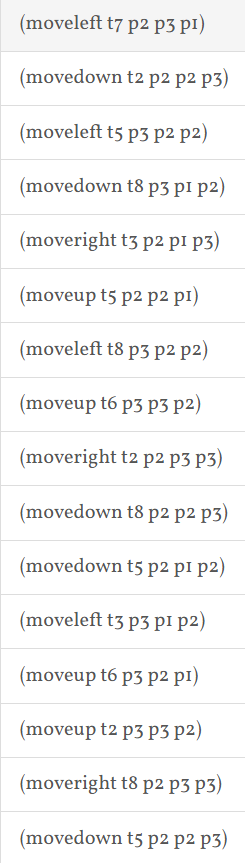
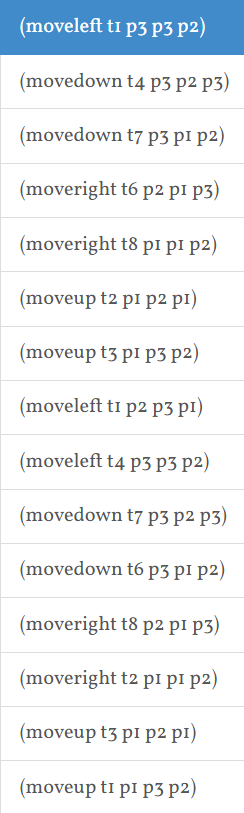
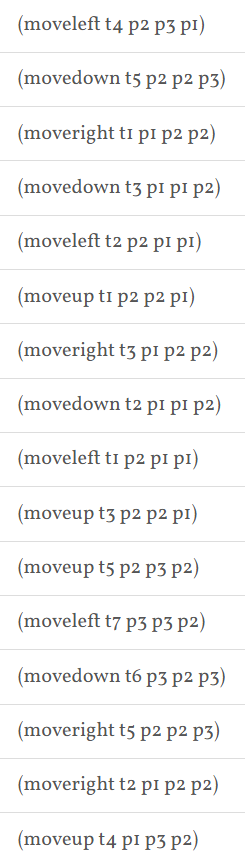
Time taken: 0.000135

Problem 5: problem5-hard.pddl



Initial state for problem 5: [[8,6,7][2,5,4][3,0,1]]

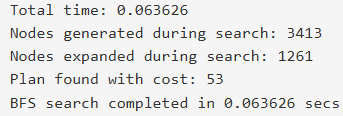
Solution for problem 5:



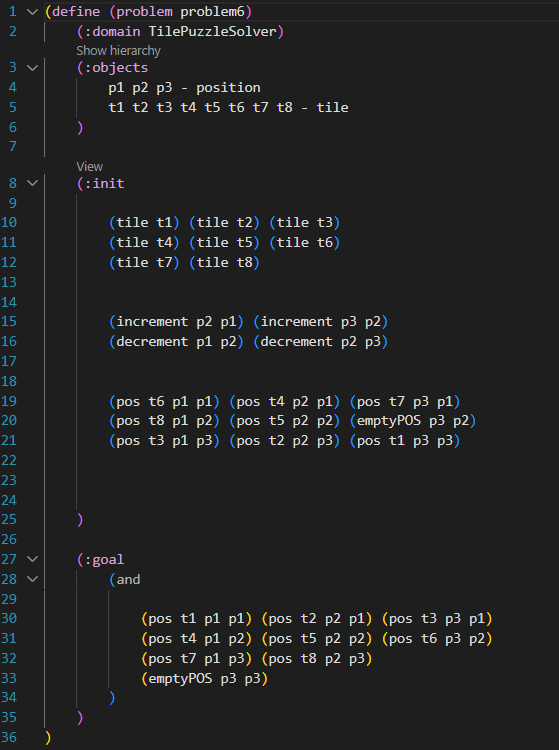
Plan length: 53

States expanded: 1261

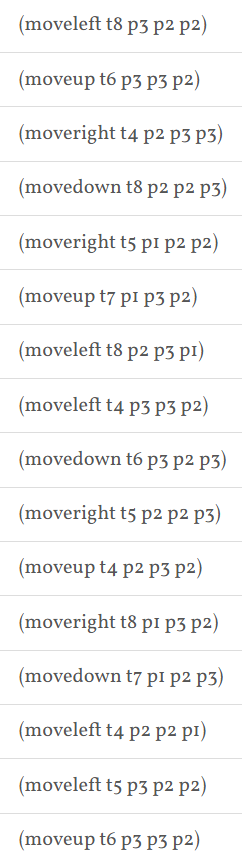
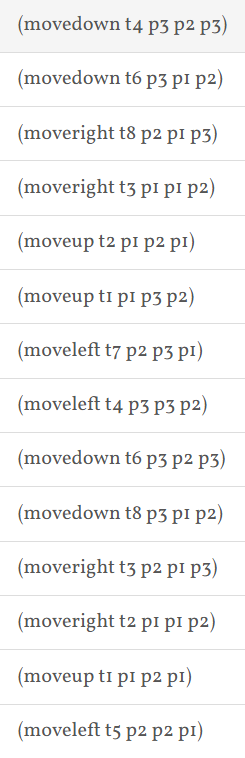
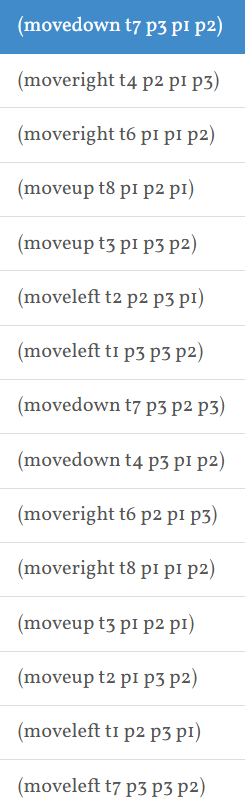
States discovered: 3413

Time taken: 0.063626

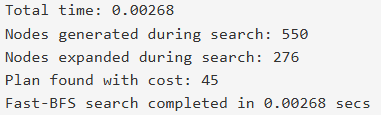
Problem 6: problem6-hard.pddl



Initial state for problem 6: [[6,4,7][8,5,0][3,2,1]]



Solution for problem 6:



Plan length: 45

States expanded: 276

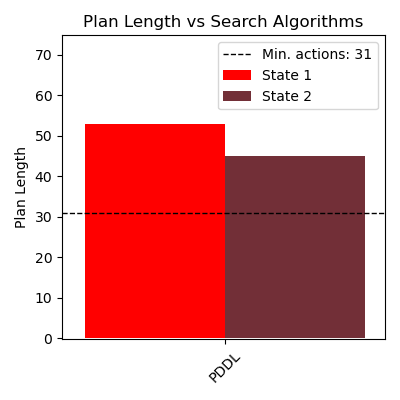
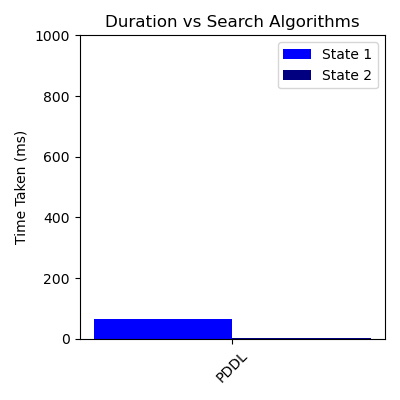
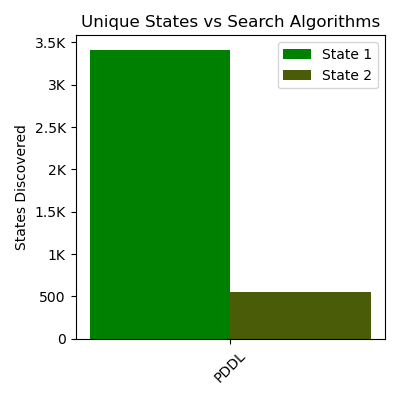
States discovered: 550

Time taken: 0.00268

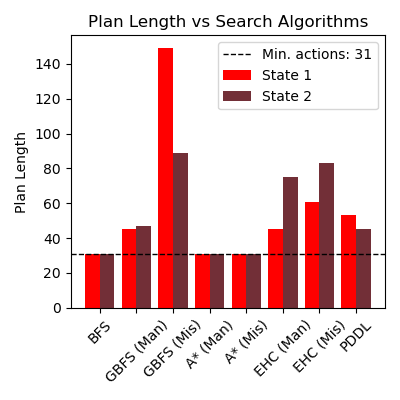
Evaluation of Part 2

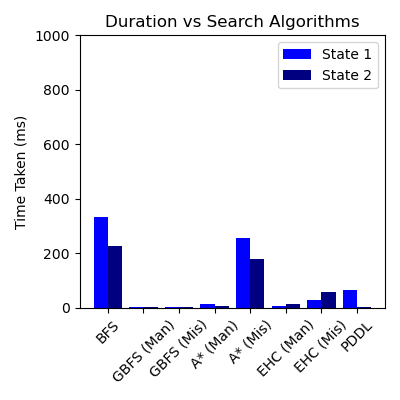
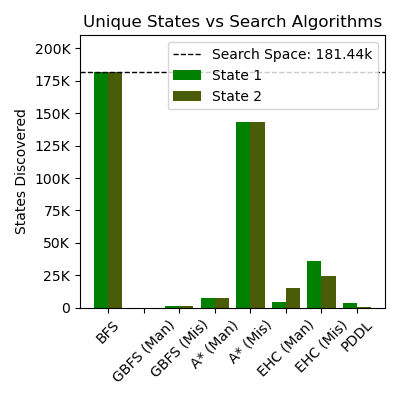
State 1 represents problem5-hard

State 2 represents problem6-hard



Comparing results of Part 1 and Part 2:





References:

Distribution of Work: