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

25605H

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.



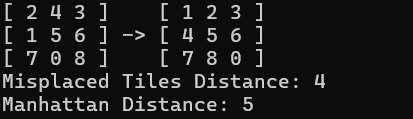
**Distance Functions Implementation**

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.

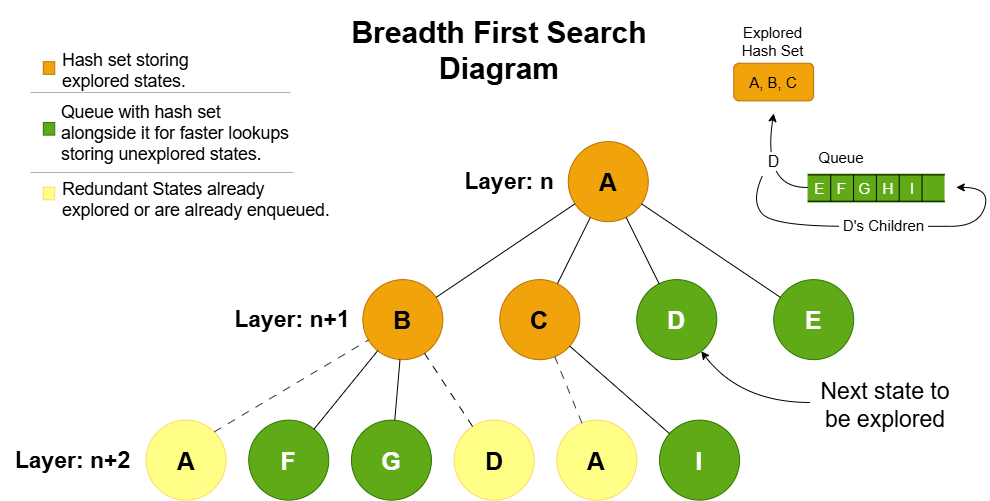
- In all implementations where a queue is used to store unexplored states a hash set is maintained alongside the queue to speed up lookup times as the operation direct-address-search in a direct address table give us O(1) time as discussed in [1], instead of O(n). This doubles the size needed to store enqueued states but since layers grow exponentially with depth and lookups happen multiple times per state dequeue, it is a trade-off which drastically improves search times.

- In all implementations where the explored states were maintained the same logic about direct-address tables from [1] the operation direct-address-insert also only takes O(1) time and so another hash set is maintained in order to store the explored states.

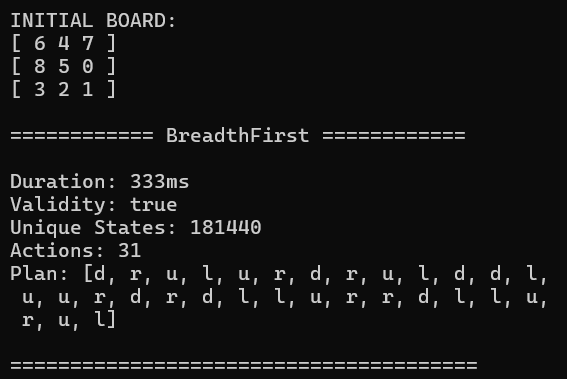
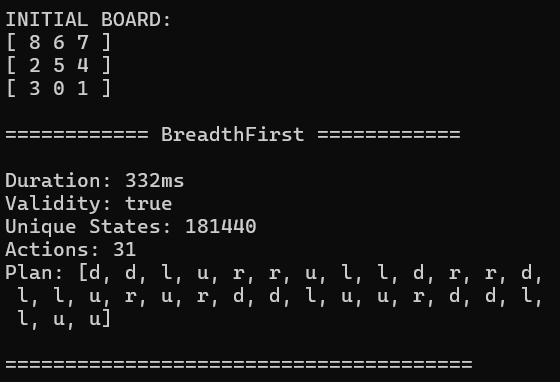
- States having boards identical to the boards of other states are considered redundant. Maintaining hash sets that store both explored and unexplored states, helps to lookup generated children states before enqueuing to avoid going in circles.

- 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**

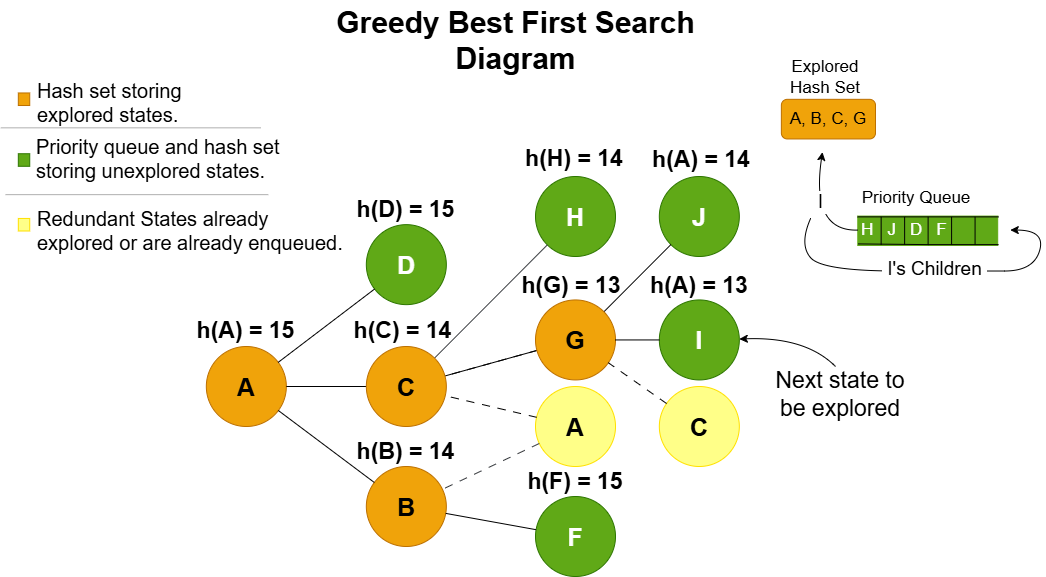
This breadth first (BFS) implementation follows the implementation described in [2, pp 81, Sec. 3.4.1] but differs slightly in that as stated earlier, a hash set is maintained alongside the queue. 

Plan Output:

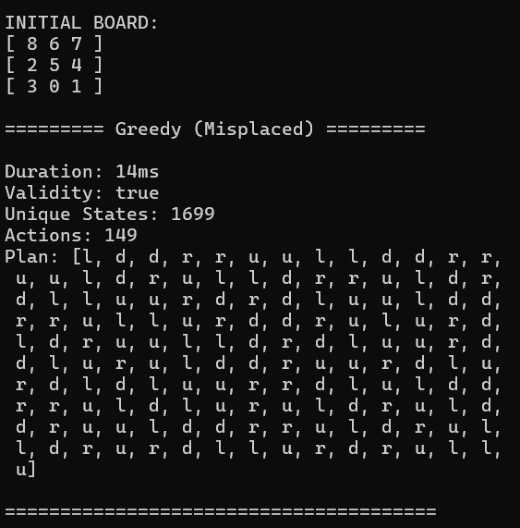
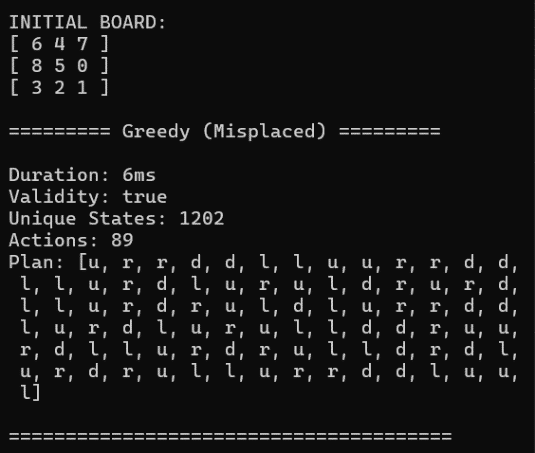
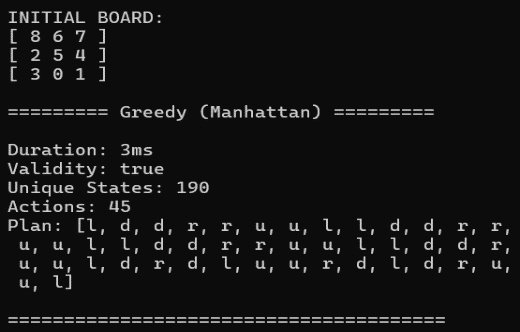


**Greedy Best First Search Implementation**

This implementation is based on [2, pp 92, Sec. 3.5.1] and is similar to that of the BFS implementation, but differs in that it maintains a priority queue to store unexplored states based on the hCost instead.



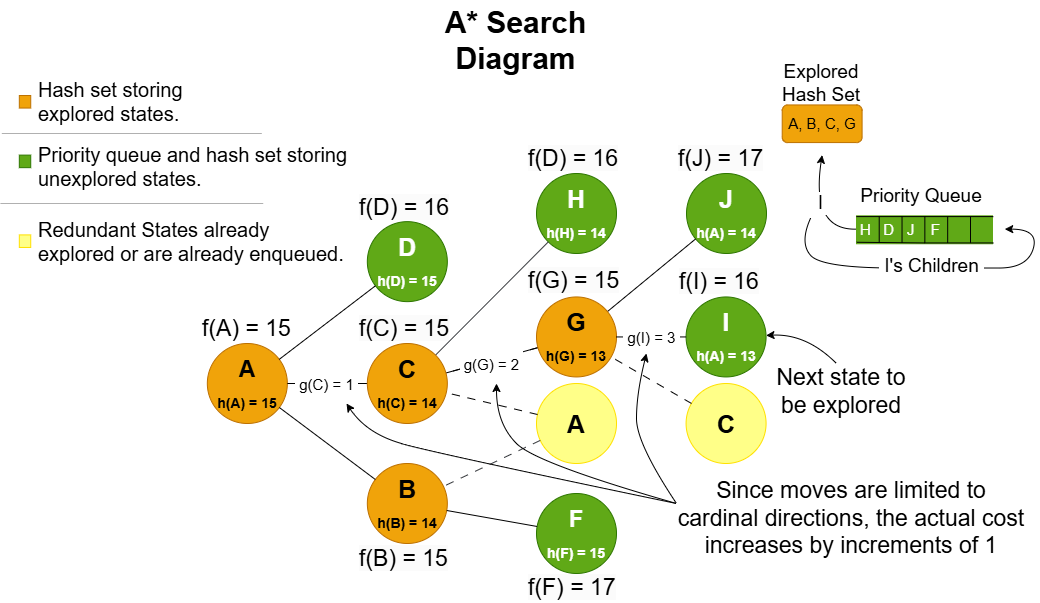
Plan Output:



**A\* Search Implementation**

This implementation is based on [2, pp 93, Sec. 3.5.2] and is the same to that of the GBFS implementation, but differs in that the priority queue storing unexplored states enqueues according to fCost, then hCost.

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.

****

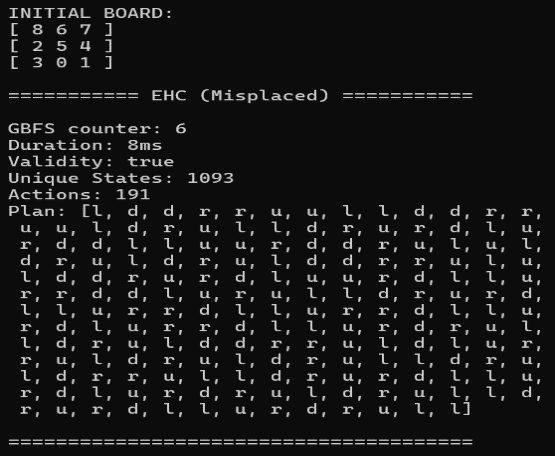
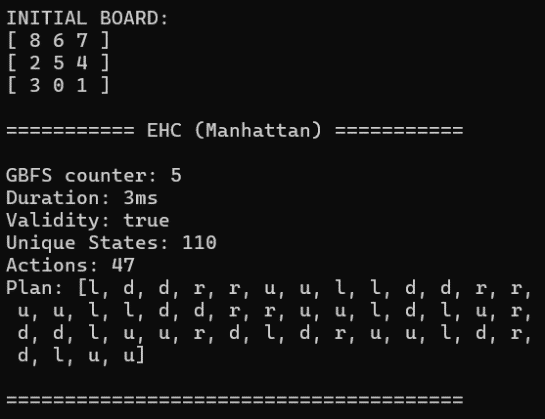
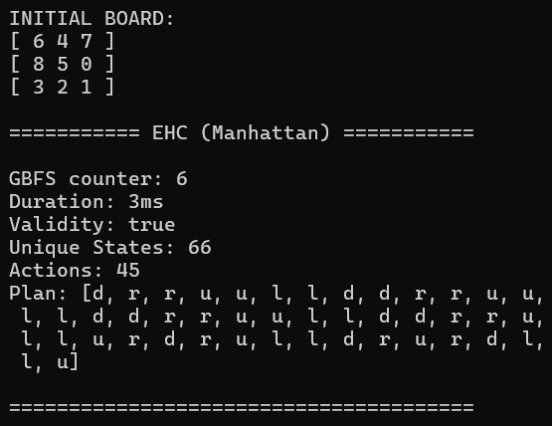
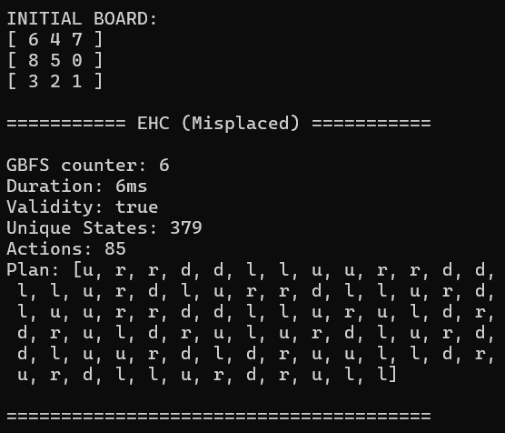
Plan Output:



**Enforced Hill Climb Search Implementation**

This implementation follows along the implementation described in [1] but uses greedy best first search (GBFS) instead of breadth first search to keep unique states low as BFS generates an unnecessarily exhaustive. It also differs in that it does not maintain an open list but simply tracks the best state and updates when a better state is found. If a state does not have any children which are heuristically closer to the goal state, it performs a GBFS search until a state with a better heuristic is found. This is repeated until no improvement is made by either the hill climb or the fall back GBFS.

Plan Output:



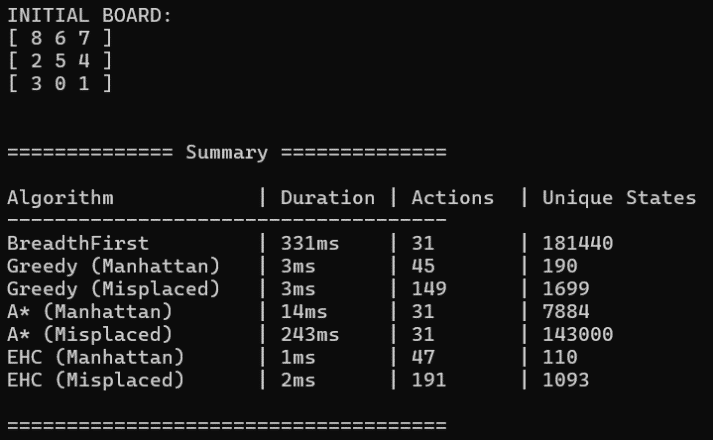
**Menu**

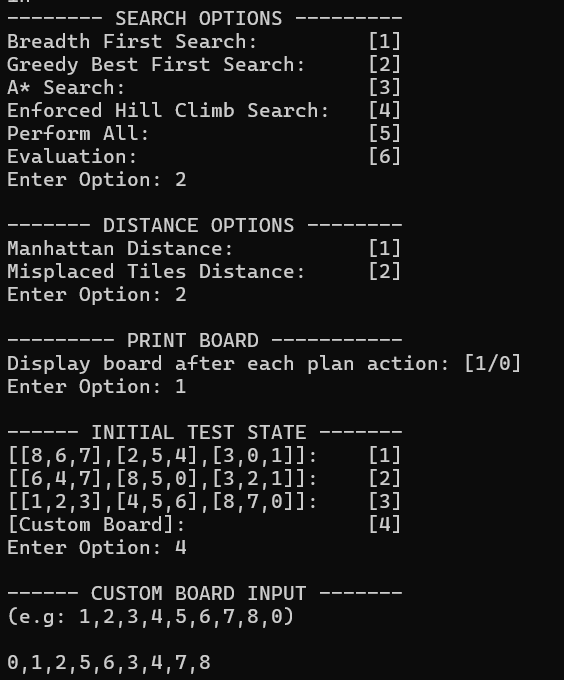
Search options include the four search algorithms, an option to perform all search algorithms with an additional, and an option to compute the evaluation metrics for the two initial test states. It also displays a summary and writes the summery to a csv file for further graph representation using a python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

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.

****Menu example: Summary example:



**Evaluation**

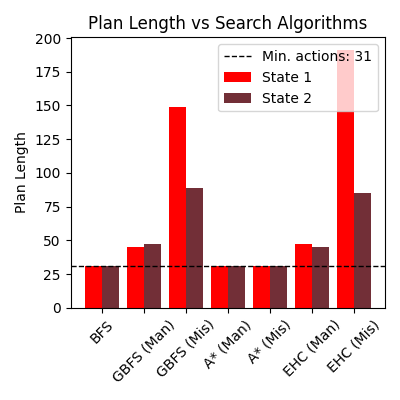
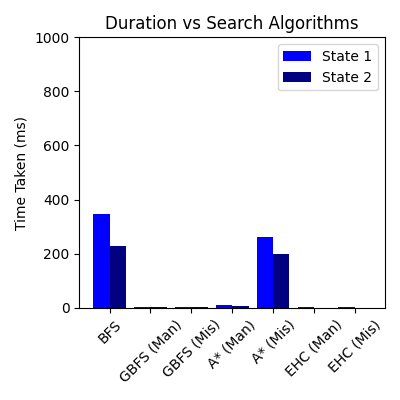
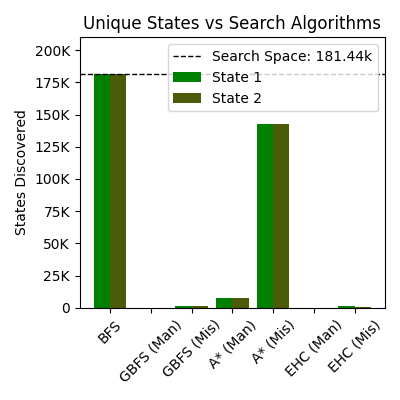
A useful figure to have is the size of the state space. We calculate it by considering all possible arrangements of and dividing by due the parity constraint since a tile swap affects the parity (even or odd nature) of the number of inversions (pairs of tiles that are in the wrong order relative to the goal state).

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. Specifically, 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, which also indicates that the longest minimum distance from a state to its goal states is 31 actions.  
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) “Characteristically, EHC search consists of prolonged periods of exhaustive search, bridged by relatively quick periods of heuristic descent.” [3].



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.

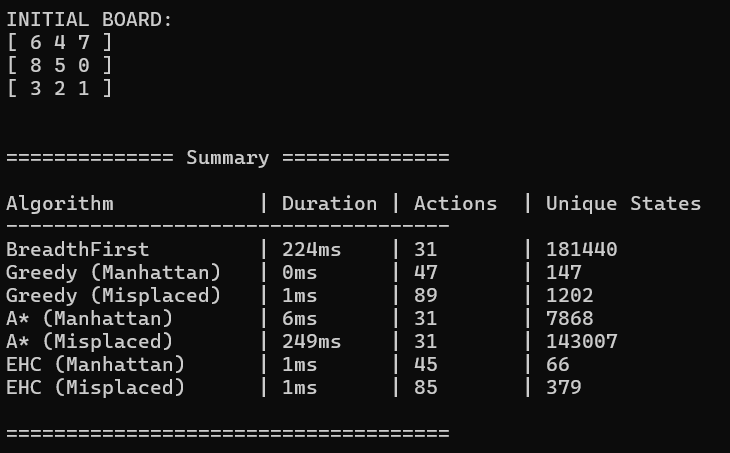
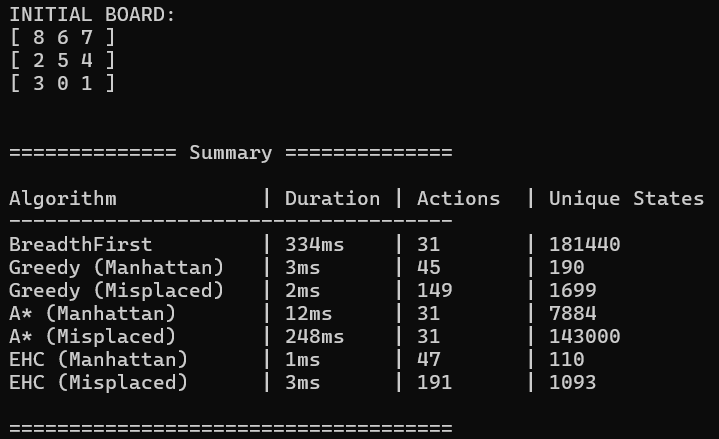
While GBFS and EHC do not produce the optimal plan they consistently perform well in terms of speed. EHC as expected demonstrates the fastest performance generating the lowest number of unique states. Although both EHC (Manhattan) and GBFS (Manhattan) perform very well in maintaining low search spaces. 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) and similarly EHC (misplaced) performed the worst with substantially longer plans than their Manhattan counterpart. Additionally, we observed that the plans generated for the initial test states, using the Manhattan distance heuristic generated a difference in plan length of 2 for both algorithms. While when evaluating with the misplaced tiles heuristic, a significant jump in the difference in plan length was recorded.

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 concluded by assessing the difference between the two initial test states.

Therefore when considering a practical application where speed is the main concern the EHC (Manhattan) is the best option while still maintaining consistent and slightly longer plan lengths than the optimal plan. This aligns with [4] in that EHC requires significantly less space than A\*. If then the optimal plan is integral to our solution, A\* (Manhattan) performs the best while still only placing third place when compared against all other search algorithms.

****

**Part 2: PDDL Implementation**

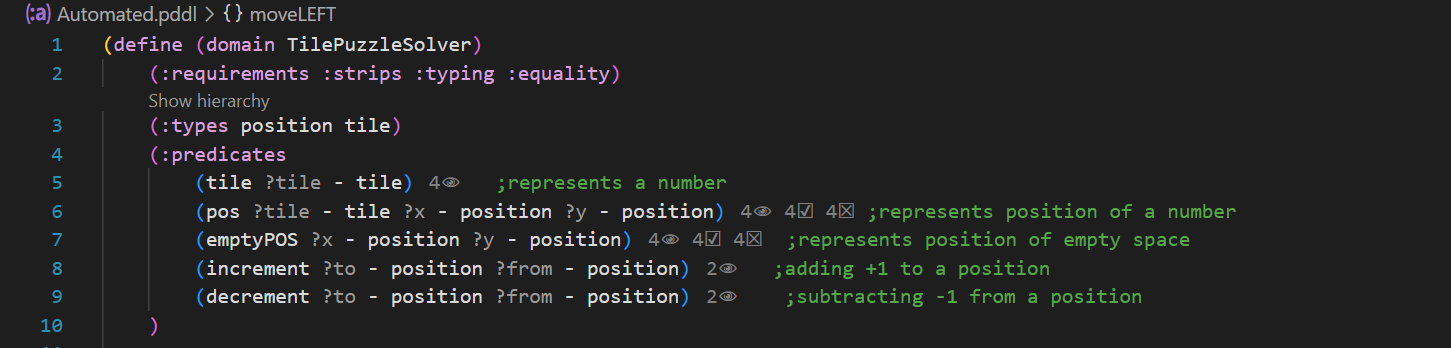
PDDL provides the ability of separating a planning problem into a planning domain and problem instances. This is great for representing planning problems due to generalization of a solution plan and also for easy problem instance creation. PDDL supports STRIPS (Stanford Research Institute Problem Solver) 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. (add-after-delete semantics in STRIPS).
* Initial state - set of facts which hold for the beginning of the planning process.
* Goal state – provides the facts required to reach final state.

[5]

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.

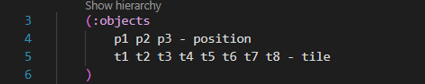
Actions:

* 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:

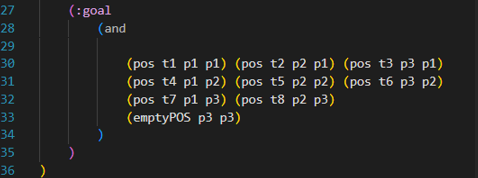
Our program ranges 6 different problem, 1-4 are solved within 2 to 3 moves while 5 and 6 are longer and have deeper state searching to solve.

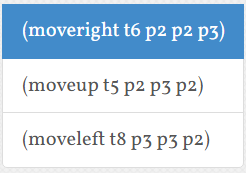
Our problems are defined as following:

* 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.



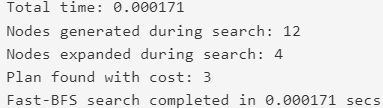
* Goal – represents the state of facts which must be true for the algorithm to stop (Final state). The goal state is the same for every PDDL problem;



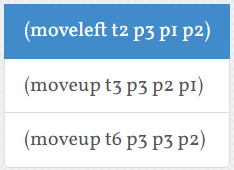
Initial state of problem 1: Solution of problem 1:

[1,2,3]  
[4,6,0]  
[7,5,8]

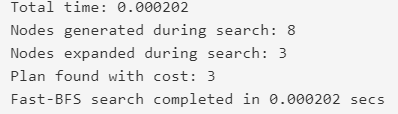
Plan output of problem 1:



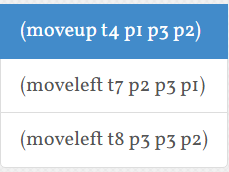
Initial state of problem 2: Solution for problem 2:

[1,0,2]  
[4,5,3]  
[7,8,6]

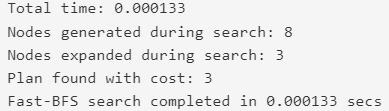
Plan output of problem 2:

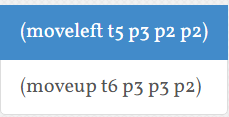


Initial state of problem 3: Solution for problem 3:

[1,2,3]  
[0,5,6]  
[4,7,8]

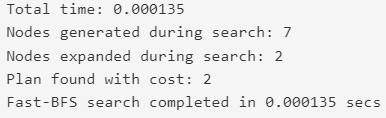
Plan output of problem 3:



Initial state for problem 4: Solution for problem 4: 

[1,2,3]  
[4,0,5]  
[7,8,6]

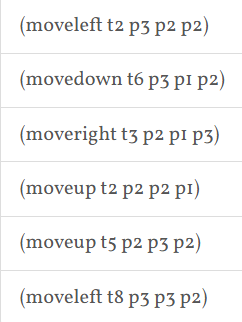
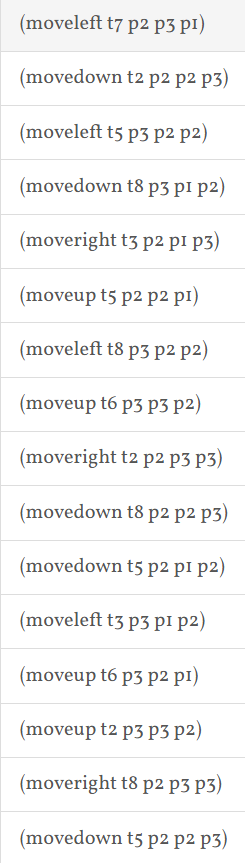
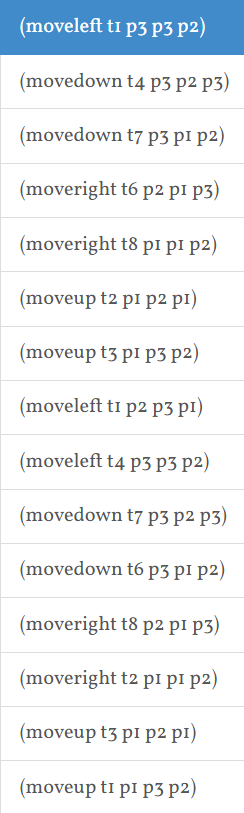
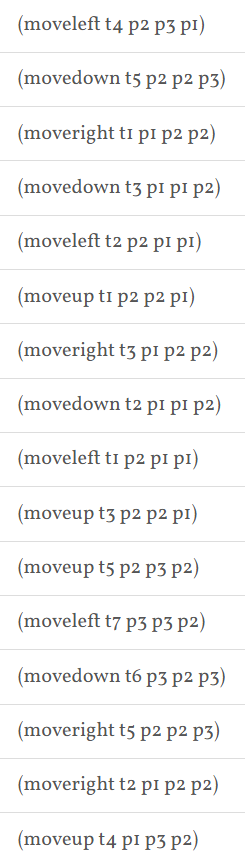
Plan output of problem 4:



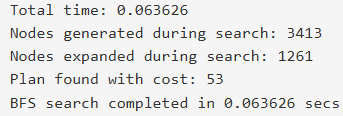
Initial state for problem 5:

[8,6,7]  
[2,5,4]  
[3,0,1]

Solution for problem 5:

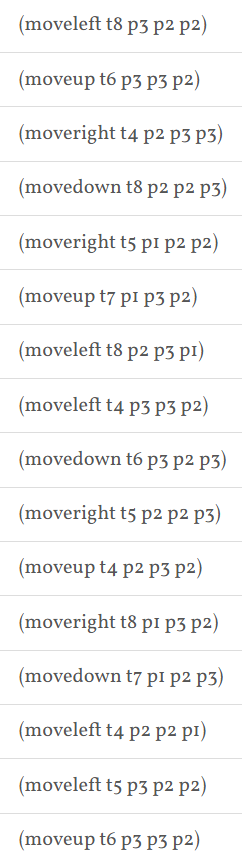
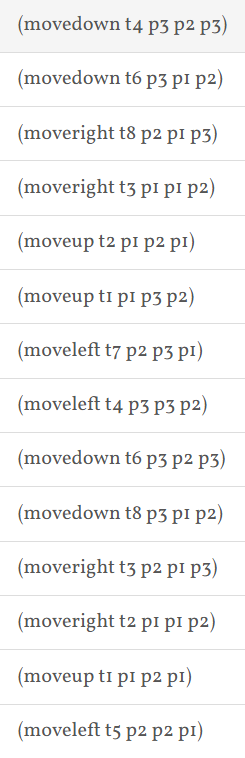
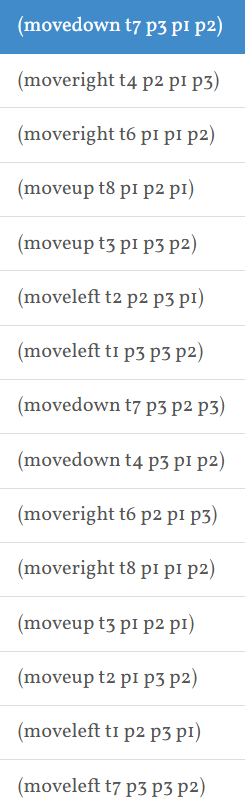


Plan output for problem 5:

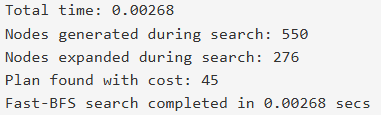


Initial state for problem 6:

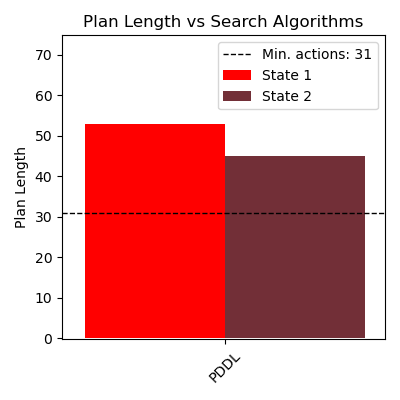
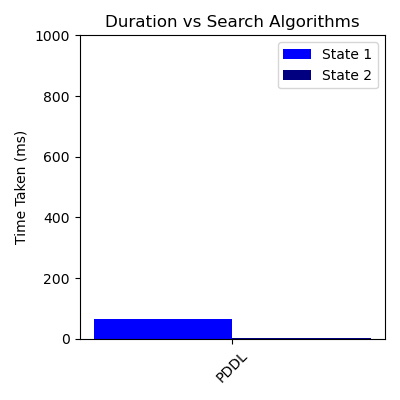
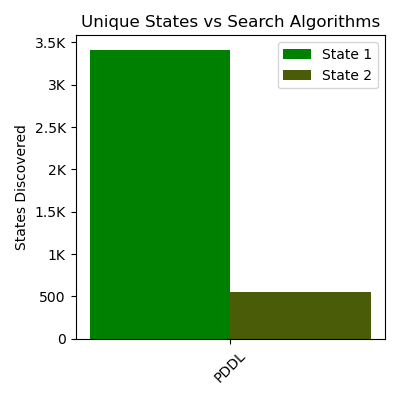
[6,4,7]  
[8,5,0]  
[3,2,1]



Solution for problem 6:



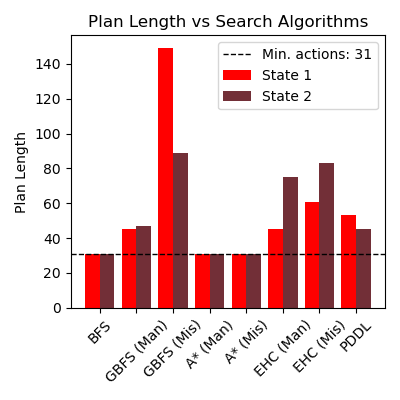
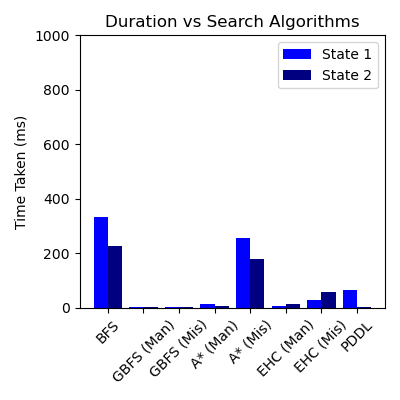
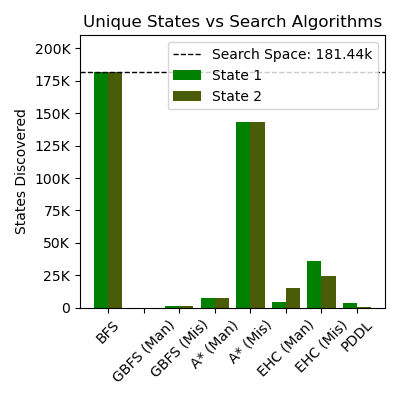
Plan output for problem 6:



Evaluation

State 1 represents problem5-hard

State 2 represents problem6-hard



Comparing results of Part 1 and Part 2:

To compare the results of the domain specific algorithms (part 1) and the domain independent planner (part 2), we can use the above visualizations and the respective outputs like analysis of numeric outputs which includes plan length, time taken, and number of states generated.

Plan Length Comparison

The length of the plan derived from the PDDL planner - domain-independent - in Part 2 for states corresponding to problem instances, State 1 and State 2, tends to be longer compared to domain-specific algorithms reviewed in Part 1. For instance:

BFS and GBFS (Manhattan heuristic) from Part 1 result in near-optimal plan lengths, close to the minimum number of actions of 31.

The general solution provided by the PDDL planner is equally effective, but a bit longer; this underlines at once the efficiency of domain-specific solvers, engineered- thanks to certain heuristics and problem knowledge-to create shorter paths.

Unique States Generated

The number of unique states explored by the PDDL planner in Part 2 is significantly lower compared to BFS in Part 1 but still higher than GBFS and A\*. From the visualizations:

BFS, being exhaustive in nature, has explored a huge number of states, which amounts to more than 175,000 states for State 1.

The heuristics and generally broader planning strategies mean that the PDDL planner benefits from finding fewer states to still find solutions.

A\* and EHC, Manhattan heuristic included, attain similar or low state explorations compared to PDDL in Part 1, showcasing the efficiency in heuristic-based search strategies as it contracts the search space.

Time Efficiency

The computation time of the solutions via PDDL planner is competitive but generally higher compared to the more specialized search algorithms developed in Part 1, such as GBFS and A\* with domain-specific heuristics. Based on the "Duration vs Search Algorithms" graph:

BFS in Part 1 generates the most states but completes in a relatively short time for simpler problems; it does not perform well in harder instances.

PDDL's performance comes between BFS and heuristic-driven methods, reflecting its balance of exploration and plan optimality.

Visualization Insights

From the visualizations of plan length, states discovered, and time taken, some underlying differences are underlined:

Domain-specific algorithms like BFS, A\*, EHC in Part 1 are strong for optimization, reflected in shorter plan lengths and focused exploration.

PDDL Part 2 provides a general solution that can solve various problem domains but loses on efficiency and conciseness.

References:

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to algorithms, 4th ed. Cambridge, Massachusett: The Mit Press, 2022, p. 273, Accessed: Dec. 30, 2024. [Online]. Available: https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf

[2] S. Russel and P. Norvig, Artificial intelligence: A Modern approach, 3rd ed. Prentice Hall, 2021, pp. 81, 92-93. Accessed: Dec. 30, 2024. [Online]. Available: https://people.engr.tamu.edu/guni/csce421/files/AI\_Russell\_Norvig.pdf

[3] A. Coles and A. Smith, "Marvin: A Heuristic Search Planner with Online Macro-Action Learning", Specific section: "Enforced Hill Climbing Search", Carnegie Mellon University, Accessed: Dec. 30, 2024. [Online]. Available: https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume28/coles07a-html/node5.html

[4] J. Bajada, Fundamentals of Automated Planning - Lecture 7 – Planning Graph Hueristics, Department of Artificial Intelligence, University of Malta, 2024, slide 27.

[5] J. Bajada, Fundamentals of Automated Planning - Lecture 4 - PDDL, Department of Artificial Intelligence, University of Malta, 2024, slide 11.

Distribution of Work: