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 includes a children method to generate all valid successor states. Successors are created if the empty tile has an in-bounds neighbour, swapping the tiles with the Swap helper function, and the child state's parent, and the action performed are tracked.



**Result Implementation**

The Result class contains the implementation of the retrace step and plan validation step.

//Runtime figures – unique states, duration

Duration of search is calculated within the method and does not include time taken to validate plan, generate results and print them out.

Unique states generated 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 in order to describe the plan in the correct direction.



Plan Validation Step - validity

After the plan is in the correct order, a plan validation step is executed to validate if a plan correctly reached the goal board. This is done by peeking the initial board and simulating all the moves in order and comparing the final result.



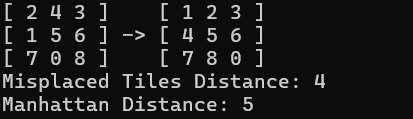
**Distance Functions Implementation**

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

Manhattan Distance: A lookup table is pre-computed, mapping tile values to their destination indices. For each tile in the start state, the table is used to find its destination index. The 2D positions are then calculated using modulus for x-coordinates and division for y-coordinates, then the sum of the absolute differences of their x and y coordinates are calculated to get the total Manhattan distance.

Misplaced Tiles: The tile values are compared at each index incrementing the cost if the values don’t match.

Testing using Figure 2, gives expected heuristic values



**Search Algorithms Implementation**

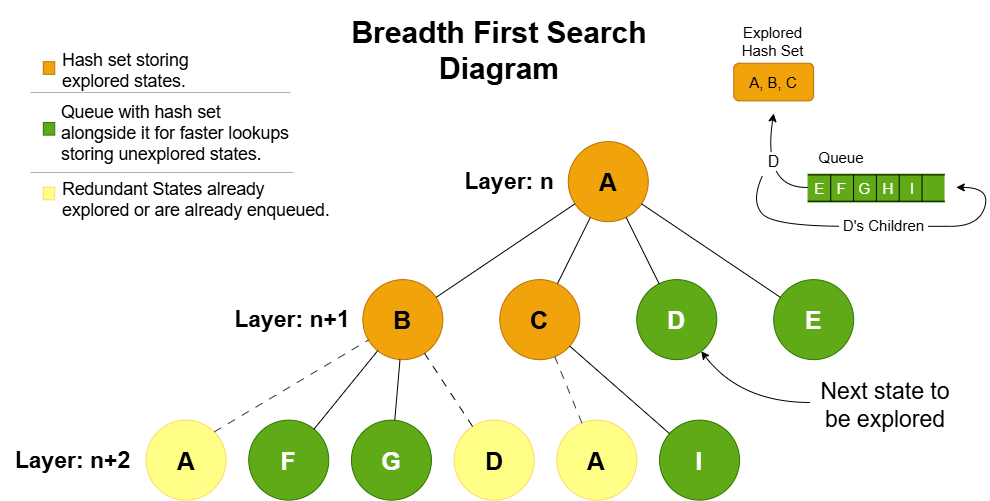
- In all implementations a queue or some variation of it is used to store unexplored states. Alongside this is a hash set used to speed up lookup times as the operation direct-address-search in a direct address table give us O(1) time as discussed in [1], instead of O(n). This doubles the size needed to store unexplored 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 a hash set is maintained, storing explored states, applying the same logic about direct-address tables from [1], since the operation direct-address-insert also only takes O(1) time.

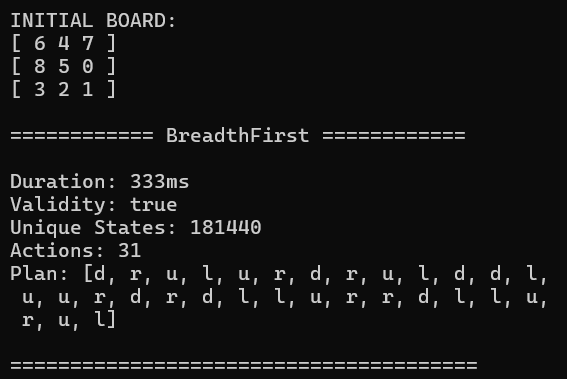
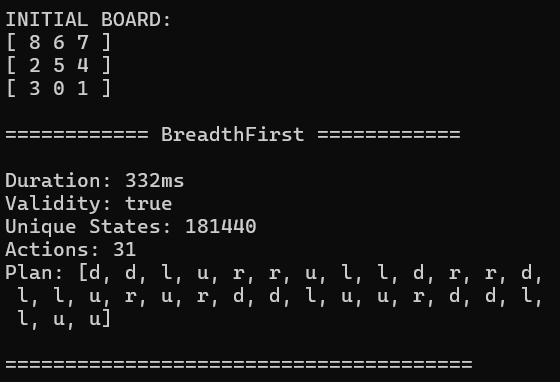
- States having boards identical to the boards of other states are considered redundant. Maintaining hash sets that store both explored and unexplored states, helps to lookup generated child states before enqueuing them to avoid going in circles.

- A search will run until either the goal state is found or the entire search space is explored.

**Breadth First Search Implementation**

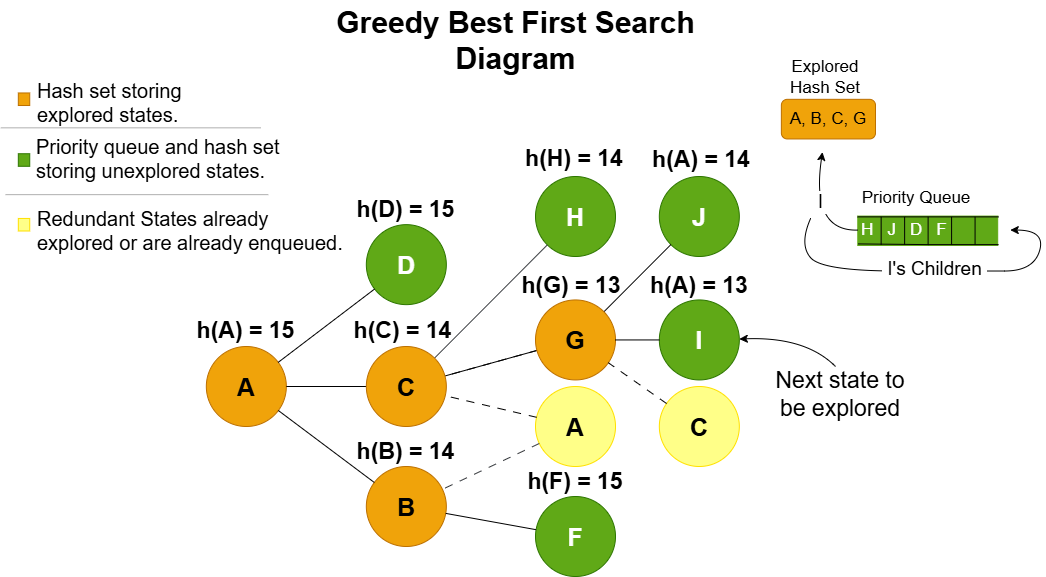
This breadth first (BFS) implementation follows the implementation described in [2, pp 81, Sec. 3.4.1] but differs slightly in that as stated earlier, a hash set is maintained alongside the queue. 

Plan Output:

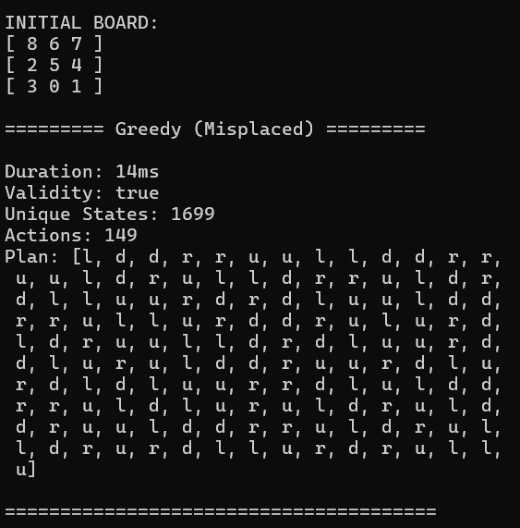
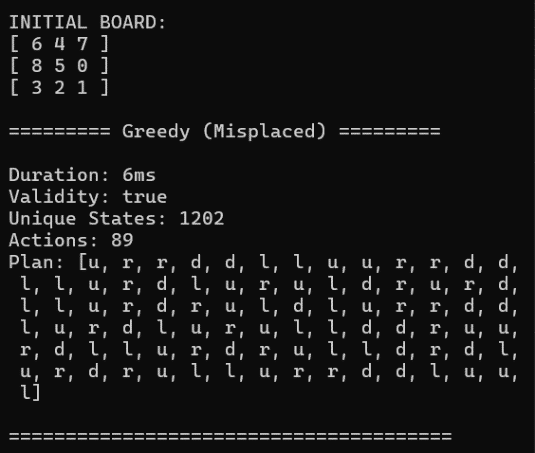
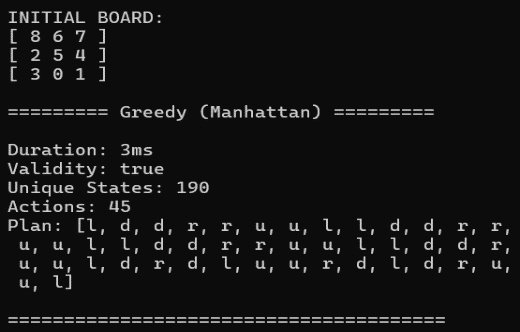


**Greedy Best First Search Implementation**

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.



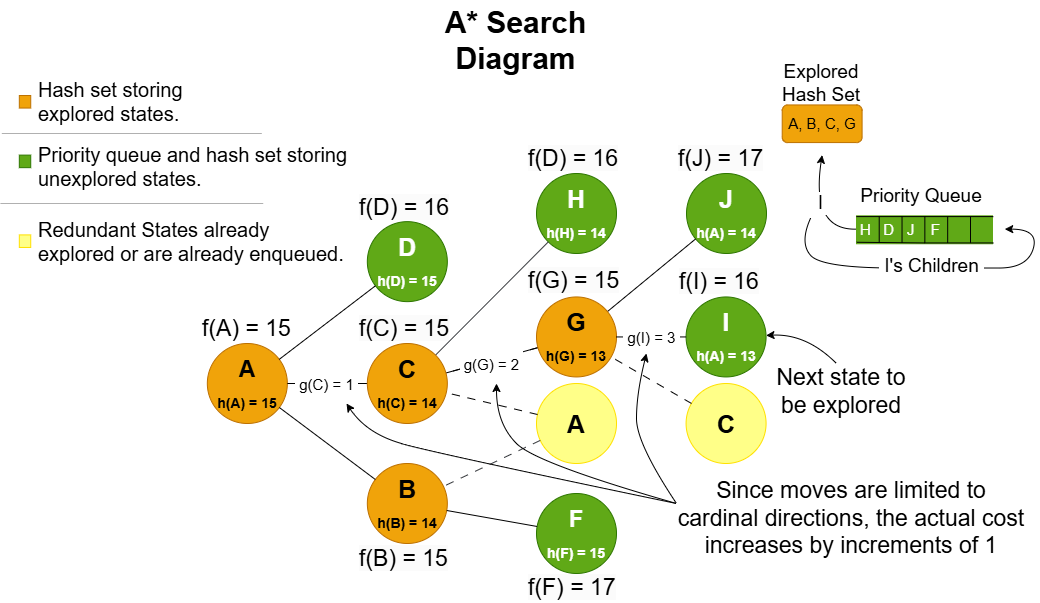
Plan Output:



**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.

****

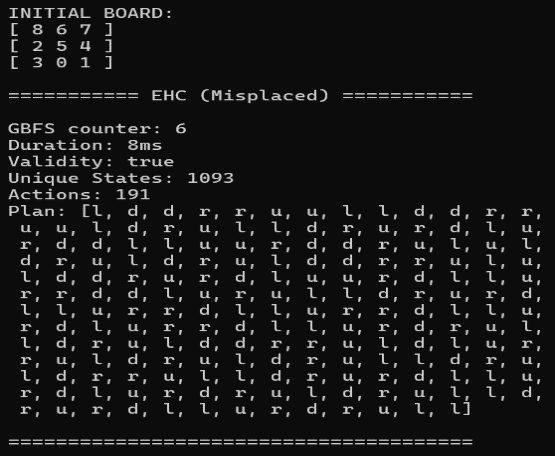
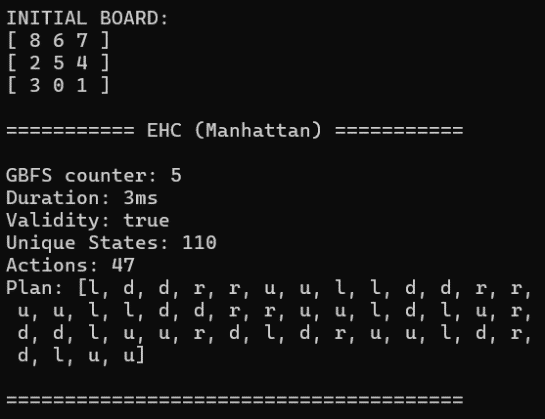
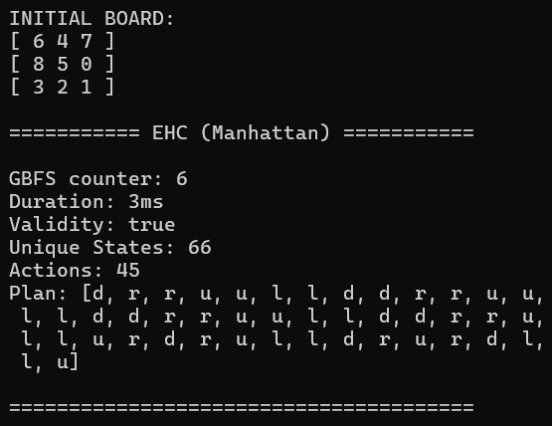
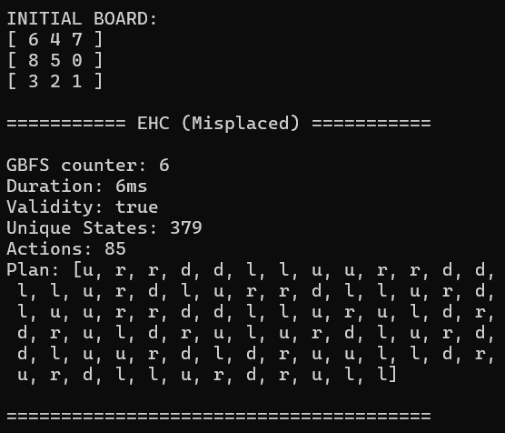
Plan Output:



**Enforced Hill Climb Search Implementation**

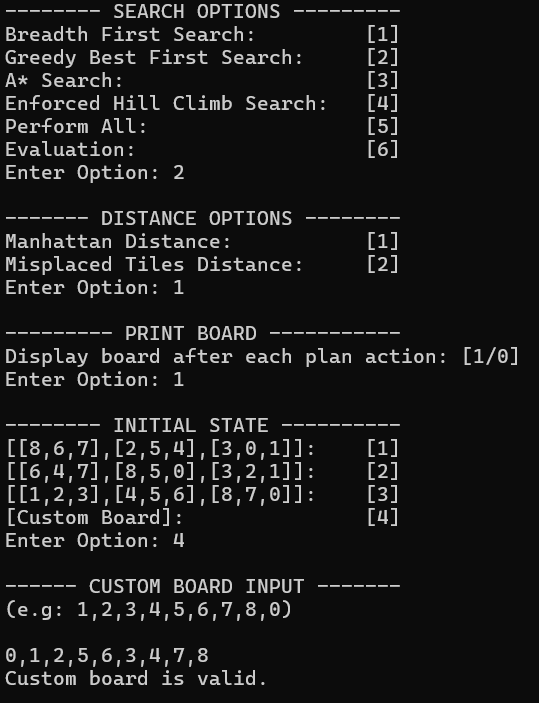
This implementation follows along the implementation described in [3] but uses greedy best first search (GBFS) instead of breadth first search to keep unique states low as BFS generates an unnecessarily exhaustive fall back. It also differs in that it does not maintain an open list but simply tracks the best state and updates when a heuristically 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:

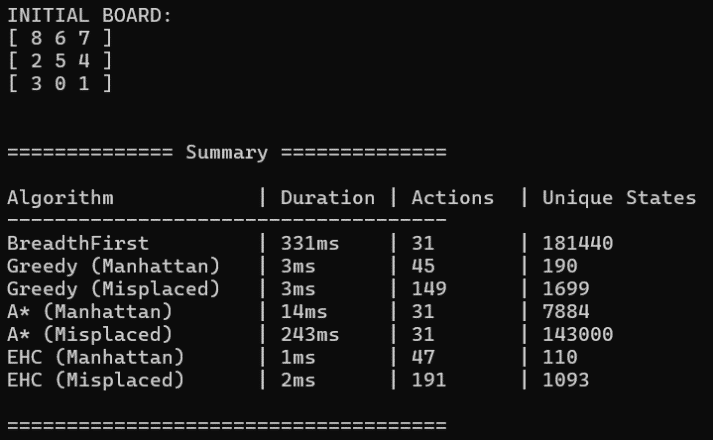
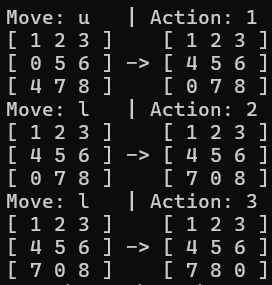


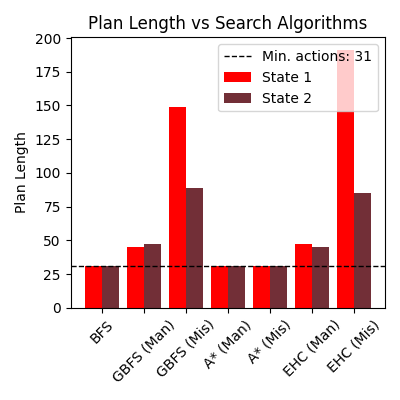
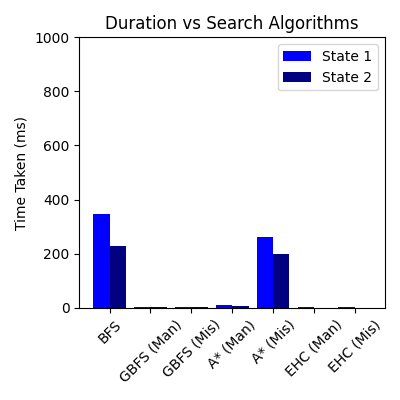
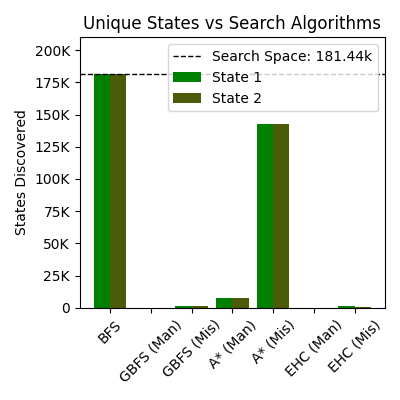
**Use Case examples**

Menu example:



The 3rd default state is an example of an unsolvable state.

****Print Board example: Summary example:



**Evaluation**

We first calculate the size of the search space. This is computed as, accounting for all possible tile arrangements and the parity constraint. The parity arises because each tile swap alters the inversion count's parity (even or odd nature), which dictates whether a state is solvable.

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 and on different machines.

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. Specifically for both test cases, the entire state space was generated, highlighting BFS’s poor scalability.

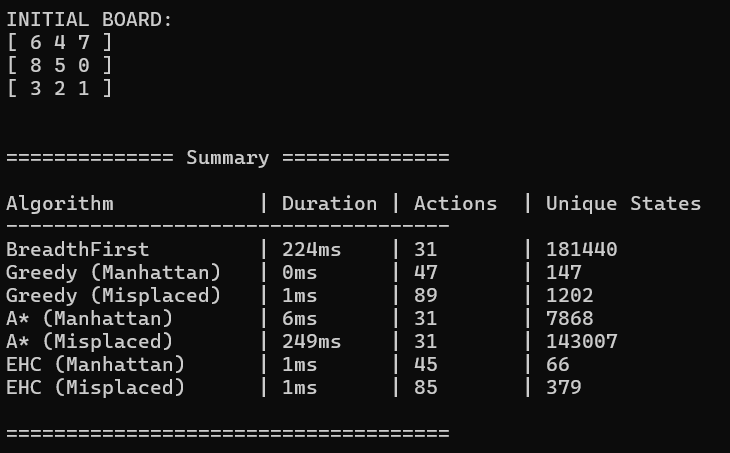
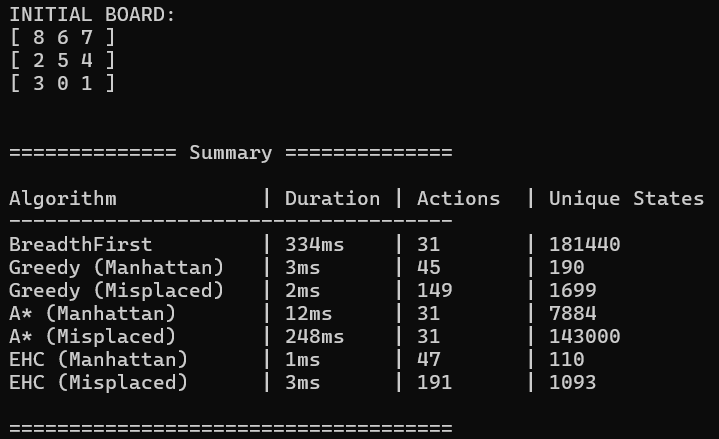
While GBFS and EHC do not produce the optimal plan they consistently perform well in terms of speed. EHC as expected demonstrates the fastest performance generating the lowest number of unique states. Although both EHC (Manhattan) and GBFS (Manhattan) perform very well in maintaining low search spaces. 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.

In practical applications where minimizing state space is critical, EHC (Manhattan) demonstrated the best performance, maintaining efficiency while producing plan lengths which were consistent and only slightly longer than the optimal. This aligns with [4] in that EHC requires significantly less space than A\*. Since “A\* search is both complete and optimal” [2, pp 93, Sec. 3.5.2], if the optimal plan is integral to our solution, A\* (Manhattan) greatly outperformed A\* (misplaced), and proved far more scalable than BFS. Although an important consideration for A\* is the heuristic’s quality, as the impact misplaced tiles heuristic had is worst seen in A\*.

****

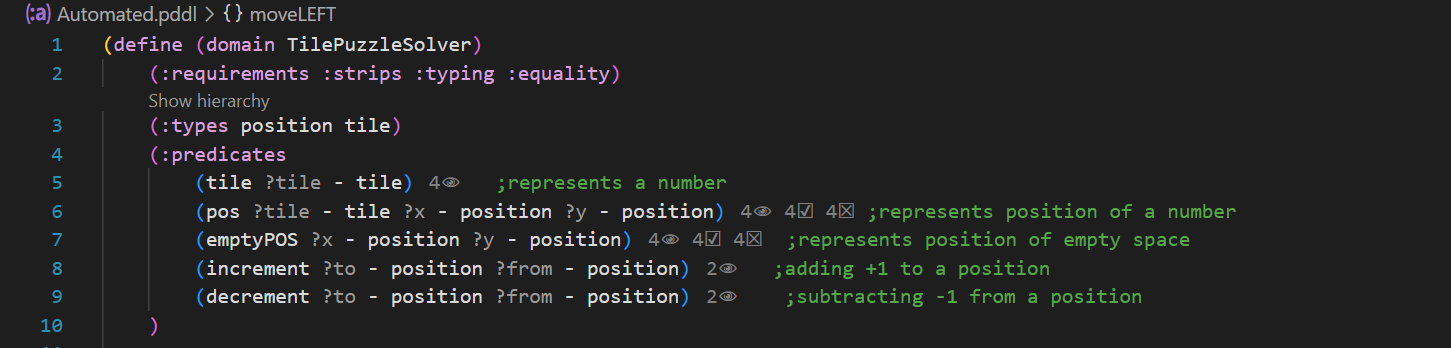
**Part 2: PDDL Implementation**

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 as suggested in [5]).
* 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.

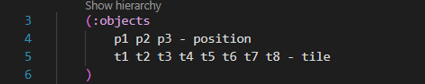
Actions:

* 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:

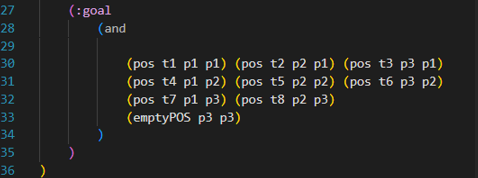
Our program ranges 6 different problem, 1-4 are solved within 2 to 3 moves while 5 and 6 are longer and have deeper state searching to solve.

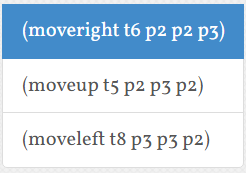
Our problems are defined as following:

* 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.



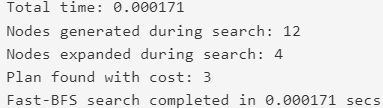
* Goal – represents the state of facts which must be true for the algorithm to stop (Final state). The goal state is the same for every PDDL problem;

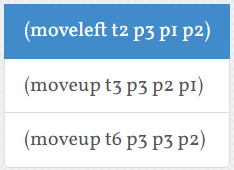


Initial state of problem 1: Solution of problem 1:

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

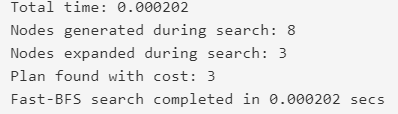
Plan output of problem 1:



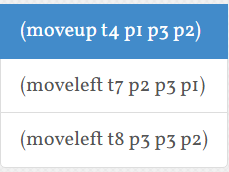
Initial state of problem 2: Solution for problem 2:

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

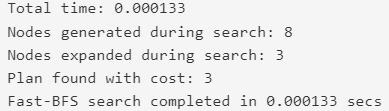
Plan output of problem 2:

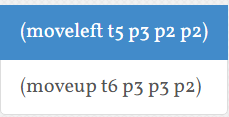


Initial state of problem 3: Solution for problem 3:

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

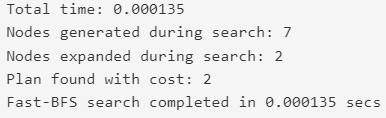
Plan output of problem 3:



Initial state for problem 4: Solution for problem 4:

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

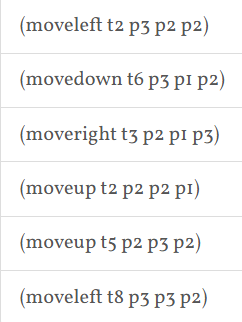
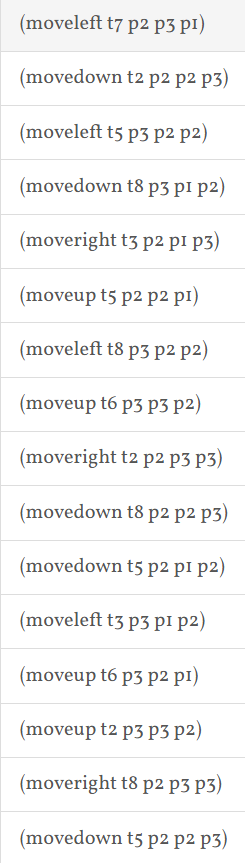
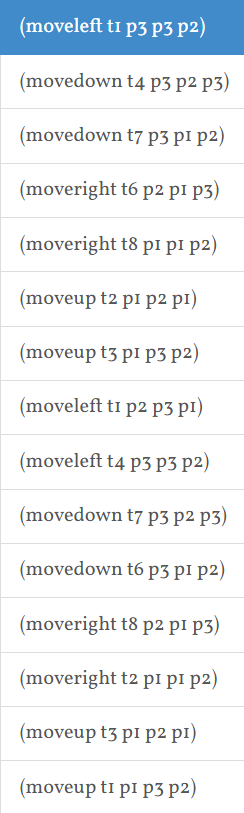
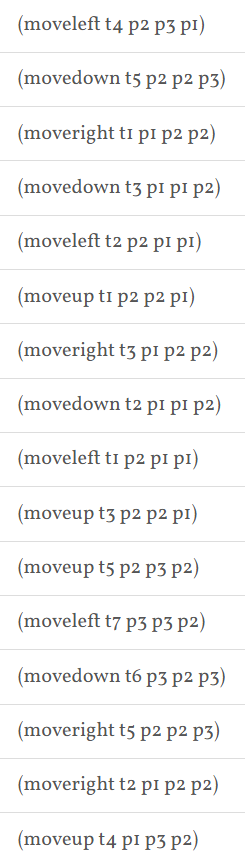
Plan output of problem 4:

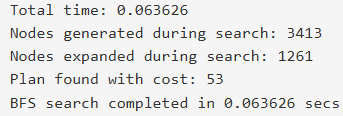


Initial state for problem 5:

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

Solution for problem 5:

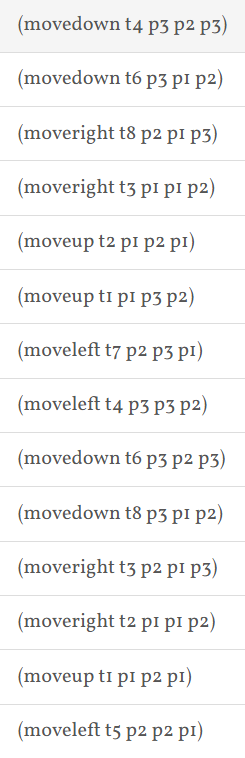
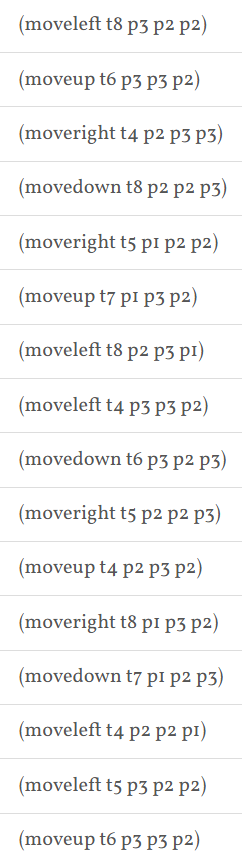
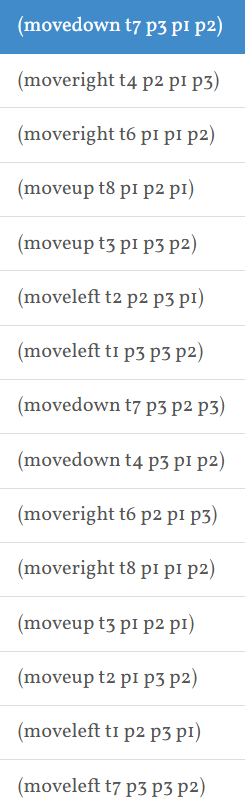


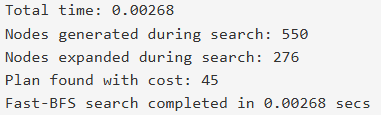
Plan output for problem 5:

Initial state for problem 6:

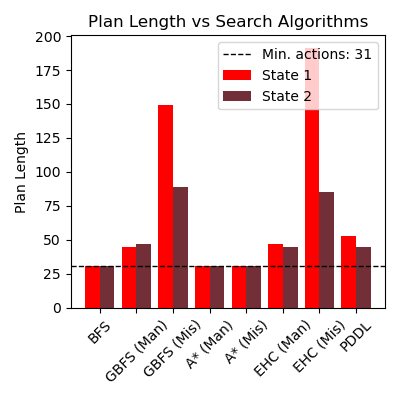
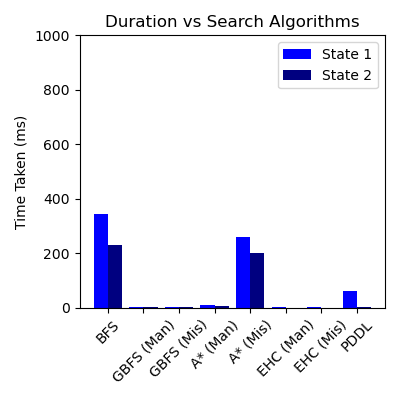
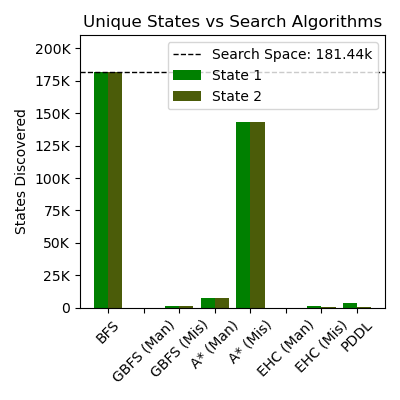
[6,4,7]  
[8,5,0]  
[3,2,1]

Solution for problem 6:



Plan output for problem 6:

1



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 domain independent planner does not generate an optimal plan as do the BFS and A\* search algorithms from part 1. Comparing therefore to the algorithms which also do not generate optimal plans 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, tended to be slightly longer compared to domain-specific algorithms which made use of the Manhattan distance heuristic, reviewed in Part 1. Though it still performed better than the poorly performing misplaced tiles heuristic.

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:

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:

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

The slightly longer general solution provided by the PDDL planner and slightly larger unique state count, hints at the advantage of domain-specific solvers, engineered- thanks to specific heuristics and problem knowledge-to create shorter plans in less time.

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:**

Matthew Farrugia: (50%)

* Part 1:
  + Implemented Greedy Best First search
  + Result.java
  + Plotting
* Part 2:
  + Implemented PDDL domain file
  + Implemented PDDL problem (1-6) files
  + Visualizations and evaluation of comparing of results

Matthew Farrugia’s Signature

Pawlu Spiteri: (50%)

* Part 1:
  + Implemented Breadth First search
  + Implemented A\* search
  + Implemented Enforced Hill Climb search
  + State.java
  + Main.java
  + DistanceFunctions.java

Pawlu Spiteri’s Signature