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 (gCost), heuristic estimate to goal state (hCost), and total estimated cost (fCost(), which is simply gCost + hCost).

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.

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.

**Result Implementation**

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.

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.

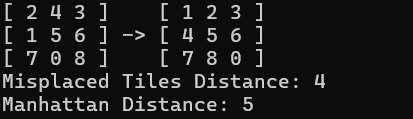
After plan is in correct order, a plan validation step is executed to validate if a plan correctly reached the goal. This is done by peeking the initial board and simulating all the moves in order and if the board matches the goal board then the plan is marked as valid.

**Distance Functions Implementation**

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**

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

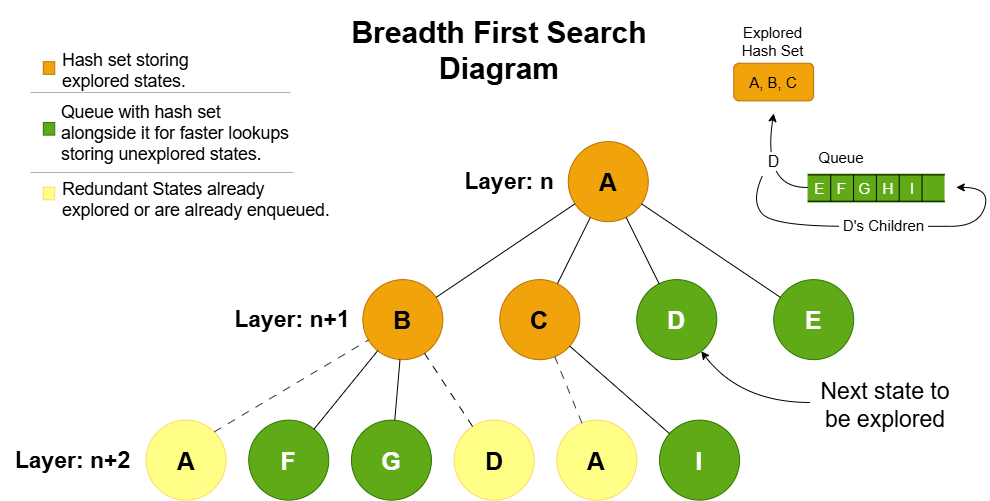
Initial test case 1:

Initial test case 2:

**Breadth First Search**

The Breadth first search implementation employs a queue to hold 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 hold explored states since its lookup and add times are both constant.

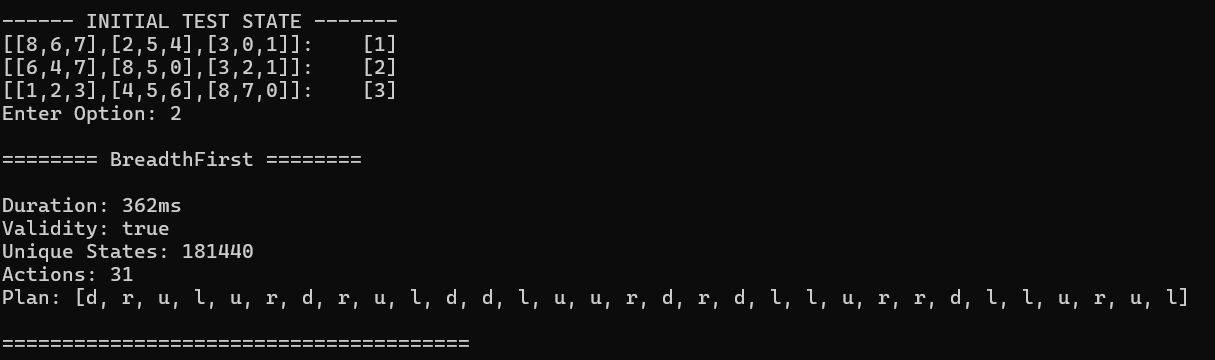
Redundant states are prevented by maintaining hash sets that store both explored and unexplored states. Before a generated child is checked against both sets



This 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.

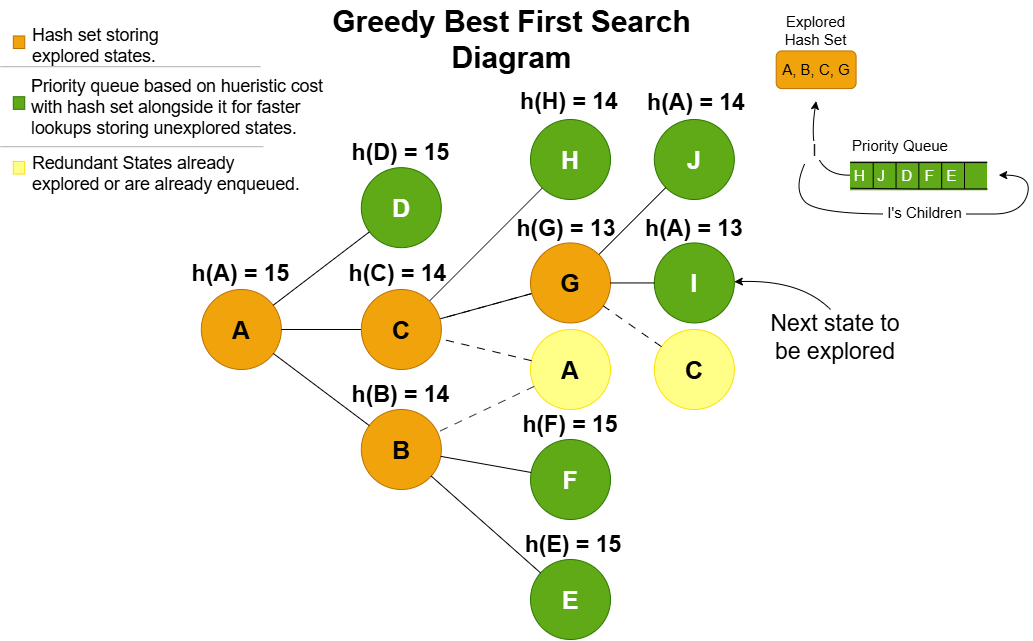
Furthermore if a goal state is at layer and layer exists, layer will be partially explored. The percentage of layer to be generated grows as distance m grows. ( being the distance from the initial state of a layer).

This makes the BFS impractical for problems with deeper state spaces or higher branching factors.





**Greedy Best First Search**

****

**A\* Search**

**Enforced Hill Climb Search**