Name: Amit Padaliya

Roll No: 2019H1030013G

Email: h20190013@goa.bits-pilani.ac.in

AI Assignment-1

Slide-1

The approach that worked most efficiently for me was A* + (Manhattan Distance + level no) as heuristic. In this approach the nodes are generated breath wise and for each child, here child means the after moving in left, right, up or down from a parent node.

The state of the board, consists of following things

Puzzle: A 1D list representing the board

Parent: The predecessor or the parent which generated the present state

Children: All the immediate successors obtained for state after moving up, down, left or right.

Pos: Position of blank tile in the puzzle

Cost: Cost includes the addition of Manhattan distance + level (at which this node was generated)

Level: Level at which the given node is generated.

A heap queue is maintained. It is min-heap, in which the criteria of ordering is the summation of the Manhattan-distance and the level at which the state is generated.

To keep track of visited nodes, a visited set is maintained. Purpose of set of make search time constant.

Algorithm

heap_queue.push(initial_state) // put initial state in the heap queue

While heap_queue is not empty:

item = heap_queuepop() // removes the first item from heap

Check **if** item is **goal** state or not

If item is goal state:

tracer = item // store this state to trace a path to its parent to get a list of moves later.

break

item.**move_left()** // find the item's child after moving 'blank(0)' in left direction and put in its **children** list

item.**move_right()** //find item's child after moving 'blank(0)' in right direction and put in its **children** list

item.**move_up()** //find item's child after moving 'blank(0)' in up direction and put in its **children** list

item.**move_down()** // find item's child after moving 'blank(0)' in down direction and put in its **children** list

for child in **children** of item:

if child **is not** visited yet

visited.add(child)

heap_queue.push(child)

Slide 2

The approach used always gives an optimal answer, because of the admissible heuristic used in A* algorithm, which is Manhattan Distance. It is admissible because it never overestimates the cost of reaching the goal state meaning that cost calculated at each node is the lowest required to reach the goal state. Because the way the Manhattan distance is calculated it such that it calculates, what the total distance of each tile of the current state is away from its correct position in the goal state.

Slide 3

The final approach is run on both google colab and local machine, while other are run only on google colab

Approach	Initial State	Nodes generated	Time taken	Run On
BFS	1	65983	1.0263 secs	Google Colab
BFS	2	Couldn't Find	Memory Error/Session Crash	Google Colab
BFS	3	Couldn't Find	Memory Error/Session Crash	Google Colab
BFS	4	Couldn't Find	Memory Error/Session Crash	Google Colab

Approach	Initial State	Nodes generated	Time taken	Run On
A* + (Hamming distance + level no.) heuristic	1	65983	0.0074 secs	Google Colab
A* + (Hamming distance + level no.) heuristic	2	Couldn't Find	Memory Error/Session Crash	Google Colab
A* + (Hamming distance + level no.) heuristic	3	Couldn't Find	Memory Error/Session Crash	Google Colab
A* + (Hamming distance + level no.) heuristic	4	Couldn't Find	Memory Error/Session Crash	Google Colab

Final approach

Approach	Initial State	Nodes generated	Time taken Local Machine	Time taken Google Colab
A* + (Manhattan Distance + level no.) heuristic	1	32	0.00225 secs	0.00161 secs
A* + (Manhattan Distance + level no.) heuristic	2	129301	6.0971 secs	3.92 secs
A* + (Manhattan Distance + level no.) heuristic	3	802977	38.1903 secs	32.10774 secs
A* + (Manhattan Distance + level no.) heuristic	4	1436627	68.8092 secs	53.870 secs