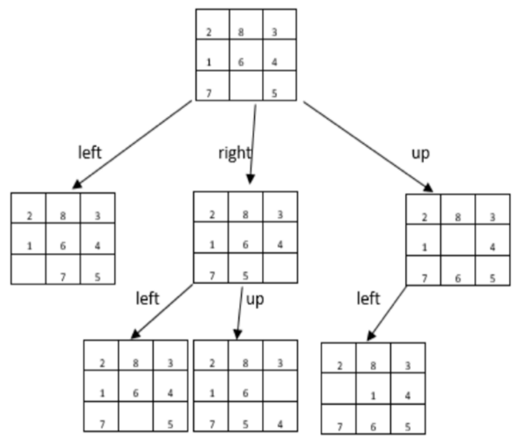
Comparing AI Solutions to the 8-puzzle State Space

1. Introduction

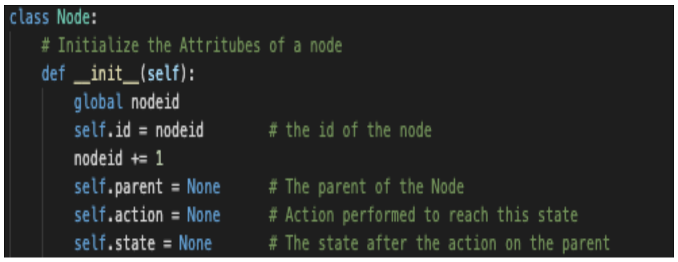
The 8-puzzle problem can be represented as a single agent puzzle where we transform the initial state of the puzzle into an objective or the goal state by sliding the blank tile. First, we represent the environment with a portrait of 3 x 3 grid with 9 pixels. Each pixel is represented by a number from 1 – 8 and the blank tile is represented by 0. A fair measure of exhibition shall be applied to the operator to allow it to act in a constructive manner in order to enter an objective or the goal state at all. There are only 4 actions that can be performed on this puzzle which is to move the blank tile up, down, right and left. When the blanktile is moved to the right, the tile at the right replaces the position of the blank tile while the blank tile replaces the position of the tile at the right. This is illustrated in the fig 1. A new state is returned when an action is performed on the Environment.
In this experiment, we analyze the performance of several search techniques in solving the 8-puzzle problem and their efficiency in finding the optimal solution with least actionable path to the goal with less memory requirement and time requirement. (Mishra & Siddalingaswamy, 2017).



–*Figure 1: A typical state of the puzzle after an action* –

1.1. Search Techniques and Tree Structure

Search is a generic method of logical thought in artificial intelligence. By definition, the sequence of steps necessary for the arrangement of Artificial intelligence problems are not understood from the early stage, but should be dictated by a deliberate experimentation and investigation of options. This is known as search conduct. In the process of searching for the true solution of a problem, the problem must initially be formulated and the goal stated and a search or investigation through different state after each action is carried out until the path to the goal is returned. In the implementation of a search algorithm, spaces are typically represented by a tree in which the status of the space is represented by nodes, and the operator is represented by edges between nodes. Edges can be undirected or guided, depending on whether the related operators are or are not invertible. The initial state or the starting state is represented as the root of the tree and the leaf of the root are the space generated after the legal actions are performed on the state. The node generally is a memory space for storing the current state, the parent, and the actions that was performed to generate this state (see fig. 2). In addition, we need to retain the variety of nodes where the leaf is to be extended or generated — this assortment is known as the fringe or the frontier. A ton of nodes will be the best representation. The investigative technique at that stage will be the



```
class Node:
    # Initialize the Attritubes of a node
    def __init__(self):
        global nodeid
        self.id = nodeid        # the id of the node
        nodeid += 1
        self.parent = None      # The parent of the Node
        self.action = None      # Action performed to reach this state
        self.state = None       # The state after the action on the parent
```

Figure 2: Implementation of the Node Class

capability selected by the following nodes to be expanded from the frontier. Given the fact that this is potentially direct, it may be expensive to measure, since the capability of the technique may need to choose the right node from the collection. The techniques of selecting these nodes leads to several types of the search algorithms. Search process are mainly categorized into uninformed or the blind search and the informed search (Parberry, 2015).

Uninformed Search

The most general search techniques are the uninformed search algorithms since there is no need for any explicit information about the domain or the environment that is- nodes are selected from the frontier with no particular knowledge about how far or close the state in the node is closer to the goal state. All that is required for an uninformed search is the set of states, the legal actions that can be performed, the initial state, and a goal state. The most effective strategy and techniques are the breadth-first, the depth-first, the iterative-deepening, and the bi-directional. These algorithms are evaluated and contrasted based on 3 differentiation:
Time-complexity: The time it takes to find the solution in the tree
Space-Complexity: The memory space that is used by the algorithm to store nodes in the frontier
Optimality: Does this algorithm returns the optimal solution

Breadth-First Search

In Breadth-first technique, the frontier is implemented in a queue like structure where nodes are selected based on First-in-First-Out (FIFO). Looking at this structure, we realize the roots or parents are evaluated first before the leaves or the children. Since the breadth-first method never produces a deeper-level node in a tree until all the nodes at a shallower level have been created, the first path to the target would be the shortest path or an optimum solution. While breadth first search returns the optimal solution or the shortest path to the goal state, it suffers from memory space. This is due to the fact that the frontier keeps track of every nodes. if the first solution is at a deeper depth and the techniques keeps expanding nodes at the shallowest, the memory system can run out of space.

Depth-First Search

In depth-first technique, the frontier is implemented using a stack structure where Last Node into the frontier is first removed and expanded (LIFO). This tackles the memory space that breath first search suffers however returns a non-optimal because it keeps going down into the deepest depth of the tree. In this case, the depth-first search technique is not optimal and not complete, even though the space complexity and the time complexity are minimal.

Uniform cost Search

In uniform cost search, the algorithm first goes through the path that cost less in the tree. Suppose we have a map of Tennessee and our algorithm is trying to find a path from Murfreesboro to Nashville. The algorithm will choose to travel through the path that is the least cost hence returning the shortest path to the Goal. The implementation of this search sort the frontier by an increasing value of the Path Cost. In the 8-puzzle game, we assume the path cost of every action is the same. If these paths are sorted into the frontier, our algorithm will become a breath first search algorithm where the root nodes are evaluated and expanded first before the children or the leaf. This will lead us back to the deficiency of the Breadth-First Search (the space complexity of $O(b^d)$ whose proof is investigated in this paper.

Iterative Deepening

An algorithm that does not suffer the pitfalls of breadth-first or depth-first search is called iterative deepening (ID). The iterative deepening performs a depth-first search with an increasing number of the depth, it starts with a limit of the depth being 1, then continues to increase the depth limit until a solution is found. As it only produces a deeper node until all the shallower nodes have a deeper node. The first solution found by this algorithm is guaranteed to be the optimum. In addition, since at every given point a depth-first is being performed Search, the ID space complexity is O(d) where d is the depth of the solution. Even though it seems that ID spends a lot of time in iterations before the solution is found and returned, the asymptotic time complexity is just the same as the depth first search which is O(d). The obvious explanation for this is that because the number of nodes at a given level of the tree increases exponentially in depth, nearly all of the time is spent in the tree.

Informed Search

An informed search is a search method that has or estimate a knowledge from the specific domain. Generally, an informed search goes through the best path in the tree using the knowledge of an estimate of how close a state is towards the goal. The experiment in this paper will be performed using the informed search and several algorithms in the informed search algorithm will be evaluated. The basic search methods are the greedy-first and the A* algorithm (Ma, 2010).

Greedy-Search

In greedy Search algorithm, a heuristic function is introduced – Heuristic is the estimation of how far and how closer the current state is to the Goal – The frontier is sorted by an increasing value of the heuristics of each state. Then the least value of heuristic is evaluated and expanded. A proper heuristic value will estimate how closer the new state is to the goal state. To build a heuristic or estimation of a map example, we could use a straight-line distance which suggest how short the distance between the current state is to the goal state which is the destination. In the context of the question of a single agent, the heuristic evaluation function is a function from a pair of states to a number that calculates the distance.

A Algorithm*

Since the Uniform-cost Search is similar to the breadth-first algorithm, it suffers the same space complexity as the breadth-first and the Greedy-search suffers the same optimality issue as the depth first search. The A* algorithm combines the efficiency of the uniform search and the Greedy First which uses both the estimation value or the heuristic and the true path cost to select the path to go through in the search Tree. The frontier is sorted by the estimated path cost f(n) value of the current state to the goal which is given by $f(n) = h(n) + g(n)$ where h(n) is the heuristic value and g(n) is the path cost of the current state to the goal state. The optimization of the A*star Algorithm lies in the heuristic function. A good estimation of a state should be an increasing value as we proceed or come closer to the goal and the goal should have a heuristic value of 0 (Mishra & Siddalingaswamy, 2017).

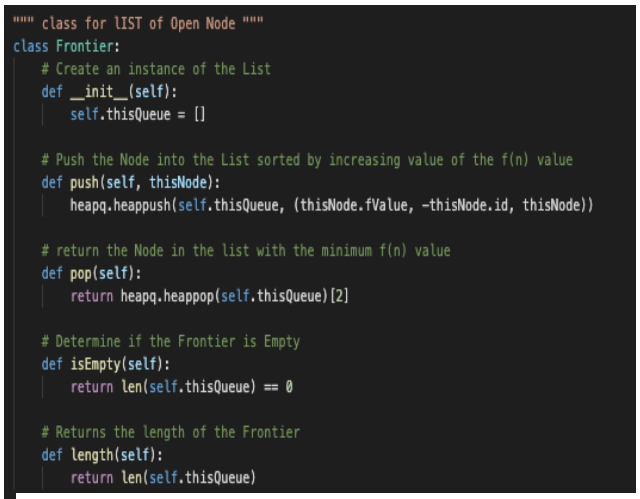
2. Methodology

The investigation and experiment were carried out using 3 heuristic estimation function, and the same path cost towards the goal. We first carried out the implementation of the Board using a cytpypes Array, this allows a minimal space usage compared to the list of list where a hidden multiple storage is created when implemented. The class of the board is created and methods for performing each action (Up, Down, Left and Right) where implemented. A "readGoal" method reads the Goal from a standard input of text file, the shuffle method shuffles the board by random movements of any of the 4 legal actions and returns a new state of the board. The Magic method "__eq__" is used to determine when two states of board are equal and the "__str__" is used to return a string value of the board state. We generate 100 different random Boards and find the path to solution using different heuristic and path cost.

The Agent

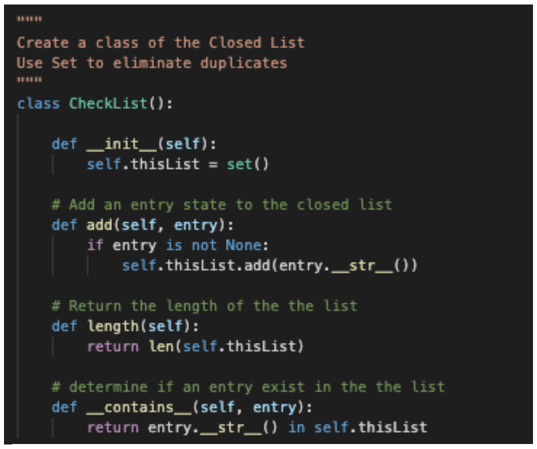
The node was implemented to keep track of the parent state, the action, the state in the node, the path cost, and the estimated path cost f(n) of the state. The node is also initialized with an identity number which increases by 1 when a new node is generated.

The frontier is implemented using a heapqueue and the nodes are sorted by an increasing f(n) value. i.e. the least value of f(n) is pushed to the front of the queue and removed first for evaluation and expansion (see figure 3). A tile breaker is also implemented in the case where the f(n) values of two nodes are equal, the frontier will push the node with less identity value to the front. A second list was also implemented and termed as the closed list. This is used to keep track of the investigated and the expanded list so as to avoid a repetition of investigated state. Since if we have visited a state, we should not be regenerating or revisiting that state. We first check the closed list that a state has not been generated before being pushed to the frontier. The agent is then implemented in Figure 4, The agent takes an input of what heuristic estimation it should use (a more detailed explanation is given later), the path cost (0, 1, 2), the goal state and the initial or starting state. The Agent accept this input, iterate over different state after performing the legal actions, searches and returns the path to the goal using the print function.



```
""" class for lIST of Open Node """
class Frontier:
    # Create an instance of the List
    def __init__(self):
        self.thisQueue = []

    # Push the Node into the List sorted by increasing value of the f(n) value
    def push(self, thisNode):
        heapq.heappush(self.thisQueue, (thisNode.fValue, -thisNode.id, thisNode))

    # return the Node in the list with the minimum f(n) value
    def pop(self):
        return heapq.heappop(self.thisQueue)[2]

    # Determine if the Frontier is Empty
    def isEmpty(self):
        return len(self.thisQueue) == 0

    # Returns the length of the Frontier
    def length(self):
        return len(self.thisQueue)
```

Figure 3: Implementation of the Frontier Queue



```
"""
Create a class of the Closed List
Use Set to eliminate duplicates
"""
class CheckList():

    def __init__(self):
        self.thisList = set()

    # Add an entry state to the closed list
    def add(self, entry):
        if entry is not None:
            self.thisList.add(entry.__str__())

    # Return the length of the the list
    def length(self):
        return len(self.thisList)

    # determine if an entry exist in the the list
    def __contains__(self, entry):
        return entry.__str__() in self.thisList
```

Figure 4: Implementation of the Closed List

The Heuristics

The heuristic function is an estimation of how close and how far we are towards the goal. 3 heuristic estimation were implemented. The *first heuristic* termed as h1 is estimating using the number of tiles that are displaced from the goal, an example of which is shown in figure 5.
The *second heuristic* h2 is the sum of Manhattan distance or city block of each tiles. A tile that is in its proper location. This distance is calculated using the formula – h(n) = $abs(x1 - x2) + abs(y1 - y2)$ where x1 and y1 is the row and column location of a tile in the state and x2, y2 is the row and column location of the same tile on the goal state. Example of this is shown in figure 5. The *third heuristic* is the sum of Euclidean distance between the tiles in the state and the goal state. Using the formula($\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$ where x1 and y1 is the row and column location of a tile in the state and x2, y2 is the row and column location. Any tile that is in its proper position will have this distance equal to 0 (see figure 5)

Initial state	Goal state
1 3 5	0 1 2
7 8 0	3 4 5
6 4 2	6 7 8

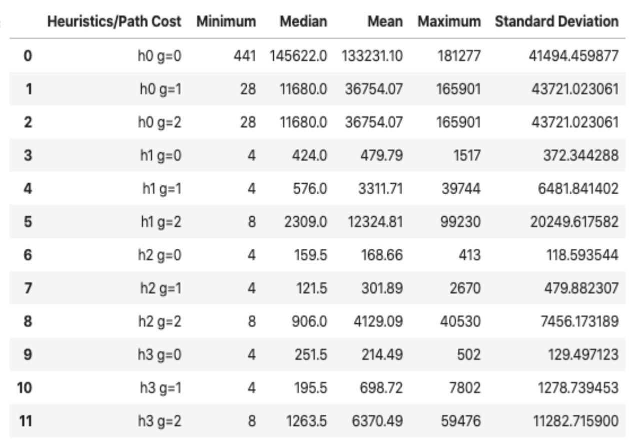
h1, number of displaced tiles = 8
h2, Manhattan Distance = 1 + 2 + 1 + 2 + 2 + 0 + 1 + 2 = 11
h3, Euclidean Distance = 1.0 + 1.41 + 1.0 +1.41 +1.411 + 0.0 +1.0 +2.0 = 9.2

Figure 3: The Heuristics

3. Result and Discussions

We generate 100 unique random boards and solves each board using the highlighted heuristics and path cost on the randomly shuffled board. We solve these boards with a different value of the heuristics.

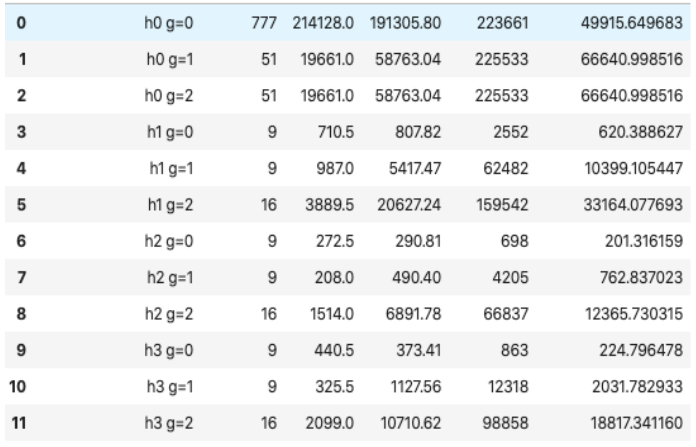
Table 1: Statistics of visited/expanded Nodes in each Case



	Heuristics/Path Cost	Minimum	Median	Mean	Maximum	Standard Deviation
0	h0 g=0	441	145622.0	133231.10	181277	41494.459877
1	h0 g=1	28	11680.0	36754.07	165901	43721.023061
2	h0 g=2	28	11680.0	36754.07	165901	43721.023061
3	h1 g=0	4	424.0	479.79	1517	372.344288
4	h1 g=1	4	576.0	3311.71	39744	6481.841402
5	h1 g=2	8	2309.0	12324.81	99230	20249.617582
6	h2 g=0	4	159.5	168.66	413	118.593544
7	h2 g=1	4	121.5	301.89	2670	479.882307
8	h2 g=2	8	906.0	4129.09	40530	7456.173189
9	h3 g=0	4	251.5	214.49	502	129.497123
10	h3 g=1	4	195.5	698.72	7802	1278.739453
11	h3 g=2	8	1263.5	6370.49	59476	11282.715900

We solve the boards with h(n) = 0, g(n) = 0. A practical overview of this solution shows that it is a depth first search since ties are broken in their order of nodes identity where the last tile to come into the frontier were first removed and explored. The maximum number of nodes visited by this algorithm is running high up 200,000 nodes with a depth of 114685. This is a very long path to the solution even though it uses less memory space as seen in Table 2 The same board were solved with a path cost of g(n) = 1 and with no heuristic h(n) = 0. We conclude that this is a uniform cost search since all nodes will have a path cost equivalent to 1. However, nodes will be expanded in an increasing order of their path cost. We realize this algorithm is similar to the breadth first and suffers the same space complexity. This is proved by the number of nodes visited and explored by the algorithm in comparison with the branching factor. The highest of this solution running over 225,000

Table 2: Statistics of Maximum nodes stored in memory in each case



0	h0 g=0	777	214128.0	191305.80	223661	49915.649683
1	h0 g=1	51	19661.0	58763.04	225533	66640.998516
2	h0 g=2	51	19661.0	58763.04	225533	66640.998516
3	h1 g=0	9	710.5	807.82	2552	620.388627
4	h1 g=1	9	987.0	5417.47	62482	10399.105447
5	h1 g=2	16	3889.5	20627.24	159542	33164.077693
6	h2 g=0	9	272.5	290.81	698	201.316159
7	h2 g=1	9	208.0	490.40	4205	762.837023
8	h2 g=2	16	1514.0	6891.78	66837	12365.730315
9	h3 g=0	9	440.5	373.41	863	224.796478
10	h3 g=1	9	325.5	1127.56	12318	2031.782933
11	h3 g=2	16	2099.0	10710.62	98858	18817.341160

Table 3: Statistics values of the branching factors

	Heuristics/Path Cost	Minimum	Median	Mean	Maximum	Standard Deviation
0	h0 g=0	1.000107	1.000130	1.000324	1.015489	0.001538
1	h0 g=1	1.577544	1.756177	1.764760	2.297397	0.106821
2	h0 g=2	1.577544	1.756177	1.764760	2.297397	0.106821
3	h1 g=0	1.022751	1.060551	1.098416	1.551846	0.104797
4	h1 g=1	1.409802	1.484251	1.480649	1.551846	0.023630
5	h1 g=2	1.551998	1.603816	1.608746	1.741101	0.026697
6	h2 g=0	1.058527	1.119452	1.151685	1.551846	0.096751
7	h2 g=1	1.252433	1.348291	1.344675	1.551846	0.047720
8	h2 g=2	1.476060	1.512776	1.518284	1.741101	0.037052
9	h3 g=0	1.053804	1.108918	1.155762	1.551846	0.111566
10	h3 g=1	1.276647	1.386966	1.385076	1.551846	0.039855
11	h3 g=2	1.513524	1.548555	1.553073	1.741101	0.032975

nodes. Looking into the statistics table in Table 4, the depth of each algorithm - depth is the number of moves that leads to a solution - we realized that only algorithms that has a path cost value returned the optimal path. Algorithms that has no information about the path cost return a non-optimal solution. This is obvious where the maximum depth in all of the 100 boards were higher than all other algorithms. Comparing the heuristics h1, h2 and h3. We can conclude that they were all admissible heuristics since they returned the optimal solutions of all the boards when there is an information about the cost of each action on the board. However, h1 evaluated and expands more nodes as seen in Table 1 and stored more nodes on the memory running into an average 5417 nodes when the path cost is 1 and over 200,000 when the path cost is 2. However, h2 had an average of 490 nodes stored in the memory and h3 had 1127 stored in the memory space. We also look at the opened list statistics, h2 heuristics value opened fewer nodes than h1 and h2. We therefore conclude that h2 is a better heuristic estimation over h1 and h3 and it is dominant over h1 and h3.

Table 4: Statistical values for the depth, d of each algorithms

	Heuristics/Path Cost	Minimum	Median	Mean	Maximum	Standard Deviation
0	h0 g=0	433	94493.0	87178.88	114685	27588.014739
1	h0 g=1	5	17.0	18.00	27	4.658001
2	h0 g=2	5	17.0	18.00	27	4.658001
3	h1 g=0	5	117.0	118.56	347	76.625591
4	h1 g=1	5	17.0	18.00	27	4.658001
5	h1 g=2	5	17.0	18.00	27	4.658001
6	h2 g=0	5	49.0	50.68	115	26.904575
7	h2 g=1	5	17.0	18.00	27	4.658001
8	h2 g=2	5	17.0	18.00	27	4.658001
9	h3 g=0	5	59.0	53.88	129	27.508412
10	h3 g=1	5	17.0	18.00	27	4.658001
11	h3 g=2	5	17.0	18.00	27	4.658001

Another conclusion in this experimentation is that the path cost selection especially for movement such as that which is legal in the 8-puzzle game is very important. A value of 2 as the path cost has a very large adverse effect on the space and time complexity. This is comparable using the number of explored nodes when the path cost is 2 and when the path costs are 1 in the table. We then look into the branching factors in table 3- branching factor refers to the factor of the solutions returning to explore the shallowest state- this is calculated using $d\sqrt{N}$ where d is the depth of the solution and N is the number of nodes stored throughout the process of solving the board. The lesser the branching factor, the better the algorithm. However, since we will not trade off the optimal path of the solution with the branching factor, we only compare the branching factor of algorithms that returns the optimal solutions. We conclude that h2 is a better heuristic with a lower branching factor.

References

Ma, X. (2010). A constructive proof for the subsets completeness of 8-puzzle state space. *2010 International Conference on Artificial Intelligence and Education (ICAIE)* (pp. 644-647). Hangzhou, China: IEEE.

Mishra, A. K., & Siddalingaswamy, P. C. (2017). Analysis of tree based search techniques for solving 8-puzzle problem. *Innovations in Power and Advanced Computing Technologies (i-PACT)*. Vellore, India: IEEE.

Parberry, I. (2015). A Memory-Efficient Method for Fast Computation of Short 15-Puzzle Solutions. *IEEE Transactions on Computational Intelligence and AI in Games. vol. 7, no. 2, pp*, pp. 200-203. IEEE.