**Part 1**

**<Task 1.1>**

def is\_solved(self):  
 *"""  
 Returns True if the puzzle is solved, False otherwise  
 """* ######## TASK 1.1 BEGIN ##########  
 #Add code to determine whether this puzzle is solved  
 num = 0  
 for i in self.state:  
 if i == num:  
 num = num + 1  
 else:  
 return False  
 return True   
 ######## TASK 1.1 END ##########

**<Task 1.2>**

- easy.txt

40 puzzles solved, Avg nodes expanded: 334, Avg search time: 0.0068, Avg solution length: 7.0

- medium.txt

40 puzzles solved, Avg nodes expanded: 28269, Avg search time: 0.318, Avg solution length: 15.0

- hard.txt

5 puzzles solved, Avg nodes expanded: 730421, Avg search time: 8.27, Avg solution length: 21.0

- random.txt

5 puzzles solved, Avg nodes expanded: 1616691, Avg search time: 18.3, Avg solution length: 20.2

- worst.txt

It took so long to get a result.

**<Task 1.3>**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Evaluation Type | Avg Nodes Expanded | Avg Search Time | Avg Solution length |
| Easy.txt | Uniform | 143 | 0.034 | 7 |
| Greedy | 53 | 0.012 | 11 |
| A\* | 13 | 0.004 | 7 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Evaluation Type | Avg Nodes Expanded | Avg Search Time | Avg Solution length |
| Medium.txt | Uniform | 6407 | 0.21 | 15 |
| Greedy | 625 | 0.017 | 76 |
| A\* | 341 | 0.01 | 15 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Evaluation Type | Avg Nodes Expanded | Avg Search Time | Avg Solution length |
| hard.txt | Uniform | 72320 | 3.4 | 21 |
| Greedy | 716 | 0.02 | 94 |
| A\* | 4888 | 0.15 | 21 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Evaluation Type | Avg Nodes Expanded | Avg Search Time | Avg Solution length |
| random.txt | Uniform | 58265 | 3.6 | 17 |
| Greedy | 363 | 0.01 | 57 |
| A\* | 11867 | 0.43 | 17 |
|  | Evaluation Type | Avg Nodes Expanded | Avg Search Time | Avg Solution length |
| worst.txt | Uniform | 181315 | 13.6 | 31 |
| Greedy | 772 | 0.02 | 101 |
| A\* | 100267 | 4.13 | 30 |

**<Task 1.4>**

1) Task 1.4.1

def check\_right\_row(curr\_index, number):  
 if number == 0:  
 return True  
  
 right\_row = get\_tile\_row(number)  
 curr\_row = get\_tile\_row(curr\_index)  
  
 if right\_row == curr\_row:  
 return True  
 return False  
  
def check\_right\_col(curr\_index, number):  
 if number == 0:  
 return True  
  
 right\_col = get\_tile\_column(number)  
 curr\_col = get\_tile\_column(curr\_index)  
  
 if right\_col == curr\_col:  
 return True  
 return False  
  
def tiles\_out\_of\_row\_column(puzzle):  
 *"""  
 This heuristic counts the number of tiles that are in the wrong row,   
 the number of tiles that are in the wrong column  
 and returns the sum of these two numbers.  
 Remember not to count the blank tile as being out of place, or the heuristic is inadmissible  
 """* ######## TASK 1.4.1 BEGIN ##########  
 # YOUR TASK 1.4.1 CODE HERE  
  
 wrong\_row = 0  
 wrong\_col = 0  
 for i in range(len(puzzle.state)):  
 if not check\_right\_row(i, puzzle.state[i]):  
 wrong\_row = wrong\_row + 1  
 if not check\_right\_col(i, puzzle.state[i]):  
 wrong\_col = wrong\_col + 1  
  
 return wrong\_row + wrong\_col  
   
 ######## TASK 1.4.1 END ##########

2) Task 1.4.2

def manhattan\_distance\_to\_goal(puzzle):  
 *"""  
 This heuristic should calculate the sum of all the manhattan distances for each tile to get to   
 its goal position. Again, make sure not to include the distance from the blank to its goal.  
 """* ######## TASK 1.4.2 BEGIN #########  
 # YOUR TASK 1.4.2 CODE HERE  
  
 # Making 2-d arrays to calculate MD  
 curr\_state = []  
 index = 0  
 for i in range(3):  
 temp = []  
 for j in range(3):  
 temp.append(puzzle.state[index])  
 index = index + 1  
 curr\_state.append(temp)  
  
 md\_sum = 0  
 for i in range(3):  
 for j in range(3):  
 curr\_num = curr\_state[i][j]  
 if curr\_num != 0:  
 goal\_x = get\_tile\_row(curr\_num)  
 goal\_y = get\_tile\_column(curr\_num)  
 dist = abs(i - goal\_x) + abs(j - goal\_y)  
 md\_sum = md\_sum + dist  
 return md\_sum  
 ######## TASK 1.4.2 END ##########

**<Task 1.5>**

1. hard.txt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Heuristic Function | Evaluation Type | Avg Nodes Expanded | Avg Search Time | Avg Solution Length |
| Top | Uniform | 72320 | 3.26 | 21 |
| Greedy | 716 | 0.02 | 95 |
| A \* | 4889 | 0.14 | 21 |
| Torc | Uniform | 72320 | 4.99 | 21 |
| Greedy | 354 | 0.02 | 84 |
| A \* | 1225 | 0.06 | 21 |
| Md | Uniform | 72320 | 4.89 | 21 |
| Greedy | 278 | 0.013 | 66 |
| A \* | 436 | 0.019 | 21 |

2. random.txt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Heuristic Function | Evaluation Type | Avg Nodes Expanded | Avg Search Time | Avg Solution Length |
| Top | Uniform | 58265 | 3.6 | 17 |
| Greedy | 363 | 0.01 | 57 |
| A \* | 11867 | 0.41 | 16 |
| Torc | Uniform | 58265 | 5.0 | 17 |
| Greedy | 247 | 0.012 | 57 |
| A \* | 3531 | 0.17 | 16 |
| Md | Uniform | 58265 | 4.9 | 17 |
| Greedy | 126 | 0.006 | 34 |
| A \* | 1225 | 0.05 | 17 |

First of all, using iterative-deepening search was very slow to get results for the harder puzzles. The space complexity is definitely better than some other approaches like greedy search, but it was not an optimal way to find solution since it is too slow.

For the uniform-cost search, the average nodes expanded were the largest among the experiments since it kept generating and expanding the nodes as this method only cares about the cost. So, it took long time to get to the right solution, and it was not an optimal method. The greedy search was the fastest algorithm from these experiments because this method only cares about the next best option. That gave me the fastest time to find the solutions, but it was the optimal because the average of solution length was the largest for the greedy search. It only cares about speed, but it doesn’t know if it is on the optimal path to the goal or not. For A star search, it was able to find the optimal path but it was slower than the greedy search and still taking lots of memory spaces. Therefore, it seemed not practical although this method is complete theoretically. Therefore, I learned that using A start could be not efficient when sample state is really large.

**Part 2**

1.

def run\_iterative\_search(start\_node):  
 *"""  
 This runs an iterative deepening search  
 It caps the depth of the search at 40 (no 8-puzzles have solutions this long)  
 """* #Our initial depth limit  
 #depth\_limit = 1  
  
 #Maximum depth limit  
 #max\_depth\_limit = 40  
   
 #Keep track of the total number of nodes we expand  
 total\_expanded = 0  
  
 start\_node.compute\_f\_value()  
 cutoff = start\_node.f\_value  
 #Keep trying until our depth limit hits 40  
 #while depth\_limit < max\_depth\_limit:  
  
 while True:  
 #Store visited nodes along the current search path  
 visited = dict()  
 visited['N'] = 0  
  
 #Mark the initial state as visited  
 visited[start\_node.puzzle.id()] = True  
   
 #Run depth-limited search starting at initial node (which points to initial state)  
 #path\_length = run\_dfs(start\_node, depth\_limit, visited)  
 path\_length = run\_dfs(start\_node, cutoff, visited)  
  
 #See how many nodes we expanded on this iteration and add it to our total  
 total\_expanded += visited['N']  
   
 #Check to see if a solution was found  
 if path\_length is not None:  
 #It was! Print out information and return the search stats  
 print('Expanded ', total\_expanded, 'nodes')  
 #print('IDS Found solution at depth', depth\_limit)  
 return total\_expanded, path\_length  
   
 # No solution was found at this depth limit, so increment our depth-limit   
 #depth\_limit += 1  
 cutoff = cutoff + 1  
  
 # No solution was found at any depth-limit, so return None,None (Which signifies no solution found)  
 return None, None  
   
def run\_dfs(node, depth\_limit, visited):  
 *"""  
 Recursive Depth-Limited Search:   
   
 Check node to see if it is goal, if it is, print solution and return path length  
 If not and if depth-limit hasn't been reached, recurse on all children  
 """* visited['N'] = visited['N'] + 1 #Increment our node expansion counter  
  
 # Check to see if this is a goal node  
 if node.puzzle.is\_solved():  
 # It is! Print out solution and return solution length  
 print('Iterative Deepening SOLVED THE PUZZLE! SOLUTION = ', node.path)  
 return len(node.path)  
   
 # Check to see if the depth limit has been reached (number of actions that have been taken)  
 #if len(node.path) >= depth\_limit:  
 node.compute\_f\_value()  
 if node.f\_value > depth\_limit:  
 # It has. Return None, signifying that no path was found  
 return None  
   
 # Generate successors and recurse on them  
   
 # Get the list of moves we can try from this node's state  
 moves = node.puzzle.get\_moves()  
   
 # For each possible move  
 for m in moves:  
 #Execute the move/action  
 node.puzzle.do\_move(m)  
 node.compute\_f\_value()  
  
  
 #Add this move to the node's path  
 node.path = node.path + m  
 #Add 1 to node's cost  
 node.cost = node.cost + 1  
 #Check to see if we have already visited this node  
 if node.puzzle.id() not in visited:  
 #We haven't. Now we will, so add it to visited  
 visited[node.puzzle.id()] = True  
   
 #Recurse on this new state  
 path\_length = run\_dfs(node, depth\_limit, visited)  
   
 #Check to see if a solution was found down this path (return value of None means no)  
 if path\_length is not None:  
 #It was! Return this solution path length to whoever called us  
 return path\_length  
  
 #Remove this state from the visited list. We only check for duplicates along current search path  
 del visited[node.puzzle.id()]  
  
 # That move didn't lead to a solution, so lets try the next one  
 # First, though, we need to undo the move (to return puzzle to state before we tried that move)  
 node.puzzle.undo\_move(m)  
 # Remove that last move we tried from the path  
 node.path = node.path[0:-1]   
 # Remove 1 from node's cost  
 node.cost = node.cost - 1  
   
 #Couldn't find a solution here or at any of my successors, so return None  
 #This node is not on a solution path under the depth-limit  
 return None

2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Heuristic Function | Avg Nodes Expanded | Avg Search Time | Avg Solution length |
| hard.txt | Top | 27242 | 0.39 | 21 |
| torc | 5869 | 0.16 | 21 |
| md | 1394 | 0.04 | 21 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Heuristic Function | Avg Nodes Expanded | Avg Search Time | Avg Solution length |
| random.txt | Top | 167633 | 2.2 | 17 |
| torc | 33956 | 0.93 | 17 |
| md | 7655 | 0.19 | 17 |

3.

The biggest difference that I noticed during implementation the iterative-deepening a star search was that it was based on depth-first search. If I compare the result of the IDA\* to the normal A\* search, the average search time for the normal A\* algorithm resulted in faster solution, and the normal A\* expanded less nodes than the implemented IDA\*. This is because the same nodes were expanded again if no solution was found at certain cutoff. After increasing the cutoff by one since every cost is one in this case, so the nodes that were expanded are expanded again with higher computed f-value. So, we can see the average nodes expanded columns are much larger for the IDA\* algorithm. However, the biggest advantage of using this revised algorithm is on the memory side since it uses depth-first search at its core which uses less memory than the normal A\* search. I was able to learn the pros and cons when using each search methods. I think that IDA\* would be more useful when we have to deal with huge sample space. But when the space is relatively small when we care more about time, then it might be better to choose the normal A\* algorithm to proceed.