**CS 205: Artificial Intelligence**

**Assignment 1**

**Name** : Yash Aggarwal

**SID**  : 862333037

**Email**  : [yagga004@ucr.edu](mailto:yagga004@ucr.edu)

**NetID**  : yagga004

**Date**  : 14/May/2023

In completing this assignment I consulted:

1. The Project Statement Handout provided.
   1. <https://d1u36hdvoy9y69.cloudfront.net/cs-205-ai/Project_1_The_Eight_Puzzle_CS_205.pdf>
2. Depth 31 Puzzles were taken from
   1. <https://www.researchgate.net/figure/8-Puzzle-problem-instances_tbl1_280545587>
3. To check if a puzzle is solvable or not in constant time and by not exploring the entire search space. This was used only for generating puzzle problems and was not used in original search code.
   1. <https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/>

All the code is original **Except**

* Using the math module to calculate the square root of a number
* Using the copy module to make deep copies of a node
* Using the time module to keep track of time
* Using the numpy module to generate random numbers
* Using the matplotlib module to plot graphs and figures
* To check if a puzzle is solvable or not in constant time and by not exploring the entire search space. This was used only for generating puzzle problems and was not used in original search code.
  + <https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/>

**Link to Code**

* The code is available on github → <https://github.com/yashUcr773/CS_205_AI/tree/main/Projects/Project%201>
* The code can also be run on google colab → [**https://colab.research.google.com/drive/18yNb0bLmRX-0jsQMP5Q\_JEtD-tOi4NMS**](https://colab.research.google.com/drive/18yNb0bLmRX-0jsQMP5Q_JEtD-tOi4NMS?usp=sharing)

**Outline of this report**

1. Cover Page: (this page)
2. Report: Pages 02 - 09
3. Sample Trace of Easy problem: Pages 10 - 12
4. Sample Medium problem: Pages 13
5. Sample Hard problem: Pages 14
6. My Code: Pages 15 - 33

**CS 205: Artificial Intelligence**

**Project 1: The 8-Puzzle**

**Project Report**

**Name** : Yash Aggarwal

**SID**: 862333037

# 1. Introduction

A sliding tile puzzle, as seen in Figure 1, is a combinatorial puzzle that consists of numbers from 1 to N where N is a perfect square (4,9,16,25…) and the numbers are arranged in a N x N grid. The last tile is removed from the puzzle leaving 1 spot open so total tiles are from 1 to N-1. The goal of the puzzle is to move the tiles in such a way that all the numbers are arranged in ascending order and the empty spot is the last tile. The 8 puzzle is a sub-problem of the sliding puzzle with N set as 9 and tiles are numbered from 1 to 8.



Figure 1: An 8 puzzle

Source: <https://play.google.com/store/apps/details?id=com.gsr.npuzzle>

This report details the findings for the 8-puzzle solved using different algorithms and a comparison between the algorithms. However, the code attached is generic and and be used to solve any NxN puzzle with any goal state.

In this project we attempt to solve the 8-puzzle using

1. Uniform Cost Search Algorithm
2. A\* Algorithm with Misplaced Tile Heuristic
3. A\* Algorithm with Manhattan Distance Heuristic

The project and report are a requirement for completion of the course CS 205: Artificial Intelligence taken under professor Dr. Eamonn Keogh in the Spring Quarter of 2023 at the University of California, Riverside.

The Language of choice is Python (version 3.9.X) and additional imports include (matplotlib, numpy, time, math, copy and os). The Report also includes the original source code and link to run it.

Within this report the word puzzle and problem are used interchangeably and denote the state of the puzzle board provided by the user.

The Puzzles in this report are classified as Easy, Medium and Hard.

* Puzzles with depth <=9 are considered to be Easy.
* Puzzles with depth >= 10 and depth <= 19 are considered medium.
* Puzzles with depth >= 20 are considered Hard.

This is an informal metric for puzzle classification used in this report.

# 2. Algorithms

The algorithms used to solve the 8-puzzle are

1. Uniform Cost Search Algorithm
2. A\* Algorithm with Misplaced Tile Heuristic
3. A\* Algorithm with Manhattan Distance Heuristic

## 2.1 Uniform Cost Search Algorithm

Uniform Cost search, also known as the Uninformed search, is a search algorithm that assigns a uniform cost (say 0) as a cost of expansion of every node. This means that the total cost to expand a node is only its depth. This also means that the cost of expanding all the nodes at any given depth d is the same for all the nodes.

When compared to A\* Search and while implementing the algorithm in the project,

g(n) = depth of the node

h(n) = 0

Total cost to expand a node = h(n) + g(n) = g(n)

## 2.2 A\* Search Algorithm

A\* search, also known as the Informed search, is a search algorithm that uses a heuristic or a cost function for each node to measure how farther or closer the expansion will take us to the solution. It is referred to as Informed search as we make informed decisions based on cost function for which nodes need to be expanded next. The node with the smallest cost is selected.

This means all the nodes have different cost of expansion that depends on their depth and how close to the final state we are.

For this project we have chosen 2 Heuristic functions.

1. Misplaced Tile Heuristic
2. Manhattan Distance Heuristic

### 2.2.1 Misplaced Tile Heuristic

The Misplaced tile heuristic or the hamming distance heuristic counts the total number of tiles that are not in their correct position when compared to the goal state. While calculating the heuristic, we do not consider the placeholder (blank, \_, 0) in the calculations.

In Figure 2, we can see there are 5 tiles namely (1,2,5,6,8) that are in incorrect positions and only 3 tiles (3,4,7) that are in correct positions. Therefore the misplaced tile heuristic or the hamming heuristic for the problem is 5.

Let us say this node is at depth d.

The total cost of expansion of this node would be g(n) + h(n) = d + 5.

Instead of just d in case of uniform search.

With this heuristic cost function we could expand the nodes with the smallest cost and thus expand cheaper nodes first, leading to goal nodes in fewer expansions.

### 2.2.2 Manhattan Distance Heuristic

The Manhattan distance heuristic counts the minimum number of moves that would be required to move a tile to its correct place assuming there are no other tiles on the board. While calculating the heuristic, we do not consider the placeholder (blank, \_, 0) in the calculations.

In Figure 2, we can see there are 5 tiles namely (1,2,5,6,8) that are in incorrect positions and it would take (1,2,2,2,3) moves respectively for each tile to be in the correct position. Therefore the Manhattan heuristic value is 10.

Let us say this node is at depth d.

The total cost of expansion of this node would be g(n) + h(n) = d + 10.

Instead of just d in case of uniform search.

With this heuristic cost function we could expand the nodes with the smallest cost and thus expand cheaper nodes first, leading to goal nodes in fewer expansions.

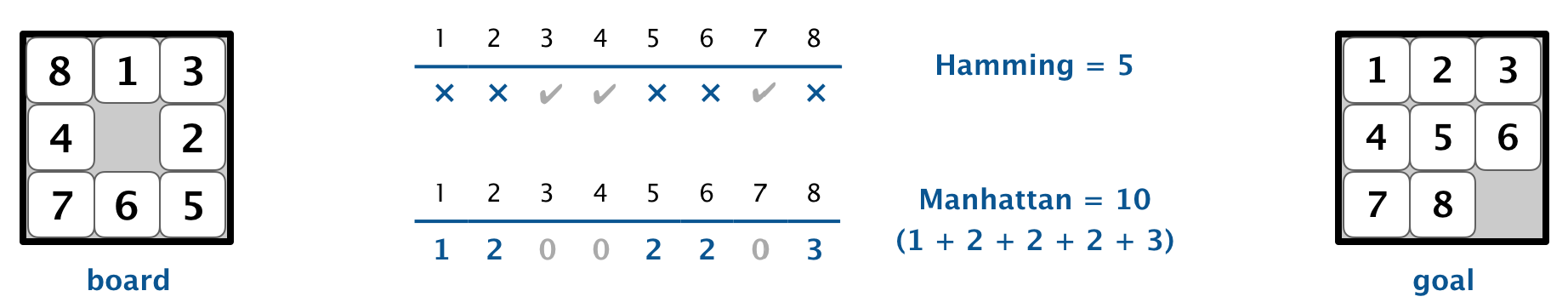


Figure 2: Misplaced Tile (Hamming) Heuristic and Manhattan Heuristic

Source: <https://coursera.cs.princeton.edu/algs4/assignments/8puzzle/hamming-manhattan.png>

# 3. Comparison of Algorithms

We will now be running our program on various 8-puzzles of varied depths and visualizing the results.

Figure 3 Denotes the list of puzzles used in testing. These were a good starting point to test the algorithms in early development stages.

****

Figure 3: List of Problem States provided by Professor Keogh

Figure 4 Denotes some of the puzzles that I created for other depths. These were generated by me using the code that is attached with this report. These were helpful in generating the reports for analysis.

****

Figure 4: List of Problem States generated by me. These are used to test the algorithms and run analysis.

In Figure 5, we can see the comparison of algorithms based on time taken to solve the puzzles at various depths. A\* Search with Manhattan distance heuristic seems to be performing the best among the three algorithms. Even on the hardest puzzle with depth 31, the algorithm seems to take merely 20 seconds to search for the answer.

The second best algorithm seems to be A\* search with Misplaced Tile heuristic. For easy to medium puzzles, The Misplaced Tile Heuristic takes little to no time. However as we move towards harder problems, we can see the sharp increase in time taken to run the algorithm with the hardest taking close to 750 seconds to run a problem with depth 31.

Finally the Uniform Search performed the worst among the three algorithms. Half way through the medium puzzles we can see it taking longer to search for the goal state with the algorithm taking around 700 seconds for the depth 31 puzzle. However, at the highest depths, we can see that Misplaced Tile heuristic takes more time than Uniform Cost Search.

The better performance of A\* algorithms can be attributed to the cost function heuristic that helps the algorithm expand only the best tile closest to solution as opposed to uniform search that uses no cost function.

As for the manhattan and misplaced tile heuristics, the former runs better than the latter as it better estimates the cost function and thus expands the nodes closest to goal.

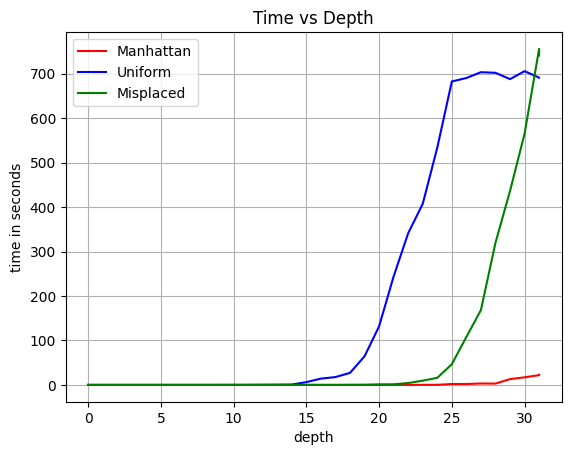


Figure 5: Comparison of all three algorithms based on time taken to reach goal state vs the depth of solution

In Figure 6, we can see the comparison of algorithms based on the number of nodes expanded to solve the puzzles of various depths. Similar results can be drawn here as Figure 5.

A\* Search with Manhattan distance heuristic seems to be performing the best among the three algorithms. Even on the hardest puzzle with depth 31, the algorithm expands around 24,000 nodes to reach the goal state.

The second best algorithm seems to be A\* search with Misplaced Tile heuristic. For easy to medium puzzles, The Misplaced Tile Heuristic expanded a low amount of nodes. However as we move towards harder problems, we can see the sharp increase in total nodes expanded with the hardest taking close to 150,000 nodes for a problem with depth 31

Finally the Uniform Search performed the worst among the three algorithms. Half way through the medium puzzles we can see it expanding a large number of nodes to search for the goal state with a maximum of 180,000 at depth 31

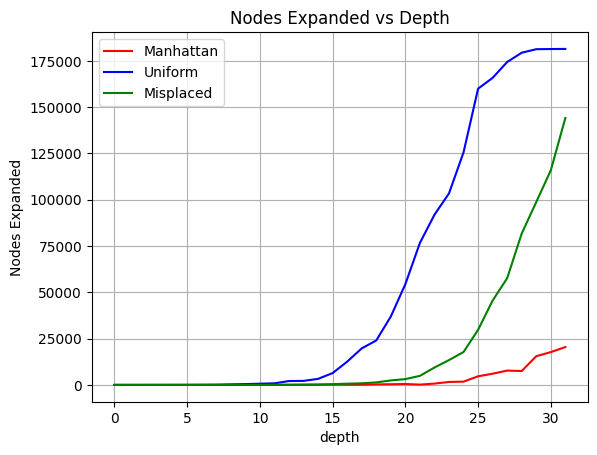


Figure 6: Comparison of all three algorithms based on number of nodes expanded to reach goal state vs the depth of solution

In Figure 7, we can see the comparison of algorithms based on the maximum length of the frontier queue at various solution depths. Similar results can be drawn here as Figure 5 and 6.

A\* Search with Manhattan distance heuristic seems to be performing the best among the three algorithms. Even on the hardest puzzle with depth 31, the algorithm had a maximum of around 9,000 nodes in its queue.

The second best algorithm seems to be A\* search with Misplaced Tile heuristic. For easy to medium puzzles, The Misplaced Tile Heuristic had a low amount of nodes in its queue. However as we move towards harder problems, we can see a sharp increase in queue size with the largest queue size of approximately 25,000 for a problem with depth 31

Finally the Uniform Search performed the worst among the three algorithms. At the start of medium puzzles, we can see it has a very large queue size and the trend continues as we move towards the hard puzzles with the hardest expanding close to 28000 nodes. At the very high depth (28, 29, 30, 31) we can see that Uniform search and Misplaced search have a similar max queue size.

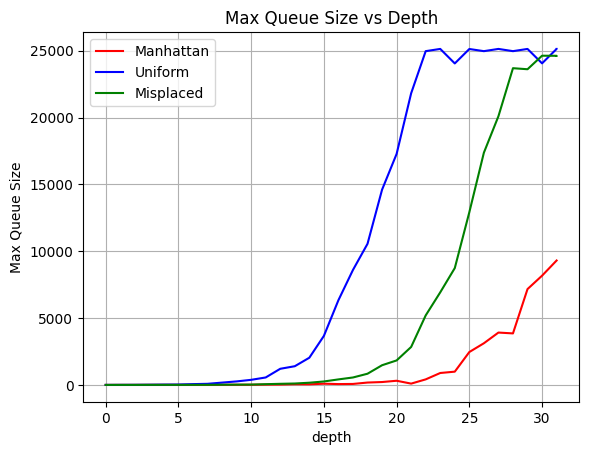


Figure 7: Comparison of all three algorithms based on maximum length of frontier queue vs the depth of solution

# 4. Conclusions

After testing the puzzles on all the algorithms, it was found that.

1. Clearly The A\* Algorithm with Manhattan Distance Heuristic is the best algorithms from the three algorithms.
2. All the algorithms did similar in terms of time, nodes expanded and max length of queue for easier puzzles (depth < 10)
3. For the medium puzzles (depth >= 10 and depth <20), we see that uniform search takes a lot more time whereas Manhattan and Misplaced heuristics are still comparable. Same is true for total nodes expanded and Max nodes in queue.
4. For the Hard Puzzles (depth >= 20), we can see that uniform search seems to take a considerable amount of time with Misplaced Heuristic not far behind but Manhattan Heuristic still seems to be very fast taking less than 20 seconds for the hardest puzzle.
5. Manhattan Distance Heuristics outperforms Misplaced Tile and Uniform cost search by a large margin and Misplaced tile heuristic performs considerably better than Uniform Cost search
6. For the hardest Puzzles at depths 29, 30 and 31, we can see that Misplaced Tile search is a little slower than the Uniform cost search and takes almost the same amount of memory. The only benefit at greater depths of using Misplaced Tile search instead of Uniform cost search is that it expands fewer nodes than Uniform Cost Search.

# 

# 5. Traceback of easy puzzle

Figures 8, 9 and 10 show the code running for an easy problem and its traceback.

In Figure 8, we can see that by choosing A\* Search with Manhattan distance heuristic on an easy puzzle of depth 7, we can find the solution in 0.001 secs with expanding only 7 nodes and having a max frontier size of 8

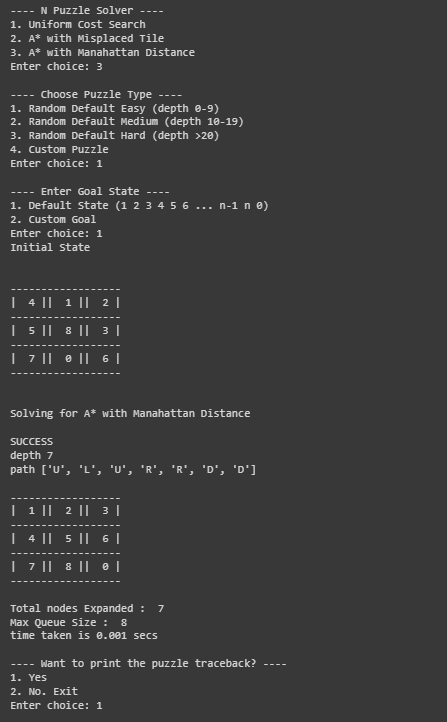
****

Figure 8: Take Input from user, search to find the solution. Display the result. (Easy)

****

Figure 9: Print Traceback with best move and its cost for the problem in figure 8

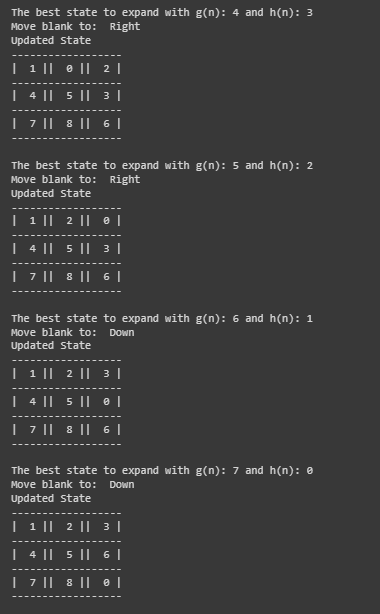
****

Figure 10: Continued Printing of Traceback from figure 9

# 6. Solution of a Medium Puzzle without Traceback

Figure 11 shows the code running for a medium problem without its traceback.

In Figure 11, we can see that by choosing A\* Search with Misplaced tile distance heuristic on a medium puzzle of depth 15, we can find the solution in 0.05 secs with expanding 637 nodes and having a max frontier size of 379

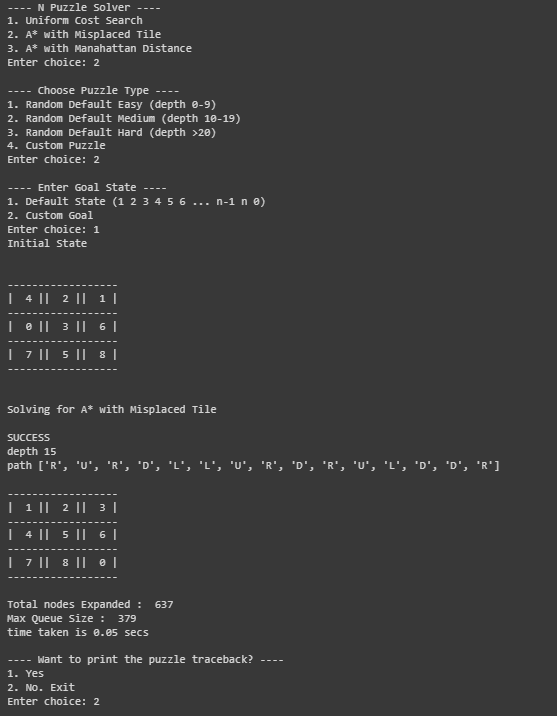
****

Figure 11: Take Input from user, search to find the solution. Display the result. (Medium)

# 7. Solution of a Hard Puzzle without Traceback

Figure 12 shows the code running for a Hard problem without its traceback.

In Figure 12, we can see that by choosing A\* Search with Manhattan distance heuristic on a hard puzzle of depth 30, we can find the solution in 11 secs with expanding 17722 nodes and having a max frontier size of 8174

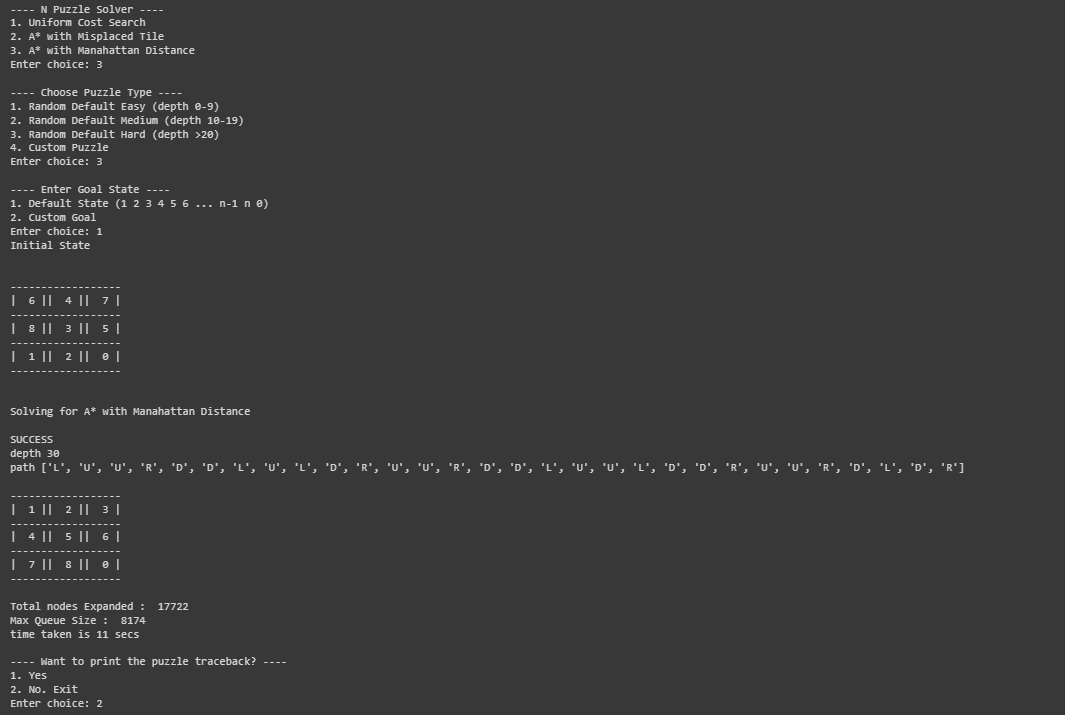
****

Figure 12: Take Input from user, search to find the solution. Display the result. (Hard)

# 8. Original Code

The original code can be found at

* The code is available on github → <https://github.com/yashUcr773/CS_205_AI/tree/main/Projects/Project%201>
* The code can also be run on google colab → [**https://colab.research.google.com/drive/18yNb0bLmRX-0jsQMP5Q\_JEtD-tOi4NMS**](https://colab.research.google.com/drive/18yNb0bLmRX-0jsQMP5Q_JEtD-tOi4NMS?usp=sharing)

Initial.py

'''

Solve 8 puzzle with

- Uniform cost search

- A\* with misplaced tile

- A\* with manhattan distance

extensible code

- can work for any N puzzle.

- can update heuristic function to use any custom function

'''

#############################################################

######## LIBRARY IMPORTS ########

#############################################################

# for square root of numbers

import math

# clear output screen

import os

# to make deep copies of nodes' states

import copy as cpy

# to track the time used for execution

import time

# for creating random states

import numpy as np

# for plotting results and graphs

import matplotlib.pyplot as plt

#############################################################

######## CONSTANTS ########

#############################################################

UNIFORM = 'Uniform'

MISPLACED = 'Misplaced'

MANHATTAN = 'Manhattan'

COLOR\_MAP = {

MANHATTAN: 'red',

MISPLACED: 'green',

UNIFORM: 'blue',

}

DIRECTIONS\_MAP = {

'R': 'Right',

'L': 'Left',

'U': 'Up',

'D': 'Down'

}

#############################################################

######## LIST OF PUZZLES ########

#############################################################

# Stores some randomly generated puzzles with puzzle state and true depth

list\_of\_easy\_puzzles = [

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

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

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

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

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

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

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

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

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

([2, 5, 3, 0, 7, 6, 1, 4, 8], 9),

]

list\_of\_medium\_puzzles = [

([2, 3, 5, 1, 4, 6, 0, 7, 8], 10),

([1, 4, 2, 7, 8, 3, 5, 0, 6], 11),

([1, 3, 6, 5, 0, 7, 4, 8, 2], 12),

([7, 2, 3, 0, 5, 6, 1, 4, 8], 13),

([1, 5, 0, 3, 2, 4, 7, 8, 6], 14),

([4, 2, 1, 0, 3, 6, 7, 5, 8], 15),

([1, 6, 7, 5, 0, 3, 4, 8, 2], 16),

([4, 8, 1, 0, 3, 5, 2, 7, 6], 17),

([7, 5, 3, 1, 4, 6, 2, 8, 0], 18),

([5, 4, 6, 3, 1, 2, 7, 0, 8], 19),

]

list\_of\_hard\_puzzles = [

([7, 1, 2, 4, 8, 5, 6, 3, 0], 20),

([3, 5, 4, 8, 7, 0, 2, 6, 1], 21),

([7, 1, 8, 5, 0, 3, 6, 4, 2], 22),

([1, 0, 8, 4, 7, 2, 3, 5, 6], 23),

([0, 7, 2, 4, 6, 1, 3, 5, 8], 24),

([5, 2, 1, 0, 8, 4, 7, 3, 6], 25),

([6, 3, 1, 4, 0, 7, 8, 2, 5], 26),

([4, 0, 7, 2, 6, 5, 8, 1, 3], 27),

([8, 6, 4, 2, 0, 7, 3, 1, 5], 28),

([5, 2, 1, 3, 8, 4, 6, 0, 7], 29),

([6, 4, 7, 8, 3, 5, 1, 2, 0], 30),

]

#############################################################

######## Utility Functions ########

#############################################################

def generate\_random\_states(state\_len):

'''

generate random states for evaulating and testing

'''

arr = [i for i in range(state\_len)]

np.random.shuffle(arr)

return arr

def validate\_state(problem\_state: list) -> bool:

'''

check if the state passed is valid or not.

checks that state length is a perfect sqaure.

checks that all numbers are unique.

checks that numbers are in range 0-len(state)

'''

# get length of puzzle

# is it 8 puzzle, 15 puzzle etc

puzzle\_size = len(problem\_state)

# check that length is perfect square

sqrt = int(math.sqrt(puzzle\_size))

if(sqrt\*sqrt) != puzzle\_size:

return False

# generate valid inputs.

# valid inputs range from 0 - n for n puzzle

valid\_inputs = [i for i in range(puzzle\_size)]

# put the problem state in a set to remove duplicates.

problem\_state\_set = set(problem\_state)

for i in problem\_state\_set:

if i not in valid\_inputs:

return False

return True

def print\_formatted\_time(time\_input):

'''

funtion to take in seconds as input and

print in Hours, minutes, and seconds

'''

hrs = int(time\_input // 3600)

mins = int((time\_input % 3600) // 60)

secs = int((time\_input % 3600) % 60)

if hrs:

print(f'time taken is {hrs} hrs, {mins} mins and {secs} secs')

elif mins:

print(f'time taken is {mins} mins and {secs} secs')

else:

print(f'time taken is {secs} secs')

def print\_time(time\_input):

'''

function to print the time with appropritate precision if between 0 and 1

else print in HH, MM, SS format

'''

if time\_input <= 1e-5:

print(f'time taken is {time\_input:.6f} secs')

elif time\_input <= 1e-4:

print(f'time taken is {time\_input:.5f} secs')

elif time\_input <= 1e-3:

print(f'time taken is {time\_input:.4f} secs')

elif time\_input <= 1e-2:

print(f'time taken is {time\_input:.3f} secs')

elif time\_input <= 1e-1:

print(f'time taken is {time\_input:.2f} secs')

elif time\_input >= 0 and time\_input <= 1:

print(f'time taken is {time\_input} secs')

else:

print\_formatted\_time(time\_input)

def print\_trace(node, goal\_state):

'''

take in any node and print the trace on how to reach this node from the parent node

'''

if node is None:

return

print\_trace(node.parent, goal\_state)

node.print\_trace\_info(goal\_state)

#############################################################

######## NODE CLASS AND CORRESPONDING FUNCTION ########

#############################################################

class Node():

'''

create a node class for the states.

stores path to current node from parent, depth etc and other properties.

has uitlity methods.

'''

# to store the states that have already been generated.

# prevents exploring repeating states.

# If a state is present here, then it has already been generated at higher depth

global\_states\_manager = {}

# initialize the node

# depth of node.

# path stores the path to be taken to reach till current node.

# state of node

# link to parent node

def \_\_init\_\_(self, depth, path, state, parent):

self.depth = depth

self.path = path

self.state = state

self.parent = parent

# puzzle length 8/15/24

self.state\_length = len(state)

# length per row

self.row\_length = int(math.sqrt(self.state\_length))

# length per column

self.col\_length = int(math.sqrt(self.state\_length))

# in case the current node is parent node, empty the globally stored states.

if self.parent is None:

Node.global\_states\_manager = {}

Node.global\_states\_manager[self.\_get\_state\_string(

self.state)] = self.depth

def spawn\_children(self):

'''

create child nodes after making valid moves on parent node.

children are not generated is their states are already present in global states manager

'''

# get index of blank tile

blank\_idx = self.state.index(0)

children\_list = []

for move in self.get\_valid\_moves():

state\_copy = cpy.deepcopy(self.state)

path = cpy.deepcopy(self.path)

if move == 'U':

path.append('U')

state\_copy[blank\_idx], state\_copy[blank\_idx -

self.row\_length] = state\_copy[blank\_idx-self.row\_length], state\_copy[blank\_idx]

elif move == 'L':

path.append('L')

state\_copy[blank\_idx], state\_copy[blank\_idx -

1] = state\_copy[blank\_idx-1], state\_copy[blank\_idx]

elif move == 'R':

path.append('R')

state\_copy[blank\_idx], state\_copy[blank\_idx +

1] = state\_copy[blank\_idx+1], state\_copy[blank\_idx]

elif move == 'D':

path.append('D')

state\_copy[blank\_idx], state\_copy[blank\_idx +

self.row\_length] = state\_copy[blank\_idx+self.row\_length], state\_copy[blank\_idx]

# check if the node is already generated.

past\_depth\_if\_generated = self.\_is\_state\_already\_generated(

state\_copy)

# if the node is never generated or if the node generated earlier has

# depth greater than current node, then generate another node with lower depth

if past\_depth\_if\_generated == -1 or past\_depth\_if\_generated > self.depth+1:

child\_node = Node(self.depth+1, path, state\_copy, self)

Node.global\_states\_manager[self.\_get\_state\_string(

child\_node.state)] = self.depth+1

children\_list.append(child\_node)

return children\_list

def get\_valid\_moves(self):

'''

get list of valid operators for each puzzle state

checks the position of blank and returns array of valid moves possible

'''

# total Valid moves.

# Move the blank space in following directions

# Up, Left, Right, Down

valid = ['U', 'L', 'R', 'D']

# get index of blank tile

blank\_idx = self.state.index(0)

# if the blank tile is in first row, cant move up

if blank\_idx >= 0 and blank\_idx < self.col\_length:

valid.remove('U')

# if the blank tile is in last row, cant move down

if blank\_idx >= (self.col\_length\*self.col\_length - self.col\_length) and blank\_idx < self.col\_length\*self.col\_length:

valid.remove('D')

# if the blank tile is in first column, cant move left

if blank\_idx % self.col\_length == 0:

valid.remove('L')

# if the blank tile is in last column, cant move right

if (blank\_idx + 1) % self.col\_length == 0:

valid.remove('R')

return valid

def manhattan\_distance\_heuristic(self, goal\_state):

'''

get value for manhattan distance for a current state and goal state

it is the shortest distance a tile needs to be moved to get to correct position

total distance is sum of all individual distances

does not include blank for calculation

'''

total\_manhattan\_distance = 0

for value in goal\_state:

if value == 0:

continue

goal\_state\_row, goal\_state\_colums = self.\_get\_row\_col\_position(

goal\_state, value)

random\_state\_row, random\_state\_colums = self.\_get\_row\_col\_position(

self.state, value)

total\_manhattan\_distance += abs(goal\_state\_colums-random\_state\_colums)+abs(

goal\_state\_row-random\_state\_row)

return int(total\_manhattan\_distance)

def misplaced\_tile\_heuristic(self, goal\_state):

'''

get value of misplaced tile heuristic for a current state and goal state.

it is the count of all the tiles that are not in correct position

does not include blank for calculation

'''

misplaced\_count = 0

for idx,value in enumerate(goal\_state):

if value == 0:

continue

if self.state[idx] != goal\_state[idx]:

misplaced\_count += 1

return misplaced\_count

def get\_heuristic\_cost(self, heuristic\_measure, goal\_state):

'''

get the heuristic value cost of expanding this node.

takes in heuristic measure and goal state.

return g(n) + h(n).

g(n) is the depth of node.

'''

g\_n = self.depth

h\_n = 0

if heuristic\_measure == MANHATTAN:

h\_n = self.manhattan\_distance\_heuristic(goal\_state)

elif heuristic\_measure == MISPLACED:

h\_n = self.misplaced\_tile\_heuristic(goal\_state)

elif heuristic\_measure == UNIFORM:

h\_n = 0

else:

h\_n = 0

return g\_n + h\_n

def print\_state(self, verbose=False):

'''

print the current node in puzzle view.

if verbose is True, also print the path to reach this node

and its depth

'''

if verbose:

print('depth', self.depth)

print('path', self.path)

self.\_print\_horizontal\_divider(self.state\_length)

for i in range(self.state\_length):

print(f'| {self.state[i]:2} |', end="")

if (i+1) % self.row\_length == 0:

self.\_print\_horizontal\_divider(self.state\_length)

print()

def print\_trace\_info(self, goal\_state):

'''

print the trace of the current node.

start from the parent and print out the heuristic code and cumulative cost.

print the path to take from parent to reach this node.

'''

if len(self.path):

print(

f'The best state to expand with g(n): {self.depth} and h(n): {self.manhattan\_distance\_heuristic(goal\_state)}')

print('Move blank to: ', DIRECTIONS\_MAP[self.path[-1]])

print('Updated State', end='')

else:

print('\nProblem State', end='')

self.\_print\_horizontal\_divider(self.state\_length)

for i in range(self.state\_length):

print(f'| {self.state[i]:2} |', end="")

if (i+1) % self.row\_length == 0:

self.\_print\_horizontal\_divider(self.state\_length)

print()

def \_get\_row\_col\_position(self, state, element):

'''

get row and column positions for a given element in a given state

used for manhattan distance

'''

idx = state.index(element)

column\_val = int(idx % self.row\_length)

r\_val = int(idx // self.row\_length)

return r\_val, column\_val

def \_print\_horizontal\_divider(self, size=8):

'''

print horizontal dividers after each row for better UI

'''

if size == 9:

print('\n------------------')

elif size == 16:

print('\n------------------------')

def \_is\_state\_already\_generated(self, state):

'''

check if the potential child state is already generated

return the depth if state exists, else return -1

'''

state\_string = "".join([str(i) for i in state])

return Node.global\_states\_manager[state\_string] if state\_string in Node.global\_states\_manager else -1

def \_get\_state\_string(self, state):

'''

convert the child node to a string for unique and easy representation

'''

state\_string = "".join([str(i) for i in state])

return state\_string

#############################################################

######## GENERAL SEARCH ########

#############################################################

def make\_queue(node: Node, goal\_state: list, heuristic\_measure: str):

'''

Initialize an empty queue.

take in a node and add it to the queue.

return the queue

'''

return [(node.get\_heuristic\_cost(heuristic\_measure, goal\_state), node)]

def is\_queue\_empty(queue: list):

'''

take in queue and check if the queue is empty

'''

return False if len(queue) > 0 else True

def remove\_front(queue: list):

'''

take in queue and sort it.

remove the first node.

return the first removed node.

'''

queue = sorted(queue, key=lambda x: x[0])

node = queue.pop(0)[1]

return queue, node

def expand\_nodes(node: Node):

'''

takes in node and operators and expands the node based on operators.

returns a list of nodes

'''

children = node.spawn\_children()

return children

def make\_node\_from\_state(state: list):

'''

call in the Node class to create Parent Node

'''

parent\_node = Node(0, [], state, None)

return parent\_node

def queueing\_function(queue, children, heuristic\_measure, goal\_state):

'''

take in queue,

take in children

put children in priority queue based on heuristic value

sort the queue

return the queue

'''

child\_queue = []

for child in children:

child\_queue.append((child.get\_heuristic\_cost(

heuristic\_measure, goal\_state), child))

queue = queue + child\_queue

queue = sorted(queue, key=lambda x: x[0])

return queue

def general\_search(initial\_state, goal\_state, queueing\_function, heuristic\_measure, verbose=False):

'''

# general search function

# refered from the problem statement doc provided.

# link to doc

# https://d1u36hdvoy9y69.cloudfront.net/cs-205-ai/Project\_1\_The\_Eight\_Puzzle\_CS\_205.pdf

# takes in problem state, goal state, queueing function and heuristic measure

# solves the problem using the queueing function

# returns final node, total nodes expanded, max queue size

'''

# create parent node

nodes = make\_queue(make\_node\_from\_state(initial\_state),

goal\_state, heuristic\_measure)

# store total nodes expanded count and max size of queue

total\_nodes\_expanded = 0

max\_queue\_size = 0

while True:

max\_queue\_size = max(max\_queue\_size, len(nodes))

if is\_queue\_empty(nodes):

print("FAILURE")

return -1, total\_nodes\_expanded, max\_queue\_size

else:

nodes, node = remove\_front(nodes)

if goal\_state == node.state:

if verbose:

print("SUCCESS")

node.print\_state(True)

print('Total nodes Expanded : ', total\_nodes\_expanded)

print('Max Queue Size : ', max\_queue\_size)

return node, max\_queue\_size, total\_nodes\_expanded

else:

total\_nodes\_expanded += 1

nodes = queueing\_function(nodes, expand\_nodes(

node), heuristic\_measure, goal\_state)

# #############################################################

# ######## UI LANDING PAGE AND INPUT VALIDATION ########

# #############################################################

def main\_block(clear\_previous=True):

'''

Print out the Landing Page.

Get Algo Choice From user

Get Puzzle Choice From user

Get Goal State From user

Search the goal state

Print Traceback

'''

############### Clear screen for First View ###############

if clear\_previous:

os.system('cls')

################# GET ALGO CHOICE #################

print('---- N Puzzle Solver ----')

print('1. Uniform Cost Search')

print('2. A\* with Misplaced Tile')

print('3. A\* with Manahattan Distance')

algo\_choice = int(input('Enter choice: '))

if algo\_choice not in [1, 2, 3]:

os.system('cls')

print('Please enter correct choice.\n')

main\_block(clear\_previous=False)

return

################ GET PUZZLE CHOICE ################

print('\n---- Choose Puzzle Type ----')

print('1. Random Default Easy (depth 0-9)')

print('2. Random Default Medium (depth 10-19)')

print('3. Random Default Hard (depth >20)')

print('4. Custom Puzzle')

puzzle\_choice = int(input('Enter choice: '))

if puzzle\_choice not in [1, 2, 3, 4]:

os.system('cls')

print('Please enter correct choice.\n')

main\_block(clear\_previous=False)

return

################ INPUT CUSTOM PUZZLE ################

problem\_state = []

if puzzle\_choice == 1:

problem\_state = list\_of\_easy\_puzzles[np.random.choice(len(list\_of\_easy\_puzzles))][0]

elif puzzle\_choice == 2:

problem\_state = list\_of\_medium\_puzzles[np.random.choice(len(list\_of\_medium\_puzzles))][0]

elif puzzle\_choice == 3:

problem\_state = list\_of\_hard\_puzzles[np.random.choice(len(list\_of\_hard\_puzzles))][0]

elif puzzle\_choice == 4:

print('\nEnter the numbers in puzzle as space seperated list.')

print('Represent blank with 0')

print('For Example: 1 2 3 4 0 5 6 7 8\n')

problem\_input = input('Numbers: ')

problem\_state = problem\_input.split(' ')

# convert string to integers

problem\_state = [int(i) for i in problem\_state]

if not validate\_state(problem\_state):

os.system('cls')

print('Pleae enter valid input state.\n')

main\_block(clear\_previous=False)

return

################ GET GOAL STATE ################

print('\n---- Enter Goal State ----')

print('1. Default State (1 2 3 4 5 6 ... n-1 n 0)')

print('2. Custom Goal')

puzzle\_choice = int(input('Enter choice: '))

if puzzle\_choice not in [1, 2]:

os.system('cls')

print('Please enter correct choice.\n')

main\_block(clear\_previous=False)

return

################ GET GOAL STATE ################

goal\_state = []

if puzzle\_choice == 1:

goal\_state = list(range(1, len(problem\_state)))

goal\_state.append(0)

elif puzzle\_choice == 2:

print('\nEnter the numbers in goal as space seperated list.')

print('Represent blank with 0')

print('For Example: 1 2 3 4 0 5 6 7 8\n')

problem\_input = input('Numbers: ')

goal\_state = problem\_input.split(' ')

# convert string to integers

goal\_state = [int(i) for i in goal\_state]

if not validate\_state(goal\_state):

os.system('cls')

print('Pleae enter valid input state.\n')

main\_block(clear\_previous=False)

return

################ FIND GOAL STATE USING SEARCH ################

os.system('cls')

print('Initial State\n')

parent\_node = Node(0, [], problem\_state, None)

parent\_node.print\_state()

if algo\_choice == 1:

print('\nSolving for Uniform cost\n')

time\_before = time.time()

final\_node, \_, \_ = general\_search(problem\_state, goal\_state, queueing\_function, UNIFORM, verbose=True)

time\_after = time.time()

total\_time = time\_after - time\_before

print\_time(total\_time)

elif algo\_choice == 2:

print('\nSolving for A\* with Misplaced Tile\n')

time\_before = time.time()

final\_node, \_, \_ = general\_search(problem\_state, goal\_state, queueing\_function, MISPLACED, verbose=True)

time\_after = time.time()

total\_time = time\_after - time\_before

print\_time(total\_time)

elif algo\_choice == 3:

print('\nSolving for A\* with Manahattan Distance\n')

time\_before = time.time()

final\_node, \_, \_ = general\_search(problem\_state, goal\_state, queueing\_function, MANHATTAN, verbose=True)

time\_after = time.time()

total\_time = time\_after - time\_before

print\_time(total\_time)

################ PRINT TRACEBACK ################

print('\n---- Want to print the puzzle traceback? ----')

print('1. Yes')

print('2. No. Exit')

traceback\_choice = int(input('Enter choice: '))

if traceback\_choice not in [1, 2]:

os.system('cls')

print('Please enter correct choice.\n')

main\_block(clear\_previous=False)

return

if traceback\_choice == 1:

print\_trace(final\_node, goal\_state)

return

main\_block()

# #############################################################

# ######## Multiple TESTS and Result Analysis ########

# #############################################################

# code to run all the puzzles and generate graphs

def run\_analysis(combined\_puzzles\_list):

'''

runs analysis on all list of puzzles provided.

solves the puzzles using all three algorithms

stores the time taken per algorithm per puzzle

stores the max queue size

stores the total nodes expanded

Plots the results in graphs

'''

goal\_state = [1,2,3,4,5,6,7,8,0]

time\_collection = {}

queue\_collection = {}

nodes\_collection = {}

for heuristic in [MANHATTAN, UNIFORM, MISPLACED]:

time\_collection[heuristic] = []

queue\_collection[heuristic] = []

nodes\_collection[heuristic] = []

for puzzle, true\_depth in combined\_puzzles\_list:

for heuristic in [MANHATTAN, UNIFORM, MISPLACED]:

print (heuristic, puzzle, true\_depth)

time\_before = time.time()

final\_node, max\_queue, total\_nodes = general\_search(puzzle, goal\_state, queueing\_function, heuristic, verbose=False)

time\_after = time.time()

total\_time = time\_after - time\_before

print (final\_node.depth)

time\_collection[heuristic].append((true\_depth, total\_time))

queue\_collection[heuristic].append((true\_depth, max\_queue))

nodes\_collection[heuristic].append((true\_depth, total\_nodes))

print ()

plt.figure(1)

for heuristic in [MANHATTAN, UNIFORM, MISPLACED]:

temp\_arr = np.array(time\_collection[heuristic])

plt.plot(temp\_arr[:,0], temp\_arr[:,1], color = COLOR\_MAP[heuristic], label=heuristic)

plt.title('Time vs Depth')

plt.xlabel('depth')

plt.ylabel('time in seconds')

plt.grid()

plt.legend()

plt.show()

plt.figure(2)

for heuristic in [MANHATTAN, UNIFORM, MISPLACED]:

temp\_arr = np.array(nodes\_collection[heuristic])

plt.plot(temp\_arr[:,0], temp\_arr[:,1], color = COLOR\_MAP[heuristic], label=heuristic)

plt.title('Nodes Expanded vs Depth')

plt.xlabel('depth')

plt.ylabel('Nodes Expanded')

plt.grid()

plt.legend()

plt.show()

plt.figure(3)

for heuristic in [MANHATTAN, UNIFORM, MISPLACED]:

temp\_arr = np.array(queue\_collection[heuristic])

plt.plot(temp\_arr[:,0], temp\_arr[:,1], color = COLOR\_MAP[heuristic], label=heuristic)

plt.title('Max Queue Size vs Depth')

plt.xlabel('depth')

plt.ylabel('Max Queue Size')

plt.grid()

plt.legend()

plt.show()

# this will take ~2hrs to run

# run\_analysis(list\_of\_easy\_puzzles + list\_of\_medium\_puzzles + list\_of\_hard\_puzzles)

#############################################################

######## Generating puzzles at different depths ########

#############################################################

# code to generate random puzzles and solving them to find the depth of puzzles.

# used to create multiple puzzles to test the algorithms and for benchmarking

# the original method to let the algorithm run till all nodes are explored to get failure takes a lot of time.

# therefore using a constant time algorithm to generate puzzles.

# code is taken from

# https://www.geeksforgeeks.org/check-instance-8-puzzle-solvable/

def get\_int\_count(arr):

'''

counts the total number of inversions to be made

'''

inv\_count = 0

empty\_value = 0

for i in range(0, 9):

for j in range(i + 1, 9):

if arr[j] != empty\_value and arr[i] != empty\_value and arr[i] > arr[j]:

inv\_count += 1

return inv\_count

def is\_solvable(puzzle):

'''

checks if the 8 puzzle is solvable or not

'''

# Count inversions in given 8 puzzle

inv\_count = get\_int\_count(puzzle)

# return true if inversion count is even.

return (inv\_count % 2 == 0)

def create\_puzzle\_book(n\_iters):

'''

create a puzzle book that stores a list of problem states are various depths

'''

puzzle\_book = {}

for \_ in range(n\_iters):

# generate a random 8 puzzle

random\_state = generate\_random\_states(9)

# check if the puzzle is solvable

if (is\_solvable(random\_state)):

# if solvable, solve the puzzle to get depth

goal\_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]

final\_node, \_, \_ = general\_search(

random\_state, goal\_state, queueing\_function, MANHATTAN, verbose=False)

# store the puzzle at appropriate depth entry

if final\_node.depth not in puzzle\_book:

puzzle\_book[final\_node.depth] = []

puzzle\_book[final\_node.depth].append(random\_state)

return puzzle\_book

# print(create\_puzzle\_book(30))

# #############################################################

# ######## Pretty Print Puzzles to display in Report ########

# #############################################################

def pretty\_print\_puzzles(combined\_puzzles):

'''

prints the puzzles in matrix form so they can be added to report

'''

for puzzle, true\_depth in combined\_puzzles:

print (f' --- {true\_depth} ---')

print (' ', puzzle[:3])

print (' ', puzzle[3:6])

print (' ', puzzle[6:9])

print ()

print ()

# pretty\_print\_puzzles(list\_of\_easy\_puzzles + list\_of\_medium\_puzzles + list\_of\_hard\_puzzles)

# pretty\_print\_puzzles(list\_of\_easy\_puzzles)