Wilfrid Laurier University

Assignment 1

8 Puzzle

October 16, 2022

Nahor Yirgaalem - 190775540 - yirg5540@mylaurier.ca

Shaheer Khan - 190693830 - khan3830@mylaurier.ca

Duong Truong - 190695030 - truo9503@mylaurier.ca

Sameer Mian - 190366140 - mian6140@mylaurier.ca

Memet Rusidovski - 130951550 - rusi1550@mylaurier.ca

CP468: Artificial Intelligence

Dr. Illias kotsireas

CP468 8 Puzzle Assignment

Iteration and expanded node are equivalent in this report

8 Puzzle

Averages of H1: Misplaced Tiles

Iterations	Moves	Time	Number of Puzzles
318	15	500 ms	15
2500	20	3 sec	30
6500	22	13 sec	40
11,000	24	33 sec	13
200,000	31	45 sec	2

Averages of H2: Manhattan Distance

Iterations	Moves	Time	Number of Puzzles
141	18	300 ms	15
375	20	400 ms	30
450	22	500 ms	40
1000	24	2.2 sec	13
9000	31	17 sec	2

Averages of H3: Euclidean Distance

Iterations	Moves	Time	Number of Puzzles
160	16	250 ms	15
310	18	270 ms	30
1300	22	600 ms	40
2000	24	1 sec	13
37000	31	100 sec	2

15 Puzzle

Averages of H1: Misplaced Tiles

Iterations	Moves	Time	Number of Puzzles
80	15	150 ms	3
1,700	21	250 ms	20
7,000	22	450 ms	45
14,000	24	840 ms	30
420,000	28	23 sec	2

Averages of H2: Manhattan Distance

Iterations	Moves	Time	Number of Puzzles
182	21	200 ms	20
11,800	28	4 sec	75
196,000	48	66 sec	2
490,000	56	250 sec	2
560,000	66	265 sec	1

Averages of H3: Euclidean Distance

Iterations	Moves	Time	Number of Puzzles
180	19	325 ms	30
600	22	700 ms	40
1800	24	7 sec	26
12,000	28	27 sec	3
43,000	32	44 sec	1

24 Puzzle

Averages of H1: Misplaced Tiles

Iterations	Moves	Time	Number of Puzzles
38	11	200 ms	17
8400	19	500 ms	40
11,300	20	1.2 sec	40
156,000	25	9.6	2
180,000	26	10.9 sec	1

Averages of H2: Manhattan Distance

Iterations	Moves	Time	Number of Puzzles
70	22	150 ms	17
233	25	350 ms	40
850	27	500 ms	40
60,000	32	31 sec	2
196000	48	66 sec	1

Averages of H3: Euclidean Distance

Iterations	Moves	Time	Number of Puzzles
100	20	300	17
600	22	500 ms	40
3,300	27	1.9 sec	40
184,000	30	100 sec	2
195,000	36	108 sec	1

Comparing H1, H2 & H3

Out of all heuristics the one that requires the least number of iterations is H2. The first heuristic is better than a blind search such as Breadth First Search (BFS), but requires many more iterations to get an answer. The run time of H1 is much faster since time is saved on calculating the cost. These savings in computing time are negated by the increased memory and iterations required. H3 performs similarly to H2, but H2 requires less iterations on average. The adding cost of squaring and taking the root for H3 does not increase the runtime. Most runtime differences came from optimizing the Python code, Iterations were still higher in H3.

Our Third Heuristic

Our third heuristics is the Euclidean Distance. This is calculated by taking the horizontal and vertical number of moves for a tile to be put in the right place. Each number is squared and added together. Then the square root is taken which gives us our H3 value.

Why is h3 an admissible heuristic?

The euclidean distance never over-estimates the cost of moving a tile to its correct spot because by Triangle Inequality the distance/cost given by H3 will always be less than the number of moves it would take to get to its spot. For example, if a tile needs to move one to the left and one up the euclidean distance will be sqrt(2) which is always less than the total moves needed of 2. At worst the tile needs to move in a straight line and in that case euclidean distance and moves are equal, but still making the heuristic admissible.

```python
from puzzles import Puzzle8
from copy import deepcopy
from queue import PriorityQueue
'''
h1 = the number of misplaced tiles. For Figure 3.28,
all of the eight tiles are out of position, so the
start state would have h1 = 8. h1 is an admissible
heuristic because it is clear that any tile that is
out of place must be moved at least once.
'''
"""
puzzles =[]
for i in range(100):
    puzzles.append(Puzzle8.Puzzle())
"""


q = PriorityQueue()
explored = {""}
cost = 0
y = Puzzle8.Puzzle()
z = Puzzle8.Puzzle(shuffle=True)



x = y

if False:
    x.puzzle = [8, 6, 7, 2, 5, 4, 3, 0, 1]#[5, 1, 4, 6, 3, 8, 0, 7, 2]  # [3, 6, 2, 5, 0, 7,
4, 1, 8]
    x.distCheck()
    x.findIndex()

explored.add(str(x.puzzle))

while x._dist != 0 and cost < 2000000:
    up = deepcopy(x)
    down = deepcopy(x)
    left = deepcopy(x)
    right = deepcopy(x)

    x1 = up.up()
    x2 = down.down()
    x3 = left.left()
    x4 = right.right()

    if x1 and str(up.puzzle) not in explored:
        q.put(up)
        explored.add(str(up.puzzle))
        up.parent_node = x
        #print(up, up.puzzle, "up", up._index, "index", "\n\n")
    if x2 and str(down.puzzle) not in explored:
        q.put(down)
        explored.add(str(down.puzzle))
        down.parent_node = x
        #print(down, down.puzzle, "down", down._index, "index", "\n\n\n")
    if x3 and str(left.puzzle) not in explored:
        q.put(left)
        explored.add(str(left.puzzle))
        left.parent_node = x
        #print(left, left.puzzle, "left", left._index, "index", "\n\n\n")
    if x4 and str(right.puzzle) not in explored:
        q.put(right)
        explored.add(str(right.puzzle))
        right.parent_node = x
        #print(right, right.puzzle, "right", right._index, "index", "\n\n")
```

```
    x = q.get()
    x._globalCost += 1

    if cost % 100 == 0:
        print(cost)
    #print(x._dist, " -------", x._globalCost)
    cost += 1


temp = x
lst = []
while temp.parent_node != None:
    lst.append(temp)
    temp = temp.parent_node

for i in lst:
    print(i)


print(x._globalCost)
print(cost)
```

```python
import math
import random
import numpy as np

'''
Zero is the place holder for the empty square.
the matrix is divided equally so,
[1,2,3,4,5,6,7,8, 0] =>

    __________
    |1 | 2| 3|
    |4 | 5| 6|
    |7 | 8| 0|
    ~~~~~~~~~~
'''


class Puzzle:

    def __init__(self, size=3, shuffle=True, manhat=False, ecd=False):
        self.size = size
        self.puzzle = []  # [1, 2, 3, 4, 5, 6, 7, 8, 0]
        self.createPuz(size)
        self._index = 8
        self._dist = 0
        self._solved = False
        self._globalCost = 0
        self.parent_node = None
        self._manhat=manhat
        self._ecd = ecd

        if(shuffle):
            self.scramble()
            self.distCheck()

    def createPuz(self, size):
        for x in range(1, size*size):
            self.puzzle.append(x)
        self.puzzle.append(0)

    def __str__(self):
        return "_____________\n| {0} | {1} | {2} |\n" \
            "| {3} | {4} | {5} |\n| {6} | {7} | {8} |\n~~~~~~~~~~~~~".format(
                *self.puzzle)

    def findIndex(self):
        i = 0
        for x in range(9):
            if self.puzzle[x] == 0:
                i = x
                self._index = i
            #print(self.puzzle[x], "{\}", end="")

        return i

    def scramble(self):
        random.shuffle(self.puzzle)
        self.findIndex()

    def distCheck(self):
        dist = 0
        if self._manhat:
            g1 = np.asarray(self.puzzle).reshape(3, 3)
            g2 = np.asarray([1, 2, 3, 4, 5, 6, 7, 8, 0]).reshape(3, 3)

            for i in range(8):
                a, b = np.where(g1 == i+1)
                x, y = np.where(g2 == i+1)
```

```
                dist += abs((a-x)[0])+abs((b-y)[0])

        if self._ecd:
            g1 = np.asarray(self.puzzle).reshape(3, 3)
            g2 = np.asarray([1, 2, 3, 4, 5, 6, 7, 8, 0]).reshape(3, 3)

            for i in range(8):
                a, b = np.where(g1 == i+1)
                x, y = np.where(g2 == i+1)
                dist +=  math.sqrt((abs((a-x)[0]) ** 2) + (abs((b-y)[0]) ** 2))
        else:
            for i, j in zip(self.puzzle, range(9)):
                if i != (j + 1) and (i != 0 ):
                    dist += 1

        self._dist = dist
        return dist

    def up(self):
        if(0 in self.puzzle[((self.size ** 2)-self.size):]):
            #print("in bottom: invalid")
            return False
        else:
            self.puzzle[self._index], self.puzzle[self._index +
                                                  3] = self.puzzle[self._index + 3],
self.puzzle[self._index]
            self.distCheck()
            self.findIndex()
            #print(self._index,"......")
            return True

    def down(self):
        if(0 in self.puzzle[0:self.size]):
            #print("in top: invalid")
            return False
        else:
            self.puzzle[self._index], self.puzzle[self._index -
                                                  3] = self.puzzle[self._index - 3],
self.puzzle[self._index]
            self.distCheck()
            self.findIndex()
            return True

    def right(self):
        if (self._index != 0 and self._index != 3 and self._index != 6):
            #swap the index to the left
            self.puzzle[self._index], self.puzzle[self._index -
                                                  1] = self.puzzle[self._index - 1],
self.puzzle[self._index]
            self.distCheck()
            self.findIndex()
            return True
        else:
            #print("Invalid Move")
            return False

    def left(self):
        if (self._index != 2 and self._index != 5 and self._index != 8):
            #swap the index to the right
            self.puzzle[self._index], self.puzzle[self._index +
                                                  1] = self.puzzle[self._index + 1],
self.puzzle[self._index]
            self.distCheck()
            self.findIndex()
            return True
        else:
            #print("Invalid Move")
```

```python
            return False

    def __iter__(self):
        for v in self.puzzle:
            yield v

    def __lt__(self, obj):
        return (self._dist + self._globalCost) < (obj._dist + obj._globalCost)


'''
Testing
x = Puzzle()


print(x.findIndex())

print(x)
x.down()
print(x)

'''
```