Project 1 CS 170. Introduction to Artificial Intelligence

Nathanael Mueller
ID 861237712
nmuel001@ucr.edu
Instructor: Dr Eamonn Keogh
6-November-2019

In completing this assignment, I consulted…

- Google images for some example starting puzzles that are solvable.

All of the algorithms and code for the classes is original. Some code that is not original…

- The code in main for the prompts and entering a custom puzzle is copied from the sample report.

CS170 Assignment 1 Write Up

Nathanael Mueller, SID: 861237712

Introduction

This assignment is the first programming assignment in Dr. Eamonn Keogh's Introduction to AI course at University of California, Riverside in Fall quarter 2019. In this assignment I solved various versions of the 8 tile puzzle using A* search with no heuristic (uniform cost search), the misplaced tile heuristic and Manhattan distance. The following writeup is to detail my findings.

The full code for the project can be found at https://github.com/c0rv0s/cs170_project1.

Comparison of Algorithms

The three algorithms used are Uniform Cost Search, A* with the Misplaced Tile heuristic and A* with the Manhattan Distance heuristic.

Uniform Cost Search

Uniform Cost Search is merely A* where *h(n)* is hardcoded to 0. The first node in the queue is always the one popped off and there is no further calculations made. Every branch has a cost of 1.

Misplaced Tile Heuristic

This heuristic counts the number of misplaced tiles and considers this the distance to the goal state. It then adds that to the number of steps to reach the current state to get the total distance from start to finish for that node. The node with the shortest distance is the one that gets popped from the queue and expanded.

Manhattan Distance Heuristic

Manhattan Distance functions the same as the Misplaced Tile heuristic except that instead of just counting the number of tiles out of place it counts the number of moves that tile would have to make to reach its goal state. After this distance has been counted for all tiles in the node, the length of steps needed to reach this state is added to create the heuristic and, as before, the node with the lowest value is popped and expanded.

Comparison of Algorithms

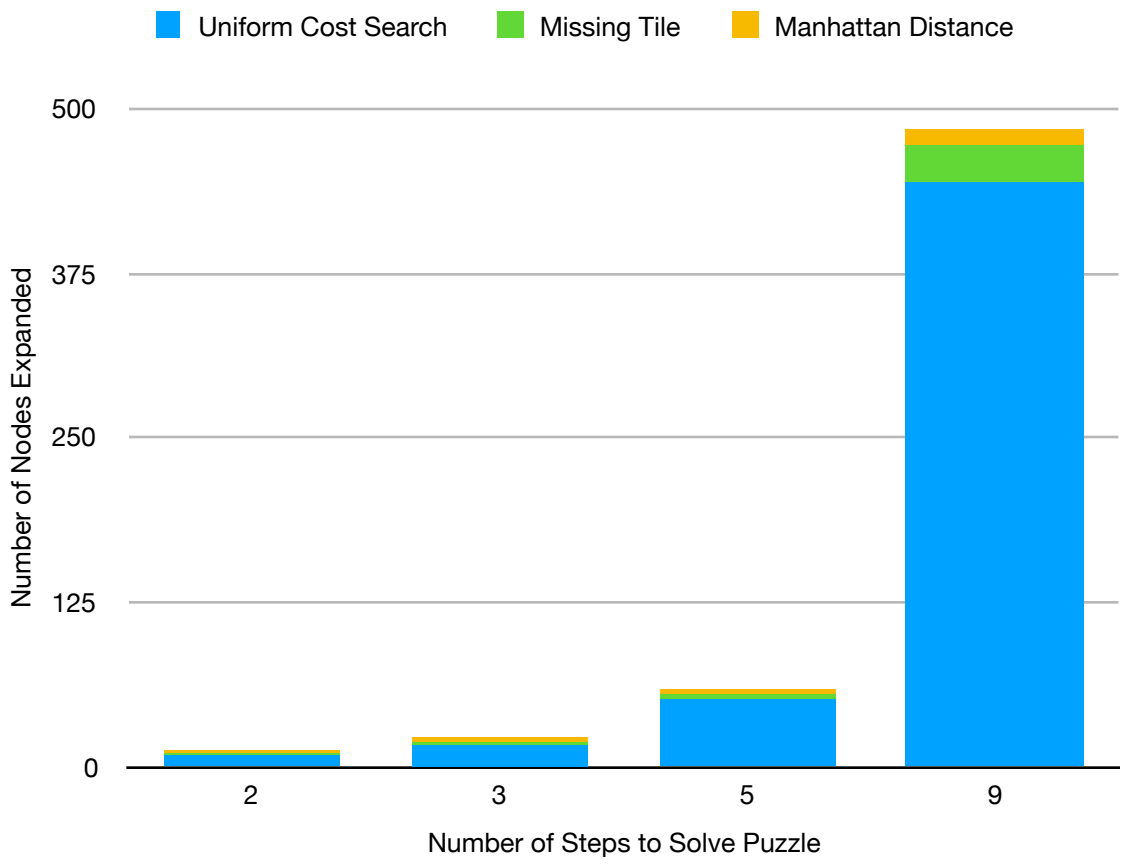
In general, Uniform Cost Search had the highest space and time complexity with the most nodes expanded and the largest maximum queue. The Misplaced Tile heuristic came next, and Manhattan Distance had the least. For easy puzzles - ones that could be solved in just a few steps - the difference was not discernible, it finished instantly. For tougher puzzles, including one that has twenty steps, the difference is significant.

The puzzles I tested on can be seen at the top of the code.

Charts/Graphs

Number of Nodes Expanded

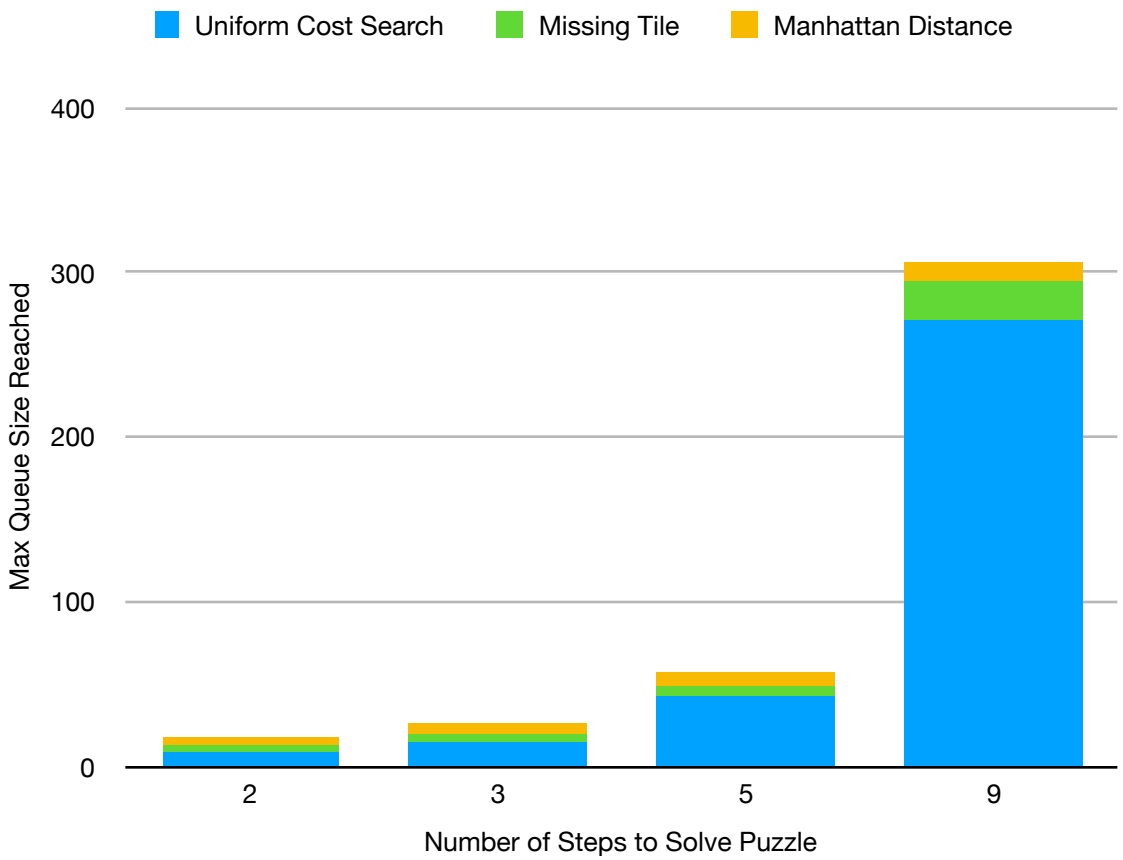
Steps to Solve	Uniform Cost Search	Missing Tile	Manhattan Distance
2	8	2	2
3	16	3	3
5	50	5	5
9	444	29	11
20	91712	4480	454



The 20 step puzzle has been omitted from the graph as it would throw off the scale so much as to make the graph unreadable.

Max Queue Size

Steps to Solve	Uniform Cost Search	Missing Tile	Manhattan Distance
2	8	5	5
3	14	6	6
5	42	7	7
9	270	25	12
20	37789	2711	282



The 20 step puzzle has been omitted from the graph as it would throw off the scale so much as to make the graph unreadable.

Conclusion

Considering the three algorithms that were used in this project, Uniform Cost Search, Misplaced Tiles, and Manhattan Distance, it can be said that:

- Uniform Cost Search is vastly less efficient than A* with either the Misplaced Tile or Manhattan Distance heuristic.
- Manhattan Distance performs much better than Misplaced Tile, but not as much better as Misplaced Tile performs over Uniform Cost Search.
- Puzzles take exponentially longer to solve the more steps that are required.
- All three algorithms perform fairly similarly for very easy to solve puzzles but Uniform Cost Search increases in space and time much faster than Misplaced Tiles or Manhattan Distance.
- Branching factor is on average 2.67.
- Uniform Cost Search simply expands nodes in the order they are discovered making it the same as a basic breadth first search. This makes its time and space complexity both $O(b^d)$ where b is the branching factor and d is the depth of the solution.

Example Traceback

```
Welcome to Nathan Mueller's 8-puzzle solver.
Type '1' to use a default puzzle, or '2' to enter your own.
1
Select a difficulty (0-4, or 5, which is impossible to solve)
1
Puzzle is:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]


Enter your choice of algorithm
   1. Uniform Cost Search
   2. A* with the Misplaced Tile heuristic
   3. A* with the Manhattan distance heuristic
3
Expanding state
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]


The best state to expand with a g(n) = 1 and h(n) = 2 is...

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


The best state to expand with a g(n) = 2 and h(n) = 3 is...

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


Goal!!
Goal state reached in 3 steps:
 ['right', 'down', 'right']
Nodes expanded: 3 Max queue size: 6
```

Code

```
#eight puzzle solver
import random
import copy
import sys
import matplotlib as plt

#some useful globals
ops = ['up','down','left','right']
goal = [[1,2,3],
        [4,5,6],
        [7,8,0]]
goal_indices = [[0,0],[0,0],[0,1],[0,2],[1,0],[1,1],[1,2],[2,0],[2,1]]
puzzles = {
"0": [[1,2,3],  #2 steps
      [4,0,6],
      [7,5,8]],
"1": [[1,2,3],  #3 steps
      [0,4,6],
      [7,5,8]],
"2": [[2,0,3],  #5 steps
      [1,4,6],
      [7,5,8]],
"3": [[1,8,2],  #9 steps
      [0,4,3],
      [7,6,5]],
"4": [[7,6,2],  #20 steps
      [5,0,1],
      [4,3,8]],
"impossible": [[1,2,3],
               [4,5,6],
               [8,7,0]]
}

#Puzzle class holds current arrangement of nummbers and the position of the zero
class Puzzle:
    def __init__(self, startPostition):
        self.puzzle = startPostition
        self.zero = [-1,-1]
        for i in range(len(startPostition)):
            if self.zero[0] >= 0:
                break
            for j in range(len(startPostition[i])):
                if startPostition[i][j] == 0:
                    self.zero = [i,j]
                    break

    def solved(self):
        return self.puzzle == goal

    def print(self):
        for i in range(len(self.puzzle)):
            print(self.puzzle[i])
        print('\n')

#each Node represents a current state of the puzzle and the steps used to get there
class Node:
    def __init__(self, puzzle, steps):
        self.puzzle = puzzle
        self.steps = steps

#move the blank tile in the specified direction and return as a new node
```

```
#return False for illegal moves
def move(n,direction):
    #check for illegal moves
    if direction not in ops:
        return False
    max_size = len(n.puzzle.puzzle) - 1
    if (direction == 'up' and n.puzzle.zero[0] == 0) or (direction == 'down' and
n.puzzle.zero[0] == max_size) or (direction == 'left' and n.puzzle.zero[1] == 0) or
(direction == 'right' and n.puzzle.zero[1] == max_size):
        return False
    #perform move and return as a new node
    x = n.puzzle.zero[0]
    y = n.puzzle.zero[1]
    new_steps = copy.deepcopy(n.steps)
    new_steps.append(direction)
    new_puzzle = Puzzle(copy.deepcopy(n.puzzle.puzzle))
    if direction == 'up':
        new_puzzle.puzzle[x][y], new_puzzle.puzzle[x-1][y] = new_puzzle.puzzle[x-1]
[y], new_puzzle.puzzle[x][y]
        new_puzzle.zero = [x-1,y]
    if direction == 'down':
        new_puzzle.puzzle[x][y], new_puzzle.puzzle[x+1][y] = new_puzzle.puzzle[x+1]
[y], new_puzzle.puzzle[x][y]
        new_puzzle.zero = [x+1,y]
    if direction == 'left':
        new_puzzle.puzzle[x][y], new_puzzle.puzzle[x][y-1] = new_puzzle.puzzle[x]
[y-1], new_puzzle.puzzle[x][y]
        new_puzzle.zero = [x,y-1]
    if direction == 'right':
        new_puzzle.puzzle[x][y], new_puzzle.puzzle[x][y+1] = new_puzzle.puzzle[x]
[y+1], new_puzzle.puzzle[x][y]
        new_puzzle.zero = [x,y+1]
    return Node(new_puzzle, new_steps)

#search algorithm
def UniformCostSearch(n, heuristic):
    max_queue_size = 0
    visited = []
    leaves = []
    print("Expanding state ")
    n.puzzle.print()
    while(not n.puzzle.solved()):
        visited.append(copy.deepcopy(n.puzzle.puzzle))
        for o in ops:
            s = move(n,o)
            if s and s.puzzle.puzzle not in visited:
                leaves.append(copy.deepcopy(s))

        # default value, uniform cost heuristic just pops first element
        index = hn = gn = 0
        shortest = sys.maxsize

        for i in range(len(leaves)):
            hn = 0
            gn = len(leaves[i].steps)
            if heuristic != "1":
                for x in range(len(leaves[i].puzzle.puzzle)):
                    for y in range(len(leaves[i].puzzle.puzzle[x])):
                        #count number of tiles out of place
                        if heuristic == "2":
                            if leaves[i].puzzle.puzzle[x][y] != 0 and
leaves[i].puzzle.puzzle[x][y] != goal[x][y]:
                                hn += 1
                        #count manhattan distance for each tile out of place
```

```
                        elif heuristic == "3":
                            num = leaves[i].puzzle.puzzle[x][y]
                            if num != 0:
                                hn += abs(x-goal_indices[num][0])
                                hn += abs(y-goal_indices[num][1])
                if hn + gn < shortest:
                    shortest = hn + gn
                    index = i

            #analytics book keeping
            if len(leaves) > max_queue_size:
                max_queue_size = len(leaves)

            n = leaves.pop(index)
            #traceback stuff
            traceback = True
            if not n.puzzle.solved() and traceback:
                print("The best state to expand with a g(n) = "+str(gn)+" and h(n) =
"+str(hn)+" is...\n")
                n.puzzle.print()
        print("Goal!!")
        return n.steps, len(visited), max_queue_size

#run the program, collect user input
def main():
    print("Welcome to Nathan Mueller's 8-puzzle solver.")

    p = Puzzle(puzzles[str(random.randint(0,4))])
    puzzle_choice = input("Type '1' to use a default puzzle, or '2' to enter your own.
\n")
    if puzzle_choice == "1":
        dif = input("Select a difficulty (0-4, or 5, which is impossible to solve)\n")
        if dif not in ["0","1","2","3","4","5"]:
            print("input not recognized")
            return False
        p = Puzzle(puzzles[dif])
    elif puzzle_choice == "2":
        print("Enter your puzzle, using a zero to represent the blank. " +
        "Please only enter valid 8-puzzles. Enter the puzzle demilimiting " +
        "the numbers with a space. RET only when finished.\n")
        puzzle_row_one = input("Enter the first row: ")
        puzzle_row_two = input("Enter the second row: ")
        puzzle_row_three = input("Enter the third row: ")
        puzzle_row_one = puzzle_row_one.split()
        puzzle_row_two = puzzle_row_two.split()
        puzzle_row_three = puzzle_row_three.split()
        for i in range(0, 3):
            puzzle_row_one[i] = int(puzzle_row_one[i])
            puzzle_row_two[i] = int(puzzle_row_two[i])
            puzzle_row_three[i] = int(puzzle_row_three[i])
        p = Puzzle([puzzle_row_one, puzzle_row_two, puzzle_row_three])
    else:
        print("input not recognized")
        return False
    print("Puzzle is: ")
    p.print()

    algo_choice = input("Enter your choice of algorithm\n   1. Uniform Cost Search\n
2. A* with the Misplaced Tile heuristic\n   3. A* with the Manhattan distance
heuristic\n")
    if algo_choice not in ["1","2","3"]:
        print("input not recognized")
        return False
```

```
    steps, nodes_expanded, max_queue_size = UniformCostSearch(Node(p,[]),algo_choice)
    print("Goal state reached in " + str(len(steps)) + " steps:\n", steps)
    print("Nodes expanded: " +str(nodes_expanded)+" Max queue size:
"+str(max_queue_size)+"\n")

if __name__== "__main__":
    main()
```