CS170 - Introduction to Artificial Intelligence:
The Eight Puzzle

Trung Lam
861270734

11/08/2021

Consultations during the completion of the project:

- Lecture Slides from Blind Search and Heuristic Search
- Numpy Documentations: https://numpy.org/doc/stable/reference/
- Python Documentations: https://docs.python.org/3/tutorial/

Non-original subroutines used in the project:

- Numpy: used mainly to assist in comparing states and matrix manipulation
- Copy: to assist in copying states for the purpose of correctly expanding states

Outline:

Introduction

The Eight Puzzle is a game where the objective is to move the tiles, initially in a random but solvable state, into the goal state. There are 8 tiles, each number 1 through 8, for example:

Program Description

Summary:

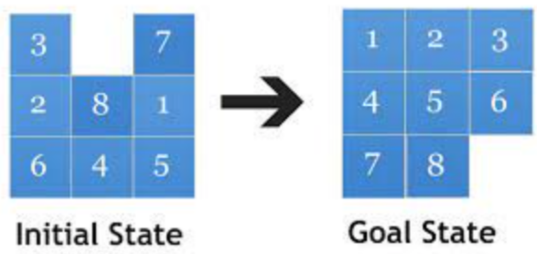
Given an initial input state given by the user, the program will solve the puzzle and return the number of nodes expanded based on one of the three algorithms: Uniform Cost Search, A* w/ Misplaced Tile Heuristic, and A* w/ Manhattan Distance Heuristic.

Uniform Cost Search:

An algorithm that expands the node based on solely the g(n) cost of the node. For this puzzle, each node has the same cost of 1. In other words, this algorithm would expand nodes in the order like breadth first search.

Misplaced Tile Heuristic:

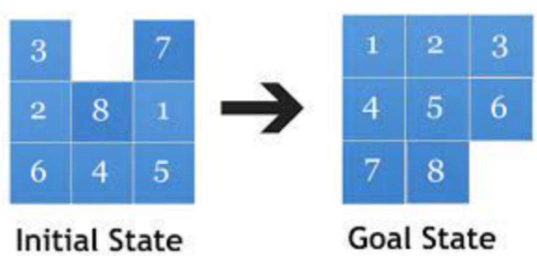
An algorithm that expands the node based on not only the g(n) cost, but also taking account the h(n) cost based on the number of tiles that are misplaced. For example:



The initial state would have h(n) = 8. And the goal would have h(n) = 0. Combining the h(n) with the g(n) would result in a cost f(n) used in the A* algorithm w/ Misplaced Tile Heuristic.

Manhattan Heuristic:

An algorithm that expands the node based on not only the g(n) cost, but also taking account the h(n) cost based on the number of moves it takes to move a single tile to it's goal position. For example, using the previous figure as reference:

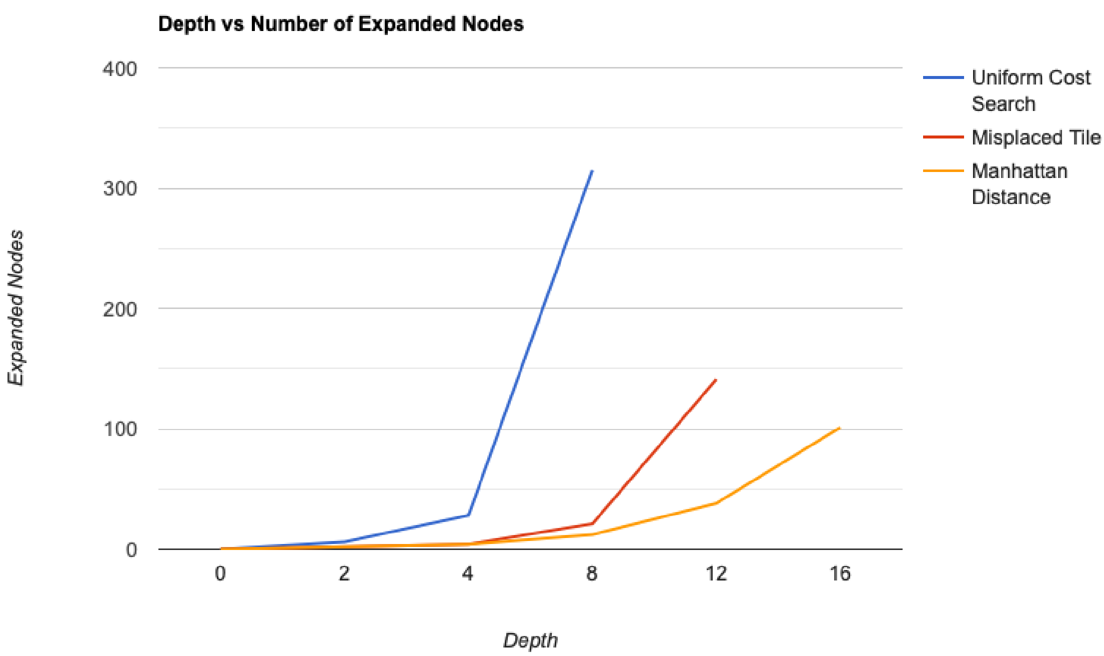


moving the 3 tile would cost 2 moves or h(n) = 2, 7 would cost 4 moves or h(n) = 4, and so on. Adding them together results in the h(n) cost for that particular state.

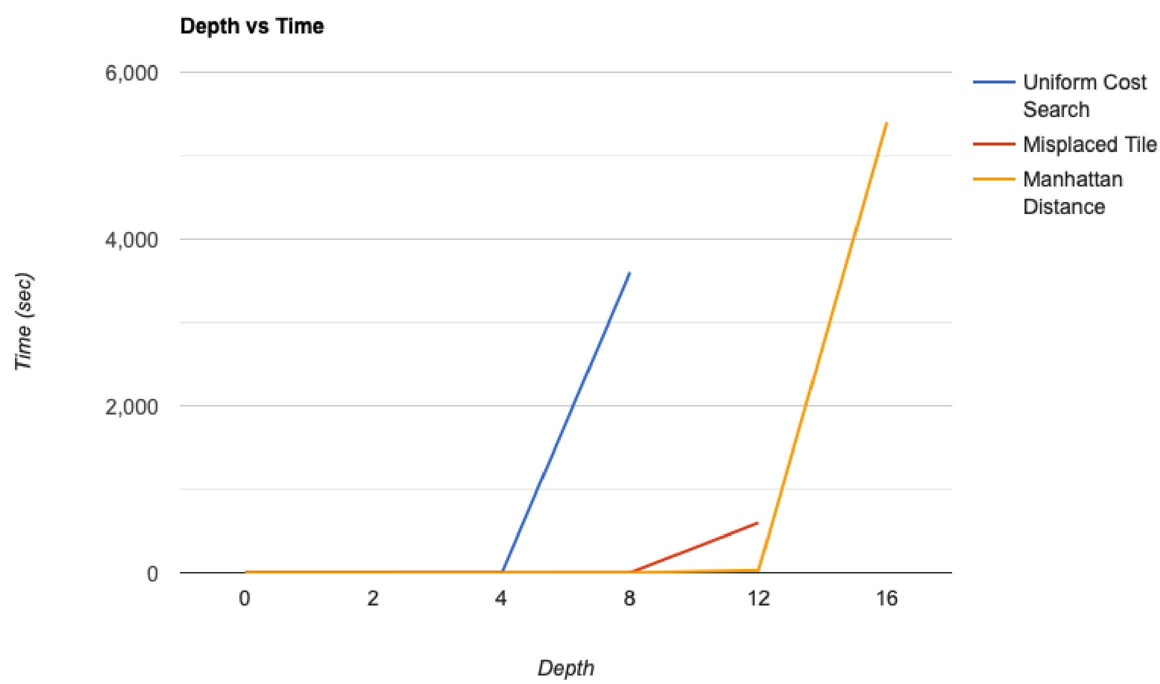
Algorithm Comparison

Using the test cases provided by Professor Eamonn Keogh in his project manual to compare the algorithms in action:

This is the result produced by each algorithm based on the number of nodes expanded:



After a certain depth, some algorithm takes an extremely long time to find the goal state, resulting in missing data for some depths per algorithm:



But based on the graph Depth vs Expanded Nodes, we can see that initially in the earlier depths, the differences between each algorithm shows very minimal differences until the puzzle gets harder to solve. Uniform Cost Search is shown to expand the most nodes as the puzzle's depth increases. Misplaced Tile being the second, and Manhattan being the best out of all three algorithms.

Conclusion

Based on the results provided by the program, we can see that the A* algorithm with the Manhattan Heuristic is far superior compared to Uniform Cost Search and the Misplaced Tile Heuristics in terms of having the least amount of nodes expanded per depth, and the least amount of time spent to find the solution.

Example Output

```
Welcome to Eight Puzzle: Please Enter an Initial State with 0 being the Blank

Enter for first row: 1 2 3
Enter for second row: 5 0 6
Enter for third row: 4 7 8

Initial State
-------------
[[1 2 3]
 [5 0 6]
 [4 7 8]]

Please Choose An Algorithm:
Enter a to input new state, Enter x to exit.
1: Uniform Cost Search, 2: Misplaced Tile Heuristic, 3: Manhattan Distance Heuristic : 1
Searching...
Success!
Nodes Expanded:  28
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Please Choose An Algorithm:
Enter a to input new state, Enter x to exit.
1: Uniform Cost Search, 2: Misplaced Tile Heuristic, 3: Manhattan Distance Heuristic : 2
Searching...
Success!
Nodes Expanded:  4
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Please Choose An Algorithm:
Enter a to input new state, Enter x to exit.
1: Uniform Cost Search, 2: Misplaced Tile Heuristic, 3: Manhattan Distance Heuristic : 3
Searching...
Success!
Nodes Expanded:  4
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Please Choose An Algorithm:
Enter a to input new state, Enter x to exit.
1: Uniform Cost Search, 2: Misplaced Tile Heuristic, 3: Manhattan Distance Heuristic : a

Enter for first row: 1 2 3
Enter for second row: 4 5 6
Enter for third row: 0 7 8

Initial State
-------------
[[1 2 3]
 [4 5 6]
 [0 7 8]]
```

Source Code

```
import numpy as np
import copy as cp

#Heuristic Cost Calculators

def a_star_cost(heuristic, init_state, state):

  if heuristic == 1:
    return depth_cost(init_state, state)
  elif heuristic == 2:
    h = misplaced_tile_h(state)
  elif heuristic == 3:
    h = manhattan_dist_h(state)
  else:
    print('Invalid Algorithm.')
    return

  g = depth_cost(init_state, state)
  return g + h

def misplaced_tile_h(state):
  goal = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 0]]

  misplaced_counter = 0
  for i in range(len(state)):
    for j in range(len(state)):
        if state[i][j] != goal[i][j]:
            misplaced_counter += 1
  return misplaced_counter

def manhattan_dist_h(state):
  goal = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 0]]

  manhattan_counter = 0
  for i in range(len(state)):
    for j in range(len(state)):
      if state[i][j] == 1:
        manhattan_counter += abs(i - 0) + abs(j - 0)
      elif state[i][j] == 2:
        manhattan_counter += abs(i - 0) + abs(j - 1)
      elif state[i][j] == 3:
        manhattan_counter += abs(i - 0) + abs(j - 2)
      elif state[i][j] == 4:
        manhattan_counter += abs(i - 1) + abs(j - 0)
      elif state[i][j] == 5:
        manhattan_counter += abs(i - 1) + abs(j - 1)
      elif state[i][j] == 6:
        manhattan_counter += abs(i - 1) + abs(j - 2)
      elif state[i][j] == 7:
        manhattan_counter += abs(i - 2) + abs(j - 0)
      elif state[i][j] == 8:
        manhattan_counter += abs(i - 2) + abs(j - 1)
      else:
        manhattan_counter += 0
  return manhattan_counter
```

```
#algorithm inspired from project description
def general_search(state, algorithm):
  #initialize queue, explored list and node counter
  queue = []
  explored = []
  node_expanded = 0
  algo = 0

  print('Searching...')
  #choose algorithm based on user input
  # 1 -> uniformed cost search
  # 2 -> a star w/ misplaced tile heuristic
  # 3 -> a star w/ manhattan heuristic
  if algorithm == '1':
    algo = 1
  elif algorithm == '2':
    algo = 2
  elif algorithm == '3':
    algo = 3
  else:
    print('Invalid Algorithm')
    return

  #if initial state is already a goal state
  if goal_state(state) == True:
    print('Success!')
    print('Nodes Expanded: ', node_expanded)
    print(state)
    return

  explored.append(state)
  queue += expand_node(state)
  node_expanded += 1

  while True:
    explore_chk = False
    #sort queue based on a star cost
    queue = sorted(queue, key=lambda x:a_star_cost(algo, state, x))

    #if queue is empty
    if not queue:
      print('Failure: Unable to Find Solution!')
      return

    #remove front of queue
    node = queue.pop(0)

    #if node has been already expanded, pop and go to the next node
    while explore_chk == False:
      prev_node = node
      for i in explored:
        if np.all(node == i):
          node = queue.pop(0)
          break
      if np.all(prev_node == node):
        explore_chk = True

    #add node to explored list
    explored.append(node)

    #if node is goal state, return
```

```
    if goal_state(node) == True:
      print('Success!')
      print('Nodes Expanded: ', node_expanded)
      print(node)
      return

    #else expand node and add to queue
    queue += expand_node(node)
    node_expanded += 1


def main():
  print('Welcome to Eight Puzzle: Please Enter an Initial State with 0 being the Blank')
  print()
  init_state = state()
  print()
  print('Initial State')
  print('-------------')
  print(init_state)
  print()
  while True:
    print('Please Choose An Algorithm:')
    print('Enter a to input new state, Enter x to exit.')
    userInput = input('1: Uniform Cost Search, 2: Misplaced Tile Heuristic, 3: Manhattan
Distance Heuristic : ')
    if userInput == '1' or userInput == '2' or userInput == '3':
      general_search(init_state, userInput)
    elif userInput == 'a':
      print()
      init_state = state()
      print()
      print('Initial State')
      print('-------------')
      print(init_state)
      print()
    elif userInput == 'x':
      return
    else:
      print('ERROR: Invalid Choice!')
      return

#--Helper Functions--

#function to calculate depth cost or g(n)
def depth_cost(init_state, state):
  g_cost = 0

  #if the state is the initial state => g(n) = 0
  if np.all(state == init_state):
      return g_cost

  #else expand the initial state
  expand_list = expand_node(init_state)

  while True:
    expand_temp = []
    g_cost += 1

    for i in expand_list:
      if np.all(state == i):
        return g_cost
```

```
    for i in expand_list:
      child_nodes = expand_node(i)
      for j in child_nodes:
        expand_temp.append(j)

    expand_list = expand_temp

#Function to check if goal state has been met
def goal_state(state):
  goal = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 0]]

  if np.all(state == goal):
    return True
  else:
    return False

#Initial State Entered by User
def state():
  init_state = []

  x, y, z = input('Enter for first row: ').split()
  init_state.append(int(x))
  init_state.append(int(y))
  init_state.append(int(z))

  x, y, z = input('Enter for second row: ').split()
  init_state.append(int(x))
  init_state.append(int(y))
  init_state.append(int(z))

  x, y, z = input('Enter for third row: ').split()
  init_state.append(int(x))
  init_state.append(int(y))
  init_state.append(int(z))

  return(np.reshape(init_state, (3, 3)))

#Function To expand nodes
def expand_node(node):
  blank_loc = []
  nodes = []
  temp = cp.copy(node)
  for i in range(len(node)):
    for j in range(len(node[0])):
      if node[i][j] == 0:
        blank_loc.append(i)
        blank_loc.append(j)
        break

  #if blank is not located at far top row, swap
  if blank_loc[0] != 0:
    temp[blank_loc[0]][blank_loc[1]], temp[blank_loc[0] - 1][blank_loc[1]] = temp[blank_loc[0]
- 1][blank_loc[1]], temp[blank_loc[0]][blank_loc[1]]
    nodes.append(temp)
    temp = cp.copy(node)

  #if blank is not located at far left column, swap
  if blank_loc[1] != 0:
    temp[blank_loc[0]][blank_loc[1]], temp[blank_loc[0]][blank_loc[1] - 1] =
temp[blank_loc[0]][blank_loc[1] - 1], temp[blank_loc[0]][blank_loc[1]]
```

```
    nodes.append(temp)
    temp = cp.copy(node)

  #if blank is not located at far bottom row, swap
  if blank_loc[0] != 2:
    temp[blank_loc[0]][blank_loc[1]], temp[blank_loc[0] + 1][blank_loc[1]] = temp[blank_loc[0]
+ 1][blank_loc[1]], temp[blank_loc[0]][blank_loc[1]]
    nodes.append(temp)
    temp = cp.copy(node)

  #if blank is not located at far right column, swap
  if blank_loc[1] != 2:
    temp[blank_loc[0]][blank_loc[1]], temp[blank_loc[0]][blank_loc[1] + 1] =
temp[blank_loc[0]][blank_loc[1] + 1], temp[blank_loc[0]][blank_loc[1]]
    nodes.append(temp)
    temp = cp.copy(node)

  return nodes

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