Calvin Pugmire

CS 470, section 001

Programming Assignment (Deterministic Search)

Task 1.2:

python eight.py \_\_\_.txt

Solved 40 puzzles from file: easy.txt

Average nodes expanded: 334.1

Average search time: 0.006458097696304321

Average solution length: 7.0

///

Solved 40 puzzles from file: medium.txt

Average nodes expanded: 28269.0

Average search time: 0.3514295816421509

Average solution length: 15.0

///

Solved 40 puzzles from file: hard.txt

Average nodes expanded: 695113.1

Average search time: 7.703120929002762

Average solution length: 21.0

///

worst.txt: Too long.

///

random.txt: Too long.

Task 1.3:

python eight.py \_\_\_.txt -s bfs -t \_

|  |  |  |
| --- | --- | --- |
| 1. u | 2. g | 3. a |
| Solved 40 puzzles from file: easy.txt  Average nodes expanded: 143.7  Average search time: 0.002734261751174927  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 6407.25  Average search time: 0.1732851445674896  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 72320.5  Average search time: 2.857686048746109  Average solution length: 21.0  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 181315.7  Average search time: 11.154147070646285  Average solution length: 30.05  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 58265.65  Average search time: 2.8835142612457276  Average solution length: 16.775 | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 52.675  Average search time: 0.0011720776557922364  Average solution length: 10.7  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 624.8  Average search time: 0.01435028314590454  Average solution length: 75.9  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 716.2  Average search time: 0.0167940616607666  Average solution length: 94.55  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 771.55  Average search time: 0.019334781169891357  Average solution length: 100.75  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 363.175  Average search time: 0.009648030996322632  Average solution length: 56.825 | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 13.35  Average search time: 0.0003902614116668701  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 341.175  Average search time: 0.00898277759552002  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 4887.875  Average search time: 0.12544544339179992  Average solution length: 21.0  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 100266.5  Average search time: 3.328859108686447  Average solution length: 30.05  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 11867.425  Average search time: 0.35192922353744505  Average solution length: 16.775 |

Task 1.5:

python eight.py \_\_\_.txt -s bfs -f \_\_\_ -t \_

|  |  |  |  |
| --- | --- | --- | --- |
|  | Uniform Cost (-t u) | Greedy (-t g) | A\* (-t a) |
| Out Of Place (-f top) | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 143.7  Average search time: 0.003397214412689209  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 6407.25  Average search time: 0.17597563266754152  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 72320.5  Average search time: 2.7566242933273317  Average solution length: 21.0  ///  worst.txt: Too long.  ///  random.txt: Too long. | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 52.675  Average search time: 0.0015218853950500488  Average solution length: 10.7  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 624.8  Average search time: 0.015580534934997559  Average solution length: 75.9  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 716.2  Average search time: 0.017748379707336427  Average solution length: 94.55  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 771.55  Average search time: 0.01885782480239868  Average solution length: 100.75  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 363.175  Average search time: 0.009101653099060058  Average solution length: 56.825 | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 13.35  Average search time: 0.00047474503517150877  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 341.175  Average search time: 0.00851036310195923  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 4887.875  Average search time: 0.12726954221725464  Average solution length: 21.0  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 100266.5  Average search time: 3.379349720478058  Average solution length: 30.05  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 11867.425  Average search time: 0.35470956563949585  Average solution length: 16.775 |
| Out Of Row+Column (-f torc) | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 143.7  Average search time: 0.006241744756698609  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 6407.25  Average search time: 0.30295884013175967  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 72320.5  Average search time: 4.40437838435173  Average solution length: 21.0  ///  worst.txt: Too long.  ///  random.txt: Too long. | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 12.6  Average search time: 0.0006503403186798095  Average solution length: 7.55  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 314.65  Average search time: 0.013648778200149536  Average solution length: 67.6  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 354.2  Average search time: 0.015412580966949464  Average solution length: 83.95  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 405.525  Average search time: 0.0178511381149292  Average solution length: 97.9  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 247.05  Average search time: 0.010746759176254273  Average solution length: 56.925 | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 10.0  Average search time: 0.0005527198314666748  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 130.0  Average search time: 0.005998432636260986  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 1225.05  Average search time: 0.056196081638336184  Average solution length: 21.0  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 40513.3  Average search time: 2.0192739069461823  Average solution length: 30.05  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 3531.8  Average search time: 0.1645221710205078  Average solution length: 16.775 |
| Manhattan (-f md) | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 143.7  Average search time: 0.005340993404388428  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 6407.25  Average search time: 0.2584782361984253  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 72320.5  Average search time: 3.90814493894577  Average solution length: 21.0  ///  worst.txt: Too long.  ///  random.txt: Too long. | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 14.25  Average search time: 0.0006896734237670898  Average solution length: 8.5  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 183.4  Average search time: 0.007053196430206299  Average solution length: 47.35  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 330.125  Average search time: 0.012123554944992065  Average solution length: 68.95  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 375.375  Average search time: 0.013834601640701294  Average solution length: 78.15  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 121.925  Average search time: 0.0048205554485321045  Average solution length: 34.075 | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 12.175  Average search time: 0.0007726728916168212  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 115.725  Average search time: 0.004433256387710571  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 779.15  Average search time: 0.02925131320953369  Average solution length: 21.0  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 26059.075  Average search time: 0.9683151245117188  Average solution length: 30.05  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 2675.875  Average search time: 0.09860233068466187  Average solution length: 16.775 |

What I learned:

-1.1:

--How to write a goal test for an 8-puzzle.

-1.2:

--How well IDS search performs for different difficulties of 8-puzzle:

---Exponential relationship between solution length and nodes expanded.

-1.3:

--How to calculate f-value for uniform cost search for 8-puzzle:

---Length of path to next node.

--How to calculate f-value for greedy search for 8-puzzle:

---Heuristic cost of next node.

--How to calculate f-value for A\* search for 8-puzzle:

---Length of path to next node + heuristic cost of next node.

--How well each search performs for different difficulties of 8-puzzle:

---Uniform cost performs worst (exponential).

---Greedy performs best (linear) but with worse solutions.

---A\* performs medium (exponential) but is best overall.

-1.4:

--How to implement tiles out of row and column heuristic for 8-puzzle.

--How to implement manhattan distance heuristic for 8-puzzle.

-1.5:

--How well each heuristic performs for different difficulties of 8-puzzle:

---TOP performs worst overall (largest rate of increase).

---TORC performs medium overall (medium rate of increase).

---MD performs best overall (lowest rate of increase).

Part 2:

python eight.py \_\_\_.txt -s ids -f \_\_\_

|  |  |  |
| --- | --- | --- |
| 1. top | 2. torc | 3. md |
| Solved 40 puzzles from file: easy.txt  Average nodes expanded: 32.025  Average search time: 0.0005502045154571533  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 1463.925  Average search time: 0.016854679584503172  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 27243.825  Average search time: 0.3032244980335236  Average solution length: 21.0  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 2714460.725  Average search time: 30.60689726471901  Average solution length: 30.05  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 167634.05  Average search time: 1.8869463622570037  Average solution length: 16.775 | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 17.05  Average search time: 0.0004268467426300049  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 467.975  Average search time: 0.009436100721359253  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 5869.9  Average search time: 0.12264817357063293  Average solution length: 21.0  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 439348.65  Average search time: 9.397001969814301  Average solution length: 30.05  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 33957.175  Average search time: 0.7218914330005646  Average solution length: 16.775 | Solved 40 puzzles from file: easy.txt  Average nodes expanded: 28.925  Average search time: 0.000604015588760376  Average solution length: 7.0  ///  Solved 40 puzzles from file: medium.txt  Average nodes expanded: 417.9  Average search time: 0.0075040459632873535  Average solution length: 15.0  ///  Solved 40 puzzles from file: hard.txt  Average nodes expanded: 3091.65  Average search time: 0.052082794904708865  Average solution length: 21.0  ///  Solved 40 puzzles from file: worst.txt  Average nodes expanded: 223070.225  Average search time: 3.737385505437851  Average solution length: 30.05  ///  Solved 40 puzzles from file: random.txt  Average nodes expanded: 20439.075  Average search time: 0.3373986542224884  Average solution length: 16.775 |

What I learned:

-How to change IDS into IDA\*:

--Set initial depth limit to A\* f-value:

---Length of path to next node + heuristic cost of next node.

--Set depth-limit-reached check to compare to A\* f-value:

---Length of path to next node + heuristic cost of next node.

--Set search type in current node’s options to A\*:

---Does not alter performance, but is a good practice.

-How well IDA\* search performs for different difficulties of 8-puzzle:

--Exponential relationship between solution length and nodes expanded.

-How well IDA\* search performs with each heuristic for different difficulties of 8-puzzle:

---TOP performs worst overall (largest rate of increase).

---TORC performs medium overall (medium rate of increase).

---MD performs best overall (lowest rate of increase).

from \_\_future\_\_ import print\_function  
from queue import PriorityQueue  
import sys  
import math  
import time  
import random  
import argparse   
  
# Code for AI Class Programming Assignment 1  
# Written by Chris Archibald  
# archibald@cs.byu.edu  
# Last Updated January 24, 2020  
  
# GOAL: (0 is the blank tile)  
# 0 1 2  
# 3 4 5  
# 6 7 8  
  
class Puzzle():  
 """  
 This is the 8-puzzle class. You shouldn't have to modify it at all.  
 """   
  
 def \_\_init\_\_(self, arrangement):  
 """  
 The state (and arrangement passed in) is a list of length 9, that stores which tile is in each place  
 In a solved puzzle, each place number holds the tile of the same number  
 i.e. solution is state = [0,1,2,3,4,5,6,7,8]  
 """  
 self.state = arrangement[:]  
 self.blank = None  
  
 for i in range(len(self.state)):  
 if self.state[i] == 0:  
 self.blank = i  
  
 def print\_puzzle(self):  
 """  
 Print a visual description of the puzzle to the output  
 """  
 k = 0  
 for i in range(3):  
 for j in range(3):  
 print('', end="")   
 print(self.state[k], end="")  
 k = k + 1  
 print('')  
   
 def get\_moves(self):  
 """  
 The moves correspond to the motion of the tile into the blank space  
 it can be U (up), D (down), R (right), or L (left)  
 """  
 invalid\_moves = []  
 if self.blank < 3:  
 invalid\_moves.append('D')  
 if self.blank > 5:  
 invalid\_moves.append('U')  
 if self.blank % 3 == 0:  
 invalid\_moves.append('R')  
 if self.blank % 3 == 2:  
 invalid\_moves.append('L')  
  
 base\_moves = ['U', 'D', 'L', 'R']  
 valid\_moves = []  
 for m in base\_moves:  
 if m not in invalid\_moves:  
 valid\_moves.append(m)  
  
 return valid\_moves  
  
 def do\_move(self, move):  
 """  
 Modify the state by performing the given move.  
 This assumes that the move is valid  
 """  
 swapi = 0  
 if move == 'U':  
 swapi = self.blank + 3  
 if move == 'D':  
 swapi = self.blank - 3  
 if move == 'L':  
 swapi = self.blank + 1  
 if move == 'R':  
 swapi = self.blank - 1  
  
 temp = self.state[swapi]  
 self.state[swapi] = self.state[self.blank]  
 self.state[self.blank] = temp  
 self.blank = swapi  
  
 def undo\_move(self, move):  
 """  
 This modifies the state by undoing the move. For use in recursive search  
 Assumes the move was a valid one  
 """  
 swapi = 0  
 if move == 'D':  
 swapi = self.blank + 3  
 if move == 'U':  
 swapi = self.blank - 3  
 if move == 'R':  
 swapi = self.blank + 1  
 if move == 'L':  
 swapi = self.blank - 1  
  
 temp = self.state[swapi]  
 self.state[swapi] = self.state[self.blank]  
 self.state[self.blank] = temp  
 self.blank = swapi  
   
 def is\_solved(self):  
 """  
 Returns True if the puzzle is solved, False otherwise  
 """  
 ######## TASK 1.1 BEGIN ##########  
  
 if self.state == [0,1,2,3,4,5,6,7,8]:  
 return True  
 else:  
 return False  
  
 ######## TASK 1.1 END ##########  
  
  
 def \_\_repr\_\_(self):  
 return "".join([str(i) for i in self.state])  
  
 def id(self):  
 """  
 Returns the string representation of this puzzle's state.   
 Useful for storing state in a dictionary  
 """  
 return self.\_\_repr\_\_()  
  
class SearchNode():  
 """  
 Our search node class  
 """  
 def \_\_init\_\_(self,cost,puzzle,path,options):  
 """  
 Initialize all the relevant parts of the search node  
 """  
 self.cost = cost  
 self.puzzle = puzzle  
 self.path = path  
 self.options = options  
 self.h = heuristic(self,self.options)  
 self.compute\_f\_value()  
   
 def compute\_f\_value(self):  
 """  
 Compute the f-value for this node  
 """  
 ######## TASK 1.3 BEGIN ##########  
   
 #Modify these lines to implement the search algorithms (greedy, Uniform-cost or A\*)  
 self.h = heuristic(self, self.options)  
 self.f\_value = 0  
  
 if self.options.type == 'u':  
 #uniform cost search algorithm  
 self.f\_value = len(self.path) # Change this to implement uniform cost search!  
  
 elif self.options.type == 'g':  
 #greedy search algorithm  
 self.f\_value = self.h # Change this to implement greedy!  
  
 elif self.options.type == 'a':  
 #A\* search algorithm  
 self.f\_value = len(self.path)+self.h # Change this to implement A\*!  
  
 else:  
 print('Invalid search type (-t) selected: Valid options are g, u, and a')  
 sys.exit()  
  
 ######## TASK 1.3 END ##########  
  
 #Comparison operator. Don't modify this or best-first search might stop working  
 def \_\_lt\_\_(self,other):  
 """  
 Comparison operator so that nodes will be sorted in priority queue based on f-value  
 """  
 return self.f\_value < other.f\_value   
  
def heuristic(node,options):  
 """  
 This is the function that is called from the SearchNode class to get the heuristic value for a node  
 """  
 if options.function == 'top':  
 return tiles\_out\_of\_place(node.puzzle)  
 elif options.function == 'torc':  
 return tiles\_out\_of\_row\_column(node.puzzle)  
 elif options.function == 'md':  
 return manhattan\_distance\_to\_goal(node.puzzle)  
 else:  
 print('Invalid heuristic selected. Options are top, torc, and md')  
 sys.exit()   
  
def tiles\_out\_of\_place(puzzle):  
 """  
 This heuristic counts the number of tiles out of place.   
 """  
 #Keep track of the number of tiles out of place  
 num\_out\_of\_place = 0  
   
 #Cycle through all of the places in the puzzle and see if the right tile is there  
 # (We ignore place 0 since that is where the blank tile goes and we shouldn't count it)  
 for i in range(1,len(puzzle.state)):  
   
 # The tile in place i ( puzzle.state[i] ) should be tile i.   
 # If it isn't increment out of place counter  
 # (To compare tile (string) with place (int), we must first convert from string to int as such:  
 # int(puzzle.state[i])  
   
 if puzzle.state[i] != i:  
 num\_out\_of\_place += 1  
   
 return num\_out\_of\_place  
   
   
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  
  
 rows = 3  
 cols = 3  
  
 num\_out\_rows = 0  
 num\_out\_cols = 0  
  
 for i in range(rows):  
 for j in range(cols):  
 if (puzzle.state[i\*rows+j] != i\*rows)\  
 and (puzzle.state[i\*rows+j] != i\*rows+1)\  
 and (puzzle.state[i\*rows+j] != i\*rows+2)\  
 and (puzzle.state[i\*rows+j] != 0):  
 num\_out\_rows += 1  
 if (puzzle.state[i\*rows+j] != j)\  
 and (puzzle.state[i\*rows+j] != rows+j)\  
 and (puzzle.state[i\*rows+j] != 2\*rows+j)\  
 and (puzzle.state[i\*rows+j] != 0):  
 num\_out\_cols += 1  
  
 return num\_out\_rows + num\_out\_cols  
   
 ######## TASK 1.4.1 END ##########  
  
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  
  
 rows = 3  
 cols = 3  
  
 lat\_diff = 0  
 lon\_diff = 0  
  
 for i in range(1,len(puzzle.state)):  
 place = puzzle.state.index(i)  
 lat\_diff += abs((place % cols) - (i % cols))  
 lon\_diff += math.floor(abs((place / rows) - (i / rows)))  
   
 return lat\_diff + lon\_diff  
   
 ######## TASK 1.4.2 END ##########   
  
   
def get\_tile\_row(tile):  
 """  
 Return the row of the given tile location (Helper function for you to use)  
 """  
 return int(tile / 3)  
  
def get\_tile\_column(tile):  
 """  
 Return the column of the given tile location (Helper function for you to use)  
 """  
 return tile % 3   
   
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 = len(start\_node.path)+start\_node.h #EDIT 1/3  
   
 #Maximum depth limit  
 max\_depth\_limit = 40  
   
 #Keep track of the total number of nodes we expand  
 total\_expanded = 0  
   
 #Keep trying until our depth limit hits 40  
 while depth\_limit < max\_depth\_limit:  
   
 #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)   
   
 #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  
   
 # 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)+node.h >= depth\_limit: #EDIT 2/3  
 # 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.options.type = 'a' #EDIT 3/3  
 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  
   
def run\_best\_first\_search(fringe, options):  
 """  
 Runs an arbitrary best-first search. To change which search is run, modify the f-value   
 computation in the search nodes  
  
 fringe is a priority queue of search nodes, ordered by f-values  
 """  
 #Create our data structure to track visited/expanded states  
 visited = dict()  
   
 #Variable to tell when we are done  
 done = False  
   
 #Main search loop. Keep going as long as we are not done and the FRINGE isn't empty  
 while not done and not fringe.empty():  
   
 #Get the next SearchNode from the FRINGE  
 cur\_node = fringe.get()  
   
 #Add it to our set of visited/expanded states (join creates a string from the state)  
 visited[cur\_node.puzzle.id()] = True  
   
 #Don't continue if the cost is too much  
 if cur\_node.cost > 200:  
 #None of the puzzles are this long, so we shouldn't continue further on this path  
 continue  
   
 #Check to see if this node's puzzle state is a goal state  
 if cur\_node.puzzle.is\_solved():  
 #It is! We are done, print out details  
 done = True  
 print('Best-First SOLVED THE PUZZLE: SOLUTION = ', cur\_node.path)  
 print('Expanded ', len(visited), 'states')  
 return len(visited), len(cur\_node.path)  
   
 else:  
 #Generate this SearchNode's successors and add them to the FRINGE  
   
 #Get the possible moves (actions) for this state  
 moves = cur\_node.puzzle.get\_moves()  
   
 #For each move, do the move, create SearchNode from successor, then add to FRINGE  
 for m in moves:  
 #Create new puzzle that new node will point to  
 np = Puzzle(cur\_node.puzzle.state)  
   
 #Execute the move/action  
 np.do\_move(m)  
   
 #Add to the FRINGE, as long as we haven't visited that puzzle  
 if np.id() not in visited:  
 #Create the new SearchNode  
 new\_node = SearchNode(cur\_node.cost + 1, np, cur\_node.path + m, options)  
   
 #Add it to the FRINGE, along with its f-value (stored inside the node)  
 fringe.put(new\_node)  
  
 #We didn't find a solution  
 if not done:  
 print('NO SOLUTION FOUND!')  
 return None,None  
   
def getOptions(args=sys.argv[1:]):  
 parser = argparse.ArgumentParser(description="8-Puzzle Solver.")  
 parser.add\_argument('file', metavar='FILENAME', type=str, help="File of puzzles to solve")  
 parser.add\_argument("-s", '--search', help="Search type: Options: ids (iterative deepening search) or bfs (best first search)", default='ids')  
 parser.add\_argument("-f", '--function', help="Heuristic function used: Options: top (tiles out of place), torc (tiles out of row/column), or md (manhattan distance)",default='top')  
 parser.add\_argument("-t", '--type', help="Evaluation function type: Options: g (greedy), u (uniform cost), or a (a-star)", default='u')  
 options = parser.parse\_args(args)  
 return options  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 #Get command line options  
 options = getOptions()  
 print(options)  
 print('Searching for solutions to puzzles from file: ', sys.argv[1])  
   
 #Open puzzle file  
 pf = open(options.file,'r')  
   
 #You can modify the maximum number of puzzles to solve if you want to test on more puzzles  
 max\_to\_solve = 40  
   
 #Variables to keep track of solving statistics  
 num\_solved = 0  
 exp\_num = 0  
 tot\_time = 0.0  
 path\_length = 0  
   
 for ps in pf.readlines():  
 print('Searching to find solution to following puzzle:', ps)  
   
 #Create puzzle from file line  
 a = [int(i) for i in ps.rstrip()]  
 p = Puzzle(a)  
   
 #Print the puzzle to the screen  
 p.print\_puzzle()  
  
 #Create the initial search node corresponding to the given puzzle state  
 start\_node = SearchNode(0,p,'',options)  
   
 #Create the priority Queue to store the SearchNodes in  
 pq = PriorityQueue()  
   
 #Insert the initial state into the Queue  
 pq.put(start\_node)  
   
 #Get initial timing info  
 start = time.time()  
   
 #Run the given search search (each returns number of nodes expanded and the length of the path found)   
 if options.search == 'bfs':  
 #Run the best-first searches  
 exp, pl = run\_best\_first\_search(pq, options)  
 elif options.search == 'ids':   
 #Use this line to run the iterative deepening-search  
 exp, pl = run\_iterative\_search(start\_node)  
 else:  
 print("Search option not valid. Can be bfs or ids")  
 sys.exit()  
  
 if exp is None:  
 print('PUZZLE NOT SOLVED')  
 break  
   
 #Keep track of statistics so we can compare search methods  
 exp\_num += exp  
 path\_length += pl  
 print('Solution path length is : ', pl)  
   
 #Calculate Timing info  
 end = time.time()  
 tot\_time += end - start  
 num\_solved += 1  
   
 #Stop after we have solved the specified number of puzzles  
 if num\_solved >= max\_to\_solve:  
 break  
   
 print('Done with solving puzzles.\n\n')  
   
 #Print out statistics about this batch  
 if num\_solved > 0:  
 print('Solved', num\_solved, 'puzzles from file: ', sys.argv[1])  
 print('Average nodes expanded: ', float(exp\_num) / float(num\_solved))  
 print('Average search time: ', tot\_time / num\_solved)  
 print('Average solution length: ', path\_length / num\_solved)  
 else:  
 print('A puzzle was not solved. This means you haven\'t correctly implemented something. Please double check your code and try again.')