CS 205 Project Instructor: Dr Eamonn Keogh

**The Eight Puzzle**

Yuan Yao [y](mailto:bertie@cs.ucr.edu)yao009@ucr.edu

SID: 861242929 11/3/2016

In completing this project, I consulted:

* <https://www.cs.princeton.edu/courses/archive/spr10/cos226/assignments/8puzzle.html> for the structure of my eight puzzle solver
* <https://docs.python.org/2/library/queue.html> for the priority queue data structure in python

All the important code is original.

**Code Section:**

**import** **Queue**

**import** **time**

**class** **Board**:

*# Initialize an n\*n board with a list of tiles.*

*# The tiles in the board class are stored in [[]]*

**def** \_\_init\_\_(self, tiles, n=3, previous=None, moves=0):

self.tiles = self.goalState = []

self.moves = moves

*# store the previous node in order to eliminate the duplicate states*

self.previous = previous

**for** i **in** range(n):

row = []

**for** j **in** range(n):

*# store the coordinate of the blank tile*

**if** tiles[i][j] == 0:

self.blank = (i, j)

row.append(tiles[i][j])

self.tiles.append(row)

self.n = n

self.goalState = [[1,2,3],[4,5,6],[7,8,0]]

*# This function is to find if the current board is equal to another.*

**def** equals(self, board):

**for** i **in** range(self.n):

**for** j **in** range(self.n):

**if** self.tiles[i][j] != board.tiles[i][j]:

**return** False

**return** True

*# This function is to find the nanhattan distance of the board*

**def** manhattan(self):

manhattan\_distance = 0

**for** row **in** range(self.n):

**for** col **in** range(self.n):

tile = self.tiles[row][col]

**if** tile != self.goalState[row][col] **and** tile != 0:

*# add row distance and column distance to manhattan\_distance*

manhattan\_distance += abs((tile - 1) / self.n - row) + abs((tile - 1) % self.n - col)

**return** manhattan\_distance

*# This function is to find the number of misplaced tiles of the board*

**def** misplaced(self):

misplaced\_tiles = 0

**for** row **in** range(self.n):

**for** col **in** range(self.n):

tile = self.tiles[row][col]

**if** tile != self.goalState[row][col] **and** tile != 0:

misplaced\_tiles += 1

**return** misplaced\_tiles

*# This function is to find all the neighbor states of the current board and return a generator*

**def** neighbors(self):

row = self.blank[0]

col = self.blank[1]

*# make a copy of the current state*

copy = []

**for** i **in** self.tiles:

r = []

**for** j **in** i:

r.append(j)

copy.append(r)

*# move blank tile up if possible*

**if** row > 0:

copy[row][col], copy[row-1][col] = copy[row-1][col], copy[row][col]

**yield** Board(copy, previous=self, moves=self.moves+1)

*# restore copy to the current state*

copy[row][col], copy[row-1][col] = copy[row-1][col], copy[row][col]

*# move blank tile down if possible*

**if** row < self.n - 1:

copy[row][col], copy[row+1][col] = copy[row+1][col], copy[row][col]

**yield** Board(copy, previous=self, moves=self.moves+1)

*# restore copy to the current state*

copy[row][col], copy[row+1][col] = copy[row+1][col], copy[row][col]

*# move blank tile left if possible*

**if** col > 0:

copy[row][col], copy[row][col-1] = copy[row][col-1], copy[row][col]

**yield** Board(copy, previous=self, moves=self.moves+1)

*# restore copy to the current state*

copy[row][col], copy[row][col-1] = copy[row][col-1], copy[row][col]

*# move blank tile right if possible*

**if** col < self.n - 1:

copy[row][col], copy[row][col+1] = copy[row][col+1], copy[row][col]

**yield** Board(copy, previous=self, moves=self.moves+1)

*# restore copy to the current state*

copy[row][col], copy[row][col+1] = copy[row][col+1], copy[row][col]

*# print the current board*

**def** printBoard(self):

**for** i **in** self.tiles:

**for** j **in** i:

**if** j == 0:

**print** ' ',

**else**:

**print** j,

**print** ' ',

**print**

*# This function is to return whther the current state is a goal state*

**def** isGoal(self):

**return** self.equals(Board(self.goalState))

*# Solver is to do A\* search*

**class** **Solver**:

**def** \_\_init\_\_(self, initial):

self.moves = 0

self.queue = Queue.PriorityQueue()

h = initial.manhattan()

*# enque the initial state into the priority queue using its h(n)*

self.queue.put((h, initial))

*# store the previous state*

self.previous = None

*# store the nodes expanded*

self.nodesCount = 0

*# store the maximum number of nodes in the queue*

self.maxNodes = 0

*# This function is to print the minimum steps to goal*

**def** solution(self, method):

node = self.search\_with\_method(method)

**if** **not** node:

**print** "No solution."

**return** -1

sol = []

**while** node:

sol.append(node)

node = node.previous

**for** i **in** sol[::-1]:

i.printBoard()

g = i.moves

h = 0

**if** method == 1:

h = 0

**elif** method == 2:

h = i.misplaced()

**elif** method == 3:

h = i.manhattan()

**print**("This node has a g(n)=**%d** and h(n)=**%d**." %(g, h)),

**if** i.isGoal():

**print** "GOAL!**\n**"

**else**:

**print** "Expanding the best state of this node...**\n**"

**return** len(sol)-1

*# Do the three search method in one function*

**def** search\_with\_method(self, method):

*# General search algorithm*

**while** **not** self.queue.empty():

**if** self.maxNodes < self.queue.qsize():

self.maxNodes = self.queue.qsize()

board = self.queue.get()

*# if it's goal, return the state*

**if** board[1].isGoal():

**return** board[1]

*# Expand the nodes if it's not the goal state*

**for** b **in** board[1].neighbors():

*# skip duplicate states*

**if** board[1].previous **and** b.equals(board[1].previous):

**continue**

*# count a expand node*

self.nodesCount += 1

*# calculate the evaluate function value for the current state*

g = b.moves

h = 0

*# uniform search just hard code h=0*

**if** method == 1:

h = 0

*# misplaced tiles heuristic*

**elif** method == 2:

h = b.misplaced()

*# manhattan distance heuristic*

**elif** method == 3:

h = b.manhattan()

f = g + h

self.queue.put((f, b))

*# If the queue is dequeued empty, there is no solution*

**else**:

**return** None

**if** \_\_name\_\_ == '\_\_main\_\_':

**print** "Welcom to Yuan's 8-puzzle solver!"

choice = raw\_input("Type 1 to use a default puzzle, or 2 to enter your own puzzle: ")

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

**if** int(choice) == 1:

**pass**

**elif** int(choice) == 2:

puzzle = []

**print** "Enter your puzzle, use 0 to represent the blank"

row = raw\_input("Enter the first row, use space or tabs between number: ")

puzzle.append(map(int, row.split()))

row = raw\_input("Enter the second row, use space or tabs between number: ")

puzzle.append(map(int, row.split()))

row = raw\_input("Enter the third row, use space or tabs between number: ")

puzzle.append(map(int, row.split()))

**print** "**\n**Enter your choice of algorithm:"

**print** " 1. Uniform Cost Search"

**print** " 2. A\* with the Misplaced Tile heuristic."

**print** " 3. A\* with the Manhattan distance heuristic."

method = raw\_input(" ")

**print**

*# Solve the puzzle*

board = Board(puzzle)

start\_time = time.time()

solver = Solver(board)

depth = solver.solution(int(method))

end\_time = time.time()

time\_elapsed = end\_time - start\_time

**print** "To solve this problem the search algorithm expanded a total of **%d** nodes." %solver.nodesCount

**print** "The maximum number of nodes in the queue at any one time was **%d**." %solver.maxNodes

**print** "The depth of the goal node was **%d**." %depth

**print** "Time cost: **%f**s" %time\_elapsed

1 2 3

4 \* 6

7 5 8

**Trace of Manhattan distance A\* on the** **: :**

Welcom to Yuan's 8-puzzle solver!

Type 1 to use a **default** puzzle, or 2 to enter your own puzzle: 2

Enter your puzzle, use 0 to represent the blank

Enter the first row, use space or tabs between number: 1 2 3

Enter the second row, use space or tabs between number: 4 0 6

Enter the third row, use space or tabs between number: 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

1 2 3

4 6

7 5 8

This node has a g(n)=0 and h(n)=2. Expanding the best state of this node...

1 2 3

4 5 6

7 8

This node has a g(n)=1 and h(n)=1. Expanding the best state of this node...

1 2 3

4 5 6

7 8

This node has a g(n)=2 and h(n)=0. GOAL!

To solve this problem the search algorithm expanded a total of 6 nodes.

The maximum number of nodes in the queue at any one time was 5.

The depth of the goal node was 2.

Time cost: 0.001114s

1 2 7

4 3 6

5 8 \*

**Trace of Manhattan distance A\* on the** **:**

Welcom to Yuan's 8-puzzle solver!

Type 1 to use a **default** puzzle, or 2 to enter your own puzzle: 1

Enter your choice of algorithm:

1. Uniform Cost Search

2. A\* with the Misplaced Tile heuristic.

3. A\* with the Manhattan distance heuristic.

3

1 2 7

4 3 6

5 8

This node has a g(n)=0 and h(n)=8. Expanding the best state of this node...

1 2 7

4 3

5 8 6

This node has a g(n)=1 and h(n)=9. Expanding the best state of this node...

1 2 7

4 3

5 8 6

This node has a g(n)=2 and h(n)=8. Expanding the best state of this node...

…

1 2 3

4 5 6

7 8

This node has a g(n)=19 and h(n)=1. Expanding the best state of this node...

1 2 3

4 5 6

7 8

This node has a g(n)=20 and h(n)=0. GOAL!

To solve this problem the search algorithm expanded a total of 1551 nodes.

The maximum number of nodes in the queue at any one time was 639.

The depth of the goal node was 20.

Time cost: 0.001114s

Summary of A\* with the Misplaced Tile heuristic on this puzzle:

To solve this problem the search algorithm expanded a total of 10232 nodes.

The maximum number of nodes in the queue at any one time was 4231.

The depth of the goal node was 20.

Time cost: 0.827453s

Summary of Uniform Cost Search on this puzzle:

To solve this problem the search algorithm expanded a total of 237497 nodes.

The maximum number of nodes in the queue at any one time was 103240.

The depth of the goal node was 20.

Time cost: 26.406672s

We can see that the manhattan distance heuristic performs better than the other two methods, and uniform cost search takes really long time.