­­­Assignment 1 CS170: Introduction to Artificial Intelligence

Osvaldo Moreno Dr. Eamonn Keogh

862016789

[omore005@ucr.edu](mailto:omore005@ucr.edu)

11-6-2019

In Completing this assignment, I consulted:

* Lecture Slides for the Search Algorithms
* T.A Ryan to help me understand better, refactor code, and debug some issues
* Several Python documentation pages:
  + https://www.geeksforgeeks.org/heap-queue-or-heapq-in-python/
  + https://dbader.org/blog/priority-queues-in-python
  + https://kite.com/python/docs/Queue.Queue.put
  + <https://dbader.org/blog/priority-queues-in-python>
  + <https://stackoverflow.com/questions/17873384/how-to-deep-copy-a-list>
* For help understanding searches I consulted some online sources
* <https://jackcanty.com/solving-8-puzzle-with-artificial-intelligence.html>
* <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>

All Important code is original. Unimportant routines that are not completely original are….

* The user interface was copied from the sample report with permission of Dr. Keogh.
* Examples were also used from the sample report.
* All functions from python libraries, i.e PriorityQueue and copy

CS170: Project 1 Write Up

Osvaldo Moreno Ornelas, SID 862016789

Introduction:

In this project we will perform an experiment. We will compare the performance of three different search algorithms. The algorithms will solve the 8-puzzle problem. The algorithms used are the following: Uniform cost search, A\* with misplaced tile heuristic, and A\* with misplaced tile and Manhattan distance.

For this project, I used the examples provided in the sample report. The language of preference for this project was Python 3.7 and run using PyCharm.

Quick Summary of the Algorithms:

**Uniform Cost Search**:

This algorithm will expand every possible option. For simplicity purposes the algorithm evaluates h(n) to be always 0. For every expansion we will add one to g(n), which at the same time is carried from the parent node.

**A\* With misplaced Tile Heuristic:**

In this algorithm the objective is to give the heuristic value based on the number of misplaced tiles. In other word, how many tiles are not in the correct position. We omit 0 to account for misplaced tiles. The expansion will occur on the node with the lowest heuristic number.

**A\* With Manhattan Distance:**

In this algorithm the objective is to give the heuristic value based on the number of tiles needed to reach the goal state. For example, a misplaced tile could take n moves to come back to its place, another will take m. The sum of the number of moves is the Manhattan distance. As in the previous algorithm, we do not account for the 0 tile. As in all search algorithms we use the cheapest node to expand the tree.

**How the Algorithms Compare**

As the puzzles got more complex it was not surprise that the number of moves got greater and greater. It is important that the heuristic value creates a difference when the algorithm is making a decision as for where to expand.

**Hardships Encountered**

Because the implementation was not as efficient as expected, test on cases four and five were not conducted. The computer was pushed to the limit and for safety, the cases were not tested. Perhaps with some refactoring, better data will be created in order to have more comparison points for all algorithms.

**Results**





|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Max Queue Size |  |
| Board Type | Uniform Cost Search | Misplaced Tile Heuristic | Manhattan Distance Heuristic |
| Trivial | 1 | 1 | 1 |
| Beginner | 6 | 3 | 3 |
| Easy | 6 | 3 | 3 |
| Medium | 126 | 3 | 3 |
| Hard |  |  |  |
| Veteran |  |  |  |

Conclusion

As seen in the data, the most inefficient algorithm was the uniformed search. The most efficient were both Manhattan and Misplaced tile. They both showed the same behavior. Perhaps, they were both really efficient at determining the best path.

Uniform cost search kept replicating over and over until one case was matched. Perhaps there is something non-efficient in the code written that made it have such great difference compared with other types of searches.

Sample Run

(Using same example as the sample report)

Welcome to an 8-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create your own.

2

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.

Enter the first row: 1 2 3

Enter the second row: 4 0 6

Enter the third row: 7 5 8

Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile Heuristic, or (3) the Manhattan Distance Heuristic.

3

1, 2, 3,

4, 0, 6,

7, 5, 8,

1, 2, 3,

4, 5, 6,

7, 0, 8,

1, 2, 3,

4, 5, 6,

7, 8, 0,

finished solving Manhattan Distance search

It took 2 levels

Number of Nodes Expanded: 7

Max queue size was 4 nodes

from Graph import puzzle

from Graph import boardNode

from Graph import queueSize

trivial = [[1, 2, 3],

 [4, 5, 6],

 [7, 8, 0]]

beginner = [[1, 2, 3],

 [4, 5, 6],

 [7, 0, 8]]

easy = [[1, 2, 0],

 [4, 5, 3],

 [7, 8, 6]]

medium = [[0, 1, 2],

 [4, 5, 3],

 [7, 8, 6]]

hard = [[8, 7, 1],

 [6, 0, 2],

 [5, 4, 3]]

veteran = [[1, 2, 3],

 [4, 5, 6],

 [8, 7, 0]]

goal\_state = [[1, 2, 3],

 [4, 5, 6],

 [7, 8, 0]]

def init\_default\_puzzle\_mode():

 selected\_difficulty = input("You wish to use a default puzzle. Please enter a desired difficulty on a scale from 0 to 5." + '\n')

 if selected\_difficulty == "0":

    print("Difficulty of 'Trivial' selected.")

    return trivial

 if selected\_difficulty == "1":

    print("Difficulty of 'beginner' selected.")

    return beginner

 if selected\_difficulty == "2":

    print("Difficulty of 'Easy' selected.")

    return easy

 if selected\_difficulty == "3":

    print("Difficulty of 'medium' selected.")

    return medium

 if selected\_difficulty == "4":

    print("Difficulty of 'hard' selected.")

    return hard

 if selected\_difficulty == "5":

    print("Difficulty of 'veteran' selected.")

    return veteran

def print\_puzzle(puzzle):

 for i in range(0, 3):

    print(puzzle[i])

 print('\n')

def getQueueSum():

   sumAll = 0

   for num in queueSize:

     sumAll += num

   return sumAll

def select\_and\_init\_algorithm(boardToSolve):

 algorithm = input("Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile Heuristic, "

 "or (3) the Manhattan Distance Heuristic." + '\n')

 solution = puzzle(boardToSolve)

 solveNode = boardNode(boardToSolve)

 if(algorithm == "1"):

     #passes in a board into the function

     solutionNode = solution.uniformedSearch(solveNode, algorithm)

     print("finished solving uniformed search\n")

     print("It took " + str(solutionNode.g) + " levels\n")

     print("Number of Nodes Expanded: " + str(getQueueSum()-1) + "\n")

     queueSize.sort()

     print("Max queue size was " + str(queueSize[len(queueSize)-1]) + " nodes\n")

 elif (algorithm == "2"):

     solutionNode = solution.uniformedSearch(solveNode, algorithm)

     print("finished solving Heuristic Tile search\n")

     print("It took " + str(solutionNode.g) + " levels\n")

     print("Number of Nodes Expanded: " + str(getQueueSum() - 1) + "\n")

     queueSize.sort()

     print("Max queue size was " + str(queueSize[len(queueSize) - 1]) + " nodes\n")

 else:

     solutionNode = solution.uniformedSearch(solveNode, algorithm)

     print("finished solving Manhattan Distance search\n")

     print("It took " + str(solutionNode.g) + " levels\n")

     print("Number of Nodes Expanded: " + str(getQueueSum() - 1) + "\n")

     queueSize.sort()

     print("Max queue size was " + str(queueSize[len(queueSize) - 1]) + " nodes\n")

"""

Here main Starts

"""

puzzle\_mode = input("Welcome to an 8-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create your own."+ '\n')

if puzzle\_mode == "1":

    select\_and\_init\_algorithm(init\_default\_puzzle\_mode())

if puzzle\_mode == "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])

    user\_puzzle = [puzzle\_row\_one, puzzle\_row\_two, puzzle\_row\_three]

    select\_and\_init\_algorithm(user\_puzzle)

from queue import PriorityQueue

import copy

"""

Completed Matrix

"""

goalMatrix = [[1, 2, 3],

[4, 5, 6],

[7, 8, 0]]

movesMatrix = [[2, 3, 2],

[3, 4, 3],

[2, 3, 2]]

"""

Trivial Matrix

"""

easy = [[1, 2, 0],

[4, 5, 3],

[7, 8, 6]]

queueSize = [] #tracks size of queue per call

nodes = 0

"""

Class for the board.

h = heuristic

g = history

board = node board

"""

class boardNode:

def \_\_init\_\_(self, userBoard):

self.weight = 0

self.h = 0

self.g = 0

self.board = []

#initialize node board

if(userBoard is not None):

self.board = userBoard.copy()

"""

overloading the less tha operator

"""

def \_\_lt\_\_(self, other):

selfP = self.h

return selfP

"""

Function to clone the board and add it to the queue

"""

def clone(self):

newBoard = []

newBoard = copy.deepcopy(self.board)

return newBoard

"""

This function switches blank in the specified position

"""

def swap(self, rowBlank, colBlank, rowMove, colMove):

self.board[rowBlank][colBlank] = self.board[rowMove][colMove]

self.board[rowMove][colMove] = 0 # put blank in new spot

"""

Get the coordinates of the blank tile

"""

def getBlank(self):

for r in range(3):

for c in range(3):

if (self.board[r][c] == 0):

return r, c

"""

Move the blank of tile

"""

def moveBlank(self, move):

blankRow,blankCol = self.getBlank()

if(move == "up"):

self.swap(self, blankRow,blankCol)

def setBoard(self, newBoard):

for r in range(3):

for c in range(3):

self.board[r][c] = newBoard[r][c]

"""

Print the board for the user to see

"""

def PrintBoard(self):

boardStr = ""

for r in range(3):

for c in range(3):

boardStr += str(self.board[r][c])

boardStr += ", "

boardStr += "\n"

return boardStr

"""

This is the driver class for the board solver

"""

class puzzle:

def \_\_init\_\_(self, board):

self.heuristicTotal = 0

self.manhattanTotal = 0

self.mainPQ = PriorityQueue()

#data structure for the tile will be an adjacency matrix

self.board = []

#initialize the matrix (root board)

for i in range(3):

self.board.append(board[i][:])

def getBoard(self):

retBoard = []

retBoard = self.board

return retBoard

"""

Get the coordinates of the blank tile

"""

def getBlank(self, newBoard):

for r in range(3):

for c in range(3):

if(newBoard[r][c] == 0):

return r,c

def getNodesMade(self, row, col):

return movesMatrix[row][col]

"""

Get the new children made based on the number of nodes

def setBoard(self, newBoard)

"""

def GetChildren(self, newBoard):

childrenList = []

blankRow, blankCol = newBoard.getBlank()

"""

Look for blank location

Clone the board as added to the child

After cloning, modify so it replects the move made

"""

if(blankRow == 0 and blankCol == 0):

"""

make 2 clones, one down and one right

"""

boardDown = boardNode(newBoard.clone())

boardRight = boardNode(newBoard.clone())

boardDown.swap(0, 0, 1, 0)

boardRight.swap(0, 0, 0, 1)

childrenList.append(boardDown)

childrenList.append(boardRight)

if(blankRow == 0 and blankCol == 1):

boardDown = boardNode(newBoard.clone())

boardRight = boardNode(newBoard.clone())

boardLeft = boardNode(newBoard.clone())

boardDown.swap(0, 1, 1, 1)

boardRight.swap(0, 1, 0, 2)

boardLeft.swap(0, 1, 0, 0)

childrenList.append(boardDown)

childrenList.append(boardRight)

childrenList.append(boardLeft)

if (blankRow == 0 and blankCol == 2):

"""

make 2 clones, one down and one right

"""

boardDown = boardNode(newBoard.clone())

boardLeft = boardNode(newBoard.clone())

boardDown.swap(0, 2, 1, 2)

boardLeft.swap(0, 2, 0, 1)

childrenList.append(boardDown)

childrenList.append(boardLeft)

if (blankRow == 1 and blankCol == 0):

boardUp = boardNode(newBoard.clone())

boardDown = boardNode(newBoard.clone())

boardRight = boardNode(newBoard.clone())

boardUp.swap(1, 0, 0, 0)

boardDown.swap(1, 0, 2, 0)

boardRight.swap(1, 0, 1, 1)

childrenList.append(boardUp)

childrenList.append(boardRight)

childrenList.append(boardDown)

if (blankRow ==1 and blankCol == 1):

boardUp = boardNode(newBoard.clone())

boardDown = boardNode(newBoard.clone())

boardRight = boardNode(newBoard.clone())

boardLeft = boardNode(newBoard.clone())

boardUp.swap(1, 1, 0, 1)

boardDown.swap(1, 1, 2, 1)

boardRight.swap(1, 1, 1, 2)

boardLeft.swap(1, 1, 1, 0)

childrenList.append(boardUp)

childrenList.append(boardRight)

childrenList.append(boardDown)

childrenList.append(boardLeft)

if (blankRow == 1 and blankCol == 2):

boardUp = boardNode(newBoard.clone())

boardDown = boardNode(newBoard.clone())

boardLeft = boardNode(newBoard.clone())

boardUp.swap(1, 2, 0, 2)

boardDown.swap(1, 2, 2, 2)

boardLeft.swap(1, 2, 1, 1)

childrenList.append(boardUp)

childrenList.append(boardDown)

childrenList.append(boardLeft)

if (blankRow == 2 and blankCol == 0):

boardUp = boardNode(newBoard.clone())

boardRight = boardNode(newBoard.clone())

boardUp.swap(2, 0, 1, 0)

boardRight.swap(2, 0, 2, 1)

childrenList.append(boardUp)

childrenList.append(boardRight)

if (blankRow == 2 and blankCol == 1):

boardUp = boardNode(newBoard.clone())

boardRight = boardNode(newBoard.clone())

boardLeft = boardNode(newBoard.clone())

boardUp.swap(2, 1, 1, 1)

boardRight.swap(2, 1, 2, 2)

boardLeft.swap(2, 1, 2, 0)

childrenList.append(boardUp)

childrenList.append(boardRight)

childrenList.append(boardLeft)

if (blankRow == 2 and blankCol == 2):

boardUp = boardNode(newBoard.clone())

boardRight = boardNode(newBoard.clone())

boardLeft = boardNode(newBoard.clone())

boardUp.swap(2, 2, 1, 2)

boardLeft.swap(2, 2, 2, 1)

childrenList.append(boardUp)

childrenList.append(boardLeft)

for child in childrenList:

child.g = newBoard.g + 1

return childrenList

"""

Get the manhattan distance

"""

def getManhattanDistance(self,childBoard):

manhattan = 0

# get value of child

# find the correct index

# make the calculation

for r in range(3):

for c in range(3):

if (childBoard.board[r][c] != goalMatrix[r][c]):

actualR, actualC = self.findInGrid(childBoard.board[r][c])

manhattan += (abs(r - actualR) + abs(c - actualC))

return manhattan

"""

find the correct spot in the goal matrix

"""

def findInGrid(self,valueToFind):

for r in range(3):

for c in range(3):

if(goalMatrix[r][c] == valueToFind):

return r,c

"""

Get the number of misplaced tiles

"""

def getMisplacedTiles(self, childBoard):

matchCount = 0

for r in range(3):

for c in range(3):

if (childBoard.board[r][c] != goalMatrix[r][c]):

matchCount += 1

# return true or false if there were any missmatches

return matchCount

"""

Priority Queue Function for the search algorithm

children is a list of child

"""

def queueingFunctionUniformed(self, nodes, children):

thisset = set((children))

for child in children:

#if child is not repeated

if(child in thisset):

#set the value of h

child.h = 0

# put(key, val)

nodes.put((int(child.g + child.h), child))

return nodes

"""

Priority Queue Function for the search algorithm

"""

def queueingFunctionMisplacedTile(self, newBoard, children):

pq = PriorityQueue()

for child in children:

child.h = self.getMisplacedTiles(child)

#put(key, val)

pq.put((child.g + child.h, child))

return pq

"""

Priority Queue Function for the mahattan A\* search algorithm

"""

def queueingFunctionManhattan(self, nodes, children):

pq = PriorityQueue()

for child in children:

child.h = self.getManhattanDistance(child)

#put(key, val)

pq.put((child.g + child.h, child))

return pq

"""

create the queue of states

"""

def makeQueue(self, initialBoard):

pq = PriorityQueue()

pq.put((0, initialBoard))

return pq

"""

check if the board is equal to the desired one

"""

def isGoalState(self, currentState):

matchCount = 0

for r in range(3):

for c in range(3):

if (currentState.board[r][c] != goalMatrix[r][c]):

matchCount += 1

# return true or false if there were any missmatches

return matchCount

"""

This solves the puzzle using Uniformed cost algorithm

refactor to this def uniformedSearch(self, newBoard, searchType):

newBoard -- board object

"""

def uniformedSearch(self, newBoard, searchType):

nodes = PriorityQueue()

nodes = self.makeQueue(newBoard)

while(1):

if(nodes.empty()):

return None

queueSize.append(nodes.qsize()) #get the sizes of the queue

key, node = nodes.get()

print(node.PrintBoard())

if(self.isGoalState(node) == 0):

return node

"""

Here we perform the search based on the one selected by user

"""

if(searchType == "1"):

nodes = self.queueingFunctionUniformed(nodes, self.GetChildren(node))

elif (searchType == "2"):

nodes = self.queueingFunctionMisplacedTile(nodes, self.GetChildren(node))

else:

nodes = self.queueingFunctionManhattan(nodes, self.GetChildren(node))