**Word Finder**

Carmen Nicholson and Eboni Williams

CSC 325-002

For our assignment we used the Depth First Search (DFS) to create a “Word Finder”. In our program, a matrix (by default: 8 x 8) is generated in order to find words entered into it. The matrix consists of a random assortment of letters, similar to that of a crossword puzzle. Using DFS, the matrix is searched for all the unique characters in each string of text given. We used the DFS in order to have the ability to keep a list of the previously visited nodes and checks all of the adjacent options. The program created has bigger applications. For instance, the program could someday be used to create a word search generator. The program would be the first step. When using a large dictionary, words that won’t be included in the list will be ones whose characters aren’t all found in the matrix. The next step would be to include a program that stores the successful matches before searching the matrix to see if the words exist (most likely using a nodal system and BFS).

**Goal**

The goal of our implementation is to find all the unique characters that exist in an alpha string of text as efficiently as possible. The main constraint resides in the fact that the characters are randomly generated. Therefore, a small a matrix (e.g. 5 x 5) may not contain all the characters in a given string of text. In addition to this, the string assumes that only characters from the English alphabet are included in the text. Any other characters will result in a “Not all characters found” output.

**Objectives**

The program’s objective is to find all the unique characters in a string using DFS. DFS’s objective is to use color in order to keep track of the vertices that have been visited. Once the search reaches a dead end, it back tracks and tries another path using a stack to get track of its location on the path.

**Tasks**

Both members contributed to wholly to the assignment. Collectively, Eboni and Carmen came up with the idea for “Word Finder” program and discussed the best way to implement it. Eboni was responsible for writing the program with input from Carmen how to do the dfs algorithm, matrix generation, and moves regulation. Eboni worked on error handling and integration; i.e., the functions that transformed the input into usable forms. For example, matches(), getLetters(), and checkFinished(). Carmen did most of the write up with contribution from Eboni on what to include.

**Implementation in Python**

#The function times how long it takes to find all the letters in an alphanumeric sequence.

from random import \*

from pythonds import Graph, Vertex

from string import ascii\_lowercase

#---------------------PUZZLE GENERATOR---------------------------------

# creates a n-by-n puzzle consisting or random letters

def puzzleGen(size = 8):

puzzle = [[choice(ascii\_lowercase) for c in range(size)] for r in range(size)]

return puzzle

#creates the graph that will be used to run DFS

def createBoard(size = 8):

game = Graph()

for row in range(size):

for col in range(size):

ID = (row\*size) + col

pos = moves(row,col,size)

for i in pos:

nid = (i[0]\*size) + i[1]

game.addEdge(ID, nid)

return game

#matches the cell in the generated puzzle with the

#node in the generated graph of a n-by-n puzzle

def matches(grid, size = 8):

pairs = {}

vertex = 0

for row in range(size):

for col in range(size):

pairs[vertex] = grid[row][col]

vertex += 1

return pairs

#------------------------------SEARCH TERMS----------------------------------------

#take the given input and breaks it down into

#the unique characters that need to be found

def getLetters(string):

string = string.split(" ")

search = {}

for index in range(len(string)):

for i in range(len(string[index])):

search[string[index][i]]= False

return search

#checks if all the search values have been found

def checkFinished(charList):

finished = True

for key in charList:

if not charList[key]:

finished = False

break

return finished

#-------------------------------------MOVES REGULATION-----------------------------------------

#creates the moves that are allowed in the game

#uses the same setup as the knight's problem

def moves(x, y, size):

newMoves = []

offsets = [ (-1, -1), (-1, 0), (-1, 1),

(0, -1), (0, 1),

(1, -1), (1, 0), (1, 1)]

for m in offsets:

X = x + m[0]

Y = y + m[1]

if legalCoord(X, size) and legalCoord(Y, size):

newMoves.append((X, Y))

return newMoves

#checks whether the coordinates exists on the board

def legalCoord(x, size):

if x >= 0 and x < size: return True

else: return False

#------------------------------DEPTH FIRST SEARCH------------------------------------

def play(graph, string):

search = getLetters(string)

puzzle = puzzleGen(8)

couples = matches(puzzle)

for aVertex in graph:

aVertex.setColor('white')

aVertex.setPred(-1)

for aVertex in graph:

if aVertex.getColor() == 'white':

if search.get(couples[aVertex.id]) != None:

search[couples[aVertex.id]] = True

dfsvisit(graph, aVertex)

if checkFinished(search): break

if not checkFinished(search):

print("Not all characters found for '",string,"'")

else:

print("All characters for '",string,"' were found.")

def dfsvisit(graph, startVertex):

startVertex.setColor('gray')

for nextVertex in startVertex.getConnections():

if nextVertex.getColor() == 'white':

nextVertex.setPred(startVertex)

dfsvisit(graph, nextVertex)

startVertex.setColor('black')

#---------------------------MAIN------------------------------------

#There are obvious errors in this code.

#So, output generated 3 times.

for i in range(3):

play(createBoard(), "eboni")

play(createBoard(), "carmen")

play(createBoard(), "pradeep")

play(createBoard(), "chowriappa")

print()