Lab 7 : Graph Search Functions

Steven Robles *

University of Texas, El Paso, TX

April 30, 2019

This paper aims to demonstrate implementations of a disjoint set forest and graph search functionality to solve a randomly produced maze. The three different types of graph search function utilized and analyzed in this lab is the Breath First implementation, Depth First based on stacks instead of queses, and a recursive Depth First Search. By timing the performance of each function, a conclusion is drawn on which of these tree implementations is the proffered method.

Contents

I Introduction **1**

II Design **2**

A Part 1: Breadth First Search (BFS) 2

B Parts 2: Depth First Search Function (DFS) - Stack 2

C Parts 3: Depth First Search Function (DFS) - Recursion 2

III Results **2**

IV Discussion **3**

V Source Code **4**

VI Academic Dishonesty **9**

I. Introduction

A graph is a set of vertices with a specific amount of edges which connects particular vertices together. It can be represented in three different forms: Matrix Form, Adjacency List, and an Edge Lists. All three methods capitalizes on existing edges, thus representing connected vertices. This data structure is commonly found when representing individual objects or items with existing relations between them. A notable example is how a friends list from an individual will be represented on a broader scale.

In this lab, we will implement and demonstrate the three different approaches of search function through the graph. The maze will be randomly generated using a disjoint set forest approach which is similar to the previous lab. The graph built for this lab will be based on the number of cells within the maze. Each cell represents a vertices within the graph. Each edge which connects the vertices represents a path between two cells which is not obstructed by an existing wall. The three search functions will solve the maze and compared according to time performances. The resulting path will be printed along the maze to verify the results.

*ID Number: 80678755

II. Design

This lab is based upon constructing a maze with a given set forest and solving it using a graph. The size of the graph depends on the number of cells within the maze. The size of the maze and the number of walls to be removed is up for user discretion, but it also affects of the number of paths, if they exits, that the program provides. If the number of wall to be removed is less than the number cells -1, then there is no guaranteed path form the cell 0 to cell n-1. If the number of walls to removed is exactly n-1, then there is a guaranteed unique path. Anything over produces more than one path. Of course, the basic design of the maze is based upon the previous lab.

A graph is then produced based on the exiting lists containing the wall within the maze. It is built in adjacency form and marks edges between vertices given that it is not located within the walls lists. With the graph produces, different methods can be utilized in order to solve the maze creating a path. The following parts showcases three different methods.

A. Part 1: Breadth First Search (BFS)

The Breadth First Search algorithm utilizes queues to transverses through the graph. The design follows similar approach to any to the basic BFS functions. This methods takes *0(—v— + —E—)* and is predicted to share the same execution time as the DFS function.

B. Parts 2: Depth First Search Function (DFS) - Stack

The Depth First Search algorithm utilizes the stack method in order to transverse through the graph. The design follows the same logic as in part 1, except this time the search algorithm searches the path in question until it reaches the final cell or cannot go further due to obstructed walls. In terms of time performance, it is predicted they it will have the same, if not similar outcomes at in part 1 since the time complexity of this function is *0(—v— + —E—)*.

C. Parts 3: Depth First Search Function (DFS) - Recursion

The recursion approach to the depth first function was a bit more challenging than the previous parts. In this method although, the function passes the array in which contains the path cells at each recursions call. This enables to only append to the specific list only when the path is found. The resulting list will only contain the cell numbers which the final path takes effectively making the drawing of the path easier. This is why the maze is only drawn with this approach.

III. Results

After the implementations of both disjoint set forest methods and all three graph search functions were successful, each functions were executed and recorded for time performance. The time values were used to evaluate each program and compares each data structure with one another. Below are their time executions and over all statistical results.

Figure 1: Standard Maze

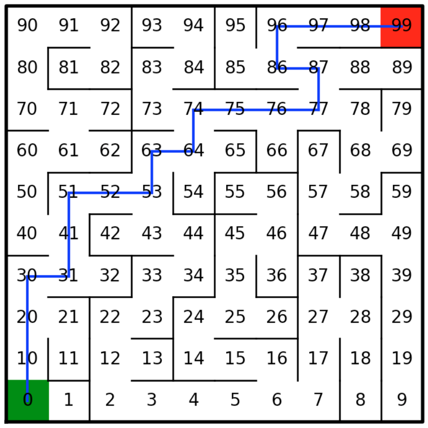
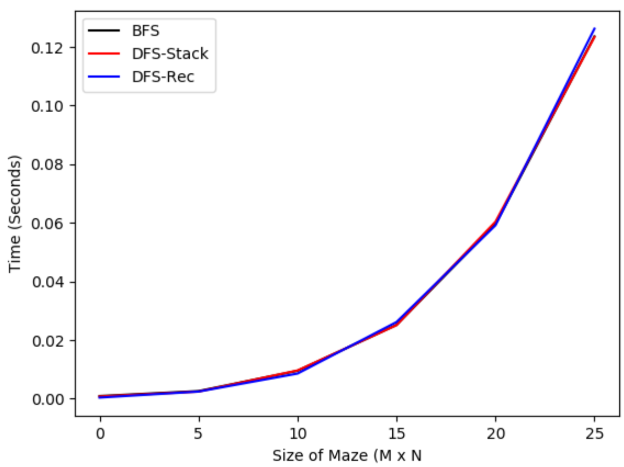


Figure 2: Run Times



IV. Discussion

Overall, implementing all three methods to to search a given constructed maze results in a successful path computation. There is a a small difference between all there methods, but the recursion method turned out to take the most time. From *Figure 3*, we can conclude that the discrepancy that both BFS and DFS do share the same time performance. A side note that must be written along the results is that the time results do not include the time it takes to draw the resulting path within the maze.

V. Source Code

```python
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 04/29/2019
Discreption: Lab 7:
        This program is desinged to act as the main file of this project. It produces the main menu and recieves
        user input to call funcitons. It also records the time it takes for each maze to be built.


"""

from BuildMaze import initiateMazeC
import matplotlib.pyplot as plt
import numpy as np
import timeit


loop = True #commences the loop

#the is the while loop which pompts the user.
while loop:
  print("1. Build and Graph Maze\n2. Time Trail")
  number = input("3. Exit\n")
  print("************************")
  #trys converting the input into an int. if it fails, the pormpt runs again
  try:
    choice = int(number)
  except:
    choice = -1

  #The fist if statement bulds the maze by asking of its dimensions first
  if choice == 1:
    cont = True
    while cont:
      rows = input("Enter value for rows : \n")
      cols = input("Enter value for columns : \n")
      remove = input("Enter number of rows to be removed : \n")
      #the following converts the input into ints if it's possible
      try :
        rows = int (rows)
        cols = int(cols)
        remove = int(remove)
        if remove < 0:
          print("Try Again")
        else :
          cont = False

      except:
        print("Try Again")
    print()
    if remove < rows*cols -1:
      print('A path from source to destination is not guaranteed to exist')
    elif remove == rows*cols -1:
      print('The is a unique path from source to destination')
      initiateMazeC(rows, cols,start, end, True)
    else:
      print('There is at least one path from source to destination')

  #Choice number two times the preformance of the functions with a determined
  #set of sisze mazes.
  elif choice == 2:
    dimensions = [[5,5],[10,10],[15,15],[20,20],[25,25],[30,30]]
    times = [[],[],[]]
```

```
    for i in range(len(dimensions)):
      #times the BFS
      start = timeit.default_timer() # starts timer
      initiateMazeC(dimensions[i][0], dimensions[i][1], 0, (dimensions[i][0]*dimensions[i][1])-1, False, 1, True)
      stop = timeit.default_timer() # ends timers
      times[0].append(stop-start)
      #times the DFS _ Stack
      start = timeit.default_timer() # starts timer
      initiateMazeC(dimensions[i][0], dimensions[i][1], 0, (dimensions[i][0]*dimensions[i][1])-1, False, 2, True)
      stop = timeit.default_timer() # ends timers
      times[1].append(stop-start)
      #times hte DFs _ Recursion
      start = timeit.default_timer() # starts timer
      initiateMazeC(dimensions[i][0], dimensions[i][1], 0, (dimensions[i][0]*dimensions[i][1])-1, False, 3, True)
      stop = timeit.default_timer() # ends timers
      times[2].append(stop-start)

    fig, ax = plt.subplots()
      #proceeds to plot the time results
    plt.xlabel('Size of Maze (M x N')
    plt.ylabel('Time (Seconds)')
    x = np.arange(0,30,5)
    plt.plot(x, times[0], 'k', label= 'BFS')
    plt.plot(x, times[1], 'r', label='DFS-Stack')
    plt.plot(x, times[2], 'b', label='DFS-Rec')
    plt.legend()
    plt.savefig('RunTimes')
    plt.show()
  #program exits
  elif choice == 3:
    print("Good Bye!")
    loop = False
  else:
    print("Try Again")
  print("************************")
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 04/29/2019
Discreption: Lab 7:
        The purpose of this program is to build construct a adjecancy lists for a graph based on the
        maze matrix. This graph will be utilized to solve the mazes in three different approcahes:
        BFS, DFS - Stacked, and DFS - Recursive

"""
import math

#bulds the graph giaven the walls lists
def buildGraph(walls, rows, columns):
  G = [[] for i in range(rows*columns)]
  v = 0
  #checks if there is a wall between to cells before making an edge
  while v <rows*columns:
    if (v + 1) %columns != 0 and [v, v+1] not in walls and [v+1, v] not in walls:
      G[v].append(v+1)
    if v %columns != 0 and [v, v-1] not in walls and [v-1, v] not in walls:
      G[v].append(v-1)
    if v - columns >= 0 and [v, v-columns] not in walls and [v-columns, v] not in walls:
      G[v].append(v-columns)
    if v +columns < rows*columns and [v, v+columns] not in walls and [v+columns, v] not in walls:
      G[v].append(v+columns)
```

```
    v+=1
  return G

#BFS method approach
def BFS(G, v):
  Q = []
  visited = [False]*(len(G))
  prev = [-1] * (len(G))
  Q.append(v)
  visited[v] = True
  while len(Q) > 0 :
    u = Q.pop(0)
    for t in G[u]:
      if visited[t]== False:
        visited[t] = True
        prev[t] = u
        Q.append(t)
  return prev

#DFS stack approach
def DFS_Stack(G, v):
  Q = []
  visited = [False]*(len(G))
  prev = [-1] * (len(G))
  Q.append(v)
  visited[v] = True
  while len(Q) > 0 :
    u = Q.pop()
    for t in G[u]:
      if visited[t]== False:
        visited[t] = True
        prev[t] = u
        Q.append(t)
  return prev

#DFS recursion approach
def DFS_Rec(G, source, visited, prev, final):
  visited[source] = True
  if source == final:
    return prev, True
  for t in G[source]:
    if visited[t] == False:
      prev, add = DFS_Rec(G, t, visited, prev, final)
      if add:
        prev.append(t)
        return prev, True
  return prev, False

#builds the graph in an adjecancy list for the other methods
def mazeGraph(walls, rows, columns):
  G = buildGraph(walls, rows, columns)
  return G
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 04/29/2019
Discreption: Lab 7:
        The purpose of this program is to build a maze with any given number of cells and make them all
        one complex cell. This will is done by created a disjoint set forest making all of the single cells
        belong to one tree. Resutls are shown and the preformances are timed. A solution path is also calculated
        by sovling it int terms of grapsh using BSF and DSF methods. They are ploted wihtin the maze.

"""
```

```python

#from Graph import DFS
from Graph import mazeGraph
from Graph import DFS_Rec
from Graph import DFS_Stack
from Graph import BFS
import matplotlib.pyplot as plt
import numpy as np
import random

#the following function is the intial call for the iterative function which solves the maze
def solveMaze(walls, rows, columns, start, finish, trackNum = 0):

    G = mazeGraph(walls, rows, columns)
    print("Adjecancy List: \n", G)
    print('Final Path For Breath Frist Search: \n', BFS(G, 0))
    print('Final Path For Detph First Search - Stack: \n', DFS_Stack(G, 0))
    finalPath, dud = DFS_Rec(G, 0, [False]*(len(G)), [], (rows*columns) -1)
    finalPath.append(0)
    print('Final Path For Detph First Search - Recursion: \n', finalPath)
    convertedPath = []
    #the solution path are convereted to x and y coordinates to plot the blue lines for the path
    for i in range(len(finalPath)):
        if i > 0:
            c1 = finalPath[i] %columns
            r1 = (finalPath[i]-c1) /columns
            c2 = finalPath[i-1]%columns
            r2 = (finalPath[i-1]-c2) /columns
            convertedPath.append([[c1 +.5, c2+.5], [r1+.5, r2 +.5]])
    return convertedPath


#plots the maze as provided in class
def draw_maze(walls,maze_rows,maze_cols, trackNum=0, start=0, finish=0, cell_nums=True):
    finalPath = solveMaze(walls, maze_rows, maze_cols, start, finish, trackNum)
    fig, ax = plt.subplots()
    for w in walls:
        if w[1]-w[0] ==1: #vertical wall
            x0 = (w[1]%maze_cols)
            x1 = x0
            y0 = (w[1]//maze_cols)
            y1 = y0+1
        else:#horizontal wall
            x0 = (w[0]%maze_cols)
            x1 = x0+1
            y0 = (w[1]//maze_cols)
            y1 = y0
        ax.plot([x0,x1],[y0,y1],linewidth=1,color='k')
    sx = maze_cols
    sy = maze_rows
    ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k')
    c1 = start%maze_cols
    r1 = (start-c1) /maze_cols
    c2 = finish%maze_cols
    r2 = (finish-c2) /maze_cols
    #fills the start and end cells with green adn red coolors
    ax.fill([c1, c1+1, c1+1, c1, c1], [r1, r1, r1+1, r1+1, r1], color = 'g')
    ax.fill([c2, c2+1, c2+1, c2, c2], [r2, r2, r2+1, r2+1, r2], color = 'r')
    ax.axis('off')
    ax.set_aspect(1.0)
    for i in range(len(finalPath)): #plots the solution path
        ax.plot(finalPath[i][0], finalPath[i][1], color = 'b')
    if cell_nums:
        for r in range(maze_rows):
            for c in range(maze_cols):
                cell = c + r*maze_cols
                ax.text((c+.5),(r+.5), str(cell), size=10,
                        ha="center", va="center")
    ax.axis('off')
```

```python
    ax.set_aspect(1.0)
    plt.show()

#bulds the wall lists as porvided
def wall_list(maze_rows, maze_cols):
    # Creates a list with all the walls in the maze
    w =[]
    for r in range(maze_rows):
        for c in range(maze_cols):
            cell = c + r*maze_cols
            if c!=maze_cols-1:
                w.append([cell,cell+1])
            if r!=maze_rows-1:
                w.append([cell,cell+maze_cols])
    return w

#initializes the disjoint set forest list
def DisjointSetForest(size):
    return np.zeros(size,dtype=np.int)-1

#comppressed find fucntion where all of the cells point
#direclty to its root
def findC(S,i):
    if S[i] < 0:
        return i
    S[i] = findC(S, S[i])
    return S[i]

#joins trees together depending on thier existing sizes
def unionBySize(S, i, j):
    ri = findC(S, i)
    rj = findC(S, j)
    if ri!= rj:
        if S[ri] < S[rj]:
            S[ri] += S[rj]
            S[rj] = ri
        else:
            S[rj] += S[ri]
            S[ri] = rj

#returns a boolean if two cells belong to the same tree or not for compressed
def checkPathC(S, walls, d):
    if findC(S, walls[d][0]) == findC(S, walls[d][1]):
        return False
    return True

#buids a maze based upon the compressed method
def initiateMazeC(maze_rows, maze_cols,start=0, end=0 ,draw=False, trackNum = 0, time =
    False):
    #initializes the components to build the mazw
    walls = wall_list(maze_rows,maze_cols)
    T = DisjointSetForest(maze_rows * maze_cols)
    setCount2 = len(T)
    #randomly removes a wall until all cells are connected
    while setCount2 > 1:
        d = random.randint(0,len(walls)-1)
        if checkPathC(T, walls, d):
            unionBySize(T, walls[d][0], walls[d][1])
            walls.pop(d)
            setCount2 -=1
    if draw:
        draw_maze(walls,maze_rows,maze_cols, trackNum, start, end)
    if time:
        G = mazeGraph(walls, maze_rows, maze_cols)
        if trackNum == 1:
            BFS(G, 0)
            return
        if trackNum == 2:
            DFS_Stack(G, 0)
            return
```

```
        if trackNum == 3:
            DFS_Rec(G, 0, [False]*(len(G)), [], (maze_rows*maze_cols) -1)
            return
```

VI. Academic Dishonesty

Scholastic Dishonesty

Any student who commits an act of scholastic dishonesty is subject to discipline. Scholastic dishonesty includes, but not limited to cheating, plagiarism, collusion, the submission for credit of any work or materials that are attributable to another person.

- **Cheating**
  - Copying form the test paper of another student
  - Communicating with another student during a test
  - Giving or seeking aid from another student during a test
  - Possession and/or use of unauthorized materials during tests (i.e. Crib notes, class notes, books, etc)
  - Substituting for another person to take a test
  - Falsifying research data, reports, academic work offered for credit
- **Plagiarism**
  - Using someone's work in your assignments without the proper citations
  - Submitting the same paper or assignment from a different course, without direct permission of instructors
- **Collusion**
  - Unauthorized collaboration with another person in preparing academic assignments

Sign:  ______________________ **Date:__04/30/2019____**