Lab 6 : Disjoint Set Forest Maze

Steven Robles *

University of Texas, El Paso, TX

April 14, 2019

This paper aims to demonstrate implementations of a disjoint set forest and compare the two different methods of approach : standard and compressed. To demonstrate these implementations, a maze is constructed utilizing a disjoint forest. The time it takes for each method to complete constructing the maze in recorded for the comparison of performance between the two .

Contents

I. Introduction

A disjoint set forest is a collection of trees that are stored in an array, or in this case in a python native lists. Each index points to the root node which creates the tree itself. In this lab, that value of a root will be represented with a negative value. Having a negative value larger than one indicates the size of the tree. This is useful information when it comes to implementing union by size.

In this lab, we will implement and demonstrate the two approaches of construction a disjoint set forest. One will be designed to use standard union and search while the other will use union by size and compressed search. As hinted in the previous paragraph, the difference between standard union and union by size is that the standard union will join tree B to tree A regardless of size while union by size will always add the smaller tree onto the larger one. Compressed search is similar to standard search excepts it modifies the tree so that each node will directly refer to the root. This lab will take illustrate these methods by building a maze. The maze will be completed only when there is one single path between any two given locations, thus resulting in one singular tree.

*ID Number: 80678755

II. Design

This lab is based upon constructing a maze given a disjoint set forest. The size of the forest is specified upon the user discretion. For simplicity, we will be working with a 10x15 sized maze. The initial calling of the disjoint forest will fill all 150 slots with -1's to indicate they are all individual tree roots. To build the maze, the program will remove a wall given that both cells separated by the wall belong to two different trees. This process is repeated until there is one single tree left.

A. Part 1: Standard Implementation

The first approach in building this maze is to utilize the standard method of union and search functions. As mentioned before, standard union adds tree B to tree A without regarding their sizes while the standard search function searched for the root tree by transversing each child. Both methods were utilized when removing a wall, checking that both cells of either side of the wall belong to different trees. This process is repeated until there is one single tree with one unique path between any two given cells. To reassure this, a counter is tracked which had an initial value of the number of trees (cells) there are an decrease by one every time a wall is removed. The whole method takes $O(n^2)$.

B. Parts 2: Compressed Implementation

The second approach in building the maze is to utilize the compressed method of union and search functions. The compressed method of union is also referred to as union by size. This means that the smaller tree will always be added to the larger. This means that the sizes of the trees within the disjoint set forest must be tracked. Since a negative value indicates that a particular cell is the root of the tree, the size of the new tree can be stored by subtracting the root value of the joining tree. For example, a tree with a root value of *-1* has a tree size of 1 while a tree with a root value of *-5* has a tree size of five cells.

The compressed search method is also utilized in this method. Instead of just finding the root like in the standard definition of the search function within a disjoint set forest, the compressed method makes each cell direct refer to its root. This is done with an iterative loop where the root is found first and as it is passed back to its initial call, it updates the pointer of every child within the tree. This method reduces the time when finding the roots of a tree later in the program for either union by size or figuring of two given cells belong to the same tree. The whole process takes $0(n)$.

C. Parts 3: Path Finder

An addition was added to the lab which solves the randomly produced maze. The solve the maze, an iteration function was called. The iteration searched for the path by checking for existing walls between cell numbers. A problem aroused when a path overlapped itself a few times, so a list was created and contained all visited cells which disallowed back tracking. Once the path was found (when the current cell reaches the value of the final cell), the list which appended all of the steps is then plotted as a blue line between the cells.

III. Results

After the implementations of both disjoint set forest methods 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

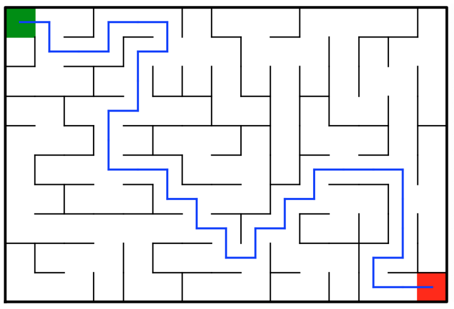


Figure 2: Compressed Maze

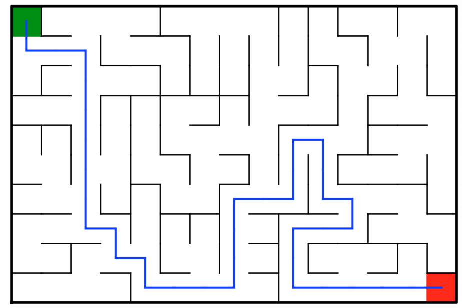
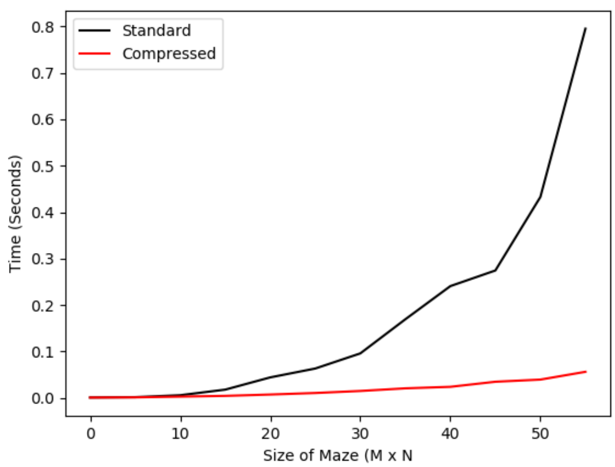


Figure 3: Run Times



IV. Discussion

Overall, implementing both methods to construct the maze from a disjoint set forest is successful and handles all of the computation well. There is a difference between time performances between the standard and compressed methods. From *Figure 3*, we can conclude that the discrepancy between the two methods increases by as the maze cell increases. It also shows that the compressed method of the disjoint set forest is exceeds in time performance which my be critical as large mazes require more computations and costs.

V. Source Code

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 04/12/2019
Discreption: Lab 6:
        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 initiateMaze
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")
      #the following converts the input into ints if it's possible
      try :
        rows = int (rows)
        cols = int(cols)
        cont = False

      except:
        print("Try Again")
    cont = True
    while cont: #this second while loop retrieves the valid start end points
      start, end = -1,-1
      print("Enter starting point between 0 and ", rows * cols -1, end = '')
      start = input(" :\n")
      print("Enter ending point between 0 and ", rows * cols -1, end = '')
      end = input(" :\n")
      try :
        start = int (start)
        end = int(end)
      except:
        print("Try Again")
      if start < 0 or start >= rows * cols or  end < 0 or end >= rows * cols:
        print("Try Again")
      else:
        cont = False

    initiateMaze(rows, cols, start, end, True)
```

```
    initiateMazeC(rows, cols, start, end, True)

  #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],[35,35],[40,40],[45,45],[50,50],[55,55],[60,60]]
    times = [[],[]]
    for i in range(len(dimensions)):
      start = timeit.default_timer() # starts timer
      initiateMaze(dimensions[i][0], dimensions[i][1])
      stop = timeit.default_timer() # ends timers
      times[0].append(stop-start)
      start = timeit.default_timer() # starts timer
      initiateMazeC(dimensions[i][0], dimensions[i][1])
      stop = timeit.default_timer() # ends timers
      times[1].append(stop-start)
    fig, ax = plt.subplots()
      #proceeds to plot the time results
    plt.close()
    plt.xlabel('Size of Maze (M x N')
    plt.ylabel('Time (Seconds)')
    x = np.arange(0,60,5)
    plt.plot(x, times[0], 'k', x, times[1], 'r')
    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: 04/12/2019
Discreption: Lab 6:
        The purpse of this program is to build a maze with 150 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 and ploted wihtin the maze

"""


import matplotlib.pyplot as plt
import numpy as np
import random
import timeit

#the followin functions checks if there is an existing wall between the current cell
#and the next intended cell
def checkWall(walls, x, y):
    for i in range(len(walls)):
        if walls[i][0] == x and walls[i][1] == y or walls[i][1] == x and walls[i][0] == y:
            return False #the jump cannot be made
    return True #returns that it is able to jump to the intended cell

#the following iterative function searches for the path between the start and finish points
def calculatePath(walls, path, rows, colums, cell, finish, history):
    if cell == finish: #path is found and returned
        return True, path
    if (cell + 1) %colums != 0: #jumpts to the right cell
```

```
        if checkWall(walls, cell, cell+1) and ((cell+1 in history) == False):
            history.append(cell+1)
            check, path = calculatePath(walls, path, rows, colums, cell+1, finish, history)
            if check:
                path.append([cell, cell+1])
                return True, path
    if cell %colums != 0 and cell: #jumps to the left cell after checking its validity
        if checkWall(walls, cell, cell-1) and ((cell -1 in history) == False):
            history.append(cell-1)
            check, path = calculatePath(walls, path, rows, colums, cell-1, finish, history)
            if check:
                path.append([cell, cell-1])
                return True, path
    if cell - colums >= 0: #jumps to the bottom cell if there is an open path
        if checkWall(walls, cell, cell-colums) and ((cell-colums  in history) == False):
            history.append(cell-colums)
            check, path = calculatePath(walls, path, rows, colums, cell-colums, finish, history)
            if check:
                path.append([cell, cell-colums])
                return True, path
    if cell +colums < rows*colums: #jumps to the top cell if it is vald
        if checkWall(walls, cell, cell+colums) and ((cell +colums in history) == False):
            history.append(cell+colums)
            check , path = calculatePath(walls, path, rows, colums, cell+colums, finish, history)
            if check:
                path.append([cell, cell+colums])
                return True, path
    return False , path

#the following function is the intial call for the iterative function which solves the maze
def solveMaze(walls, rows, columns, start, finish):
    finalPath = []
    history = [start]
    holder, finalPath = calculatePath(walls, finalPath, rows, columns, start, finish, history)
    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)):
        c1 = finalPath[i][0]%columns
        r1 = (finalPath[i][0]-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, start=0, finish=0):
    finalPath = solveMaze(walls, maze_rows, maze_cols, start, finish)
    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
```

```python
    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')
    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

#standard find function to find the root of the tree
def find(S,i):
    # Returns root of tree that i belongs to
    if S[i]<0:
        return i
    return find(S,S[i])

#joins two cell's if they belong to different trees
def union(S,i,j):
    ri = find(S,i)
    rj = find(S,j)
    if ri!=rj:
        S[rj] = ri  # Make j's root point to i's root

#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
def checkPath(S, walls, d):
    if findC(S, walls[d][0]) == findC(S, walls[d][1]):
        return False
    return True


#builds a maze based upon the stndard method
def initiateMaze(maze_rows, maze_cols, start=0, end=0, draw=False):
```

```
    #initializes the components to build the maze
    walls = wall_list(maze_rows,maze_cols)
    S = DisjointSetForest(maze_rows * maze_cols)
    setCount = len(S)
    #randomly removes a wall until all cells are connected
    while setCount > 1:
        d = random.randint(0,len(walls)-1)
        if checkPath(S, walls, d):
            union(S, walls[d][0], walls[d][1])
            walls.pop(d)
            setCount -=1
    if draw:
        draw_maze(walls,maze_rows,maze_cols, start, end)

#buids a maze based upon the compressed method
def initiateMazeC(maze_rows, maze_cols, start=0, end=0 ,draw=False):
    #initializes the components to build the mazw
    walls2 = 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(walls2)-1)
        if checkPath(T, walls2, d):
            unionBySize(T, walls2[d][0], walls2[d][1])
            walls2.pop(d)
            setCount2 -=1
    if draw:
        draw_maze(walls2,maze_rows,maze_cols, start, end)
```

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:__04/14/2019____**