**Lab #7**

CS 2302

Ana Luisa Mata Sánchez

# Introduction

Use a disjoint set forest to build a maze. The maze should contain a collection of cells separated by walls in such a way that there is exactly one simple path (that is, a path that does not visit any cell more than once) separating any two cells.

To build a maze, let M be the number of rows and N be the number of columns of your square maze. When all walls are present, each of the M ∗ N cells in the maze belongs to a different set. Thus you have M ∗ N sets in your disjoint set forest. When you remove a wall, if the cells that were separated by that wall belonged to different sets, you must unite these sets. This process is repeated until all cells belong to a single set; at that point you display the maze.

# Proposed solution design and implementation

**Module 1 – create\_standard\_dsf\_maze**

This method creates a disjoint set forest using the standard union and find functions. In the case that there are more walls to be popped because number of walls>number of cells – 1, it will continue to pop random walls even after a unique path has been found. This method acts as an adjacency list builder as well. As it removes walls, it creates the edge that exists between the cells.

**Module 2 – create\_compressed\_dsf\_maze**

This method creates a disjoint set forest using the compressed union and union by size functions. In the case that there are more walls to be popped because number of walls>number of cells – 1, it will continue to pop random walls even after a unique path has been found. This method acts as an adjacency list builder as well. As it removes walls, it creates the edge that exists between the cells.

**Module 3 – BFS**

This method implements Breadth First Search using a queue and a visited array. While there are still items in the queue, it will remove the current object from the queue and go to each of the connected vertices. As it goes through each vertex, it marks them as visited, adds the current element as the prev and adds them to the queue.

**Module 4 – DFS**

This method implements Depth First Search using a stack and a visited array. While there are still items in the stack, it will remove the current object from the stack and go to each of the connected vertices. As it goes through each vertex, it marks them as visited, adds the current element as the prev and adds them to the stack.

**Module 5 – Recursive DFS**

This method implements Depth First Search using recursion. In this method the visited and prev arrays are global variables. It marks the current vertex as visited, goes to each connected vertex, and adds them to the path and calls the method again with the next vertex.

**Module 6 – draw\_graph**

This method takes the original draw\_graph method and changes the placement of the circles so that they are ordered in the same rows and columns as the maze.

**Module 7 – draw\_maze\_bfs**

This method uses the draw\_maze method and draws the solved path that is obtained with BFS. It draws the lines depending on if the next element is right, left, up or down.

**Module 8 – draw\_maze\_dfs**

This method uses the draw\_maze method and draws the solved path that is obtained with DFS. It draws the lines depending on if the next element is right, left, up or down.

**Module 9 – draw\_maze\_recdfs**

This method uses the draw\_maze method and draws the solved path that is obtained with the recursive DFS. It draws the lines depending on if the next element is right, left, up or down.

# Experimental results

|  |  |  |
| --- | --- | --- |
| **Maze Size (RowxCol) and number of walls** | **Non-compressed output** | **Compressed Output** |
| 8x5  Walls = 38 | A path from source to destination is not guaranteed to exist  ######## Maze using standard find and union ########  Time it took to get BFS path: 0.0  Time it took to get DFS path: 0.0  Time it took to get recursive DFS path: 0.0  Time it took to create the maze: 0.003020048141479492  Maze row size: 8  Maze column size: 5 | ######## Maze using compressed find and union by size ########  Time it took to get BFS path: 0.0  Time it took to get DFS path: 0.0  Time it took to get recursive DFS path: 0.0  Time it took to create the maze: 0.0019948482513427734  Maze row size: 8  Maze column size: 5 |
| 8x5  Walls = 39 | There is a unique path from source to destination  ######## Maze using standard find and union ########  Time it took to get BFS path: 0.0  Time it took to get DFS path: 0.0  Time it took to get recursive DFS path: 0.0  Time it took to create the maze: 0.0060117244720458984  Maze row size: 8  Maze column size: 5 | ######## Maze using compressed find and union by size ########  Time it took to get BFS path: 0.0  Time it took to get DFS path: 0.0  Time it took to get recursive DFS path: 0.0  Time it took to create the maze: 0.0019884109497070312  Maze row size: 8  Maze column size: 5 |
| 8x5  Walls = 40 | There is at least one path from source to destination  ######## Maze using standard find and union ########  Time it took to get BFS path: 0.0  Time it took to get DFS path: 0.0  Time it took to get recursive DFS path: 0.0  Time it took to create the maze: 0.004017353057861328  Maze row size: 8  Maze column size: 5 | ######## Maze using compressed find and union by size ########  Time it took to get BFS path: 0.0  Time it took to get DFS path: 0.0  Time it took to get recursive DFS path: 0.0  Time it took to create the maze: 0.001992940902709961  Maze row size: 8  Maze column size: 5 |

In order to test the running times of the DFS, BFS, and recursive DFS I assumed that the programmed would always receive the number of walls to create a unique path. In an attempt to save time, I also used the compressed maze method.

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Compressed BFS | Compressed DSF | Compressed Recursive DSF |
| 30x30 | 0 | 0.000997543 | 0 |
| 40X40 | 0.000996828 | 0.000982285 | 0.001012087 |
| 50x50 | 0.000997305 | 0.000997782 | 0.000996828 |
| 60X60 | 0.001994848 | 0.001980305 | 0.002024889 |
| 70X70 | 0.002991438 | 0.00299716 | 0.001963139 |
| 80X80 | 0.00399065 | 0.003987789 | 0.002992392 |
| 90X90 | 0.005983114 | 0.005982399 | 0.004987955 |

# Conclusion

According to the tests, BFS is faster than DSF. The recursive DSF is also slower than both BFS and the regular DSF.

**“I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.”**

-Ana Luisa Mata Sánchez

# Appendix

|  |
| --- |
| # Author: Ana Luisa Mata Sanchez |
|  | # Course: CS2302 |
|  | # Assignment: Lab #7 |
|  | # Instructor: Olac Fuentes |
|  | # Description: Program to draw and solve mazes using DFS and BFS |
|  | # T.A.: Anindita Nath, Maliheh Zargaran |
|  | # Last modified: 04/29/2019 |
|  | # Purpose: To compare differences between algorithms |
|  |  |
|  | import matplotlib.pyplot as plt |
|  | import numpy as np |
|  | import random |
|  | import time |
|  |  |
|  | ###################################### Code provided and written by Dr. Fuentes ###################################### |
|  | def DisjointSetForest(size): |
|  | return np.zeros(size,dtype=np.int)-1 |
|  |  |
|  | def dsfToSetList(S): |
|  | #Returns aa list containing the sets encoded in S |
|  | sets = [ [] for i in range(len(S)) ] |
|  | for i in range(len(S)): |
|  | sets[find(S,i)].append(i) |
|  | sets = [x for x in sets if x != []] |
|  | return sets |
|  |  |
|  | def find(S,i): |
|  | # Returns root of tree that i belongs to |
|  | if S[i]<0: |
|  | return i |
|  | return find(S,S[i]) |
|  |  |
|  | def find\_c(S,i): #Find with path compression |
|  | if S[i]<0: |
|  | return i |
|  | r = find\_c(S,S[i]) |
|  | S[i] = r |
|  | return r |
|  |  |
|  | def union(S,i,j): |
|  | # Joins i's tree and j's tree, if they are different |
|  | ri = find(S,i) |
|  | rj = find(S,j) |
|  | if ri!=rj: |
|  | S[rj] = ri |
|  |  |
|  | def union\_c(S,i,j): |
|  | # Joins i's tree and j's tree, if they are different |
|  | # Uses path compression |
|  | ri = find\_c(S,i) |
|  | rj = find\_c(S,j) |
|  | if ri!=rj: |
|  | S[rj] = ri |
|  |  |
|  | def union\_by\_size(S,i,j): |
|  | # if i is a root, S[i] = -number of elements in tree (set) |
|  | # Makes root of smaller tree point to root of larger tree |
|  | # Uses path compression |
|  | ri = find\_c(S,i) |
|  | rj = find\_c(S,j) |
|  | if ri!=rj: |
|  | if S[ri]>S[rj]: # j's tree is larger |
|  | S[rj] += S[ri] |
|  | S[ri] = rj |
|  | else: |
|  | S[ri] += S[rj] |
|  | S[rj] = ri |
|  |  |
|  |  |
|  |  |
|  | 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 |
|  |  |
|  | def draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=False): |
|  | 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') |
|  | 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') |
|  | ax.set\_aspect(1.0) |
|  |  |
|  | ###################################### MY CODE ###################################### |
|  |  |
|  | #Method that removes walls and creates the dsf using the standard union and find methods |
|  | def create\_standard\_dsf\_maze(S,walls,numWalls,numCells): |
|  | G =[] |
|  | for i in range(numCells): |
|  | G.append([]) |
|  | #If there is only one set it means that all cells are reacheable from any cell |
|  | while len(dsfToSetList(S))>1 and numWalls>0: |
|  | #Finds a wall to remove |
|  | d = random.randint(0,len(walls)-1) |
|  | if find(S,walls[d][0]) != find(S,walls[d][1]): |
|  | #make the elements belong to the same set |
|  | union(S,walls[d][0],walls[d][1]) |
|  | G[walls[d][0]].append(walls[d][1]) |
|  | G[walls[d][1]].append(walls[d][0]) |
|  | #remove the wall |
|  | walls.pop(d) |
|  | numWalls = numWalls -1 |
|  | if numWalls>0: |
|  | while(numWalls>0 and len(walls)>0): |
|  | d = random.randint(0,len(walls)-1) |
|  | G[walls[d][0]].append(walls[d][1]) |
|  | G[walls[d][1]].append(walls[d][0]) |
|  | walls.pop(d) |
|  | numWalls = numWalls -1 |
|  | return G |
|  |  |
|  | #Method that removes walls and creates the dsf using the union by size and compressed find methods |
|  | def create\_compressed\_dsf\_maze(SC,wallsC,numWalls,numCells): |
|  | G =[] |
|  | for i in range(numCells): |
|  | G.append([]) |
|  | #If there is only one set it means that all cells are reacheable from any cell |
|  | while len(dsfToSetList(SC))>1 and numWalls>0: |
|  | #Finds a wall to remove |
|  | dC = random.randint(0,len(wallsC)-1) |
|  | #If the elements that share a wall are not in the same set, remove it |
|  | if find\_c(SC,wallsC[dC][0]) != find\_c(SC,wallsC[dC][1]): |
|  | #make the elements belong to the same set |
|  | union\_by\_size(SC,wallsC[dC][0],wallsC[dC][1]) |
|  | G[wallsC[dC][0]].append(wallsC[dC][1]) |
|  | G[wallsC[dC][1]].append(wallsC[dC][0]) |
|  | #remove the wall |
|  | wallsC.pop(dC) |
|  | numWalls = numWalls -1 |
|  | if numWalls>0: |
|  | while(numWalls>0 and len(wallsC)>0): |
|  | dC = random.randint(0,len(wallsC)-1) |
|  | G[wallsC[dC][0]].append(wallsC[dC][1]) |
|  | G[wallsC[dC][1]].append(wallsC[dC][0]) |
|  | wallsC.pop(dC) |
|  | numWalls = numWalls -1 |
|  | return G |
|  |  |
|  | #Method for Breadth First Search inspired by the given graph search pseudocode |
|  | def BFS(G, v): |
|  | prev = np.zeros(len(G),dtype=int)-1 |
|  | visited = [] |
|  | for i in range(len(G)): |
|  | visited.append(False) |
|  | Q = [] |
|  | #Start the queue with the first element |
|  | Q.append(v) |
|  | #Mark first element as visited |
|  | visited[v] = True |
|  | while (Q!=[]): |
|  | #Remove current element from queue |
|  | u = Q.pop(0) |
|  | #Go to connected vertices |
|  | for t in G[u]: |
|  | #If the vertex hasn't been visited |
|  | if(visited[t]==False): |
|  | #Mark as visited |
|  | visited[t] = True |
|  | #Add to path |
|  | prev[t] = u |
|  | #Add to the queue |
|  | Q.append(t) |
|  | #Return path |
|  | return prev |
|  |  |
|  | #Method for Depth First Search inspired by the given graph search pseudocode |
|  | def DFS(G, v): |
|  | prev = np.zeros(len(G),dtype=int)-1 |
|  | visited = [] |
|  | for i in range(len(G)): |
|  | visited.append(False) |
|  | S = [] |
|  | #Start the stack with the first element |
|  | S.insert(0,v) |
|  | #Mark first element as visited |
|  | visited[v] = True |
|  | while (S!=[]): |
|  | #Remove current element from stack |
|  | u = S.pop(0) |
|  | #Go to connected vertices |
|  | for t in G[u]: |
|  | #If the vertex hasn't been visited |
|  | if(visited[t]==False): |
|  | #Mark as visited |
|  | visited[t] = True |
|  | #Add to path |
|  | prev[t] = u |
|  | #Add to Stack |
|  | S.insert(0,t) |
|  | #Return path |
|  | return prev |
|  |  |
|  | #Method for Recursive Depth First Search inspired by the given graph search pseudocode |
|  | def RecDFS(G, source): |
|  | #Mark element as visited |
|  | visitedd[source] = True |
|  | #Go to connected vertices |
|  | for t in G[source]: |
|  | #Mark as visited |
|  | if visitedd[t] == False: |
|  | #Add to path |
|  | prevv[t] = source |
|  | #Go to next element |
|  | RecDFS(G, t) |
|  |  |
|  | #Testing method that prints a path inspired by the given graph search pseudocode |
|  | def printPath(prev, v): |
|  | if (prev[v] !=-1): |
|  | printPath(prev, prev[v]) |
|  | print(" - ") |
|  | print(v) |
|  |  |
|  | #Method to draw the graph representation of the maze, edited version of the draw\_graph code from Dr. Fuentes |
|  | def draw\_graph(G,maze\_rows,maze\_cols): |
|  | fig, ax = plt.subplots(figsize=(maze\_rows+(maze\_rows//4), maze\_cols+(maze\_cols//4))) |
|  | n = len(G) |
|  | coords =[] |
|  | for i in range(n): |
|  | #Calculate circle location using the calculations in draw\_maze |
|  | x0 = (i%maze\_cols) |
|  | y0 = (i//maze\_cols) |
|  | coords.append([x0,y0]) |
|  | for i in range(n): |
|  | for dest in G[i]: |
|  | ax.plot([coords[i][0],coords[dest][0]],[coords[i][1],coords[dest][1]], |
|  | linewidth=1,color='k') |
|  | for i in range(n): |
|  | ax.text(coords[i][0],coords[i][1],str(i), size=10,ha="center", va="center", |
|  | bbox=dict(facecolor='w',boxstyle="circle")) |
|  | ax.set\_aspect(1.0) |
|  | ax.axis('off') |
|  |  |
|  |  |
|  | #Method to draw the solved bfs maze, edited version of the draw\_maze code from Dr. Fuentes |
|  | def draw\_maze\_bfs (walls,maze\_rows,maze\_cols,G,cell\_nums=False): |
|  | fig, ax = plt.subplots() |
|  | plt.text(maze\_cols//2, -maze\_cols//16, "BFS", fontdict=None, withdash=False) |
|  | 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') |
|  | 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") |
|  | #Get BFS path from 0 to the top-right corner |
|  | iBFST = time.time() |
|  | prev = BFS(G, (maze\_rows\*maze\_cols)-1) |
|  | fBFST = time.time() |
|  | print("Time it took to get BFS path:", fBFST-iBFST) |
|  |  |
|  | #Starting vertex |
|  | v = 0 |
|  | #Stop when it reaches the end |
|  | while prev[v] !=-1: |
|  | #If the next element in the path is to the left |
|  | if v == prev[v]+1: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x-1 |
|  | y2 = y |
|  | #If the next element in the path is to the right |
|  | elif v == prev[v]-1: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x+1 |
|  | y2 = y |
|  | #If the next element in the path is down |
|  | elif v == prev[v]+maze\_cols: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x |
|  | y2 = y-1 |
|  | #If the next element in the path is up |
|  | else: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x |
|  | y2 = y+1 |
|  | ax.plot([x,x2],[y,y2],linewidth=1,color='r') |
|  | #Move to next element |
|  | v = prev[v] |
|  | ax.axis('off') |
|  | ax.set\_aspect(1.0) |
|  |  |
|  | def draw\_maze\_dfs (walls,maze\_rows,maze\_cols,G,cell\_nums=False): |
|  | fig, ax = plt.subplots() |
|  | plt.text(maze\_cols//2, -maze\_cols//16, "DFS", fontdict=None, withdash=False) |
|  | 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') |
|  | 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") |
|  |  |
|  | #Get DFS path from 0 to the top-right corner |
|  | iDFST = time.time() |
|  | prev = DFS(G, (maze\_rows\*maze\_cols)-1) |
|  | fDFST = time.time() |
|  | print("Time it took to get DFS path:", fDFST-iDFST) |
|  |  |
|  | #Starting vertex |
|  | v = 0 |
|  | #Stop when it reaches the end |
|  | while prev[v] !=-1: |
|  | #If the next element in the path is to the left |
|  | if v == prev[v]+1: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x-1 |
|  | y2 = y |
|  | #If the next element in the path is to the right |
|  | elif v == prev[v]-1: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x+1 |
|  | y2 = y |
|  | #If the next element in the path is down |
|  | elif v == prev[v]+maze\_cols: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x |
|  | y2 = y-1 |
|  | #If the next element in the path is up |
|  | else: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x |
|  | y2 = y+1 |
|  | ax.plot([x,x2],[y,y2],linewidth=1,color='r') |
|  | #Move to next element |
|  | v = prev[v] |
|  | ax.axis('off') |
|  | ax.set\_aspect(1.0) |
|  |  |
|  | def draw\_maze\_recdfs (walls,maze\_rows,maze\_cols,G,cell\_nums=False): |
|  | fig, ax = plt.subplots() |
|  | plt.text(maze\_cols//2, -maze\_cols//16, "Recursive DFS", fontdict=None, withdash=False) |
|  | 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') |
|  | 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") |
|  |  |
|  | #Create global variables for the recursive DFS |
|  | global visitedd |
|  | global prevv |
|  | visitedd = [False]\*len(G) |
|  | prevv = np.zeros(len(G),dtype=int)-1 |
|  |  |
|  | #Get recursive DFS path from 0 to the top-right corner |
|  | iRDFST = time.time() |
|  | RecDFS(G, (maze\_rows\*maze\_cols)-1) |
|  | fRDFST = time.time() |
|  | print("Time it took to get recursive DFS path:", fRDFST-iRDFST) |
|  |  |
|  | #Starting vertex |
|  | v = 0 |
|  | #Stop when it reaches the end |
|  | while prevv[v] !=-1: |
|  | #If the next element in the path is to the left |
|  | if v == prevv[v]+1: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x-1 |
|  | y2 = y |
|  | #If the next element in the path is to the right |
|  | elif v == prevv[v]-1: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x+1 |
|  | y2 = y |
|  | #If the next element in the path is down |
|  | elif v == prevv[v]+maze\_cols: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x |
|  | y2 = y-1 |
|  | #If the next element in the path is up |
|  | else: |
|  | x = (v%maze\_cols)+.5 |
|  | y = (v//maze\_cols)+.5 |
|  | x2 = x |
|  | y2 = y+1 |
|  | ax.plot([x,x2],[y,y2],linewidth=1,color='r') |
|  | #Move to next element |
|  | v = prevv[v] |
|  | ax.axis('off') |
|  | ax.set\_aspect(1.0) |
|  |  |
|  |  |
|  |  |
|  | plt.close("all") |
|  | maze\_rows = 35 |
|  | maze\_cols = 35 |
|  | numCells = maze\_rows\*maze\_cols |
|  |  |
|  | #wall list & dsf for standard method |
|  | walls = wall\_list(maze\_rows,maze\_cols) |
|  | S = DisjointSetForest(maze\_rows\*maze\_cols) |
|  |  |
|  | #wall list & dsf for compressed method |
|  | wallsC = wall\_list(maze\_rows,maze\_cols) |
|  | SC = DisjointSetForest(maze\_rows\*maze\_cols) |
|  |  |
|  | #draw initial maze |
|  | draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True) |
|  | numWalls = int(input('Enter number of walls\n')) |
|  |  |
|  | #Display message |
|  | if numWalls<(numCells-1): |
|  | print("A path from source to destination is not guaranteed to exist\n") |
|  | elif numWalls==(numCells-1): |
|  | print("There is a unique path from source to destination\n") |
|  | else: |
|  | print("There is at least one path from source to destination\n") |
|  |  |
|  | print("######## Maze using standard find and union ########\n") |
|  |  |
|  | iStandardMazeT = time.time() |
|  | G = create\_standard\_dsf\_maze(S,walls,numWalls,numCells) |
|  | fStandardMazeT = time.time() |
|  |  |
|  | #Draw graph representation |
|  | draw\_graph(G,maze\_rows,maze\_cols) |
|  | #Draw resulting maze |
|  | draw\_maze(walls,maze\_rows,maze\_cols) |
|  | #Draw all three solved mazes |
|  | draw\_maze\_bfs(walls,maze\_rows,maze\_cols,G) |
|  | draw\_maze\_dfs (walls,maze\_rows,maze\_cols,G) |
|  | draw\_maze\_recdfs (walls,maze\_rows,maze\_cols,G) |
|  |  |
|  | print("Time it took to create the maze:", fStandardMazeT-iStandardMazeT) |
|  | print("Maze row size:", maze\_rows) |
|  | print("Maze column size:", maze\_cols) |
|  |  |
|  | print("\n######## Maze using compressed find and union by size ########\n") |
|  |  |
|  | iCompressedMazeT = time.time() |
|  | Gc = create\_compressed\_dsf\_maze(SC,wallsC,numWalls,numCells) |
|  | fCompressedMazeT = time.time() |
|  |  |
|  | #Draw graph representation |
|  | draw\_graph(Gc,maze\_rows,maze\_cols) |
|  | #Draw resulting maze |
|  | draw\_maze(wallsC,maze\_rows,maze\_cols) |
|  | #Draw all three solved mazes |
|  | draw\_maze\_bfs(wallsC,maze\_rows,maze\_cols,Gc) |
|  | draw\_maze\_dfs (wallsC,maze\_rows,maze\_cols,Gc) |
|  | draw\_maze\_recdfs (wallsC,maze\_rows,maze\_cols,Gc) |
|  |  |
|  | print("Time it took to create the maze:", fCompressedMazeT-iCompressedMazeT) |
|  | print("Maze row size:", maze\_rows) |
|  | print("Maze column size:", maze\_cols) |