Laurence Labayen

Dec 4, 2019

Lab #7:

Algorithm Design Techniques

CS2302 Data Structures - MW 10:30am

Professor: Dr. Olac Fuentes T.A.: Anindita Nath

Fall 2019

Description:

In this last lab assignment for data structures, we were tasked to demonstrate our ability to implement 3 algorithm design techniques. This includes randomization, backtracking, and dynamic programming. Using these techniques, we were able to solve NP-complete problems such as detecting Hamiltonian cycles and edit distance problems. The first part of this lab was to use randomization to determine if there is a Hamiltonian cycle in an undirected graph by using the given pseudocode on the lab instructions:

> *"Randomized Hamiltonian(V,E)*
> *for i in range(maximum trials)*
>
> *let Eh be a random subset of E of size V*
> *if graph (V,Eh) has 1 connected component and the in-degree of every vertex in V is 2*
>
> *return Eh # Eh forms a Hamiltonian cycle*
> *return None # No Hamiltonian cycle was found"*

In the second part of the lab were also tasked to determine if there is a Hamiltonian cycle in a graph but this time using backtracking technique. Additionally, our implementation had to be as efficient as possible using recursive calls. Lastly, an edit distance problem with constraints was to be implemented and solved using dynamic programming.

> *"Modify the edit distance function provided in class to allow replacements only in the case where the characters being interchanged are both vowels, or both consonants. For example, 'a' can be replaced by 'e' but not by 's', and 't' can be replaced by 'w' but not by 'u'. "*

Solution design and Implementation:

Randomization:

Given two parameters, V for an input graph and tests for the number of desired tries for the randomization. First, we turn V to an edge list to be traversed easier in our loop of tests. Inside the loop with the given range, we randomly pick edges selected from our edge list. Next, create an adjacency list graph and we insert each randomly selected edge into an adjacency list. By using an adjacency list, we can use the provided connected components function from the class website to determine if there is a Hamiltonian cycle. Lastly, we check if the in-degree of every vertex in our list is equal to 2.

```python
def connected_components(g):
    vertices = len(g.al)
    components = vertices
    s = DSF.DSF(vertices)
    for v in range(vertices):
        for edge in g.al[v]:
            components -= s.union(v,edge.dest)
    return components#, s

# Function to detect Hamiltonian cycle using randomization
# Inputs: V as graph, and tests as number of random tests desired
# Output: Boolean. True if Hamiltonian cycle is detected, False if otherwise
def ham_random(V, tests):

    # turn v into edge list to be picked from randomly
    edge_list=V.as_EL()
    al=AL.Graph(len(V.al),weighted=V.weighted, directed=V.directed)

    for t in range(tests):
        # add random edges seleceted from edge list into list
        edge=random.sample(edge_list.el, len(V.al))

        # use list of random edges and insert into adjacency list

        for i in range(len(edge)):
            al.insert_edge(edge[i].source,edge[i].dest)

        # check if there is only 1 connected componenet
        if connected_components(al) == 1:

            # check in degree of every vertex is 2
            for i in range(len(al.al)):
                if in_degree(al,i) != 2:
                    return False
            return True
```

Backtracking:

With this implementation, I used a main and a utility function. The main function takes parameter V (graph) and converts it to an edge list for ease of traversal, similar to our randomization implementation. We also, create an empty edge list to be used in our utility function. Then we call our utility function with previously mentioned graphs passed as arguments. I first started with a base case where we stop if the number of edges in our first edge list graph (V) is the same as the number of vertices. This satisfies the condition where our subset is the size of vertices. If this is true, we go into our conditions to check for a Hamiltonian Cycle. Here, we convert our edge list graph to an adjacency list to be used to check for connected components and to check for in-degrees. Our second base case is if the list of edges is empty, return None. Then we go into our recursive calls, where we take the first edge and add it in our list of edges and assign to our graph. Our first recursive call is taking the graph and the first edge of our list. If that does not return None, return the value. The second recursive call is with the first edge removed from our argument.

```python
# Backtracking helper function
# Inputs: edge_list as list of edges and graph as input graph
# Output: returns None is Hamiltonian cycle is not detected
# and True if there is a cycle
def ham_backtrack_(V,Eh):
    # Base Case
    if len(V.el) == V.vertices:
        graphAL = V.as_AL()
        # check if there is only 1 connected componenet
        if connected_components(graphAL) == 1:
            # check in degree of every vertex is 2
            for i in range(len(graphAL.al)):
                if in_degree(graphAL, i) != 2:
                    return None
            return graphAL
    # check if list of edges is empty
    if len(Eh) == 0:
        return
    else:
        # Recursive calls
        V.el = V.el + [Eh[0]] # Take first edge
        a = ham_backtrack_(V,Eh[1:])
        if a is not None:
            return a
        V.el.remove(Eh[0]) # Do not take first edge
        return ham_backtrack_(V,Eh[1:])

# Backtracking main function.
# Input: V as input graph
def ham_backtrack(V):
    # Convert V as an edge list and assign it to Eh
    Eh = V.as_EL()

    # Create an edge list graph with the same parameters as V
    el = EL.Graph(len(V.al), weighted=V.weighted, directed=V.directed)

    return ham_backtrack_(el,Eh.el)
```

Dynamic Programming:

Our last task is to implement dynamic programming to solve the edit distance problem. Additionally, constraints are added to where replacements are only allowed for the characters being changed are both vowels or both consonants. To do this the edit distance provided in the class website had to be modified as per lab instructions. I first created a list of vowels (a,e,i,o,u). Second, was to modify the code to only make replacements if current characters are both vowels or both consonants. If the check is valid, then it will find the minimum of the three values (insert, replace, remove). Otherwise, replacement will not be considered in our minimum value.

```python
# From class website, with modifications to only allow replacements with both
# vowels or both consonants
# Inputs: s1, s2 as strings
# Output: Minimum number of operations to convert s1 to s2
def edit_distance_modified(s1,s2):
    v = ['a','e','i','o','u']

    d = np.zeros((len(s1)+1,len(s2)+1),dtype=int)
    d[0,:] = np.arange(len(s2)+1)
    d[:,0] = np.arange(len(s1)+1)
    for i in range(1,len(s1)+1):
        for j in range(1,len(s2)+1):
            if s1[i-1] ==s2[j-1]:
                d[i,j] =d[i-1,j-1]
            else:
                # allow replacements only in the case where the characters
                # being interchanged are both vowels, or both consonants
                if (s1[i-1] in v and s2[j-1] in v) or (s1[i-1]
                not in v and s2[j-1] not in v):
                    n = [d[i,j-1],d[i-1,j-1],d[i-1,j]]
                    d[i,j] = min(n)+1
                else:
                    n = [d[i,j-1],d[i-1,j]]
                    d[i,j] = min(n)+1
    return d[-1,-1]
```

Experimental Results:

Menu:

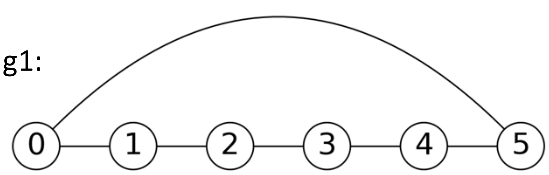
```
1. Test known graphs for Hamiltonian cycle
2. Test custom graph with random edges for Hamiltonian cycle
3. Test modified and unmodified edit distance function with input strings
4. Test for running times using edit distance with random pairs of words
```

1. Known graphs with or without Hamiltonian Cycle

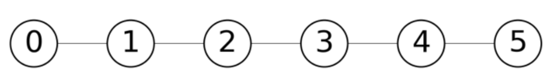
```python
    print('1. Test known graphs for Hamiltonian cycle')
    print('2. Test custom graph with random edges for Hamiltonian cycle')
    print('3. Test modified and unmodified edit distance function with input strings')
    print('4. Test for running times using edit distance with random pairs of words')

    choice=int(input('Select choice: '))

    if choice==1:
        g1 = AL.Graph(6) # Graph Hamiltonian cycle
        g1.insert_edge(0,1)
        g1.insert_edge(1,2)
        g1.insert_edge(2,3)
        g1.insert_edge(3,4)
        g1.insert_edge(4,5)
        g1.insert_edge(5,0)
        g1.draw()

        g2 = AL.Graph(6) # Graph without Hamiltonian cycle
        g2.insert_edge(0,1)
        g2.insert_edge(1,2)
        g2.insert_edge(2,3)
        g2.insert_edge(3,4)
        g2.insert_edge(4,5)
        g2.draw()

        print('Randomization:',ham_random_test(g1,1000)) #Ham cycle
        print('Backtracking:',ham_backtrack_test(g1)) #Ham cycle

        print('Randomization:',ham_random_test(g2,1000)) #Not Ham cycle
        print('Backtracking:',ham_backtrack_test(g2)) #Not Ham cycle
```

```
Select choice: 1
g1:
Randomization: Hamiltonian Cycle
Backtracking: Hamiltonian Cycle

g2:
Randomization: Not a Hamiltonian Cycle
Backtracking: Not a Hamiltonian Cycle
```

g1:

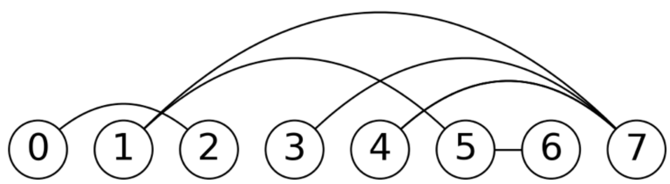


g2:

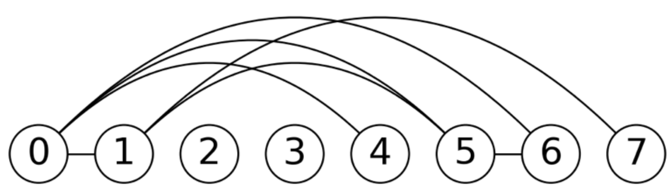


2. Test custom size graph with random edges for Hamiltonian cycle

```
Select choice: 2

Enter graph size: 8

Enter number of random edges: 9

Randomization: Not a Hamiltonian Cycle
Backtracking: Not a Hamiltonian Cycle
```




```
Select choice: 2

Enter graph size: 8

Enter number of random edges: 8

Randomization: Not a Hamiltonian Cycle
Backtracking: Not a Hamiltonian Cycle
```



3. Test modified and unmodified edit distance function with input strings

```
Select choice: 3

Enter word #1: blots

Enter word #2: boots

unmodified edit distance: 1
modified edit distance: 2
```

```
Select choice: 3

Enter word #1: quality

Enter word #2: quantity

unmodified edit distance: 2
modified edit distance: 2
```

```
Select choice: 3

Enter word #1: aaaa

unmodified edit distance: 4
modified edit distance: 8

Enter word #2: bbbb
```

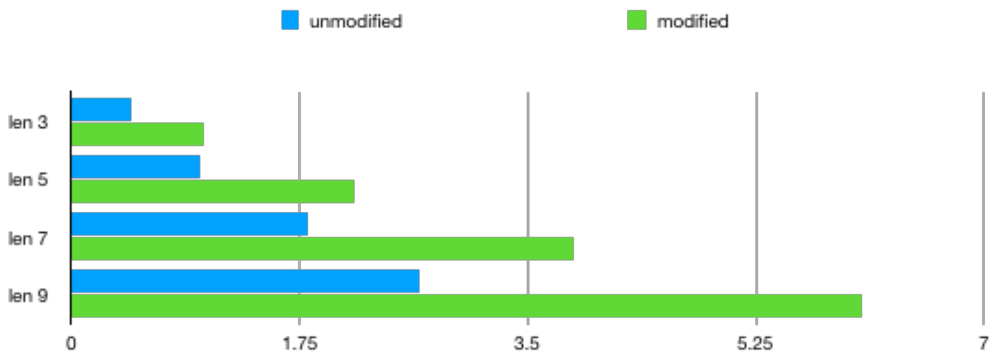
```
Select choice: 3

Enter word #1: boots

Enter word #2: boats

unmodified edit distance: 1
modified edit distance: 1
```

```
Select choice: 3

Enter word #1: endlessness

Enter word #2: extensiveness

unmodified edit distance: 7
modified edit distance: 7
```

4. Test for running times using edit distance with random pairs of words

Test data done with 10000 comparisons of various word lengths, time shown in seconds.

	len 3	len 5	len 7	len 9
unmodified	0.4605	0.9917	1.8174	2.6652
modified	1.0185	2.1676	3.8591	6.0621




```
Select choice: 4

Enter number of edit distance comparisons: 10000

Enter desired length of word (3, 5, 7, or 9): 3

unmodified edit distance running time with 10000 comparisons: 0.4605
modified edit distance running time with 10000 comparisons: 1.0185 seconds
```

```
Select choice: 4

Enter number of edit distance comparisons: 10000

Enter desired length of word (3, 5, 7, or 9): 5

unmodified edit distance running time with 10000 comparisons: 0.9917
modified edit distance running time with 10000 comparisons: 2.1676 seconds
```

```
Select choice: 4

Enter number of edit distance comparisons: 10000

Enter desired length of word (3, 5, 7, or 9): 7

unmodified edit distance running time with 10000 comparisons: 1.8174
modified edit distance running time with 10000 comparisons: 3.8591 seconds
```

```
Select choice: 4

Enter number of edit distance comparisons: 10000

Enter desired length of word (3, 5, 7, or 9): 9

unmodified edit distance running time with 10000 comparisons: 2.6652
modified edit distance running time with 10000 comparisons: 6.0621 seconds
```

Conclusion:

This lab has taught us to use different algorithm techniques to be used for problem solving implementations especially with NP-complete such as Hamiltonian cycles and edit distance problems that are not solvable in realistic time. Running the program with different inputs has shown me that the randomization approach is less precise than backtracking but faster. On the other hand, backtracking takes longer for larger graphs but you get a definite output. The approach on this was slightly easier than other labs as we were given pseudocode and a base code to modify.

Appendix:

(modifications in word_test in lab7.py function Dec 7, 2019)

Line 209:
edit_distance(words[random.randint(0,len(words)-1)],words[random.randint(0,len(words)-1)])

Line 218:
edit_distance_modified(words[random.randint(0,len(words)-1)],words[random.randint(0,len(words)-1)])

LAB7.py:

```
import DSF
import graph_AL as AL
import graph_EL as EL
import random
import numpy as np
import time

# Function to get number of in-degrees of a given vertex
# Inputs: G as graph, v as the vertex
# Output: Number of in-degrees of vertex v
def in_degree(G, v):
    indeg = 0
    for i in range(len(G.al)):
        for j in G.al[i]:
            if j.dest == v:
```

```
                                        indeg += 1
            return indeg

# From class website
def connected_components(g):
    vertices = len(g.al)
    components = vertices
    s = DSF.DSF(vertices)
    for v in range(vertices):
        for edge in g.al[v]:
            components -= s.union(v,edge.dest)
    return components#, s

# Function to detect Hamiltonian cycle using randomization
# Inputs: V as graph, and tests as number of random tests desired
# Output: Boolean. True if Hamiltonian cycle is detected, False if otherwise
def ham_random(V, tests):

    # turn v into edge list to be picked from randomly
    edge_list=V.as_EL()


    for t in range(tests):
        # add random edges seleceted from edge list into list
        edge=random.sample(edge_list.el, len(V.al))

        # use list of random edges and insert into adjacency list
        al=AL.Graph(len(V.al),weighted=V.weighted, directed=V.directed)
        for i in range(len(edge)):
            al.insert_edge(edge[i].source,edge[i].dest)

        # check if there is only 1 connected componenet
        if connected_components(al) == 1:

            # check in degree of every vertex is 2
            for i in range(len(al.al)):
                if in_degree(al,i) != 2:
                    return False
            return al

# Randomized Hamiltonian cycle tester
# Inputs: V as an adjacency list graph, test as range of tests desired
# Output: Determines if V is a Hamiltonian cycle graph
def ham_random_test(V,tests):

    for i in range(100):
        ham=ham_random(V, tests)
        if isinstance(ham, AL.Graph):
            ham.draw()
            return "Hamiltonian Cycle"
    return "Not a Hamiltonian Cycle"

# Backtracking helper function
# Inputs: edge_list as list of edges and graph as input graph
```

```
# Output: returns None is Hamiltonian cycle is not detected
# and True if there is a cycle
def ham_backtrack_(V,Eh):
        # Base Case
        if len(V.el) == V.vertices:
                graphAL = V.as_AL()
     # check if there is only 1 connected componenet
                if connected_components(graphAL) == 1:
                        # check in degree of every vertex is 2
                        for i in range(len(graphAL.al)):
                                if in_degree(graphAL, i) != 2:
                                        return None
                        return graphAL
   # check if list of edges is empty
        if len(Eh) == 0:
                return
        else:
     # Recursive calls
                V.el = V.el + [Eh[0]] # Take first edge
                a = ham_backtrack_(V,Eh[1:])
                if a is not None:
                        return a
                V.el.remove(Eh[0]) # Do not take first edge
                return ham_backtrack_(V,Eh[1:])

# Backtracking main function.
# Input: V as input graph
def ham_backtrack(V):
   # Convert V as an edge list and assign it to Eh
   Eh = V.as_EL()

   # Create an edge list graph with the same parameters as V
   el = EL.Graph(len(V.al), weighted=V.weighted, directed=V.directed)

   return ham_backtrack_(el,Eh.el)

# Simplified Backtracking Hamiltonian cycle test
# Input: V as graph
# Output: Determines if graph is creates a Hamiltonian cycle or not
def ham_backtrack_test(V):
   ham=ham_backtrack(V)
   if isinstance(ham, AL.Graph):
      ham.draw()
      return "Hamiltonian Cycle"
   else:
      return "Not a Hamiltonian Cycle"

# From class website
# Inputs: s1, s2 as strings
# Output: Minimum number of operations to convert s1 to s2
def edit_distance(s1,s2):
   d = np.zeros((len(s1)+1,len(s2)+1),dtype=int)
   d[0,:] = np.arange(len(s2)+1)
   d[:,0] = np.arange(len(s1)+1)
```

```
    for i in range(1,len(s1)+1):
        for j in range(1,len(s2)+1):
            if s1[i-1] ==s2[j-1]:
                d[i,j] =d[i-1,j-1]
            else:
                n = [d[i,j-1],d[i-1,j-1],d[i-1,j]]
                d[i,j] = min(n)+1
    return d[-1,-1]

# From class website, with modifications to only allow replacements with both
# vowels or both consonants
# Inputs: s1, s2 as strings
# Output: Minimum number of operations to convert s1 to s2
def edit_distance_modified(s1,s2):
    v = ['a','e','i','o','u']

    d = np.zeros((len(s1)+1,len(s2)+1),dtype=int)
    d[0,:] = np.arange(len(s2)+1)
    d[:,0] = np.arange(len(s1)+1)
    for i in range(1,len(s1)+1):
        for j in range(1,len(s2)+1):
            if s1[i-1] ==s2[j-1]:
                d[i,j] =d[i-1,j-1]
            else:
                # allow replacements only in the case where the characters
                # being interchanged are both vowels, or both consonants
                if (s1[i-1] in v and s2[j-1] in v) or (s1[i-1]
                not in v and s2[j-1] not in v):
                    n = [d[i,j-1],d[i-1,j-1],d[i-1,j]]
                    d[i,j] = min(n)+1
                else:
                    n = [d[i,j-1],d[i-1,j]]
                    d[i,j] = min(n)+1
    return d[-1,-1]

# Function to test words from words_alpha.txt file used in lab #1
# Inputs: size as the size of word pairs to be tested, choice
# as the desired word length (3,5,7,9)
# Output: Prints running time of unmodified and modified edit distance
# with desired size of pairs
def word_test(size,choice):
    # Read word file with words
    wordSet = list(open("words_alpha.txt").read().splitlines())

    len3,len5,len7,len9=[],[],[],[]

    # Insert words in different lists depending on length
    for i in wordSet:
        if len(i)==3:
            len3.append(i)
        if len(i)==5:
            len5.append(i)
        if len(i)==7:
            len7.append(i)
```

```
        if len(i)==9:
            len9.append(i)

    if choice==3:
        words=len3
    if choice==5:
        words=len5
    if choice==7:
        words=len7
    if choice==9:
        words=len9


    timer=0

    start = time.perf_counter()
    for j in range(size):
        edit_distance(words[random.randint(0,len(words)-1)],words[random.randint(0,len(words)-1)])

    end = time.perf_counter()
    timer += end - start
    print('unmodified edit distance running time with', size, 'comparisons:', str(round(timer,4)))

    start = time.perf_counter()

    for k in range(size):
        edit_distance_modified(words[random.randint(0,len(words)-1)],words[random.randint(0,len(words)-
1)])

    end = time.perf_counter()
    timer += end - start
    print('modified edit distance running time with', size,
          'comparisons:', str(round(timer,4)), 'seconds')

# Function to create custom graph with variable size and number of random edges.
# Inputs: size as number of vertices in the graph and num_edges and desired number of edges
# Output: graph with desired number of vertices and random edges
def custom_graph(size, num_edges):
    g=AL.Graph(size)

    for i in range(num_edges):
        g.insert_edge(random.randint(0,size-1),random.randint(0,size-1))

    return g

if __name__=="__main__":

    print('1. Test known graphs for Hamiltonian cycle')
    print('2. Test custom graph with random edges for Hamiltonian cycle')
    print('3. Test modified and unmodified edit distance function with input strings')
    print('4. Test for running times using edit distance with random pairs of words')

    choice=int(input('Select choice: '))
```

```
    if choice==1:
        g1 = AL.Graph(6) # Graph Hamiltonian cycle
        g1.insert_edge(0,1)
        g1.insert_edge(1,2)
        g1.insert_edge(2,3)
        g1.insert_edge(3,4)
        g1.insert_edge(4,5)
        g1.insert_edge(5,0)
        g1.draw()

        g2 = AL.Graph(6) # Graph without Hamiltonian cycle
        g2.insert_edge(0,1)
        g2.insert_edge(1,2)
        g2.insert_edge(2,3)
        g2.insert_edge(3,4)
        g2.insert_edge(4,5)
        g2.draw()

        print('g1:')
        print('Randomization:',ham_random_test(g1,1000)) #Ham cycle
        print('Backtracking:',ham_backtrack_test(g1)) #Ham cycle
        print('\ng2:')
        print('Randomization:',ham_random_test(g2,1000)) #Not Ham cycle
        print('Backtracking:',ham_backtrack_test(g2)) #Not Ham cycle

    if choice==2:
        graph_size=int(input('Enter graph size: '))
        edge_count=int(input('Enter number of random edges: '))

        g3=custom_graph(graph_size, edge_count)
        g3.draw()
        print('\nRandomization:',ham_random_test(g3,1000))
        print('Backtracking:',ham_backtrack_test(g3))

    if choice==3:
        word1=str(input('Enter word #1: '))
        word2=str(input('Enter word #2: '))

        print('\nunmodified edit distance:',edit_distance(word1,word2))
        print('modified edit distance:',edit_distance_modified(word1,word2))

    if choice==4:
        num_pairs=int(input('Enter number of edit distance comparisons: '))
        length=int(input('Enter desired length of word (3, 5, 7, or 9): '))
        print()
        word_test(num_pairs,length)
```

DSF.py:

```
# Implementation of disjoint set forest (or union/find data structure)
# Programmed by Olac Fuentes
# Last modified November 13, 2019
```

```
from scipy.interpolate import interp1d
import numpy as np
import matplotlib.pyplot as plt

class DSF:
    # Constructor
    def __init__(self, sets):
        # Creates forest with 'sets' root nodes
        self.parent = np.zeros(sets,dtype=int)-1

    def find(self,i):
        # Returns root of tree that i belongs to
        if self.parent[i]<0:
            return i
        return self.find(self.parent[i])

    def union(self,i,j):
        # Makes root of j's tree point to root of i's tree if they are different
        # Return 1 if a parent reference was changed, 0 otherwise
        root_i = self.find(i)
        root_j = self.find(j)
        if root_i != root_j:
            self.parent[root_j] = root_i
            return 1
        return 0

    def draw(self):
        scale = 30
        fig, ax = plt.subplots()
        for i in range(len(self.parent)):
            if self.parent[i]<0:
                ax.plot([i*scale,i*scale],[0,scale],linewidth=1,color='k')
                ax.plot([i*scale-1,i*scale,i*scale+1],[scale-2,scale,scale-2],linewidth=1,color='k')
            else:
                x = np.linspace(i*scale,self.parent[i]*scale)
                x0 = np.linspace(i*scale,self.parent[i]*scale,num=5)
                diff = np.abs(self.parent[i]-i)
                if diff == 1:
                    y0 = [0,0,0,0,0]
                else:
                    y0 = [0,-6*diff,-8*diff,-6*diff,0]
                f = interp1d(x0, y0, kind='cubic')
                y = f(x)
                ax.plot(x,y,linewidth=1,color='k')
                ax.plot([x0[2]+2*np.sign(i-self.parent[i]),x0[2],x0[2]+2*np.sign(i-self.parent[i])],[y0[2]-
1,y0[2],y0[2]+1],linewidth=1,color='k')
            ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
             bbox=dict(facecolor='w',boxstyle="circle"))
        ax.axis('off')
        ax.set_aspect(1.0)

if __name__ == "__main__":
    plt.close("all")
```

```
    s = DSF(6)
    print(s.parent)
    s.draw()

    s.union(0,1)
    print(s.parent)
    s.draw()

    s.union(4,2)
    print(s.parent)
    s.draw()

    s.union(3,5)
    print(s.parent)
    s.draw()

    s.union(1,5)
    print(s.parent)
    s.draw()
```

Graph_AL.py

```
# Adjacency list representation of graphs
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
#import graph_AM as AM
import graph_EL as EL
import sys

sys.setrecursionlimit(150000)

class Edge:
    def __init__(self, dest, weight=1):
        self.dest = dest
        self.weight = weight

class Graph:
    # Constructor
    def __init__(self, vertices, weighted=False, directed = False):
        self.al = [[] for i in range(vertices)]
        self.weighted = weighted
        self.directed = directed

    # Insert Edge function from class website
    def insert_edge(self,source,dest,weight=1):
        if source >= len(self.al) or dest>=len(self.al) or source <0 or dest<0:
            print('Error, vertex number out of range')
```

```
    if weight!=1 and not self.weighted:
        print('Error, inserting weighted edge to unweighted graph')
    else:
        self.al[source].append(Edge(dest,weight))
        if not self.directed:
            self.al[dest].append(Edge(source,weight))
# Delete Edge helper function from class website
def delete_edge_(self,source,dest):
    i = 0
    for edge in self.al[source]:
        if edge.dest == dest:
            self.al[source].pop(i)
            return True
        i+=1
    return False
# Insert Edge function from class website
def delete_edge(self,source,dest):
    if source >= len(self.al) or dest>=len(self.al) or source <0 or dest<0:
        print('Error, vertex number out of range')
    else:
        deleted = self.delete_edge_(source,dest)
        if not self.directed:
            deleted = self.delete_edge_(dest,source)
    if not deleted:
        print('Error, edge to delete not found')
# Display function from class website
def display(self):
    print('[',end='')
    for i in range(len(self.al)):
        print('[',end='')
        for edge in self.al[i]:
            print('('+str(edge.dest)+','+str(edge.weight)+')',end='')
        print(']',end=' ')
    print(']')
# Draw function from class website
def draw(self):

    scale = 30
    fig, ax = plt.subplots()
    for i in range(len(self.al)):
        for edge in self.al[i]:
            d,w = edge.dest, edge.weight
            if self.directed or d>i:
                x = np.linspace(i*scale,d*scale)
                x0 = np.linspace(i*scale,d*scale,num=5)
                diff = np.abs(d-i)
                if diff == 1:
                    y0 = [0,0,0,0,0]
                else:
                    y0 = [0,-6*diff,-8*diff,-6*diff,0]
                f = interp1d(x0, y0, kind='cubic')
                y = f(x)
                s = np.sign(i-d)
                ax.plot(x,s*y,linewidth=1,color='k')
```

```
                if self.directed:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.plot(xd,yd,linewidth=1,color='k')
                if self.weighted:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.text(xd[2]-s*2,yd[2]+3*s, str(w), size=12,ha="center", va="center")
            ax.plot([i*scale,i*scale],[0,0],linewidth=1,color='k')
            ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
             bbox=dict(facecolor='w',boxstyle="circle"))
        ax.axis('off')
        ax.set_aspect(1.0)

    # as_EL converts current adjacency list graph to an edge list
    def as_EL(self):

        # create an empty graph with the same length as current graph
        edgelist = EL.Graph(len(self.al),self.weighted, self.directed)

        # insert edges using a nested loop
        for i in range(len(self.al)):
            for j in self.al[i]:
                edgelist.insert_edge(i, j.dest, j.weight)
        return edgelist
    # as_AM converts current adjacency list graph to an adjacency matrix
    def as_AM(self):

        # create an empty graph with the same length as current graph
        matrix =  AM.Graph(len(self.al), self.weighted, self.directed)

        # insert edges using a nested loop
        for i in range(len(self.al)):
            for j in self.al[i]:
                matrix.insert_edge(i, j.dest, j.weight)

        return matrix

    def as_AL(self):
        return self

    # Breadth first search function used to return path
    def BFS(self, s,end):

        # Mark all the vertices as not visited
        visited = [False] * (len(self.al))

        # Create a queue for BFS
        queue = [[s]]

        while queue:
```

```
        # pop element from queue and assign it to s
        s = queue.pop(0)

        # if end is found, return
        if s[-1]==end:
            print('From AL BFS')
            return s


        # Get all adjacent vertices of the
        # popped vertex s. If a adjacent
        # has not been visited, then mark it
        # visited and append it
        for i in self.al[s[-1]]:
            if visited[i.dest] == False:
                queue.append(s + [i.dest])
                visited[i.dest] = True

# Depth first search function used to return path
def DFS(self, s, end):
    # start an empty list of visited elements
    visited=[]

    # call DFS helper function
    print('From AL DFS')
    return self.DFS_(visited, s, end)

# Depth first search helper function used to return path
def DFS_(self, visited, s, end):

    # check if s is in the visited list
    if s not in visited:
        # check if visited is not empty and if the last element of
        # visited is the end element
        if len(visited) > 0 and visited[-1]==end:
            return
        # append s to visited list
        visited.append(s)

        # call function recursively with the starting element as the
        # destination of the neighbours of s
        for neighbour in self.al[s]:
            self.DFS_(visited, neighbour.dest, end)

    return visited

# Function to print the path in the correct format as shown in the lab
# instructions [b0,b1,b2,b3]
def path_steps(self, func):
    if func == 'DFS':
        search_path = self.DFS(0,len(self.al)-1)
    if func == 'BFS':
        search_path = self.BFS(0,len(self.al)-1)
```

```
    for i in search_path:
        print (i, [int(x) for x in list('{0:04b}'.format(i))])

# Modified draw function from class website used to highlight path
#found from BFS or DFS
def draw_path(self, func):
    scale = 30
    fig, ax = plt.subplots()

    if func == 'DFS':
        search_path = self.DFS(0,len(self.al)-1)
    if func == 'BFS':
        search_path = self.BFS(0,len(self.al)-1)

    # create path list to be used to highlight path
    path = []
    for j in range(len(search_path)-1):
        path.append((search_path[j], search_path[j+1]))


    for i in range(len(self.al)):
        for edge in self.al[i]:

            # highlighted path
            if (i, edge.dest) in path or (edge.dest, i) in path:
                line_color = "#ff007f"

            else:
                line_color = "#eeefff"

            d,w = edge.dest, edge.weight
            if self.directed or d>i:
                x = np.linspace(i*scale,d*scale)
                x0 = np.linspace(i*scale,d*scale,num=5)
                diff = np.abs(d-i)
                if diff == 1:
                    y0 = [0,0,0,0,0]
                else:
                    y0 = [0,-6*diff,-8*diff,-6*diff,0]
                f = interp1d(x0, y0, kind='cubic')
                y = f(x)
                s = np.sign(i-d)
                ax.plot(x,s*y,linewidth=1,color=line_color)
                if self.directed:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.plot(xd,yd,linewidth=1,color=line_color)
                if self.weighted:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.text(xd[2]-s*2,yd[2]+3*s, str(w), size=12,ha="center", va="center")
        ax.plot([i*scale,i*scale],[0,0],linewidth=1,color='k')
```

```
        ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
         bbox=dict(facecolor='w',boxstyle="circle"))
    ax.axis('off')
    ax.set_aspect(1.0)
    plt.show(block = True)

  def in_degree_check(self):
    for edges in self.al:
        if len(edges) != 2:
            return False
    return True
```

Graph_EL.py

```
# Edge list representation of graphs
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
#import graph_AM as AM
import graph_AL as AL
import sys

sys.setrecursionlimit(150000)

class Edge:
    def __init__(self, source, dest, weight=1):
        self.source = source
        self.dest = dest
        self.weight = weight

class Graph:
    # Constructor
    def __init__(self, vertices, weighted=False, directed = False):
        self.el = []
        self.vertices = vertices
        self.weighted = weighted
        self.directed = directed
        self.representation = 'EL'

    # Insert edge function that adds an edge given its source, destination and weight
    def insert_edge(self,source,dest,weight=1):

        if weight!=1 and not self.weighted:
            print('Error, inserting weighted edge to unweighted graph')
        else:
            # insert edge to graph
            self.el.append(Edge(source,dest,weight))

    # Delete edge function that deletes an edge given its source and destination
    def delete_edge(self,source,dest):
```

```
        # find position of the given edge and remove from graph
        for i in self.el:
            if i.source==source and i.dest==dest:
                self.el.remove(i)

    # Displays all the edges in the graph
    def display(self):
        print('[',end='')
        for i in self.el:
            print('('+str(i.source)+','+str(i.dest)+','+str(i.weight)+')',end='')
        print(']',end=' ')
        print('')

#    # Draw function that converts the current graph to an adjacency list then draws
#    # the graph (as per lab instructions)
#    def draw(self):
#
#        # converts current graph to an adjacency list to be used for the function
#        adjlist = self.as_AL()
#
#        scale = 30
#        fig, ax = plt.subplots()
#        for i in range(len(adjlist.al)):
#            for edge in adjlist.al[i]:
#                d,w = edge.dest, edge.weight
#                if self.directed or d>i:
#                    x = np.linspace(i*scale,d*scale)
#                    x0 = np.linspace(i*scale,d*scale,num=5)
#                    diff = np.abs(d-i)
#                    if diff == 1:
#                        y0 = [0,0,0,0,0]
#                    else:
#                        y0 = [0,-6*diff,-8*diff,-6*diff,0]
#                    f = interp1d(x0, y0, kind='cubic')
#                    y = f(x)
#                    s = np.sign(i-d)
#                    ax.plot(x,s*y,linewidth=1,color='k')
#                    if self.directed:
#                        xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
#                        yd = [y0[2]-1,y0[2],y0[2]+1]
#                        yd = [y*s for y in yd]
#                        ax.plot(xd,yd,linewidth=1,color='k')
#                    if self.weighted:
#                        xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
#                        yd = [y0[2]-1,y0[2],y0[2]+1]
#                        yd = [y*s for y in yd]
#                        ax.text(xd[2]-s*2,yd[2]+3*s, str(w), size=12,ha="center", va="center")
#            ax.plot([i*scale,i*scale],[0,0],linewidth=1,color='k')
#            ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
#             bbox=dict(facecolor='w',boxstyle="circle"))
#        ax.axis('off')
#        ax.set_aspect(1.0)
```

```
def as_EL(self):
    return self

# as_AM converts current edge list graph to an adjacency matrix
def as_AM(self):

     # create an empty graph with the same length as current graph
    matrix =  AM.Graph(self.vertices, self.weighted, self.directed)

    # insert edges using a loop
    for i in self.el:
        matrix.insert_edge(i.source, i.dest, i.weight)
    return matrix

# as_AM converts current adjacency matrix graph to an adjacency list
def as_AL(self):

     # create an empty graph with the same length as current graph
    adjlist =  AL.Graph(self.vertices, self.weighted, self.directed)

    # insert edges using a loop
    for i in self.el:
        adjlist.insert_edge(i.source, i.dest, i.weight)
    return adjlist

# Breadth first search function used to return path
def BFS(self, s,end):

    # Mark all the vertices as not visited
    visited = [False] * (self.vertices)
    # Assign visited element at s to True
    visited[s]=True
    # Create a queue for BFS
    queue = [[s]]
    # Create a path list with s as the first element
    path=[s]

    while queue:

        # pop element from queue and assign it to s
        s = queue.pop(0)
        if s==end:
            return


        # Get all adjacent vertices of the
        # popped vertex s. If a adjacent
        # has not been visited, then mark it
        # visited and append it
        for i in self.el:
            if visited[i.dest] == False:
                queue.append(i.dest)
                visited[i.dest] = True
                path.append(i.dest)
```

```
        print('From EL BFS')
        return path

# Depth first search function used to return path
def DFS(self, s, end):

    # start an empty list of visited elements
    visited=[]

    # call DFS helper function
    print('From EL DFS')
    return self.DFS_(visited, s, end)

# Depth first search helper function used to return path
def DFS_(self, visited, s, end):

    # check if s is in the visited list
    if s not in visited:

        # check if visited is not empty and if the last element of
        # visited is the end element
        if len(visited) > 0 and visited[-1]==end:
            return

        # append s to visited list
        visited.append(s)

        # call function recursively with the starting element as the
        # destination of the neighbours of s
        for neighbour in self.el:
            self.DFS_(visited, neighbour.dest, end)

    return visited

# Function to print the path in the correct format as shown in the lab
# instructions [b0,b1,b2,b3]
def path_steps(self, func):
    if func == 'DFS':
        search_path = self.DFS(0,self.vertices-1)
    if func == 'BFS':
        search_path = self.BFS(0,self.vertices-1)

    for i in search_path:
        print (i, [int(x) for x in list('{0:04b}'.format(i))])

# Modified draw function from class website used to highlight path
#found from BFS or DFS
def draw_path(self, func):

    if func == 'DFS':
        search_path = self.DFS(0,self.vertices-1)
    if func == 'BFS':
        search_path = self.BFS(0,self.vertices-1)
```

```
    scale = 30
    fig, ax = plt.subplots()

    # create path list to be used to highlight path
    path = []

    for j in range(len(search_path)-1):
        path.append((search_path[j], search_path[j+1]))

    adjlist=self.as_AL()

    for i in range(len(adjlist.al)):
        for j in adjlist.al[i]:
            # highlighted path
            if (i, j.dest) in path or (j.dest, i) in path:
                line_color = "#ff007f"

            else:
                line_color = "#eeefff"

            d,w = j.dest, j.weight
            if self.directed or d>i:
                x = np.linspace(i*scale,d*scale)
                x0 = np.linspace(i*scale,d*scale,num=5)
                diff = np.abs(d-i)
                if diff == 1:
                    y0 = [0,0,0,0,0]
                else:
                    y0 = [0,-6*diff,-8*diff,-6*diff,0]
                f = interp1d(x0, y0, kind='cubic')
                y = f(x)
                s = np.sign(i-d)
                ax.plot(x,s*y,linewidth=1,color=line_color)
                if self.directed:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.plot(xd,yd,linewidth=1,color=line_color)
                if self.weighted:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.text(xd[2]-s*2,yd[2]+3*s, str(w), size=12,ha="center", va="center")
        ax.plot([i*scale,i*scale],[0,0],linewidth=1,color='k')
        ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
         bbox=dict(facecolor='w',boxstyle="circle"))
    ax.axis('off')
    ax.set_aspect(1.0)
    plt.show(block = True)

def rev(self):
    g=Graph(self.vertices, self.weighted, self.directed)

    for i in self.el:
```

```
        g.insert_edge(i.dest, i.source, i.weight)
    self.el=g.el
```

Academic Honesty Statement:

“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.”

-Laurence Labayen