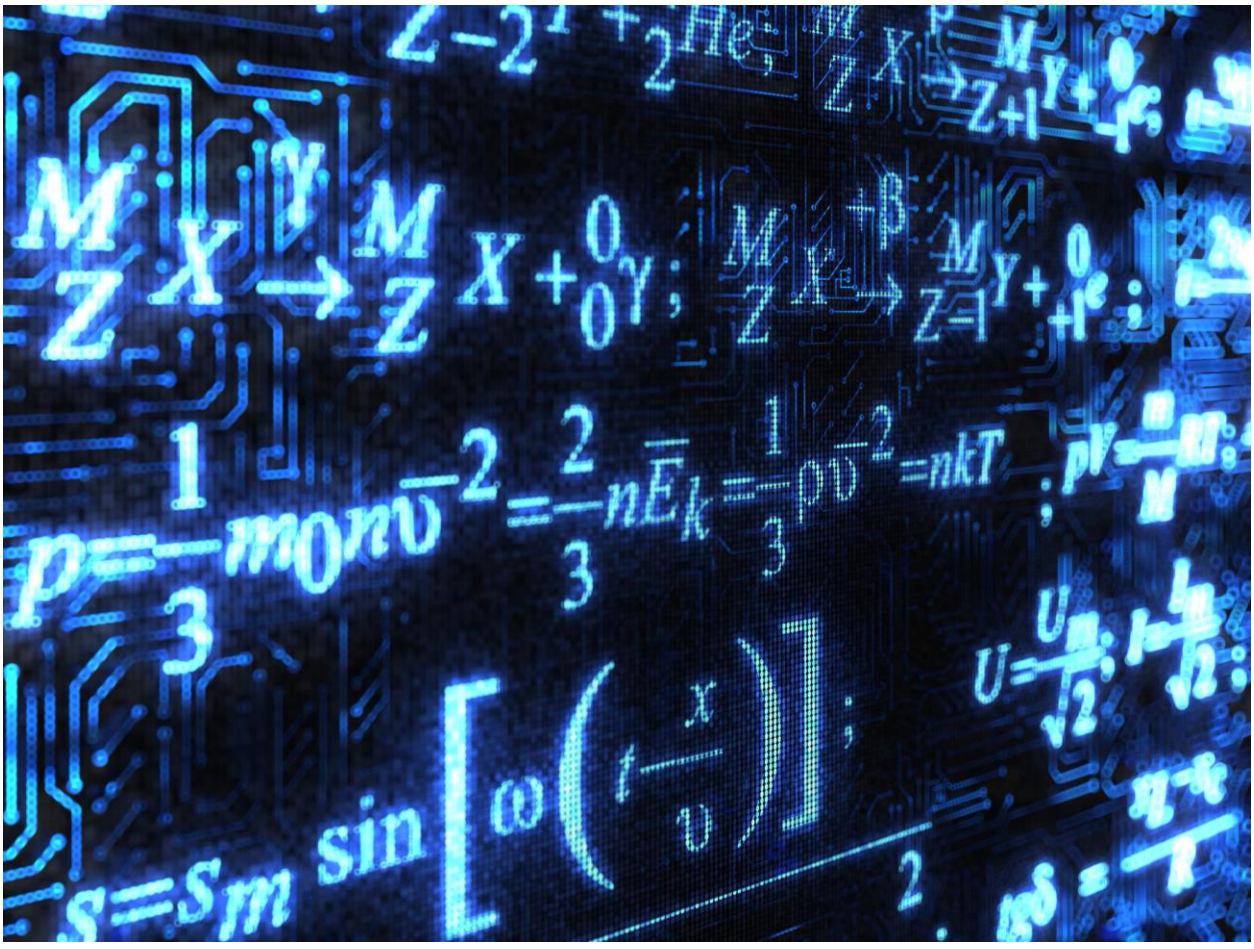
In the name of God

Project Discrete Mathematics

Erfan Nouhi 40326403



1. Introduction
2. Graph Theory
3. Graph implementation
4. Vertex Degree
5. Stack
6. Queue
7. BFS traversal
8. DFS traversal
9. Shortest path
10. Dijkstra Algorithm
11. Bellman Ford Algorithm
12. Connected Components
13. Centrality
14. Conclusion

KNT University of Technology

Prof. Hossein Khasteh

Introduction

This report belongs to my project about graphs and implementation of different operations upon them. this is my final discrete mathematic class project by 2025. My project is about the friendship graph and implementation of some algorithms and operations on that. in this report PDF, I have tried to mention my work in detail and clarify what I have learned in this project and what I have done to make my work more understandable. As I run through this report to inform you about my work I will discuss about graph theory the introduction to graph terminology and other mathematical concepts.

Graph Theory

In this part I want to discuss about the concepts and mathematical definitions of the graph theory.

Graph theory was first introduced in the works of mathematician Euler.

Graphs are basically a combination of nodes and edges.

Often, we call a node: “vertex”.

We use edges to connect a nodes or vertices together.

graphs can have direction as we call them directed graphs and they can have no directions to which we call them undirected graphs.

the edges can also have weight, if the edges have weight, we call the graph weighted graph.

Think of an undirected graph. Imagine we have a vertex that tree edges are connected to it. we say the degree of this vertex is 3 because 3 edges pass through it.

And now Think that we have a directed graph. In this kind of graphs, we may have a vertex which have one edge coming into it and for example 2 edges going out of it. Then we say that in degree of this vertex is 1 and the out degree is 2.

In degree = id = 1, out degree = od = 2

Connected Graph

Think of the graph that if you take one of its vertices you can Pick it up.

***Path* is kind of route through the graph in which you do not repeat any vertex nor edge.**

A graph is connected if we have at least one path between each pair of vertices.

***Cycle* is like a path, but you connect the first vertex and the last vertex together.**

A graph may not be connected, but it has some connected components.

Graph Implementation

the question is how to implement the graph in computer There are two main bases to implement graphs in computer, first one is using adjoint matrix and the next one is using adjency list.

in my project i have used agency adjoint matrix.

Adjoint matrix

Adjoint matrix is kind of matrix which you use to. Show the graph in computer. if your graph is undirected, you may use zero and one to show whether between two vertices the age exists or not.

	A	B	C
A	0	0	1
B	0	0	1
C	1	1	0

The matrix above is the representation of an undirected graph. If you want a directed graph, you may have an asymmetric matrix.

For representing a weighted graph, we can put another number rather than 0 or one representing the edge's weight. But still, you can have zero or one and even you can have negative weighted edges.

Adjency List

Adjency lists or another method for representation of graphs.

```
nodes = [a,b,c,d,e,f]

node_indexes = [0,1,2,3,4,5]

Adj_list = [[2,1], [0,2], [0,1], [4,5], [3,5], [3,4]]
```

In this type of representation, you have a 2-dimensional array which states that each index's adjoint vertices are in the array (second dimension) in that index.

```
[
  { "name": "Ali", "friends": ["Hossein", "Zahra","Mahdi"] },
  { "name": "Zahra", "friends": ["Ali", "Fatemeh", "Saeed"] },
  { "name": "Hossein", "friends": ["Ali", "Nazanin"] },
  { "name": "Reza", "friends": ["Samira","Kamran"] },
  { "name": "Fatemeh", "friends": ["Zahra", "Sara", "Saeed"] },
  { "name": "Saeed", "friends": ["Fatemeh", "Zahra",
"Parisa"] },
  { "name": "Mahdi", "friends": ["Amir", "Neda","Ali"] },
  { "name": "Samira", "friends": ["Reza", "Parisa",
"Nazanin"] },
  { "name": "Amir", "friends": ["Mahdi", "Neda"] },
  { "name": "Parisa", "friends": ["Saeed", "Samira", "Sara"] },
```

```
  { "name": "Sara", "friends": ["Fatemeh", "Parisa", "Arman"] },
  { "name": "Nazanin", "friends": ["Samira", "Hossein",
"Nima"] },
  { "name": "Neda", "friends": ["Mahdi", "Amir"] },
  { "name": "Nima", "friends": ["Nazanin", "Arman", "Farzad"] },
  { "name": "Arman", "friends": ["Sara", "Nima", "Mona"] },
  { "name": "Azadeh", "friends": ["Mona", "Elham"] },
  { "name": "Farzad", "friends": ["Nima", "Mona", "Elham"] },
  { "name": "Mona", "friends": ["Arman", "Farzad",
"Azadeh","Kamran"] },
  { "name": "Elham", "friends": ["Azadeh", "Farzad",
"Kamran"] },
  { "name": "Kamran", "friends": ["Elham", "Mona", "Reza"] }
]
```

In the code box above, you can see my JSON file which I have a friendship graph made from this. each names represent a vertex in my graph and the friends which represent the edges for example if Ali and were Zahra are friends we put an edge between them. As you can see, it's like a representation by adjency list.

```
#Erfan Nouhi

import matplotlib.pyplot as plt

import json

import networkx as nx

with open('C:\Users\0000\OneDrive\دسکتاپ\git-hub
work\4032_FinalProject_DMath\Erfan Nouhi 40326403-
1\src\friendship\friendship.json',"r") as file:

members = json.load(file)
```

```
adjoint_matrix = [[0 for _ in range(len(members))] for _ in
range(len(members))]

names = [] for member in members:

     names.append(member["name"])

for i in range(len(members)):

current_name_index = i

current_name = names[i]

current_friends = members[i]["friends"]

for friend_name in current_friends:

friend_index = names.index(friend_name)

   adjoint_matrix[current_name_index][friend_index] = 1


G = nx.Graph()
```

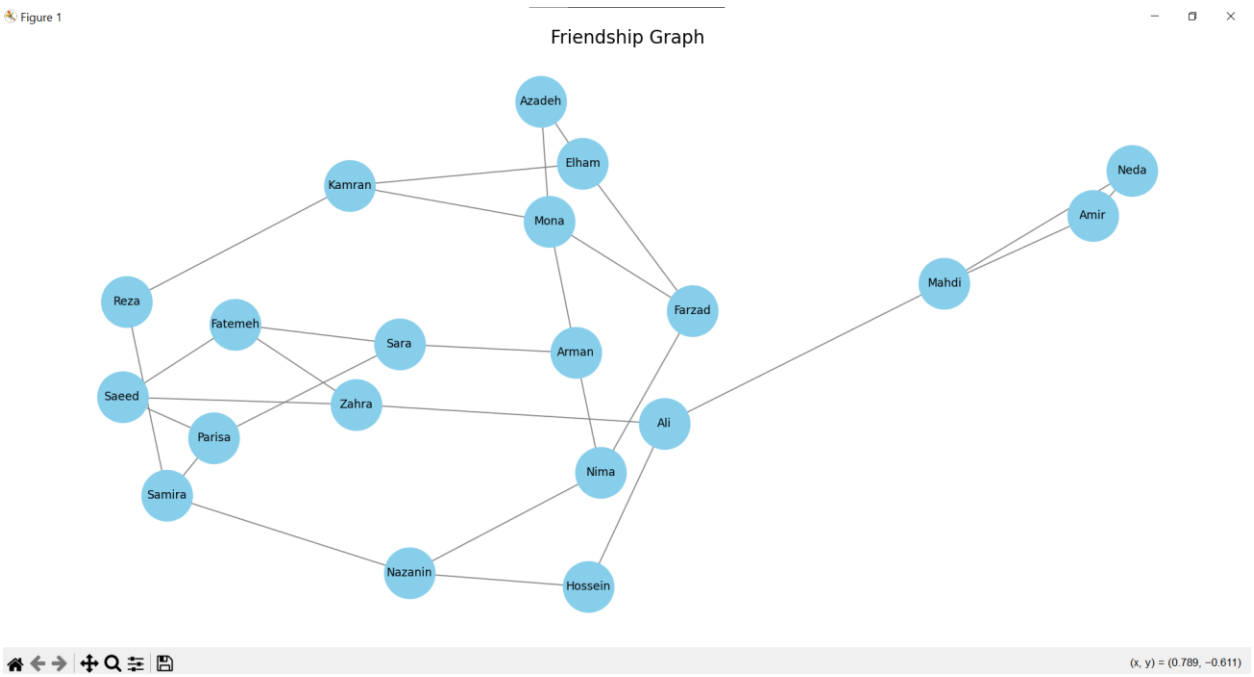
Add nodes

```
G.add_nodes_from(names)
```

Add edges based on the adjacency matrix

```
for i in range(len(adjoint_matrix)):

for j in range(i + 1, len(adjoint_matrix)):

# Only upper triangle to avoid duplicates

if adjoint_matrix[i][j] == 1:

G.add_edge(names[i], names[j])
```

```
plt.figure(figsize=(14,10))

pos = nx.spring_layout(G, seed=1,k=0.5)

nx.draw_networkx_nodes(G, pos, node_color='skyblue',
node_size=2000)

nx.draw_networkx_edges(G, pos, edge_color='gray')

nx.draw_networkx_labels(G, pos, font_size=10, font_family='sans-
serif')

plt.title("Friendship Graph", fontsize=16) plt.axis('off')

plt.tight_layout() plt.show()

for member in members:

print("name ",member["name"])

for friend in member["friends"]:

print('\t',friend)
```



Friendship graph

```
names = [] for member in members:
names.append(member["name"])

for i in range(len(members)):

current_name_index = i

current_name = names[i]

current_friends = members[i]["friends"]

for friend_name in current_friends:

friend_index = names.index(friend_name)

    adjoint_matrix[current_name_index][friend_index] = 1
```

I have loaded the JSON file to my Python program and then I have made an adjoint matrix and names list from that. as i have mentioned earlier my project is about friendship graph and friendship graph is a mutual relationship graph which means that the graph is undirected. and as i said earlier i may represent this undirected graph with adjoint matrix using zero and one as elements.

Now look the above code block of red part and you can see the rest of the code is going to use networkx. Library which is used to work with difficult work with complicated networks. here i used my adjoint matrix to show whether between two vertices i have an edge or not and if i have i add the edge between these two vertices. simultaneously i have used matplotlib to demonstrate the graph.

This graph is connected as you may see in your running time. you can make it unconnected if you want by just changing the JSON file.

Vertex Degree

I have considered this Graph to be undirected, so I don't have indegree and outdegree however this is just a simple degree which is the number of edges passing through the vertex.

If a vertex degree is 0 it would be alone and if the vertex's degree is the (number of vertices –1) this is the full vertex. Here it means that if a vertex is full, that person has as many as friends as possible. If it is alone vertex, it means there is no friends for that specific person.

```
def in_out_deg(name):

     name_index = names.index(name)

print(sum(adjoint_matrix[name_index]),"this graph is
undirected")


in_out_deg("Hossein")
```

As we know in adjoint matrix a row represents a person and if there is one inside this row and a column it means that this person has a friend with that specific column. We sum up the ones in one specific row and we get the number of edges passing through that specific row which represents a person. And if it is a weighted graph, we can get the number of non-None elements in that room. Which we use non instead of 0 in weighted graph. Because the graph may have a 0 weighted edge.

Stack

Stack is a kind of data structure which like a container, you can just put on the top of it, and you can peek something again from the top. This this data structure is just one sided and you can work with one side of it and the top element is called head.There are different operations that you can do upon this kind of data structure. Some basic ones are like peeking or checking the size of this stack. You can put, push which push means that you peek the top element, the head, and you remove it from the stack. and the next element becomes the head.

Queue

Queue is another data structure which you can append elements, and you can remove elements, but this is a two-sided data structure which It's not like the stack that is just one sided. You can add element to the rear of the queue, which is called enqueue and you can dequeue, this kind of phrase is that you can remove the element from the front. Again, like stack data structure you can do several operations upon queue data structure, for example you can enqueue, dequeue, pick the front element or check the size of that and check whether it is empty or not.

We use these two data structures to implement DFS and BFS traversal for graphs.

DFS

```
def DFS(visited,start_name):

     start_index = names.index(start_name)

     visited[start_index] = True

     print(start_name," ")

for i in range(len(names)):
    if adjoint_matrix[start_index][i] != 0 and not visited[i]:
        DFS(visited,names[i])
```

To implement DFS algorithm, which is depth first search for graphs, we use recursion. For example, we start in a specific vertex, and we go to its neighbors then we repeat this, until we reach the end of the graph and that means there is no-unchecked vertices left. We need to keep track of the visited vertices we use the visited array to keep record of them and then mark the vertex as visited we put it inside this visited array and we print it. We go through the unvisited neighbors of that specific node and do this recursively. here actually in depth we are using a stack data structure.

BFS

def BFS(start_name): start_vertex = names.index(start_name) queue = [start_vertex] visited = [False for _ in range(len(names))] visited[queue[0]] = True

```
while len(queue) > 0:
    current_node = queue.pop(0)
    print(names[current_node] , " ")
```

```
    for i in range(len(names)):
        if adjoint_matrix[current_node][i] != 0 and not
visited[i]:
            queue.append(i)
            visited[i] = True
```

To carry out BFS algorithm we use Queue data structure. As you can see in the above code box, I have used the queue which I put elements in that and as I said, enqueue. We start at a vertex, we put it inside the queue and then we pick it up (dequeue) , and we store it inside another variable and then we go through the neighbors of that specific vertex, we append them to the queue. Next, we visit them in visited array.

Shortest Path

Now in this section I want to introduce the shortest path's concepts and algorithms as I learned in my own tutorial on W3schools. We have several algorithms to find the shortest path, the shortest path means that you will find a path with minimum weight between two specific vertices, if the graph is weighted. For example, if we have several cities and we want to find out the minimum mileage that you may take to reach another city from the from the origin.

- Dijkstra

- Bellman Ford
- BFS
- DFS
- .....

We can also use BFS and DFS themselves but, I have didn't do such a thing . I have implemented Dijkstra and Bellman Ford algorithms.

Dijkstra Algorithm

```
def dijkstra(start_name):

     start_index =  names.index(start_name)

     distances = [float('inf')] * len(names)

     distances[start_index] = 0

     visited = [False] * len(names)

for _ in range(len(names)):
    min_distance = float('inf')
    u = None
    for i in range(len(names)):
        if not visited[i] and distances[i] < min_distance:
            min_distance = distances[i]
            u = i

    if u is None:
        break

    visited[u] = True

    for v in range(len(names)):
        if adjoint_matrix[u][v] != 0 and not visited[v]:
```

```
            alt = distances[u] + adjoint_matrix[u][v]
            if alt < distances[v]:


                distances[v] = alt
return distances
```

Dijkstra algorithm is used to find out the minimum weight path from one specific vertex to all the other vertices. We need some arrays to keep track of work. For example, we need a visited array like the thing we used in DFS, and we need a distances array which is the result and shows the minimum weight path distance from one particular Vertex into the others. We set all the distances array's elements to Infinity. We just reset the specific vertex that we want to start with zero. We go through the unvisited vertices and then we find out which one has the minimum distance currently. If we find this specific vertex, we will continue. If not, the program is over, and we have visited all the vertices, and the program is finished. We mark this vertex as visited and then we go through the neighbors of this vertex if they are not visited and we will find out if the current minimum path through these specific vertex and the added weight from the edge between this vertex and its or neighbor is less than the neighbor's minimum path currently, then we will relax that neighbor and that means we will update the minimum path of that neighbor.

Bellman Ford Algorithm

```
def bellman_ford(start_name):

     start_index = names.index(start_name)

     distances = [float('inf')] * len(names)

     distances[start_index] = 0

for i in range(len(names)- 1):
    for u in range(len(names)):
        for v in range(len(names)):
            if adjoint_matrix[u][v] != 0:
```

```
                if distances[u] + adjoint_matrix[u][v] <
distances[v]:
     distances[v] = distances[u] + adjoint_matrix[u][v]

return distances
```

Bellman Ford algorithm is somehow like Dijkstra algorithm, but it doesn't have any visited array. We go through the adjoint matrix with (number of vertices –1) times and we do something which leads to finding minimum path toward all the other vertices from a particular vertex. These number of times which we do this loop it's because in a graph with V vertices we will have at most. (V - 1) path length. We go through all edges to relax them if their current minimum path length is higher than the minimum length path of its parent plus the edge weight between these two vertices.

Connected Components

```
def
dfs(visited:list,vertext_index,component:list,vertices:list,adj_
matrix:list):

     component.append(vertext_index)

     visited[vertext_index] = True

for i in range(len(vertices)):
    if adj_matrix[vertext_index][i] and not visited[i]:
        dfs(visited,i,component,vertices,adj_matrix)


def find_components(visited:list,adj_matrix:list,vertices:list):

result = []

for i in range(len(vertices)):
    component = []
    if not visited[i] :
        dfs(visited,i,component,vertices,adj_matrix)
    if len(component):
```

```
        result.append(component)

return result


visited = [False] * len(members)

components = find_components(visited,adjoint_matrix,names)
```

In this part I will discuss about finding the connected components. As I formerly said that graphs could be connected or unconnected to find the connected components, I use DFS algorithm here. The result is a two-dimensional array which separates the connected components from each other. Each sub arrays of these two- dimensional array is a connected component. As we know, DFS algorithm works with the neighborhood, which means if we have an unvisited vertex, we go through its neighbors, so the neighbors are connected to that vertex, and it means that DFS works with one connected component. We go through all the unvisited vertices, and we passed them to DFS function. Then we will get the result as it is a kind of array of all connected vertices together, we will append it into the two-dimensional result array and then we go to the next unvisited vertices in which maybe we will find another connected component then if we find that we will append it to result array.

Centrality

I have learned that centrality mainly means that which vertex is the most important or which vertices are the most important vertices.

We can implement the centrality of the graph with respect to minimum sum of distances to other vertices or for example to find the centrality for vertex which connects other vertices together. But in this project, I have used the first one which I implemented the centrality algorithm based on the minimum sum of distances from one vertex to all the others.

```
def dijkstra(start_name):
```

```
     start_index = names.index(start_name)

     distances = [float('inf')] * len(names)

     distances[start_index] = 0

     visited = [False] * len(names)

for _ in range(len(names)):
    min_distance = float('inf')
    u = None
    for i in range(len(names)):
        if not visited[i] and distances[i] < min_distance:
            min_distance = distances[i]
            u = i

    if u is None:
        break

    visited[u] = True

    for v in range(len(names)):
        if adjoint_matrix[u][v] != 0 and not visited[v]:
            alt = distances[u] + adjoint_matrix[u][v]
            if alt < distances[v]:
                distances[v] = alt

return distances


distances_sum = []

for name in names:

distances = dijkstra(name)

sum_distances = sum(distances)
distances_sum.append(sum_distances)

for name,distance_sum in zip(names,distances_sum):
print(f"{name} sum of distances from rest nodes is
{distance_sum}")
```

```
min_sum_distance = 100000 for i in range(len(distances_sum)): if
distances_sum[i] < min_sum_distance: min_sum_distance =
distances_sum[i]
```

To reach the goal, I have used Dijkstra algorithm to find the minimum path from one vertex to all the others. I have made a loop through all the vertices and summed up the minimum path from that vertex to others which is delivered from Dijkstra for each vertex. I have put this sum of distances into a particular array called distances sum and then I find the minimum index, the minimum value from this array and map it to the name of the person for this index.

Conclusion

In this project report, I have considered my learning path, mathematical concepts behind graphs and implementation of different algorithms for these graphs. I have tried to convey my knowledge in best way I can. I will be really pleased that you give some feedback. I used some different technologies like python programming language, Visual Studio code, git and GitHub, therefore you can access my codes on my GitHub account which follows, Also, I have learned a lot on w3schools.com which is my source for this project.

Erfan Nouhi GitHub

W3schools.com