Logo

Description automatically generated

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

IT3910E - Project I

**Visualization of several graph algorithms and applications**

|  |  |
| --- | --- |
| *Authors:* | Student ID: |
| Nguyễn Văn Hưng | 20204913 |
|  |  |
|  |  |
|  |  |

Instructor

Cao Tuấn Dũng

Feb 2023

Contents

[Acknowledgement 2](#_Toc127215360)

[1. Research Problem 2](#_Toc127215361)

[2. Graph and algorithms 4](#_Toc127215364)

[2.1. Theoretical basis 4](#_Toc127215365)

[2.2. Tools used in project 13](#_Toc127215366)

[3. Main content 14](#_Toc127215371)

[3.1. Survey 14](#_Toc127215372)

[3.2.Files creating and management 20](#_Toc127215373)

[3.3. Main program 22](#_Toc127215374)

[3.4. Future improvement 26](#_Toc127215375)

[4. Advanced application 26](#_Toc127215379)

ACKNOWLEDGEMENT

I would like to express my deep gratitude to Prof.Dung for the guidance and for giving me the opportunity to work on this project. He inspired me to pursue our interests and delve deeper into the subject.

# 1. Research problem

For a programmer, or a computer scientist, understanding these algorithms is very important to be able to apply them in practice. If we were to write a piece of software, we would have to evaluate how quickly the software would work. Such estimates would be much less accurate if one had no understanding of runtime or complexity. Plus, knowing the algorithm of what we're doing will help us predict special circumstances that cause software to slow down or crash.

Of course, we will often encounter problems that have not been studied before. Now we have to come up with new techniques ourselves, or apply old ones more creatively. The more knowledge we have about algorithms, the more likely we are to successfully solve the problem. In many cases, a new problem can be brought up to an older problem without much effort, provided that one has a sufficiently deep knowledge of the old problem.

There are many examples showing real-world problems that require an understanding of algorithms. Pretty much everything you're doing with a computer is based on some algorithm that someone had to work very hard to figure out. Even the simplest application of a modern computer requires algorithms to manage memory data and retrieve information from the hard drive. In other words, an algorithm is like a road map that helps us solve a clear, specific problem.

Some basic application:

* In **Computer science** graphs are used to represent the flow of computation.
* **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
* In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook’s Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.
* In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind [Google Page Ranking Algorithm](https://www.geeksforgeeks.org/page-rank-algorithm-implementation/).
* In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.
* In **mapping system** we use graph. It is useful to find out which is an excellent place from the location as well as your nearby location. In GPS we also use graphs.
* **Facebook** uses graphs. Using graphs suggests mutual friends. it shows a list of the f following pages, friends, and contact list.
* **Microsoft** **Excel** uses DAG means Directed Acyclic Graphs.

Because the applications of algorithms, notably graph algorithms, are very large, spanning almost all fields, so their study and in-depth study are essential. Through this Project we will achieve:

• Deeper and better understanding of how some basic graph algorithms work.

• Enhance programming skills through visualization and modeling of algorithms.

• Understand how graph algorithms work in applications and then apply them to learning and real work.

# 2. Graph and algorithms

# 2.1. Theoretical basis

# a) The concept

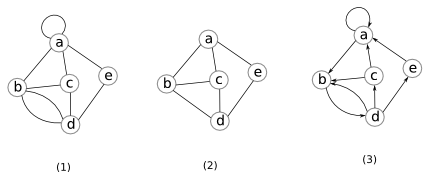
A graph, denoted G(V,E), consists of two components:

1. The set V, including objects, is called the set of vertices (vertex) of the graph

2. The set E⊆V2, consisting of a pair of vertices, is called the set of edges (vertex) of the graph.

We will denote n, m as the number of vertices and edges of the graph, i.e, |V|=n,|E|=m, respectively. The number of vertices in a graph is sometimes called the degree of the graph.

The vertices we will denote with lowercase letters like u,v,x,y,z. The edge between two vertices u,v can be either scalar or directed. In the first case we will denote the edge as uv, and in the latter case we will denote it as u→v to specify the direction of the edge from u to v. Usually when we say edge we mean the undirected edge and for a directed edge we call it an arc. Figure (1,2) of the figure below shows an undirected graph (the edges are undirected) and figure (3) to the right of the figure below represents a directed graph.



In figure (1), the edge (a,a) is called the loop edge and the two edges between the pair of vertices (b,d) are called the two parallel edges. A graph is called a simple graph if it has no repeating and parallel edges (Figure (2)). If a graph is not a single graph, we call it a multigraph. In the series of graphs here, we mainly consider single graphs. Therefore, when we say graph, we will implicitly mean a simple graph. We have:

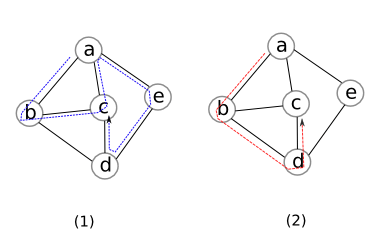
**Fact 1: If G(V,E) is a simple undirected graph, then**

If the graph G(V,E) is undirected, for each edge uv, vertex v is said to be adjacent to edge uv. The vertex u is called the neighbor of v. The degree of vertex v, usually denoted d(v), is the number of neighbors of vertex v. If the graph G(V,E) is directed, for each arc u → v, the vertex v is called the successor of u and the vertex u is called the preceding vertex of v. The in-degree of v is the number of vertices immediately preceding v and the out-degree of v is the number of vertices immediately following v. For example, the forward step of d in figure (3) is 1 and the backward step is 3.

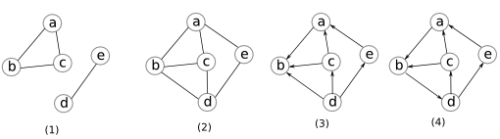
We call H(VH,EH) a subgraph of G if VH⊆V and EH⊆E.

A path is a sequence of edges {e1,e2,…,ek} where any two adjacent edges ei and ei+1 share a common vertex.

Note that the path can pass through a vertex more than once. In case each vertex is visited exactly once, we will call it a single path. For example in the diagram, in the figure below, {ab,bc,ca,ae,ed,dc} is a path between a and c and {ab,bd,dc} is a simple path between a and c. A closed walk is a path that begins and ends at the same point. A cycle is a single path that begins and ends at the same point. It can be said that a cycle is a closed path that passes through every point exactly once except for the beginning and the end. The above concepts, if applied to directed graphs, we will add the word "directed" in front.



An undirected graph is said to be connected if there exists a path between every pair of points. A directed graph is (weakly) connected if the undirected graph is obtained from the graph by ignoring the direction of the connected edge. A directed graph is said to be strongly connected if there exists a directed path between every pair of points. Obviously, if a directed graph is strongly connected, it is also weakly connected. However, the opposite is not necessarily true (for example?). For example, graph (1) below is not connected, graph (2) is connected, graph (3) is weakly connected (but not strongly) and graph (4) is strongly connected.



If a (undirected) graph is not connected, the set of connected vertices forms a connected component. Similarly, we can define the connected component (weak or strong) for a directed graph. A graph without acyclic is called a forest. A forest with only one connected component is called a tree. The concept of directed trees and forests is similar to directed graphs.

**Fact 2: If G(V,E) is a tree, then m=n−1 . If G(V,E) is a forest, then m ≤ n−1.**

Perhaps we will stop defining the concept here. There are many more concepts we will define as we need. Nearly all the basic concepts are listed in [2] that readers can refer to.

In the next section, we consider scalar G(V,E). The operations with directed graphs can be extended and applied in a similar way.

# b) Graph representation

We can represent the graph with an adjacency matrix A of size n×n where:

**A[u,v] = (1)**

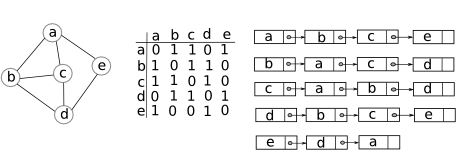
It can be seen immediately that the size of this representation is O(n2) regardless of whether the number of edges is more or less. According to Fact 1, the number of edges m of a graph can be up to O(n2) edges (we call it thick graph). Therefore, this representation can be said to be suitable for thick graphs. However, many graphs (especially real-life graphs like social networks) have the number of edges m=O(n) (we call them sparse graphs). Therefore, this representation is quite expensive for sparse graphs.

To save memory, for each vertex u∈V, we store a list of adjacent vertices. Thus, vertex u needs a list with d(u) elements. Hence the total number of elements of the lists is:

The sum is because each edge is counted twice in the sum of the degrees of the two adjacent vertices. Such a representation is called an adjacency list. This representation is suitable for sparse graphs as well. Despite saving memory, this representation is not suitable for some graph operations. The following table compares the two representations just presented.

|  |  |  |
| --- | --- | --- |
|  | Adjacency matrix | Adjacency list (linked list) |
| Space |  |  |
| Test |  |  |
| List all neighbors of |  |  |
| Add an edge |  |  |
| Delete an edge |  |  |

Examples of two graph representations are shown in the figure below:



You can also combine the adjacency list representation with some other data structure. In particular, instead of using a linked list to represent vertices adjacent to a vertex , we can also use a hash table or tree structure to represent it. In the framework of the articles here, we rarely use (or do not use) such structures.

Alternatively, we can represent the graph by listing all pairs that satisfy . This representation has a memory of O(m). However, performing the basic operations in this representation would be very expensive. It is sometimes possible to combine this representation with the adjacency list representation to take advantage of both representations while the memory is still linear.

# c) Graph traversal

Problem : Given a graph G(V,E) and a vertex s ∈V, print vertices v such that there is a path from s to v.

We call the above problem the problem of traversing the graph from a vertex s. For the sake of simplicity, we will assume that the graph is connected. The general way to traverse the graph is as follows: We will use two types of labels to assign to the vertices of the graph: unvisited and visited. Initially all vertices are marked as unvisited. We will maintain a set C (how to implement C set we will learn later), initially empty. We will repeat the following 2 steps:

1. Get a vertex u in C (REMOVE(C) procedure below).

2. Mark u as visited.

3. Put the neighbors of u with unvisited labels in C. The ADD(C,v) procedure below will put the vertex v in the set C.

The algorithm stops when C=∅. Pseudocode as follows:

GenericGraphTraverse(G(V,E),s):  
    mark all vertices unvisited  
    Add(C,s)  
    **while** C≠∅  
        u← Remove(C)         *≪(∗)≫*  
        **if** u is unvisited  
            mark u *visited*  
            **for** all uv∈E and v is *unvisited*         *≪(∗∗)≫*  
                Add (C,v)

Remark: A vertex can be included more than once in the set C (so C is not a set because it has many of the same elements). For example, consider 3 adjacent vertices u,v,w. The u vertex is removed from the first C; mark u as visited. Soon, v and w will be put into C. Next, take v out of C and mark v as visited. Now we continue to put w into C again because according to the above pseudocode, w is a neighbor of v and has an unvisited label. Here, we will not check if a vertex is already in C before we put it in C.

From the above pseudocode, we see, set C stores vertices adjacent to at least one visited vertex.

Algorithm Analysis: Assume that we use a structure to implement C such that adding or taking any vertex (line (∗) and last row) is done in O(1) time ( for example, if implementing C using a linked list, adding or removing the vertex at the top of the list can be done in O(1) time). We have a few observations:

1. The vertices that have been removed from C and marked as visited will never be returned to set C again.

2. Each time vertex v is inserted into C, one of its neighbors will be marked as visited. Therefore, vertex v will be inserted into C no more than d(v) times.

3. Every time we take a vertex u from C, we will traverse all of u's neighbors. This operation takes O(d(u)) time. According to comment 1, this traversal can only be performed at most once.

From the above comments, we infer that the total computation time of the algorithm is

In the case of a non-connected graph, we have to traverse the connected components one by one. Since the graph has at most n connected components, we have:

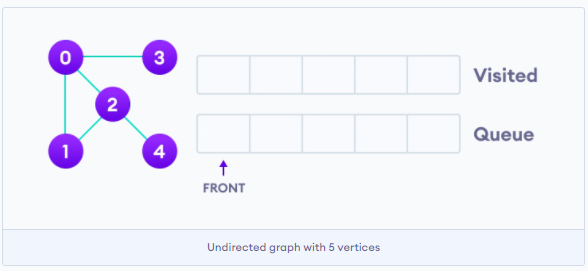
**Theorem : We can traverse the graph in time.**

If we were to implement C using a linked list, then perhaps nothing would be interesting. However, if we implement C using a queue or a stack, we get some interesting properties from the graph. In case we execute C using a queue, we call the algorithm Breath First Search (BFS). In case we execute C by stack, we call the algorithm depth-first search (DFS).

# d) Some basic algorithms

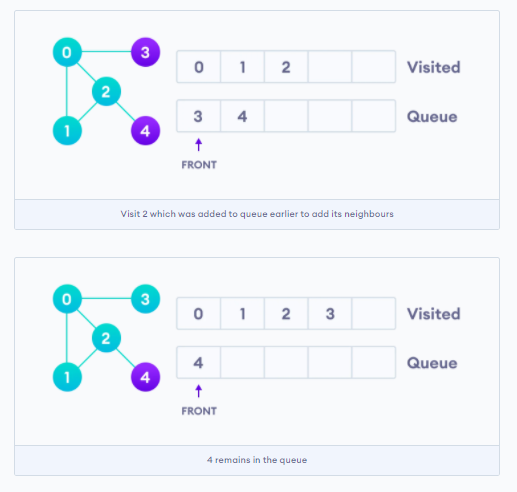
1. **Breath-first search algorithm BFS**

BFS(G(V,E),s):  
    **for each** v∈V  
        d[v]←+∞  
    C← an empty Queue  
    Enqueue(C,s)  
    d[s]←0  
    **while** C≠∅  
        u← Dequeue(C)  
        **if** u is unvisited  
            mark u *visited*  
            **for** all uv ∈ E and v is *unvisited*         *≪(∗∗)≫*  
                Enqueue(C,v)





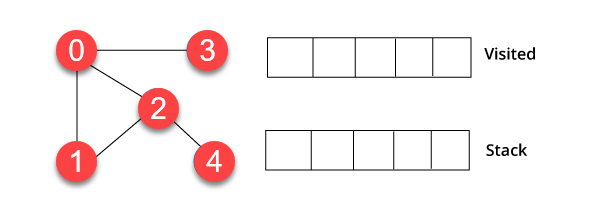


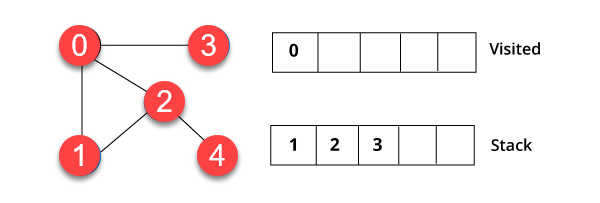


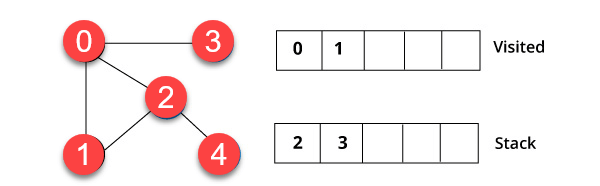
* + - BFS uses the principle First In First Out (FIFO) and is implemented by queue

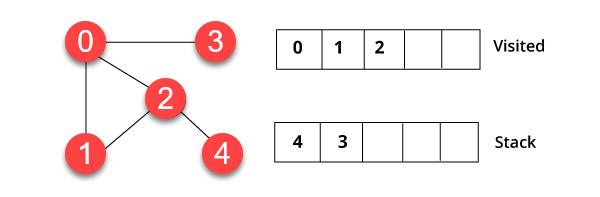
1. **Depth-first search algorithm DFS**

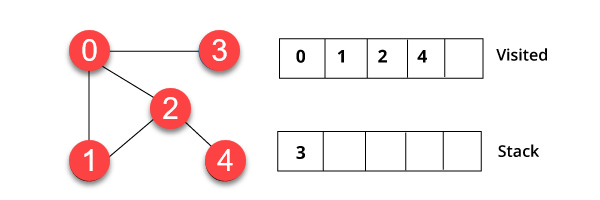
DFS(G(V,E),s):  
    C← an empty Stack  
    Push(C,s)  
    **while** C≠∅  
        u← Pop(C)  
        **if** u is unvisited  
            mark u *visited*  
            **for** all uv∈E and v is *unvisited*         *≪(∗∗)≫*  
                Push (C,v)











* + - DFS uses the principle Last In First Out (LIFO) and is implemented by stack

# e) Detecting connected components

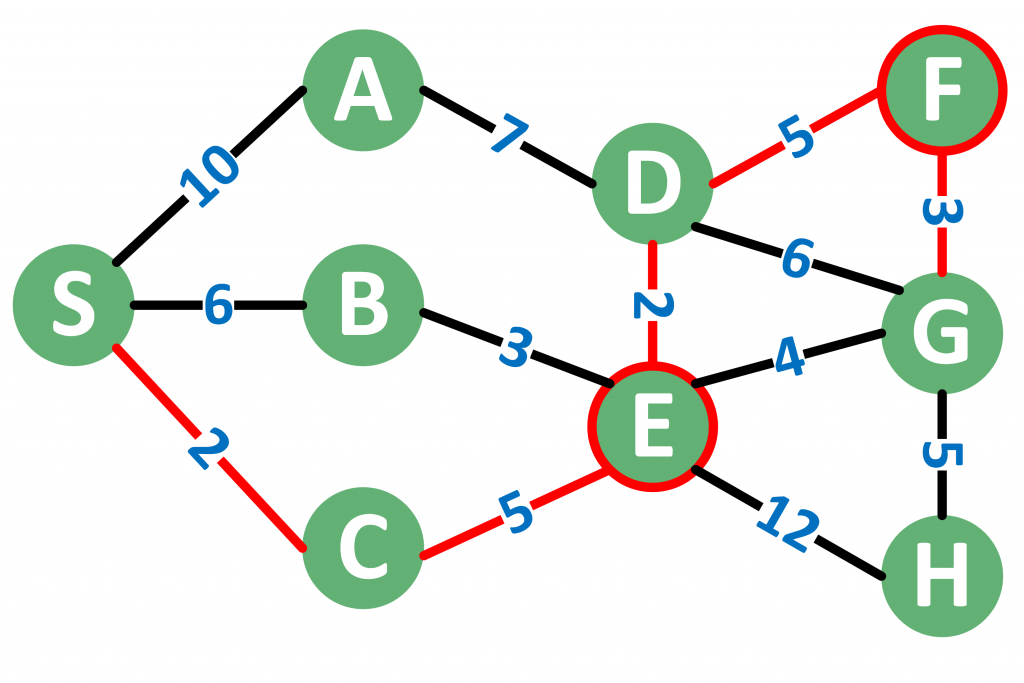
One of the simplest applications for graph traversal is to discover connected components. To discover the connected components of a (undirected) graph, we repeat the following operation: select an unvisited vertex u and perform a visit to the vertices in the connected component containing u. The following procedure returns the number of connected components of the input graph

ConnecteComponents(G(V,E)):  
    mark all vertices *unvisited*  
    count←0  
    **for** all vertices s∈V  
        **if** s is *unvisited*  
            GraphTraversal(G(V,E),s)        *[[any graph traversal algorithm]]*  
            count←count+1  
    return count

# f) Shortest path finding problem

This is a subset of the graph traversal problem. In this problem, we’re given a weighted, directed or undirected graph. **The task is to find the shortest path that starts from a source node S and ends at a goal node G. In addition, the problem requires that the resulting path visits certain nodes, in an order**.

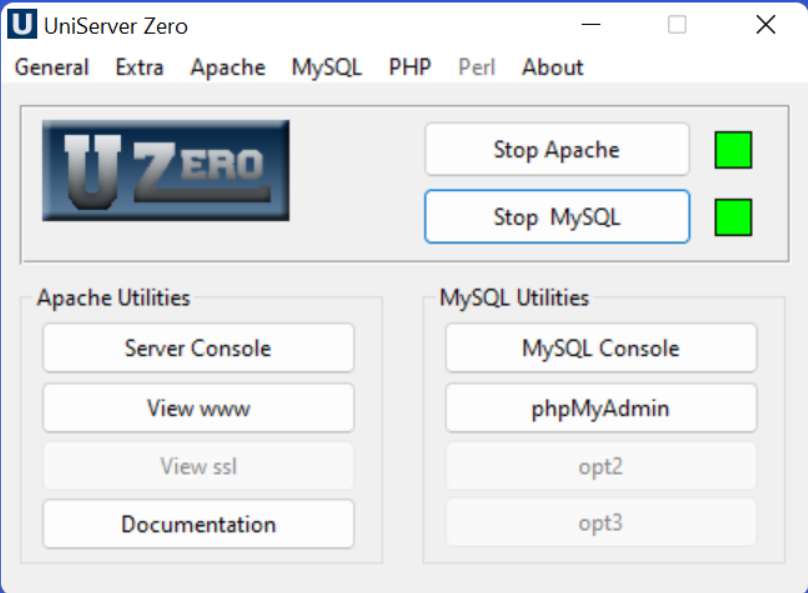
Let’s take the following graph as an example:



# 2.2. Tools used in Project

To make it easier to visualize the models when visualizing the algorithms, I will put it all into a simple web app server. This makes the program interface more beautiful, users are easier to manipulate. In this project I, I used The Uniform Server application to create a local host server for the purpose of deploying the program.

The interface of the application looks as follows:



I have combined the use of HTML, CSS and JavaScript to create the interface and algorithm for the program.

# 3. Main content

# 3.1. Survey

# a) Visualgo.net

VisuAlgo was conceptualized in 2011 by Dr Steven Halim as a tool to help his students better understand data structures and algorithms, by allowing them to learn the basics on their own and at their own pace.

VisuAlgo contains many advanced algorithms that are discussed in Dr Steven Halim's book ('Competitive Programming', co-authored with his brother Dr Felix Halim and his friend Dr Suhendry Effendy) and beyond. Today, a few of these advanced algorithms visualization/animation can only be found in VisuAlgo.

A picture containing map

Description automatically generated

In the main screen part is to display the graph before, during and after browsing

Text, timeline

Description automatically generated

In the bottom right corner of the screen is a pseudocode. Every time the main screen is active to a certain part, the code will highlight that active part. For example, in the image above, the graph is traversing the next element in the queue Q.

A picture containing PowerPoint

Description automatically generated

In the lower left corner will be the place for the user to choose the algorithm as well as edit the graph. When we choose Edit graph, we can add nodes, edges, weighted or not

Chart, radar chart

Description automatically generated

Advantages:

- Eye-catching graphics, easy to follow

- The system has many algorithms for users to choose easily

- There is a code to easily monitor how the algorithm works

- There is a manual when choosing the algorithm

- Users can manually edit the graph, edit the node...

Disadvantages:

- It is not possible to choose the starting and ending points by yourself when choosing find shortest path mode, so when choosing an algorithm, the graph will traverse from a random vertex of the end.

- The added weight is determined and cannot be edited, so it is not realistic or intuitive

# b) Cs.usfca.edu

Graphical user interface, text

Description automatically generated

At the beginning of the website, users can choose one of many popular algorithms today. But when choosing one of the algorithms, the web app becomes quite simple

Diagram

Description automatically generated with medium confidence

The web application consists of only two main parts, which are selection, graph adjustment, and algorithmic performance display.

In the graph adjustment section, we can only choose to start vertex, the graph is directed or not and what to display on the screen (regular graph or adjacency matrix or adjacency list)

Advantages:

- Easy to use, easy to follow

- There are certain necessary functions

Disadvantages:

- Unable to select the end vertex

- The graph that the user is selected is random, no editing is allowed

- Only the table shows the distance from point A to point B without showing the code to specify the partial operation

- The biggest disadvantage is that the website is still sketchy and has little functionality

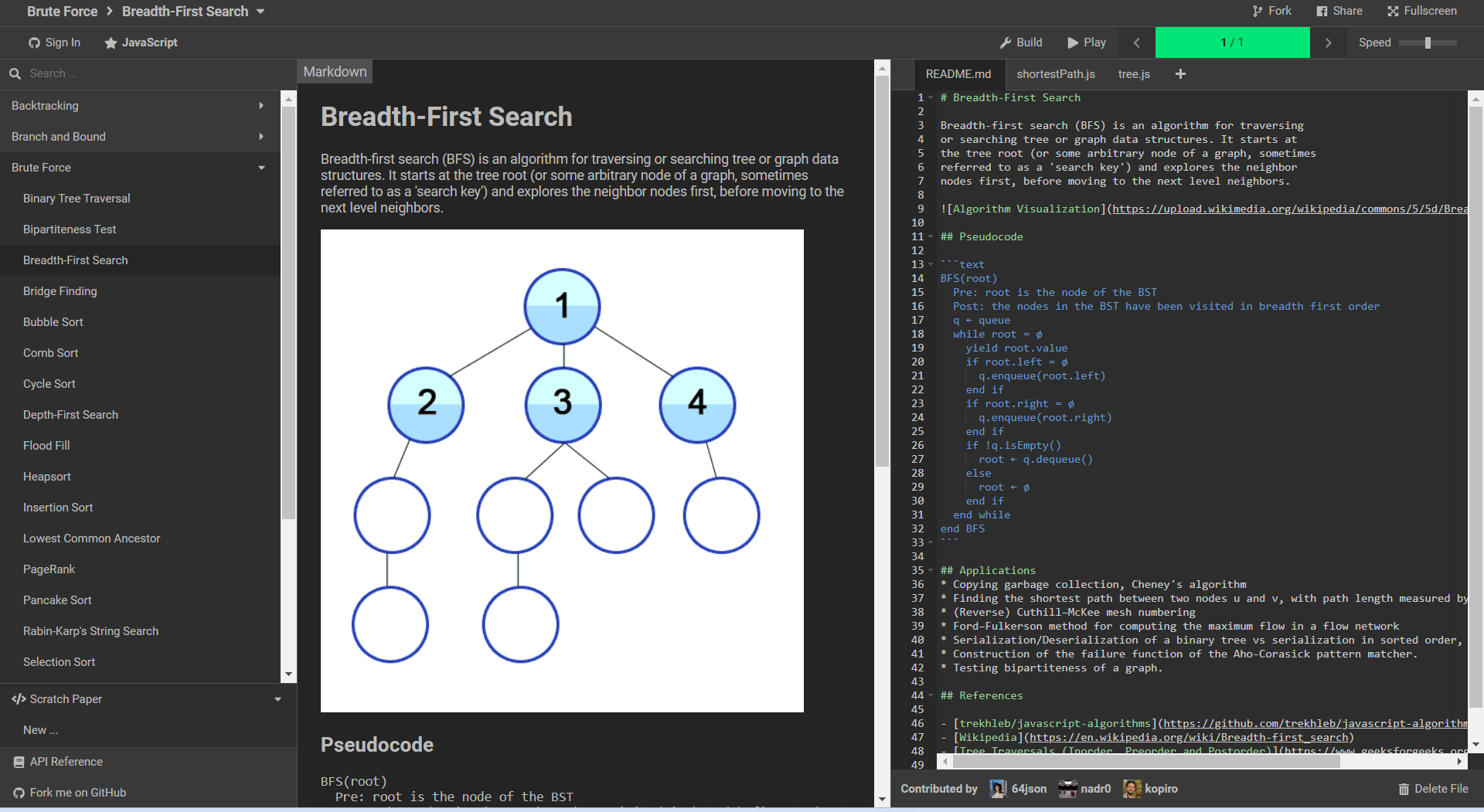
# c) Algorithm-visualizer.org

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence



The website consists of 3 main parts:

- Many algorithms are listed by category on the left side of the screen

- The right part we can freely edit the code, the data of the adjacency matrix or consult the theory of the algorithm through the file README.md

- The middle of the screen is used to represent graphs as well as theory and pseudocode

- There are outputs of the paths passed

Advantages:

- Very beautiful and eye-catching design

- Categorize each workspace, easy to use and edit

- Can show code and pseudocode for users to follow easily

Disadvantages:

- Cannot be edited by adding nodes, edges, weights directly in the demo

- The code is for reading only, don't understand which part works corresponding to the active part in the demo

- If you want to edit the graph, you must edit it directly in the code, it is very difficult for those who have never read it and have no experience.

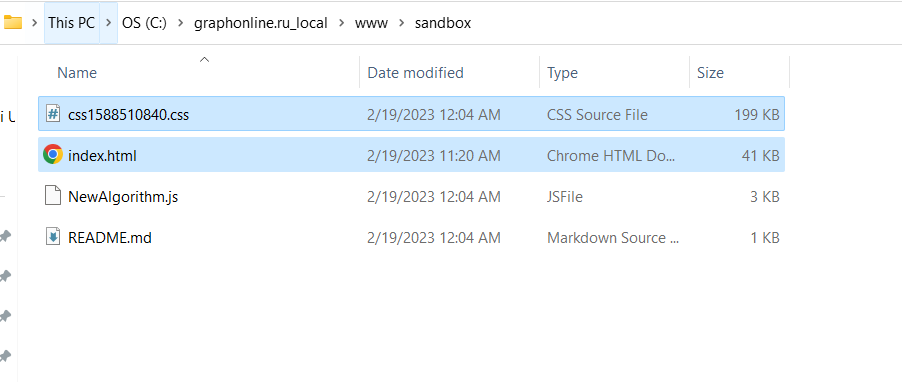
- Code distribution is not good, must separate the matrix data part

# General comment:

Every website has its own pros and cons. The common feature of websites is that they do not allow direct editing of graphs, often use random or have to edit code. There is a website that highlights the code that is working very well like Visualgo.net, Algorithm-visualizer.org does a very good job with the LogTracer part… If you can combine all the advantages and add some tweaks, it will be much more optimized.

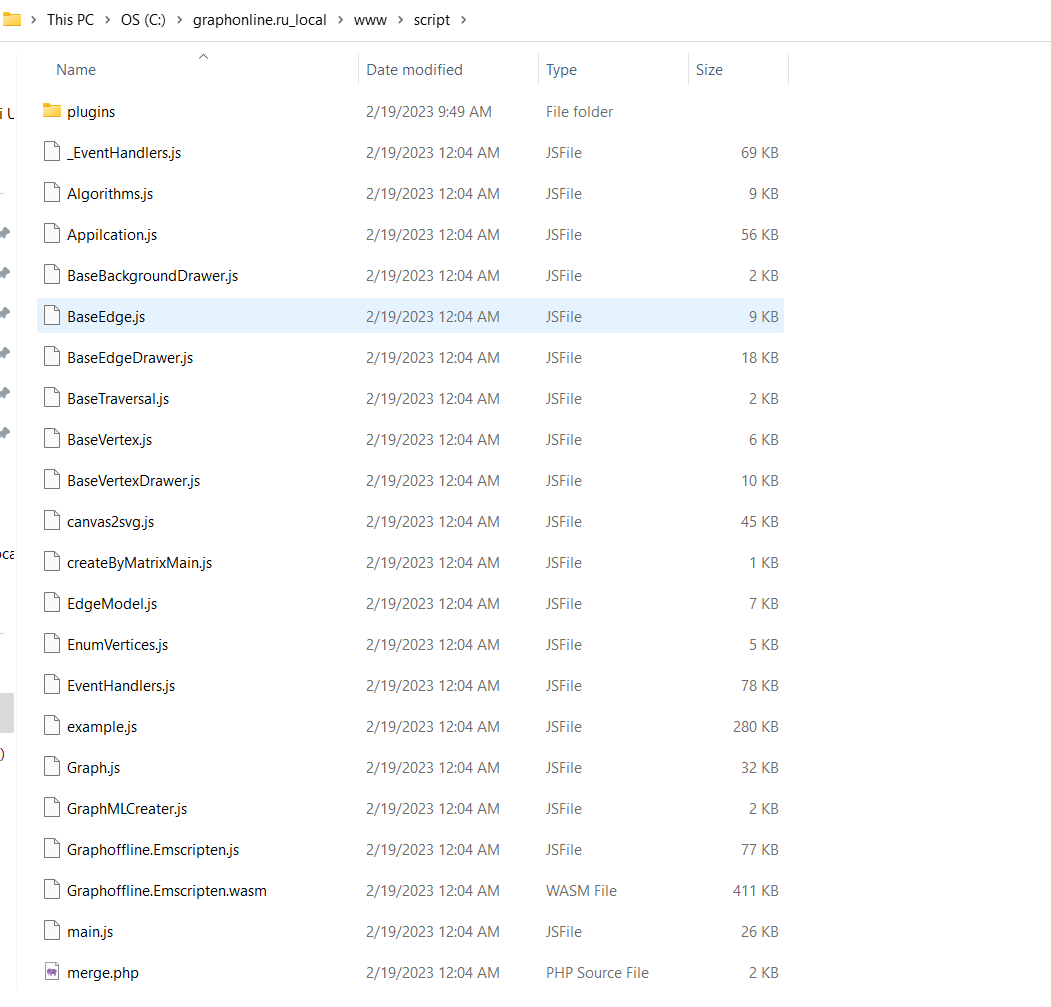
# 3.2. Files creating and management

The first step is to create a graphical interface using html and css files. These 2 files will be placed side by side.



File index.html will be linked to file NewAlgorithm.js to run graph algorithms on it.

But the whole main code for running program is on script file.



All code is written in JavaScipts.

Files about vertex and edge are to create and setting them. Files about graph have mission to create the adjacency matrix for algorithms.

Information about the coordinate parameter when the user creates the vertex will be captured by the EventHandler.js file. And finally, file Algorithms.js will take all these arguments to run the algorithm for graph depending on the algorithm user choose in the screen.

File merge.php is used to linked all the other files to make a complete program. And I use UniController to deploy the program on local host web.

# 3.3. Main Program

The opening interface of the program is as follows:

Graphical user interface, text, application

Description automatically generated

The program is based on 3 main components:

* Add vertex

Vertex button is where users create nodes just by clicking on the screen.

There are many parameters of node such as shape, size, color ... through Settings.

Besides the user can edit the node name.

Graphical user interface, application

Description automatically generated

* Connect Vertices

When choose 2 different nodes, a pop up will appear:

Graphical user interface, text, application, email

Description automatically generated

The program has inherited the advantage of the websites on it is the ability to set weight and name for the graph. This greatly affects the operation of the algorithm because most algorithms to find the shortest or longest path are based on weights to evaluate.

We are allowed to choose directed or undirected graph.

Similar to node, edges can also be edited through Setting.

Graphical user interface

Description automatically generated Graphical user interface, application

Description automatically generated

Besides the main parts of the program, the Setting is also very important in changing the shape and style of the graph, making it easier for users to visualize the graph in many practical problems.

* Algorithms

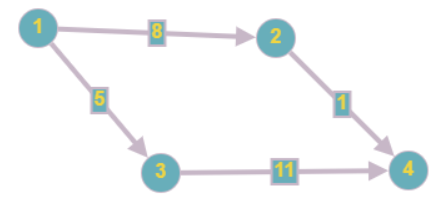
Graphical user interface, text, application, email

Description automatically generated

Algorithm is the most important and complex part of the program. Like most other algorithm illustration programs, users also have a lot of different algorithm options to display on graphs from simple algorithms like BFS, DFS, Dijkstra... to complex ones like Find Maximum flow, Floyd-Warshall...

JavaScripts were used to generate the algorithm's code. Not only because it is often used to code web pages, interact well with HTML, Css; but it is also very fast and powerful in handling algorithmic operations because it is equipped with many modules.

An example with a simple graph follows:

Diagram

Description automatically generated

After using the BFS graph traversal algorithm, the graph will in turn display the vertices being browsed as above. Obviously, since we use the graph traversal algorithm, the weights do not affect, but only the order of generation affects the queue.

We can also view the adjacency matrix through Graph button:

A picture containing square

Description automatically generated

* Program evaluation

Advantages:

* Has a clear, easy-to-use interface
* Inheriting some advantages of the graph algorithm illustration websites as above such as being able to design the graph as you like, illustrating the adjacency matrix, having many algorithms to choose from…
* There are some other improvements such as being able to give weight to each edge, can choose that edge is directed or undirected, after creating the node can still move the position as you want without deleting.

Disadvantages:

* In general, the app is still sketchy and has not changed much compared to other websites. The layout of the buttons is still messy, disjointed, and needs to be more concise.
* Some important parts still do not meet user needs

# 3.4. Future improvements

* If I have more time, the interface design and file organization, code organization will be more taken care of. At the same time, the disadvantages of this web app should also be eliminated.

For example:

Instead of having all the category buttons on the same line, I can separate sections with different functions. The graphic design buttons will be placed separately in a pop up like the visualgo.net website. The main screen will only show the graph that has been created.

* Besides, I will also update many other algorithms. Since algorithms are the most important part, they must be divided into separate categories like Graph Traversal, Find Shortest Path...
* The code that the algorithm is using must be highlighted on the screen so that it is easy for users to follow and visualize
* Added an e-lecture section to provide knowledge about the algorithm, how it works before starting to create the graph

# 4. Advanced application

Currently, algorithms have quite large applications in current software products/systems. It appears not only in navigation software and applications related to the transportation industry (eg google map, grab, uber, fast delivery, ....) but also in networks and telecommunications.

For example, a phone call from a person in Hanoi to a person in Ho Chi Minh City goes through transceivers, internet data from your computer goes to the network provider's server must use algorithm to achieve maximum speed. It is the algorithm to find the shortest path.

Search algorithm you can see it in many current software products, typically Google. You may think, searching is quite simple, when you look in turn at each cell, each line of data to see if there is something you are looking for? But put the status, you have billions of billions of items thrown around in a house, how long will it take you to find the item you want. Know for sure, it is extremely difficult for Google to return the results you requested within a few seconds. That required a very powerful algorithm, and is still in need of improvement to this day.

So this web app is an effective tool to help not only learners but also programmers easily visualize the problem they need to solve, the algorithm to choose for maximum performance.

Suppose in a social network structure as follows:

Diagram

Description automatically generated

If we set Colleague has weight 1

Teacher – 4

Friend – 3

And use a Find Shortest Path algorithm like Dijkstra to label the relationship between 2 people.

Chart

Description automatically generated with medium confidence

Because the shortest length is 4 so Dũng may be Hung’s teacher.

In reality the relationships will be much more complicated but I will just demo you a simple example.