**HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION FACULTY OF INFORMATION TECHONOLOGY**

**\_\_\_\_\_\_\_\*\*\*\*\*\_\_\_\_\_\_\_**



**FINAL TERM PROJECT**

**Coruse name: C2\_S3\_DIPR430685E\_01FIE**

**USING TREE STRUCTURES TO REPRESENT FAMILY TREES AND SEARCH FOR RELATIONSHIPS BETWEEN TWO PEOPLE**

**Lecturer name:** Assoc. Prof. Hoang Van Dung

**List of memebers**

|  |  |  |
| --- | --- | --- |
| **Student ID** | **Student name** | **Contribution (%)** |
| 24110080 | Mai Quốc Bảo | 100% |
| 24110087 | Lê Hải Đăng | 100% |
| 24110120 | Trương Khắc Nhật Phi | 100% |

*Ho Chi Minh City, 09/2025*

**CONTENTS**

[**FIGURE LIST** 1](#_Toc210612001)

[**TABLE LIST** 3](#_Toc210612002)

[**PART 1: INTRODUCTION** 4](#_Toc210612003)

[**1.1. Reason for choosing the topic** 4](#_Toc210612004)

[**1.2. Problem statement** 5](#_Toc210612005)

[**1.3. Aims and Implementation Requirements** 5](#_Toc210612006)

[**1.4. Scope and Subjects** 6](#_Toc210612007)

[**PART 2: THEORETICAL BASIS FOR PROJECT IMPLEMENTATION** 6](#_Toc210612008)

[**2.1. Programing Language** 6](#_Toc210612009)

[**2.1.1. Introduction to C++** 7](#_Toc210612010)

[**2.1.2. C++ Characteristics** 7](#_Toc210612011)

[**2.2. Coding Environment** 8](#_Toc210612012)

[**2.4. Additional Project Management Tool** 9](#_Toc210612013)

[**2.5. Basic Graph Knowledge** 9](#_Toc210612014)

[**2.5.1. Depth-First Search (DFS)** 9](#_Toc210612015)

[**2.5.2. Breadth-First Search (BFS)** 12](#_Toc210612016)

[**2.5.3. Lowest Common Ancestor (LCA)** 14](#_Toc210612017)

[**2.5.4. Optimization LCA With Binary Lifting (Binary Jumping)** 17](#_Toc210612018)

[**PART 3: ANALYSIS AND SOLUTION DESIGN** 20](#_Toc210612019)

[**3.1. User Interface Design** 20](#_Toc210612020)

[**3.2. Tree Graph Drawing And Relationship Finding Functions Design** 28](#_Toc210612021)

[**PART 4: EXECUTION GUIDE, EXPERIMENTS AND RESULTS ANALYSIS** 47](#_Toc210612022)

[**4.1. Program Execution Guide** 47](#_Toc210612023)

[**4.2. Experiments And Results Analysis** 47](#_Toc210612024)

[**4.2.1. Experiments** 47](#_Toc210612025)

[**4.2.2. Results Analysis** 50](#_Toc210612026)

[**PART 5: CONCLUSIONS** 51](#_Toc210612027)

[**5.1 Results Achieved** 51](#_Toc210612028)

[**5.2 Development Orientation** 51](#_Toc210612029)

[**WORK ALLOCATION TABLE** 52](#_Toc210612030)

[**REFERENCES** 53](#_Toc210612031)

# **FIGURE LIST**

[Figure 1: isocpp.org, the C++ official website 6](#_Toc210611771)

[Figure 2: Visual Studio’s interface 7](#_Toc210611772)

[Figure 3. Github repository management 8](#_Toc210611773)

[Figure 4: DFS Implementation using Recursion 10](#_Toc210611774)

[Figure 5: DFS Implementation using Stack 11](#_Toc210611775)

[Figure 6: BFS Implementation using Queue 13](#_Toc210611776)

[Figure 7: LCA Implementation using Naïve Approach 16](#_Toc210611777)

[Figure 8: LCA Implementation using Binary Lifting 18](#_Toc210611778)

[Figure 9: User Interface 19](#_Toc210611779)

[Figure 10: OnSubmit() Function 22](#_Toc210611780)

[Figure 11: OnFindButton() Function 24](#_Toc210611781)

[Figure 12: MyFrame() Function 25](#_Toc210611782)

[Figure 13: OnLoadFile() Function 26](#_Toc210611783)

[Figure 14: Struct Vertex and Edge 27](#_Toc210611784)

[Figure 15: GetVertexIndex() Function 28](#_Toc210611785)

[Figure 16: AddVertex() Function 29](#_Toc210611786)

[Figure 17: AddEdge() Function 30](#_Toc210611787)

[Figure 18: AssignTreeID() Function 31](#_Toc210611788)

[Figure 19: CountComponents() Function 32](#_Toc210611789)

[Figure 20: First Part of ArrangeForest() Function 33](#_Toc210611790)

[Figure 21: Second Part of ArrangeForest() Function 34](#_Toc210611791)

[Figure 22: Third Part of ArrangeForest() Function 35](#_Toc210611792)

[Figure 23: Draw() Function 38](#_Toc210611793)

[Figure 24: IsTree() Function 40](#_Toc210611794)

[Figure 25: FindPathBFS() Function 43](#_Toc210611795)

[Figure 26: BuildLCAForest() Function 44](#_Toc210611796)

[Figure 26: Open .txt FILE 46](#_Toc210611797)

[Figure 27: First testcase 47](#_Toc210611798)

[Figure 28: Second testcase 47](#_Toc210611799)

[Figure 29: Third Testcase 48](#_Toc210611800)

# **TABLE LIST**

[Table 1: UI Objects 21](#_Toc210611881)

[Table 2: Variables For Tree and Lowest Common Ancestor 28](#_Toc210611882)

[Table 3: Variable for Trees Drawing 34](#_Toc210611883)

[Table 4: Result Table 49](#_Toc210611884)

# 

# **PART 1: INTRODUCTION**

## **1.1. Reason for choosing the topic**

Our group selected the topic “Using tree structures to represent family trees and search for relationships between two people” because it effectively bridges theoretical concepts with practical applications. A family tree serves as a clear example of a hierarchical data structure, where parent–child relationships can be visually and accurately represented using tree structures in graph theory. By exploring this subject, we can apply concepts such as nodes, edges, parent–child and ancestor–descendant relationships, along with tree traversal algorithms, to a problem that is familiar in everyday life.

Moreover, analyzing and identifying relationships between individuals in a family tree not only strengthens our understanding of discrete mathematics and graph theory but also highlights the practical value of abstract models in addressing real-world challenges like data organization and relationship queries. This topic also provides our group with an opportunity to sharpen algorithmic thinking, enhance programming skills, and strengthen the connection between theory and practice.

## **1.2. Problem statement**

Given a dataset describing family relationships (parent–child pairs), construct a tree-based representation of the family (or a forest if there are multiple roots). Then, for any two queried individuals A and B, determine whether they are related. If they are, identify the type of relationship as well as the shortest connection path between them.

***Input Data***

* A list of pairs (parent, child) (represented by names or IDs), where each pair denotes a direct parent → child relationship.
* A list of queries, where each query is a pair (A, B) asking to determine the relationship between A and B.

***Output Data***

* Check whether A and B belong to the same lineage.
* If they do, provide the connecting path between A and B.

## **1.3. Aims and Implementation Requirements**

***Aims:***

* To apply concepts of graph theory in representing tree structures.
* To develop a method of determining the relationship between two vertices in a tree.
* To implement an algorithm for finding the Lowest Common Ancestor (LCA) of two particular vertices.
* To strengthen our knowledge in the field of graph theory and its utilization in solving real-world problem.

***Implementation Requirements:***

* Demonstrate the tree structure clearly.
* Allow input and illustration of vertices and edges in the tree.
* Showcase output and clear highlight on the drawn tree.
* Implement an efficient algorithm to determine the relationship between two selected vertices.
* Implement an algorithm to compute the Lowest Common Ancestor (LCA) of two vertices.

## **1.4. Scope and Subjects**

***Scope:***

* The project is limited to tree structures in graph theory, where the graph is connected and acyclic.
* The focus is on determining association between two vertices and computing their Lowest Common Ancestor (LCA).
* Only basic tree operations and LCA are considered.

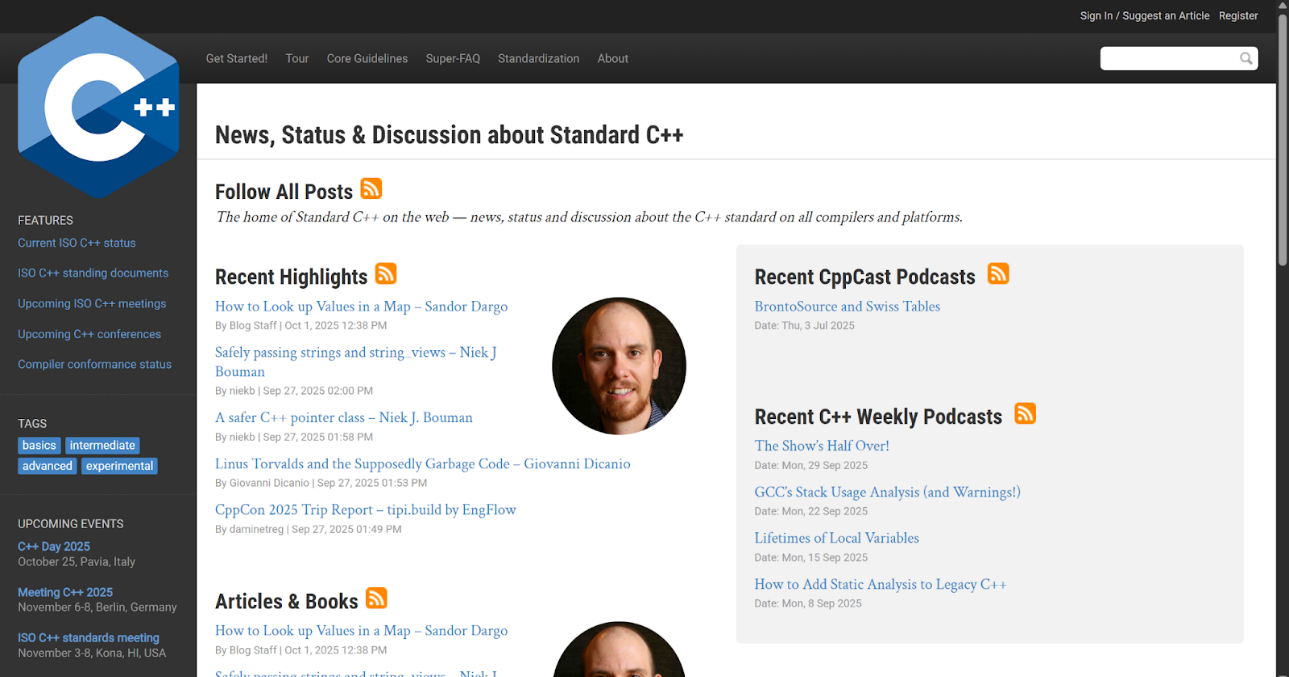
***Subjects:***

* Vertices and edges forming a tree structure.
* The relationship between two particular vertices in the tree.
* Users interacting with the implemented textbox and buttons to submit data and acquire results.

# **PART 2: THEORETICAL BASIS FOR PROJECT IMPLEMENTATION**

## **2.1. Programing Language**

In this project, C++ is chosen as the main programming language since our solid background in this language. Moreover, the language itself offers extreme performance and efficient memory control, making the project easier to develop.



#### Figure 1: [isocpp.org](http://isocpp.org), the C++ official website

### **2.1.1. Introduction to C++**

C++ is a compiled, object-oriented, and strongly typed high-level language developed by Bjarne Stroustrup in the early 1980s as an extension of C. It combines the performance and flexibility of C with support for object-oriented programming.

C++ provides a great amount of features, including classes, inheritance, polymorphism, templates, and exception handling, making it appropriate for both low-level system programming and high-level application establishment. The language also allows modular programming with the help of libraries and packages, encouraging code reusability and maintainability.

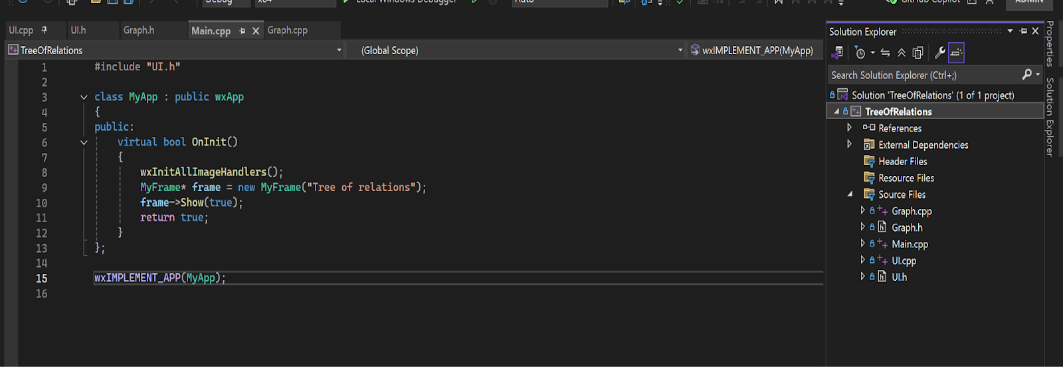
One of the major strengths of C++ is its balance between performance and abstraction. It allows developers to demonstrate efficient programs close to the hardware level while still offering advanced traits of modern programming languages. Therefore, C++ has become one of the most commonly used languages in software engineering, especially in fields such as software engineering, game development and algorithm research.

### **2.1.2. C++ Characteristics**

* ***Compiled Language****:* Enhances performance compared to interpreted languages.
* ***Multi-Paradigm***: Supports procedural, object-oriented, and generic programming.
* *Object-Oriented*: Encapsulation, inheritance, and polymorphism allow modular, reusable code.
* ***Rich Standard Library***: The STL offers ready-to-use data structures and algorithms.
* ***Performance and Control***: Direct memory manipulation with pointers and references.
* ***Popularity and Community***: Widely used with extensive documentation and global support.

## **2.2. Coding Environment**

The coding environment for this project is Visual Studio Code . It provides a user-friendly interface, enables various supportive extensions that enhance coding productivity. This allows the working process to be more effective, easier to debug and better-operated.



#### Figure 2: Visual Studio’s interface

**2.3. Supporting Library**

The main supporting libraries exploited in this project include:

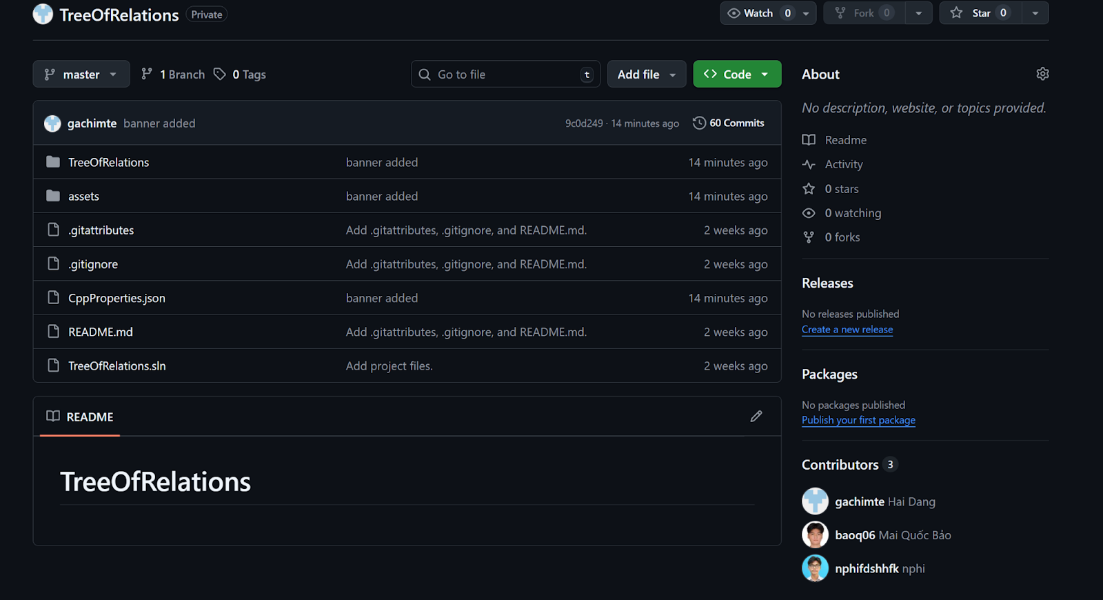
* STL (Standard Template Library): used for main data structures such as vector, queue, map, set, etc. to illustrate adjacency lists, implement graph traversals, and execute the lowest common ancestor.
* wxWidgets: utilized for UI implementation. The library offers tools to build graphical user interfaces, allowing visualization of the tree structure and the blood relationship (or non-blood relationship) between two specific given vertices.

## **2.4. Additional Project Management Tool**

To streamline our workflow and strengthen collaboration, we’ve chosen GitHub as our main platform. Trusted by over 100 million developers and organizations worldwide, GitHub is one of the leading tools for software development and project management. Built on top of the Git version control system, it offers a cloud-based environment where team members can work on the same codebase without interfering with one another. Every change is carefully tracked and documented, which not only ensures code integrity but also allows us to roll back to earlier versions whenever needed.

One of GitHub’s major strengths is its branching and merging system. This feature enables us to experiment freely, test new ideas, and work on different parts of the project in parallel—without affecting the stability of the main codebase.

In addition, GitHub’s built-in features—such as issue tracking, pull requests, and project boards—make collaboration more transparent and organized, giving our team a clear overview of tasks, responsibilities, and overall progress.



#### Figure 3. Github repository management

## **2.5. Basic Graph Knowledge**

### **2.5.1. Depth-First Search (DFS)**

***Algorithm Introduction***

The Depth-First Search (DFS) algorithm is a graph traversal technique used to explore vertices and edges of a graph in a systematic way.

DFS was officially introduced by Pierre Rosenstiehl and Robert Tarjan in the 1970s as part of their foundational work on graph theory and algorithmic techniques. However, the underlying idea of exploring structures deeply before backtracking had already been applied informally in mathematics and logic earlier.

Since then, DFS has become a cornerstone in computer science. It is not just a standalone tool but also the backbone for many advanced techniques, such as topological sorting, identifying strongly connected components, and even solving puzzles or mazes.

***Algorithm Concept***

The primary idea of the Depth-First Search (DFS) algorithm is to explore a graph by moving as deep as possible along one path before backtracking. DFS starts from a given node then visits an unvisited node, then continues to move forward to the next unvisited node, and so on. When no further unvisited neighbors are available, the algorithm backtracks to the most recent node that still has unexplored neighbors and continues the process.

This strategy makes sure that all vertices reachable from the starting node are visited. By going deep first in a systematic way and then backtracking to the other branches, DFS provides a natural way to explore all paths, detect connectivity, and analyze the structure of graphs.

***Algorithm Description***

***Initialization:***

* Mark all vertices as unvisited.
* Choose a starting vertex**.**

***Visit the starting vertex:***

* Mark it as visited.
* Record or process the vertex according to the problem requirements (e.g., print its value, store in a list).

***Recursive exploration:***

* For each neighbor of the current vertex: If the neighbor has not been visited, recursively apply DFS to that neighbor.

***Backtracking:***

* When a vertex has no unvisited neighbors left, backtrack to the previous vertex and continue exploring other neighbors.

***Completion:***

* Repeat until all vertices reachable from the starting vertex have been visited.
* If the graph is disconnected, apply DFS to each unvisited vertex until all vertices are covered.



#### Figure 4: DFS Implementation using Recursion



#### Figure 5: DFS Implementation using Stack

### **2.5.2. Breadth-First Search (BFS)**

***Algorithm Introduction***

The Breadth-First Search (BFS) algorithm is an algorithm used to explore graphs or trees. Unlike DFS, which goes deep along a single branch, BFS works level by level—visiting all neighbors of a node before moving on to the next layer.

The idea behind BFS emerged during early graph theory research in the mid-20th century and gained prominence with the rise of computer science. In the 1950s and 1960s, it was studied extensively, particularly for solving shortest path problems and exploring large networks.

Today, BFS is considered a cornerstone of graph algorithms. It’s widely taught in computer science courses on data structures and algorithms, often presented side by side with DFS as one of the essential traversal strategies every student should know.

***Algorithm Concept***

The main idea of BFS is to explore a graph layer by layer. Starting from a chosen vertex, the algorithm first visits all of its immediate vertices, then moves on to the neighbors of those neighbors, and so on. This makes sure that vertices are visited in order of their distance from the starting vertex.

To achieve this, BFS uses a *queue* data structure:

* Push the starting vertex into the queue and mark it as visited.
* Repeatedly remove a vertex from the front of the queue.
* For each unvisited neighbor of that vertex, mark it as visited and push it into the queue.
* Continue until the queue becomes empty.

This approach guarantees that the algorithm always processes the closest nodes first, which makes BFS very effective for finding the shortest path in an unweighted graph.

***Algorithm Description***

***Initialization:***

* Mark all vertices as unvisited.
* Choose a starting vertex and mark it as visited.
* Enqueue the starting vertex into a queue.

***Level-by-level traversal:***

* While the queue is not empty:
  + Dequeue a vertex from the front of the queue.
  + Process the vertex as required (e.g., print, store, or analyze).
  + For each neighbor of the current vertex: If the neighbor is unvisited, mark it as visited and enqueue it.

***Completion:***

* Continue the process until the queue is empty.
* If the graph is disconnected, repeat BFS starting from each unvisited vertex until all vertices are covered.

BFS ensures that vertices are visited in increasing order of distance (number of edges) from the starting vertex. Its implementation usually uses a queue to maintain the order of exploration, which allows for a systematic, level-by-level traversal.



#### Figure 6: BFS Implementation using Queue

### **2.5.3. Lowest Common Ancestor (LCA)**

***Algorithm Introduction***

The Lowest Common Ancestor (LCA) problem is a classic challenge in graph theory and computer science, particularly when working with rooted trees. The idea is simple: given two nodes in a tree, the LCA is the deepest (or lowest) node that is an ancestor of both.

The concept traces back to early studies of hierarchical structures—such as genealogy trees—where finding common ancestors was an essential task. With the rise of computer science in the late 20th century, researchers developed efficient algorithms to solve LCA queries, especially in applications involving large trees. These breakthroughs made LCA methods highly valuable in areas like computational biology, database systems, and network routing.

Today, LCA algorithms are widely used in many domains, including:

* *Hierarchical data processing*: such as file systems or XML/HTML document trees.
* *Network optimization*: improving routing and shortest path computations.
* *Tree analysis in biology and genealogy***:** studying evolutionary and family trees.

***Algorithm Concept***

One of the most straightforward ways to find the Lowest Common Ancestor (LCA) of two nodes in a rooted tree is to rely directly on the definition of an ancestor. The process works in three main steps:

* *Find the depth of each node*: The depth is the number of edges from the root to that node.
* *Equalize the depths*: If one node is deeper, move it upward (toward the root) until both nodes are at the same depth.
* *Move both nodes upward at the same time*: From this point, move both nodes upward step by step until they meet. The node where they meet is the LCA.

This approach is easy for people to understand because it mirrors how we think about ancestry: bring both nodes to the same level, then trace their parent chains until they meet.

***Algorithm Description***

***Preprocessing:***

* Compute the depth of every node in the tree (typically using DFS or BFS from the root).
* Store the parent of each node for quick upward traversal.

***Equalize depths:***

* Given two nodes *u* and *v*, compare their depths.
* If one node is deeper, move it upward (toward the root) until both nodes are at the same depth.

***Move upward simultaneously:***

* Once both nodes are at the same level, move them upward together, one step at a time.
* Continue this process until *u* and *v* point to the same node.

***Find the LCA:***

* When u and v meet, that node is their Lowest Common Ancestor.



#### Figure 7: LCA Implementation using Naïve Approach

### **2.5.4. Optimization LCA With Binary Lifting (Binary Jumping)**

While the naïve LCA algorithm is easy to understand, it becomes inefficient for tall trees or when we need to answer many queries—since each query could take time proportional to the tree’s height.

To address this, a powerful optimization known as Binary Lifting (or Binary Jumping) is used. The central idea is to preprocess the tree so that each node knows its ancestors at distances of powers of two: 1, 2, 4, 8, and so on. In other words, instead of moving one step at a time, a node can “jump” directly to its 2^k-th ancestor.

This preprocessing allows us to bring two nodes to the same depth and then lift them together much more efficiently. Instead of climbing step by step, we move them upward in logarithmic jumps until they converge.

As a result, Binary Lifting reduces the query time to O(log N) after an O(N log N) preprocessing phase, making it highly effective for handling multiple LCA queries on large trees.

***Preprocessing****:*

* Perform a DFS (or BFS) from the root to compute the depth of each node.
* For each node *u*, precompute its *2^k-th* ancestor for all k such that *2^k ≤ log₂N.* For example:
  + *up[u][0] = parent of u*
  + *up[u][k] = up[ up[u][k-1] ][k-1]*
* This preprocessing step takes *O(N log N)* time.

***Equalizing depths:***

* Given two nodes u and v, if their depths are different, lift the deeper node upward.
* Instead of moving step by step, use the binary representation of the depth difference to jump in powers of two.

***Binary jumping to find LCA****:*

* If u and v are already the same, then that node is the LCA.
* Otherwise, starting from the highest power of two down to 0:
  + If *up[u][k] ≠ up[v][k]*, then move both u and v up to *up[u][k]* and *up[v][k]*.
* After this process, *u* and *v* will be direct children of the LCA.

***Return the parent****:*

* The parent of either *u* or *v* at this point is the Lowest Common Ancestor.

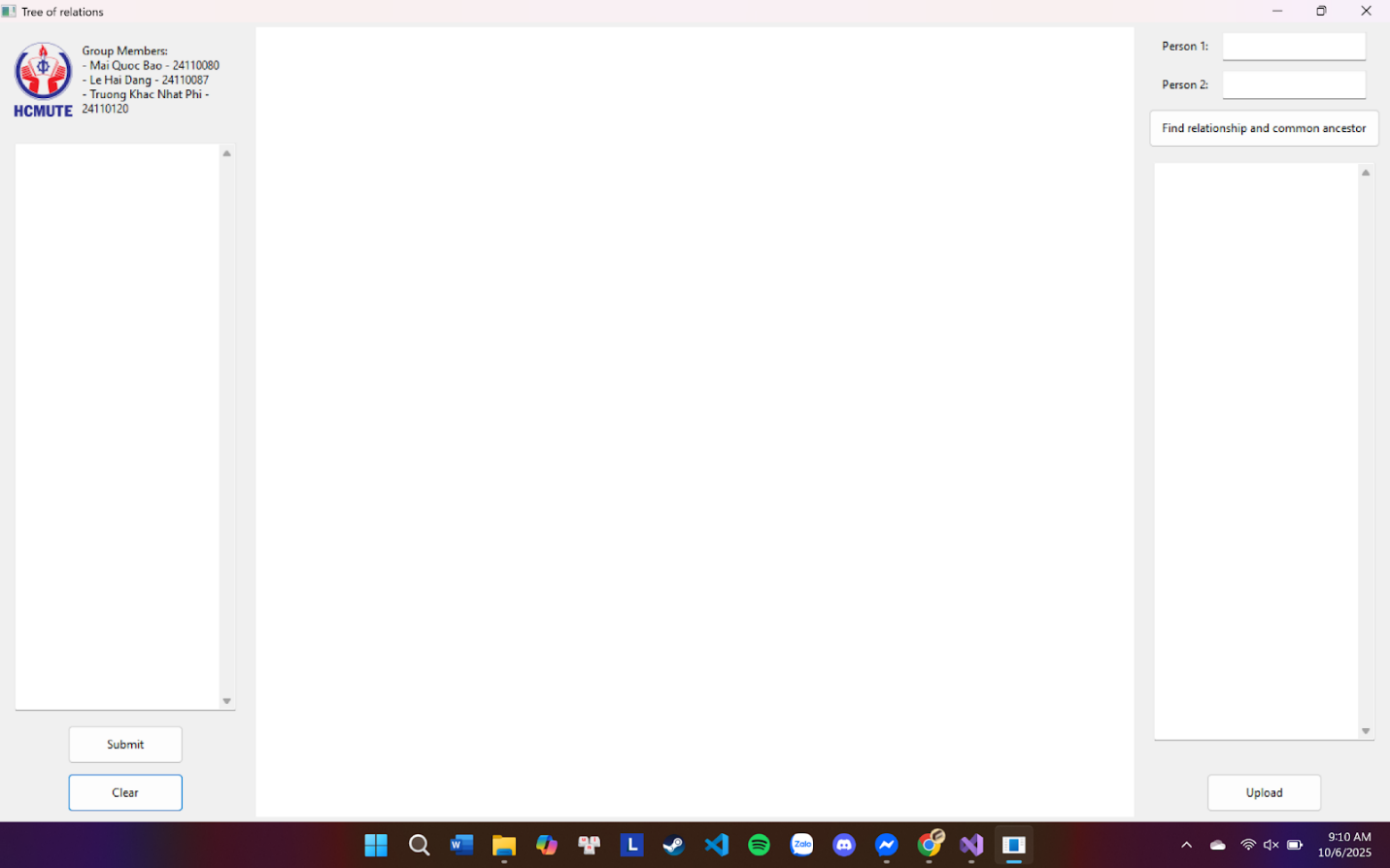


#### Figure 8: LCA Implementation using Binary Lifting

# **PART 3: ANALYSIS AND SOLUTION DESIGN**

## **3.1. User Interface Design**

When executed (in the Visual Studio IDE) or opened (open the application itself), the program will display an interface like below:



#### Figure 9: User Interface

In order to construct an interface like so, we first declared multiple objects with the help of the wxWidgets library. The objects used are listed in the table below:

|  |  |  |
| --- | --- | --- |
| **Numbering** | **Name of the objects** | **Purpose** |
| 1 | wxPanel\* main\_panel | The big main panel. It contains the whole layout. |
| 2 | wxPanel\* left\_panel | The left panel. It contains the banner (school’s logo and members’ information), input textbox and the two buttons “Submit” and “Clear”. |
| 3 | wxPanel\* right\_panel | The right panel. It is split into half, top right and bottom right panel. |
| 4 | wxPanel\* banner | The panel inside the left panel, created to store the logo and members’ information. |
| 5 | wxPanel\* top\_right\_panel | The top panel inside the right panel, created to store the two input textboxes and the “Find” button. |
| 6 | wxPanel\* bottom\_right\_panel | The bottom panel inside the right panel, created to store the output textbox and the “Load” button. |
| 7 | wxBoxSizer\* mainbox | The main horizontal sizer of the whole layout. It contains three sections: the left panel, the canvas, and the right panel. |
| 8 | wxBoxSizer\* left\_sizer | The vertical sizer of the left panel. |
| 9 | wxBoxSizer\* right\_sizer | The vertical sizer of the right panel. |
| 10 | wxBoxSizer\* bsizer | The horizontal sizer of the banner. |
| 11 | wxBoxSizer\* logo\_sizer | The vertical sizer of the logo. |
| 12 | wxBoxSizer\* top\_sizer | The vertical sizer of the top right panel. |
| 13 | wxBoxSizer\* bottom\_sizer | The vertical sizer of the bottom right panel. |
| 14 | wxBoxSizer\* rowA | The horizontal sizer for label1 and textbox1. |
| 15 | wxBoxSizer\* rowB | The horizontal sizer for label2 and textbox2. |
| 16 | wxTextCtrl\* left\_textbox | The input textbox in the left panel. |
| 17 | wxTextCtrl\* top\_right\_textbox1 | The input textbox on rowA. |
| 18 | wxTextCtrl\* top\_right\_textbox2 | The input textbox on rowB. |
| 19 | wxTextCtrl\* right\_textbox | The output textbox in the bottom right panel. |
| 20 | wxImage ilogo | The loaded logo picture. |
| 21 | wxString members | The text string consists of the members’ information. |
| 22 | wxStaticText\* membername | The text displays members. |
| 23 | wxStaticText\* label1 | The “A” label in the top right panel. |
| 24 | wxStaticText\* label2 | The “B” label in the top right panel. |
| 25 | MyCanvas\* canvas | The canvas used to display family trees. |
| 26 | wxButton\* submit\_button | The button used for drawing trees from the input. |
| 27 | wxButton\* clear\_button | The button used for clearing inputs, outputs and the canvas. |
| 28 | wxButton\* find\_button | The button used for displaying the relationship between 2 vertices and their lowest common ancestor. |
| 29 | wxButton\* upload | The button used for uploading existing adjacency lists from devices (.txt files only). |

##### *Table 1: UI Objects*

After creating a handful of objects, we start constructing the layout for the interface. These objects are declared in class MyFrame and implemented in the source file UI.cpp. The design of the user interface is divided into three main parts, which are the three panels. In addition, the sizers are declared to resize the component objects (textboxes, labels, buttons, etc.) to fit the area of the panel automatically in horizontal or vertical directions. We also create functions for the buttons to operate smoothly.



#### Figure 10: OnSubmit() Function

The function *OnSubmit()* is used for the “Submit” button. When called it retrieves the text value given inside the input textbox, then clears the canvas completely (to avoid overdrawn incident). If the textbox is empty, the function terminates immediately. Otherwise, it will split the input into lines, each line will have two tokens (if there are more than two tokens then an error message will pop up). The tokens are then added into the canvas using *canvas->AddEdge*. The function then calls *CountComponents()* to count the connected components in the forest (if there are more than 2 trees then an error message will pop up and the canvas will be cleared). Finally, it calls the *ArrangeForest()* to arrange the trees.





#### Figure 11: OnFindButton() Function

The function *OnButtonFind()* is used to determine the relationship between two selected individuals. First, the function retrieves the data of the two given individuals. It then checks their relationship by determining whether they belong to the same family tree. If they do, the function examines their relationship through their lowest common ancestor (LCA) to accurately identify the specific familial connection.



#### Figure 12: MyFrame() Function

The function *OnClear()* is created for the sake of clearing inputs and outputs (called in the “Clear” button). It simply clears the left textbox, the two textboxes in the top right panel, the canvas and the output textbox.



#### Figure 13: OnLoadFile() Function

The function *OnLoadFile()* is accounted for reading the existing adjacency list from devices (used for the “Upload” button). It generates an open file dialog with a filter that allows users to open existing .txt files only, if users click “Cancel”, the function terminates. After that it opens the file as a binary data stream (if it can not be opened then an error message will pop up). Next, it will generate a text input stream to read text files with UTF-8 encoding, and read until the end of the file, then add everything into the “content” string. It then displays the file’s content on the left textbox, clear the canvas (to avoid overdrawn incidents) and draw the trees (same method used for the *OnSubmit()* function).

## **3.2. Tree Graph Drawing And Relationship Finding Functions Design**

We have defined two structs to simplify the implementation and improve code readability.



#### Figure 14: Struct Vertex and Edge

Struct *Vertex* is used to define a data type for storing a vertex, which includes the following information:

* ***wxPoint pos*** is used to store the coordinates of the vertex on the screen.
* ***int r***is used to store the radius of the circle representing that vertex.
* ***wxString******label*** represents the name of the vertex.

Struct *Edge* defines a custom data type designed to represent an edge, containing information about two vertices, specifically their indices within the map *vertexMap*.

Before implementing the functions, it is necessary to declare the variables that will be used in the program.

|  |  |  |
| --- | --- | --- |
| **Numbering** | **Name of the variables** | **Purpose** |
| 1 | vector<Vertex> vertices | List containing all the vertices in the tree (or forest). |
| 2 | vector<Edge> edges | List containing all the edges in the tree (or forest). |
| 3 | map<wxString, int> vertexMap | Map containing the ID number of each vertex |
| 4 | vector<int> treeID; | List containing the ID of each |
| 5 | vector<int> highlightPath; | List containing the vertices that is used to highlight on the canvas |
| 6 | vector<pair<int, int>> highlightEdges; | List containing the edges that is used to highlight on the canvas |
| 7 | vector<vector<int>> up; | The two-dimensional array that stores the ancestors of each vertex at powers of two. |
| 8 | int highlightLCA | Vertex which is the lowest common ancestor |

##### *Table 2: Variables For Tree and Lowest Common Ancestor*

The following functions are responsible for drawing the tree graph and finding the relationship between the two selected vertices.



#### Figure 15: GetVertexIndex() Function

The function *GetVertexIndex()* is used to check whether a vertex named *label* exists in the map *vertexMap*. If it exists, the function returns the index assigned to that vertex; otherwise, it returns -1.



#### Figure 16: AddVertex() Function

The function *AddVertex()* is responsible for adding a new vertex to the list *vertices*, and then inserting it into the map *vertexMap* by assigning it a new index.



#### Figure 17: AddEdge() Function

The function *AddEdge()* is responsible for adding a new edge to the list *edges*. This function performs this task by generating data for two vertices, which represent the two endpoints of the edge.

Initially, the function checks whether each vertex already exists in the map *vertexMap*.

* If a vertex is already present, the function simply assigns its existing information from the map *vertexMap*.
* If a vertex does not exist, the function creates a new information for it by:
  + Randomly assigning coordinates to determine the position *pos* of the vertex on the screen.
  + Setting the vertex radius to a default value of 20 (a value selected by group).

Once the vertex information has been fully defined, the function proceeds to add the vertex to the list *edges* and assigns it a new index number in the map vertexMap to maintain consistent reference and identification within the data structure.



#### Figure 18: AssignTreeID() Function

The function *AssignTreeID()* is used to distinguish vertices that belong to different trees. First, the function constructs an adjacency list using the data from the map list *edges*. Then, it applies the *BFS* algorithm to assign a unique identifier to *treeID* for each vertex within the same tree.



#### Figure 19: CountComponents() Function

Function *CountComponents()* is used to count the number of connected components (i.e., the number of trees) in the graph. The function works by performing a BFS starting from an unvisited vertex, and each time such a traversal occurs, it increases the variable *comps* by 1. After all vertices have been visited, the function returns the value of *comps*, which represents the total number of trees in the graph.



#### Figure 20: First Part of ArrangeForest() Function

The function *ArrangeForest()* is used to arrange the vertices of the trees on the screen in a way that is visually appealing, well-organized, and prevents them from overlapping.

First, we create an adjacency list using the information from the list *edges*. Then, we then define the following variables and constants:

`

|  |  |  |
| --- | --- | --- |
| **Numbering** | **Name of the variables** | **Purpose** |
| 1 | int numTree | Storing the number of trees (connected components) in the forest |
| 2 | int levelHeight | Define the spacing between level of nodes in a tree |
| 3 | int baseNodeSpacing | Define the horizontal spacing between nodes of the same level |
| 4 | int baseTreeGap | Define the horizontal spacing between adjacent tree |
| 5 | int offSetX | Horizontal starting position for arranging each tree |
| 6 | int treeCount | Determine how many trees have already been arranged |

##### *Table 3: Variable for Trees Drawing*



#### Figure 21: Second Part of ArrangeForest() Function

The function processes each tree one by one. The code above uses BFS to retrieve the vertices that belong to the current tree and assigns a value of 1 to the array *incomp*. Vertices that do not belong to the current tree will have the array *incomp* set to 0.



#### Figure 22: Third Part of ArrangeForest() Function

After that, the function selects the first vertex from the list *comp* to serve as the root. The function *dfsArrragge()* uses recursion and operates as follows:

* Find the child nodes of the current node and add them to the list *children*.



* If the list *children* is empty, it means the current vertex is a leaf node, and we assign coordinates to it. Otherwise, we recursively traverse the children of that vertex to accurately determine the coordinates of the current vertex.





After completing the function, we call it to arrange the current tree. Then, we prepare to draw the next tree.



After all the trees have been drawn, we adjust their coordinates so that the trees are centered on the screen.





#### Figure 23: Draw() Function

The *Draw()* function is used to render the tree on the screen. It follows these rules:

* Regular edges are drawn in black.
* Edges that lie on the path between two selected vertices are drawn in red.
* The LCA vertex is drawn in green.
* Vertices on the path between the two selected vertices are drawn in red.
* All other vertices are drawn in black.



#### Figure 24: IsTree() Function

The function *IsTree()* checks whether a graph is a tree. First, it constructs an adjacency list from the information in the list *edges*. Then, it uses BFS to perform the check.

Here, the queue is implemented using a *pair*, where the first element is the current vertex and the second element is the previous vertex. During the BFS traversal, if the next vertex to be visited is the same as the previous vertex, it indicates a cycle. In that case, the function returns that the graph is not a tree.

If no cycle is found but the BFS does not visit all vertices, the function also returns that the graph is not a tree. If both conditions are satisfied—no cycles and all vertices are visited—then the graph is confirmed to be a tree.





#### Figure 25: FindPathBFS() Function

The function *FindPathBFS()* is used to find the path between two selected vertices and store the information in two lists, *highlightPath* and *highlightEdges*. The function executes BFS and uses the trace list to keep track of the visited vertices. After BFS completes, the path is reconstructed by backtracking through the trace list.



#### Figure 26: BuildLCAForest() Function

The function *BuildLCAForest()* is a simple preprocessing for the LCA algorithm on the trees in the forest.

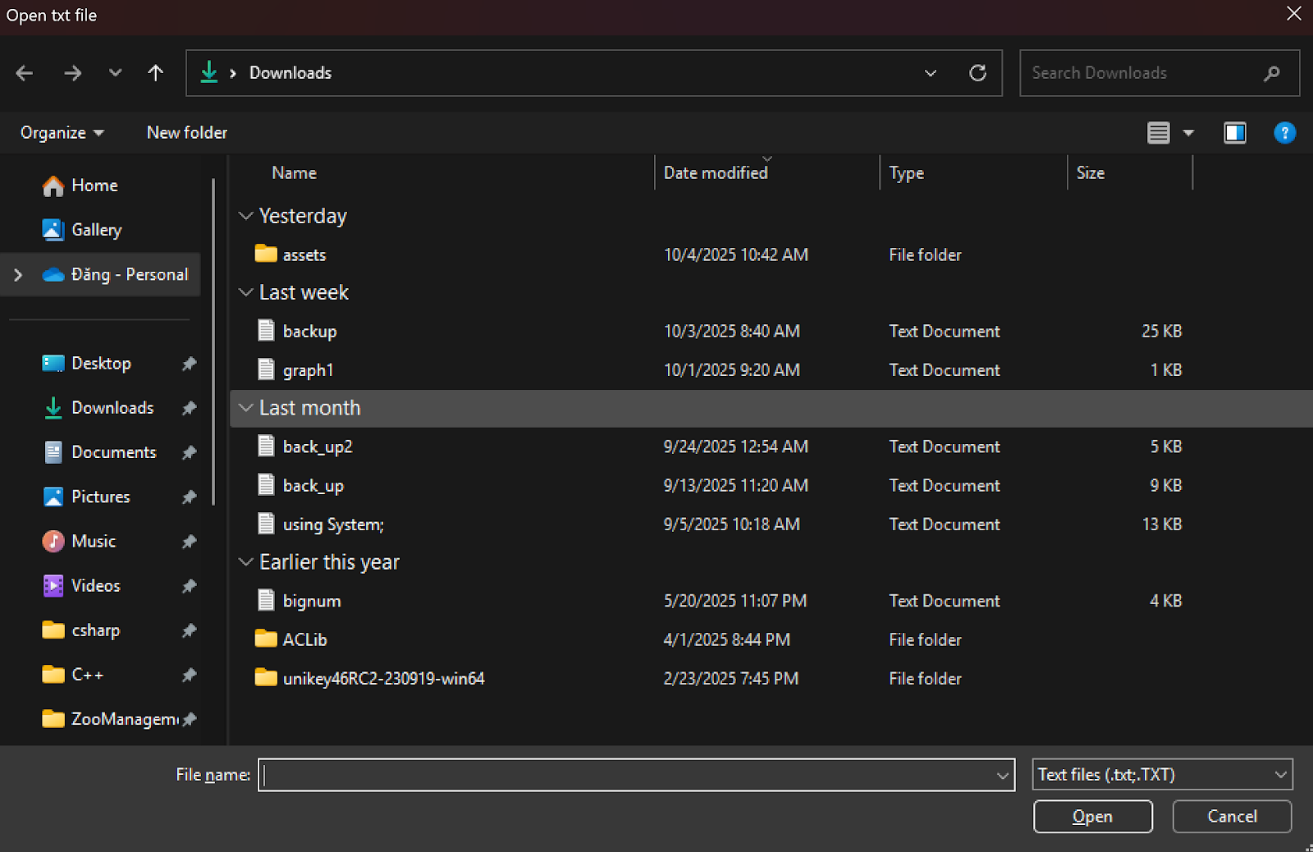


The function *FindLCA*() is simply used to find the lowest common ancestor (LCA) of two selected vertices.

# **PART 4: EXECUTION GUIDE, EXPERIMENTS AND RESULTS ANALYSIS**

## **4.1. Program Execution Guide**

***Step 1*:** Type the input manually (two vertices per line only), then click “Submit” to draw the trees. Or you can choose to click “Upload” and read files from your own device (the program will automatically draw the trees if you choose this option).



#### Figure 26: Open .txt FILE

**Step 2:** Type the two vertices you want to clarify relationship and their lowest common ancestor into the two textboxes (A and B), and then click “Find” to see the visualization on the canvas, and the outputs on the right textbox.

**Step 3:** You can click “Clear” to clear everything and type another case (or choose another adjacency list). This step is completely optional.

## **4.2. Experiments And Results Analysis**

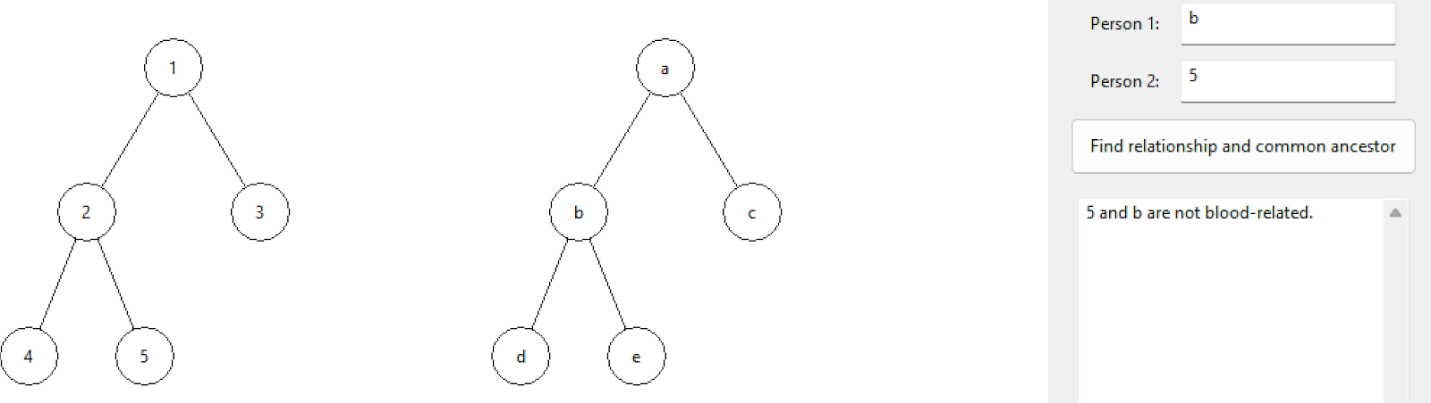
### **4.2.1. Experiments**

**First case:** 

#### Figure 27: First testcase

The nodes selected are “Trung” and “Vuong”. When clicked, the button will highlight the trees and showcase the results as displayed. Their lowest common ancestor is node “Bao”, highlighted in green. The red lines are drawn for the purpose of illustrating the two vertices' blood relationship. As shown on the canvas, “Vuong” is two generations younger than “Vuong”, and is the son of “Phi”, thus, their relationship is granduncle and grandnephew.

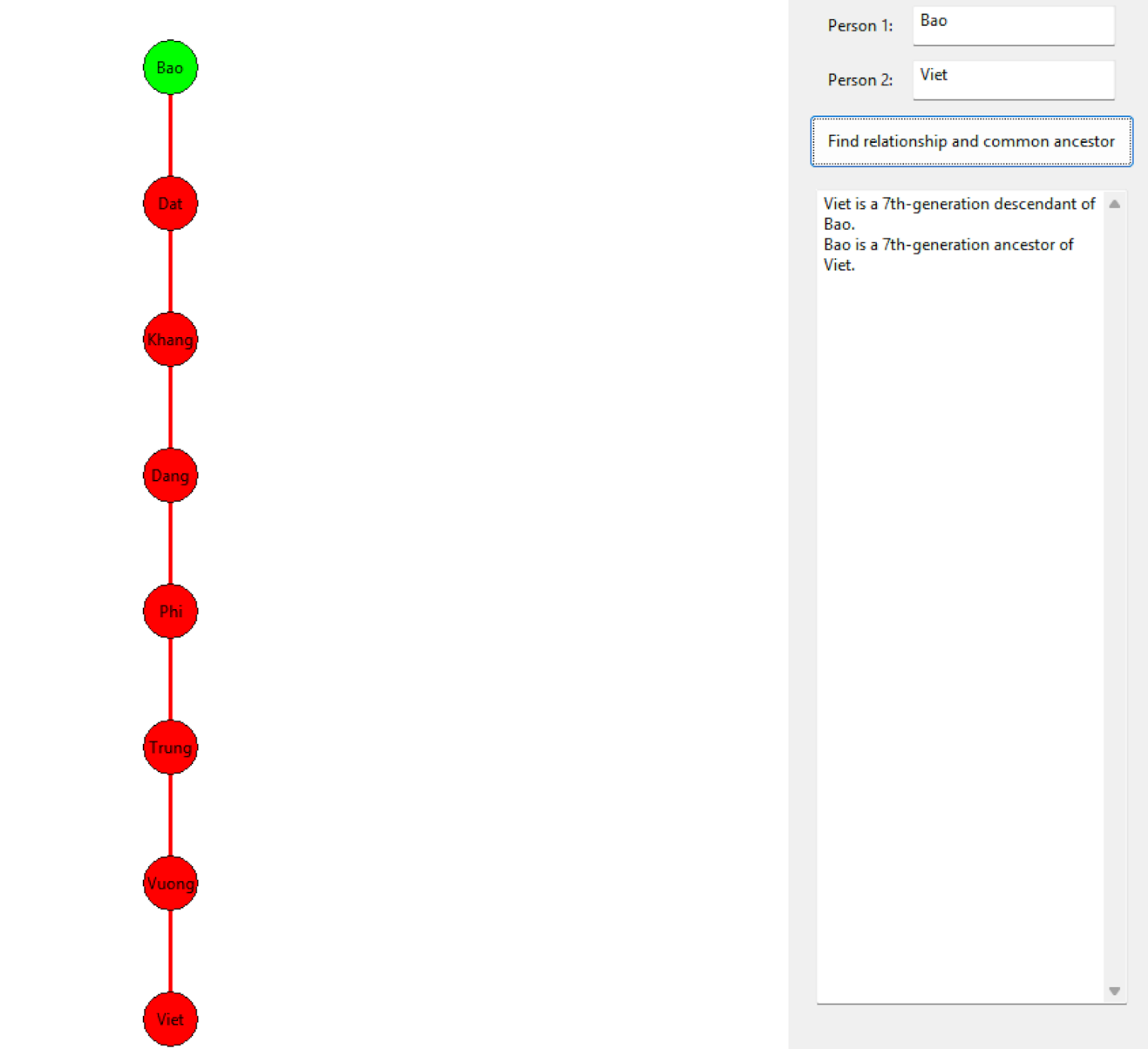
**Second case:**



#### Figure 28: Second testcase

The nodes selected are “b” and “5”. When clicked, the button will not highlight anything since those two vertices are not blood-related. Their lowest common ancestor does not exist. There are also no red lines. As a result, their relationship is “not blood-related”.

**Third case:**

****

#### Figure 29: Third Testcase

The nodes selected are “Bao” and “Viet”. When clicked, the button will highlight the trees and showcase the results as displayed. Their lowest common ancestor is “Bao” himself, highlighted in green. The red lines are drawn for the purpose of illustrating the two vertices' blood relationship. As shown on the canvas, “Viet” is seven generations younger than “Bao”, which is very far from each other. Thus, their relationship is ancestor and descendant.

### **4.2.2. Results Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Numbering** | **Selected vertices** | **Blood relationship** | **Lowest Common Ancestor** |
| 1 | Trung - Vuong | Granduncle - Grandnephew | Bao |
| 2 | b – 5 | Not blood-related | Does not exist |
| 3 | Bao - Viet | Ancestor - Descendant | Bao |

##### *Table 4: Result Table*

# **PART 5: CONCLUSIONS**

## **5.1 Results Achieved**

In general**,** the program has successfully met most of the proposed requirements. The group implemented core functionalities including adding vertices and edges, automatically arranging the forest, checking whether the graph is tree, finding paths between two vertices, and determining the Lowest Common Ancestor (LCA). In addition, the program provides a visual interface that enables users to easily observe and interact with the tree structure.

## **5.2 Development Orientation**

* Optimize the forest arrangement algorithm to improve performance when handling a large number of vertices.
* Integrate additional algorithms such as DFS and other advanced methods to enhance functionality.
* Extend visualization features, for example, supporting weighted graphs or animating traversal processes.
* Redesign the user interface to be more intuitive and user-friendly.
* Refactor and optimize the source code for better maintainability and scalability.

# **WORK ALLOCATION TABLE**

|  |  |  |
| --- | --- | --- |
| **List of members** | **The tasks accomplished** | **Completion Rate** |
| Mai Quốc Bảo | Responsible for researching and implementing the core algorithms of the project, developing the relationship-finding functions, ensuring accuracy and optimization. Also responsible for writing the detailed report. | 100% |
| Lê Hải Đăng | In charge of designing the user interface, setting up the work environment, implementing supporting libraries and managing the Github repository. Additionally, assists with report writing and Canva preparation. | 100% |
| Trương Khắc Nhật Phi | Tasked with supporting, making adjustments to user interface, developing the tree-drawing function, taking the lead in preparing the Canva (content and presentation), and assisting in report writing. | 100% |

# **REFERENCES**

[1]. <https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/>, introduction to the BFS algorithm.

[2]. <https://www.geeksforgeeks.org/dsa/depth-first-search-or-dfs-for-a-graph/>, introduction to the DFS algorithm.

[3]. <https://wiki.vnoi.info/algo/data-structures/lca-binlift.md>, study on the Lowest Common Ancestor (LCA) algorithm in genealogical trees.

[4]. [https://www.youtube.com/playlist?list=PLFk1\_lkqT8MbVOcwEppCPfjG](https://www.youtube.com/playlist?list=PLFk1_lkqT8MbVOcwEppCPfjGOGhLvcf9G), instructions for installing wxWidgets.

[5]. Course material, Hoang Van Dung, Discrete mathematics and Graph theory.