# Comp151 Lab14b

Lab14b consists of **three independent graph applications:**

* WhichRoadsCanBeClosed
* StrongWeakDisjoint
* HowManyQuestions

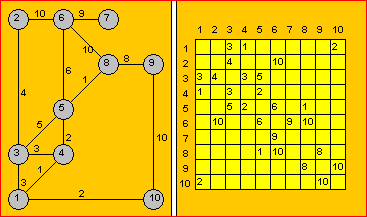
# WhichRoadsCanBeClosed

Public Works Department in your county asked for your help in deciding which roads can be closed for repairs such that every town in the county can be reached and the total road mileage is minimized. All the roads in the county are two-way roads.

Solve the problem by applying the following *Prim’s Algorithm* for finding a **Minimum Spanning Tree** (MST) embedded in a **weighted connected undirected** graph. It finds a tree that is made up of all the nodes in the graph and a subset of the edges such that the sum of the edge weights is a minimum.

The algorithm is stated formally as follows: <http://en.wikipedia.org/wiki/Prim%27s_algorithm>

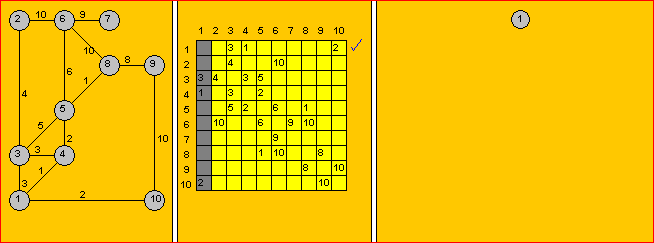
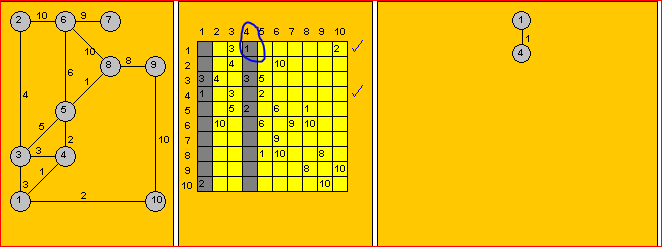
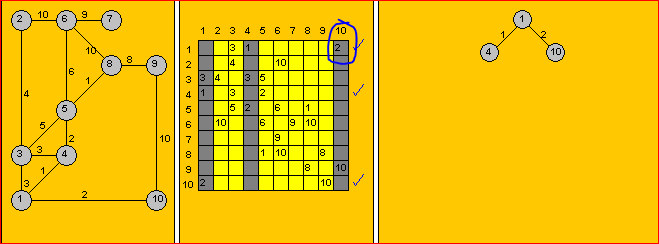
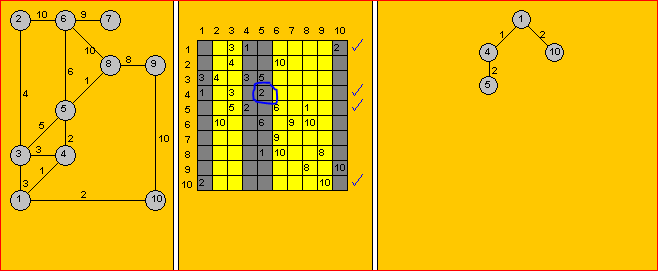
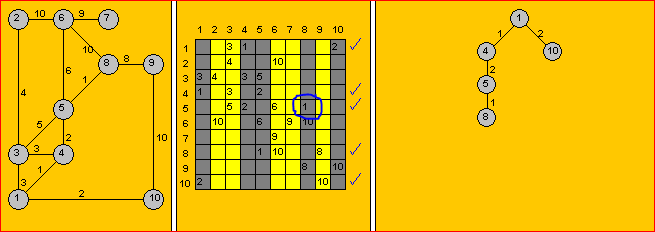
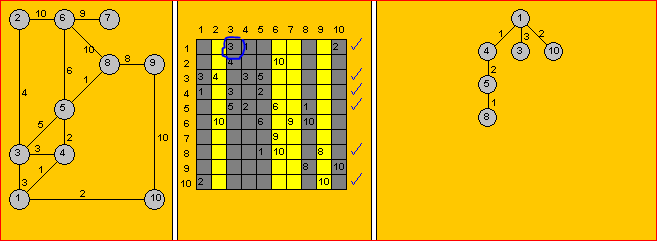
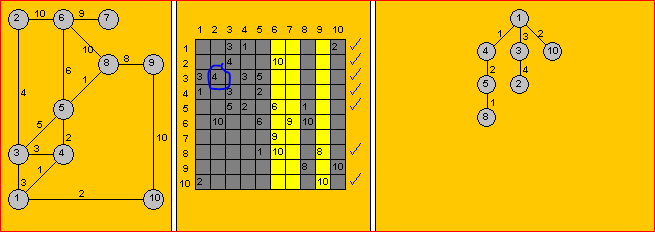
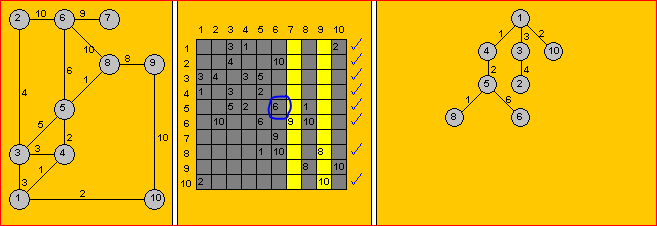
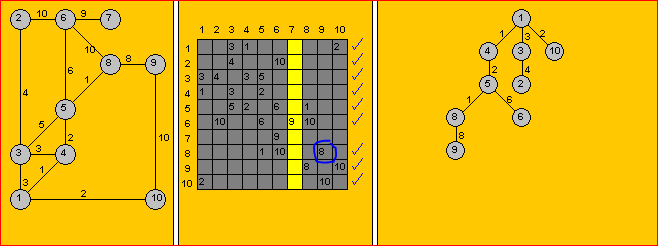
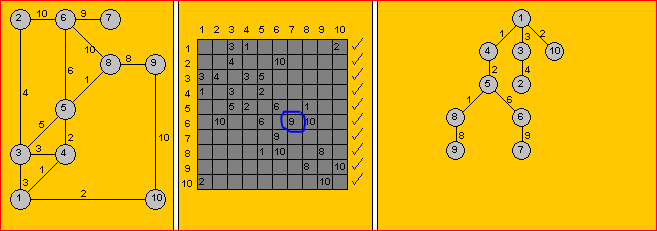
WhichRoadsCanBeClosed class uses the **adjacency matrix** to represent the graph. This array is specified by the costs. main asks the user for the number of nodes and the probability of edge existing between two vertices. Constructor is to generate weights and edges randomly. Since we will not use directed graphs, this array will be symmetrical:



The *Prim’s Algorithm* steps are as follow:

1. *We put a mark beside the first row and we gray-out the first column*
2. *Between the elements that are not grayed-out and belong to a row with a mark we choose the least A(j,k). If all elements are grayed-out, the algorithm terminates.*
3. *We put mark beside the kth row and we gray-out the kth column. We return to step b.*

NOTE that in every step *Vk* node becomes *Vj*‘s child:

1. 
2. 
3. 
4. 
5. 
6. 
7. 
8. 
9. 
10. 

Your task is to complete the class implementation following the above steps. Please note:

* When randomly generating the graph, the **weight** should be in a range: [1 … 25].
* The output of the finished program should:
  + display the generated graph
  + if the generated graph is not connected the minimum spanning tree cannot be computed, so the appropriate message must be displayed
  + if the graph is connected:
    - display the computed adjacency matrix for the minimum spanning tree for that graph. Row 0 and column zero are not displayed since they are used for checkmarks.
    - display the minimum spanning tree in level order
    - display the edges which are no longer present in the minimum spanning tree (displayRoadsThatCanBeClosed method)
* Your implementation should optimize the algorithm: notice that after step 6 row 1 has no more slots left that are not grayed out hence the subsequent step should ignore this row. After step 7 row 3 should no longer be considered as so on.

See sample runs below for examples of connected and disconnected graphs (I used Random with seed of 61) :

How many nodes in your graph?

10

Probability of edge? (type 70 for 70%)

50

\*\*\*\*\* GENERATED GRAPH \*\*\*\*\*

[ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]

[ 1] 16 1 24 5 24 2

[ 2] 16 21 24

[ 3] 12 3 21

[ 4] 1 14 18

[ 5] 24 21 21 23 22

[ 6] 12 21 4 3

[ 7] 5 14 23 4 14

[ 8] 7

[ 9] 24 24 3 22 3 7 4

[10] 2 21 18 14 4

The graph has cycles.

\*\*\*\*\* Computed MINIMUM SPANNING TREE for the above graph \*\*\*\*\*

[ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]

[ 1] 1 2

[ 2]

[ 3] 3

[ 4] 1

[ 5]

[ 6] 4 3

[ 7] 4

[ 8] 7

[ 9] 3 3 7 4

[10] 2 4

The following roads can be closed:

1 ==== 2

1 ==== 5

1 ==== 7

1 ==== 9

2 ==== 5

2 ==== 9

3 ==== 6

3 ==== 10

4 ==== 7

4 ==== 10

5 ==== 6

5 ==== 7

5 ==== 9

7 ==== 10

The minimum spanning tree is acyclic.

\*\*\*\*\* Computed MINIMUM SPANNING TREE in Level-order \*\*\*\*\*

level 1: 1

level 2: 4 10

level 3: 9

level 4: 3 6 8

level 5: 7

Process finished with exit code 0

If the generated graph is **not connected** the minimum spanning tree **cannot be computed**:

How many nodes in your graph?

8

Probability of edge? (type 70 for 70%)

25

\*\*\*\*\* GENERATED GRAPH \*\*\*\*\*

[ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8]

[ 1] 24

[ 2] 5 22

[ 3] 5 21

[ 4]

[ 5] 21

[ 6] 22

[ 7] 24

[ 8]

The graph is not connected, the minimum spanning tree will not be calculated

Process finished with exit code 0

The following UML design was utilized in implementation of the above solution:

**A screenshot of a cell phone

Description automatically generated**

# StrongWeakDisjoint

Let us consider the case where the roads connecting our towns are one-way streets. The towns are represented as vertices of a graph and the roads are represented as the graph’s edges. Since the direction of the travel is not bi-directional, the vertices and edges form a directed graph, therefore any town can be reached from any other town if and only if the graph is **strongly connected**.

For example, in the following graph it will not be possible since the graph is **weakly connected** (we cannot reach any vertex from vertex 2):

A close up of a clock

Description automatically generated

To determine if an undirected graph is **connected**, we would perform a breath-first-traversal beginning at any vertex and then check to see if all the vertices are visited. This would not be however sufficient to determine if a **directed graph is** **strongly connected**.

If the traversal is initiated at every vertex and each of these traversals visits every vertex in the graph, then we could state that the graph is **strongly connected** – this would however be inefficient **O(n3)** solution.

The following algorithm presents an alternative **O(n2) method** for determining if a directed graph is strongly connected and if it is not, also giving a rapid way of determining which vertices have paths connecting them.

**STEP 1**

The algorithm begins by copying the **adjacency matrix** representing the given graph into another array referred to as ***reachability matrix.*** Then it modifies the reachability matrix by placing a **1** in column **j** of row **i** if there is a path from vertex **vi** to vertex **vj**. The path could be a single edge (a path length of 1, which would already be present in the reachability matrix) or the path could consist of a sequence of edges (a path length > 1 which would not appear in the adjacency matrix).

**1** in row **i**, column **j** of the **adjacency matrix** indicates that there is an **edge** between vertices **vi** and **vj**, however, **1** in the corresponding element in the **reachability matrix** indicates that **there is a path** from vertex **vi** to vertex **vj**.

For example, the graph to check (as shown in the picture above) has the following reachability matrix:

A screenshot of a cell phone

Description automatically generated

The algorithm has changed the **0** in columns **2** and **4** of row **0** of the adjacency matrix (which indicates that **there is no edge** from vertex **0** to vertex **2** or **4**), to a **1** in reachability matrix (which indicates that **there is a path** from vertex **0** to vertex **2** and **there is a path** from vertex **0** to vertex **4**). Other rows of reachability matrix show similar changes. The only row unchanged is row **2**, since there are no paths from it to any other vertex in the graph.

**STEP2**

After generating the reachability matrix, we can determine if the graph is strongly connected by examining the reachability matrix elements. If all elements of the matrix are **1**, except for the elements along the main (upper-left-to-lower-right) diagonal, then the graph is strongly connected.

In addition, if the graph is not strongly connected, we can rapidly determine if there is a path from vertex **vi** to vertex **vj** by simply checking if reachability matrix at indexes **[i][j]** is **1**.

BASIS FOR THE ALGORITHM

The basis for the algorithm is the transitive property in mathematics: if a=b and b=c, then a=c.

Therefore, if there is a path from vertex **vb** to vertex **vc** then there is a path to **vc** from every vertex that can reach **vb**. The algorithm examines each element of the reachability matrix working its way across the columns beginning with row **0**. When it finds an element with a value of **1** (e.g. **reachabilityMatrix[b][c]** is **1**, indicating a path exists from vertex **vb** to vertex **vc**), it indexes its way down column **b** of the matrix to find the vertices with a path to **vb** (e.g. **reachabilityMatrix[a][b]** is **1** indicating that there is a path from **va** to **vb**). If this is a case, there must be a path from **va** to **vc** (through vertex **vb**) so **reachabilityMatrix[a][c]** is set to **1**.

Provisions are made in the algorithm to not place a **1** along the diagonal of the matrix.

**STEP3**

If the directed graph is **not strongly connected** it can be **weakly connected**. To check for that we will create **adjMatrixUndirected** by copying elements from the original **adjMatrixDirected** in such a way that the new matrix is symmetrical (all the edges are bi-directional), and check if this graph is connected as we would for an undirected graph.

If the graph is not weakly connected either than it means that it is **disjoint**.

**Your task is to implement methods defined in StrongWeakDisjoint.java class.**

UML diagram:

A screenshot of a cell phone

Description automatically generated

Sample run:

# \*\*\* Checking graphs' connectivity \*\*\*

# \*\*\*\*\* GRAPH TO CHECK \*\*\*\*\*

# [ 0][ 1][ 2][ 3][ 4]

# [ 0] 0 1 0 1 0

# [ 1] 0 0 1 1 0

# [ 2] 0 0 0 0 0

# [ 3] 0 0 0 0 1

# [ 4] 1 0 0 1 0

# \*\*\*\*\* REACHABILITY MATRIX \*\*\*\*\*

# [ 0][ 1][ 2][ 3][ 4]

# [ 0] 0 1 1 1 1

# [ 1] 1 0 1 1 1

# [ 2] 0 0 0 0 0

# [ 3] 1 1 1 0 1

# [ 4] 1 1 1 1 0

# -->The graph is weakly connected.

# \*\*\*\*\* GRAPH TO CHECK \*\*\*\*\*

# [ 0][ 1][ 2][ 3][ 4]

# [ 0] 0 1 0 1 0

# [ 1] 0 0 1 1 0

# [ 2] 0 1 0 0 0

# [ 3] 0 0 0 0 1

# [ 4] 1 0 0 1 0

# \*\*\*\*\* REACHABILITY MATRIX \*\*\*\*\*

# [ 0][ 1][ 2][ 3][ 4]

# [ 0] 0 1 1 1 1

# [ 1] 1 0 1 1 1

# [ 2] 1 1 0 1 1

# [ 3] 1 1 1 0 1

# [ 4] 1 1 1 1 0

# -->The graph is strongly connected.

# \*\*\*\*\* GRAPH TO CHECK \*\*\*\*\*

# [ 0][ 1][ 2][ 3][ 4]

# [ 0] 0 1 0 1 0

# [ 1] 0 0 0 1 0

# [ 2] 0 0 0 0 0

# [ 3] 0 0 0 0 1

# [ 4] 1 0 0 1 0

# \*\*\*\*\* REACHABILITY MATRIX \*\*\*\*\*

# [ 0][ 1][ 2][ 3][ 4]

# [ 0] 0 1 0 1 1

# [ 1] 1 0 0 1 1

# [ 2] 0 0 0 0 0

# [ 3] 1 1 0 0 1

# [ 4] 1 1 0 1 0

# -->The graph is disjoint.

# Process finished with exit code 0

# HowManyQuestions

Your teacher is going to give a test where each student is to answer one question. None of the neighboring students should have the same question. How many questions are needed?

Graph Coloring Algorithm is used to solve this type of problems. It does not guarantee to use the minimum number of questions, but it guarantees an upper bound on the number of questions. The algorithm never uses more than d+1 questions where d is the maximum degree of vertices in the given graph – where the *degree of a vertex* is the number of edges connected to the vertex.

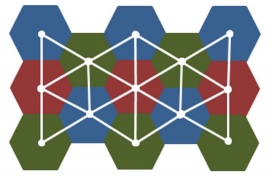
The Basic Greedy Graph Coloring Algorithm is as follow:

1. Assign first vertex with the first color.
2. Do following for remaining V-1 vertices.
   1. Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v, assign a new color to it.
   2. If there are no more colors available to choose from the problem cannot be solved

**Example**: Map coloring of the states in the U.S. – none of the neighboring states share the same color:



**Example**: GSM is a cellular network with its entire geographical range divided into hexagonal cells. Each cell has a communication tower which connects with mobile phones within the cell. All mobile phones connect to the GSM network by searching for cells in the immediate vicinity. GSM networks operate in only four different frequency ranges. The reason why only four different frequencies suffice is clear: the map of the cellular regions can be properly colored by using only four different colors! So, the vertex coloring algorithm may be used for assigning at most four different frequencies for any GSM mobile phone network as shown in the figure below:



**Your task** is to utilize the above concept to compute how many questions are needed and which question should be assigned to each student. Constructor is given a boolean matrix that indicates if a connection exists between vertexes. Based on the passed parameter the number of vertices is saved in the instance variable and an undirected graph that is represented as **adjacency list** is created. A **vertex** in the graph represents a **student**, an **edge** connects a pair of students that are **neighbors**.

Your program starts with the number of questions set to two and tries to find the solution where none of the neighbors have the same question. If the solution is not found one more question is added, and the process is repeated. The program stops when the solution is found and displays to the user the number of questions needed to satisfy the requirement.

See the sample run below:

Created graph 1:

The graph has 5 vertexes with the following neighbors:

Vertex 0 has neighbors: 1 2

Vertex 1 has neighbors: 0 2 3

Vertex 2 has neighbors: 0 1 3

Vertex 3 has neighbors: 1 2 4

Vertex 4 has neighbors: 3

=================

Created graph 2:

The graph has 5 vertexes with the following neighbors:

Vertex 0 has neighbors: 1 2 3

Vertex 1 has neighbors: 0 2 3 4

Vertex 2 has neighbors: 0 1 4

Vertex 3 has neighbors: 0 1 4

Vertex 4 has neighbors: 1 2 3

=================

Created graph 3:

The graph has 5 vertexes with the following neighbors:

Vertex 0 has neighbors: 1 2 3

Vertex 1 has neighbors: 0 2 3 4

Vertex 2 has neighbors: 0 1 3

Vertex 3 has neighbors: 0 1 2 4

Vertex 4 has neighbors: 1 3

=================

Created graph 4:

The graph has 24 vertexes with the following neighbors:

Vertex 0 has neighbors: 1 23

Vertex 1 has neighbors: 0 2

Vertex 2 has neighbors: 1 3

Vertex 3 has neighbors: 2 4

Vertex 4 has neighbors: 3 5

Vertex 5 has neighbors: 4 6

Vertex 6 has neighbors: 5 7

Vertex 7 has neighbors: 6 8

Vertex 8 has neighbors: 7 9

Vertex 9 has neighbors: 8 10

Vertex 10 has neighbors: 9 11

Vertex 11 has neighbors: 10 12

Vertex 12 has neighbors: 11 13

Vertex 13 has neighbors: 12 14

Vertex 14 has neighbors: 13 15

Vertex 15 has neighbors: 14 16

Vertex 16 has neighbors: 15 17

Vertex 17 has neighbors: 16 18

Vertex 18 has neighbors: 17 19

Vertex 19 has neighbors: 18 20

Vertex 20 has neighbors: 19 21

Vertex 21 has neighbors: 20 22

Vertex 22 has neighbors: 21 23

Vertex 23 has neighbors: 0 22

=================

Created graph 5:

The graph has 24 vertexes with the following neighbors:

Vertex 0 has neighbors: 1 10 11

Vertex 1 has neighbors: 0 2 9 10 11

Vertex 2 has neighbors: 1 3 8 9 10

Vertex 3 has neighbors: 2 4 7 8 9

Vertex 4 has neighbors: 3 5 6 7 8

Vertex 5 has neighbors: 4 6 7

Vertex 6 has neighbors: 4 5 7 16 17

Vertex 7 has neighbors: 3 4 5 6 8 15 16 17

Vertex 8 has neighbors: 2 3 4 7 9 15 16 17

Vertex 9 has neighbors: 1 2 3 8 10 13 14 15

Vertex 10 has neighbors: 0 1 2 9 11 12 13 14

Vertex 11 has neighbors: 0 1 10 12 13

Vertex 12 has neighbors: 10 11 13 22 23

Vertex 13 has neighbors: 9 10 11 12 14 21 22 23

Vertex 14 has neighbors: 8 9 10 13 15 20 21 22

Vertex 15 has neighbors: 7 8 9 14 16 19 20 21

Vertex 16 has neighbors: 6 7 8 15 17 18 19 20

Vertex 17 has neighbors: 6 7 16 18 19

Vertex 18 has neighbors: 16 17 19

Vertex 19 has neighbors: 15 16 17 18 20

Vertex 20 has neighbors: 14 15 16 19 21

Vertex 21 has neighbors: 13 14 15 20 22

Vertex 22 has neighbors: 12 13 14 21 23

Vertex 23 has neighbors: 12 13 22

=================

\*\*\*\*\*\* Checking if 2 questions are enough \*\*\*\*\*\*

\*\*\* Checking graph 1 \*\*\*

--> The solution does not exist - not enough choices

\*\*\* Checking graph 2 \*\*\*

--> The solution does not exist - not enough choices

\*\*\* Checking graph 3 \*\*\*

--> The solution does not exist - not enough choices

\*\*\* Checking graph 4 \*\*\*

--> The solution exists with 2 questions:

Student 0 ---> Question 0

Student 1 ---> Question 1

Student 2 ---> Question 0

Student 3 ---> Question 1

Student 4 ---> Question 0

Student 5 ---> Question 1

Student 6 ---> Question 0

Student 7 ---> Question 1

Student 8 ---> Question 0

Student 9 ---> Question 1

Student 10 ---> Question 0

Student 11 ---> Question 1

Student 12 ---> Question 0

Student 13 ---> Question 1

Student 14 ---> Question 0

Student 15 ---> Question 1

Student 16 ---> Question 0

Student 17 ---> Question 1

Student 18 ---> Question 0

Student 19 ---> Question 1

Student 20 ---> Question 0

Student 21 ---> Question 1

Student 22 ---> Question 0

Student 23 ---> Question 1

\*\*\* Checking graph 5 \*\*\*

--> The solution does not exist - not enough choices

\*\*\*\*\*\* Checking if 3 questions are enough \*\*\*\*\*\*

\*\*\* Checking graph 1 \*\*\*

--> The solution exists with 3 questions:

Student 0 ---> Question 0

Student 1 ---> Question 1

Student 2 ---> Question 2

Student 3 ---> Question 0

Student 4 ---> Question 1

\*\*\* Checking graph 2 \*\*\*

--> The solution exists with 3 questions:

Student 0 ---> Question 0

Student 1 ---> Question 1

Student 2 ---> Question 2

Student 3 ---> Question 2

Student 4 ---> Question 0

\*\*\* Checking graph 3 \*\*\*

--> The solution does not exist - not enough choices

\*\*\* Checking graph 5 \*\*\*

--> The solution does not exist - not enough choices

\*\*\*\*\*\* Checking if 4 questions are enough \*\*\*\*\*\*

\*\*\* Checking graph 3 \*\*\*

--> The solution exists with 4 questions:

Student 0 ---> Question 0

Student 1 ---> Question 1

Student 2 ---> Question 2

Student 3 ---> Question 3

Student 4 ---> Question 0

\*\*\* Checking graph 5 \*\*\*

--> The solution exists with 4 questions:

Student 0 ---> Question 0

Student 1 ---> Question 1

Student 2 ---> Question 0

Student 3 ---> Question 1

Student 4 ---> Question 0

Student 5 ---> Question 1

Student 6 ---> Question 2

Student 7 ---> Question 3

Student 8 ---> Question 2

Student 9 ---> Question 3

Student 10 ---> Question 2

Student 11 ---> Question 3

Student 12 ---> Question 0

Student 13 ---> Question 1

Student 14 ---> Question 0

Student 15 ---> Question 1

Student 16 ---> Question 0

Student 17 ---> Question 1

Student 18 ---> Question 2

Student 19 ---> Question 3

Student 20 ---> Question 2

Student 21 ---> Question 3

Student 22 ---> Question 2

Student 23 ---> Question 3

\*\*\*\*\* DONE - all graphs were assigned solutions \*\*\*\*\*

Process finished with exit code 0

The following UML design was utilized in implementation of the above solution:

**A screenshot of a cell phone

Description automatically generated**

**This is your last Comp151 project, have fun!!!**