Assignment 2

This second assignment focuses on a DFS search. You will accomplish a number of tasks in this assignment:

- You will learn what an *upstream repository* is, set it up for your project and use it to update your files.
- You will continue using Eclipse[1] to review and edit the Java code for the assignment, and to compile the project files into executables.
- You will use the Gradle[2] tool to manage your Java project and run tests.
- You will continue learning about Java programming by studying the provided code and filling in the requested parts.
- You will gain a deeper understanding of depth-first search by completing the provided implementations of DFS to locate graph cycles, connected components, and checking for bipartiteness.

Let's get started!

Updating your repository

Earlier in the term you made a copy of my GitLab repository over to your account, we called this a *fork*. You then cloned your GitLab repository to your local computer. Your computer now has a *local* repository, which is linked to a *remote* repository, the one in your GitLab account. The name **origin** is used to identify this remote repository. You can see this when you look in GitKraken. You should see a "Remotes" section on the left, and an "origin" repository in there.

Since then you have added many commits to your local repository and pushed them onto your remote repository (origin). At the same time, I have created the second assignment files and added them to *my* GitLab repository. What you will need to do is copy these updates from my repository over to yours. This is a 2/3-step process:

1. We will link your local repository to *my* GitLab repository as a second remote. This is called **upstream repository**, the idea being that we want changes made to that branch to trickle downstream to your repository. You only need to do this once. In future labs you'll be able to skip this step.
2. After we have set up this upstream repository, we will *fetch* changes from that repository into your local repository, and merge those changes in to your repository.

[1] https://www.eclipse.org/
[2] https://gradle.org/

3. We will then push those changes to *your* remote (*origin*).

Then you'll be able to start work on the assignment.

Let's get to it!

Setting up an upstream

Before we start anything, make sure that you have NO uncommitted changes. You must either commit what you have, if it is ready to commit, or stash the changes, allowing you to retrieve them later.

To add a new upstream remote, we will use GitKraken's interface. Hover over the "Remotes" section on the left, and a Plus button should appear to add a new remote. Click on it, and a little window will appear to let you choose the remote. Find the remote that is in the skiadas account and called algorithms−assignments. Its name will likely be skiadas/algorithms−assignments. Then as the name for this repo put upstream. Then click "Add Remote". You should now be seeing a new remote on the left, called upstream, and you should be seeing some new branches showing up in the main screen, corresponding to that remote.

Merging the upstream changes

Once again the first key step is to make sure that you have NO uncommitted changes. You must either commit what you have, if it is ready to commit, or stash the changes, allowing you to retrieve them later.

We now want to merge in the changes that the upstream remote has into your remote.

1. First make sure that your local master is the active branch. It should have a little checkbox next to it.
2. We should make sure to bring in the newest version of the upstream remote. Right-click on the "upstream" remote in the "Remotes" section on the left, and choose "fetch upstream".
3. Now right-click on the upstream/master branch within the "Remotes" section to bring up its context menu. One of the options will say "Merge upstream/master onto master". Click it.
4. You should now be ready to go. Whenever you push your changes to your origin, these commits from the upstream will go along with them. If you like, you can go ahead and push now.

The assignment

Introductory stuff

In the assignment you work with a simple graph implementation and depth-first-search implementations on it. You should already be familiar with the depth-first-search material from the book.

As in the previous assignment, you will find numerous TODO comments throughout the file. They mark the locations where you need to add something to the implementations. Some times it will be one or two lines, some times it may be more. If you use Eclipse, you can see blue rectangles on the right side of the code indicating the location of all TODOs, and you can click on them to go there. You can keep track of your progress that way, so make sure to remove the TODO from the methods you have already implemented.

A standard printout of a run of the various methods can be obtained by running the main method of the DFS class (so running the DFS class as an application). To do that, you will need to first build using the corresponding gradle task, and then run the runDFS gradle task from the run group of tasks (you may need to refresh the gradle project and possibly close and reopen the gradle tasks to make it appear).

Alternatively, you can go to your terminal (or Eclipse's console), go to the main project folder and run:

```
java -cp build/libs/algorithms-assignments.jar utils.DFS
```

If you like to write more tests, you can create test files under /src/test/java. Ask for help if you are not sure how to start those tests. You can use the :test Gradle task to run the tests.

You should make a new *commit* in GitKraken after completing each self-contained part, typically after each method you complete. Do not try to commit everything at once.

When you are done with the assignment, in order to "submit" it, simply **push** your commits via GitKraken. These will now be posted on your account in GitLab and I will be able to follow them from there.

Important: This assignment assumes that you have a working DLList implementation from the first assignment. If that is not the case, talk to me about replacing uses of DLList with the Java class LinkedList.

The structure of the files

There are two files of concern in this assignment. SimpleGraph.java, and DFS.java.

- The SimpleGraph class contains a simple implementation of a graph. You do not need to make any changes to it, but you should take a look at it and see what methods are provided (You can also run the :javadoc task to generate documetation for these files and look at that documentation, but the actual file is short and contains more details).

The SimpleGraph class represents vertices via integers $0, 1, \ldots, n-1$. There is an order method that returns the number n of vertices. The edges of the graph are stored in adjacency list form: There is an array of DLList<Integer> lists, whose i-th entry contains a list of all the neighbors of i. So if j is contained in the list stored at edges[i], then there is an edge from i to j. The method outgoingEdges returns this list.

The class can be used for either directed graphs, using the addEdge method to add edges, or as undirected graphs, using the addBidirectionalEdge method to add edges. You will need these methods if you want to write your own example graphs.

- The DFS class contains four inner classes implementing depth-first-search for various goals. Your assignment will be to fill in those implementations. They all follow some similar patterns:
  - There are a number of static properties/fields maintaining information about the search, as as which vertices have been visited already, and any other information we may need.
  - All classes use a visited array to keep track of which vertices have been visited. When a vertex is visited for the first time, it must be marked as visited.
  - All classes use a parent array that holds the parent links between a vertex and its parent. The parent array must be initialized with values equal to -1. The Arrays.fill method can help out with that. When the DFS search follows a tree edge, it also sets an entry in the parent array.
  - There is a constructor initializing the main graph fields.
  - There is a run method which starts the DFS search. It will typically do two things:
    - Initialize other fields that keep track of the traversal information.
    - Find an unvisited vertex and start a dfs descent from that vertex. Repeat this until all vertices are visited.
  - There is a recursive dfs method that does a search starting from a vertex. It is the heart of the search, and it is called by the run method. It has the following main elements:
    - It starts by recording that the vertex is visited, and any other related information.
    - It proceeds through the outgoing edges from this vertex.
      - It ignores an edge that points back to the parent of the current vertex, since we just traversed that edge in the other direction.
      - If the edge is a *back-edge*, i.e. towards an already-visited vertex, then the algorithm may need to do something, depending on the goal of the search.

        · If the edge is a *tree-edge*, i.e. towards a yet-unvisited vertex, then the algorithm must update the parent entry, and then may record some other information and then recursively call dfs on this vertex.
    * It concludes the visit by doing any needed cleanup.
- There may be other diagnostic/printing methods.

All classes must do the following things in addition to their own extra needs. The code you add must ensure these steps happen.

- Initialize the visited array entries to false and the parent array entries to -1 at the start of the run method.
- Mark the current vertex as visited at the beginning of the dfs method.
- Update the parent array whenever a tree edge is encountered, then recursively calling dfs on the target/child vertex of that tree edge.

The four inner classes are as follows:

- simpleDFS demonstrates the algorithm process and it keeps track of any relevant information, printing diagnostic messages along the way. In it you will maintain a number of properties/fields:
    * A stack keeps track of the vertices as they get visited or are in the process of being visited. It is not technically needed as the recursive calls contain in them the same information, but it can be useful for visualizing the process.
    * pushOrder and popOrder fields record the order in which vertices are being put in the stack or removed from it.
- ComponentsDFS uses a depth-first search to collect the connected components of the graph. Uses a Set<Integer> structure to store each component, and a DLList of such structures to hold the list of all components. Its main elements are:
    * A components DLList holding the components as they are being constructed. Depending on how you implement things, the “current component” on which vertices are to be added will be either the last or the first element on that list.
    * When the run method starts a new dfs from a root vertex, a new component set must be created. You will need to use a concrete class like HashSet rather than the interface Set to actually create a set for this new component.
    * When a vertex is visited, it is added to the current component set.
- CycleDFS tries to detect if there is a cycle in the graph. Cycles exist whenever a back edge is encountered. Such an edge connects the vertex currently being visited to one of its ancestors. It is therefore an indication that this vertex and its ancestor are connected in two different ways: Through the dfs-tree that brought us from the ancestor down to the current vertex, and this new back-edge that closes the loop. This class contains the following elements:

    * A cycle list that starts empty, and if there is a cycle then we will store in the list the vertices in the cycle.
    * If a back edge is encountered, we record the cycle in the cycle variable and return.
    * As a first step when we consider a new edge, we check the cycle list and if it is non-empty we must return early to stop the search, since a cycle has been found.
  - BipartiteDFS tries to determine if the graph is bipartite. It does so by assigning one of two colors to each vertex as we descend on it, and watching for conflicts. It contains the following elements:
    * A colors array uses the numbers 1 and 2 to place the vertices in one of two "groups" based on their color.
    * A failed boolean variable keeps track of whether the coloring has failed. Whenever we look at an edge we check if the coloring has failed and return early if it has.
    * A new color of 1 is assigned to a new root vertex.
    * Each time a tree-edge is encountered, a color is assigned to the new vertex opposite to the color from the parent vertex (i.e. 2 if the parent vertex had 1, and vice versa).
    * Each time a back-edge is encountered, the colors of the two vertices it connects are checked. If those colors disagree, then the coloring has failed and the graph is not bipartite.

The actual tasks

Now on to the tasks you need to complete. Make sure you have read the previous section for an understanding of the structure of the files.

1. We start with the SimpleDFS class.

   - Your work starts at its run method. At the beginning of the method you must initialize the visited and parent arrays, create empty pushOrder and popOrder lists, and an empty stack.
   - Next you need to work on its dfs method.
     - At the begining of the method you must add the current vertex to the stack, mark it as visited, and also add it to the pushOrder list.
     - Then the dfs method goes through the outgoing edges from this vertex. On back-edges a message is printed, and you don't need to do anything.
     - When a tree-edge is encountered, a message is printed and you must add a recursive call to dfs for the new vertex. Don't forget to update the parent entry before you do that.
     - When the loop over the edges is completed, the current vertex has become a dead-end. You must pop it from the stack and add it to the popOrder list.

2. Next we have to implement the ComponentsDFS class. This is probably the easiest of the four classes to implement.

    - Your work starts at its run method.
        - At the beginning of the method you must initialize the visited array, and create an empty components list.
        - When the run method is about to visit a brand new root vertex, a new component must be created and placed in the components list.
    - Next you need to work on its dfs method.
        - At the begining of the method you must add the current vertex to the current component. Depending on how you implemented the run method above, this would be either the first or the last element of the components list.
        - Then the dfs method goes through the outgoing edges from this vertex. If a tree-edge is encountered, you must recursively call dfs on the new vertex. Don't forget to update the parent entry before you do so.
        - Nothing needs to happen on back edges, so there is no else clause.

3. The third inner class to implement is the CycleDFS class. It looks for a back-edge and uses it to construct a cycle.

    - Your work starts at its run method.
        - At the beginning of the method you must initialize the visited and parent arrays, and also create an empty cycle list to hold the cycle if you find one. The parent array will contain the dfs tree link back from a vertex to its parent and we will use it to construct the cycle when we encounter it.
        - Nothing else needs to happen in the run method.
    - Next you need to work on its dfs method.
        - At the begining of the method you must check if the cycle list is non-empty. This means we found a cycle already, and can simply return right away.
        - Then the dfs method goes through the outgoing edges from this vertex. You must check if a cycle has been found and return early if it has.
        - If a tree-edge is encountered, then you must assign the correct parent entry and recursively call dfs on the new vertex.
        - When a back-edge is encountered, you must use its information to add in the cycle variable the vertices that comprise the cycle loop that the back-edge just closed. We can then return early from our dfs method as a cycle has been found.

4. The last inner class we have to work on is the BipartiteDFS class. It uses the DFS search to determine if the graph is bipartite, and what the two parts of it are. It keeps "colors" for the vertices in the colors array, and also a boolean failed variable to indicate if the search has failed or not.

- Your work starts at its run method.
  - At the beginning of the method you must initialize the visited array, and a colors array filled with 0s, and also set the failed boolean to false.
  - When the run method is about to start a dfs search from a root vertex, you must assign a color to that vertex. It is safe to use the color 1, since this is a new component that will not connect to the previous components (If you want a challenge, make it so that brand new root vertices are colored alternately, so 1 for the first one, 2 for the second, 1 or the third etc. You will probably need to add a new field to achieve that).
- Next you need to work on its dfs method.
  - At the begining of the method you must check if the search has failed, and if so return early.
  - When a tree-edge is encountered, you must assign a color to the new vertex before recursing on it, and that color must be the opposite of the color used for the current vertex.
  - When a back-edge is encountered, you must check if the color assignments to the two vertices of the edge are the same or not. If they are the same, that means that the graph is not bipartite because the 2-coloring is impossible, and we can set the failed variable to true and return. If the color assignments are opposite, then this back-edge is not a problem and we can simply continue our run.