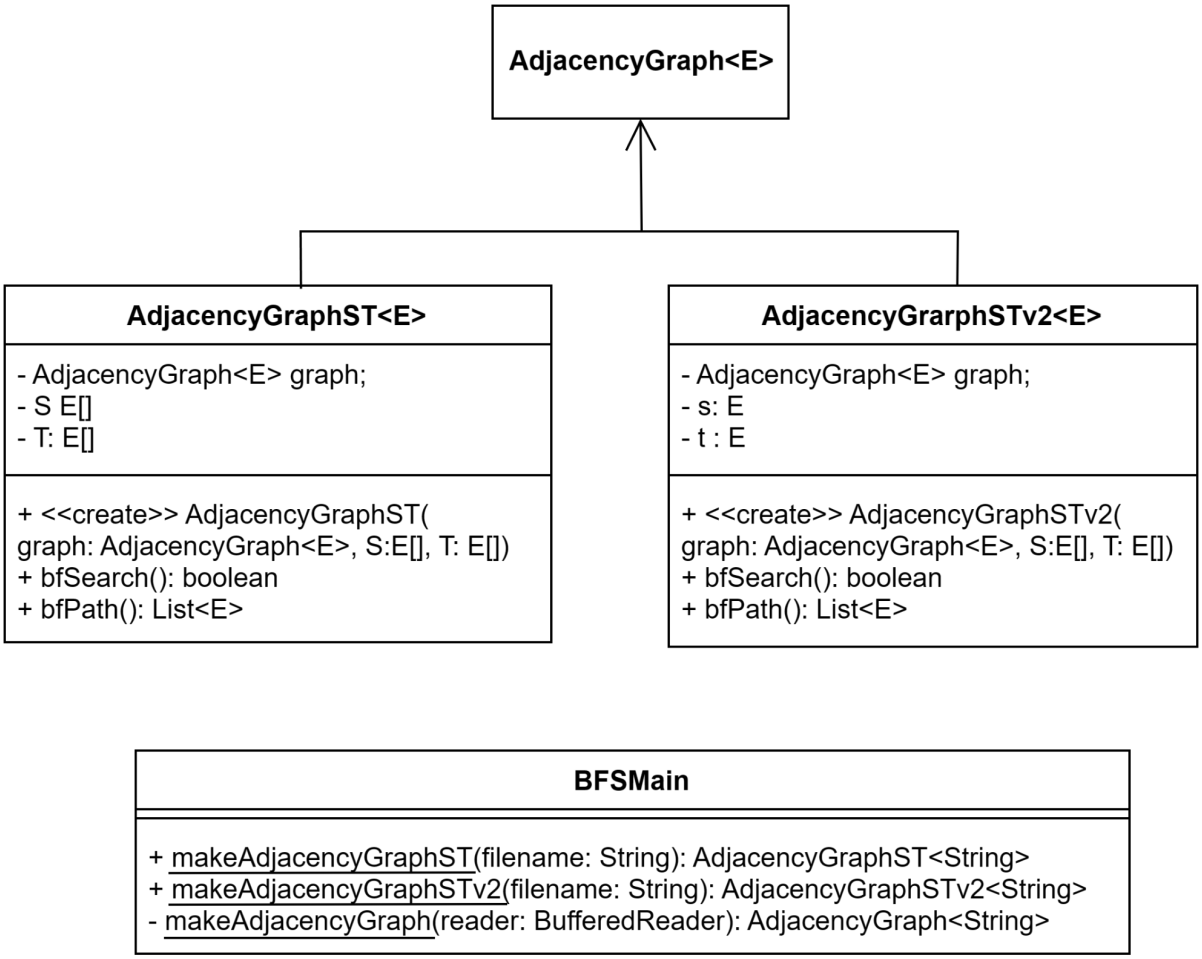
Graphs & Breadth-First Search

Goals of the Assignment

Practice using graphs and searches with graphs. This assignment will require the Graph interface and the Vertex and AdjacencyGraph classes implemented during class as well as the implementation of the BFS algorithm. JUnit unit testing is not expected for this assignment. Read this document ***in its entirety*** before asking for help from the course staff.

A Path Between Two Sets

Consider a graph with subsets of vertices called **S** and **T**. We say that a path exists from **S** to **T** if there exists a path that connects at least one vertex in **S** to at least one vertex in **T**. We call **S** and **T** the **source** and **target** sets, respectively. Your task for this assignment is to use the Breadth-First Search (BFS) algorithm to ascertain whether or not a path exists from the source to the target set, and if it does, to find the shortest path.



Part 1 - Naive Searches

1. All of your code should go in the `unit07.sourceTarget` package.

2. Using the UML diagram above, begin writing the **AdjacencyGraphST** class that represents a graph with source and target. The new class is generic and uses the `AdjacencyGraph` class discussed in the lecture. The new graph class provides the following BFS methods that work with the source and the target.
    a. `bfSearch()` returns true if there exists a path between the source and the target, and false otherwise.
    b. `bfPath()` returns the shortest path connecting one of the vertices in the source to one of the vertices in the target if it exists. The method returns null otherwise.

3. Create the **BFSMain** class and add a static method named `makeAdjacencyGraphST(String filename)`. The method reads the specified file to construct an instance of AdjacencyGraphST and returns it. You will use files named "**graph*.txt**" in the **data/bfs** directory of your repository. Take a moment to examine the text files.
    a. The first line is a list of values in the source vertices, separated by a blank space.

b. The second line is a list of values in the target vertices, separated by a blank space.

a. Each of the remaining lines represents an adjacency list where the first value is a vertex in the graph, and the remaining values are the vertices to which the first value is connected. For example, the line "A B C" indicates that vertex A is connected to vertex B and vertex C. Assume that the connections between neighbors are *directed*.

b. The same vertex may appear on multiple lines in the file. You only add each value to the graph *once*.

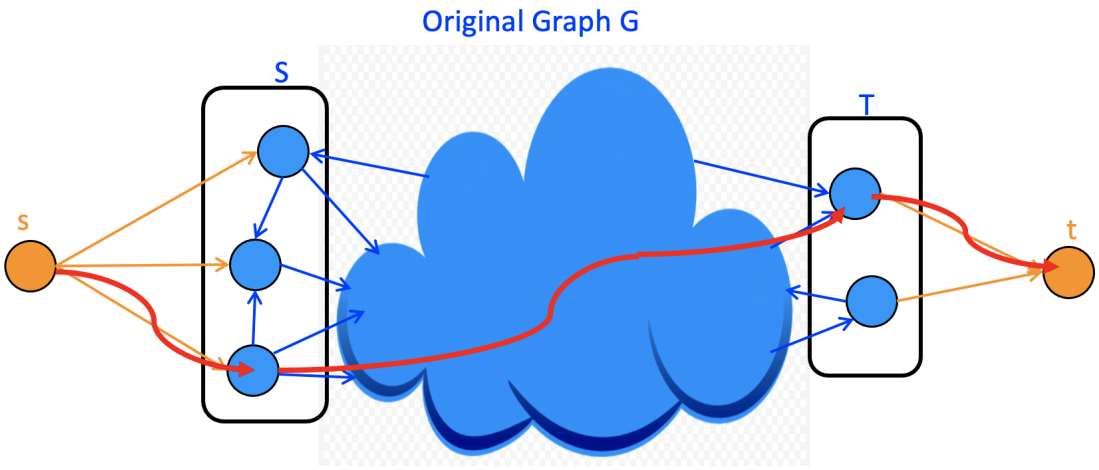
c. If an IOException occurs while trying to read a graph, you may rethrow it.

Consider writing a helper method for constructing an AdjacencyGraph as you can reuse this in the following requirements for Part 2.

3. Add main() to the **BFSMain** class with an appropriate signature. Use the `makeAjacencyGraphST` method to create instances of the new graph class and test the new bfSearch and bfPath methods. Several graph files and a solution to each graph have been provided in the data/bfs folder.

Part 2 - Fast Searches

As you may have noticed, the `bfSearch` and `bfPath` methods you wrote in Part 1 are not very efficient because they perform BFS for *every* pair of start and end vertices. However, there is a more efficient way to determine whether or not a path exists between two sets. The idea is to add two dummy vertices, **s** and **t**, and connect them to the original graph in the following way: **s** is connected to every vertex in the source set **S**, and every vertex in the target set **T** is connected to **t**. We can use the modified graph with our search methods that only work with two end vertices. With the dummy source and target vertices, a one-time search should suffice to get the desired answer.



1. Create a class named "**`AdjacencyGraphSTv2`**" that represents the updated graph including those additional vertices and edges as described earlier (orange in the image

above) in addition to the original ones (blue in the image). Use the UML diagram on the top page to implement the class.

2. In your **BFSMain** class, write the `makeAdjacencyGraphSTv2(String filename)` method that you will use to create an instance of the new version of your graph class.

3. Test your new class in `BFSMain.main()`.

Submission Instructions

You must ensure that your solution to this assignment is pushed to GitHub *before* the start of the next lecture period.