Union find Optimization:
1. union-find by rank:
rank by the height:
1 -> 2. => 1 and 2, both have height 0
since they have an edge, we do a union. 2 (0) —> 1 (1)
1 - 3 => 3 => find 1 return 1, find 3 return 3. Then we call union. We put the short tree under tall tree to minimize the overall height and speed up the find
4 - 5 => create representation 4 and 5
2. path compression
3 hop path will be broken down to both connect to the root.

Do not use direct address map. Waste space by having large integer vertixes

Assignment 0: Sockets and Graphs

Due date: Monday May 27th at 11:00pm Waterloo time

ECE 454: Distributed Computing

Instructor: Dr. Wojciech Golab wgolab@uwaterloo.ca

A few house rules

Collaboration:

- groups of 1, 2 or 3 students (please self-organize using LEARN)

Managing source code:

- do keep a backup copy of your code outside of ecelinux, for example using GitLab (https://git.uwaterloo.ca/)
- do not post your code in a public repository (e.g., GitHub free tier)

Software environment:

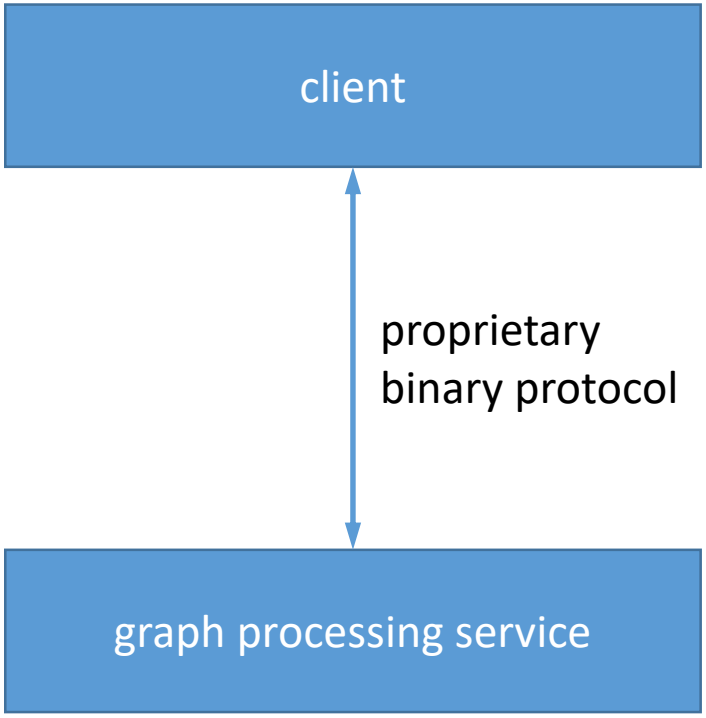
- test on ecelinux servers, which support Java version 1.8 and 1.11
- the environment used for grading will provide Java 1.11

Overview

- In this assignment you will develop a client-server system for processing graph data. You will be given some client and server starter code, and asked to complete the server's implementation in Java using sockets.
- The assignment has several objectives:
  1. to learn the basics of socket programming in Java
  2. to gain an appreciation of the difficult choice between single-threaded versus parallel code to process graph data

- This assignment is worth 8% of your final course grade.

Software Architecture



Protocol

- The client reads a graph from an input file and sends it in a request message to the server using TCP/IP.
- The server processes the graph and returns a response back to the client using the same TCP connection.
- The request and response messages follow the same format:
  - The first 4 bytes comprise a header that indicates the size (i.e., number of bytes) of the data payload that follows. This value is encoded as a **32-bit signed two's complement integer** using **big-endian** byte ordering.
  - The data payload is the **UTF-8** encoding of a **character string** that represents either the input graph (for a request) or the server's output (for a response). The string typically contains multiple line breaks.

Input

The input is an **undirected** graph represented as a **list of edges**. Each line contains one edge, which is a pair of vertex labels listed in ascending order, separated by a space.

Do not assume that vertex labels are consecutive, or that they start at 1. A vertex label may be any non-negative Java int. Vertex labels may be very large even if the input graph is small, and should not be used as array indexes.

Example input:

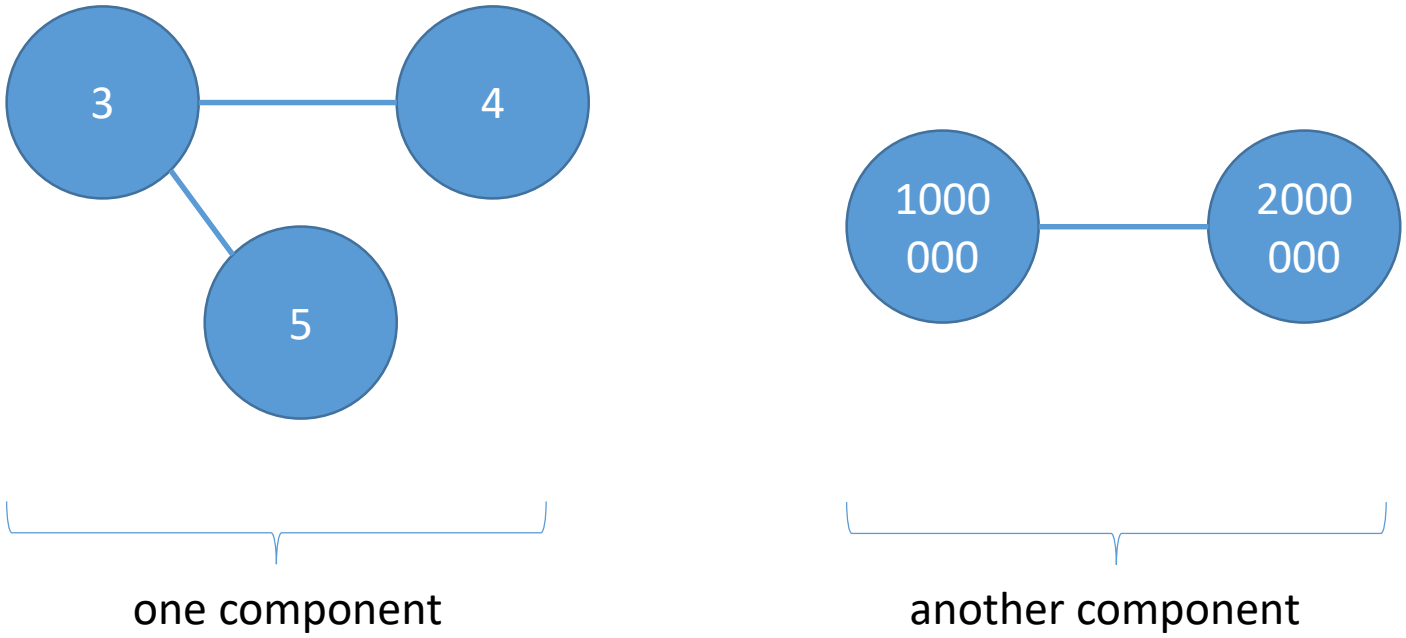
1000000 2000000

3 4

3 5

Input

The graph shown in the previous slide can be visualized as follows:



Output

Your goal is to solve the **connected components problem**. The output is a collection of lines, each comprising a vertex followed by its component label, separated by a space. The **order of the lines does not matter**. For each component, the component label must be equal to the label of some vertex in that component (but not necessarily equal to the min or max vertex label).

Example output for the input shown earlier:

2000000 1000000

1000000 1000000

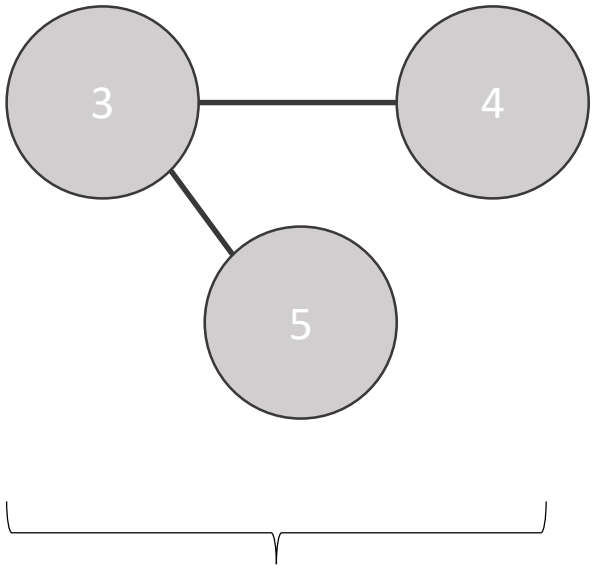
3 4

4 4

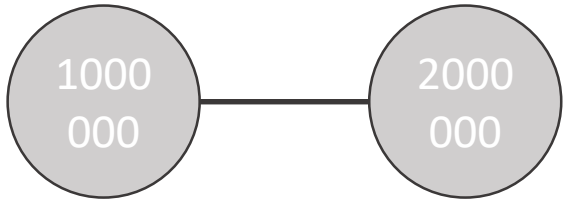
5 4

Output

The output shown in the previous slide can be visualized as follows:



component 4



component 1000000

Your coding task

- Your mission is to complete the Java server implementation by adding code for connection handling and graph processing.
- The Java server is implemented in the file **CCServer.java** in the default Java package. You may create additional source code files in the default Java package as you complete the implementation.
- The server program accepts one command line argument, namely the **TCP port number**. Do not change this part of the code.
- Inside the server program, you must implement a loop that performs the following steps:
  1. Accept a new client connection.
  2. Process requests from this connection repeatedly, **one at a time**.
  3. When the client closes the connection, go back to step 1.
- The above loop terminates when the server process receives a SIGINT (Ctrl + C), SIGTERM, or SIGKILL.

Group membership file

Include in your submission a text file called group.txt that lists the **Nexus IDs** of your group members (and nothing else), with one group member per line. The file should only contain **alphanumeric characters and line breaks**.

Example:

bsimpson
nmuntz
rwiggum

Packaging and submission

- All classes must be in the default Java package.
- Please include a group.txt file, as explained earlier.
- Use the provided package.sh script to create a tarball for electronic submission, and upload it to the appropriate LEARN dropbox before the deadline.
- The tarball should contain only your Java files and your group.txt file, all in the root directory of the archive.
- Do not include any code that does not compile.
- Do not use any external libraries or jar files.

Evaluation

Grading scheme:

Correctness:	50%
Performance:	50%

A penalty of up to 100% will be applied in the following cases:

- solution cannot be compiled or throws an exception during testing (on a good input)
- solution produces incorrect outputs, for example due to a logic error or concurrency bug
- brittle solution, for example one that runs out of memory because it uses vertex labels to index array elements

How to test your code

- The largest of the input data sets used for grading will be similar in size to the provided huge.txt graph.
- For a passing grade, your solution must produce correct outputs in all test cases and process each input graph in under 10 seconds on the two eceubuntu hosts, which use 3.4 GHz Intel i7 processors, while running on only two cores with 1GB maximum Java heap size.
- We will test your code on exactly two cores. The test script will control the number of cores assigned to your process using the taskset command. Example:

```
taskset -c 0,1 java -Xmx1g CCServer 10123
```

Good luck!

Hints: where to start

- First, find an efficient single-threaded algorithm for computing connected components in an undirected graph. (Hint: use the well-known "union-find" algorithm with union by rank and path compression optimizations.)
- Next, write code for parsing request messages and generating response messages.
- Test the client and server on small inputs. Run at least one test where the client and server processes are on different hosts. Ensure that your server can process many requests back-to-back.
- Once your code is working correctly on small inputs, try some larger inputs. Ensure that your implementation works with a 1GB heap, even on inputs with very large vertex labels.
- Finally, try to make your code faster. You can optimize your single-threaded implementation, or harness multi-core parallelism, or apply both strategies in different parts of the code.
- **Note:** Your server code will processes requests one at a time, and so parallelism in this context refers to the processing of one large request by multiple threads.

Hints: judging correctness

- The provided client code saves the server's output to a file. To determine whether the server's output is correct for the given sample input graphs, compare the server's output against the provided sample output.
- Since the correct output is not unique, a comparator program called Compare.java is provided with the starter code to help you decide if two outputs are equivalent.
- The provided run_client.sh script demonstrates how to use the comparator.