CS 210 Data Structures (Shen) – Fall 2022

Assignment 4 (**120** points)

Due Date – **11:59pm**, **11/22/2022**, **NO LATE** submission

You **must** do this assignment on your own.

A self-driving system is being rolled out to a high-speed train network. To maximize profitability, the company decides to start the system deployment to cities from which the driving system can drive the train to reach many other cities. You are asked to write code in **Part A** to figure out which cities to roll out the self-driving system first, and analyze the runtime computational complexity of your code for **Part B**.

Part A – Coding (100 points)

Given a list of cities, and a list of one-way routes from one city to another that have been tested for the self-driving system, implement an algorithm to:

- **Find** all reachable cities starting from each city following the one-way routes,
    - The self-driving system can drive a train from the starting city and stop at several cities before reaching the destination cities, **all cities** including the **starting** city and the **destination** city **along all paths** would be counted as **reachable** cities from the city where the train starts.
- **Sort** the cities in descending order of number of reachable cities and using ascending alphabetical order of the city code as the tiebreaker.

You will implement this algorithm in the **citiesSortedByNumOf_Its_ReachableCities_byTrain** function using a **recursive DFS algorithm** in the ConnectedCities class in **connectedcities.cpp** (code skeleton is provided on Canvas).

Code Skeleton given to you to start with:

- A **CityNode** class representing a city vertice / node in a connected city **directed graph** using an **adjacency list** representation. You must **NOT** change this class.
- A **ConnectedCities** class that has the declaration of the **citiesSortedByNumOf_Its_ReachableCities_byTrain** function to be implemented. You **must NOT** change the **signature** of this function; otherwise, your code would automatically fail auto-grading tests on Gradescope.
    - You will write the implementations for the function following what's specified in the **comment section above the function**. You may add helpers in the ConnectedCities class for helping your implementation.
    - **Important**:
        a. You need to write **at least one helper function for implementing the recursive Depth-first search (DFS)**, refer to the Hint for DFS comment section inside the ConnectedCities::citiesSortedByNumOf_Its_ReachableCities_byTrain function in connectedcities.cpp.
    - **Arguments** of **citiesSortedByNumOf_Its_ReachableCities_byTrain** function**:**
        a. cities: a list of cities with each city identified by a unique two letter code, like "SD", "LA", "SF", "SJ", "NY", etc.
        b. routes: pairs of one-way train routes with each one-way train route represented by a pair of city codes; for example, route {"SD", "LA"} means train can go one-way from San Diego (SD) to Los Angeles (LA). An example collection of routes could be: { {"SD", "LA"}, {"LA", "SJ"}, {"SJ", "SF"} }.

- **Returned value** of **citiesSortedByNumOf_Its_ReachableCities_byTrain** function**:**
    a. A list of CityNode objects in **descending** order of number of reachable cities and using **ascending** alphabetical order of city code as the tiebreaker.
    b. As put in the hints in the function, you can leverage the **std::stable_sort**(). One way to do the two-level of sorting here is to first sort the list by City code in ascending order, then sort by number of reachable cities in descending order.

- **Makefile:** A makefile to allow you to automate the compilation of your code. Check out the "Frequently Asked Questions" link in the C/C++ programming in Linux / Unix reference page on Canvas for simple Makefile reference.
    - Use the "make" command to compile. Use "make clean" to delete compiled object code and the executable.
    - Use **./citydfs** to execute.
    - You MUST **NOT** change the executable file name **citydfs in the Makefile**, otherwise, autograding will automatically fail**.**

Required:

1) You **must** follow the **specification** in the **comment section above** the function signature for your implementation.
2) You must use a **recursive** approach to implement the **Depth First Search** to find reachable cities as specified in the function comment section. Using an iterative approach would result in a **30% penalty** to your assignment 4 grade.

Hints:

1) See comments inside the function for hints.
    a. For high level recursive DFS algorithm: refer to zyBook Figure **20.6.1 Recursive depth-first search**.
    b. For sorting, you can leverage **std::stable_sort()**, you can obviously implement your own sorting.
2) You may add necessary helper functions in ConnectedCities class as needed.
3) You can use any C++ standard template library classes as you see fit.

Testing:

1) **You MUST write your own testing code (see grading below). A driver.cpp** (available on Canvas) is provided with sample testing code to get you started on writing your own testing cases. **Follow the comments** towards the end of the driver.cpp to write test code to test your implementation according to the specifications.
2) Please note: Autograder would only reveal the feedback of a limited amount of test cases (including the valgrind memory leaking detection, see below). A few test case results will be **withheld and only published** after the due.
   This is done to encourage you to write and use your own testing code for testing. Remember, that is part of your responsibilities as a software engineer or computer scientist. Establishing good habits of writing testing code would go a long way to help you become a true computer professional.

Programming and testing:

- Please refer to C/C++ programming in Linux / Unix page.
- You may use **C++ 11** standard for this assignment, see the given Makefile.
- We strongly recommend you set up your local development environment under a Linux environment (e.g., Ubuntu 18.04 or 20.04, or CentOS 8), develop and test your code there first, then port your

code to Edoras (e.g., filezilla or winscp) to compile and test to verify. The gradescope autograder will use a similar environment as Edoras to compile and autograde your code.

Grading for Part A:

Passing 100% auto-grading may NOT give you a perfect score for this Part A. The satisfaction of the algorithm requirements (see above), you code structure, coding style, and commenting will also be part of the rubrics (**see Syllabus Course Design - assignments**). Your code shall follow industry best practices:

- Testing code: your testing code in driver.cpp will be inspected as part of the manual grading for Part A (refer to comments in driver.cpp), points will be deducted for NO, minimal or insufficient testing code, meaning your tests need to cover necessary testing for the function you implemented. Again, refer to the comments towards the end of the driver.cpp.
- Be sure to comment your code appropriately. Code with no or minimal comments are automatically lowered one grade category.
- NO global variables.
- NO hard code – Magic numbers, etc.
- Have proper code structure between .h and .c / .cpp files, do not #include .cpp files.
- Design and implement clean interfaces between modules.

Part A grade breakdown:

- Main Autograding Test Cases (80%)
- Quality of your testing code (refer to comments in driver.cpp) (20%)

Part B – Complexity Analysis (20 points)

Show that the upper bound of the time complexity of your **citiesSortedByNumOf_Its_ReachableCities_byTrain** implementation would be **$O(c^2 + c * r)$**,

where:

1) c is the number of cities (nodes).
2) r is the number of direct self-driving routes (edges) between cities.

You would need to analyze the complexity of your implementation logic for the worst-case scenario as a function of the number of nodes and edges in the graph, then simplify it to the O notation.

Hints: Try to analyze the time complexity for each step of your algorithm, then sum the complexity from all steps, i.e., total time complexity = complexity of inserting nodes to construct the graph, finding the reachable cities from each city node in the graph, then sorting the city nodes by the city name, then by the number of their reachable cities.

Turning In

You need to submit the following program artifacts on Gradescope. Make sure that all submitted files contain your **name** and **Red ID**.

- **Part A:**
  - You should ONLY submit **connectedcities.h, connectedcities.cpp, and driver.cpp** source code files. **Do not** upload the Makefile, nor any of the compiled .o files.
  - Make sure you type the **Single Programmer Affidavit** (refer to the template on Canvas) as part of the comments at the beginning of your **connectedcities.cpp file**.
- **Part B: One PDF file**. It is recommended that you type it up, however, a scanned copy is allowed so long as the handwriting is legible. You can convert Word (.docx) files to PDF using the "Save as" functionality. You must only submit a **single** PDF file, image files (.jpg, etc) **will not** be accepted.

- **Important**:
  - Upload your files directly to Gradescope.
  - Do NOT **compress / zip** files into a ZIP file and submit, submit all files as they are.
  - Do NOT submit any .o files.
- **Number of submissions:**
  - Please note the autograder submission count when submitting on Gradescope. For this assignment, you are **limited to a maximum of 10 submissions.** As stressed in the class, you are supposed to do the testing in your own dev environment instead of using the autograder for testing your code. It is also the responsibility of you as a programmer to sort out the test cases based on the requirement specifications instead of relying on the autograder to give the test cases.

Academic honesty

Posting this assignment to any online learning platform and asking for help is considered academic dishonesty and will be reported.

An automated program structure comparison algorithm will be used to detect code plagiarism.

- The plagiarism detection generates similarity reports of your code with your peers as well as from online sources. We will also include solutions from the popular learning platforms (such as **Chegg**, etc.) as part of the online sources used for the plagiarism similarity detection. Note not only the plagiarism detection checks for matching content, but also it checks the structure of your code.
- Refer to Syllabus for penalties on plagiarisms.
- Note the provided source code in the code skeleton would be excluded in the plagiarism check.