MAC0328 GRAPH ALGORITHMS: PROGRAMMING ASSIGNMENT 0

ANCESTOR QUERIES

MARCEL K. DE CARLI SILVA AND HELOISA DE LAZARI BENTO

DUE DATE: 08/SEP/2024 AT 11:59PM

1. INTRODUCTION

In this warm-up programming assignment, your task is to write a program that preprocesses a given arborescence $T = (V, A)$ on n vertices in $O(n)$ time, and after that it answers multiple ancestor queries, each one in $O(1)$ time. This assignment will be graded out of 20 marks. The assignment 0 grade will be summed to the grade of assignment 1, graded out of 80 marks, to complete one official grade for assignment 1 out of 100 marks.

Your code must be written in `C++17` and use the BGL library. Please refer to Section 4 for more details.

2. DEFINITIONS

Let us recall the definition of branchings. Let $D = (V, A)$ be a digraph. A subset $B \subseteq A$ is a *branching* if

(i) B has no loops,
(ii) $|\delta^{\text{in}}_B(v)| \leq 1$ for each $v \in V$, and
(iii) each vertex of D is reachable in (V, B) by a root of B;

a *root* of B is a vertex $r \in V$ such that $\delta^{\text{in}}_B(r) = 0$.

An *arborescence* is a branching that has at most one root. If $B \subseteq A$ is an arborescence of the digraph $D = (V, A)$, it is usual to say that the digraph (V, B) is an *arborescence*.

The digraphs in Figure 1 are arborescences.

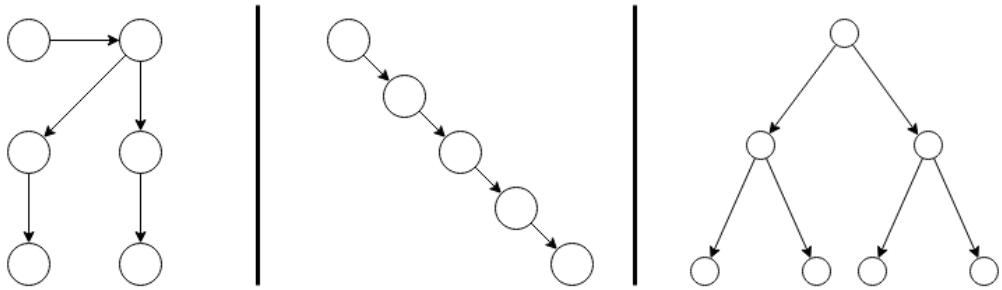

FIGURE 1. Examples of arborescences

INSTITUTO DE MATEMÁTICA E ESTATÍSTICA, UNIVERSIDADE DE SÃO PAULO, R. DO MATÃO 1010, 05508-090, SÃO PAULO, SP
E-mail address: {mksilva,heloisa}@ime.usp.br.

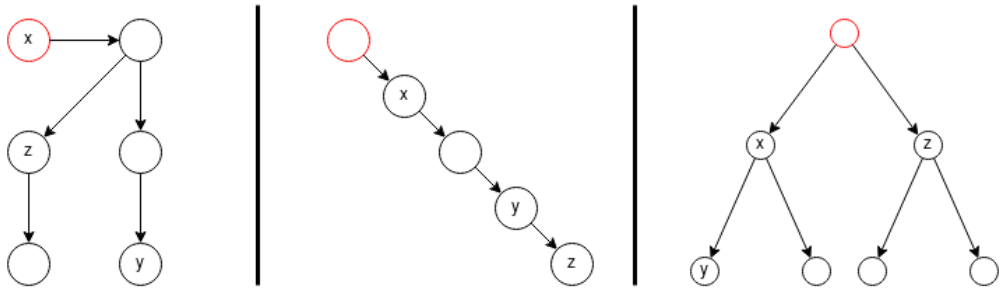

FIGURE 2. Examples of arborescences and ancestrality

Let $T = (V, A)$ be an arborescence, and let $x, y \in V$. Recall that x is an ancestor of y if x is in the **unique** path from the root to y. In Figure 2 we have three examples of arborescences. The roots are marked in red. In all examples, x is an ancestor of y, and z is not an ancestor of y.

3. Test Cases and Grading

Your program should read and preprocess an arborescence from the standard input in $O(n)$ time. After that, the program reads the number of queries, and then it reads and answers each query in $O(1)$ time. Each query consists of two vertices x and y. The program prints `YES` if x is an ancestor of y and `NO` otherwise.

Each test case has the following format:

- The first line has only one integer $1 \leq n \leq 10^5$, the number of vertices of the arborescence. The vertices will be indexed by integers from 1 to n and the root is the vertex 1.
- Each line from the second one to the nth one has two integers, i and j, indicating that ij is an arc of the arborescence. (Your program does not need to verify that the input is an arborescence.)
- The $(n+1)$-st line has only one integer, $1 \leq q \leq 10^5$, the number of queries.
- The following q lines will each contain two integers x and y, such that $x, y \in \{1, \ldots, n\}$, which represents the query "is x an ancestor of y?".

Your code is considered to pass in one test case if it answers ALL queries correctly. There are approximately 50 test cases, and your grade will be proportional to how many test cases your program passes. You will have access to approximately 10 of these 50 test cases, which you may use to debug and test your code.

Warning: Note that even if your code passes all public test cases, it may fail to work on the remaining, "secret" test cases. If the code does not pass all public cases, then it will probably fail in most "secret" cases.

You can find the public test cases in the public repository for the course at https://gitlab.uspdigital.usp.br/mksilva/mac0328-2024-public/-/tree/master/assignments/asgt0/tests.

4. Submission and Implementation Details

In the folder https://gitlab.uspdigital.usp.br/mksilva/mac0328-2024-public/-/tree/master/assignments/asgt0 of the public repository, you will also find 5 template files, which you must use in this assignment. We now describe how the files interact with each other and which ones you should modify.

There is a `Makefile`, which you should adapt to the location of the `boost` library in your computer. It is not recommended to modify the other flags, since they will be used when building the program for grading.

There are two template header files: `asgt.h` and `arb.h`. The header file `asgt.h` **must not be modified**. (In particular, you will not submit it, and we will use the file we distributed to build the program for grading.) The grading driver `main.cpp` interacts with your program **only through the interface specified at** `asgt.h`. For grading purposes, `main.cpp` need not be the same as the one that we distributed.

The functions declared by `asgt.h` must be defined in your `asgt.cpp` file, which you will submit. In the latter file, you may write and call any extra functions other than the ones declared in `asgt.h`. The `read_arb` and `preprocess` functions should run in $O(n)$ time, and `is_ancestor` should run in $O(1)$ time.

The type of the graph that is read and processed is called `Arb`, and it is declared in the header file `arb.h`. In this header file, **nothing must be modified**, except for the declaration of the class `HeadStart`. This class should store whatever information you deem useful to be returned by the `preprocess` function. As per `asgt.h`, this nugget of information (an object of class `HeadStart`) is passed (by reference to `const`) to the query function `is_ancestor`, so that object should give you quick access to what is need in order to answer queries in constant time. In particular, since `preprocess` must run in $O(n)$ time, one cannot for instance store a matrix with the answer to all possible queries.

Your solution most likely will consist of a modification of the **recursive** reachability algorithm covered in class. (More details will be covered in Lecture 06.) Your computing environment may have issues with a low limit for stack size. One of the ways in which this may manifest itself is if your program segfaults, though of course there are a multitude of other reasons that could make your code segfault. The diagnosis may be confirmed by Valgrind. To address this, you may run the command `ulimit -s 256000` in your shell, which ensures a stack size of approximately 256 MB. The computing environment used for grading will have a similarly sized execution stack.

You should submit a compressed archive `NUSP.tar.gz` through Moodle, obviously with `NUSP` replaced by your university ID number. The compressed archive must have precisely two files, inside no directory, namely `asgt.cpp` and `arb.h`.

Failure to follow these instructions exactly will be penalized.