Algorithms and Data Structures II (1DL231)
Uppsala University — Autumn 2022
Assignment 3

Based on assignments by Pierre Flener,
but revised by Frej Knutar Lewander and Justin Pearson

— Deadline: 13:00 on Friday 16 December 2022 —

It is strongly recommended to read the *Submission Instructions* and *Grading Rules* at the end of this document even before attempting to solve the following problems. It is also strongly recommended to prepare and attend the help sessions.
For this assignment you will need the two Python skeleton files `sensitive.py` and `party_seating.py`

Problem 1: Controlling the Maximum Flow

A flow network is a directed graph $G = (V, E)$ with a source s, a sink t, a nonnegative capacity $c(u, v)$ on each edge, and a nonnegative flow $f(u, v)$ on each edge. An edge of a flow network is called *sensitive* if decreasing its capacity always results in a decrease of the maximum flow; in other words, decreasing the capacity of a sensitive edge by a single unit reduces the maximum flow of the network. Perform the following tasks:

A. Design and implement an efficient algorithm as a Python function $\mathit{sensitive}(G, s, t)$ for a flow network G with source s and sink t with maximum flow, where the flow $f(a, b)$ is an integral flow amount over the edge (a, b). Your function should return a sensitive edge (u, v) if and only if one exists. If no sensitive edge exists, then the function should return $(\mathit{None}, \mathit{None})$.

   Two points will be deducted from your score if your algorithm does not exploit the proof (in CLRS3) of the max-flow min-cut theorem or deletes vertices or edges: there is no need to implement the Ford-Fulkerson algorithm, and there is no need to run it twice!

   Note that you are ***not*** given a residual network; if your algorithm requires a residual network for the given flow network G, then you need to compute it first.

B. Compute the worst-case time complexity of your algorithm.

Problem 2: The Party Seating Problem

A number of guests will attend a party. For each guest g, we are given a set $\mathit{known}[g]$ of the other guests known by g; these sets are symmetric in the sense that if guest g_1 knows guest g_2, then g_2 also knows g_1; the sum of the lengths of these lists is denoted by ℓ. To encourage their guests to meet new people, the party organisers would like to seat all of the guests at two tables, arranged in such a way that no guest knows any other guest seated at the same table. We call this the *two-table party seating problem*. Perform the following tasks:

A. Formulate the two-table party seating problem as a graph problem.

B. Design and implement an efficient algorithm as a Python function *party*(*known*) that returns *True* if and only if the two-table party seating problem has a solution. If such an arrangement exists, then the algorithm should also return one, in the form of two sets of guests, namely one set for each table; else the algorithm should return *False* and two empty seating sets.

C. Argue that your algorithm has a time complexity of $\mathcal{O}(|known| + \ell)$.

D. We can modify the party seating problem to more than two tables in the following manner. The party is attended by p groups of guests, and there are q tables. All members of a group know each other; no guest knows anyone outside his or her group. The sizes of the groups are stored in the array *Group*, and the sizes of the tables in the array *Table*. The problem is to determine a seating arrangement, if it exists, such that at most one member of any group is seated at the same table. Give a formulation of this modified party seating problem as a maximum-flow problem. (You do ***not*** need to provide an implemented algorithm for this task.)

Submission Instructions

- Identify the team members and state the team number inside the report and ***all*** code.
- State the problem number and task identifier for each answer in the report.
- Take Part 1 of the demo report at `http://user.it.uu.se/~justin/Hugo/courses/ad2/demorep` as a *strict* guideline for document structure and as an indication of its expected quality of content.
- Comment *each* function according to the AD2 coding convention at `http://user.it.uu.se/~justin/Hugo/courses/ad2/codeconv`.
- Test *each* function against *all* the provided unit tests.
- Write *clear* task answers, source code, and comments: write with the precision that you would expect from a textbook.
- Justify *all* task answers, except where explicitly not required.
- State in the report *all* assumptions you make that are not in this document. Every legally re-used help function of Python can be assumed to have the complexity given in the textbook, even if an analysis of its source code would reveal that it has a worse complexity.
- *Thoroughly* proofread, spellcheck, and grammar-check the report.
- Match *exactly* the uppercase, lowercase, and layout conventions of any filenames and I/O texts imposed by the tasks, as we will process submitted source code automatically.
- Do *not* rename any of the provided skeleton codes, for the same reason.
- Import the commented Python source-code files *also* into the report: for brevity, it is allowed to import only the lines between the copyright notice and the unit tests.

- Produce the report as a *single* file in PDF format; all other formats will be rejected.
- Remember that when submitting you implicitly certify (a) that your report and all its uploaded attachments were produced solely by your team, except where explicitly stated otherwise and clearly referenced, (b) that each teammate can individually explain any part starting from the moment of submitting your report, and (c) that your report and attachments are not freely accessible on a public repository.
- Submit (by only *one* of the teammates) the solution files (one report and up to two Python source-code files) without folder structure and without compression via *Studium*

Grading Rules

For each problem: If the requested source code exists in a file with *exactly* the name of the corresponding skeleton code, ***and*** it imports *only* the libraries imported by the skeleton code, and it runs *without* runtime errors under version 3.6.9 of Python, and it produces correct outputs for *all* the provided unit tests and *some* of our grading tests in *reasonable* time on the Linux computers of the IT department, and it has the comments prescribed by the AD2 coding convention for *all* the functions, and it features a *serious* attempt at algorithm analysis, then you get at least 1 point (read on), otherwise your final score is 0 points. Furthermore:

- If the code has a *reasonable* algorithm, and it *passes most* of our grading tests, and the report addresses *all* the tasks and subtasks, then, your final score is 3 or 4 or 5 points, depending *also* on the quality of the Python source-code comments and the report part for this problem; you are *not* invited to the grading session for this problem.
- If the code has an *unreasonable* algorithm, or it *fails many* of our grading tests, ***or*** the report does *not* address all the tasks and subtasks, then your initial score is 1 or 2 points, depending *also* on the quality of the Python source-code comments and the report part for this problem; you *might be* invited to the grading session for this problem, where you can try and increase your initial score by 1 point into your final score.

However, if the coding convention is insufficiently followed or the assistants figure out a minor fix that is needed to make your code run as per our instructions, then, instead of giving 0 points up front, the assistants may at their discretion deduct 1 point from the score thus earned.

Considering there are three help sessions for each assignment, you must earn at least 3 points (of 10) on each assignment until the end of its grading session, including at least 1 point (of 5) on each problem and at least 15 points (of 30) over all three assignments, in order to pass the *Assignments* part (2 credits) of the course.

For timetable reasons, there is ***no*** solution session for this assignment.