Algorithms and Data Structures II (1DL231)
Uppsala University — Autumn 2024

Assignment 3

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

— Deadline: 13:00 on Thursday 2nd January 2025 —

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.

The submission must be your own work. You are not use generative AI for coding assistance.

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 $sensitive(G, s, t)$ 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 $(None, None)$.

   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.

C. Is each edge that is at capacity (an edge is at capacity if its flow equals its capacity) always sensitive? Why or why not?

D. A real-world example of a flow network is where the edges are roads and the nodes are intersections. The source would be the position of some vehicle and the sink would be the destination of the vehicle. For each edge, the capacity of the edge is the maximum amount of traffic the corresponding road can handle and the flow is the current amount of traffic on the corresponding road. Complete the following sub-tasks:

   (a) Briefly describe what a sensitive edge correspond to in this example.

   (b) Motivate why finding sensitive edges in the network could be of importance in this example.

If you pass only Tasks A and B, then your score is up to 3 points for the Problem. If you pass only Tasks A to C, then your score is up to 4 points for the Problem. If you pass Tasks A to D (and their sub-tasks), then your score is up to 5 points for the Problem.

Problem 2: The Party Seating Problem

A number of guests will attend a party. For each guest g, we are given a set $known[g]$ 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. Hint consider using a bipartite graph.

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 their 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.)

If you pass only Tasks A to C, then your score is up to 3 points for the Problem. If you pass Tasks A to D, then your score is up to 5 points for the Problem.

Submission Instructions

- Identify yourself 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 you, except where explicitly stated otherwise and clearly referenced, (b) that you can 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 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 (for all of the following statements)

- there are no runtime errors when running the code under Python 3 on a Linux computer of the IT department, and
- the code passes *all* of the provided unit tests and *all* of our grading tests,

then your report will be graded for the problem (continue reading). Otherwise, the final score is automatically 0 points for the problem, and the report will not be graded for the problem. Furthermore, if (for all of the following statements)

- the requested source code exists in a file with *exactly* the name of the corresponding skeleton code (except for any characters introduced by studium);
- the code imports *only* the libraries imported by the skeleton code;
- the code has the comments prescribed by the AD2 coding convention for *all* the functions; and
- the source code features a *serious* attempt at algorithm analysis,

Then no points will be deducted from the final score for the problem. Otherwise, points will be deducted from the final score of the problem, at the discretion of the assistant.

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.