Algorithms and Data Structures II (1DL231)
Uppsala University – Autumn 2017
Assignment 3

Prepared by P. Flener, A. Ek, C. Pérez Penichet, H.-P. Vo, and K. Winblad

— Deadline: 13:00 on Friday 2018-01-12 —
— Deadline before grading session: 13:00 on Thursday 2018-01-04 —

It is strongly recommended to read the *Submission Instructions* and *Grading Rules* at the end of this document ***before*** attempting to solve the following problems.

Problem 1: Controlling the Maximum Flow (*mandatory*!)

A flow network is a directed graph $G = (V, E)$ with a source s, a sink t, and a nonnegative capacity $c(u, v)$ 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 flow of a sensitive edge by a single unit reduces the maximum flow of the entire network. Perform the following tasks:

a. Design and implement an efficient algorithm as a Python function $sensitive(G, s, t, F)$ for a flow network G with source s and sink t, and a previously computed maximum flow matrix F, where the element $F[a][b]$ is an integral flow amount over the edge (a, b). Your function should return a sensitive edge (u, v) if and only if such a sensitive edge 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: 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 F, then you need to compute it first. Furthermore, the skeleton generates a test case using the NetworkX function $ford\_fulkerson(G, s, t)$ to compute a maximum-flow matrix for a network G. You are encouraged to use *ford_fulkerson* to generate maximum flows for your own test cases; however, as always, you are ***not*** allowed to use any NetworkX graph functions in your function: if your implementation of the *sensitive* function requires an elementary graph or flow algorithm, then you must implement it yourself. Note that the sanity checker and our grading are based on NetworkX version 1.11, not the very recent version 2.0.

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

Solo teams must perform all the tasks.

Problem 2: Reliable Communications

A communication network consists of a set of stations, which act as transmitters and receivers, and a set of communication channels (e.g., wires) between pairs of stations. The network can be modelled as an undirected graph $G = (V, E)$, where the vertex set V is the set of stations and the edge set E is the set of communication channels. Communication along the channel $(u, v) \in E$ has a failure probability $f(u, v)$, where $0 \leq f(u, v) \leq 1$. For example, if $f(v_1, v_5) = 0.2$, then the probability that a communication along the channel connecting stations v_1 and v_5 will fail is 20%. All these probabilities are ***independent***. In order to let two given stations $u, v \in V$ communicate with each other, a path $\langle u = v_0, v_1, \ldots, v_k = v \rangle$ in G must be chosen. Perform the following tasks:

a. Design and implement an efficient algorithm as a Python function $\mathit{reliable}(G, u, v)$ for the problem of finding a path from u to v with the lowest failure probability. Hint: Consider a path $\langle v_0, v_1, \ldots, v_k \rangle$: since the probabilities of failure on the different edges are independent, the probability that the overall communication between v_0 and v_k fails is given by $1 - (1 - f(v_0, v_1)) \cdot \ldots \cdot (1 - f(v_{k-1}, v_k))$.

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

Solo teams must perform all the tasks.

Note: As the graph G is undirected, you may assume it contains no self-loops. Furthermore, you may assume that (i) the graph is connected, and (ii) $u \neq v$ (that is, the start and end of the path are not the same vertex).

Problem 3: The Party Seating Problem

A number of guests will attend a party. For each guest g, we are given a duplicate-free list $\mathit{Known}[g]$ of the other guests known by g; these lists 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 $\mathit{party}(\mathit{Known})$ 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 duplicate-free lists of guests, namely one list for each table; else the algorithm should return *False* and two empty seating lists.

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

d. We can generalise 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 generalised party seating problem as a maximum flow problem. (You do ***not*** need to provide an implemented algorithm for this task.)

Solo teams may omit task d.

Submission Instructions

All task answers, including documented source code, ***must*** be in a ***single*** report in ***PDF*** format; all other formats are rejected. Furthermore:

- Identify the team members and state the team number inside the report.
- Certify that your report and all its uploaded attachments were produced solely by your team, except where explicitly stated otherwise and clearly referenced, that each teammate can individually explain any part starting from the moment of submitting your report, and that your report and attachments are not freely accessible on a public repository. Without such an honour declaration, your report will ***not*** be graded.
- State the problem number and task identifier for each answer in the report.
- Take Section 1 of the demo report at `http://user.it.uu.se/~pierref/courses/AD2/demoReport` as a ***strict*** guideline for document structure and content, as well as an indication of its expected quality.
- Comment each function according to the AD2 coding convention at the *Student Portal*.
- Write clear task answers, source code, and comments.
- Justify all task answers, except where explicitly not required.
- State any assumptions you make that are not in this document. Any 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 your report.
- Match exactly the uppercase, lowercase, and layout conventions of any filenames and I/O texts imposed by the tasks, as we will process your source code automatically.
- Make a sanity check of all your Python functions on *random* tests via the server at `http://ad2checker.it.uu.se`. Beware: passing random tests does ***not*** guarantee that your code will pass the *boundary* and *stress* tests of our grading test suite (see below), hence you are ***strongly*** encouraged to devise your own *boundary* and *stress* tests.

Only ***one*** of the teammates submits the solution files (one PDF report with answers to ***all*** the tasks, plus up to three commented Python source code files, which are ***also*** imported into the report), ***without*** folder structure and ***without*** compression, via the *Student Portal*, whose clock may differ from yours, by the given ***hard*** deadline.

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 depends only on the libraries imported by the skeleton code, ***and*** it runs without runtime errors under version 2.7.∗ of Python (use `python`), ***and*** it produces correct outputs for 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 at the *Student Portal*, ***and*** it features a serious attempt at algorithm analysis, ***then*** you get at least 1 point (of 5), ***otherwise*** you get 0 points. Furthermore:

- If your function has a ***reasonable*** algorithm and ***passes most*** of our grading tests, ***and*** your report is ***complete***, then you get 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 your function has an ***unreasonable*** algorithm *or* ***fails many*** of our grading tests, ***or*** your report is ***incomplete***, then you get an initial mark of 1 or 2 points, depending also on the quality of the Python source code comments and the report part for this problem; you are invited to the grading session for this problem, where you can try and increase your initial mark by 1 point into your final mark.

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

Considering that there are three help sessions for each assignment, you must get minimum 5 points (of 15) on each assignment until the end of its grading session, including minimum 1 point (of 5) on each mandatory problem, and minimum 23 points (of 45) 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. For reports submitted by the first deadline, which is before the exam, a grading session is scheduled, but you cannot submit again. For reports submitted by the second deadline, which is at the end of period 2, there can***not*** be a grading session, for timetable reasons: each initial score is the final score.