Algorithms and Data Structures II (1DL231)
Uppsala University — Autumn 2020
Assignment 2

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

— Deadline: 13:00 on Monday 29 November 2021 —

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.

Problem 1: Search-String Replacement

For this assignment you will need the two Python skeleton files `difference.py` and `recompute_mst.py`

A useful feature for a search engine is to suggest a replacement string when a search string given by the user is not known to the search engine. In order to suggest and rank replacement strings, the search engine must have some measure of the minimum difference between the given search string and a possible replacement string. For example, over the alphabet $\mathcal{A} = \{\mathrm{A}, \ldots, \mathrm{Z}\}$, let the user's search string be $u =$ DINAMCK and let a suggested replacement string be $r =$ DYNAMIC. The *minimum difference* of strings u and r is the minimum cost of changes transforming u into r, where a *change* is either altering a character in u in order to get the corresponding character in r, or skipping a character in u or r. A *positioning* of two strings is a way of matching them up by writing them in columns, using a dash (–) to indicate that a character is skipped. For example:

D	I	N	A	M	–	C	K
D	Y	N	A	M	I	C	–

The *difference* of a positioning is then the sum of the resemblance costs of the character pairs in each column of the positioning, as given by a resemblance matrix $\mathcal{R}$. For an alphabet $\mathcal{A}$, we have that $\mathcal{R}$ is an $(|\mathcal{A}|+1) \times (|\mathcal{A}|+1)$ matrix, as it must include the dash in addition to the $|\mathcal{A}|$ characters of the alphabet. For example, the positioning above has a difference of:

$$\mathcal{R}[\mathrm{D},\mathrm{D}] + \mathcal{R}[\mathrm{I},\mathrm{Y}] + \mathcal{R}[\mathrm{N},\mathrm{N}] + \mathcal{R}[\mathrm{A},\mathrm{A}] + \mathcal{R}[\mathrm{M},\mathrm{M}] + \mathcal{R}[–,\mathrm{I}] + \mathcal{R}[\mathrm{C},\mathrm{C}] + \mathcal{R}[\mathrm{K},–]$$

For example, if $\mathcal{R}[x, y] = 1$ for all $x, y \in \mathcal{A} \cup \{–\}$ with $x \neq y$ and if $\mathcal{R}[x, x] = 0$ for all $x \in \mathcal{A} \cup \{–\}$, then the difference of a positioning is the *number* of changes; another resemblance matrix could store the Manhattan distances on a QWERTY keyboard between characters of the alphabet (see the skeleton code).

Given two strings u and r of possibly different lengths over an alphabet $\mathcal{A}$ that does not contain the dash character, and given an $(|\mathcal{A}|+1) \times (|\mathcal{A}|+1)$ resemblance matrix $\mathcal{R}$ of integers, which ***cannot*** be assumed to be symmetric, perform the following tasks:

A. Give a recursive equation for a parameterised quantity after stating its meaning in terms of all its parameters (do not rename the problem parameters u, r, $\mathcal{A}$, and $\mathcal{R}$). Use the equation to justify that the problem of computing the minimum difference of u and r relative $\mathcal{R}$ has optimal substructure and overlapping subproblems, so that dynamic programming is applicable to it.

B. Motivate your choice between bottom-up iteration (for which you must argue for the chosen nesting and iteration order of the loops) and top-down recursion, and implement an efficient dynamic programming algorithm for this problem as a Python function $\mathit{min\_difference}(u, r, \mathcal{R})$, assuming that the *last* row and *last* column of $\mathcal{R}$ pertain to the dash character.

C. Extend your algorithm from Task B to return also a positioning for the minimum difference. Implement the extended algorithm as a Python function $\mathit{min\_difference\_align}(u, r, \mathcal{R})$.

D. Argue that the time complexity of your extended algorithm is $\mathcal{O}(|u| \cdot |r|)$.

We are ***not*** implying that search engines actually use such a dynamic programming algorithm for suggesting search-string replacements.

Problem 2: Recomputing a Minimum Spanning Tree

Given a connected, weighted, undirected graph $G = (V, E)$ with ***non-negative*** edge weights, as well as a minimum(-weight) spanning tree $T = (V, E')$ of G, with $E' \subseteq E$, consider the problem of incrementally updating T if the weight of a particular edge $e \in E$ is updated from $w(e)$ to $\widehat{w}(e)$. There are four cases:

1. $e \notin E'$ and $\widehat{w}(e) > w(e)$
2. $e \notin E'$ and $\widehat{w}(e) < w(e)$
3. $e \in E'$ and $\widehat{w}(e) < w(e)$
4. $e \in E'$ and $\widehat{w}(e) > w(e)$

Perform the following tasks:

A. For each of the four cases: describe in plain English with mathematical notation an efficient algorithm for updating the minimum spanning tree, and argue that the algorithm is correct and has a time complexity of $\mathcal{O}(1)$ or $\mathcal{O}(|V|)$ or $\mathcal{O}(|E|)$.

B. For ***at least one*** case that does ***not*** take constant time, say case $i \in 1\,..\,4$, implement your algorithm as a Python function $\mathit{update\_MST\_i}(G, T, e, w)$ for $w = \widehat{w}(e)$. If you give functions for multiple values of i, then indicate which one you want to be graded, else we choose the one with the lowest i.

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.