

MATH1058 coursework [40% of the total marks]

Consider the following problem:

Shortest Path Length (SPL): given a directed graph $G = (V, A)$, two nodes $s, t \in V$, and a length function $\ell : A \rightarrow \mathbb{R}^+$, find the length of a shortest s - t path.

In this coursework, you are asked to write a Python code which implements two variants of Dijkstra's algorithm for the solution of SPL, run your code to produce data on how the two variants behave in terms of computing time as the size of the graph increases, analyze this data in Excel, and, finally, write up your findings in LaTeX. The LaTeX report is conceived to be similar to a research paper, albeit in a smaller scale.

Python

Write a Python script `dijkstra.py` containing two functions, `dijkstra1()` and `dijkstra2()`, capable of computing the length of a shortest path between the first node of the graph (0) and a second node `other_node` supplied as input.

Note that, in this activity, we are only interested in the length of a shortest path, rather than in which arcs it contains. The functions should have two inputs: the successors list of the graph, `successors`, and the destination node `other_node`.

Assume that `successors` is implemented as a list of dictionaries where, for all i in V , `successors[i]` is a dictionary whose keys are the indices of the nodes adjacent to node i and whose values are the lengths of the arcs from i to such nodes.

The function `dijkstra1()` should contain an implementation of Dijkstra's algorithm in the version seen in the lectures (which solves the "Single Source Shortest Path" problem by computing the shortest path length between 0 and each other node in the graph), modified to return, at the very end of the algorithm, the length of a shortest path between $s=0$ and $t=\text{other_node}$.

The function `dijkstra2()` should contain a modified implementation which, after the destination node `other_node` has been reached, halts the execution of the algorithm (this is correct; do you see why?). Its return value should be the same as `dijkstra1()`.

Comparing the practical efficiency of these two versions of the algorithm is the main aim of this activity.

Notes on the implementation

Consider the following example of a successors list

```

successors = [{1 : 2, 2 : 5},
               {2 : 3},
               {3 : 4, 1: 8},
               {0 : 21}]

```

which encodes the graph reported in Figure~1.

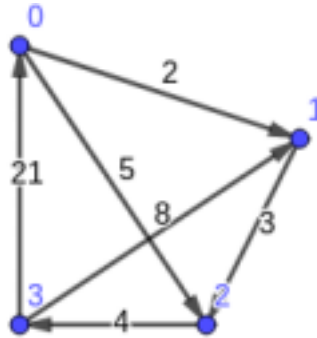


Figure 1: Graph represented by `successors`

As an example of how to navigate a graph encoded with this data structure, given a Python variable `v` equal to the index of a node $v \in V$ of the graph, you can loop over the forward star of `v` (called $\delta^+(\{v\})$ in the lectures) with the following code:

```

for w, lvw in successors[v].items():
    print("node ", w, " is a successor of node ", v)
    print("the length of the corresponding arc is", lvw)

```

Suggestion: compute the shortest paths from node 0 to each other node in the previous graph on a piece of paper, and check in Python whether your implementations produce the correct results. You could also construct a larger graph and test them there.

Data generation

Download the file `graph_generation.py` from Blackboard and import it (with `import graph_generation`) in your script `dijkstra.py`. `graph_generation.py` contains a function `graph_timings()` which, given as input

- the name of a function (written by you) containing an implementation of an algorithm for the solution of the SPL problem to be tested (`dijkstra1()` or `dijkstra2()`)

- your (8 digit) student ID

generates a set of graphs (based on your ID) and runs the supplied implementations (`dijkstra1()` or `dijkstra2()`) on all of them. The function not only runs the supplied implementation, but also measures the amount of microseconds taken to compute the shortest path length between node $s=0$ and each other node $t=\text{other_node}$ in the graph, recording it in an Excel file (see further).

You can run the function on both implementations with:

```
graph_generation.generate_timings(`dijkstra1`, ID)
graph_generation.generate_timings(`dijkstra2`, ID)
```

When running, the function produces an output similar to this one:

```
Your implementation named dijkstra1 will be run on a total of 8 graphs
Working on graph number 1 of size n = |V| = 8 and m = |A| = 12
Working on graph number 2 of size n = |V| = 16 and m = |A| = 56
Working on graph number 3 of size n = |V| = 16 and m = |A| = 24
Working on graph number 4 of size n = |V| = 32 and m = |A| = 112
Working on graph number 5 of size n = |V| = 24 and m = |A| = 36
Working on graph number 6 of size n = |V| = 48 and m = |A| = 168
Working on graph number 7 of size n = |V| = 32 and m = |A| = 48
Working on graph number 8 of size n = |V| = 64 and m = |A| = 224
```

Excel

When called as specified before, `generate_timings()` produces two .xls files: `dijkstra1_data.xls` and `dijkstra2_data.xls`. Each of them features a column for each graph on which the experiments have been run, containing:

- the number of nodes in the graph $n = |V|$
- the number of arcs in the graph $m = |A|$
- exactly $n-1$ rows reporting the computing time (in microseconds) measured when running the implementation you supplied as input on each of the $n-1$ nodes in V different from 0.

You are asked to create a new spreadsheet called `analysis.xls`. To begin with, it should contain the following quantities, for each graph:

- the average time (we will call it “empirical time” from now on, omitting “average”) taken by each of the two implementations of Dijkstra’s algorithm you wrote
- the theoretical complexity of Dijkstra’s algorithm (in the version of the algorithm seen in the lectures), computed as a function of the number of nodes

Analysis

Let y_1 and y_2 be the empirical computing time of one of the two implementations as measured by `generate_timings()` (and then averaged by you on each graph) and let $n = |V|$ be the number of nodes of the graph.

- Part I:
 - You are asked to estimate two functions, namely, $y_1 = c_1 n^{k_1}$ and $y_2 = c_2 n^{k_2}$, corresponding to the empirical computing time of, respectively, `dijkstra1()` and `dijkstra2()`, from the data you have gathered.
 - This can be done by linear (actually, affine) regression. Indeed, for $n, k, c > 0$, $y = cn^k$ implies $\log y = \log c + k \log n$; letting $y' := \log y$ and $n' := \log n$, we deduce $y' = \log c + kn'$; one could, therefore, plot y' versus n' and, then, infer k and $\log c$ (respectively, slope intercept of the affine function $y' = \log c + kn'$) by affine regression.
 - In Excel, slope and intercept of an affine function can be estimated with the functions `SLOPE` and `INTERCEPT`.
- Part II:
 - Analyze the speedup of `dijkstra2()` with respect to `dijkstra1()` as a function of n , defined as: $\frac{y_2}{y_1}$. Does this quantity increase with n ?

LaTeX report

Write a LaTeX document to explain your results and your analysis. The report should contain four sections, plus an abstract and, if necessary, an appendix (see further):

- Abstract [2 marks]:
 - provide a short summary (no more than 10 lines) of the activity
 - the abstract should be fairly general and, in particular, it should not contain any mathematical formulas
- Introduction [5 marks]:
 - formally introduce the problem tackled in this activity
 - briefly describe Dijkstra’s algorithm in the version presented in the lectures to solve the Single Source Shortest Path (SSSP) problem; you are suggested (although it is not mandatory) to include its pseudocode
 - report the algorithm’s complexity and explain how it is deduced
 - clarify the difference between the SSSP problem and the one (SPL) tackled in this activity
- Solution approach and implementation [10 marks]
 - explain how to modify Dijkstra’s algorithm to compute (and return) the length of a single shortest path between a given pair of nodes
 - explain how you implemented the two functions `dijkstra1()` and `dijkstra2()` in Python, reporting a copy of your code and commenting on it; if the two implementations are very similar, you can report the code of one of them only, explaining in words what has to be changed to obtain the other one
 - comment on the differences between the two implementations
 - if you do not have enough space to report your code in full, you can include only a (relevant) snippet in this section, reporting your code (but just your code, without any extra text around it which is not Python code or Python comments) as an appendix; the appendix will not count towards the page limit
- Experimental results and analysis [15 marks]
 - briefly mention which machine the experiments are run on and what version of Python and what libraries you are using (such as, e.g., NetworkX or numpy)
 - explain what experiments you have run and their purpose

- report in a table the theoretical and empirical computing times; the table should have a column for each value of n on which the experiments have been run
- report your observations on how the empirical computing time of the two implementations, compared to the theoretical one, varies with respect to the number of nodes in the graph; in particular, illustrate the empirical complexity in the form $y_1 = c_1 n^{k_1}$ and $y_2 = c_2 n^{k_2}$ you have deduced for the two implementations, and explain how it compares to the theoretical computing time
- comment on the difference between the empirical computing time of `dijkstra1()` and `dijkstra2()`; do the data suggest that the difference in efficiency between the two would change with the size of the instance?
- report two charts produced in Excel comparing, as a function of the number of nodes of the graph, the theoretical complexity of Dijkstra’s algorithm to the empirical computing time of `dijkstra1()` and `dijkstra2()`:
 - * in linear-linear scale
 - * in log-log scale.
- *Caveat*: you should apply suitable scaling factors to the two empirical computing times for the linear-linear chart to be legible—you have computed those factors already though, have you not?
- Conclusions [6 marks]
 - report a brief summary of your findings (no more than 20 lines)
 - you can think of the conclusions as a more informative version of your abstract, providing more detail on your findings
- Bibliography [2 marks]
 - cite any sources you mentioned in your document
 - for instance, when introducing Dijkstra’s algorithm you could cite a textbook, the lecture slides, or even a web source

Length of your report and what to submit

Your latex document should be *no more* than four pages long, excluding the appendix. Reports longer than four pages *will not be marked*. Use the `template.tex` template (accessible on Blackboard), without altering any of its typesetting parameters (font size, margins). You should submit (via Blackboard) four files:

- `dijkstra.py`
- `analysis.xls`
- `report.zip` (a compressed archive containing all the files necessary to compile your report—if you have figures, do not forget to include them)
- `report.pdf`: your report as a PDF file

Style, a few notes

- Write *everything* in first person plural: “we assume, we report on, we analyze” even if there is a single author to your report (you).
- Write *almost everything* in present tense: “we illustrate, we conclude that, we report”, but do switch to present perfect in the conclusions; present perfect sentences do make sense here and there outside the conclusions, such as when introducing the experimental results: “the code has been written in Spyder” or “the experiments have been run on a Unix machine”. The general idea is that what you are writing about is happening *while you are writing about it*.
- Avoid “you” as impersonal subject: use “one can see” or “we can see”, rather than “you can see”.
- In general, if you can write something in first person plural, do so; for instance, use “we observe that” rather than “it can be observed that”: it is you who are making that observation