Project 4: WAN Traffic Engineering

Future Internet

ETH Zürich - Spring Semester 2020-2021

General Future Internet project stipulations:

- This is 1 of 4 projects. Together they account for 50% of your final course grade.
- The projects are not mandatory. No submission for a project will result in the minimum grade for that particular one.
- Always cite and reference appropriately. Do not use other students' code outside of your group.

This particular project stipulations:

- Threshold 1 deadline (by this deadline, you must have done part A and B, and surpassed the 3pt threshold for part C): 24 May 2021 at 15:00.
- Submission deadline: 31 May 2021 at 15:00.
- The project must be done in groups of two.
- You can receive 12.5 points for this project: part A (3pt), part B (3pt), and part C (6.5pt). 8 points is the minimum amount needed to pass this project.
- The programming language is Python.
- Part A and B must be done without using any libraries or other people's code, with the exception of what the course skeleton provides explicitly (*i.e., networkx*, *numpy*, *ortools*, *ortoolslpparser*, *subprocess*, and modules that come along with Python 3 by default).
- In part C you are free to use whatever you want (with proper references/citations and excluding other students' code) to achieve maximum performance.
- After the deadline, you will have an interview. Each group member must be able to individually explain and demo all parts.

1 Introduction

In this project, the goal is to gain an understanding of how wide area network (WAN) traffic engineering is done. A WAN is used to connect users to the Internet. Different users are connected to the network. Each user has its own demands of the network: with whom and how much it wants to communicate. The task of the WAN operator is to take into account the real-time traffic demands from its users and the network topology. It uses an algorithm to route traffic of its users over the topology. This art is called *traffic engineering*. In the first two parts, you are familiarized with two different ways to determine the rates of the flows which satisfy the demands. In the third part, you are tasked with finding the best possible paths.

2 Multi-commodity over-paths flow problem

Input (G, D)**.** The directed graph $G = (V, E)$ consists of $|V|$ nodes and $|E|$ directed edges. The capacity of each edge $e = (i, j)$ is $c(e) = c((i, j)) \geq 0$. There are no duplicate edges (i.e. two edges between the same source and destination). The traffic demand for this project is modeled as a binary demand matrix D. Each entry D_{st} expresses that source node s wants to send to target node t. In other words, $\forall (s, t) \in V \times V : D_{st} = 0 \ \vee \ 1$, and if $s = t$ then $D_{st} = 0$. In the context of this assignment, if $D_{st} = 1$ it means that s wants to send *as much flow as possible* to t.

Output (Π, r)**.** Given (G, D), we want two output variables. First, we want the function $\Pi : (s, t) \in V \times V \rightarrow A \subseteq P_{st}$, which maps each possible source-destination pair to a set of paths, $\Pi(s, t)$ for short. A is a subset of the set P_{st} which contains all possible paths from s to t. A path must be acylic. Second, we want the function $r : \forall (s, t) \in V \times V : p \in \Pi(s, t) \rightarrow \mathbb{R} \geq 0$ which maps each path to a non-negative rate, $r(p)$ for short. The cumulative rate allocation cannot exceed link capacity constraints. In other words, $\forall e \in E : \sum_{p \in \Pi(s,t): e \in p} r(p) \leq c(e)$.

Goal. The amount of flow f_{st} sent from s to t is the sum of the rates of all its paths, in other words $f_{st} = \sum_{p \in \Pi(s,t)} r(p)$. We want to maximize the lowest f_{st} for which $D_{st} = 1$ while abiding by all constraints (*i.e.,* link capacity constraints, non-negative rates).

3 Skeleton and submission

3.1 Skeleton

A student project skeleton can be downloaded at:

```
http://bach02.ethz.ch/wan_te_assignment.tar.gz
```

The file formats are described in §7.

3.2 Hand-in and leaderboard

We created a personal Git repository for each of the groups for the projects:

```
https://gitlab.inf.ethz.ch/COURSE-FI2021/grpGG-fi2021.git
```

In the remainder of this document, we refer to this directory, appended with "/project4", as `yourgitrepo` (*i.e.,* somewhere on your computer `grpGG-fi2021/project4 = yourgitrepo`). The student project skeleton should be directly at the root of this directory. For each of the parts, we ask you to submit your solutions there by pushing. We strongly recommend you use this repository throughout your development cycle. For all parts your solutions are evaluated live, and displayed on a leaderboard for all to awe. The leaderboard is located at:

```
http://bach02.ethz.ch/leaderboard.html
```

3.3 Dependencies

The skeleton we provide depends on the *networkx*, *numpy*, *ortools*, and *ortoolslpparser* modules. Python version 3.7+ is required. They can be installed via:

```
python3 -m pip install networkx
python3 -m pip install numpy
python3 -m pip install ortools
python3 -m pip install git+https://github.com/snkas/python-ortools-lp-parser.git@v1.5.2
```

The module *ortoolslpparser* is maintained by the TAs, as such any feedback is welcome on that module.

4 Part A: Max-min fair rate allocation (3pt)

In this part, we will provide you with the graph G, paths Π, and demands D. We want you to determine the rate allocation r for each path. The rate must be calculated using the max-min fair allocation (MMFA) algorithm. The algorithm is described in section II.B of [2]. Assume that 1 path = 1 flow (for the $s \rightarrow t$ pairs with demand of course). An example MMFA outcome is shown in Fig. 1.

There are in total N different problems. The .graph, .demand and .path files are located at:

`yourgitrepo/ground_truth/input/a/*`.

Please submit your code at:

`yourgitrepo/code/...`

Please submit your N .rate solution files as:

```
yourgitrepo/myself/output/a/rate0.rate
yourgitrepo/myself/output/a/rate1.rate
...
yourgitrepo/myself/output/a/rate{N − 1}.rate
```

You can view whether your solution is correct by executing:

```
cd yourgitrepo/code; python evaluator_myself.py
```

There are 40 total test cases, which are split up in 'public' and 'hidden'. You only receive the ground truth for the public first 20 ($s \in \{0, 1, ..., 19\}$). The outcome for the hidden 20 ($s \in \{20, 21, ..., 39\}$) are only shown on the leaderboard. These hidden test cases are generated using the same procedure as $s \in \{5, 6, ..., 19\}$ – so if you get 20/20 locally you should get 40/40 on the leaderboard.

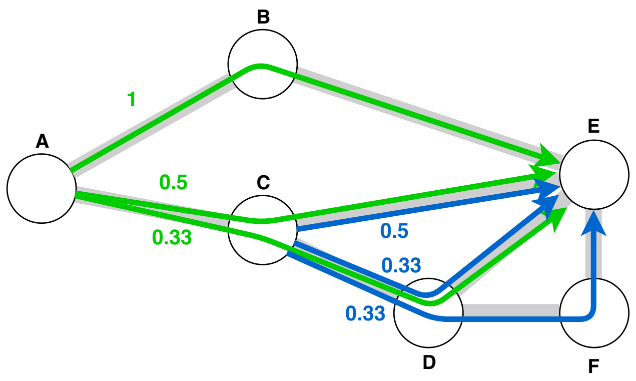

Figure 1: Max-min fair rate allocation of the paths (3 paths/node pair) (two demands: $A \to E, C \to E$).

5 Part B: Optimal rate allocation through a linear program (3pt)

In this part, we will again provide you with G, Π and D. We want you to optimally maximize the minimum amount of flow demand (described more elaborately in §2) with already-provided paths: write a linear program which optimizes this goal. Inspiration for the linear program can be taken from Table 2 of [1]. The linear program in that paper is *similar* to what you should write. Your program should maximize a barrier variable Z and have three types of constraints: (1) for each demand the sum of its path rates must be larger than the barrier, (2) no link capacities are exceeded by the sum of the path rates on them, and (3) path rates must be positive (≥ 0).

An example optimal outcome is shown in Fig. 2. The input and submission is the same as for part A, except that now you should do it in:

```
yourgitrepo/code/...
yourgitrepo/ground_truth/input/b/...
yourgitrepo/myself/output/b/...
```

You can view whether your solution is correct by executing:

```
cd yourgitrepo/code; python evaluator_myself.py
```

There are 40 total test cases, which are split up in 'public' and 'hidden'. The procedure is the same as for part A.

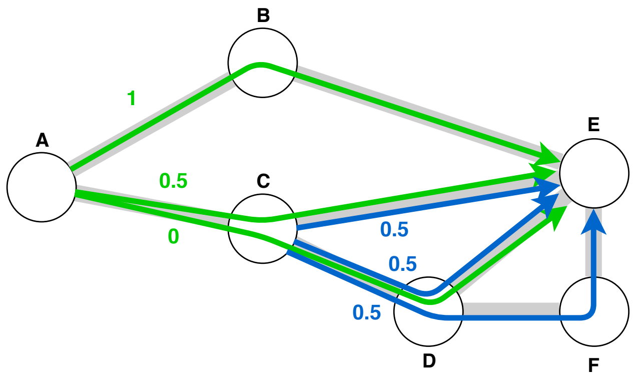

Figure 2: Optimal rate allocation of the paths (3 paths/node pair) (two demands: $A \to E, C \to E$).

6 Part C: Finding the best paths (6.5pt)

In this part, we provide you with G and D. We want you to generate both the paths Π and the rate allocation for those paths r. The rate allocation linear program must be taken over from part B (*i.e.,* you have optimal rate allocation). In this part, we restrict G and D:

- Graph G is an undirected random regular graph of $n = 120$ nodes and degree $d = 8$.
- Demand matrix D is a full random reciprocated permutation pairing. For example, for $n = 6$, a possible outcome is: $(1, 3); (3, 1); (2, 5); (5, 2); (0, 4); (4, 0)$ ($D_{s,t}$ is 1 for these).

Your goal is to provide a set of optimal paths for each pair of nodes *for the class of full random reciprocated permutation pairing demand matrices, not for a specific instance of D*. As such, **you must generate paths between all** $n \times (n - 1)$ **node pairs**. The amount of paths per (src, dst)-pair you are allowed to generate at most is defined by $k = 10$.

A good starting point is exploring the functions `networkx` already provides out of the box and to go from there (*e.g.,* all_shortest_paths, all_simple_paths, shortest_simple_paths). Think about (and possibly, research or empirically analyze) what path diversity a random regular graph (a.k.a. Jellyfish) offers and how a permutation traffic pairing should make use of that path diversity.

The testing is done using 50 random regular graphs we provide, identified by $s \in \{0, 1, ..., 49\}$. The names of the graph files are:

`yourgitrepo/ground_truth/input/c/graph{`s`}.graph`

There are two scores:

- **C-public**: Your generated paths of the 50 graphs are used with the one example demand matrix we supply:

  `yourgitrepo/ground_truth/input/c/demand.demand`

  Which you can view yourself by executing:

  ```
  cd yourgitrepo/code; python evaluator_myself.py
  ```

- **C-hidden**: Your generated paths of the 50 graphs will be used on 50 hidden full random reciprocated permutation pairing demand matrices. On the server, we calculate the optimal split across your paths for these demand matrix instances using (our own implementation of) the linear program. This is computationally expensive, as such this score is only updated at a long time interval (as of writing, approximately 30 minutes).

In conclusion, your output files should be:

`yourgitrepo/myself/output/c/rate{`s`}.rate` (used only for C-public)
`yourgitrepo/myself/output/c/path{`s`}.path` (used for both C-public and C-hidden)

A score is the arithmetic mean of the goal defined in §2 among all 50 scenarios. Your goal is to reach the highest C-hidden score on the leaderboard located at:

`http://bach02.ethz.ch/leaderboard.html`

We supply you with...

- `yourgitrepo/code/generator_random_regular_graph.py`
- `yourgitrepo/code/generator_random_permutation_pairing.py`

... to generate more test cases yourself if you want to.

Be sure to write the your most original team name in the `yourgitrepo/team_name.txt`. It will be displayed publicly on the leaderboard. Secondly, you must write an explanation of your algorithm in `yourgitrepo/README.md`.

7 File formats

Both in the provided project data and in the handed-in data, the following file formats must be adhered to. Nodes are identified by an integer $0 \leq i < N$. Rates and capacities are real positive numbers.

Graph G (*.graph). New-line-separated text file with a description of a directed edge on each line:

```
[src],[dst],[capacity]
```

There are no two edges with the same source and destination. The link capacity is greater than 0.

Demand matrix D (*.demand). New-line-separated text file with a description of a (unit) demand on each line:

```
[src],[dst]
```

There are no duplicate demand lines.

Paths Π (*.path). New-line-separated text file with a path on each line:

```
[src]-[a]-[b]-...-[z]-[dst]
```

There are no duplicate path lines. There are no cyclic paths. There must be exactly as many paths as rates defined in the corresponding rate file.

Rate allocation r (*.rate). New-line-separated text file with a rate allocation for each path on each line:

```
[rate]
```

There must be exactly as many rates defined as paths in the corresponding path file.

8 Frequently asked questions (FAQ)

Why does the relative import in the skeleton fail?

If you get the following error when executing the skeleton:

```
Traceback (most recent call last):
File "skeleton_a.py", line 24, in <module>
from . import wanteutility
ValueError: Attempted relative import in non-package
```

You should be using `python3` (Python 3).

Is it normal that there are so many zeros in the .rate file I generate?

Yes. The .rate file encodes the rate sent over each path. Only a subset of the paths are for the specific demand instantiation. For the paths not involved, the rate is logically zero.

Is the capacity of a bi-directional edge shared between the two uni-directions?

No. A bi-directional edge $a \leftrightarrow b$ with capacity c is equivalent to two directional edges $a \to b, b \to a$ with their own independent capacities $c_{a\to b} = c, c_{b\to a} = c$.

For the C-hidden score, do you run our Python code?

No. We ask you to provide all paths for a set of graphs. We then use our own implementation of part B to calculate the optimal rate distribution for a hidden set of demand matrices. As such, we never have to run your Python code. This means that you can do part C without having done part A or B. The amount of points you are awarded for part C is based on your C-hidden score.

Why is my C-hidden score 0, whereas my C-public score is not?

Extremely likely because you only provided paths for the pairs mentioned in the single demand file we give you as an example in part C. You must provide paths for all pairs of nodes.

Any tips on how to speed up my algorithm?

Python supports easily implementable parallelism: `https://chriskiehl.com/article/parallelism-in-one-line`. You could also consider caching the result of unchanging compute-intensive tasks across runs (*e.g.*, the generation of the initial set of candidate paths).

References

[1] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *USENIX NSDI*, 2018.

[2] Dritan Nace, Nhat-Linh Doan, Eric Gourdin, and Bernard Liau. Computing optimal max-min fair resource allocation for elastic flows. *IEEE/ACM Transactions on Networking (TON)*, 14(6):1272–1281, 2006.