Implementation Project: Computing maximum flow using the MWU method

CS-E3190 - Principles of Algorithmic Techniques

Serhii Korzh (802172)

December 31, 2019

List of resources

I have used the following sources during the writing of my source code:

1. **Wikipedia article on the MWU method**
   I got acquainted with the general idea, history and typical use cases of the Multiplicative Weight Update method here.

2. **"The Multiplicative Weights Update Method: a Meta Algorithm and Applications" survey by Sanjeev Arora, Elad Hazan, and Satyen Kale**
   I used the survey listed in the implementation project page in to get myself familiar with their generalised approach to the MWU method. Particulary, I analysed Sections 1 and 2, as well as Subsections 3.2 and 3.3, where the authors described how MWU can be used to solve Linear Programs, specifically, packing and covering LPs, and how the Multicommodity Flow Problem can be represented by them.

3. **"Faster and Simpler Algorithms for Multicommodity Flow and other Fractional Packing Problems" paper by Naveen Garg and Jochen Könemann**
   The paper is referenced in my previous resource as the original source of an algorithm for solving the Multicommodity Flow problem. It describes the algorithm in greater detail, yet not explicitly using the MWU method. My solution uses a simplified version adapted to the maximum flow problem with an explicit weight update step based on the MWU method described in the survey.

Project Manual

Execution instructions

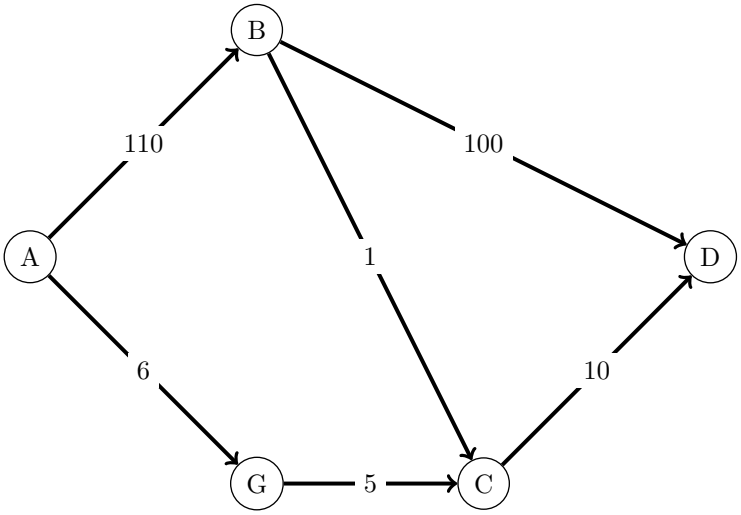
The source code is written in Python, version 3.7 was used for development and testing, but other 3.x versions should execute the program properly. The code is located in `src/main.py` file. A virtualisation and packaging tool Pipenv was used. Provided the machine has Python 3.7 and `pipenv`, it is suffitient to run the `pipenv install` to build the project and `pipenv run python src/main.py` to run the program. The flow network is specified in a JSON format and is placed in the `network.json` file (`--file` command line argument must be used for a different file location). The ϵ parameter of the MWU method can be passed through the `--eps` argument (a float number $\left[0, \frac{1}{2}\right]$), which is set to $\frac{1}{10}$ by default.

Running time analysis

Lower values of ϵ result in better approximation precision, though, greater running time – the complexity of the algorithm is $\mathcal{O}(\frac{E \log E}{\epsilon^2} T_{sp})$, where T_sp is the complexity of the shortest path subroutine used in the algorithm (in my implementation, the Bellman-Ford algorithm is used which has the complexity of $\mathcal{O}(VE)$; E denotes the number of edges in the graph and V denotes the number of vertices and E – the number of edges of the graph representing the flow network. Note that more efficient shortest path algorithms can significantly speed up the program (e.g. Dijkstra with Fibonacci heaps), however, the exact implementation of the shortest path subroutine is nonessential to the working of the max flow algorithm.

Running examples

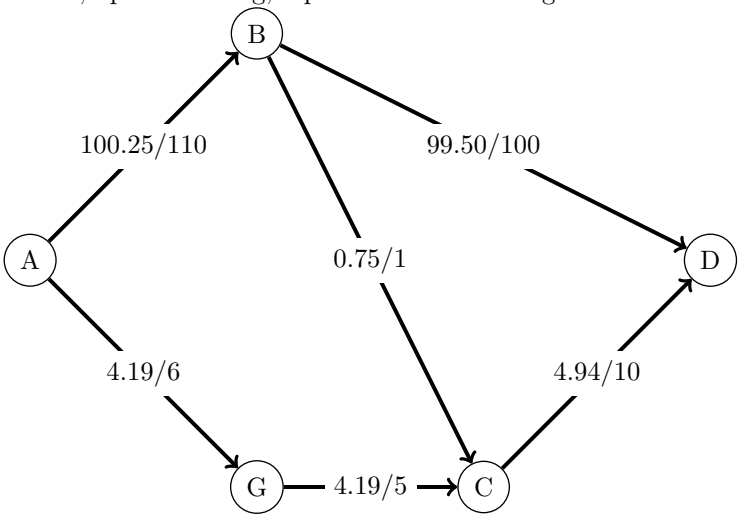
A sample input included with the project represents the following flow network:



The following is the output of the program with epsilon set to 0.1:

```
(Approximately) optimal flow value: 104.44
Flow:
{('A', 'B'): 100.24624420864615,
 ('A', 'G'): 4.192078934561482,
 ('B', 'C'): 0.747503231704939,
 ('B', 'D'): 99.4987409769412,
 ('C', 'D'): 4.939582166266422,
 ('G', 'C'): 4.192078934561482}
```

Which, upon rounding, represents the following flow:



Given the actual optimal flow value of 106, the algorithm result approximates it well.