Columbia Journal of Mathematics

Multiplicative Weights: An Elegant Application to Maximum Flow

By Chon Hou (Jophy) Ye

Abstract

Multiplicative weights is a class of meta-algorithms commonly found in learning theory. It typically carries out several rounds of queries to oracles/agents, each time learning weights in an online manner given feedback from the current system to capture proficiency of them. It has found its use in game theory, machine learning, fast algorithms for optimization, etc.

We begin with the classical example of multiplicative weights weighted majority. In the second part, we will see a delicate usage of multiplicative weights for approximating the maximum network flow, a well-known problem in theoretical computer science with many practical usages, accompanied by visualization from our simulation. The algorithm approximates maximum flow by repeatedly solving a related, computationally easier problem, the electrical flow of a circuit, whose parameters are derived from multiplicative weights. Multiplicative weights come in to adjust the resistances of the circuit online so that edge capacities are gradually obeyed. It is an elegant piece of work drawing insights from learning theory, physics, numerical methods, and theoretical computer science.

This journal is intended for undergraduate readers broadly interested in mathematics and theoretical computer science, who have developed some mathematical maturity and are familiar with basic algorithms.

1. Weighted Majority Algorithm

Consider the following classic problem from learning theory.

Prediction from Expert Advice Predict a series of booleans $\hat{b}_1, \hat{b}_2, \ldots$ (e.g., prediction of the daily movement of a stock, real-time yes/no contest), given n *experts* who each has an opinion every round. In round k, the decision maker (*learner*) decides $\hat{b}_k$ by weighing in all the experts' suggestions $s_{k1}, s_{k2}, \ldots, s_{kn} \in \{0, 1\}$ in an *online* (on-the-fly) fashion, possibly by considering their performance in previous rounds, but without any prior knowledge about the experts.

Received by the editors September 22, 2025.

© 2025 Ye, Chon Hou. This is an open access article distributed under the terms of the Creative Commons BY-NC-ND 4.0 license.

An example is that Einstein is in a crowd of undergraduate physics major students. Despite not knowing who Einstein is, it is possible to aggregate binary advices from the crowd on the fly every round to closely match Einstein's wisdom.

Algorithm (Weighted Majority with Randomized Rounding [6][7]). Fix a hyperparameter $\eta > 0$ (think of the learning rate). Initialize a multiplicative weight $w_i^{(0)} \leftarrow 1$ for each expert[1]. For round $t = 1, 2, \ldots, T$,

(1) Select expert i with probability $\propto w_i^{(t-1)}$ and follow their advice, more specifically, with the (normalized) distribution $\{w_1/W, \ldots, w_n/W\}$ over experts, where $W^{(t)} = \left\|\mathbf{w^{(t)}}\right\|_1 = \sum_i w_i^{(t)}$ is the L^1 norm of $\mathbf{w}^{(t)}$.
(2) For any expert i that turns out to be incorrect, scale down its weight: $w_i^{(t)} \leftarrow (1-\eta) w_i^{(t-1)}$; otherwise, keep it unchanged.

In essence, w_i is our current "confidence" in expert i, which we update throughout the game, and use to probabilistically choose experts. An expert is penalized (multiplicatively) when they make a mistake (whether chosen or not). However, it may not be obvious that such a meta-algorithm is very competitive.

Theorem. Let m^*, m be the number of mistakes that the optimal expert and the decision maker make, respectively. Then,

$$\mathrm{E}[m] - m^* \leq \eta m^* + \frac{\ln n}{\eta}.$$

Proof. Let $M_i^{(t)}$ be the indicator variable that expert i is wrong in the t-th round, and $M^{(t)}$ be the indicator that we, the decision maker, are wrong. Then,
(1)

$$W^{(t)} = \sum_{i=1}^{n} (1 - \eta M_i^{(t)}) w_i^{(t-1)} = (1 - \eta \,\mathrm{E}[M^{(t)}])\, W^{(t-1)} \leq \exp(-\eta \,\mathrm{E}[M^{(t)}])\, W^{(t-1)},$$

where we used $1 - \epsilon x \leq e^{-\epsilon x}$ for $x > 0$ for the last inequality. As such,

$$W^{(t)} = \exp\left(-\eta \sum_{t'=1}^{t} \mathrm{E}[M^{(t')}]\right) W^{(0)} \leq \exp(-\eta \,\mathrm{E}[m])\, W^{(0)} = \exp(-\eta \,\mathrm{E}[m])\, n.$$

Since $W^{(t)}$ *is at least the best expert's* $w_i^{(t)}$,

$$(1-\eta)^{m^*} \leq \max_i w_i^{(t)} \leq W^{(t)} \leq e^{-\eta \,\mathrm{E}[m]} n. \tag{2}$$

With some more elementary algebra and inequalities, we can show the theorem (see Appendix A). This model of analysis will appear again in the next section. □

[1]Notation-wise, the superscript in $?^{(i)}$ always denotes the value of a variable ? on iteration i as we may constantly update it.

It implies

$$\frac{\mathrm{E}[m] - m^*}{T} \leq \eta + O\left(\frac{1}{T}\right),$$

since $m^* \leq T$. We can make T large enough so that the left-hand side, i.e., the percentage of expected additional mistakes of the learner (compared to the best expert), can be arbitrarily close to η, which is a hyperparameter we can freely control (decrease)[2].

Note the power of this result: the experts are individual *black boxes*; they may have unbounded compute resources, may be probabilistic, could be omniscient or incompetent, or can be extremely competitive for the first half and give wrong answers for the second half (basically anything). Even if *we have no prior knowledge of the best expert (or it may not even be fixed) before the end*, weighted majority always matches the best performing expert as $T \to \infty$.

2. **Maximum Flow Approximation**

In this section, we will employ multiplicative weights to approximate maximum flow with a sequence of electrical flows, which is easier to compute exactly or approximate. The algorithm repeatedly solves for the electrical flow of networks with different resistances that are dynamically adjusted by the weights, and takes the mean. I will present a *simplified* version of the work by P. Christiano, J. A. Kelner, A. Mądry, D. Spielman, S.-H. Teng (2010) [1], which marks the beginning of a line of approximation-with-electric-flow research for maximum flow and has prompted improvements since 1998 [2].

2.1. *Maximum Flow.* We study maximum flow on an *undirected* graph (or *network*) $G = (V, E)$, with a source vertex s and a sink vertex t. We wish the compute a *flow* f, which assigns some units of flow along every (directed) pair of neighbors u, v ($\{u, v\} \in E$) s.t. [6]

- **Edge Capacity** Every edge $e \in E$ has a capacity $u_e > 0$ to obey: $|f_e| \leq u_e$.
- **Flow Conservation** For all vertex $v \neq s, t$,

$$\text{(out-flow from } v\text{)} \qquad \sum_{\{v,u\} \in E} f_{v,u} = \sum_{\{u,v\} \in E} f_{u,v} \qquad \text{(in-flow to } v\text{)}.$$

- **Maximum Flow Value** *Flow value* (net flow through the network)

$$F = \sum_{\{s,v\} \in E} f_{s,v} = \sum_{\{v,t\} \in E} f_{v,t}$$

is maximized. Denote by F^* the optimal flow value of the given network.

[2]The caveat is that convergence takes longer since bad-performing experts are discounted less.

Maximum flow can capture, e.g., the maximum rate of water flow through pipes of various sizes, and the maximum speed of file upload through packet switching between a network of servers. Consider the following network, with source $s = 0$ and sink $t = 5$.

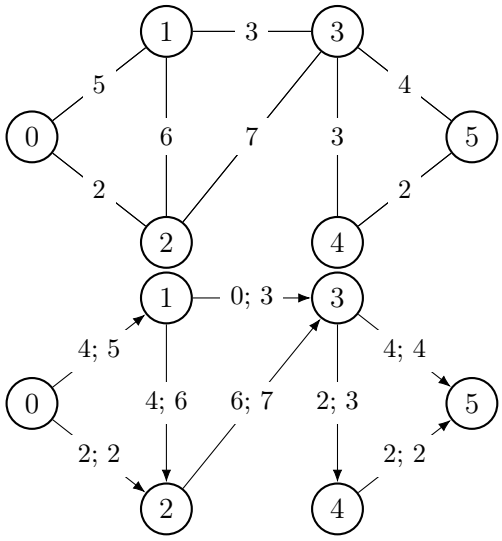

Figure 1. Example Network with its Maximum Flow
The notation $x; y$ stands for x units of flow through an edge with capacity y.

The optimal flow value is 6, achievable by sending 4 units through $(0, 1, 2, 3, 5)$ and 2 units through $(0, 2, 3, 4, 5)$.

Lemma. The (edge-wise) sum of two s-t flows $f_1 + f_2$ is a feasible flow, except that the capacity constraints may not be satisfied.

Certain problems are not easy to compute exactly, but admit algorithms for approximation to arbitary precision. The Ford-Fulkerson (1956) [5] algorithm is a classic greedy algorithm for maximum flow, which repeatedly pushes flow through paths from s to t. In fact, in many settings, sufficiently-close approximation to the optimum is useful enough. In [1], an $(1 - O(\epsilon))$-approximation[3] algorithm running in $\tilde{O}(nm^{1/3}\epsilon^{-11/3})$ time[4] is constructed, with some more technicality than presented here.

2.2. *Electrical Flow.* *Electrical flow* is a related notion of flow. View G as an electric network where each edge $e \in E$ is assigned a *resistance* $r_e > 0$, which is collected into a vector $\mathbf{r} \in \mathbb{R}^m$.

[3]The produced approximation only falls short of the optimum by at most the optimum's $O(\epsilon)$ fraction.

[4]The higher the precision one wants, the longer the runtime. From an applied perspective, it can be adjusted based on the application.

For any s-t flow f, define its *energy with respect to* $\mathbf{r}$ as $E_{\mathbf{r}}(f) = \sum_e r_e f(e)^2$. For now, edge capacity constraints need not be satisfied for electrical flow[5]. By the laws of physics, the *electrical flow* (*current*) of value F is the *unique* s-t flow minimizing energy.

The unique electrical flow of a given network can be computed "quickly" by solving a linear system, unlike maximum flow, which is solvable with linear programming.

Note: *The resistances will be chosen so that after computing electrical flows* $f^{(1)}, \ldots, f^{(N)}$, *the mean* $(f^{(1)} + \cdots + f^{(N)})/N$ *is a feasible* s-t *flow satisfying capacity constraints.*

Intuition

It makes sense that state-of-the-art electrical flow algorithms are more efficient than maximum flow's. In physics terms, the uniqueness flow of electrical flow is implied by *Kirchhoff's current & voltage laws*[a]. Maximum flow, unlike electrical flow, isn't uniquely defined by the network. Also, from a numerical method perspective, linear programming optimization is harder than solving linear systems.

[a] which is how we solve for it with linear systems

2.3. *Multiplicative Weights*. Let us define the *congestion* of an $e \in E$ (with respect to a flow f) to be

$$\operatorname{cong}_f(e) := \frac{|f_e|}{u_e}.$$

Note that an s-t flow is feasible if and only if all $\operatorname{cong}_f(e) \leq 1$. First, assume we have the following oracle to invoke, which we will construct in the next section.

(ϵ, ρ)-oracle For parameters $\epsilon, \rho > 0$, when given the optimal flow value F^*[6] and a set of multiplicative weights collected into a vector $\mathbf{w} \geq \mathbf{1}$ ($w_e \geq 1 \ \forall e \in E$), an (ϵ, ρ)-oracle computes an s-t flow f satisfying:

(1) $|f| = F^*$
In other words, flow value is optimal.

(2) $\operatorname{cong}_f(e) \leq \rho, \forall e \in E$ (*local*)
In other words, flow capacities may be temporarily violated but still obeyed up to a factor of ρ.

(3) $\sum_e w_e \operatorname{cong}_f(e) \leq (1+\epsilon) \sum_e w_e$ (*global*)
Together, the flow satisfies a (much better than 2, i.e., $\epsilon \ll \rho$) *weighted average* of flow constraint.

[5] Since it is a flow, flow conservation must be obeyed.

[6] It would be odd to assume knowledge of F^* and only compute the actual flow. Indeed, the paper performs binary search for a $(1 - O(\epsilon))$-approximation of F^*; the oracle can return such flow if the given target is a lower bound of F^*. Again, this is a simplification made to capture the gist.

Intuition

Therefore, for our successive electrical flows, certain constraints may be violated moderately. However, in a weighted perspective, the flow along edges together satisfies a tighter bound (3).

When weight w_e is high, the oracle is more inclined towards decreasing $\operatorname{cong}_f(e)$. This "put a large fraction of the weight on the violated constraints" [1].

When $\operatorname{cong}_f(e) > 1$, we hope less current flows through e in future electrical flows, so we increase w_e "more".

Algorithm (Multiplicative Weight Portion). Initialize $w_e^{(0)} = 1$ for all edges e, and the number of rounds $N = 2\rho \ln m/\epsilon^2$. For round $t = 1, 2, \ldots, N$,

(1) Use the oracle to compute a s-t flow $f^{(t)}$ with weights $\mathbf{w}^{(t-1)}$ and F^*.
(2) Update weights:

$$w_e^{(t)} = w_e^{(t-1)} \left(1 + \frac{\epsilon}{\rho} \operatorname{cong}_{f^{(t)}}(e)\right).$$

Finally, return the average (scaled) flow $\overline{f} = \frac{(1-\epsilon)^2}{(1+\epsilon)N} \sum_i f^{(i)}$.

2.4. *Electrical Flow Oracle Construction.* In [1], an $(\epsilon, 3\sqrt{m/\epsilon})$-oracle ($0 < \epsilon < 1/2$) was presented. A rough sketch of the construction is to set resistances

$$r_e := \frac{1}{u_e^2}\left(w_e + \frac{\epsilon \|\mathbf{w}\|_1}{3m}\right)$$

and compute the unique electrical flow of the network[7]. The proof can be found in the paper.

Intuition

It makes sense that $r_e \propto 1/u_e$, since the higher the capacity, flow through e can potentially increase, which is captured by a decrease in resistance.

r_e also strictly increases with w_e, which effectively hinders flow when the edge is heavily penalized.

2.5. *Convergence of Multiplicative Weights.* Given such oracle, we proceed to study the convergence of averaged electrical flow. Jump to section 2.6 for a case study with visualization.

Lemma. Let $W^{(t)} = \left\|\mathbf{w}^{(t)}\right\|_1 = \sum_e w_e^{(t)}$, which strictly increases with t. For all $t \geq 1$,

$$W^{(t)} \leq W^{(t-1)} \exp\left(\frac{(1+\epsilon)\epsilon}{\rho}\right).$$

[7]Since we are already in the approximation regime, the paper does not exactly compute, but approximate, each electrical flow to a certain precision, which has performance benefits.

In particular, $W^{(N)} \leq m \exp\left(\frac{(1+\epsilon)\epsilon}{\rho} N\right)$.

Proof. By constraint 3 of an (ϵ, ρ)-oracle,

$$W^{(t)} = \sum_e w_e^{(t)} = \sum_e w_e^{(t-1)} \left(1 + \frac{\epsilon}{\rho} \text{cong}_{f^{(t)}}(e)\right) \leq W^{(t-1)} + \frac{\epsilon}{\rho}(1+\epsilon) W^{(t-1)}$$

$(1 + x \leq e^x)$

$$\leq W^{(t-1)} \exp\left(\frac{(1+\epsilon)\epsilon}{\rho}\right).$$

□

Lemma. For any edge $e \in E$,

$$w_e^{(N)} \geq \exp\left(\frac{(1+\epsilon)\epsilon N}{(1-\epsilon)\rho} \text{cong}_{\bar{f}}(e)\right).$$

Intuition

Note that the lower bound increases exponentially with average congestion $\text{cong}_{\bar{f}}(e)$ and N, which matches the multiplicative update nature of $\mathbf{w}$.

Proof. More generally, for round $t \geq 0$,

$$w_e^{(t)} = \prod_{j=1}^{t} \left(1 + \frac{\epsilon}{\rho} \text{cong}_{f^{(j)}}(e)\right)$$

$$\geq \prod_{j=1}^{t} \exp\left(\frac{(1-\epsilon)\epsilon}{\rho} \text{cong}_{f^{(j)}}(e)\right) = \exp\left(\frac{(1-\epsilon)\epsilon}{\rho} \sum_{j=1}^{t} \text{cong}_{f^{(j)}}(e)\right)$$

by the inequality $\exp((1-\epsilon)\epsilon x) \leq 1 + \epsilon x$ for $x \in [0,1]$ and $\epsilon \in (0, 1/2)$ $(\text{cong}_{f^{(j)}}(e) \leq \rho)$. Plugging in the scaling coefficient for $\overline{f}$ (section 2.3) yields the result. □

Theorem. $\overline{f}$ is a feasible s-t flow with $|\overline{f}| \geq (1 - O(\epsilon))F^*$.

Proof. By the two lemmas,

$$\exp\left(\frac{(1+\epsilon)\epsilon N}{(1-\epsilon)\rho} \text{cong}_{\bar{f}}(e)\right) \leq w_e^{(N)} \underset{(\star)}{\leq} \left\|\mathbf{w}^{(N)}\right\|_1 = W^{(N)} \leq m \exp\left(\frac{(1+\epsilon)\epsilon N}{\rho}\right).$$

Finally, feasibility!

$$\text{cong}_{\bar{f}}(e) \leq (1-\epsilon) + \frac{(1-\epsilon)\rho}{(1+\epsilon)\epsilon N} \ln m = 1 - \epsilon + \frac{\epsilon(1-\epsilon)}{2(1+\epsilon)} \leq 1.$$

Since each electric flow $f^{(i)}$ has value F^* (owing to the oracle), the flow value of $\overline{f}$ is

$$|\overline{f}| = \underbrace{\frac{(1-\epsilon)^2}{1+\epsilon}}_{\text{scaling coefficient}} F^* \geq (1 - O(\epsilon))\, F^*.$$

□

Intuition

Note the similarity in analysis between these 2 multiplicative weight applications. Comparing with inequality (2),

(1) ($\star$) Sum of weights is used to upper bound every individual weight.
(2) Certain weights $w^{(N)}$ is lower-bounded by a *local* "terribleness" of the past rounds.
    In weighted majority, it is bounded by the best performance of experts (ineq. 1); here, it is bounded by the congestion of individual edges, which is then "locally" bounded by constraint 2.
(3) $W^{(N)}$ is upper-bounded by a *global* "terribleness".
    In weighted majority, it is bounded by the learner, who aggregates expert opinions; here, it is bounded by the "global" constraint 3 of an (ϵ, ρ)-oracle.

2.6. *Visualization*. Consider again the network in 2.1 to picture how the algorithm works. I simulated the simplified[8] algorithm for 111,595 rounds of multiplicative weights with $\epsilon = 10^{-2}$ and plotted some flows. The code is attached in the Appendix A.

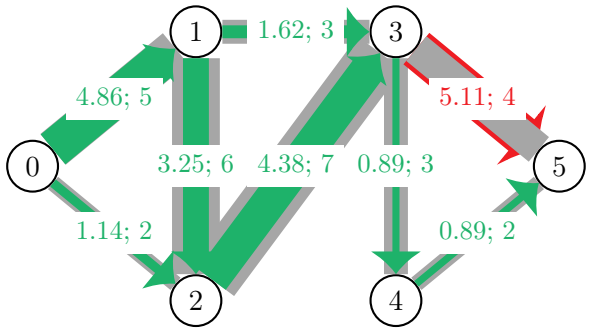

Figure 2. $f^{(1)}$: #1 electrical flow after round 1
Think of water flow through pipes. Grey regions indicate the size (capacity) of the pipes. Green represents flow satisfying the capacity constraint, while red shows violating flow.
The width of an edge is proportional to the flow/capacity.

After round 1, $f^{(1)}$ is as shown in figure 2. The flow along most edges is below the respective capacity (though not yet saturated), with the exception of $\{3, 5\}$, whose flow 5.11 is over capacity 4.

[8]Electric flow is exactly computed, and the optimal F^* is provided rather than carrying out binary search (just for the purpose of visualization).

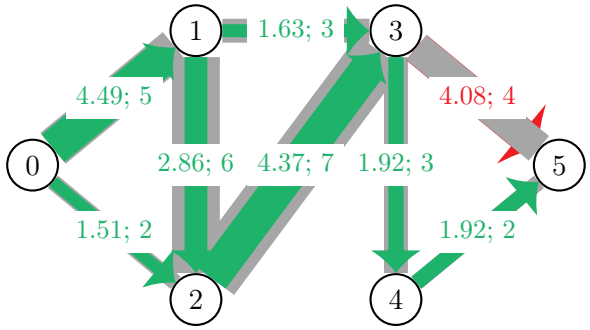

Figure 3. $f^{(20,000)}$: #1 electrical flow after round $20,000$

After $50,000$ rounds, flow on the left-hand-side changes slightly, whereas the flow on $\{3,5\}$ further decreased, now to a value lower than the capacity. Flow on $\{4,5\}$ overshot to $2.04 > 2$ since it was under-penalized, but we slowly adjusted it back with the later rounds.

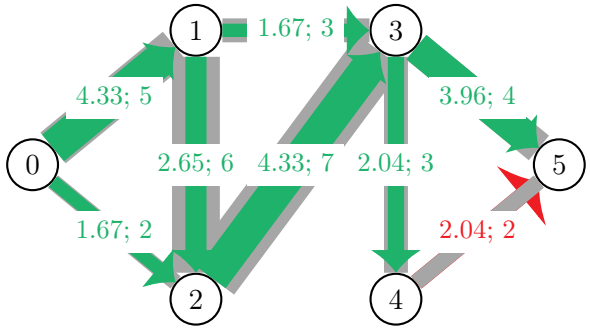

Figure 4. $f^{(50,000)}$: #1 electrical flow after round $50,000$

After all rounds, our electrical flow has converged towards a maximum flow—$\{3,5\}$ and $\{4,5\}$, edges incident to 5, are all saturated.

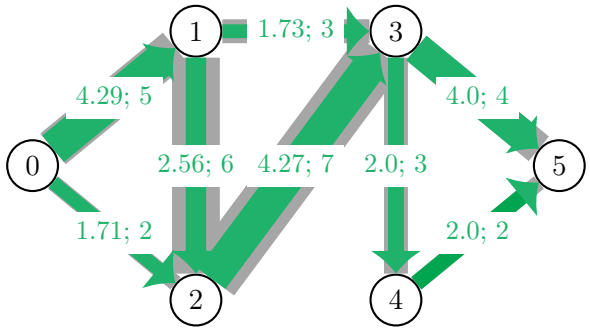

Figure 5. $f^{(111,595)}$: #1 electrical flow after round $111,595$

Note that in this example, despite that electrical flow itself converges to a maximum flow, the flow the algorithm produces is actually the mean of all these flows (properly scaled).

Consider a second example, where $s = 0$, $t = 9$.

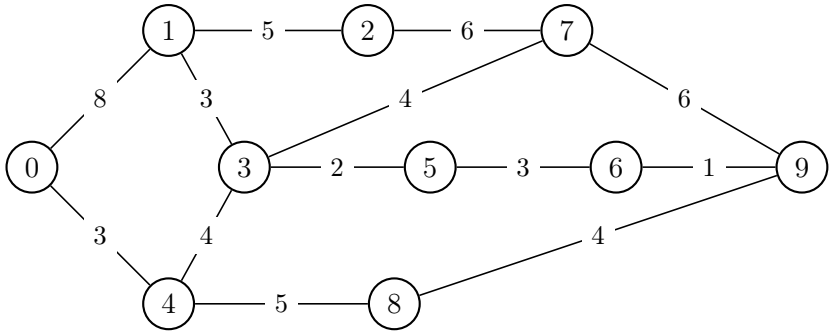

Figure 6. #2 max flow example

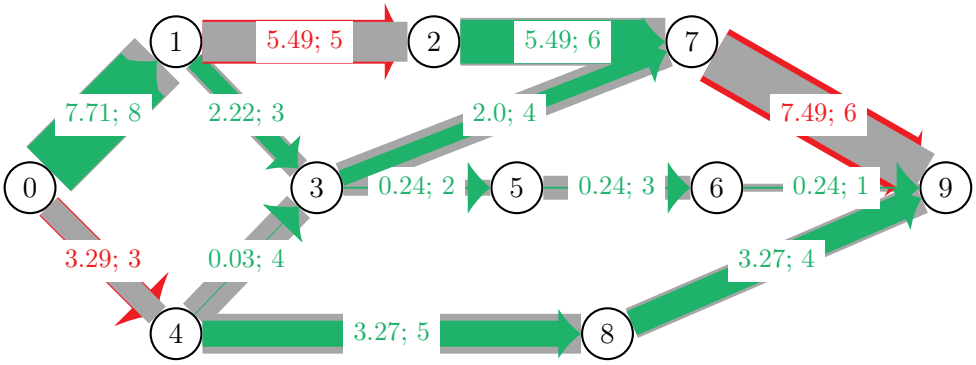

Figure 7. $f^{(1)}$: #2 electrical flow after round 1

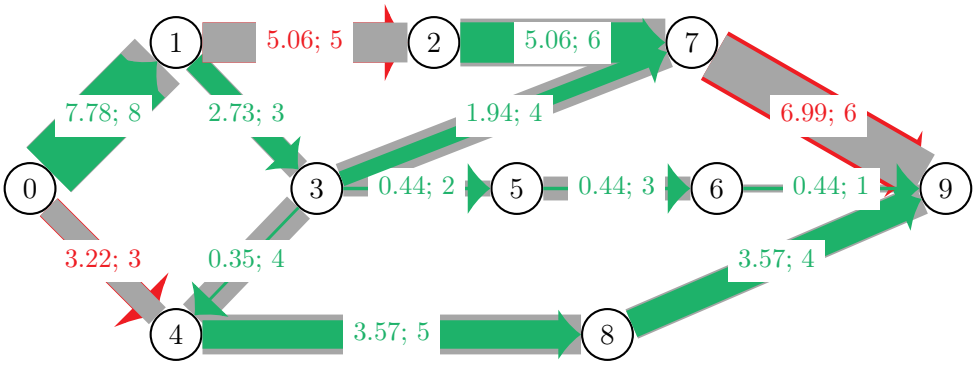

Figure 8. $f^{(50,000)}$: #2 electrical flow after round 50,000

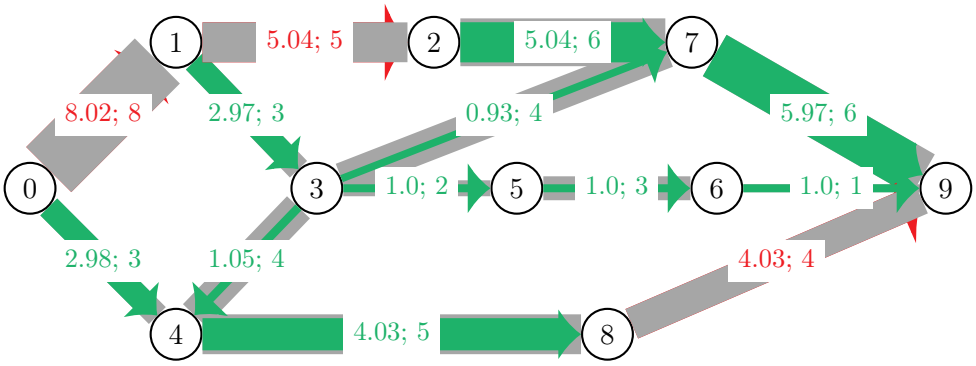

Figure 9. $f^{(130,000)}$: #2 electrical flow after round 130,000

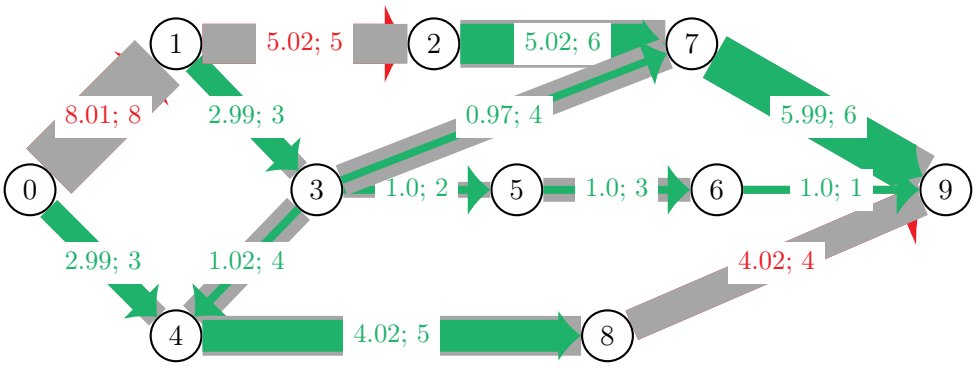

Figure 10. $f^{(175,470)}$: #2 electrical flow after round 175, 470

2.7. *Further Work.* To appreciate the full beauty of this algorithm, please refer to the full paper [1], which introduced such paradigm of approximation for maximum flow, and led to a breakthrough in time complexity since the work by Goldberg and Rao (1998) [2], which implied an approximation algorithm for maximum flow in $\tilde{O}(m\sqrt{n}\epsilon^{-1})$ time.

With some further work, the paper established an algorithm to approximate max flow in $\tilde{O}(mn^{1/3}\epsilon^{-11/3})$.

In 2022, a $m^{1+o(1)}$ time (*almost linear time*) algorithm to exactly compute maximum flow was discovered [3]. It involves more data structures and sparsification, but is "absurdly fast," said Daniel Spielman of Yale University, quoted by Quanta Magazine [4].

References

[1] Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. 2011. *Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs.* In Proceedings of the forty-third annual ACM symposium on Theory of computing (STOC '11). Association for Computing Machinery, New York, NY, USA, 273-282. https://doi.org/10.1145/1993636.1993674.

[2] Andrew V. Goldberg and Satish Rao. 1998. *Beyond the flow decomposition barrier.* J. ACM 45, 5 (Sept. 1998), 783-797. https://doi.org/10.1145/290179.290181.

[3] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. 2023. *Almost-Linear-Time Algorithms for Maximum Flow and Minimum-Cost Flow.* Commun. ACM 66, 12 (December 2023), 85-92. https://doi.org/10.1145/3610940

[4] Klarreich, Erica. 2022. *Researchers Achieve 'Absurdly Fast' Algorithm for Network Flow.* Quanta Magazine, 28 Sept. 2022.

[5] Lester R. Ford and Delbert R. Fulkerson. *Maximal flow through a network.* Canadian Journal of Mathematics, 8:399-404, 1956. https://doi.org/10.4153/CJM-1956-045-5.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. (2022). *Introduction to algorithms*, 1015-1022. MIT Press.

[7] Nick Littlestone, Manfred K. Warmuth. 1994. *The Weighted Majority Algorithm.* Information and Computation, Volume 108, Issue 2, 212-261, 1994, ISSN 0890-5401. https://doi.org/10.1006/inco.1994.1009.

Appendix A. Remaining Algebraic Work of Weighted Majority

Note the inequality $-x - x^2 \leq \ln(1-x) \leq -x$ for $0 \leq x \leq 1/2$. From $(1-\eta)^{m^*} \leq e^{-\eta \,\mathrm{E}[m]} n$,

$$\mathrm{E}[m] \leq -\frac{m^* \ln(1-\eta)}{\eta} + \frac{\ln n}{\eta} \leq (1+\eta)m^* + \frac{\ln n}{\eta}.$$

Appendix B. Python Implementation of the Simplified Algorithm

```
"""
Compute for an illustration for 'Multiplicative Weight':
  - Runs a sequence of electrical flows to approximate max flow
"""

import sys
import math
import doctest

import numpy as np
import pandas as pd
from tqdm import tqdm


class ElectricNet():
    """
    Implement the algorithm on page 10, with exact electrical flow solving.

    >>> net = ElectricNet([(3,0,3), (0,2,2), (3,1,5), (1,2,6)], s=3, t=2)
    >>> net.weigh(np.array([1.9, 1.2, 1.8, 1.2]), 4)
    >>> net.solve(4)
    array([0.65339637, 0.65339637, 3.34660363, 3.34660363])
    >>> net = ElectricNet([(3,0,3), (0,2,2), (3,1,3), (1,2,2), (0,1,1)], s=3, t=2)
    >>> net.weigh(np.array([1.9, 1.2, 1.9, 1.2, 3]), 4)
    >>> net.solve(4)
    array([2.00000000e+00, 2.00000000e+00, 2.00000000e+00, 2.00000000e+00,
           9.34728954e-17])
    >>> net = ElectricNet([(0,4,2), (3,2,1), (1,3,2), (4,3,8), (0,2,5)], s=4, t=1)
    >>> net.weigh(np.array([1, 2, 3, 7, 5]), 9)
    >>> net.solve(5)
    array([-0.08347026, -0.08347026, -5.    ,  4.91652974, 0.08347026])
    """
    def __init__(self, edge_list: list[tuple], s: int, t: int) -> None:
        # edge_list contains tuples (u, v, upper-bound) representing an edge {a, b}
        assert all(isinstance(u, int) and isinstance(v, int) and cap > 0
                   for (u, v, cap) in edge_list)
        self.n = max(max(u, v) for (u, v, cap) in edge_list) + 1
        self.m = len(edge_list)
```

```
        assert 0 <= s < self.n and 0 <= t < self.n and s != t
        self.s, self.t = s, t
        self.edges = edge_list

        # graph Laplacian matrix
        self.L = None

        # n x m incidence matrix
        self.B = np.zeros((self.n, self.m))
        for edge_idx, (u, v, cap) in enumerate(self.edges):
            self.B[u, edge_idx] = -1
            self.B[v, edge_idx] = 1

        # m x m capacity (diagonal) matrix (not computed yet)
        self.C = None

    def weigh(self, weights: np.ndarray[float], eps: float) -> None:
        """Given multiplicative weights, compute resistances and then Laplacian."""
        assert len(weights) == self.m and all(w > 1 or np.isclose(w, 1) for w in weights)
        weight_sum = weights.sum()
        capacities = []  # list of capacities, i.e., inverse of resistance
        self.L = np.zeros((self.n, self.n))
        for (u, v, cap), w in zip(self.edges, weights):
            # Compute resistance based on capacity, multiplicative weight, epsilon.
            resistance = (w + eps * weight_sum / 3 / self.m) / cap**2
            capacity = 1 / resistance
            capacities.append(capacity)
            self.L[u, u] += capacity
            self.L[v, v] += capacity
            self.L[u, v] -= capacity
            self.L[v, u] -= capacity
        self.C = np.diag(capacities)
        assert np.isclose(self.L, self.B @ self.C @ self.B.T).all()

    def solve(self, net_flow: float) -> np.array:
        assert isinstance(net_flow, (int, float))
        ksi = np.zeros((self.n, 1))
        ksi[self.s], ksi[self.t] = -1, 1
        f: np.array = self.C @ self.B.T @ np.linalg.pinv(self.L) @ ksi * net_flow
        assert np.isclose(self.B @ f, ksi * net_flow).all() # Kirchhoff's second law
        assert f.shape == (self.m, 1)
        return f.flatten()


class MulWeightSolver():
    def __init__(self, edge_list: list[tuple], s: int, t: int):
        # edge_list contains tuples (a, b, upper-bound) representing an edge {a, b}
        assert all(isinstance(u, int) and isinstance(v, int) and cap > 0
                   for (u, v, cap) in edge_list)
        self.n = max(max(u, v) for (u, v, cap) in edge_list) + 1
        self.m = len(edge_list)
        assert 0 <= s < self.n and 0 <= t < self.n and s != t
```

```
        self.s, self.t = s, t
        self.edges = edge_list

    def with_electric_flow(self, eps: float, optimal_flow: float,
                           round_multiplier: float = 10**-1.5):
        """
        Run multiplicative weights for the desired number of rounds, given the
        optimal maximum flow value.
        `round_multiplier` can be supplied to manually increase/decrease the num
        of rounds, for visualization purpose.

        Returns (flows, weights), i.e., a list of flow vectors and a list of
        lists of multiplicative weights.
        """
        electric_net = ElectricNet(self.edges, self.s, self.t)

        rho = 3 * (self.m / eps)**0.5
        rounds = math.ceil(round_multiplier * 2 * rho * math.log(self.m) / eps**2)
        # initialize multiplicative weights
        weight = np.ones((self.m, ), dtype=np.longdouble)
        f = np.zeros((self.m, ), dtype=np.longdouble)

        weights: list[np.array] = [weight] # the mul weights of each round
        flows: list[np.array] = [f]      # the flows of each round
        for _ in tqdm(range(rounds)):
            electric_net.weigh(weight, eps)
            f: np.array = electric_net.solve(optimal_flow)
            flows.append(f)
            for edge_idx, (u, v, cap) in enumerate(self.edges):
                weight[edge_idx] *= 1 + eps / rho * (abs(f[edge_idx]) / cap)
            weights.append(weight)
        assert len(weights) == len(flows)
        return flows, weights

    def solve(self, eps: float, optimal_flow: float):
        flows, weights = self.with_electric_flow(eps, optimal_flow)
        ans = ((1 - eps)**2 / (1 + eps)) * (sum(flows) / len(flows)) # average scaled
        return flows, weights, ans
```

Department of Mathematics, Columbia University

E-mail: columbiajournalofundergradmath@gmail.com

https://journals.library.columbia.edu/index.php/cjum