Implementation of a Combinatorial Maximum Concurrent Flow Algorithm

Matt Jordan and Jin Pan[*]

Massachusetts Institute of Technology
jordanm@mit.edu jinpan@mit.edu

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

I. Introduction

The multicommodity flow problem requires routing multiple commodities from their respective sources to their respective sinks along a directed graph so that the net flow across all commodities on any single edge does not exceed the capacity of that edge. Multicommodity flow problems arise in many different contexts where distinct resources need to be routed across a network. For example, multicommodity flow problems are solved when routing across communication networks or determining transportation of goods. [CITE]

Historically, this problem and its many variants can be expressed as a large linear program. This allows the multitude of linear programming algorithms to solve this problem exactly in a theoretically polynomial. The structure of this problem has been used to modify interior point methods to generate faster runtimes in practice, but the size of the linear program quickly gets prohibitively large. In practice, fully polynomial approximation schemes can very closely approximate a solution to the multicommodity flow problem, even for large problem instances. In practice, getting within 1% or even 5% of the optimal solution is often good enough, and can be attained much more quickly with approximation schemes than with linear programming methods. These fully polynomial approximation schemes often rely on subroutines which can be efficiently computed in practice, such as minimum cost flow or single-source shortest paths.

In this paper, we implement an algorithm for solving multicommodity flow variants first introduced by Garg in 2007 [CITE]. Specifically, we offer an implementation solving the maximum concurrent flow problem. We also implement several heuristics that take advantage of certain features of a problem instance to further speed up the algorithm, both in theory and practice. The paper is organized as follows. In section 2 we formally present the problem and define maximum concurrent flow in addition to discussing previous contributions towards creating an efficient fully polynomial approximation scheme for this problem. In section 3, we present the

[*]A thank you or further information

algorithm and offer a brief analysis of its runtime and correctness. In section 4, we present several heuristics implemented to speed up runtime in practice. Section 5 contains our description of our implementation, experiments as well as the experimental results and discussion. We include a real-world application of our algorithm in section 6, and finally conclude in section 7.

II. Background

Given a directed graph $G(V, E)$ with edge capacities $c : e \to \mathbb{R}^+$, and k commodities with source s_j and sink t_j for commodity j. Each commodity also has an associated demand $d(j)$. The problem of maximum concurrent flow is to find a feasible flow that maximizes λ, where for a given flow f that routes f_j units of commodity j from s_j to t_j,

$$\lambda = \min_j (f_j/d(j))$$

We remember that a feasible flow preserves capacity constraints:

$$\sum_j f_j(u,v) \le c(u,v)$$

where $f_j(u,v)$ represents the flow of commodity j through edge (u,v) for each edge $((u,v)) \in E$, and maintains flow conservation:

$$\sum_{v \in V} f_j(u,v) = 0$$

for $v \neq s_j, t_j$, for all commodities j and defining $f_j(u,v) = -f_j(v,u)$. Intuitively, we can describe the maximum concurrent flow problem as the following situation: we want to send commodities to their respective sinks, but instead of shooting for the bare minimum of demand satisfaction, we want to maximize the ratio of supply to demand for all commodities.

Typically this problem is formulated under the context of a linear program. If we let $\mathcal{F}_|$ be the set of flows that send d(j) units of commodity j from s_j to t_j, letting $\mathcal{F}$ be the union of $\mathcal{F}_|$ for $1 \le j \le k$, we can formulate this as the following LP:

$$\begin{aligned} &\max \lambda \\ \text{s.t. } &\sum_{f \in \mathcal{F}} f_e \cdot x(f) \le c(e) \quad \forall e \in E \\ &\sum_{f \in \mathcal{F}_|} x(f) \ge \lambda \quad \forall 1 \le j \le k \\ &x \ge 0,\ \lambda \ge 0. \end{aligned}$$

In this case, $x(f)$ is defined as the amount of times we use the primitive flow f, sending $d(j)$ units of commodity j from s_j to t_j for some j. Alternatively, we can model this problem with a different LP formulation, operating on paths instead of flows. Let $\mathcal{P}_\rceil$ be the set of paths starting at s_j and ending at t_J for some j that contain edge e, and let $\mathcal{P}_|$ be the set of paths from s_j to t_j. Our LP formulation is as follows:

$$\begin{aligned} &\max \lambda \\ \text{s.t. } &\sum_{p \in \mathcal{P}_\rceil} x(p) \le c(e) \quad \forall e \in E \\ &\sum_{p \in \mathcal{P}_|} x(p) \ge \lambda \cdot d(j) \quad \forall 1 \le j \le k \\ &x \ge 0,\ \lambda \ge 0. \end{aligned}$$

In this case, $x(p)$ can be defined as the amount of flow we send along path p. Both formulations solve an equivalent problem, however it is clear that both formulations are exponential in size. For this reason many fully polynomial approximation schemes have been developed. We will now briefly discuss previous algorithms and recent work on this problem.
The first fully polynomial approximation schemes (FPAS) for solving multi-commodity flow problems and their variants arose in the early 90's with Leighton et. al [CITE]. Since the story of min-cost multi-commodity flow (MCMCF) and maximum concurrent flow (MCF) are so intertwined, we will describe the background of MCMCF, keeping in mind that FPAS' for MCMCF are easily extensible to MCF, often using the same tricks, both in theory and to speed up implmentation in practice.
Theoretically, these algorithms run faster than interior-point methods for solving LP's, but it was several years after the development of the theoretical development of the first FPAS's that an efficient implementation was developed, attaining a speed two-to-three orders of magnitude faster than state-of-the-art linear program solvers [CITE]. The main idea of how these early FPAS' work is a rerouting method, generalizing on fractional packing techniques[CITE-karger abstract]. Initially, the algorithm finds an initial flow satisfying the demands, but possibly violating the capacities. Then the algorithm repeatedly picks a commodity via round-robin fashion and computes a single-commodity minimum-cost flow in the auxiliary graph, where the arcs have 'cost' that is exponential in the current flow through the graph. A fraction of this commodity's flow is rerouted to the newly computed minimum-cost flow. The 'goodness' of the reroutings are stored in a potential function, which is guaranteed to generate a $1+\omega$ solution in $(\omega^{-3}kmn)$ time for the minimum cost multi-commodity flow problem [CITE]. This was soon reduced to a quadratic dependence on ω. Since then, these bounds have since been increased to $(\omega^{-2}m^2)$ in 2000[Cite]. The current state-of-the-art bound for maximum concurrent multicommodity flow in $O(k^2\omega^2 2^{O(\sqrt{(\log|V|\log\log|V|)})})$, using a combination of a non-Euclidean generalization of gradient descent, flow sparsifiers, and an $O(m^{o(1)})$-competitive oblivious routing scheme [cite].

III. Algorithm

I. Presentation of algorithm

Avoiding the theoretical hangups of the most recent publication improving the bounds for maximum concurrent flow, we choose to implement the maximum concurrent flow FPAS presented by Garg and Könemann, and several heuristics to speed up runtime in practice. The basic implementation gives a runtime of $(\omega^{-}2(k+m)m)$. The dependency on k can be removed with the implementation of our heuristics as described later. For the remainder of this section, we will describe the algorithm implemented in this paper (herein referred to as Garg-MCF), offering both pseudocode and a brief look at the analysis.
The main idea behind Garg-MCF relies on the rerouting and fractional packing methods described in section 2. The essential ingredients of these methods include finding an initial infeasible solution, and then rerouting (typically calling to a known combinatorial optimization subroutine) based on a length function that grows exponentially in the amount that is rerouted. Garg-MCF has been theoretically shown to be much faster and simpler than previous algorithms. This simplicity makes the algorithm comprehensible. We will be more explicit in how Garg-MCF works now.

We refer to the second linear program formulation of maximum concurrent flow, which we will refer to as P-MCF. Taking the dual of P-MCF generates the following linear program, which defines a length $l(e)$ for each edge and a variable $z(j)$ for each commodity.

$$\begin{aligned} \min \sum_{e \in E} c(e)l(e) \\ \text{s.t. } \sum_{e \in p} l(e) \geq z(j) \quad \forall 1 \leq j \leq k, \forall p \in \mathcal{P}_| \\ \sum_{j=1}^{k} d(j) \cdot z(j) \geq 1 \\ l, z \geq 0 \end{aligned}$$

Recalling that $\mathcal{P}_|$ is the set of paths from s_j to t_j. Intuitively, the first constraint maintains that, when tight, $z(j)$ is the value of the length of the shortest path from s_j to t_j. Keeping with Garg's notation, we define the objective value to be a function of the length assignment l:

$$D(l) := \sum e \in Ec(e)l(e)$$

and define $\alpha(l)$ as the second constraint:

$$\alpha(l) := \sum_{j=1}^{k} d(j) \cdot z(j)$$

Then since we'll have a minimal objective value when the second constraint is tight, this LP can be viewed as the assignment of positive lengths to edges such that $\frac{D(l)}{\alpha(l)}$ is minimized.
With notation in hand, the algorithm runs as follows. We first assign a length of $\delta / c(e)$ to each edge, where δ is carefully picked to make the analysis work out. Next we proceed in phases. For each phase, we have k iterations. For each iteration, we loop through each commodity j and reroute $d(j)$ units of flow from s_j to t_j. We do this with a series of steps. Let $l^s_{i,j}$ refer to the length function at the i^{th} phase, the j^{th} iteration and the $s + 1^{th}$ step. Then during each step, we compute the minimum cost path from s_j to t_j under this length function and route as much flow along that path as possible. That is, we wish to route a total of $d(j)$ flow from s_j to t_j over all steps during an iteration, so let $d^s_{i,j}$ refer to flow remaining to be rerouted during an iteration, after s steps. Then for each step, we compute a path p, and we route $f^{s+1}_{i,j}$ flow along this path, where

$$f^{s+1}_{i,j} = \min(\min_{e \in p}(c(e)), d^s_{i,j})$$

We then decrease the amount of flow remaining to be routed by $f^{s+1}_{i,j}$. Since we routed flow along every edge in p, we need to update the length function for these edges, so we do this by multiplying $l(e)$ by $1 + \epsilon \frac{f^{s+1}_{i,j}}{c(e)}$ for an ϵ that we define later, dependent upon ω. We terminate the iteration when $d^p_{i,j} = 0$ for some step number p. We repeat this process for each commodity during an iteration, and we repeat phases until we reach our stopping condition, which we define when $D(l_i) \geq 1$. We then have a graph that has flow along edges, but is almost surely infeasible. We can scale the flow along each edge by dividing flow by $\log_{1+\epsilon} \frac{1}{\delta}$, which makes the flow feasible. From here, we can calculate λ directly. For clarity, we present the pseudocode of this algorithm:

II. Analysis

We offer a brief overview of the analysis. A more complete and rigorous analysis can be viewed in the original paper.

```
$G \leftarrow (V, E)$;
$\epsilon, \delta \leftarrow$ calculate_epsilon($\omega$), calculate_delta($\omega$);
for $e \in E$ do
    $l(e) = delta/c(e)$;
while $D(l) < 1$ do
    for $j = 1$ to $k$ do
        $d_j \leftarrow d(j)$;
        while $d_j \neq 0$ do
            $p \leftarrow$ shortest_path($s_j, t_j$);
            $min\_capacity \leftarrow \infty$;
            for $e \in p$ do
                $min\_capacity \leftarrow \min(c(e), min\_capacity)$;
            $f \leftarrow \min(min\_capacity, d_j)$;
            for $e \in p$ do
                $e.flow \leftarrow e.flow + f$;
                $l(e) \leftarrow l(e) * (1 + \epsilon * f/c(e))$;
```

II.1 Approximation Ratio

First we claim that $\lambda > \frac{t-1}{\log_{1+\epsilon}\frac{1}{\delta}}$ where t is the stopping phase: For every $c(e)$ units of flow routed through each edge e, we increase the length of e by a factor of $1+\epsilon$. Since the initial length is $\frac{\delta}{c(e)}$ and $D(t-1) < 1$, so $l_{t-1,k}(e) < \frac{1}{c(e)}$. So we can bound the flow through e in the first $t-1$ phases is less than $\log_{1+\epsilon}\frac{1}{\delta}$. If we scale the flow by the scaling factor described above of $\log_{1+e}\frac{1}{\delta}$ we get a feasible flow, and have that

$$\lambda > \frac{t-1}{\log_{1+\epsilon}\frac{1}{\delta}}$$

Now if we compare the ratio,γ, of the value of the primal LP λ to the value of the dual solution, β using the above bound, we get that the ratio is strictly less than $\frac{\beta}{t-1}\log_{1+\epsilon}\delta^{-1}$.
Now we seek to bound $\frac{\beta}{t-1}$. We examine the objective value after the final step of the j^{th} iteration of the i^{th} phase, $D(l^p_{i,j})$:

$$D(l^p_{i,j}) = \sum_{e\in E} c(e)l(e) \leq D(l^0_{i,j-1}) + \epsilon d(j)\mathrm{dist_j}(l^p_{i,j})$$

where $\mathrm{dist_j}$ defines the length of the shortest path from s_j to t_j where the argument defines the length function. This follows since we have to route exactly $d(j)$ units of flow along a sequence of paths no longer than $\mathrm{dist_j}(l^p_{i,j})$. We can then sum over all iterations within a phase to show

$$D(i) \leq D(i-1) + \epsilon\alpha(i)$$

where $D(i)$ refers to the objective value after the i^{th} phase, and $\alpha(i)$ refers to α after the i^{th} phase. If we define the minimum value of $D(i)/\alpha(i)$ to be β, then we can rewrite this as

$$D(i) \leq \frac{D(i-1)}{1-\epsilon/\beta}$$

. Since we set all lengths initially to $\delta / c(e)$, $D(0) = m\delta$, so we can show that

$$D(i) \leq \frac{m\delta}{1-\epsilon} e^{\frac{\epsilon(i-1)}{\beta(1-\epsilon)}}$$

Assuming that $\beta \geq 1$. This assumption will be lifted in section 4. We stop at the first t such that $D(t) \geq 1$, so we have that

$$1 \leq \frac{m\delta}{1-\epsilon} e^{\frac{\epsilon(t-1)}{\beta(1-\epsilon)}}$$

which can be rearranged to bound $\frac{\beta}{t-1}$ above by $\frac{\epsilon}{(1-\epsilon) ln \frac{1-\epsilon}{m\delta}}$.
So now we have that $\gamma < \frac{\beta}{t-1} \log_{1+\epsilon} \delta^{-1}$ we can plug in the bound for $\frac{beta}{t-1}$. By setting

$$\delta = (m \cdot (1-\epsilon)^{-1})^{\frac{-1}{\epsilon}})$$

and working through the mathematics, we show that $\gamma \leq (1-\epsilon)^{-3}$. Thus we can pick an ϵ such that $(1-\epsilon)^{-3}$ is no more than the approximation ratio $1+\omega$.

II.2 Runtime

Finally we argue for runtime. It is enough to bound the number of steps for each iteration, and the number of phases. We start with steps.
For each step except the last one per iteration, we increase the length of at least one edge by a factor of $1+\epsilon$. Since each edge has length between δ and $1+\epsilon$, we can do no more than $m \log_{1+\epsilon} \frac{delta}{1+\epsilon}$ steps total, plus one more step for each iteration. We bound the total number of iterations as k times the number of phases. The number of phases, then is t such that

$$1 < \frac{Beta}{t-1} \log_{1+\epsilon} \delta^{-1}$$

where this above inequality comes from our bound on γ and weak duality for LP's, thus the number of phases is less than $1 + \beta \log_{1+\epsilon} \delta^{-1}$ so we have that

$$t = \lceil \frac{\beta}{\epsilon} \log_{1+\epsilon} \frac{m}{1-\epsilon} \rceil$$

. Using the β scaling trick we will describe in the next section, we can show that the total number of phases is at most $T \log k$ where

$$T = \frac{2}{\epsilon} \log_{1+\epsilon} \frac{m}{1-\epsilon}$$

.

IV. Heuristics

In this section we present several heuristics that don't speed up the theoretical running time but reduce the number of phases or shortest path computations in practice.

I. Demand scaling

This heuristic allows us to assume that $\beta \geq 1$. Recall that β is defined to be the minimum value of $\frac{D(i)}{\alpha(i)}$, and is necessary is achieving a bound on γ, the ratio between the primal and dual solutions. That is, the number of phases depends on β. Notice that scaling all demands by a factor of c scales α by a factor of c^{-1}, so scaling all demands by c scales β by c. If we let z_j be the maximum possible flow of commodity j, we define z to be the maximum fraction of the demands that can be routed independently to each commodity. Thus, β is bounded above by z and bounded below by $\frac{z}{k}$. Thus, we scale each demand by $\frac{k}{z}$ such that $\frac{z'}{k} = 1$ where z' represents a calculation of z with the updated demands. In doing this, we bound β below by 1. We can then rescale demands accordingly if our procedure doesn't stop within $\frac{2}{\epsilon} \log_{1+\epsilon} \frac{m}{1-\epsilon}$ based on equation EQ [update this later], which implies that $\beta \geq 2$, so scaling demands by a factor of 2 maintains $\beta \geq 1$. This multiplies the total number of phases by a factor o $\log k$ and costs $k \cdot T_{mf}$ where T_{mf} is the time to compute a max flow.

II. 2-approximation for Beta estimation

This heuristic allows us to greatly reduce the number of phases we perform by bounding β between 1 and 2. To do this, we first compute a 2-approximation to the problem using $O(\log k \log m)$ phases, and returns a $\hat{\beta}$ such that $\beta \leq \hat{\beta} \leq 2\beta$. Then we don't have to scale demands using the above procedure and reduces the number of phases to $O(\log m(\log k + \epsilon^{-1})$.

III. Karakosta's method for shared sources

This heuristic allows us to remove the depedence upon k, the number of commodities, using a technique first introduced by Karakosta [CITE]. Since we compute shortest paths using a single-source shortest path method, with no additional runtime we obtain the minimum cost path from the source to all possible sinks. Grouping the commodities by shared sources, we can run the iterations for every commodity in a particular group at the same time. To do this we make only one call to Djikstra's for each group for each step and route a scaled flow along the path associated with each commodity. We scale each flow for commodity j by the ratio of the demand remaining for commodity j to the sum of demands remaining for all other commodities within this group. The above analysis still holds since we scale at least one edge by a factor of $1+\epsilon$ for each step except the last. The number of iterations then, is bounded by the number of groups, which is at most n. Thus our runtime of $O(\omega^{-2}(k \log m + m) \log m T_{sp})$ becomes $O(\omega^{-2}(m \log m + m) \log m T_{sp})$, or $(w^{-2}m^2)$.

IV. Multiple routing

We'll fill this in once we implement it.

V. Experimental Results

We have implemented Garg-MCF in the Python programming language and have tested it upon a variety of problems. Due to the automatic garbage collection of Python, clock time of the implementation should be taken with a grain of salt. Instead, we measure the number of calls to the single-source shortest path subroutine. Since we make one call per step this serves as a valid measurement of runtime. We first describe the construction of random graphs.

We construct random directed graphs using the networkx module's gnm_random_graph routine, feeding parameters of n and m. Then we randomly create commodities by a grouping procedure. We take the number of commodities as a parameter and a distribution which defines the number of shared source commodities we have, randomly choosing sinks for the commodities, ensuring that there exists a directed path from the source to the sink for each commodity. Capacities and demands are chosen randomly. This set of procedures allows us to randomly generate a directed graph with a given number of nodes, edges, commodities, and we can explicitly the number of commodity groups. With this in hand, we can proceed to test the algorithm's dependence on ω and k, taking note of the effect of karakosta's heuristic for shared source commodities.

I. Dependence on the parameter ω

II. Dependence on the parameter k

.

VI. Conclusions

That's all.

References

[Figueredo and Wolf, 2009] Figueredo, A. J. and Wolf, P. S. A. (2009). Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20:317–330.