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. It also requires that the net flow across all commodities on any single edge does not exceed the capacity of that edge and that flow is conserved. 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.

Historically, this problem and its many variants have been expressed as a large linear program. This allows us to use the multitude of linear programming algorithms to solve this problem exactly in polynomial time. 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 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 in 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 algorithm and offer a brief

[*]A thank you or further information

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, results and discussion. Finally we offer our concluding remarks in section 6.

II. Background

We formally introduce the problem of maximum concurrent flow here. We start with 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 the ratio of flow sent to commodity j to the demand of commodity j, for all commodities. Formally, maximum concurrent flow finds a flow f 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:

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

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. This differs from other multicommodity flow variants, such as the maximum multi-commodity flow problem where the goal is to simply maximize total throughput from sources to sinks with no demands on each commodity.

Typically the problem of maximum concurrent flow is formulated under the context of a linear program. If we let $\mathcal{F}_j$ be the set of flows that send $d(j)$ units of commodity j from s_j to t_j, letting $\mathcal{F} = \cup_{j=1}^{k} \mathcal{F}_j$, the set of flows that route $d(j)$ units of flow from s_j to t_j for some j. We can formulate a linear program with a variable $x(f)$ for each element $f \in \mathcal{F}$ as follows:

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

Where f_e is defined as the flow crossing edge e in the primitive flow f. In this case, $x(f)$ is defined as the fractional 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. Decoded, the first constraint maintains that no edge has flow exceeding its capacity. For a particular j, the second constraint ensures that we send more than $\lambda d(j)$ flow to t_j.

Alternatively, we can model this problem with a different LP formulation, operating on paths instead of flows. Let $\mathcal{P}_j$ be the set of paths starting at s_j and ending at t_J. Then we can define $\mathcal{P}$ to be the union of $\mathcal{P}_j$ for all j, that is, $\mathcal{P}$ is the set of all paths from s_j to t_j for any j. Let $\mathcal{P}_e$ be the set of paths in $\mathcal{P}$ such that edge e is in the path. Then our LP formulation assigns a variable $x(p)$ for each path $p \in \mathcal{P}$ and has the following description:

$$\begin{aligned} &\max \lambda \\ \text{s.t.} \sum_{p \in \mathcal{P}_e} x(p) \leq c(e) \;\; &\forall\, e \in E \\ \sum_{p \in \mathcal{P}_j} x(p) \geq \lambda \cdot d(j) \;\; &\forall\, 1 \leq j \leq k \\ x \geq 0,\; \lambda \geq 0. \end{aligned}$$

In this case, $x(p)$ can be defined as the amount of flow we send along path p. Then the first set of constraints ensures we don't send flow across an edge that exceeds the edge's capacity. The second constraint then maintains that the total flow sent from s_j to t_j is greater than λ times the demand $d(j)$, ensuring that we maximize the minimal ratio of flow to demand over all commodities. 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.

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]. Typically, variants of multi-commodity flow problems have often been solved with similar techniques. That is, the tricks that tend to work well for solving one variant of a multi-commodity flow problem can easily be extended to other multi-commody flow variants. For this reasion, in discussing the background of maximum concurrent flow (MCF), we will describe the background of min-cost multi-commodity flow (MCMCF) which has typically been the problem on which most of the following techniques were first developed. Keep 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 reroutes flow using a single-commodity minimum-cost flow in the auxiliary graph. The costs on the edges of the auxiliary graph are initialized to a small value, but are scaled for every unit of flow sent through them, ultimately making the cost of an edge exponential in the amount of flow routed through it. 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]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 complicated methods 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 $(\omega^{-2}(k+m)m)$ for the MCF problem. 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 introduced in section 2. The intuition behind Garg-MCF comes from fractional packing techniques first introduced by Shahrokhi and Matula.

In plain english, we start by assigning a 'length' to each edge dependent upon the error term within our approximation ω and the capacity of that edge. Then we repeatedly satisfy the demand of each commodity by sending as much flow as we can (independently) along the shortest path from the source to the sink, where shortest is dependent upon the 'length' function. Then we scale the 'length' of each edge by a factor dependent upon the amount of flow we just sent across it. In this way, the 'length' is exponential in the amount of flow being sent across the edge. Intuitively, since we call shortest paths based on this length function, this causes us to spread our flow across edges and reroute in such a fashion that doesn't force all our flow on one path. We stop after the length functions grow 'large enough'. The precise definition of 'large enough' and that this computes a $1+\omega$ approximation is not entirely obvious, but is believable under the context of the analysis. We will be more formal 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 we'll refer to as D-MCF, 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 $\mathcal{P}_j$ 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:

$$D(l) := \sum_{e\in E} c(e)l(e)$$

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 an initial 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 loop through each commodity j in a series of k iterations, one for each commodity. For each j^{th} iteration then, we consider 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, directly after the s^{th} step. Then during each step, we compute the minimum cost path from s_j to t_j under this length function,$l^s_{i,j}$ 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 refer to flow remaining to be rerouted during an iteration, after s steps. Initially $d^0 = d(j)$ and conclude the iteration when $d^q = 0$ for some q.

Then for each step, we compute a path p, and we route $f^{s+1}_{i,j}$ flow along this path, where $f^{s+1}_{i,j}$ is the maximum allowable flow we can send, or in other words, the minimum between the capacity of the minimum capacity edge in p and the flow remaining to be sent during this iteration. So

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

We then decrease the amount of flow remaining to be routed, d^s by $f^{s+1}_{i,j}$. Since we routed flow along every edge in p, we also need to update the length function for these edges, so we do this by multiplying $l^s_{i,j}(e)$ 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 $D(l^s_{i,j}) \geq 1$. 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. Motivation for calculation of δ, ϵ, and the final scale factor, $\log_{1+\epsilon} \frac{1}{\delta}$, will be briefly described in the next section, and fully derived in the original paper. For clarity, we present the pseudocode of this algorithm:

IV. Analysis

We offer a brief, high-level overview of the analysis. We point the reader to the original paper for a more in-depth analysis.

I. Approximation Ratio

To prove the correctness of Garg-MCF, we care about approximating a feasible solution to the dual LP, D-MCF. The key idea here is compare our calculated λ value to the best-possible solution to the dual LP, D-MCF. Bounding this ratio above by $(1 + \omega)$ will attain the appropriate approximation ratio, since the ratio is bounded below by 1, according to weak duality. We'll briefly describe how this is done below.

We first assume that the optimal objective value, β, will be at least 1. We can remove this assumption later. Then we notice a relation between the objective value at phase i and phase $i - 1$ and use this to establish a bound on the optimal objective value divided by the number of phases we must run through to reach our stopping condition. We next can create a lower bound for λ using the knowledge of how much flow we must route through each edge up to the penultimate

Algorithm 1: GARG-MCF without heuristics

```
$G \leftarrow (V, E)$;
$\epsilon, \delta \leftarrow$ 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($s_j, t_j$);
            $min\_cap \leftarrow \infty$;
            for $e \in p$ do
                $min\_cap \leftarrow \min(c(e), min\_cap)$;
            $f \leftarrow \min(min\_capacity, d_j)$;
            $d_j \leftarrow d_j - f$;
            for $e \in p$ do
                $e.flow \leftarrow e.flow + f$;
                $l(e) \leftarrow l(e) \cdot (1 + \epsilon \cdot f / c(e))$;
    $scale\_factor \leftarrow \log_{1+\epsilon} \delta^{-1}$;
    for $e \in E$ do
        $e.flow = e.flow / scale\_factor$;
return calculate_lambda(G);
```

phase. It is here that our scaling factor is derived, and a lower bound for λ established dependent on the number of phases completed and ϵ.

Now we can consider the ratio between λ and the solution to D-MCF, since we have bounds on both terms. If we can show that $\beta/\lambda \leq \text{poly}(\epsilon)$ then by weak duality we have that $\lambda \leq \beta$, so our computed λ is within a factor of poly(ϵ) of β. Strong duality holds that β is the value of the optimal solution to MCF. With the math of the paper we arrive at the claim that

$$\frac{\beta}{\lambda} \leq (1 - \epsilon)^{-3}$$

Thus, if we choose ϵ such that

$$(1 - \epsilon)^{-3} \leq 1 + \omega$$

we arrive at our desired $1 + \omega$ approximation.

II. Scaling Beta

We now lift the previous assumption that $\beta \geq 1$. The key idea here is that by scaling all the demands by a constant factor, we also scale the value of the optimal λ, and therefore β. We find bounds above and below for β based on the current demand scheme, and then scale the demands such that the lower bound for β is 1. It turns out that for any phase number i in which the algorithm has not yet terminated, i is strictly less than $\frac{\beta}{\epsilon} \log_{1+\epsilon} \frac{m}{1-\epsilon}$. Then we can run phases until we have computed enough phases to ensure that $\beta \geq 2$, a number we'll call T. In this case we scale demands of all commodities, effectively reducing β by a factor of two. If the ratio of the upper bound and lower bound for β was initially c, then we have to run for at most $T \log c$ phases.

III. Running time

The scaling of β gives us a bound on the number of phases we must compute. In each phase, we compute k iterations, so all that is left is to bound the number of steps per iteration. The key idea here is that $l(e)c(e)$ is δ for each edge e initially, and no more than $1+\epsilon$ at termination, since we terminate as soon as $\sum_e l(e)c(e) \geq 1$ and we scale each edge by no more than $1+\epsilon$ each time we scale. Thus the number of steps is at most $m \log_{1+\epsilon} \frac{m}{1-\epsilon}$ plus the number of iterations. Putting this all together, we end up with a runtime of

$$O(\omega^{-2}(k \log m + m) \log m) T_{sp})$$

where T_{sp} is the time to compute a single-source shortest path subroutine.

V. Heuristics

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

I. 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 $O(\log m(\log k + \epsilon^{-1})$.

II. 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 equivalent to 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 $O(\omega^{-2}(n \log m + m) \log m T_{sp})$, or $(w^{-2}m^2)$.

VI. 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 specify 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

.

VII. 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.