Implementation of a Combinatorial Maximum Concurrent Flow Algorithm

Matt Jordan and Jin Pan

Abstract— **Maximum Concurrent Flow (MCF) is used in a wide range of optimization problems where one needs to send multiple commodities concurrently across a network to meet a set of demands. In particular, MCF optimizes the minimum fraction of the how much any one commodity is satisfied. Here, we implement a fully polynomial approximation scheme for MCF first introduced by Garg. We summarize the algorithm and analysis and discuss various heuristics that affect theoretical and experimental runtime. We offer a novel heuristic based on sloppy β scaling that demonstrates better experimental runtime. We then present an implementation of this algorithm and these heuristics upon a series of randomly generated graphs to demonstrate the effect that varying parameters has upon runtime. We conclude that our implementation is indeed quadratic in the error term, and we observed that several heuristics offer useful speedups in practice.**

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 concurrently routed across a network. For example, multicommodity flow problems are solved when routing across communication networks or determining routing of goods across a transportation network.

Historically, this problem and its many variants have been solved by formulating the problem as a large linear program [2]. 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 [1]. In particular, 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 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. 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 minimal ratio of flow sent to commodity j to the demand of commodity j, over 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 multicommodity flow problem where the goal is to simply maximize total throughput from sources to sinks with no demands on each commodity. Maximizing the minimum ratio is often more desirable in practice, as this ensures that all commodities are partially satisfied to some ratio, instead of an uneven satisfaction of demands.

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 any fixed j. We can formulate a linear program with a variable $x(f)$ for each element $f \in \mathcal{F}$ as follows:

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

Where f_e is defined as the amount of flow that f sends across edge e. 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 p in $\mathcal{P}$ such that edge e is in the p. Then our LP formulation assigns a variable $x(p)$ for each path $p \in \mathcal{P}$ and has the following description:

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

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 the size of the graph. 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 [8]. Typically, variants of multicommodity flow problems have often been solved with similar techniques. That is, the tricks that tend to work well for solving one variant of a multicommodity flow problem can easily be extended to other multi-commody flow variants. Because of this, 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's 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 [2].

The main idea of how these early FPAS' work is a rerouting method, generalizing on fractional packing techniques [6]. 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 subroutine on the auxiliary graph until the flow is feasible. The costs on the edges of the auxiliary graph are initialized to a small value, but are scaled for every unit of flow rerouted through them, ultimately making the cost of an edge exponential in the amount of flow sent through it. The 'goodness' of the reroutings are stored in a potential function, which is guaranteed to generate a $1+\omega$ solution in $\tilde{O}(\omega^{-3}kmn)$ time for the minimum cost multi-commodity flow problem [6]. Since then, these bounds have since been increased to $\tilde{O}(\omega^{-2}m^2)$ in 2000 [3]. Most recently, Jonathon Kelner of MIT proposed a method to find an ω approximate maximum concurrent multicommodity flow problem in $O(m^{1+o(1)}\omega^{-2}k^2)$ time using a combination of a non-Euclidean generalization of gradient descent, flow sparsifiers, and an $O(m^{o(1)})$-competitive oblivious routing scheme [7].

III. Algorithm

A. 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 $\tilde{O}(\omega^{-2}(k+m)m)$ for the MCF problem [1]. 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. In plain english, we compute a $(1+\omega)$ approxmation as follows. We start by adjusting the demands of each commodity such that the optimal solution to the LP is at least 1; this is done for the sake of the analysis. We proceed by assigning a 'length' to each edge dependent upon ω and the capacity of that edge. 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 rescale the demands as we proceed to speed up runtime, but also maintain that the optimal λ is at least 1. 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.

Consider 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.

$$\min \sum_{e\in E} c(e)l(e)$$

$$\text{s.t. } \sum_{e\in p} l(e) \geq z(j) \quad \forall 1 \leq j \leq k, \forall p \in \mathcal{P}_|$$

$$\sum_{j=1}^{k} d(j)\cdot z(j) \geq 1$$

$$l, z \geq 0$$

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, also a function of the length assignment l:

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

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 scale the demands of each commodity in a fashion to ensure that the optimal value is at least 1. This scaling factor is referred to as $\frac{k}{z}$ with the derivation shown in the original paper. The exact mechanism by which this is done isn't as important as noticing that scaling the demands by a constant factor scales the value of λ by the inverse of that constant factor. Next, we assign an initial length of $\delta/c(e)$ to each edge, where δ is carefully picked to make the analysis work out. Then 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, updating the length after each step. 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 $D(l^s_{i,j}) \geq 1$, which is checked at the end of every phase. Also, upon completion of a certain number of phases, T, we can guarantee that the optimal value is greater than 2, so we can rescale the demands of each commodity by a factor of 2, and we maintain the invariant that the optimal value is at least 1. Upon completion of the algorithm, we have a graph that has flow along edges, but almost surely overflows capacity bounds. We can scale the flow along each edge by dividing flow by $\log_{1+\epsilon}\delta^{-1}$, which makes the flow feasible. From here, we can calculate λ directly. Motivation for calculation of δ, ϵ, and the final scale factor, $\log_{1+\epsilon}\delta^{-1}$, will be briefly described in the next section, and fully derived in the original paper. For clarity, we present the pseudocode of this algorithm in Algorithm 1.

Algorithm 1: GARG-MCF without heuristics

```
G ← (V, E);
ε, δ ← calculate_epsilon(ω), calculate_delta(ω);
for e ∈ E do
    l(e) = δ/c(e);
scale_demands(k/z);
phase_since_rescale ← 0;
while D(l) < 1 do
    if phase_since_rescale > T then
        scale_demands(2);
        phase_since_rescale ← 0;
    for j = 1 to k do
        d_j ← d(j);
        while d_j ≠ 0 do
            p ← shortest_path(s_j, t_j);
            min_cap ← ∞;
            for e ∈ p do
                min_cap ← min(c(e), min_cap);
            f ← min(min_capacity, d_j);
            d_j ← d_j − f;
            for e ∈ p do
                e.flow ← e.flow + f;
                l(e) ← l(e) · (1 + ε · f/c(e));
    phase_since_rescale+ = 1;
scale_factor ← log_{1+ε} δ^{-1};
for e ∈ E do
    e.flow = e.flow/scale_factor;
return calculate_lambda(G);
```

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.

A. 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 to 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 sketch 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 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 β and λ, 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 $\text{poly}(\epsilon)$ of β. Strong duality implies 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.

B. 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 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. In the full proof, Garg showed that $c = m$, so we have to run for at most $T\log m$ phases.

C. Running time

The scaling of β gives us a bound on the number of phases we must compute. Since we compute k iterations per phase, this also bounds the number of iterations we must complete. To compute runtime, we need to calculate the total number of steps we can perform. The key insight here is that there are two different types of steps: steps where we saturate an edge, and steps that do not. Notice that in each iteration, all steps saturate an edge, except the last step in the iteration. This is to say that for all but the last iteration, at least one edge has a length that is scaled by a factor of $1+\epsilon$. Also note 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 total number of saturating steps is $m\log_{1+\epsilon}\delta^{-1}$. By setting δ to $(m/(1-\epsilon))^{-1/\epsilon}$, the number of saturating steps is at most $\frac{m}{\epsilon}\log_{1+\epsilon}\frac{m}{1-\epsilon}$. Since the total number of steps is the number of saturating steps plus the number of non-saturating steps, and the number of non-saturating steps is equivalent to the number of iterations, we have that the total number of steps is no more than $T\log m + \frac{m}{\epsilon}\log_{1+\epsilon}\frac{m}{1-\epsilon}$. For appropriate choice of ϵ we combine these to show that we get a runtime of $(\omega^{-2}(k\log m + m)\log m)T_{sp})$ 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 and one novel heuristic that reduces the number of phases or shortest path computations in practice, in addition to affecting the theoretical runtime.

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

B. Karakostas' method for shared sources

This heuristic allows us to remove the depedence upon k, the number of commodities, using a technique first introduced by Karakostas [5]. 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 $\tilde{O}(w^{-2}m^2)$.

C. Removal of demand-scaling

This novel heuristic was developed to further simplify the algorithm and has been shown to provide a slight decrease in runtime in practice. The key idea here is that we consider whether or not it is possible for β to be less than 1. If so, we do not do any demand-scaling. Otherwise, we scale demands as outlined in Garg-MCF. We can do this only under the provision that we do not check the termination condition until the end of each phase. We offer sketches of proofs that this heuristic can only improve runtime (in practice), and maintains a $(1 + \omega)$ approximation.

For runtime, we have that the runtime is equivalent to the number of steps times the time to compute a shortest path. Each step is either a saturating step, where we send flow such that an edge is saturated, or it isn't, in which case it is the last step of the iteration. Our bound on the number of saturating steps remains the same, by the same arguments presented in the above analysis. The number of unsaturating steps, however, depends on the number of phases, and therefore on the scaling of demands. Suppose that demand-scaling causes demands to decrease. Then each iteration routes less flow through, and since the total flow pushed through the graph upon termination is not dependent upon the scaling factor, this increases the number of non-saturating steps we have to perform. In other words, this scaling increases the number of iterations and therefore the runtime. If demand-scaling causes demands to increase, then by not doing demand scaling, we lose out on potential improvement, so we still scale demands in this case. Thus, we offer no changes to theoretical runtime, but show a runtime improvement in practice.

It should be noted that the analysis presented by Garg assumed that $\beta \geq 1$, and this constraint is loosened by not scaling demands. We claim that, given an adjustment of the flow-scaling factor applied upon termination, we maintain a $(1 + \omega)$ approximation. Note that we only check the termination condition at the end of every phase. This implies that the ratios of flow sent for each commodity are constant (i.e., we send $d(j)$ flow of commodity j for each phase if we don't scale, and $cd(j)$ flow of commodity j each phase when we do scale, for scaling factor c). Thus we do not artificially increase the objective value by sending more flow to any one commodity, relative to the demand of that commodity. However, this relaxation breaks the analysis presented in Garg claiming that the flow-scaling factor returns the generated flow to feasibility. We remedy this by noting that the ratios of flow sent through a commodity to their demand is constant, and thus by not-scaling demands, we maintain the appropriate ratios of flows for a $(1+\omega)$ approximation. We return this flow to feasibility by simply finding the maximum overloaded edge m, and scaling the flow across all edges by the ratio of $c(m)$ to the flow through m.

VI. Experimental Results

We have implemented Garg-MCF in the Python programming language and tested it upon a variety of input problems using the PyPy JIT compiler. Due to the automatic garbage collection and high level nature of Python, the clock time of our implementation should be taken with a grain of salt. We report the number of calls to the single-source shortest path subroutine as a more reliable performance indicator, as we make one call per phase of the algorithm. We first describe the construction of random graphs.

We construct random directed graphs using the networkx module's $\mathrm{gnm\_random\_graph}$ routine, feeding parameters of n and m [4]. 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 at random. 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 parameters $\{\omega, k, n\}$, taking note of the effect of Karakostas' heuristic for shared source commodities.

We initially validated the correctness of our algorithms by running on several small graphs ($m <\leq 10$) with multiple commodities that could be optimized by hand. Included in these test cases were multiple commodities with shared and unshared sources, and demands that vary orders of magnitudes to empirically validate relaxation on demand-scaling. We found our implementation provided an answer within the $(1 + \omega)$ approximation ratio on all test cases and thus proceed confidently with the following experiments.

A. Dependence on the parameter ω

In theory, the Garg-MCE algorithm runs with a quadratic dependence on ω, which we observe in practice, both in terms of number of calls to our shortest paths function, and also wall time.

We observe that the algorithm works correctly and even slightly more efficiently if we do *not* scale β such that it is bounded above by 1, as previous analysis assumes.

The two-approximation heuristic was implemented on top of Karakostas' heuristic and we do not observe it outperforming just Karakostas' heuristics. However, our data hints that the two-approximation may be asymptotically more optimal, but, regrettably, limited precision arithmetic on primitive floats and expensive computations on arbitrary precision decimals render testing smaller values of ω infeasible.

We include data on physical time performance in the appendix.

B. Dependence on the parameter k

The Garg-MCE algorithm has a linear dependence on k, and the Karakostas heuristic absorbs this factor.

Again, we observe our vanilla implementation outperforming the implementation with β scaling, the Karakostas heuristic outperforming both, and the two-approximation having similar performance.

C. Dependence on the parameter n

Our algorithm has a quadratic dependence on the number of edges, comparable to the number of nodes, given our sparse graph. We expect there to be a linear dependence of the number of shortest path calls on the number of nodes.

A surprising result is that this dependence is experimentally not very strong for relatively small n: we observe that the number of calls to shortest paths does not necessarily increase with an increase in graph size.

D. Dependence on the distribution

Our algorithm does not strictly depend on how the commodities are distributed, but the Karakostas heuristic takes advantage of multiple commodities that start at the same source. We expect the Karakostas heuristic to significantly outperform baseline in the case where all the commodities start at the same source, and underperform when all the commodities start at differing sources.

Somewhat surprisingly, our baseline vanilla and beta algorithms are dependent on the input distribution. We hypothesize that this may be due to underlying caching optimizations that work more efficiently when the search algorithm is more predictable and visits the same nodes initially.

VII. Conclusions

Maximum Concurrent Flow has many applications where one needs to route mulitple commodities simultaneously across a network. Like other variants of multi-commodity flow, this can be solved exactly using linear programming methods, but in practice approxmiation algorithms are often good enough and run much more efficiently. In this paper we implemented a fully polynomial approximation scheme for solving MCF first introduced by Garg, quadratic in both error and the size of the graph (ignoring polylog factors), when appropriate heuristics are applied. We develop a novel heuristic that runs theoretically no slower than the original algorithm, as well as implementing a heuristic that reduces the algorithm's dependence on the number of commodities.

Our results show that the number of shortest path computations is quadratic in the error term and is roughly linear in the number of commodities. The number of shortest path computations in the original algorithm also appears to be roughly linear in the number of edges. The technique first proposed by Karakostas to reduce dependence on k has been

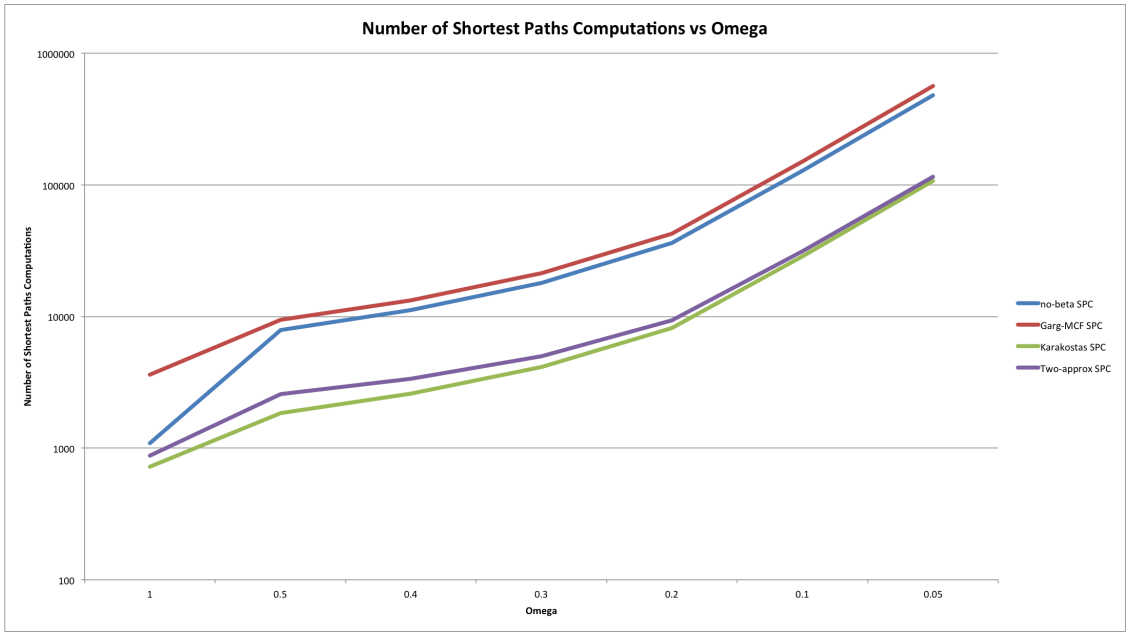

Fig. 1. Plots of number of shortest-path computations versus error term ω for all heuristics. This data was obtained from running our algorithm, with the respective heuristics activated, on graphs with 100 nodes, 400 directed edges, 10 commodities (split into two groups of size 6 and 4 that shared the same source) on 10 different random graphs. Of the 10 measured shortest path computations for each ω, we dropped the minimum and maximum, and reported the average over the remaining.

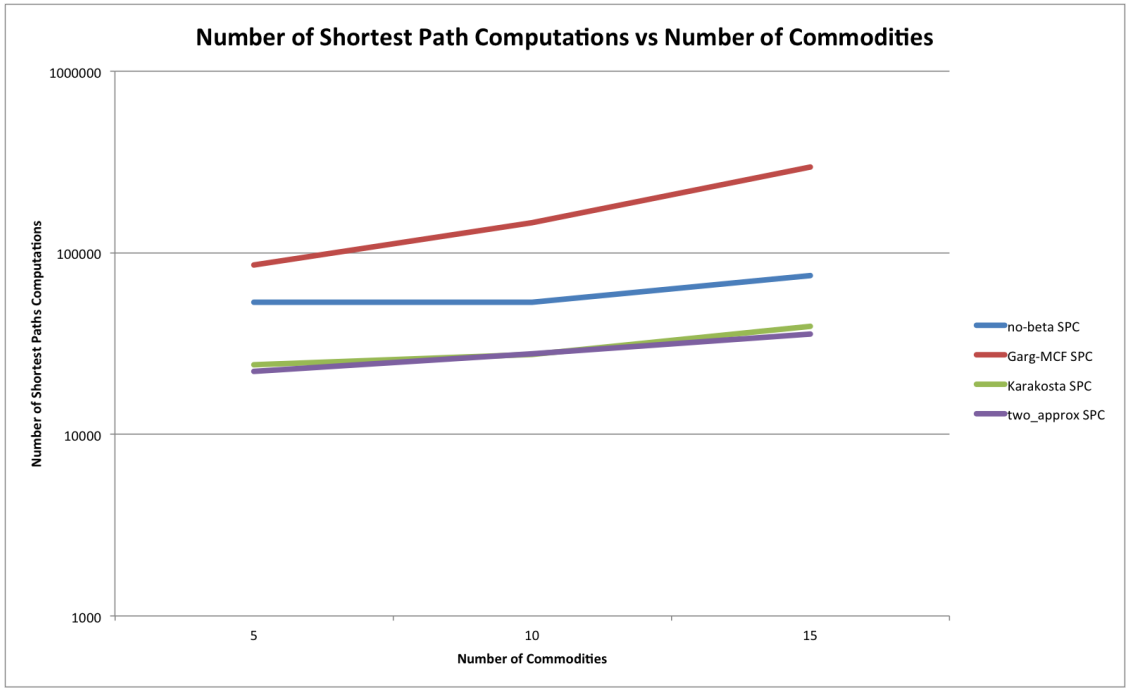

Fig. 2. Plots of number of shortest-path computations versus number of commodities k for all heuristics. This data was obtained from running our algorithm, with the respective heuristics activated, on graphs with 100 nodes, 400 directed edges, and a error margin of 10%, 10 times.

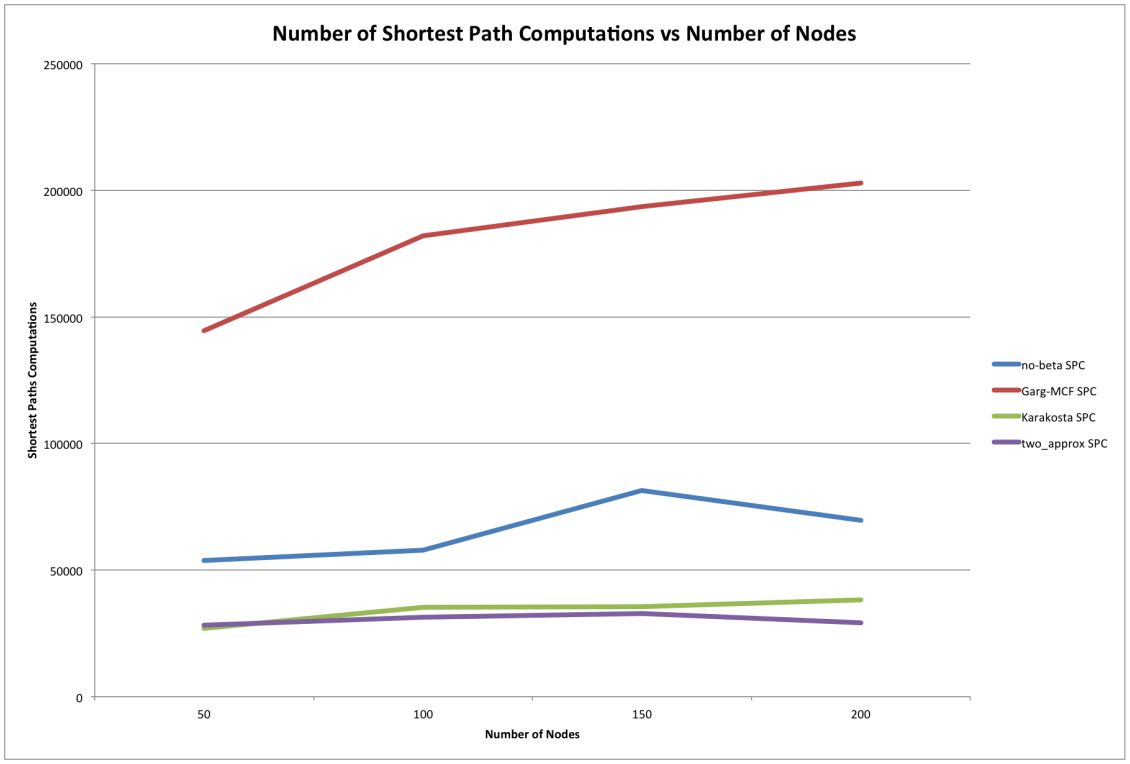

Fig. 3. Plots of number of shortest-path computations versus number of nodes for all heuristics. This data was obtained from running our algorithm, with the respective heuristics activated, 10 commodities (split into two groups of size 6 and 4 that shared the same source), and an error margin of 10%, 10 times. We construct our graphs such that the number of edges is always four times the number of nodes.

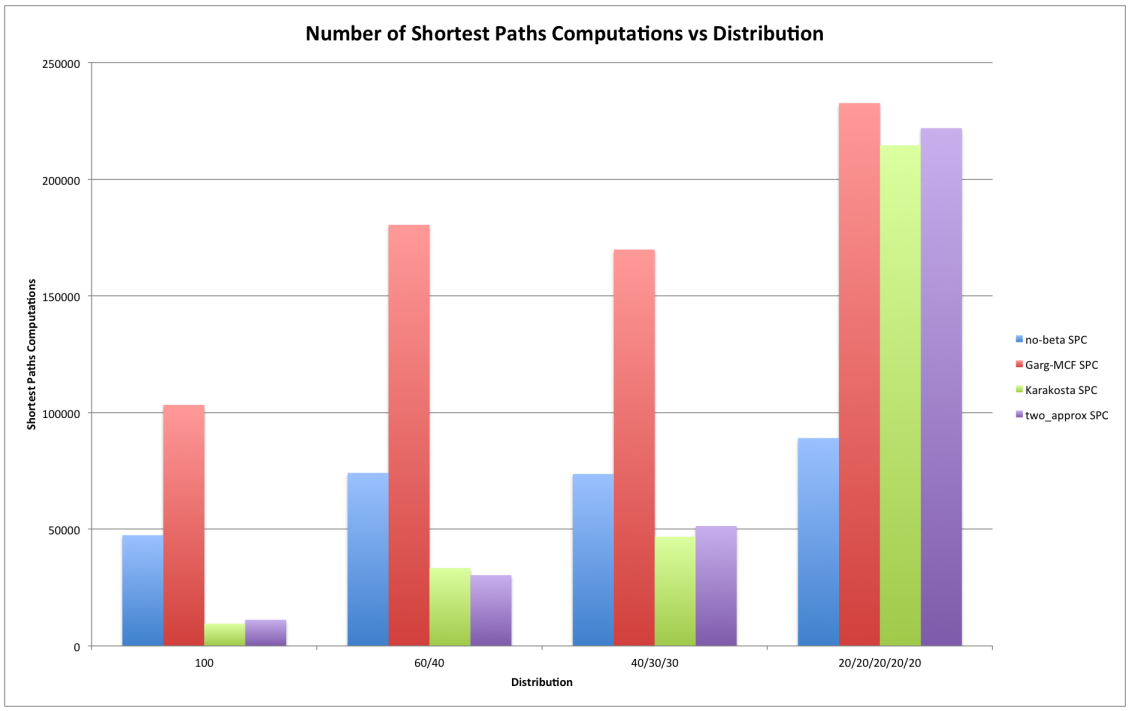

Fig. 4. Plots of number of shortest-path computations versus number of nodes for all heuristics.This data was obtained from running our algorithm, with the respective heuristics activated, on graphs with 100 nodes, 400 directed edges, 10 commodities, 10 times. Each time, we vary the distribution of commodities that share the same source.

shown to make fewer calls to the shortest paths subroutine than the vanilla implementation, particularly on problem instances where there is a large number of shared-source commodities. The two-approximation heuristic was implemented on top of Karakostas' algorithm, and we observed similar numbers of calls to the shortest path subroutine, implying that we do not see a dramatic decrease in runtime in practice from this heuristic, despite the theoretical benefits. Finally, we observed that our novel heuristic that did not scale demands unnecessarily outperformed the vanilla algorithm proposed by Garg in shortest paths computations and wall time. This leads to hypotheses that we could be sloppier with demand-scaling at the beginning of our algorithm and carefully scale-demands as we approach termination to further reduce the number of phases performed.

Our codebase can be downloaded from https://github.com/jinpan/6.854-project.

REFERENCES

[1] Naveen Garg and Jochen Koenemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM Journal on Computing*, 37(2):630–652, 2007.

[2] Andrew V Goldberg, Jeffrey D Oldham, Serge Plotkin, and Cliff Stein. *An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow*. Springer, 1998.

[3] Michael D Grigoriadis and Leonid G Khachiyan. Approximate minimum-cost multicommodity flows in $\tilde{O}(\epsilon^{-2}knm)$ time. *Mathematical Programming*, 75(3):477–482, 1996.

[4] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[5] George Karakostas. Faster approximation schemes for fractional multicommodity flow problems. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 166–173. Society for Industrial and Applied Mathematics, 2002.

[6] David Karger and Serge Plotkin. Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 18–25. ACM, 1995.

[7] Jonathan A. Kelner, Lorenzo Orecchia, Yin Tat Lee, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. *CoRR*, abs/1304.2338, 2013.

[8] Tom Leighton, Fillia Makedon, Serge Plotkin, Clifford Stein, Éva Tardos, and Spyros Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Sciences*, 50(2):228–243, 1995.

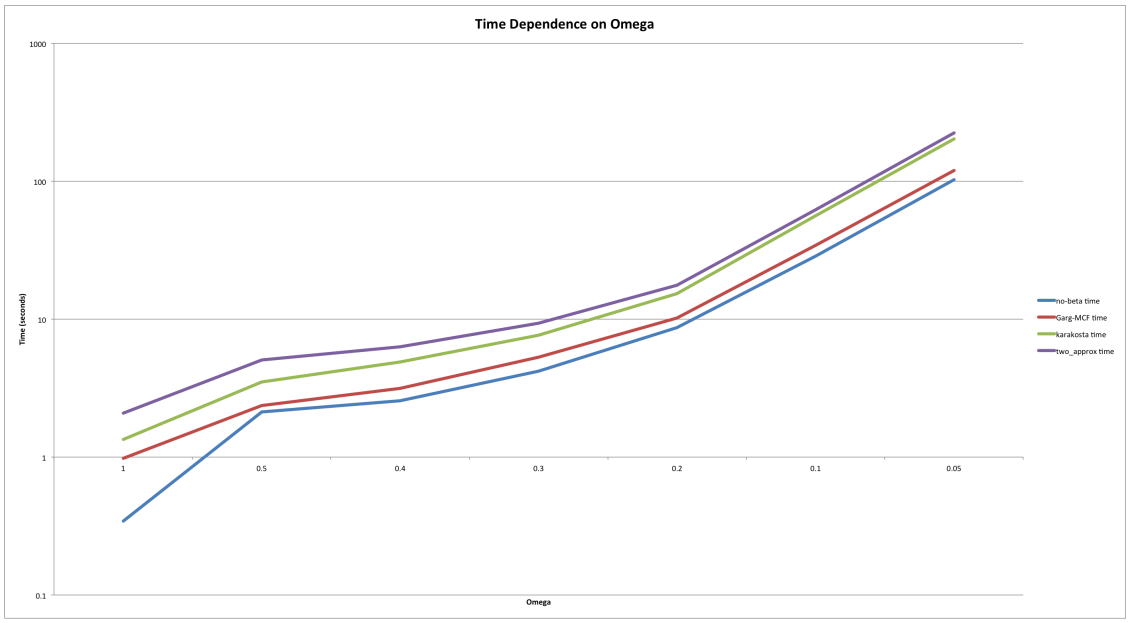

Fig. 5. Plots of time in seconds versus error term ω for all heuristics

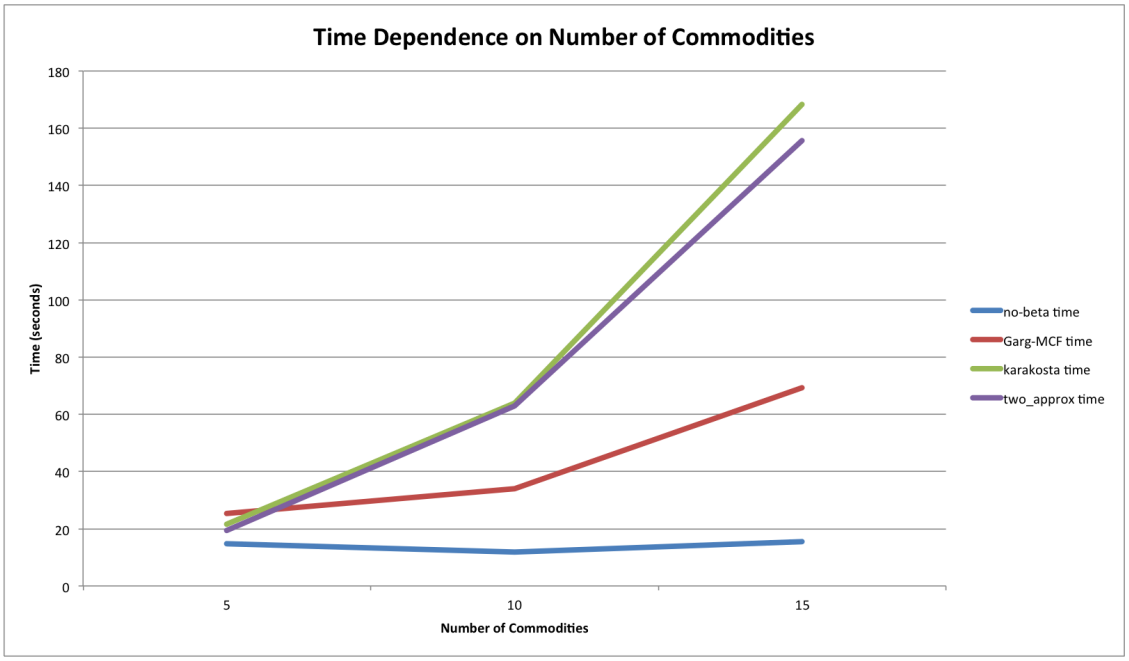

Fig. 6. Plots of time in seconds versus number of commodities for all heuristics

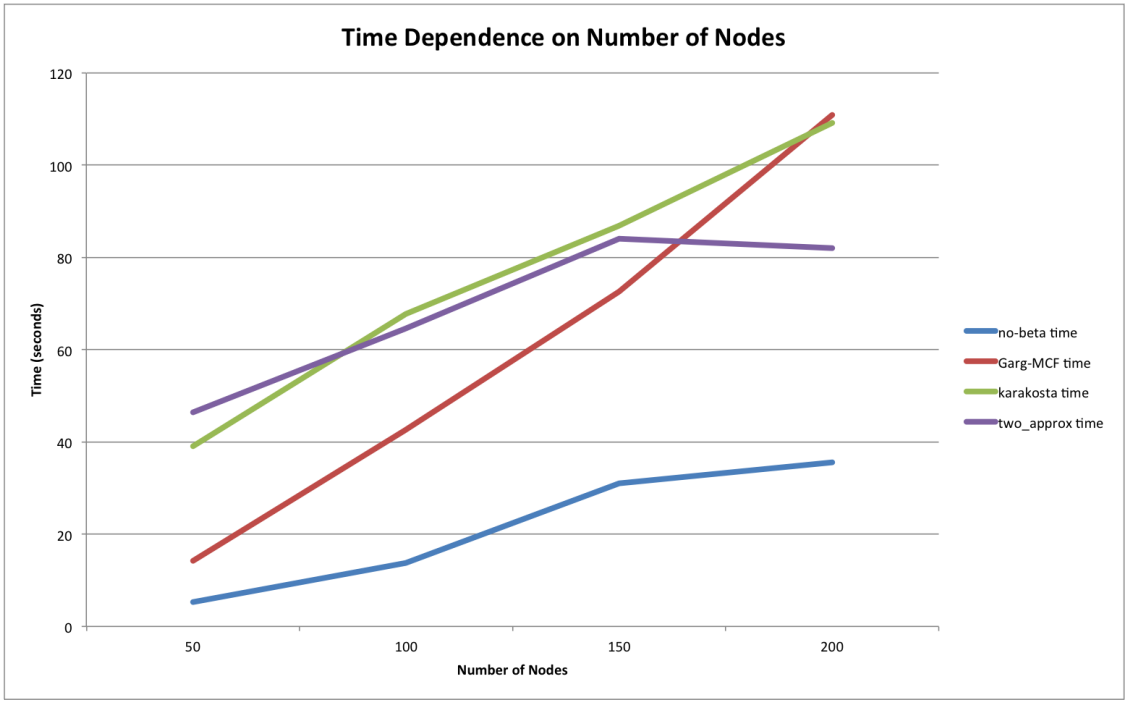

Fig. 7. Plots of time in seconds versus number of nodes for all heuristics

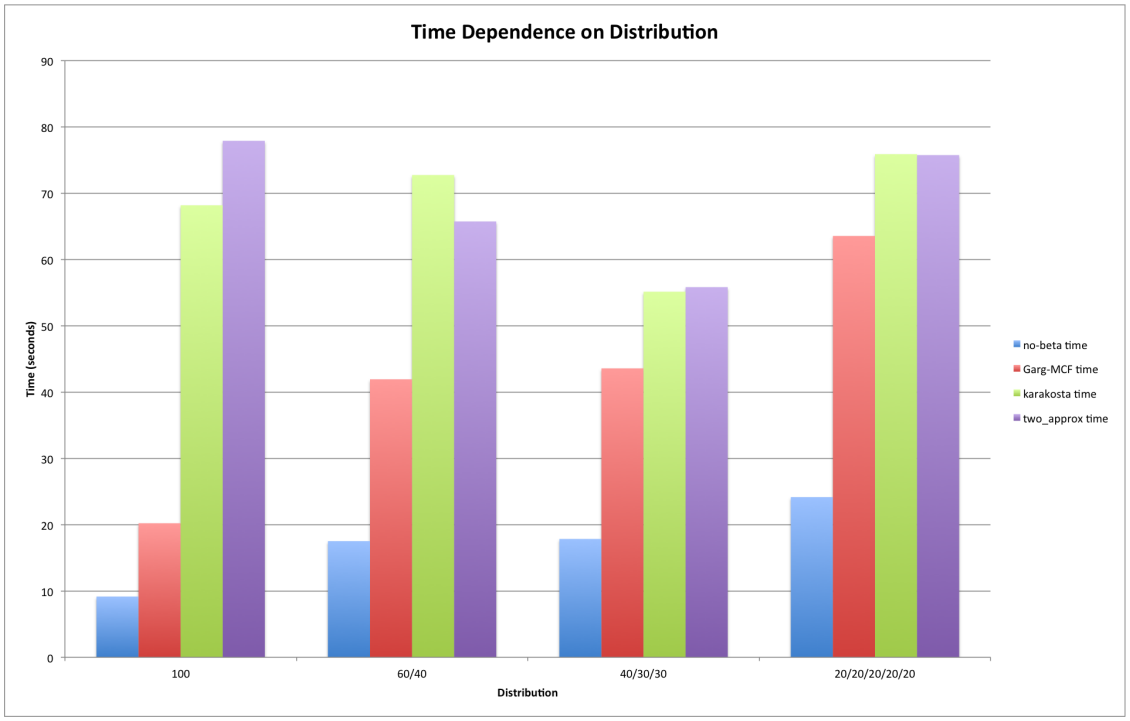

Fig. 8. Plots of time in seconds versus distribution of commodities