Implementing Approximation Algorithms for the Single-Source Unsplittable Flow Problem

Jingde Du
and
Stavros G. Kolliopoulos

In the *single-source unsplittable flow* problem, commodities must be routed simultaneously from a common source vertex to certain sinks in a given graph with edge capacities. The demand of each commodity must be routed along a single path so that the total flow through any edge is at most its capacity. This problem was introduced by Kleinberg [1996a] and generalizes several NP-complete problems. A cost value per unit of flow may also be defined for every edge. In this paper, we implement the 2-approximation algorithm of Dinitz, Garg, and Goemans [1999] for congestion, which is the best known, and the $(3, 1)$-approximation algorithm of Skutella [2002] for congestion and cost, which is the best known bicriteria approximation. We study experimentally the quality of approximation achieved by the algorithms and the effect of heuristics on their performance. We also compare these algorithms against the previous best ones by Kolliopoulos and Stein [1999]

Categories and Subject Descriptors: G.2.2 [**Discrete Mathematics**]: Graph Algorithms—*Graph Algorithms*; *Network Problems*

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Approximation algorithms, network flow, unsplittable flow

1. INTRODUCTION

In the *single-source unsplittable flow* problem (UFP), we are given a directed graph $G = (V, E)$ with edge capacities $u : E \to \mathbb{R}^+$, a designated source vertex $s \in V$, and k commodities each with a terminal (sink) vertex $t_i \in V$ and associated demand $d_i \in \mathbb{R}^+$, $1 \le i \le k$. For each i, we have to route d_i units of commodity i along a single path from s to t_i so that the total flow through an edge e is at most its capacity $u(e)$. We set $d_{\max} = \max_{1\le i\le k} d_i$, $d_{\min} = \min_{1\le i\le k} d_i$, and $u_{\min} = \min_{e\in E} u_e$. As is standard in the relevant literature we assume that no edge can be a bottleneck, i.e., $d_{\max} \le u_{\min}$. We will refer to instances which satisfy this assumption as *balanced*, and ones which violate it as *unbalanced*. Instances in which $d_{\max} = \rho u_{\min}$, $\rho > 1$,

Authors' address: J. Du, Department of Mathematics and Statistics, York University. Email: duj6@hotmail.com. Part of this work was done while at the Dept. of Computing and Software, McMaster University, partially supported by NSERC Grant 227809-00.
S. G. Kolliopoulos, Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Panepistiomopolis Ilissia, Athens, 157 84, Greece. Email: sgk@di.uoa.gr. Part of this work was done at the Dept. of Computing and Software, McMaster University, partially supported by NSERC Grant 227809-00.

are *ρ-unbalanced*. A relaxation of Ufp is obtained by allowing the demands of commodities to be split along more than one path; this yields a standard maximum flow problem. We will call a solution to this relaxation, a *fractional* or *splittable* flow.

We also use the following terminology. As a flow function on the edges, an *unsplittable flow* f can be specified by a set of paths $\{P_1, \cdots, P_k\}$, where P_i starts at the source s and ends at t_i, such that $f(e) = \sum_{i:e \in P_i} d_i$ for all edges $e \in E$. An unsplittable flow f is called *feasible* if in addition to satisfying all the demands, f respects the capacity constraints, i.e., $f(e) \leq u(e)$ for all $e \in E$. If a cost function $c : E \rightarrow \mathbb{R}^+$ on the edges is given, then the cost $c(f)$ of flow f is given by $c(f) = \sum_{e \in E} f(e) \cdot c(e)$. The cost $c(P_i)$ of an path P_i is defined as $c(P_i) = \sum_{e \in P_i} c(e)$ so that the cost of an unsplittable flow f given by paths $P_1, \cdots, P_k$ can also be written as $c(f) = \sum_{i=1}^{k} d_i \cdot c(P_i)$. In the version of the Ufp *with costs,* apart from the cost function $c : E \rightarrow \mathbb{R}^+$ we are also given a budget $B \geq 0$. We seek a feasible unsplittable flow whose total cost does not exceed the budget. For $a, b \in \mathbb{R}^+$ we write $a \,|\, b$ and say that b is *a-integral* if and only if $b \in a \cdot \mathbb{N}$.

The feasibility question for Ufp (without costs) is strongly NP-complete [Kleinberg 1996a]. Various optimization versions can be defined for the problem. In this study we focus on *minimizing congestion*: Find the smallest $\alpha \geq 1$ such that there exists a feasible unsplittable flow if all capacities are multiplied by α. Among the different optimization versions of Ufp the congestion metric admits the currently best approximation ratios. Moreover congestion has been studied extensively in several settings for its connections to multicommodity flow and cuts.

Previous work. Ufp was introduced by Kleinberg [1996a] and contains several well-known NP-complete problems as special cases: Partition, Bin Packing, scheduling on parallel machines to minimize makespan [Kleinberg 1996a]. In addition Ufp generalizes single-source edge-disjoint paths and models aspects of virtual circuit routing. A ρ-approximation algorithm for congestion, $\rho \geq 1$, is a polynomial-time algorithm that outputs an unsplittable flow with congestion at most ρ times the optimal. The first constant-factor approximations were given in [Kleinberg 1996b]. Kolliopoulos and Stein [2002; 1999] gave a 3-approximation algorithm for congestion which also guarantees a flow cost of value at most 2 times the optimal cost of a fractional solution. A *bicriteria* (ρ_1, ρ_2)-approximation algorithm for congestion and cost is a polynomial-time algorithm which is guaranteed to output a solution which simultaneously has congestion at most ρ_1 times the optimal and cost at most ρ_2 times the given budget. In this notation, [Kolliopoulos and Stein 2002] gave a $(3, 2)$-approximation. Dinitz, Garg, and Goemans [1999] improved the congestion bound to 2. To be more precise, their basic result is: any splittable flow satisfying all demands can be turned into an unsplittable flow while increasing the total flow through any edge by less than the maximum demand [Dinitz et al. 1999]. This result is tight if the congestion achieved by the fractional flow is used as a lower bound. Skutella [2002] improved the $(3, 2)$-approximation algorithm for congestion and cost [Kolliopoulos and Stein 2002] to a $(3, 1)$-approximation algorithm.

In terms of negative results, Lenstra, Shmoys, and Tardos [1990] show that the minimum congestion problem cannot be approximated within less than $3/2$, unless $P = NP$. Skutella [2002] shows that, unless $P = NP$, congestion cannot be approximated within less than $(1+\sqrt{5})/2 \approx 1.618$ for the case of $\big((1+\sqrt{5})/2\big)$-unbalanced instances. Erlebach and Hall [2002] prove that for arbitrary $\varepsilon > 0$ there is no $(2-\varepsilon, 1)$-approximation algorithm for congestion and cost unless $P = NP$. Matching this bicriteria lower bound is a major open question.

This work. As a continuation of the experimental study initiated by Kolliopoulos and Stein [1999], we present an evaluation of the current state-of-the-art algorithms from the literature. We implement the two currently best approximation algorithms for minimizing congestion: (i) the 2-approximation algorithm of Dinitz, Garg, and Goemans [1999] (denoted DGGA) and (ii) the $(3, 1)$-approximation algorithm of Skutella [2002] (denoted SA) which simultaneously mininimizes the cost. We study experimentally the quality of approximation achieved by the algorithms, and the effect of heuristics on approximation and running time. We also compare these algorithms against two implementations of the Kolliopoulos and Stein [2002; 1999] 3-approximation algorithm (denoted KSA). Extensive experiments on the latter algorithm and its variants were reported in [Kolliopoulos and Stein 1999].

The primary goal of our work is to investigate experimentally the quality of approximation. We also consider the time efficiency of the approximation algorithms we implement. Since our main focus is on the performance guarantee we have not extensively optimized our codes for speed and we use a granularity of seconds to indicate the running time. Our input data comes from four different generators introduced in [Kolliopoulos and Stein 1999]. The performance guarantee is compared against the congestion achieved by the fractional solution, which is always taken to be 1. This comparison between the unsplittable and the fractional solution mirrors the analyses of the algorithms we consider. Moreover it has the benefit of providing information on the "integrality" gap between the two solutions. In general terms, our experimental study shows that the approximation quality of the DGGA is typically better, by a small absolute amount, than that of the KSA. Both algorithms behave consistently better than the SA. Nevertheless the latter remains competitive for minimum congestion even though it is constrained by having to meet the budget requirement. All three algorithms achieve approximation ratios which are typically well below the theoretical ones.

Organization of the paper. In Section 2 we review the 2-approximation algorithm for minimum congestion. Skutella's $(3, 1)$-approximation algorithm for the single-source unsplittable min-cost flow problem is presented in Section 3. Sections 4 and 5 provide respectively our experimental framework and the experimental results for the implementations of the two algorithms. In Section 6 we conclude and discuss some future work.

2. THE 2-APPROXIMATION ALGORITHM FOR MINIMUM CONGESTION

In this section we briefly present the DGGA [Dinitz et al. 1999] and give a quick overview of the analysis as given in [Dinitz et al. 1999]. The skeleton of the algorithm is given in Fig. 1.

We explain the steps of the main loop. Certain edges, labeled as *singular*, play a special role. These are the edges (u, v) such that v and all the vertices reachable from v have out-degree at most 1. To construct an *alternating cycle* C we begin from an arbitrary vertex v. From v we follow outgoing edges as long as possible, thereby constructing a *forward* path. Since the graph is acyclic, this procedure stops, and we can only stop at a terminal, t_i. We then construct a *backward* path by beginning from any edge entering t_i distinct from the edge that was used to reach t_i and following singular incoming edges as far as possible. We thus stop at the first vertex, say v', which has another edge leaving it. We now continue by constructing a forward path from v'. We proceed in this manner till we reach a vertex, say w, that was already visited. This creates a cycle. If the two paths containing w in the cycle are of the same type, then they both have to be forward paths and we glue them into one forward path. Thus the cycle consists of alternating forward and backward paths.

DGG-ALGORITHM:

```
Input:   A directed graph $G = (V, E)$ with a source vertex $s \in V$, $k$ commodities
         $i = 1, \cdots, k$ with terminals $t_i \in V \setminus \{s\}$ and positive demands $d_i$, and a
         (splittable) flow on $G$ satisfying all demands.
Output:  An unsplittable flow given by a path $P_i$ from $s$ to each terminal $t_i$, $1 \leq$
         $i \leq k$.

remove all edges with zero flow and all flow cycles from $G$;
preliminary phase:
$i := 1$;
while $i \leq k$ do
    while there is an incoming edge $e = (v, t_i)$ with flow $\geq d_i$ do
        move $t_i$ to $v$;
        add $e$ to $P_i$;
        decrease the flow on $e$ by $d_i$;
        remove $e$ from $G$ if the flow on $e$ vanishes;
    $i := i + 1$;
main loop:
while outdegree($s$) $> 0$ do
    construct an alternating cycle $C$;
    augment flow along $C$;
    move terminals as in the preliminary phase giving preference to singular edges
    with flow $= d_i$;
return $P_1, \cdots, P_k$;
```

Fig. 1. Algorithm DGGA.

We augment the flow along C by decreasing the flow along the forward paths and increasing the flow along the backward paths by the same amount equal to $\min\{\varepsilon_1, \varepsilon_2\}$. The quantity $\varepsilon_1 > 0$, is the minimum flow along an edge on a forward path of the cycle. The second quantity, ε_2, is equal to $\min(d_j - f(e))$ where the minimum is taken over all edges $e = (u, v)$ lying on backward paths of the cycle and over all terminals t_j at v for which $d_j > f(e)$. If the minimum is achieved for an edge on a forward path then after the augmentation the flow on this edge vanishes and so the edge disappears from the graph. If the minimum is achieved for an edge (u, t_j) on a backward path, then after the augmentation the flow on (u, t_j) is equal to d_j. So, in this case, the edge (u, t_j) is removed from the graph after the terminal t_j is moved to u. Therefore, at least one edge is eliminated from the graph during each while-loop-iteration and the algorithm stops when the source s is isolated.

Analysis Overview. The correctness of the algorithm is based on the following two facts: the first is that at the beginning of any iteration, the in-degree of any vertex containing one or more terminals is at least 2; the second, which is a consequence of the first fact, is that as long as all terminals have not reached the source, the algorithm always finds an alternating cycle.

At each iteration, after augmentation either the flow on some forward edge vanishes and so the edge disappears from the graph or the flow on a backward edge (u, t_j) is equal to d_j and so the edge disappears from the graph after moving the terminal t_j to u, decreasing the flow on the edge (u, t_j) to zero and removing this edge from the graph. So, as a result of each iteration, at least one edge is eliminated and the algorithm makes progress.

Before an edge becomes a singular edge, the flow on it does not increase. After the edge becomes a singular edge we move at most one terminal along this edge and then this edge vanishes. Thus the total unsplittable flow through this edge is less than the sum of its initial flow and the maximum demand and the performance guarantee is at most 2. We refer the reader to [Dinitz et al. 1999] for more details.

Running Time. Since every augmentation removes at least one edge, the number of augmentations is at most m, where $m = |E|$. An augmenting cycle can be found in $O(n)$ time, where $n = |V|$. The time for moving terminals is $O(kn)$, where k denotes the number of terminals. Since there are k terminals, computing ε_2 requires $O(k)$ time in each iteration. Therefore the running time of the algorithm is $O(nm+km)$.

Heuristic Improvement. We have a second implementation with an added heuristic. The purpose of the heuristic is to try to reduce the congestion. The heuristic is designed so that it does not affect the theoretical performance guarantee of the original algorithm, but as a sacrifice, the running time is increased. In our second implementation, we use the heuristic only when we determine an alternating cycle. We always pick an outgoing edge with the smallest flow to go forward and choose an incoming edge with the largest flow to go backwards. The motivation behind the heuristic is as follows. Recall the definition of the quantity $\min\{\varepsilon_1, \varepsilon_2\}$ above by which we decrease (increase) flow along the forward (backward) edges of the alternating cycle. The heuristic aims to make this quantity as small as possible. Flow increases on singular edges are due to augmentations of this type. We would like these increases to be small, hence improving the congestion. One expects this should happen if we take conservative steps, i.e., move flow by small increments, during the process of gathering the flow of a commodity on a single path.

For most of the cases we tested, as we show in the experimental results in Section 5, the congestion is reduced somewhat. For some cases, it is reduced a lot. The running time for the new implementation with the heuristic is $O(dnm+km)$, where d is the maximum value of incoming and outgoing degrees among all vertices, since the time for finding an alternating cycle is now $O(dn)$.

3. THE $(3, 1)$-APPROXIMATION ALGORITHM FOR CONGESTION AND COST

The KSA [Kolliopoulos and Stein 2002; 1999] iteratively improves a solution by doubling the flow amount on each path utilized by a commodity which has not yet been routed unsplittably. In other words the "fractionality" of the flow is halved in every iteration. This scheme can be implemented to give a $(2, 1)$-approximation if all demands are powers of 2. Skutella's algorithm [Skutella 2002] combines this idea with a clever initialization which rounds down the demands to powers of 2 by removing the most costly paths from the solution. In Fig. 2 we give the algorithm for the case of powers of 2 [Kolliopoulos and Stein 2002; 1999] in the more efficient implementation of Skutella [2002]. The main idea behind the analysis of the congestion guarantee is that the total increase on an edge capacity across all iterations is bounded by $\sum_{l<\log(d_{\max}/d_{\min})} d_{\min}2^l \leq d_{\max}$.

Power-Algorithm:

```
    Input:   A directed graph $G = (V, E)$ with non-negative costs on the edges, a source
             vertex $s \in V$, $k$ commodities $i = 1, \cdots, k$ with terminals $t_i \in V \setminus \{s\}$ and
             positive demands $d_i = d_{\min} \cdot 2^{q_i}$, $q_i \in \mathbb{N}$, $q_1 \leq q_2 \leq \cdots \leq q_k$, and a
             (splittable) flow $f_0$ on $G$ satisfying all demands.
    Output:  An unsplittable flow given by a path $P_i$ from $s$ to each terminal $t_i$, $1 \leq$
             $i \leq k$.

    $i := 1; j := 0;$
    **while** $d_{\min} \cdot 2^j \leq d_{\max}$ **do**
        $j := j + 1$; $\delta_j := d_{\min} \cdot 2^{j-1}$;
        for every edge $e \in E$, set its capacity $u_e^j$ to $f_{j-1}(e)$ rounded up to the nearest
        multiple of $\delta_j$;
        compute a feasible $\delta_j$-integral flow $f_j$ satisfying all demands with $c(f_j) \leq$
        $c(f_{j-1})$;
        remove all edges $e$ with $f_j(e) = 0$ from $G$;
        **while** $i \leq k$ and $d_i = \delta_j$ **do**
            determine an arbitrary path $P_i$ from $s$ to $t_i$ in $G$;
            decrease $f_j$ along $P_i$ by $d_i$;
            remove all edges $e$ with $f_j(e) = 0$ from $G$;
            $i := i + 1$;
    **return** $P_1, \cdots, P_k$;
```

Fig. 2. The SA after all demands have been rounded down to powers of 2

Running time. The running time of the Power-Algorithm is dominated by the time to compute a δ_j-integral flow f_j in each while-loop-iteration j. Given the flow f_{j-1}, this can be done in the following way [Skutella 2002]. We consider the subgraph of the current graph G which is induced by all edges e whose flow value $f_{j-1}(e)$ is not δ_j-integral. Starting at an arbitrary vertex of this subgraph and ignoring directions of edges, we greedily determine a cycle C; this is possible since, due to flow conservation, the degree of every vertex is at least two. Then, we choose the *orientation of the augmentation* on C so that the cost of the flow is not increased. We augment flow on the edges of C whose direction is identical to the augmentation orientation and decrease flow by the same amount on the other edges of C until the flow value on one of the edges becomes δ_j-integral. We delete all δ_j-integral edges and continue iteratively. This process terminates after at most m iterations and has thus running time $O(nm)$. The number of while-loop-iterations is $1 + \log(d_{\max}/d_{\min})$. The running time of the first iteration is $O(nm)$ as discussed above. Still, since f_{j-1} is $(d_{\min} \cdot 2^{j-2})$-integral in each further iteration $j \geq 2$, the amount of augmented flow along a cycle C is $d_{\min} \cdot 2^{j-2}$ and after the augmentation the flow on each edge of C is $(d_{\min} \cdot 2^{j-1})$-integral and thus all edges of C will not be involved in the remaining cycle augmentation steps of this iteration. So the computation of f_j from f_{j-1} takes only $O(m)$ time. Moreover the path P_i can be determined in $O(n)$ time for each commodity i and the total running time of the Power-Algorithm is $O(kn + m \log(d_{\max}/d_{\min}) + nm)$.

We now present the General-Algorithm [Skutella 2002] which works for arbitrary demand values. In the remainder of the paper when we refer to the SA we mean the General-Algorithm. It constructs an unsplittable flow by rounding down the demand values such that the rounded demands satisfy the condition for using the Power-Algorithm. Then, the latter algorithm is called to compute paths $P_1, \cdots, P_k$. Finally, the original demand of commodity i, $1 \le i \le k$, is routed across path P_i. In contrast the KSA rounds demands *up* to the closest power of 2 before invoking the analogue of the Power-Algorithm.

We may assume that the graph is acyclic, which can be achieved by removing all edges with flow value 0 and iteratively reducing flow along directed cycles. This can be implemented in $O(nm)$ time using standard methods.

In the first step of the General-Algorithm, we round down all demands d_i to

$$\bar{d}_i := d_{\min} \cdot 2^{\lfloor \log(d_i/d_{\min}) \rfloor}.$$

Then, in a second step, we modify the flow f such that it only satisfies the rounded demands $\bar{d}_i$, $1 \le i \le k$. The algorithm deals with the commodities i one after another and iteratively reduces the flow f along the most expensive s-t_i-paths within f (ignoring or removing edges with flow value zero) until the inflow in node t_i has been decreased by $d_i - \bar{d}_i$. So, when we reroute this amount of reduced flow along any s-t_i-paths within the updated f, the cost of this part of the flow will not increase. Since the underlying graph has no directed cycles, a most expensive s-t_i-path can be computed in polynomial time. Notice that the resulting flow $\bar{f}$ satisfies all rounded demands. Thus, the Power-Algorithm can be used to turn $\bar{f}$ into an unsplittable flow $\tilde{f}$ for the rounded instance with $c(\tilde{f}) \le c(\bar{f})$. The General-Algorithm constructs an unsplittable flow $\hat{f}$ for the original instance by routing, for each commodity i, the total demand d_i (instead of only $\bar{d}_i$) along the path P_i returned by the Power-Algorithm and the cost of $\hat{f}$ is bounded by $c(\hat{f}) = c(\tilde{f}) + \sum_{i=1}^{k}(d_i - \bar{d}_i)c(P_i) \le c(\bar{f}) + \sum_{i=1}^{k}(d_i - \bar{d}_i)c(P_i) \le c(f)$.

Skutella [2002] shows that the General-Algorithm finds an unsplittable flow whose cost is bounded by the cost of the initial flow f and the flow value on any edge e is less than $2f(e) + d_{\max}$. Therefore, if the instance is balanced, i.e., the assumption that $d_{\max} \le u_{\min}$ is satisfied, an unsplittable flow whose cost is bounded by the cost of the initial flow and whose congestion is less than 3 can be obtained. Furthermore, if we use a minimum-cost flow algorithm to find a feasible splittable flow of minimum cost for the initial flow, the cost of an unsplittable flow obtained by the General-Algorithm is bounded by this minimum cost.

Running time. The procedure for obtaining $\bar{f}$ from f can be implemented to run in $O(m^2)$ time; in each iteration of the procedure, computing the most expensive paths from s to all vertices in the current acyclic network takes $O(m)$ time, and the number of iterations can be bounded by $O(m)$. Thus, the running time of the General-Algorithm is $O(m^2)$ plus the running time of the Power-Algorithm, i.e., $O(m^2 + kn + m \log(d_{\max}/d_{\min}))$. The first term can be usually improved using a suitable min-cost flow algorithm [Skutella 2002]. We examine this further in Section 5.

In our implementation, the variable δ_j adopts only the distinct rounded demand values. We have two reasons for doing that. The first is that it is not necessary for δ_j to adopt a value of the form $d_{\min} \cdot 2^i$ when it is not a rounded demand value and as a result of this we could have fewer iterations. The second reason is because of the following heuristic we intend to use.

Heuristic improvement. We have a second implementation of the SA in which we try to select augmenting cycles in a more sophisticated manner. When we look for an augmenting cycle in iteration j, at the current vertex we always pick an outgoing or incoming edge on which the flow value is not δ_j-integral and the difference between δ_j and the remainder of the flow value with respect to δ_j is minimal. The motivation is of a similar nature as with the heuristic described earlier for DGGA. The flow increments on an edge with respect to the original fractional solution contribute to the additive $d_{\max}$ term in the performance guarantee. Keeping these increases as small as possible by choosing to push flow along edges that are close to being δ_j-integral might improve the final congestion.

Unfortunately, the benefit of this heuristic seems to be very limited. We give details in Section 5. As mentioned above, in our implementation the variable δ_j adopts only the different rounded demand values. Since the time for finding an augmenting cycle in the implementation with the heuristic is $O(dn)$, where d is the maximum value of in- and outdegrees among all vertices, the worst-case running time for the implementation with the heuristic is $O(m^2 + dknm)$.

4. EXPERIMENTAL FRAMEWORK

Software and hardware resources. We conducted our experiments on a sun4u sparc SUNW Ultra-5_10 workstation with 640 MB of RAM and 979 MB swap space. The operating system was SunOS, release 5.8 Generic_108528-14. Our programs were written in C and compiled using gcc, version 2.95, with the -03 optimization option. A second round of experiments was performed at a later point in time on a different sparc platform that was available to us. The details for these experiments are in the Appendix.

Codes tested. The fastest maximum flow algorithm to date is due to Goldberg and Rao [1998] with a running time of $O(\min\{n^{2/3}, m^{1/2}\} m \log\left(n^2/m\right) \log U)$, where U is an upper bound on the edge capacities which are assumed to be integral. In practice, however, preflow-push [Goldberg and Tarjan 1988] algorithms are the fastest. We use the preflow-push Cherkassky-Goldberg code kit [Cherkassky and Goldberg 1997b] to find a maximum flow as an initial fractional flow. We assume integral capacities and demands in the unsplittable flow input. We implement and test the following codes:

`2alg`. This is the DGGA without any heuristic.

`2alg_h`. Version of `2alg` with the heuristic described in Section 2.

`3skut`. This is the SA without any heuristic.

`3skut_h`. Version of `3skut` with the heuristic described at the end of Section 3.

In addition we compare against the programs 3al and 3al2 used in [Kolliopoulos and Stein 1999], where 3al is an implementation of the KSA. The program 3al2 is an implementation of the same algorithm, where to improve the running time the edges carrying zero flow in the initial fractional solution are discarded. Note that both the DGGA and the SA discard these edges as well.

Input classes. We generated data from the same four input classes designed by Kolliopoulos and Stein [1999]. For each class we generated a variety of instances varying different parameters. The generators use randomness to produce different instances for the same parameter values. To make our experiments repeatable the seed of the pseudorandom generator is an input parameter for all generators. If no seed is given, a fixed default value is chosen. We used the default seed in generating all inputs. The four classes used are defined next. Whenever the term “randomly” is used in the following, we mean *uniformly at random.* For the inputs to 3skut and 3skut_h, we also generate randomly a cost value on each edge using the default seed.

genrmf. This is adapted from the GENRMF generator of Goldfarb and Grigoriadis [1988] [Badics 1991]. The input parameters are a b c1 c2 k d. The generated network has b frames (grids) of size $a \times a$, for a total of $a * a * b$ vertices. In each frame each vertex is connected with its neighbors in all four directions. In addition, the vertices of a frame are connected one-to-one with the vertices of the next frame via a random permutation of those vertices. The source is the lower left vertex of the first frame. Vertices become sinks with probability $1/k$ and their demand is chosen uniformly at random from the interval $[1, d]$. The capacities are randomly chosen integers from $(c1, c2)$ in the case of interframe edges, and $(1, c2 * a * a)$ for the in-frame edges.

noigen. This is adapted from the noigen generator used in [Chekuri et al. 1997; Nagamochi et al. 1994] for minimum cut experimentation. The input parameters are n d t p k. The network has n nodes and $\lfloor n(n-1)d/200 \rfloor$ edges. Vertices are randomly distributed among t components. Capacities are chosen uniformly from a prespecified range $[l, 2l]$ in the case of intercomponent edges and from $[pl, 2pl]$ for intracomponent edges, p being a positive integer. Only vertices belonging to one of the $t-1$ components not containing the source can become sinks, each with probability $1/k$. The desired effect of the construction is for commodities to contend for the light intercomponent cuts. Demand for commodities is chosen uniformly form the range $[1, 2l]$.

rangen. This generates a random graph $G(n, p)$ with input parameters n p c1 c2 k d, where n is the number of nodes, p is the edge probability, capacities are in the range $(c1, c2)$, k is the number of commodities and demands are in the range $(0, d)$.

satgen. It first generates a random graph $G(n, p)$ as in rangen and then uses the following procedure to designate commodities. Two vertices s and t are picked from G and the maximum flow is computed from s to t. Let v be the value of the flow. New nodes corresponding to sinks are incrementally added each connected only to t and with a randomly chosen demand value. The process of adding new sinks stops when the total demand reaches v (i.e., the value of the minimum s-t cut), or when the total number of commodities reaches the input parameter k. Therefore k defines a crude upper bound on the number of commodities.

	1		2		6		12		25		50	
program	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time
2alg	1.40	0	1.50	0	1.63	0	1.63	0	1.75	0	1.75	1
2alg_h	1.33	0	1.50	0	1.42	0	1.60	0	1.75	0	1.64	0
3al	1.55	1	1.56	1	1.67	4	1.64	9	1.64	23	1.70	54
3al2	1.45	0	1.67	1	1.78	2	1.57	5	1.67	13	1.75	32
3skut	1.70	0	1.64	1	2.22	1	2.11	1	2.11	2	2.33	6
3skut_h	1.70	1	1.64	0	2.22	1	2.11	1	2.11	2	2.33	6

Table I. family sat_density: satgen -a 1000 -b i -c1 8 -c2 16 -k 10000 -d 8; 9967, 20076, 59977, 120081, 250379, 500828 edges; 22, 61, 138, 281, 682, 1350 commodities; balanced family; i is the expected percentage of pairs of vertices joined by an edge.

	2		6		12		25		50		100	
program	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time
2alg	1.30	0	2.37	0	2.14	0	1.43	0	1.26	0	1.09	0
2alg_h	1.30	0	1.73	0	2.14	0	1.49	0	1.17	0	1.09	0
3al	2.21	0	1.73	0	2.41	1	1.70	1	1.26	1	1.14	1
3al2	2.21	0	1.88	0	2.29	0	1.52	1	1.27	1	1.18	1
3skut	1.40	0	1.65	0	2.15	0	1.85	0	1.45	0	1.49	1
3skut_h	1.40	0	1.65	0	2.15	0	1.85	0	1.45	0	1.49	0

Table II. family noi_commodities: noigen 1000 1 2 10 i; 7975 edges; 2-unbalanced family; i is the expected percentage of sinks in the non-source component.

	2		5		10		20		50		70	
program	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time
2alg	2.71	8	1.77	14	1.41	15	1.21	21	1.09	33	1.06	42
2alg_h	2.47	8	1.75	14	1.36	15	1.19	19	1.08	31	1.06	42
3al	2.76	6	1.89	13	1.57	23	1.37	49	1.19	138	1.14	196
3al2	2.79	4	1.89	7	1.54	14	1.30	33	1.16	91	1.15	132
3skut	3.35	1	2.39	1	2.07	2	1.76	1	1.63	2	1.62	3
3skut_h	3.35	1	2.39	1	2.07	1	1.76	2	1.63	2	1.62	3

Table III. family rmf_commDem: genrmf -a 10 -b 64 -c1 64 -c2 128 -k i -d 128; 29340 edges; 2-unbalanced family; i is the expected percentage of sinks among the vertices.

	32		64		128		256		512		1024	
program	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time
2alg	4.67	0	3.75	0	7.50	0	7.00	0	7.50	0	8.00	0
2alg_h	2.50	0	3.25	0	6.50	0	7.00	0	7.50	1	7.50	0
3al	4.67	0	8.00	0	8.00	1	8.00	6	7.50	30	7.50	136
3al2	4.67	0	6.50	0	7.50	0	8.50	3	7.50	17	8.00	82
3skut	5.00	0	7.00	0	7.50	0	7.00	1	7.50	4	7.50	19
3skut_h	5.00	0	7.00	0	7.50	0	7.00	1	7.50	4	7.50	18

Table IV. family ran_dense: rangen -a i*2 -b 30 -c1 1 -c2 16 -k i -d 16; 1182, 4807, 19416, 78156, 313660, 1258786 edges; 16-unbalanced family; i*2 is the number of vertices.

5. EXPERIMENTAL RESULTS

In this section we give an overview of the experimental results. In all algorithms we study, starting with a different fractional flow may give different unsplittable solutions. Hence in order make a meaningful comparison of the experimental results of the SA against the results of the DGGA and KSA, we use the same initial fractional flow for all three. If the SA was used in isolation, one could use, as mentioned in Section 3, a min-cost flow algorithm to find the initial fractional flow and therefore obtain a best possible budget.

The implementations follow the algorithm descriptions as given earlier. In the case of the SA, after finding the initial fractional flow f, one has to iteratively reduce flow, for each commodity i, along the most expensive s-t_i-paths used by f until the inflow in terminal t_i has been decreased by $d_i - \bar{d}_i$, where $\bar{d}_i$ stands for the rounded demand. Instead of doing this explicitly, as Skutella [2002] suggests, we set the capacity of each edge e to $f(e)$ and use an arbitrary min-cost flow algorithm to find a min-cost flow that satisfies the rounded demands. Because of this, the term $O(m^2)$ in the running time of Algorithm 3 in Section 3 can be replaced by the running time of an arbitrary min-cost flow algorithm. The running times of the currently best known min-cost flow algorithms are $O(nm\log(n^2/m)\log(nC))$ [Goldberg and Tarjan 1990], $O(nm(\log\log U)\log(nC))$ [Ahuja et al. 1992], and $O((m\log n)(m+n\log n))$ [Orlin 1993]. The code we use is again due to Cherkassky and Goldberg [1997a]. The experimental results for all the implementations are given in Tables I–VII. The wall-clock running time is given in seconds, with a running time of 0 denoting a time less than a second. We gave earlier the theoretical running times for the algorithms we implement but one should bear in mind that the real running time depends also on other factors such as the data structures used. Apart from standard linked and adjacency lists no other data structures were used in our codes. As mentioned in the introduction, speeding up the codes was not our primary focus. This aspect could be pursued further in future work.

The DGGA vs. the KSA. We first compare the results of the 2- and the 3-approximation algorithms since they are both algorithms for congestion without costs. On a balanced input (see Tables I, IX, X and XI), the congestion achieved by the DGGA, with or without heuristics, was typically less than or equal to 1.75. The congestion achieved by the KSA was almost in the same range. For each balanced input, the difference in the congestion achieved by these two algorithms was small, but the DGGA's congestion was typically somewhat better. The obvious difference occurred in running time. Before starting measuring running time, we use the Cherkassky-Goldberg code kit to find a feasible splittable flow (if necessary we use other subroutines to scale up the capacities by the minimum amount needed to achieve a fractional congestion of 1), and then we create an array of nodes to represent this input graph (discarding all zero flow edges) and delete all flow cycles. After that, we start measuring the running time and applying the DGGA. The starting point for measuring the running time in the implementation of the KSA is also set after the termination of the Cherkassky-Goldberg code.

To test the robustness of the approximation guarantee we relaxed on several instances the balance assumption by allowing the maximum demand to be twice the minimum capacity or more, see Tables II–VI. Taking into account the analysis of the algorithms from Sections 2 and 3 one can quantify robustness as follows: making the family ρ-unbalanced should yield a performance guarantee less than $1 + \rho$ for DGGA and less than $2 + \rho$ for KSA and SA. Even in the extreme case when the balance assumption was violated by a factor of 16, as in Table IV, the code `2alg` achieved 8 and the code `2alg_h` achieved 7.5. Relatively speaking though the absolute difference in the congestion achieved between `2alg, 2alg_h` and `3al, 3al2` is more pronounced in Table IV compared to the inputs with small imbalance in Tables II, III. See the big differences in the second column of Table IV (in Column 2, `2alg`: 3.75 and `3al2`: 6.5). The differences in Columns 3 and 4 of Table IV are also sizable. Recall that an absolute difference of 1 (or even 0.5) on the congestion of an edge e translates to an excess flow equal to 100% (50%) of the original edge capacity u_e. See Table X in the Appendix for a balanced version of the ran_dense family. Hence the DGGA seems more robust against highly unbalanced inputs. This is consistent with the behavior of the KSA which keeps increasing the edge capacities by a fixed amount in each iteration before routing any new flow. In contrast, the DGGA increases congestion only when some flow is actually rerouted through an edge. As shown in Table IV, the SA which behaves similarly to the KSA behaves also less robustly for highly unbalanced inputs.

This effect was reproduced at a much smaller scale in Table VIII where the comparison should be made with the balanced instances of Table I. See the Appendix for a detailed comparison. All the algorithms behave in a reasonably robust manner: making the family 2-unbalanced keeps the congestion of DGGA well below 3 and that of KSA and SA well below 4. Nevertheless the differences between DGGA and KSA are much smaller compared to Table IV as the graph becomes denser. This should come as no surprise given the specific design of the sat_density family. The total demand saturates a cut that separates the source from the sink. Therefore all algorithms are expected to behave less robustly on this family when the input becomes unbalanced.

We also observed that the benefit of the heuristic used in our `2alg_h` implementation showed up in our experimental results. For most of the inputs, the congestion was improved, although rarely by more than 5%. Considerable improvements were obtained for unbalanced dense inputs, see Tables IV and VIII. The average improvement over the six columns of Table VIII was 11.79%. Theoretically, the running time for the program with the heuristic should increase by a certain amount. But in our experiments, the running time stayed virtually the same. This phenomenon was beyond what we expected.

In summary, the DGGA performs typically better than the KSA for congestion. With the occasional exception where the KSA outperformed the DGGA, the average improvement for Tables I–IV is 7.19%, 12.98%, 8.63%, and 30.47%. We calculated these numbers as follows. For a single column of a table we took c_1 to be the best congestion achieved between `2alg, 2alg_h` and c_2 the best congestion achieved between `3al, 3al2` and computed $(c_2 - c_1)/c_2$. Then we took the average over all the columns of the table where $c_2 > c_1$. For example there are five such columns in Table II and four such columns in Table IV. The outlined behavior is consistent with the fact that the DGGA is a theoretically better algorithm, although the theoretical advantage translates to a much smaller advantage in practice. The improvement effect weakens considerably on large, dense networks. Consider the last two columns from each of the Tables VIII–XI. Only on half of them DGGA beats KSA. Over those cases the average improvement is 6.35%. KSA routes unsplittably by computing max flows for subproblems containing commodities whose demands are close in value. This global perspective as opposed to the localized processing of DGGA might be beneficial in graphs with a rich path structure.

The difference in the running time for these two approximation algorithms was fairly significant in our experiments especially for dense graphs with a large number of commodities. The DGGA runs much faster than the implementation of the KSA we used. We proceed to give two possible reasons for this phenomenon.

The first reason is the difference in complexity for these two implementations. Recall that the running time of the DGGA is $O(mn + km)$ and the running time of the implementation of the KSA that we used is $O(k(d_{\max}/d_{\min})m)$ [Kolliopoulos and Stein 2002; 1999]. We emphasize that a polynomial-time implementation is possible (see [Kolliopoulos and Stein 2002; 1999]). In fact Skutella's Power-Algorithm can be regarded as a much more efficient implementation of essentially the same algorithm. The second reason is that the DGGA processes the graph in a localized manner, i.e., finding an alternating cycle locally and increasing a certain amount of flow on it, while the `3al` and `3al2` codes repeatedly compute maximum flows on the full graph.

	3		6		12		24		48	
program	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time
2alg	1.10	0	1.07	0	1.04	0	1.01	0	1.01	0
2alg_h	1.12	0	1.05	0	1.02	0	1.01	0	1.01	0
3al	1.22	2	1.13	1	1.12	1	1.07	1	1.08	1
3al2	1.22	1	1.13	0	1.11	1	1.04	1	1.09	1
3skut	1.46	0	1.43	0	1.37	0	1.37	0	1.38	0
3skut_h	1.46	0	1.43	0	1.37	0	1.37	0	1.38	0

Table V. family noi_components: noigen 1000 1 i 10 50; 7975 edges; 2-unbalanced family; i is the number of components.

The Skutella algorithm. We now examine the congestion achieved by the SA. On a balanced input (see Tables I, IX, X and XI), the congestion achieved by the SA was typically greater than or equal to 1.64 and less than or equal to 2.33. The corresponding ranges are $[1.08, 1.75]$ for the DGGA and $[1.15, 1.78]$ for the KSA. More

	2		4		16		64	
program	cong.	time	cong.	time	cong.	time	cong.	time
2alg	2.30	0	2.47	0	1.36	1	1.11	28
2alg_h	1.89	0	2.64	0	1.37	1	1.09	27
3al	2.26	0	2.18	0	1.48	5	1.21	103
3al2	1.82	0	2.72	0	1.46	3	1.20	77
3skut	2.72	0	2.71	0	1.87	1	1.72	1
3skut_h	2.72	0	2.71	0	1.87	0	1.72	2

Table VI. family rmf_depthDem: genrmf -a 10 -b i -c1 64 -c2 128 -k 40 -d 128; 820, 1740, 7260, 29340 edges; 2-unbalanced family; i is the number of frames.

	1		2		3		4		5		6	
program	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time
3skut	2.22	6	1.26	0	1.50	3	5.33	18	1.19	0	1.61	2
3skut_h	2.22	6	1.28	0	1.48	3	5.33	18	1.20	0	1.58	3

Table VII. Effect of our heuristic on the SA. Here the input instances in Columns 1 to 6 are the modified input instances in the last columns of Tables 1 to 6 whose original demands, denoted by d, are modified as follows to the value d': $d' = 1$ if $d = 2$; $d' = 2^2$ if $d = 3$; $d' = 2^4$ if $8 \leq d < 16$; $d' = 2^6$ if $32 \leq d < 64$; for all other cases, d are not changed. Note that the maximum demand value in our input instances is equal to $128 = 2^7$.

precisely, the absolute difference in congestion between the SA and the DGGA or KSA is on average around 0.4. We think that this nontrivial difference in congestion is partially caused by the involvement of costs on the edges and the simultaneous performance guarantee of 1 for cost of the SA. The constraint that the flow found at each step should not increase the cost limits the routing options. Another reason for the higher congestion was suggested by an anonymous referee: during the essential part of the algorithm where routing decisions are being made, SA underestimates the actual demand of commodities since demands have been rounded down. In that sense, the algorithm is overly optimistic and ignores the problem caused by rounding. In contrast, DGGA only works with the original demands and KSA is overly pessimistic during its routing decisions since demands have been rounded up.

In the implementation of the $(3, 1)$-approximation algorithm we start measuring the running time just before applying a min-cost flow algorithm [Cherkassky and Goldberg 1997a] to find a min-cost flow for the rounded demands. Before that starting point of running time, we use the Cherkassky-Goldberg code kit to find a feasible splittable flow (if necessary, as we did before, we use other subroutines to scale up the capacities by the minimum amount to get the optimal fractional congestion), and then we create the input data for the min-cost flow subroutine, i.e., setting the capacity of each edge to its flow value and the demand of each commodity i to $\bar{d}_i$. For balanced input instances in Table I, the running time of the SA is much better than that of the KSA but slightly more than that of the DGGA. Actually, as we observed in testing, most of the running time for the SA is spent in finding the initial min-cost flow.

To test the robustness of the approximation guarantee achieved by the SA we used the instances with the relaxed balanced assumption. Even in the extreme case when the balance assumption was violated by a factor of 16, as in Table IV, the code

`3skut` achieved 7.50. Similarly to DGGA and KSA a smaller degree of robustness was also exhibited on the 2-unbalanced sat_density family in Table VIII.

The absolute difference in congestion achieved by the codes `2alg`, `3al` and `3skut` is typically small. The only big difference occurred in the second output column in Table IV (`2alg`: 3.75, `3al`: 8.00 and `3skut`: 7.00). Still, similar to the output in Table I, the congestion achieved by the codes `2alg` and `3al` for an unbalanced input was typically better, see Tables II–VI. Given the similarities between the KSA and SA the reason is, as mentioned above, the involvement of costs on the edges and a simultaneous performance guarantee of 1 for cost in the $(3, 1)$-approximation algorithm. For the running time, things are different. We can see from Tables III and VI that the code `3skut` runs much faster than `2alg` and `3al` when the size of the input is large. This is probably because after the rounding stage the number of the distinct rounded demand values, which is the number of iterations in our implementation, is small (equal to 7 in Tables III and VI) and the number of augmenting cycles (to be chosen iteratively) in most of the iterations is not very large. If this is the case, the execution of these iterations could be finished in a very short period of time and the total running time is thus short too.

Effect of the heuristic on the SA. No benefit of the heuristic used in our `3skut_h` implementation showed in Tables I–VI and VIII–XI. This is because in each iteration (except the stage of finding a min-cost flow) the non-zero remainder of flow value on each edge with respect to the rounded demand value of the current iteration is exactly the same in our input instances. More precisely, in our input instances, the variable δ_j adopts all values $d_{\min} \cdot 2^i$ between $d_{\min}$ and $d_{\max}$, and in this case, in iteration j the remainder of flow value on any edge with respect to $\delta_j = d_{\min} \cdot 2^{j-1}$ is either $d_{\min} \cdot 2^{j-2}$ or 0. So the amount of augmented flow along an augmenting cycle C is $d_{\min} \cdot 2^{j-2}$ and after the augmentation the flow on each edge of C is δ_j-integral and thus all edges of C will not be involved in the remaining augmentation procedure of this iteration. This is also probably the reason why sometimes the SA runs faster than the DGGA. When the variable δ_j would not adopt all values $d_{\min} \cdot 2^i$ between $d_{\min}$ and $d_{\max}$, the heuristic proved to be of some marginal usefulness. This can be seen in Table VII. The congestion was improved in Columns 3 and 6 by 0.02 and 0.03, respectively, but in Columns 2 and 5 the congestion increased by 0.02 and 0.01, respectively.

Effect of the number of commodities on the three algorithms. A noteworthy phenomenon that applies to all the three algorithms is that on several instances the performance seemed in general to improve as the number of commodities increases (cf. Tables II and III). One might think that this is due to the fact that for a small number of commodities, a large part of the topology of the graph remains completely unexploited by the fractional solution. At least for DGGA and SA once some edge is not used in the fractional solution it becomes irrelevant. A closer look reveals that on dense graphs, the performance actually becomes somewhat worse as the number of commodities increases. Compare Tables IX and Table I and Tables XI and X and see also the detailed discussion in the Appendix. We believe that both these opposing effects are explained by the fractional solution which forms each time the basis of the approximation algorithm. A fractional solution which is constrained in its choice of routes (as for example in the case of a sparse graph with many sinks that compete for the same paths) is a better basis for computing an unsplittable flow compared to a fractional solution which uses a wide array of paths (as for example in the case of a dense graph with many sinks). One would think that in terms of the unsplittable problem itself, the latter scenario should be more tractable. Existing methods, however, offer no way to attack the unsplittable problem directly without going through the fractional flow first.

In summary, in most of our experiments the DGGA and KSA achieved lower congestion than the SA. Relative gains of the order of 35% or more are common especially for Tables I, III and IV. Given the similarity of the KSA and SA this is because (i) the SA has a simultaneous performance guarantee for the cost which constrains the choice of paths and (ii) the SA underestimates the actual demand of commodities since it makes routing decisions with the rounded down values. The SA remains competitive and typically achieved approximation ratios well below the theoretical guarantee. The `3skut` code runs much faster than `3al` and occasionally faster than the `2alg` code.

6. DISCUSSION

The relative performance of the three main codes with respect to congestion could be expressed by the following crude ranking: DGGA, KSA, SA. This behavior is by and large consistent with the theoretical analysis and specific properties of SA as explained in Section 5. One should, however, bear in mind all the nuances and caveats presented. For example, the advantage of DGGA over KSA weakens considerably on dense graphs with a large number of commodities. On the instances tested all three algorithms perform quite well with respect to congestion, noticeably better than the theoretical bounds. With respect to running time, a secondary aspect of this work, the comparison between KSA and SA was useful. The superiority of the implementation proposed by Skutella [Skutella 2002] which we actually used in the SA code, against the inefficient implementation used in the KSA code [Kolliopoulos and Stein 1999] was overwhelming.

A noteworthy phenomenon seems to be the effect of the fractional solution on the quality of the unsplittable solution obtained. This raises two interesting questions. (i) How can one search the graph for candidate paths without using exclusively the fractional solution as a guide? This issue could be of theoretical interest as well. Results in this vein exist for the multiple-source problem [Kolliopoulos 2005]. (ii) For instances with a small number of commodities, would it make sense to generate at random dummy commodities before computing the unsplittable flow? The answers to these two questions could give rise to sophisticated heuristics with an impact on the practical performance.

Acknowledgement. We thank the two anonymous referees for constructive criticism.

REFERENCES

Ahuja, R. K., Goldberg, A. V., Orlin, J. B., and Tarjan, R. E. 1992. Finding Minimum-Cost Flows by Double Scaling. *Mathematical Programming 53*, 243–266.

Badics, T. 1991. genrmf. `ftp://dimacs.rutgers.edu/pub/netflow/generators/network/ - genrmf/`.

Chekuri, C., Goldberg, A. V., Karger, D., Levine, M., and Stein, C. 1997. Experimental study of minimum cut algorithms. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*. 324–333.

Cherkassky, B. V. and Goldberg, A. V. 1997a. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms 22*, 1–29.

Cherkassky, B. V. and Goldberg, A. V. 1997b. On implementing the push-relabel method for the maximum flow problem. *Algorithmica 19*, 390–410.

Dinitz, Y., Garg, N., and Goemans, M. X. 1999. On the single-source unsplittable flow problem. *Combinatorica 19*, 1–25.

Erlebach, T. and Hall, A. 2002. NP-hardness of broadcast sheduling and inapproximability of single-source unsplittable min-cost flow. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*. 194–202.

Goldberg, A. V. and Rao, S. 1998. Beyond the flow decomposition barrier. *Journal of the ACM* 45, 783–797.

Goldberg, A. V. and Tarjan, R. E. 1988. A new approach to the maximum flow problem. *Journal of the ACM 35,* 4 (Oct.), 921–940.

Goldberg, A. V. and Tarjan, R. E. 1990. Solving minimum-cost flow problems by successive approximation. *Mathematics of Operations Research 15,* 3, 430–466.

Goldfarb, D. and Grigoriadis, M. 1988. A computational comparison of the Dinic and Network Simplex methods for maximum flow. *Annals of Operations Research 13*, 83–123.

Kleinberg, J. M. 1996a. Approximation algorithms for disjoint paths problems. Ph.D. thesis, MIT, Cambridge, MA.

Kleinberg, J. M. 1996b. Single-source unsplittable flow. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*. 68–77.

Kolliopoulos, S. G. 2005. Edge-disjoint paths and unsplittable flow. Chapter to appear in the *Handbook of Approximation Algorithms and Metaheuristics* edited by T. F. Gonzalez.

Kolliopoulos, S. G. and Stein, C. 1999. Experimental evaluation of approximation algorithms for single-source unsplittable flow. In *Proceedings of the 7th Integer Programming and Combinatorial Optimization Conference, Springer-Verlag LNCS*. Vol. 1610. 153–168.

Kolliopoulos, S. G. and Stein, C. 2002. Approximation algorithms for single-source unsplittable flow. *SIAM J. Computing 31,* 3, 919–946.

Lenstra, J. K., Shmoys, D. B., and Tardos, E. 1990. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming 46*, 259–271. Preliminary version in Proc. FOCS 1987.

Nagamochi, H., Ono, T., and Ibaraki, T. 1994. Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming 67*, 325–241.

Orlin, J. B. 1993. A faster strongly polynomial minimum cost flow algorithm. *Operations Research 41*, 338–350.

Skutella, M. 2002. Approximating the single-source unsplittable min-cost flow problem. *Mathematical Programming* Ser. B 91(3), 493–514.

Appendix

The additional experiments in this section were conducted in order to gain more insight into the effect of the balance condition and the graph density on the performance of the algorithms. We ran this second set of experiements on a different sparc platform that was available to us: a sun4u sparc SUNW Ultra-Enterprise workstation with 2 GB of RAM and 5.8 GB swap space. The operating system was SunOS, release 5.8 Generic_117350-20. Our programs were written in C and compiled using gcc, version 2.95, with the -03 optimization option. We report running

	1		2		6		12		25		50	
program	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time
2alg	1.89	0	2.11	0	2.63	0	2.75	0	2.75	0	2.63	0
2alg_h	1.78	0	1.91	0	1.92	0	2.50	0	2.50	1	2.33	0
3al	2.08	1	2.42	1	2.45	4	2.73	8	2.64	21	2.73	56
3al2	2.00	0	2.42	0	2.27	2	2.50	5	2.55	11	2.73	30
3skut	2.11	0	2.44	1	2.67	0	2.78	0	3.00	2	3.33	7
3skut_h	2.11	0	2.44	0	2.67	1	2.78	1	3.00	2	3.33	7

Table VIII. family sat_density: satgen -a 1000 -b i -c1 8 -c2 16 -k 10000 -d 16; 9957, 20045, 59904, 119937, 250033, 500155 edges; 12, 30, 65, 137, 336, 677 commodities; 2-unbalanced family; i is the expected percentage of pairs of vertices joined by an edge.

	1		2		6		12		25		50	
program	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time
2alg	1.38	0	1.33	0	1.50	0	1.75	0	1.56	0	1.56	0
2alg_h	1.38	0	1.33	0	1.33	0	1.56	0	1.44	0	1.50	0
3al	1.30	0	1.55	1	1.44	3	1.73	7	1.56	19	1.73	47
3al2	1.40	0	1.50	1	1.56	2	1.73	4	1.62	11	1.56	28
3skut	1.36	0	1.64	0	2.00	0	2.00	0	2.22	2	2.11	5
3skut_h	1.36	0	1.64	0	2.00	0	2.00	1	2.22	2	2.11	5

Table IX. family sat_density: satgen -a 1000 -b i -c1 8 -c2 16 -k 10*i -d 8; 9955, 20035, 59899, 119920, 249947, 499978 edges; 10, 20, 60, 120, 250, 500 commodities; balanced family; i is the expected percentage of pairs of vertices joined by an edge.

	32		64		128		256		512		1024	
program	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time
2alg	1.40	0	1.46	0	1.50	0	1.75	0	1.63	0	1.75	1
2alg_h	1.40	0	1.45	0	1.40	0	1.75	0	1.63	0	1.67	0
3al	1.63	0	1.67	0	1.56	1	1.44	3	1.63	13	1.67	64
3al2	1.50	0	1.56	0	1.44	1	1.63	0	1.56	3	1.67	12
3skut	1.75	0	1.75	0	1.79	0	1.88	0	2.00	3	2.22	18
3skut_h	1.75	0	1.75	0	1.79	0	1.88	0	2.00	4	2.22	17

Table X. family ran_dense: rangen -a i*2 -b 30 -c1 8 -c2 16 -k i -d 8; 1182, 4807, 19416, 78156, 313660, 1258786 edges; balanced family; i*2 is the number of vertices; the number of commodities is i.

times for the sake of completeness. Because of the platform difference they are not directly comparable to the running times reported in the main body of the paper.

Tables VIII and IX are closely related to Table I. The input instances in Table VIII, were generated by changing the value of the parameter d from 8 to 16, so they form a 2-unbalanced family. We obtained the input instances in Table IX by changing the upper bound on the number of commodities, i.e., the value of the parameter k, from 10000 to $10 * i$, where $i = 1, 2, 6, 12, 25$, and 50. So the total demand in each instance in this table is below the value of the min-cut separating the source from the sinks. We give now some detailed comparisons.

Table I vs. Table VIII: On a 2-unbalanced input, the congestion achieved by the DGGA, with or without heuristics, was less than or equal to 2.75, the KSA congestion was at most 2.73, and the SA congestion at most 3.33, compared with the maximum congestions 1.75 by DGGA, 1.78 by KSA, and 2.33 by SA in Table I.

	32		64		128		256		512		1024	
program	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time	cong.	time
2alg	1.08	0	1.56	0	1.50	0	1.42	0	1.56	0	1.63	0
2alg_h	1.13	0	1.30	0	1.22	0	1.30	0	1.38	0	1.50	0
3al	1.15	0	1.30	0	1.45	1	1.44	3	1.67	13	1.56	61
3al2	1.56	0	1.44	0	1.44	1	1.38	0	1.36	3	1.56	11
3skut	1.50	0	1.79	0	1.38	0	2.00	0	2.22	3	2.00	14
3skut_h	1.50	0	1.79	0	1.38	0	2.00	0	2.22	2	2.00	14

Table XI. family ran_dense: rangen -a i*2 -b 30 -c1 8 -c2 16 -k i/2 -d 8; 1182, 4807, 19416, 78156, 313660, 1258786 edges; balanced family; i*2 is the number of vertices; the number of commodities is i/2.

So the upper bound of the congestion increased by about an additive 1 as a result of a 2-unbalanced input. The average congestion changed from 1.61 and 1.54 for DGGA with and without heuristics, 1.63 and 1.65 for KSA, and 2.02 for SA in Table I to 2.46 and 2.16 for DGGA, 2.51 and 2.41 for KSA, and 2.72 for SA in Table VIII. These values increased about 0.8.

Table I vs. Table IX: The largest congestions for DGGA, KSA and SA in Table IX are 1.75, 1.73, and 2.22, respectively, which are almost the same as the largest congestions 1,75, 1.78 and 2.33 achieved in Table I. The average congestions in Table IX are 1.51 and 1.42 for DGGA, 1.55 and 1.56 for KSA, and 1.89 for SA, which are a little bit lower than 1.61 and 1.54 for DGGA, 1.63 and 1.65 for KSA and 2.02 for SA achieved in Table I. So the congestion did not improve much although we made the total demand quite below the min-cut by cutting down the number of commodities by more than 50%.

Tables X and XI represent balanced versions of the ran_dense family. Recall that Table IV represents a 16-unbalanced version of the ran_dense family. Since all the input instances in Tables X and XI are balanced, the congestion falls in the normal range of the corresponding algorithm's guarantee. Still, the congestion in Table XI is somewhat better than that in Table X. Obviously this is because the number of commodities in Table XI is half of that in Table X.

We focus now on the impact of the density of the underlying graph on the performance. We compare Tables IX–XI, which all represent balanced instances on dense graphs against Tables II, III and V which all represent 2-unbalanced instances on sparse graphs. A special mention is due to the differences between Tables II and V. Both belong to the noi_commodities family but in Table V we increase the number of components. This creates a separation effect for the commodities; paths have to go through the bottleneck of the intercomponent edges. This decreases the number of path options available to the fractional solution. Therefore we believe it is natural that the Table V instances give us better performance than the comparable last two columns of Table II.

Let us return to the sparse vs. dense comparison. The algorithms behave in a consistent manner on dense instances, with smaller variation in performance as the size of the graph and the number of commodities increases. Across all columns in Tables IX–XI, the congestion of `2alg` ranges in the interval $[1.08, 1.75]$ which becomes $[1.33, 1.75]$ if we exclude the first column of the Table XI. The corresponding intervals have length at least 1 for Tables II and III. Decreasing the number of commodities in a dense graph (compare columnwise Table IX vs. Table I and Table XI vs. Table X) improves the congestion by a small amount. The opposite effect takes place in the sparse instances of Table II and III (compare in each table the columns against each other) where the congestion improves considerably as the number of commodities increases. It does seem then that in graphs with rich path structure the performance worsens as the number of commodities increases. A possible explanation is that the fractional solution has a tendency to "spread out" exploiting the many path options that are available. If this happens, transforming it to an unsplittable one will inevitably hit some of the edges with much more flow than the amount they carry in the fractional solution.

Finally, we remark that for the purposes of the dense vs. sparse comparison, we deliberately "penalized" the sparse instances by letting them be 2-unbalanced. The performance of an algorithm can only get worse on unbalanced instances. Therefore our conclusion that the congestion on sparse instances improves as the number of commodities increases is amplified by the input imbalance.