Parallel Pipelined Filter Ordering with Precedence Constraints

Amol Deshpande [#1], Lisa Hellerstein [*2]
[#] *University of Maryland, College Park, MD USA*
[1] amol@cs.umd.edu
[*] *Polytechnic Institute of NYU, Brooklyn, NY USA*
[2] hstein@cse.poly.edu

In the parallel pipelined filter ordering problem, we are given a set of n filters that run in parallel. The filters need to be applied to a stream of elements, to determine which elements pass all filters. Each filter has a *rate limit* r_i on the number of elements it can process per unit time, and a *selectivity* p_i, which is the probability that a random element will pass the filter. The goal is to maximize throughput. This problem appears naturally in a variety of settings, including parallel query optimization in databases and query processing over Web services.

We present an $O(n^3)$ algorithm for the above problem, given tree-structured precedence constraints on the filters. This extends work of Condon et al. and Kodialam, who presented algorithms for solving the problem without precedence constraints. Our algorithm is combinatorial and produces a sparse solution. Motivated by join operators in database queries, we also give algorithms for versions of the problem in which "filter" selectivities may be greater than or equal to 1.

We prove a strong connection between the more classical problem of minimizing total work in sequential filter ordering (A), and the parallel pipelined filter ordering problem (B). More precisely, we prove that **A** is solvable in polynomial time for a given class of precedence constraints if and only if **B** is as well. This equivalence allows us to show that **B** is NP-Hard in the presence of arbitrary precedence constraints (since **A** is known to be NP-Hard in that setting).

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms: Algorithms

Additional Key Words and Phrases: pipelined filter ordering; parallel databases; query optimization; flow algorithms; sequential testing

1. INTRODUCTION

In answering a database query, it is often necessary to determine which tuples satisfy a given set of predicates. In a factory setting, it is common to subject manufactured items to a series of quality tests, to determine which of the items pass all of the tests. In both cases, the goal is to determine which elements satisfy a given set of filters. The *filter ordering* problem addresses the problem of ordering the application of the filters so as to achieve this goal as efficiently as possible.

In this paper, we are concerned with filter ordering when the filters can operate in parallel, but have limits on their capacity. This problem arises naturally in many settings. In the context of combining information from multiple Web services, the Web services may have pre-specified rate limits on the number of queries that they will answer per second, and the goal is to maximize the number of tuples that can be processed per second (cf. Srivastava et al. [2006]). Similarly, in parallel database query optimization, different operators in the query may be evaluated on different processors, to achieve higher scalability. In a factory, tests on manufactured items may be executed at different stations on the factory floor. In all these cases, we would like to choose the operator or test order to ensure that the processing power of the system as a whole is not wasted.

Specifically, we consider the problem of maximizing throughput in *parallel pipelined filter ordering*. There are n filters. Each filter is executed on a separate processor. There is a stream of elements that must be passed through the filters. A filter can only be applied to one element at a time. As soon as an element does not pass a filter, it is discarded. If an element passes a filter, it is either output (if it has passed all filters) or sent on to be tested by another filter. The filters operate in parallel. Each filter i has a known *rate limit* $r_i > 0$, which is a limit on the (expected) number of elements it can process per unit time. It also has a known *selectivity* $0 < p_i < 1$, the probability that a randomly chosen element will pass that filter. We assume that the filters are independent of each other; in other words, the probability that an element will pass a filter does not depend on whether it passes any other filter.

Because our work was motivated by database applications, we use database terminology in presenting our results. We refer to the filters as *operators*, and the elements as *tuples*.

Figure 1 shows an instance of the problem, with four operators, $O_1, \cdots, O_4$. The edges indicate the precedence constraints. The goal of most pipelined filter ordering problems is to find a *single* permutation of the operators in which to apply the operators to the tuples [Ibaraki and Kameda 1984; Burge et al. 2005; Babu et al. 2004]. However, it is easy to see that, in the parallel setting, a single permutation would not maximize the throughput, and we must instead use a set of permutations simultaneously. This observation was originally made by Kodialam [2001] who considered the problem of determining the maximum throughput in a sequential testing system. In the example shown in Figure 1, a single permutation can achieve a maximum throughput of 900, whereas using three permutations simultaneously as shown, we can achieve a throughput of 1560 tuples per unit time. Our goal is to find the set of permutations and the associated probability weights, collectively called a *routing*, that maximizes the throughput. The probability weights must sum to 1. When a new tuple enters the system, it is assigned one of these permutations, chosen randomly according to the probability weights (called the *routing for that tuple*).

Kodialam gave an $O(n^3)$ algorithm for the parallel pipelined filter ordering problem which exploited the polymatroid structure of a certain space associated with the problem instance [Kodialam 2001]. (He also considered a queueing-theoretic version of the parallel pipelined filter ordering problem, in which r_i is the *maximum*, rather than the expected, number of tuples that O_i can process in unit time, and excess tuples are stored in queues; he showed that this version can be reduced to the version discussed here.) Subsequently, Condon et al. gave a conceptually simpler $O(n^2)$ algorithm for the same problem [Condon et al. 2009].

We extend the work of Condon et al. to give an $O(n^3)$ algorithm for solving the problem when there are tree-structured precedence constraints between the filters. The ability to handle such precedence constraints is particularly important in a database query optimization setting, because tree-structured precedence constraints commonly arise when executing multi-way join queries with acyclic join query graphs. Most of the prior algorithms for similar problems (including the seminal work by Ibaraki and Kameda [1984]) handle such constraints. We show that the parallel pipelined filter ordering problem is NP-hard for arbitrary precedence constraints.

Like the algorithm of Condon et al., our algorithm is combinatorial. It also has the

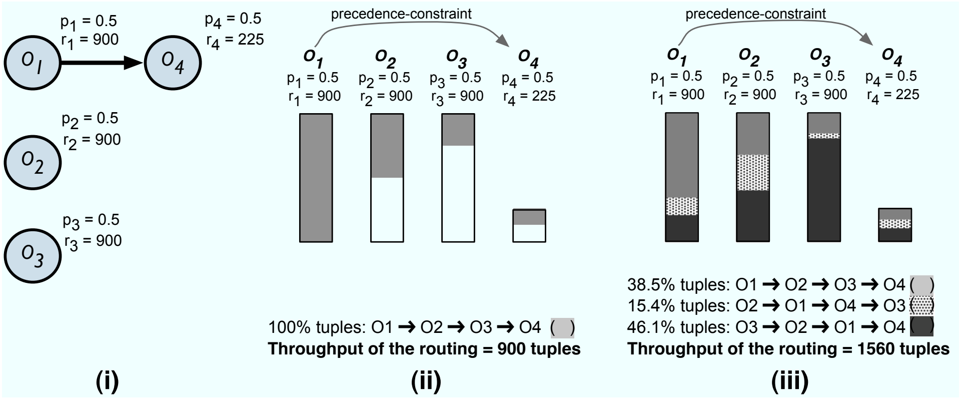

p1 = 0.5
r1 = 900
p4 = 0.5
r4 = 225
O1
O4
p2 = 0.5
r2 = 900
O2
p3 = 0.5
r3 = 900
O3
precedence-constraint
O1 O2 O3 O4
p1 = 0.5 r1 = 900
p2 = 0.5 r2 = 900
p3 = 0.5 r3 = 900
p4 = 0.5 r4 = 225
100% tuples: O1 → O2 → O3 → O4
Throughput of the routing = 900 tuples
38.5% tuples: O1 → O2 → O3 → O4
15.4% tuples: O2 → O1 → O4 → O3
46.1% tuples: O3 → O2 → O1 → O4
Throughput of the routing = 1560 tuples
(i)
(ii)
(iii)

Fig. 1. (i) An example problem instance with 4 operators; (ii) Using a single permutation results in a throughput of 900 tuples, and it does not utilize the processing power of operators O_2, O_3, and O_4 fully (shaded areas indicate the processing capacity currently utilized); (iii) The optimal solution uses 3 permutations to achieve a throughput of 1560 tuples. We revisit this example in more detail later.

property, highly desirable in practice, that it produces a *sparse* solution using $O(n)$ permutations. The solution can then be converted to use at most n permutations by using a standard procedure for converting an optimal solution to a linear program into a basic optimal solution.

We note that our algorithms, like those of Condon et al. and Kodialam, find solutions that achieve maximum throughput when the system is in a steady state, or equivalently, if the system is viewed as operating on an infinite stream of tuples. They do not seek to optimize throughput for a small set of tuples, or during startup or ending phases.

Motivated by join operators in database queries, we also consider the case when some of the operator selectivities are greater than 1 (i.e., we do not restrict $p_i < 1$). A database join between two relations R and S, requires us to find, for each tuple $r \in R$, all "matching" tuples from S, say $s_1, \cdots, s_k$, such that r and s_i satisfy the join condition for all i. Under certain simplifying assumptions [Krishnamurthy et al. 1986; Deshpande and Hellerstein 2008], such joins can be modeled as pipelined operators: when a tuple r passes this operator, it may return multiple tuples $r.s_1, \cdots, r.s_k$. We make the assumption that when a tuple r enters a non-selective operator and generates new tuples $r.s_1, \cdots, r.s_k$, all new tuples follow the same subsequent route; i.e., we consider only solutions of this type.[1] In practice, higher throughput may be attainable by allowing new tuples to follow different routes. We leave as an open question the appropriate way to model a system with a mix of selective and non-selective operators, and the related question of which types of routings and solution spaces should be considered.

[1] We can show that this assumption does not reduce attainable throughput in the context of a simple model in which the expected behavior of an operator is in fact its actual behavior (so that at an operator with selectivity p, each tuple entering the operator generates exactly p new tuples). We used just such a simple model in the case of selective operators, and as mentioned above, a queueing theoretic justification for the simple model was given in that case by Kodialam. Unfortunately, there are fundamental difficulties in producing a similar justification in the case of non-selective operators.

We call operators with selectivity ≥ 1 "non-selective" operators (if the selectivity of an operator is < 1, we call it a "selective" operator). We obtain polynomial-time algorithms for two cases: (1) when the precedence graph is restricted to be a forest of chains, and (2) when the precedence constraints are tree-structured, but all operators have selectivity greater than 1.

We also prove a strong connection between the problem of minimizing total work in the sequential pipelined filtering ordering problem (**A**), and the problem of maximizing throughput in parallel pipelined filter ordering (**B**). More precisely, using linear programming duality, we show that problem **A** is solvable in polynomial time for a given class of precedence constraints if and only if problem **B** is solvable in polynomial time for that class. (A related LP duality was previously exploited in a more general setting by Liu et al. in developing approximation algorithms for generic max-throughput problems [Liu et al. 2008].) This equivalence yields the aforementioned hardness result for arbitrary precedence constraints, since problem **A** is known to be NP-Hard with arbitrary precedence constraints [Ibaraki and Kameda 1984; Burge et al. 2005]. In addition, since Ibaraki and Kameda [1984] gave a polynomial-time algorithm for the classical problem with tree-structured precedence constraints, the equivalence yields an alternative to the main algorithm in this paper. The alternative algorithm uses the ellipsoid algorithm, and is both less efficient and more complicated than the main algorithm in this paper. Its one advantage is that it works for a combination of selective and non-selective operators.

2. RELATED WORK

The problem of choosing the order in which to apply a given set of selection predicates to the tuples of a relation is considered one of the central problems in database query optimization, and has been deeply studied in literature under a variety of cost models and scenarios. In this context, pipelined filter ordering is sometimes called *selection ordering*.

Ibaraki and Kameda [1984] and Krishnamurthy et al. [1986] analyzed what we call the *classical* (sequential) version of this problem, where the goal is to minimize the total expected cost of applying the predicates to a tuple (called *sum-cost* or *total work* metric). They presented polynomial-time optimal algorithms for this problem under the assumptions that the predicates are independent of each other, and that the precedence constraints between the selection predicates form a forest of rooted trees. The problem is known to be NP-Hard for arbitrary precedence constraints [Ibaraki and Kameda 1984; Burge et al. 2005], or in the presence of correlations [Babu et al. 2004].

The selection ordering problem has received renewed attention in the recent years in the context of environments like the web [Chaudhuri et al. 1995; Goldman and Widom 2000; Etzioni et al. 1996; Srivastava et al. 2006; Srivastava et al. 2005], continuous high-speed data streams [Avnur and Hellerstein 2000; Babu et al. 2004], and sensor networks [Deshpande et al. 2005]. These environments present significantly different challenges and cost structures than traditional centralized database systems. Selection ordering problems have also been studied in other areas such as fault detection and machine learning (see e.g., Shayman and Fernandez-Gaucherand [2001] and Kaplan et al. [2005]), under names such as learning with attribute costs [Kaplan et al. 2005], minimum-sum set cover [Feige et al. 2004], and satisficing search [Simon and Kadane 1975].

Our work is closely related to recent work by Srivastava et al. [2006], who study query optimization over Web services. Abstractly their problem is identical to the parallel pipelined filter ordering problem that we study here (each Web service can be thought of as a parallel filter). They consider a somewhat different solution space than ours in that they allow sending a tuple to multiple Web services in parallel. This can reduce the total number of tuples sent to the Web services if the Web services are non-selective. This does result in additional processing at the querying site, to combine the results returned by the Web services – that cost is inconsequential in their model. However, their focus is on finding a single routing over the operators to be used for all the tuples, in contrast to our approach of using different routings for different tuples simultaneously. For the case with only selective operators, our approach subsumes their approach; for selective operators, there is no advantage to being able to send a tuple to multiple filters in parallel. But, for a mix of selective and non-selective operators, neither their approach nor ours is necessarily superior to the other.

More recently, Liu et al. [2008] also considered the problem of conjunctive query evaluation in a parallel setting, and provided randomized flow algorithms that generate approximate solutions. However, they don't allow precedence constraints between the operators.

The algorithms we present are based on a characterization of query execution as tuple flows. We refer the reader to Condon et al. [2006; 2009] for a discussion of related work on flow algorithms.

Similar to some of the previous work on developing algorithms in the presence of precedence constraints among operators or tasks, we first develop an algorithm for the special case of *forest of chains* precedence constraints (Section 3.4); we then extend that algorithm to general tree-structured precedence constraints by *eliminating forks* in the tree bottom-up. We note that the basic idea of eliminating forks bottom-up to extend an algorithm that works for forests of chains to trees (more generally to *series-parallel graphs*) can be attributed to Lawler [1978], who used it to derive an efficient polynomial-time algorithm for job scheduling in the presence of precedence constraints. It was adapted directly to solve the classical selection ordering problem by Ibaraki and Kameda [1984], who observed that the Adjacent Sequence Interchange (ASI) property required by Lawler's algorithm was satisfied by the selection ordering problem. Our problem of finding the optimal routing in a parallel environment does not obey an equivalent property, and hence applying this basic idea to our problem is more involved, and requires careful analysis (Section 3.5).

3. MTTC ALGORITHM: SELECTIVE OPERATORS

In this section, we present our algorithm for solving the parallel pipelined filter-ordering problem, with tree-structured precedence constraints, when all operators are selective (i.e., $p_i < 1, \forall i$). The algorithm maximizes *tuple throughput*, i.e., the number of tuples that can be processed in unit time. We call this the *MTTC* problem (*max-throughput with tree-structured precedence constraints*).

We begin with a formal problem definition (Section 3.1) and some preliminary lemmas (Section 3.2). Next, we review the algorithm for the case with no precedence constraints from Condon et al. [2009] (Section 3.3). We then present our first key algorithm, for the special case when the precedence constraints are a *forest of chains* (Section 3.4). Then, we present an algorithm for the general case that recursively reduces arbitrary tree-structured constraints to forests of chains (Section 3.5). We conclude the section with a general technique for reducing the number of permutations used in the routing (Section 3.6).

3.1 Definition of the MTTC Problem

The input to the MTTC problem is a list of n operators, $O_1, \ldots, O_n$, associated selectivities $p_1, \ldots, p_n$ and rate limits $r_1, \ldots, r_n$, and a precedence graph G on the operators. Graph G is a forest of rooted trees. The p_i and r_i are rational values satisfying $0 < p_i < 1$ and $r_i > 0$.

The goal in the MTTC problem is to find an optimal tuple routing that maximizes throughput. The routing specifies, for each permutation of the operators, the number of tuples to be sent along that permutation per unit time[2]. A tuple sent along a permutation π travels through the operators in the order specified by π, until it is either eliminated by an operator or it has traveled through all the operators. A tuple is eliminated by operator O_i with probability $(1 - p_i)$, and the probability that a tuple is eliminated by one operator is independent of the probability that it is eliminated by other operators.

The routing must not exceed the rate limits of the operators, and must obey the precedence constraints defined by G.

Below we give a linear program formally defining the MTTC problem. We use the following notation. Let π be a permutation of the operators $\mathcal{O} = \{O_1, \ldots, O_n\}$. The k^{th} element of π is denoted by $\pi(k)$. The index of operator $\pi(k)$ is denoted by $\pi'(k)$, so $\pi(k) = O_{\pi'(k)}$. Let $\phi_G(n)$ be the subset of the $n!$ permutations of $\mathcal{O}$ that obey the precedence constraints defined by G. For $i \in \{1, \ldots, n\}$ and $\pi \in \phi_G(n)$, $g(\pi, i)$ denotes the probability that a tuple sent according to permutation π reaches operator O_i without being eliminated. Thus if $\pi(1) = O_i$, then $g(\pi, i) = 1$; otherwise, $g(\pi, i) = p_{\pi'(1)} p_{\pi'(2)} \ldots p_{\pi'(m-1)}$, where m is such that $\pi(m) = O_i$. Define real-valued variables f_π, one for each $\pi \in \phi_G(n)$, where each f_π represents the number of tuples routed along permutation π per unit time. We call the f_π *flow variables*. The number of flow variables is thus at most $n!$.

MTTC LP: Given $r_1, \ldots, r_n > 0$, $p_1 \ldots, p_n \in (0, 1)$, and a precedence graph G on $\{O_1, \ldots, O_n\}$ that is a forest of trees, find an assignment to the variables f_π, for all $\pi \in \phi_G(n)$, maximizing

$$F = \sum_{\pi \in \phi_G(n)} f_\pi$$

subject to the constraints:

(1) $\sum_{\pi \in \phi_G(n)} f_\pi g(\pi, i) \leq r_i$ for all $i \in \{1, \ldots, n\}$ and

(2) $f_\pi \geq 0$ for all $\pi \in \phi_G(n)$.

We refer to the first set of constraints involving the r_i as *rate constraints* (since they guarantee that the rate limits are not exceeded). If assignment K to the f_π satisfies the rate constraint for r_i with equality, we say that O_i is *saturated* by K. If K saturates all the operators, we say that K is a *saturating* routing. The value F achieved by K is the *throughput* of K, and we call K a *routing*.

Given two operators O_i and O_j, we say that:

[2] These values are normalized using the total throughput at the end, to obtain probabilities to be used for actual routing during execution.

— O_i *can saturate* O_j if $r_i p_i \geq r_j$.
— O_i *can just saturate* O_j if $r_i p_i = r_j$.
— O_i *can overfill* O_j if $r_i p_i > r_j$.

A *chain* is a tree in which each internal node has exactly one child. A chain in the precedence graph of an MTTC instance is *proper* if each internal node in the chain can saturate its child.

3.2 Preliminary Lemmas

In this section, we present four main lemmas on which our algorithms are based. We first present a lemma proved earlier by Condon et al. [2006; 2009]. Building on that earlier lemma, we then present three new lemmas.

Let K be a feasible solution to an MTTC instance, and let $\mathcal{O} = \{O_1, \ldots, O_n\}$. We say $Q \subseteq \mathcal{O}$ is a *saturated suffix* of K if Q is non-empty and (1) the operators in Q are saturated by K and (2) if $f_\pi > 0$ in K, then the elements of $\mathcal{O} - Q$ precede the elements of Q in π (i.e., no tuples flow from an operator in Q to an operator in $\mathcal{O} - Q$).

LEMMA 3.1. *(Condon et al. [2006; 2009]) (The saturated suffix lemma) If feasible solution K to the MTTC LP has a saturated suffix Q, then K is an optimal solution and achieves throughput*

$$F^* = \frac{\sum_{O_i \in Q} r_i(1-p_i)}{(\prod_{O_j \in \mathcal{O}-Q} p_j)(1 - \prod_{O_i \in Q} p_i)}$$

Proof. See Condon et al. [2009]. □

Intuitively, the above lemma says the following: If in a solution, there is a set of operators that are all saturated *and* there is no flow from an operator in this set to an operator not in the set, then the solution is optimal.

Condon et al. actually proved the above lemma assuming that there were no precedence constraints. Since adding precedence constraints to an MTTC instance can only reduce the maximum throughput attainable, the lemma also holds with constraints.

The next lemma concerns solutions in which all operators are saturated.

LEMMA 3.2. *Let I be an instance of the MTTC problem without precedence constraints. Let $Z = \frac{\sum_{i=1}^n r_i(1-p_i)}{(1-\prod_{i=1}^n p_i)}$. Let F^* be the optimal value of the objective function for instance I. If $F^* = Z$, then every optimal routing for I saturates all the operators. Further, if there exists a routing that saturates all the operators, then that routing is optimal, and achieves throughput Z.*

Proof. For any operator O_i, since operator O_i has a rate limit of r_i, O_i can process at most r_i flow units per unit time. Since O_i has selectivity p_i, it discards at most $r_i(1-p_i)$ amount of flow per unit time. Thus under any feasible routing, the total amount of flow discarded by all the processors per unit time is at most $Z = \sum_{i=1}^n r_i(1-p_i)$; this is the precise amount if all processors are saturated, and is a strict upper bound otherwise.

Consider a routing achieving throughput F^*. Of the F^* amount of flow that is sent into the system in this routing, the amount that successfully travels through all the processors (i.e., is not discarded by any) is $F^* \prod_{i=1}^n p_i$. Thus the total amount that is discarded by some processor, per unit time, is $F^*(1 - \prod_{i=1}^n p_i)$. By the above, $F^*(1 -$

$\prod_{i=1}^{n} p_i) \leq \sum_{i=1}^{n} r_i(1-p_i)$, with equality iff the processors are all saturated. Since $Z = \frac{\sum_{i=1^n} r_i(1-p_i)}{(1-\prod_{i=1}^{n} p_i)}$, $F^* = Z$ implies that all operators are saturated.

Finally, given a routing that saturates all the operators, by the above arguments the routing achieves throughput Z. □

Next, we prove the reverse of the saturated suffix lemma: in the case of no precedence constraints, every optimal solution has a saturated suffix.

LEMMA 3.3. *Let I be an instance of the MTTC problem without precedence constraints, and let K be an optimal routing for I. Then K has a saturated suffix.*

Proof. If K saturates all operators, then it trivially satisfies the condition. Let K be such that it does not saturate all operators. Next we give a constructive algorithm for finding a saturated suffix of K, thus proving the lemma.

Let $\mathcal{O}_S$ denote the set of operators that are saturated by K. Since K is optimal, $\mathcal{O}_S$ is non-empty (otherwise we can increase f_π for some permutation π to obtain a better solution). Further, $\mathcal{O}_S \subset \mathcal{O}$. Check if $\mathcal{O}_S$ is a saturated suffix of K, i.e., for any permutation π that is assigned a positive flow in K, check if $\mathcal{O}_S$ forms a suffix of π. If yes, we are done.

Otherwise, let π be a permutation, with $f_\pi > 0$, that does not have $\mathcal{O}_S$ as a suffix. Thus there exist operators O_i, O_j, such that $O_i \in \mathcal{O} - \mathcal{O}_S$ (O_i is unsaturated), $O_j \in \mathcal{O}_S$ (O_j is saturated), and O_j appears immediately before O_i in π. Let π' denote the permutation obtained by swapping O_i and O_j in π.

We now modify K by decreasing f_π by ϵ and increasing $f_{\pi'}$ by ϵ, where $0 < \epsilon < f_\pi$, and ϵ is strictly smaller than the remaining capacity of O_i. This does not change the total throughput, and further does not change the number of tuples routed through any operator other than O_i and O_j. At the same time, O_i remains unsaturated, but O_j now becomes unsaturated (since the flow through it decreases by a small amount). We adjust $\mathcal{O}_S$ accordingly by removing O_j.

We repeat this process until there is no such permutation π. Let K' denote the resulting routing. Hence K' has the saturated suffix property (if K' does not have any saturated operators, then we can increase its throughput, thus contradicting the optimality of K).

Let $\mathcal{O}'_S$ denote a saturated suffix of K'. In other words, the operators in $\mathcal{O}'_S$ are all saturated, and for any permutation π that has been assigned a positive flow in K', the operators in $\mathcal{O}'_S$ form a suffix of π.

Now, note that the set of permutations that are assigned positive flow in K' is a superset of the set of permutations that are assigned positive flow in K. Further, $\mathcal{O}'_S \subseteq \mathcal{O}_S$. Hence, $\mathcal{O}'_S$ must be a saturated suffix of K as well. □

The next and final lemma in this section is a stronger version of the saturated suffix lemma.

LEMMA 3.4. *Let I be an instance of the MTTC problem with no precedence constraints, and let the operators in I be numbered so that $r_1 \geq r_2 \ldots \geq r_n$. Let*

$$F^* = min_{k \in \{1,\ldots,n\}} \frac{\sum_{i=k}^{n} r_i(1-p_i)}{(\prod_{j=1}^{k-1} p_j)(1-\prod_{i=k}^{n} p_i)}.$$

Then F^* is the optimal value of the objective function for instance I. If k' is the largest *value of k achieving the value F^*, then there exists an optimal routing for which $\{O_{k'}, O_{k'+1}, \dots, O_n\}$ is a saturated suffix.*

Proof. For $a, b \in \{1, \dots, n\}$, let $S_{a,b} = \sum_{i=a}^{b} r_i(1-p_i)$, $P_{a,b} = \prod_{i=a}^{b} p_i$, and $h_{a,b} = \frac{S_{a,b}}{(1-P_{a,b})}$. Thus $F^* = min_{k\in\{1,\dots,n\}} \frac{1}{P_{1,k-1}} h_{k,n}$.

The algorithm in Condon et al. [2006; 2009], when run on input I, outputs an optimal routing K with a saturated suffix. The saturated suffix is $\{O_{k^*}, \dots, O_n\}$ for some $1 \leq k^* \leq n$, and the throughput achieved by K is $\frac{1}{P_{1,k^*-1}} h_{k^*,n}$. That previous work also shows that the value of the objective function for I is $F^* = min_{k\in\{1,\dots,n\}} \frac{1}{P_{1,k-1}} h_{k,n}$, and thus $F^* = \frac{1}{P_{1,k^*-1}} h_{k^*,n}$.

It remains to prove that I also has a routing achieving throughput $\frac{1}{P_{1,k'-1}} h_{k',n}$. If $k' = k^*$, this is trivially true.

Suppose $k' \neq k^*$. Then $k' > k^*$. Routing K sends F^* amount of flow into the system, with all flow traveling first through the operators in $Q_{pref} = \{O_1, \dots, O_{k^*-1}\}$ and then through the operators in $Q_{suff} = \{O_{k^*}, O_{k^*+1}, \dots, O_n\}$. Of the F^* flow sent into the system, $P_{1,k^*-1}F^*$ of it reaches Q_{suff}, and it saturates those operators. Let I_1 be the induced MTTC instance created by keeping only the operators in Q_{suff} (i.e., by discarding the other operators). It follows from the above that there is a saturating routing for I_1 that achieves throughput $P_{1,k^*-1}F^*$. By Lemma 3.2, $P_{1,k^*-1}F^* = h_{k^*,n}$.

We show that, in fact, there exists a saturating routing $\hat{K}$ for I_1 whose saturated suffix is $\{O_{k'}, O_{k'+1}, \dots, O_n\}$. Routing $\hat{K}$ can be used to construct the following routing for the original MTTC instance on $O_1, \dots, O_n$: Send F^* flow through Q_{pref} as specified by K, and then send the $P_{1,k^*-1}F^*$ surviving (i.e., not eliminated) flow through Q_{suff} as specified by $\hat{K}$. Since $\{O_{k'}, \dots, O_n\}$ is a saturated suffix of this routing, this proves the lemma.

It remains to show that $\hat{K}$ exists.

Claim: Let $a, k, b \in \{1, \dots, n\}$ where $a \leq k \leq b$. Then $\frac{1}{P_{a,k-1}} h_{k,b} < h_{a,b}$ iff $\frac{1}{P_{a,k-1}} h_{k,b} < h_{a,k-1}$, and $\frac{1}{P_{a,k-1}} h_{k,b} = h_{a,b}$ iff $\frac{1}{P_{a,k-1}} h_{k,b} = h_{a,k-1}$.

The claim is easily shown to be true by algebraic manipulation, the definitions of P and h, and the fact that the selectivities p_i are strictly between 0 and 1.

Let I_3 be the instance induced from I_1 by keeping only the operators in $\{O_{k'}, \dots, O_n\}$. By the definition of k', for all j such that $k' < j \leq n$, $\frac{1}{P_{1,k'-1}} h_{k',n} < \frac{1}{P_{1,j-1}} h_{j,n}$, and hence $h_{k',n} < \frac{1}{P_{k',j-1}} h_{j,n}$. Therefore, by the result stated at the start of this proof, there is an optimal routing for I_3 for which $\{O_{k'}, \dots, O_n\}$ is a saturated suffix, and hence this routing is saturating for I_3 and achieves a throughput of $h_{k',n}$. Let K_3 denote this routing.

Let I_2 be the MTTC instance induced from I_1 by keeping only the operators in $\{O_{k^*}, \dots, O_{k'-1}\}$. Note that by the definitions of k' and k^*, $\frac{1}{P_{1,k^*-1}} h_{k^*,n} = \frac{1}{P_{1,k'-1}} h_{k',n}$, and multiplying both sides by P_{1,k^*-1} we get that

$$h_{k^*,n} = \frac{1}{P_{k^*,k'-1}} h_{k',n} \tag{1}$$

Applying the claim to Equation 1, we get that

$$h_{k^*,k'-1} = \frac{1}{P_{k^*,k'-1}} h_{k',n} \tag{2}$$

We now show that there is a routing achieving throughput $h_{k^*,k'-1}$ on I_2. Assume not. Then by the properties of F^* given at the start of this proof, there exists i such that $k^* < i \leq k'$ and

$$\frac{1}{P_{k^*,i-1}} h_{i,k'-1} < h_{k^*,k'-1} \tag{3}$$

Combining Equation 2 and Inequality 3, we get

$$\frac{1}{P_{k^*,i-1}} h_{i,k'-1} < \frac{1}{P_{k^*,k'-1}} h_{k',n} \tag{4}$$

Multiplying both sides by $P_{k^*,i-1}$, we get that

$$h_{i,k'-1} < \frac{1}{P_{i,k'-1}} h_{k',n} \tag{5}$$

Applying the claim to Inequality 5 yields

$$h_{i,n} < \frac{1}{P_{i,k'-1}} h_{k',n} \tag{6}$$

Multiplying both sides of the above equation by $\frac{1}{P_{1,i-1}}$, we get that

$$\frac{1}{P_{1,i-1}} h_{i,n} < \frac{1}{P_{1,k'-1}} h_{k',n} \tag{7}$$

But this contradicts the definition of k' in the statement of the lemma, since in the definition for F^*, setting k' to k must minimize the given expression, and thus setting k to i cannot achieve a smaller value for it. So $h_{k^*,k'-1}$ is the value of the maximum throughput for I_2, and there is a routing K_2 achieving this throughput on I_2.

We now construct $\hat{K}$ from K_2 and K_3. It must send a total of $\frac{1}{P_{k^*,k'-1}} h_{k',n}$ flow through the operators of I_1.

By Equation 2, $h_{k^*,k'-1} = \frac{1}{P_{k^*,k'-1}} h_{k',n}$, so $\hat{K}$ first uses K_2 to route $\frac{1}{P_{k^*,k'-1}} h_{k',n}$ flow through the operators in I_2. After this flow has passed through the operators in I_2, the amount of flow that has not been eliminated is $P_{k^*,k'-1}(\frac{1}{P_{k^*,k'-1}} h_{k',n}) = h_{k',n}$. Since K_3 achieves throughput of $h_{k',n}$, $\hat{K}$ routes this remaining flow through the operators in I_3 using K_3, and thus $\{O_{k'}, O_{k'+1}, \ldots, O_n\}$ is a saturated suffix of $\hat{K}$ as desired. □

The above lemmas are the basis for our algorithms and their correctness proofs. The idea behind our algorithm is to construct a routing with a saturated suffix. This may be impossible due to precedence constraints; as a simple example, consider a two-operator instance where O_1 must precede O_2, but O_1 cannot saturate O_2. However, by reducing rate limits of certain operators, we can construct a routing with a saturated suffix that is also optimal with respect to the original rate limits.

3.3 The MTTC Problem with No Precedence Constraints

Our algorithm for the MTTC problem builds on the algorithm of Condon et al. [Condon et al. 2009] which solves the MTTC problem with no precedence constraints. We begin by reviewing that algorithm and its correctness proof. It builds a routing K incrementally. The routing consists of pairs (π, x) indicating that x amount of flow is to be sent along permutation π.

Let the operators be numbered such that $r_1 \geq \ldots \geq r_n$. Let π be the permutation $(O_1, \ldots, O_n)$. Since all selectivities are < 1, π obeys the property that each operator in the ordering can't overfill its predecessor; this "can't overfill" property is an invariant of the algorithm with respect to the *residual* rate limits, i.e., the remaining available capacities of the operators given the current flow through them.

We execute the following recursive procedure. Conceptually, we start by sending flow through the operators according to ordering π, beginning with no flow, and increasing the amount at a continuous rate. As flow is increased, the remaining available capacities (i.e., the residual rate limits) of the operators decrease. Suppose that before any operator is saturated, an operator O_i suddenly becomes able to just saturate its predecessor O_j in π (i.e., $r'_i p_i = r'_j$, where r'_i and r'_j are the residual rate limits of O_i and O_j). We stop increasing the flow at this point. Let x be the resulting flow value. (We actually calculate x analytically.) No additional flow can be added along π without violating the "can't overfill" invariant with respect to the residual limits. We place (π, x) into (initially empty) output routing K. We then modify π by swapping O_i and O_j. In addition, we "paste" O_i to the front of O_j, forming a "superoperator" (O_i, O_j). *Throughout the paper, we use the word "superoperator" to refer to an operator that was formed by combining one or more other operators.*

All subsequent flow sent into (O_i, O_j) will be sent first to O_i and then immediately to O_j. (Because O_i can just saturate O_j, this means that ultimately either both operators will be saturated, or neither.)

We treat superoperator (O_i, O_j) as a new single operator. Its rate limit is defined to be the residual rate limit of O_i and its selectivity is defined to be $p_i p_j$. We now have a new ordering π on a set of $n - 1$ operators. We set the rate limits of the remaining operators to equal their residual rate limits. We recurse on the $n - 1$ operators and the modified π. In each recursive call, we add another (π, x) to routing K.

The recursion stops when, during some call, as we send increasing flow along π, some operator O_k becomes fully saturated before (or at the same time as) any operator O_i becomes able to just saturate its predecessor O_j. In this case x becomes the amount of flow causing the saturation, (π, x) is added to K, and we terminate.

The optimality of routing K is shown as follows. Since the algorithm increases flow along a permutation at a continuous rate, no operator in a permutation can attain the ability to overfill its predecessor without first becoming able to just saturate its predecessor, thus triggering a halt to the flow increase. Thus when the flow increase is stopped, the permutation satisfies the "can't overfill" property. Swapping O_i and O_j preserves the property. Thus the property holds at the end of the final call. It follows that because O_k is saturated, so are all (super)operators following O_k. Thus K has a saturated suffix and hence is optimal.

We illustrate the execution of the algorithm with the following example (Figure 2). Consider an instance with two operators O_1 and O_2, where $r_1 = 3$, $r_2 = 2$, and $p_1 = p_2 = 1/2$.

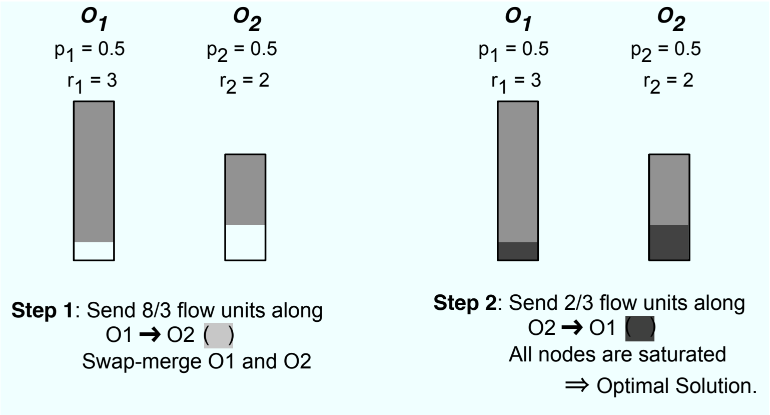


Fig. 2. Illustration of the algorithm for the case with no precedence constraints

Let $\pi = (O_1, O_2)$. After sending $x = 8/3$ flow along ordering π, the residual capacity of O_1 is $3 - 8/3 = 1/3$ and the residual capacity of O_2 is $2 - 1/2 * 8/3 = 2/3$, so O_2 can now just saturate O_1. We place $(\pi, 8/3)$ into our routing. We swap O_1 and O_2 in π and form a superoperator (O_2, O_1) with rate limit 2/3 and selectivity $1/2 * 1/2 = 1/4$. Treating it as a single operator in π, a trivial repetition of the above procedure (on one operator) finds that sending $2/3$ flow units results in saturation of (super)operator (O_2, O_1). These units are sent along permutation $\pi' = (O_2, O_1)$, so $(\pi', 2/3)$ is added to the routing. The result is an optimal routing saturating both operators, whose total throughput is $8/3 + 2/3 = 10/3$.

3.4 The MTTC Algorithm for Chains

We now present the MTTC algorithm for the special case in which precedence graph G is a forest of chains. It is a generalization of the algorithm just described. That algorithm does not work here because *precedence constraints may be violated* when:

(1) initially ordering the operators in decreasing order of rate limits, or when

(2) swapping the order of some O_i and O_j in π.

We handle the first problem via a preprocessing procedure which adjusts the rate limits of the operators so that if O_i precedes O_j in the precedence graph, then the rate limit of O_i is always higher than rate limit of O_j.

To avoid the second problem we add additional steps to the above algorithm, yielding a new procedure that we call RouteChains. Details are below.

3.4.1 *Preprocessing procedure.* In the preprocessing procedure, we first make each chain of G proper, as follows. For each non-leaf operator O_i in the chain, beginning from the top of the chain and proceeding downward, we execute the following step: *Let O_j be the child of O_i. If O_i can't saturate O_j, then reset the rate limit r_j of O_j to be $r_i p_i$.*

Although the above procedure reduces the rate limits of some operators, it does not reduce the maximum throughput attainable. Because of the precedence constraints, all flow into an operator O_j must first pass through its parent O_i, so at most $r_i p_i$ flow can ever reach O_j.

We finish preprocessing by generating a sorted list of the operators, in descending order of their rate limits.

3.4.2 *The RouteChains procedure.* RouteChains is a recursive procedure that incrementally constructs a routing K, consisting of pairs of the form (π, x), indicating that x amount of flow is to be sent along permutation π.

Define an *l*-superoperator (linear superoperator) to be a permutation π' of a subset of the operators, such that each operator in π' (but the last) can just saturate its successor in π'. The selectivity of π', denoted $\sigma(\pi')$, is the product of the selectivities of its component operators, and its rate limit, denoted $\rho(\pi')$, is the rate limit of its first operator.

The inputs to RouteChains are as follows.

RouteChains: Inputs

1. Rate limits $r_1, \ldots, r_n$ (all > 0), and selectivities $p_1, \ldots, p_n$ (all strictly between 0 and 1), for a set of operators $\mathcal{O} = \{O_1, ..., O_n\}$,
2. A precedence graph G with vertex set $\mathcal{O}$ consisting of proper chains,
3. A permutation π of $\mathcal{O}$ obeying the constraints of G,
4. An ordered partition $P = (\pi_1, \ldots, \pi_m)$ of π into subpermutations π_i, where each π_i is an *l*-superoperator. The *l*-superoperators of P must satisfy the following "can't overfill" condition: for $2 \leq j \leq m$, $\rho(\pi_j)\sigma(\pi_j) \leq \rho(\pi_{j-1})$. That is, each *l*-superoperator in P cannot overfill its predecessor.

RouteChains: Output

RouteChains returns a routing K that is optimal for the MTTC instance I defined by the input rate limits, selectivities, and precedence constraints.

The inputs to RouteChains must satisfy the above properties in order to be valid. In particular, note that precedence graph G must consist of *proper* chains.

For the initial call to RouteChains, following preprocessing, we use the given rate limits, selectivities, operators, and graph G. We set permutation π to be the list of operators sorted in descending order of their rate limits. We set partition P to be the trivial partition where each *l*-superoperator π_i consists of the single operator O_i. Because the chains of G are proper, π obeys the precedence constraints. Because π orders the operators in decreasing order of their rate limits, and the operators are selective, the "can't overfill" condition is satisfied. Thus the initial inputs to RouteChains are valid.

RouteChains: Execution

RouteChains first calculates the minimum $x \geq 0$ such that sending x flow units along permutation π triggers one of the following stopping conditions:

Stopping Condition 1: Some operator is saturated.

Stopping Condition 2: Some *l*-superoperator π_i, $2 \leq i \leq m$, can just saturate its predecessor π_{i-1}.

Stopping Condition 3: Some operator O_i can just saturate O_j, where O_j is the child of O_i in a chain of G, and O_i and O_j are contained in distinct *l*-superoperators of P.

The value of x is calculated based on the following observations. Suppose $\pi = (O_1, \ldots, O_n)$. For any operator O_j, if y flow units are sent along π, then $y \prod_{k=1}^{j-1} p_k$ units will reach operator O_j. The residual rate limit of O_j will then be $r_j - y\prod_{k=1}^{j-1} p_k$. Thus saturation of O_j occurs when $y = \frac{r_j}{\prod_{k=1}^{j-1} p_k}$. Similarly, it can be shown that for $2 \leq j \leq m$, *l*-superoperator π_j becomes able to just saturate π_{j-1} when $y = \frac{\rho(\pi_j)\sigma(\pi_j)-\rho(\pi_{j-1})}{\prod_{k=1}^{j} \sigma(\pi_k)-\prod_{k=1}^{j-2} \sigma(\pi_k)}$. Finally, for $1 \leq i < j \leq n$, O_i becomes able to just saturate O_j when $y = \frac{r_i p_i - r_j}{\prod_{k=1}^{i} p_k - \prod_{k=1}^{j-1} p_k}$. Thus x can be calculated by taking the minimum of $O(n)$ values.

After RouteChains computes x, what it does next is determined by the lowest-numbered stopping condition that was triggered by sending x flow along permutation π.

If Stopping Condition 1 was triggered, then RouteChains returns $K = \{(\pi, x)\}$. In other words, we have a saturated suffix and we are done.

Else, if Stopping Condition 2 was triggered for some π_i, then RouteChains chooses one such π_i. It swaps π_i and π_{i-1} in $(\pi_1, \ldots, \pi_m)$ and concatenates them into a single *l*-superoperator, yielding a new partition into *l*-superoperators:

$$P' = (\pi_1, \ldots, \pi_{i-2}, \pi_i\pi_{i-1}, \pi_{i+1}, \ldots, \pi_m)$$

and a new permutation:

$$\pi' = (\pi_1\pi_2 \ldots \pi_{i-2}\pi_i\pi_{i-1}\pi_{i+1} \ldots, \pi_m)$$

We call this operation a *swap-merge*. RouteChains then calls itself recursively, setting P to P', π to π', the r_i's to the residual rate limits, and keeping all other input parameters the same. The recursive call returns a set K' of flow assignments. RouteChains returns the union of K' and $\{(\pi, x)\}$.

Else if Stopping Condition 3 was triggered by a parent-child pair O_i, O_j, then RouteChains chooses such a pair and *absorbs O_j into O_i* as follows.

If O_i and O_j are contained in *l*-superoperators (O_i) and (O_j), each containing no other operators, RouteChains deletes the *l*-superoperator (O_j), and adds O_j to the end of (O_i), to form *l*-superoperator (O_i, O_j). Otherwise, let w, z be such that O_i is in π_w and O_j is in π_z. Let a, b be such that $\pi_w(a) = O_i$ and $\pi_z(b) = O_j$. RouteChains splits π_w into two parts, $A = (\pi_w(1), \ldots, \pi_w(a))$ and $B = (\pi_w(a+1), \ldots, \pi_w(s))$ where $s = |\pi_w|$. It splits π_z into three parts, $C = (\pi_z(1), \ldots, \pi_w(b-1))$, $D = (\pi_z(b), \ldots, \pi_w(c-1))$, and $E = (\pi_z(c), \ldots, \pi_w(t))$, where $t = |\pi_z|$ and c is the minimum value in $\{b+1, \ldots, t\}$ such that $\pi_z(c)$ is not a member of the same precedence chain as O_j; if no such c exists, it sets c to be equal to $t+1$ and E to be empty. RouteChains adds D to the end of A, forming four *l*-superoperators AD, B, C, E out of π_w and π_z. It then forms a new partition P' from P by replacing π_w in P by AD, B, in that order, and π_z by C, E in that order. If any elements of P' are empty, RouteChains removes them. Let π' denote the concatenation of the *l*-superoperators in P'.

Partition P' may not satisfy the "can't overfill" precondition with respect to the residual rate limits. (For example, it may be violated by *l*-superoperator AD and its predecessor.) In this case, RouteChains performs a modified topological sort on P'. It forms a directed graph G' whose vertices are the *l*-superoperators in P', with a directed edge from one *l*-superoperator to a second if there is an operator O_i in the first *l*-superoperator, and an operator O_j in the second, such that O_j is a child of O_i in a chain of G. Since π' obeys the precedence constraints, G' is a directed acyclic graph. RouteChains sorts the *l*-superoperators in P' by executing the following step until G' is empty: *Let S be the*

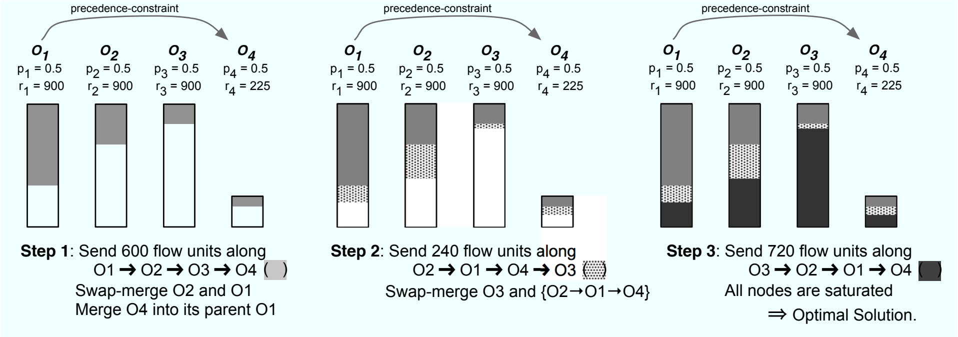


Fig. 3. Illustration of the algorithm for the problem instance shown in Figure 1

set of vertices (l-superoperators) in G' with no incoming edges. Choose the element of S with highest residual rate limit, output it, and delete it and its outgoing edges from G'. RouteChains resets P' to be the *l*-superoperators listed in the order output by the sort, and π' to be the concatenation of those *l*-superoperators.

RouteChains then executes a recursive call, using the initial set of operators, precedence constraints, and selectivities, and setting $\pi = \pi'$, $P = P'$ and the rate limits of the operators to be equal to their residual rate limits. The recursive call returns a set K' of flow assignments. RouteChains returns the union of K' and $\{(\pi, x)\}$.

3.4.3 *Example.* We illustrate our algorithm using the 4-operator instance shown in Figure 1, which has three chains and one precedence constraint, between O_1 and O_4. We assume the rate limits of 900 for operators O_1, O_2, and O_3, and a rate limit of 225 for O_4. The selectivity of each operator is set to be 0.5.

- We arbitrarily break the ties, and pick the permutation $O_1 \rightarrow O_2 \rightarrow O_3 \rightarrow O_4$ to start adding flow.
- When 600 units of flow have been added, O_2 exactly saturates O_1 (**stop. cond. 2**). We swap-merge O_2 and O_1 creating *l*-superoperator O_{21}.
- At the same time (after adding 0 units of flow), we find O_1 exactly saturates its child O_4 (**stop. cond. 3**). We absorb O_4 into its parent, creating superoperator O_{214}. There is no need to re-sort.
- After sending 240 units along $O_2 \rightarrow O_1 \rightarrow O_4 \rightarrow O_3$, we find that O_3 saturates O_{214} (**stop. cond. 2**). We swap-merge them to get a single super operator O_{3214}.
- We send 720 units along $O_3 \rightarrow O_2 \rightarrow O_1 \rightarrow O_4$, at which point all operators are saturated, and we achieve optimality (**stop. cond. 1**).

The throughput achieved is 1560; the best sequential plan (single permutation) can only process 900 tuples per unit time.

3.4.4 *Analysis of the chains algorithm.* The correctness of the MTTC algorithm for chains relies on the correctness of RouteChains. We begin with the following lemma.

LEMMA 3.5. *If the initial inputs to RouteChains are valid, then the following invariants hold on all recursive calls:*

— (a) The chains of G are proper;

— (b) Each element π_i of P is an l*-superoperator that cannot overfill its predecessor π_{i-1} and*

— (c) Permutation π obeys the precedence constraints.

Proof. The invariants clearly hold at the start of the initial call. We show that if they hold at the start of a call, they will also hold at the start of the next call.

The condition that triggers a parent-child absorption ensures that Invariant (a) is preserved. Since the residual capacity of an operator decreases continuously as flow into it increases continuously, if an increasing amount of flow is added along a permutation π, one operator cannot lose the ability to saturate another before first being able to just saturate it. Once a child is absorbed in a parent, the parent will be able to just saturate the child in all future permutations.

The condition that triggers a swap-merge ensures that Invariant (b) is not violated when adding new flow to the routing along permutation π. If an increasing amount of flow is added along permutation π, one *l*-superoperator cannot attain the ability to overfill another without first becoming able to just saturate it, triggering a swap-merge. The swap-merge creates a new *l*-superoperator from π_{i-1} and π_i, and it is easy to verify from the definitions of its rate limit and selectivity that the new partition is made up of *l*-superoperators and still satisfies Invariant (b).

The only other way that Invariant (b) could be violated would be as a result of a parent-child absorption. The partition produced by a parent-child absorption clearly consists of *l*-superoperators. It remains to show that the partition resulting from a modified topological sort has the property that no *l*-superoperator in the partition can overfill its predecessor. The partition produced by the sort has the property that for each pair of adjacent *l*-superoperators π_{i-1} and π_i, either $\rho(\pi_{i-1}) \geq \rho(\pi_i)$, or there is an edge in the precedence graph from an operator in π_{i-1} to an operator in π_i. In the former case, clearly π_i cannot overfill π_{i-1}. In the latter, since the chains of G are proper and the operators are selective, the precedence constraint is from an operator O_j in π_{i-1} with a higher rate limit to an operator O_k in π_i with strictly lower rate limit. By the properties of *l*-superoperators, and because operators are selective, the rate limit of O_k is an upper bound on $\rho(\pi_i)\sigma(\pi_i)$ (the amount of flow that can be output by *l*-superoperator π_i), and the rate limit of O_j is a lower bound on $\rho(\pi_{i-1})$. Thus π_i cannot overfill π_{i-1}.

We now show (c). The modified topological sort executed during a parent-child absorption respects the constraints. We show that the swap-merges do as well. Suppose the inputs to a recursive call to RouteChains satisfy (a), (b), and (c), and a swap-merge is performed on π_{i-1} and π_i following the addition of x flow along permutation π. If this causes a violation of the constraints, then there exist operators O_{k^*} and O_{l^*}, such that O_{k^*} is the parent of O_{l^*} in a chain of G, O_{k^*} was in π_{i-1} and O_{l^*} was in π_i. After the swap-merge, each operator in the merged *l*-superoperator $\pi_i\pi_{i-1}$ can exactly saturate its successor within that *l*-superoperator (if any). Since selectivities are less than one, after the swap-merge the residual capacity of O_{l^*} is greater than the residual capacity of O_{k^*}, and O_{k^*} cannot saturate O_{l^*}. But since all chains were proper at the start of the call, O_{k^*} could saturate O_{l^*} before the addition. Thus for some $0 \leq x^* < x$, adding x^* flow along permutation π would have made O_{k^*} able to just saturate O_{l^*}, contradicting the minimality of x. □

Each instance L of RouteChains has an associated MTTC instance consisting of the operators $\mathcal{O}$, rate limits r_i, selectivities p_i, and precedence graph G (but not π or the partition P).

The following lemma proves optimality of the routing output by RouteChains. Recall that in order for an input to RouteChains to be valid, the precedence graph must consist of a forest of proper chains.

LEMMA 3.6. *If the initial inputs to RouteChains are valid, then the routing K output by RouteChains is optimal for the associated MTTC instance. It is still optimal even if all precedence constraints are removed from the MTTC instance.*

Proof. The routing K output by RouteChains is easily seen to satisfy the rate limits. By the previous lemma, it satisfies the precedence constraints. To prove that it achieves maximum throughput, we do induction on the number of parent-child absorptions.

Base case: No parent-child absorptions.

We show that in this case the output routing K has a saturated suffix. By the saturated suffix lemma, it immediately follows that K is optimal for the associated MTTC instance, even if all precedence constraints are removed.

Assume for contradiction that K does not have a saturated suffix. Then there are operators O_i and O_j such that O_i appears before O_j in some permutation used in K, and O_i is saturated by K, but O_j is not. If O_j precedes O_i in some other permutation used in K, then there must have been a swap-merge of the *l*-superoperators containing O_i and O_j, since we have assumed no parent-child absorptions. But a swap-merge would put O_i and O_j in the same *l*-superoperator, and with no parent-child absorptions, O_i and O_j would be in the same *l*-superoperator in the final recursive call and hence would both be saturated by K, or both not. Therefore O_i precedes O_j in every permutation used by K, including the last, when O_i becomes saturated. But by Lemma 3.5, since no superoperator can overfill its predecessor, it is impossible for O_i to become saturated without first triggering Stopping Condition 2, or causing O_j to become saturated as well, which is a contradiction.

Induction step: Assume that output routing K is optimal, even if precedence constraints are removed, if the number of parent-child absorptions is i, where $i \geq 0$.

For RouteChains instance L, let $I(L)$ denote the MTTC instance associated with L, i.e., the instance defined by the input rate limits, selectivities, and precedence constraints of L.

Suppose RouteChains performs exactly $i+1$ parent-child absorptions on a RouteChains instance L.

We would now like to prove, as in the base case, that RouteChains constructs a routing with a saturated suffix. However, it is not clear how to do this directly, because the modified topological sort may change the ordering of the operators in ways that are difficult to analyze. So, we take a different approach.

Let q be such that the first parent-child absorption (when running RouteChains on L) occurs in the qth recursive call. Let L_{q+1} denote the input instance for the $q+1$st recursive call of RouteChains. The only difference between $I(L)$ and $I(L_{q+1})$ is that $I(L_{q+1})$ has reduced rate limits.

Since the $q+1$st recursive call to RouteChains results (recursively) in only i absorptions, and the input to the $q+1$st recursive call is valid (by Lemma 3.5), by induction the $q+1$st recursive call outputs a routing K_{q+1} for $I(L_{q+1})$ that is optimal even if all precedence constraints are removed.

Let $\hat{K}$ denote the set of flow assignments (π, x) that are computed in the first q recursive calls made by RouteChains(L). Thus RouteChains(L) returns the routing $\hat{K} \bigcup K_{q+1}$.

Let L' be L with the precedence constraints removed. Let $I(L')$ be the MTTC instance associated with L' (so $I(L')$ is just $I(L)$ with precedence constraints removed). Compare the execution of RouteChains(L) and RouteChains(L'). In the first $q-1$ recursive calls, the computed flow assignments (π, x) are the same for L and L'. The (π, x) computed in the qth call for L and L' both use the same permutation π; the value of x may be greater for L' (since a parent-child absorption was not triggered), but cannot be less. Since running RouteChains on L' is guaranteed to produce an optimal routing for $I(L')$ (by the base case, since there are no parent-child absorptions), there must be a way to augment $\hat{K}$ with additional flow to obtain an optimal routing for $I(L')$. The routing output by running RouteChains on L' can thus be written as the union of two routings, $\hat{K}$ and a routing $\tilde{K}$, where $\tilde{K}$ is an optimal routing for MTTC instance $I(L_{q+1})$ with precedence constraints removed.

When running RouteChains on instance L, starting from the $q+1$st recursive call, the algorithm attempts to solve $I(L_{q+1})$, but with precedence constraints. By induction, it finds a routing that is optimal even without the precedence constraints, and thus has the same throughput as $\tilde{K}$. This routing, combined with $\hat{K}$, is therefore optimal for both $I(L)$ and $I(L')$. □

It is worth noting the following lemma, which is a direct consequence of Lemma 3.6.

LEMMA 3.7. *If I is an MTTC instance whose precedence graph is a forest of proper chains, then the maximum throughput attainable for I would be the same even if precedence constraints were removed.*

The following lemma follows directly from the above two lemmas, and Lemma 3.3.

LEMMA 3.8. *Let I be an MTTC instance whose precedence graph is a forest of proper chains. Then the routing produced by RouteChains for I has the saturated suffix property.*

This last lemma suggests that it may be possible to modify the RouteChains procedure so that the saturated suffix property can be proved more directly. We leave this as an open question.

The following theorem gives bounds for the MTTC algorithm for chains.

THEOREM 3.1. *There is an $O(n^2 \log n)$ algorithm for solving the MTTC problem when the precedence graph consists of a forest of chains. The algorithm outputs a routing that uses at most $4n-3$ distinct permutations.*

Proof. The algorithm consists of the preprocessing procedure that makes chains proper and sorts the operators, and the RouteChains algorithm, run on the initial inputs as described above. Correctness follows immediately from the previous lemmas.

The running time of the algorithm is dominated by the time taken by RouteChains. Using standard data structures, RouteChains can be implemented so that each recursive call takes time $O(n \log n)$. The modified topological sort can be run in time $O(n \log n)$ because its input graph has at most n edges. Each recursive call adds one (π, x) to the output routing.

Let r be the number of parent-child absorptions, and t the number of swap-merges. The number of recursive calls is $r + t + 1$. Each swap-merge decreases the number of *l*-superoperators by 1 and each parent-child absorption increases it by at most 2. The initial number of *l*-superoperators is n, the final number is at least 1, so $n+2r-t \geq 1$. Since an absorbed child always remains with its parent, $r \leq n-1$ and hence $t \leq 3n-3$. The stated bounds follow. □

3.5 MTTC Algorithm for General Trees

We now describe the MTTC algorithm for arbitrary tree-structured precedence graphs. Define a "fork" to be a node in G with at least two children; a chain has no forks. Intuitively, the algorithm works by recursively eliminating forks from G, bottom-up. Given a fork in a tree of G whose subtrees are chains, the MTTC algorithm essentially replaces those chains (in a strategic way) by a single new chain; each node in the new chain is a superoperator made up of a subset of operators from the replaced chains. For each superoperator, the MTTC algorithm runs RouteChains on the operators in it to compute an optimal routing. The algorithm then executes a recursive call on the modified graph, which has one less fork. Finally, it combines the routings computed for the superoperators with the routing computed during the recursive call.

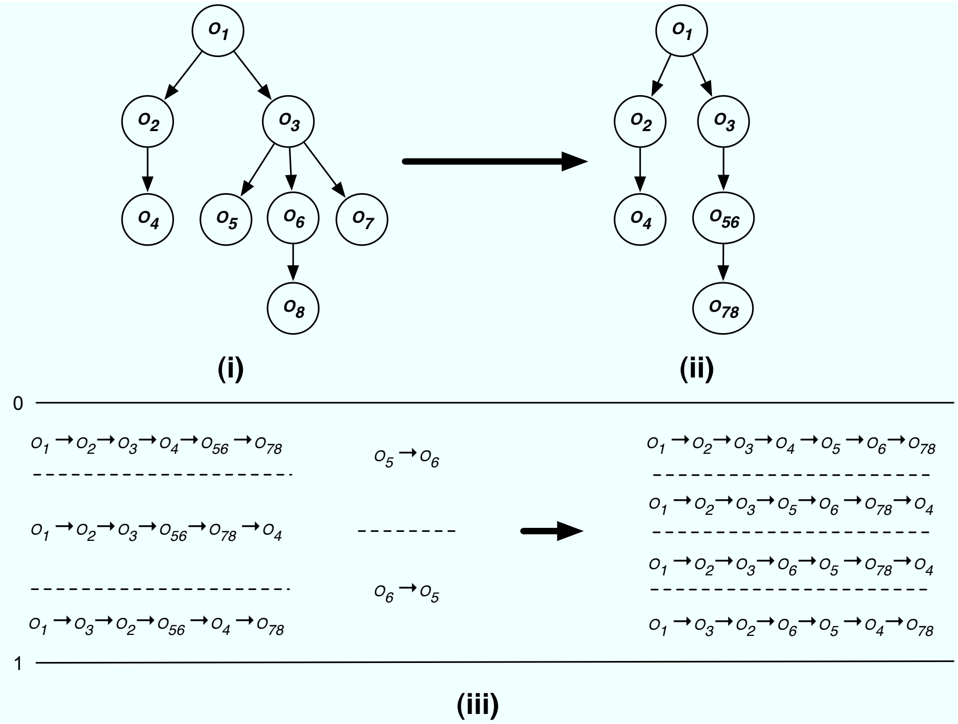
3.5.1 *Example.* To provide intuition, we illustrate the execution of one recursive call to the MTTC algorithm. (We actually illustrate a simplified process to give the intuition; we discuss the actual process below in Section 3.5.2.)

Let the input graph be the one shown in Figure 4 (i). This graph has several forks; let the next fork we eliminate be at node O_3. The subtrees under O_3 form a forest of three chains. A new set of operators is constructed from this forest of chains as follows:

- The three chains are made proper.
- RouteChains is used to find an optimal routing K' for these three chains. Suppose that $\{O_7, O_8\}$ is a saturated suffix of K'. Let K_{78} denote the routing (over O_7 and O_8) derived from K' that saturates O_7 and O_8.
- A new operator O_{78} is constructed corresponding to O_7 and O_8. Its rate limit is set to be the throughput achieved by K_{78}, and its selectivity is set to $p_7 p_8$.
- O_7 and O_8 are removed from the three chains, and from the sorted list of operators. RouteChains is applied to operators O_5 and O_6. Suppose the output routing K_{56} saturates both operators.
- A new operator O_{56} is constructed to contain K_5 and K_6. Its rate limit is set to the throughput of K_{56} and its selectivity is set to $p_5 p_6$.
- A new precedence graph is constructed as shown in Figure 4 (ii).

Having eliminated a fork from the graph, the resulting problem is recursively solved to obtain a routing K'', which is then combined with K_{56} and K_{78} to obtain a routing for the original problem, using a technique from [Condon et al. 2009]. We illustrate this with an example (Figure 4 (iii)).

Suppose K'', the optimal solution for the reduced problem (Figure 4 (ii)), uses three permutations:
$(O_1, O_2, O_3, O_4, O_{56}, O_{78})$, $(O_1, O_2, O_3, O_{56}, O_{78}, O_4)$, and $(O_1, O_2, O_3, O_{56}, O_4, O_{78})$, and let the total flow be t. Further, suppose the first and third permutations each carry $\frac{1}{4}t$ flow, and the second carries $\frac{1}{2}t$ flow. Similarly, suppose routing K_{56} for O_{56} sends half the flow along permutation (O_5, O_6) and half along (O_6, O_5). These two routings are shown graphically in the first two columns of Figure 4 (iii). In each column, the height of the region allocated to the three permutations indicates the fraction of flow allocated to that



o1
o2
o3
o4
o5
o6
o7
o8
(i)
o1
o2
o3
o4
o56
o78
(ii)
0
o1 → o2 → o3 → o4 → o56 → o78
o1 → o2 → o3 → o56 → o78 → o4
o1 → o3 → o2 → o56 → o4 → o78
o5 → o6
o6 → o5
o1 → o2 → o3 → o4 → o5 → o6 → o78
o1 → o2 → o3 → o5 → o6 → o78 → o4
o1 → o2 → o3 → o6 → o5 → o78 → o4
o1 → o3 → o2 → o6 → o5 → o4 → o78
1
(iii)

Fig. 4. (i) An example precedence graph; (ii) The forest of chains below operator O_3 is replaced by a single chain to obtain a new problem; (iii) The solution for the new problem and for the operator O_{56} are combined together.

permutation by the associated routing. In the third column we superimpose the divisions from the first two columns. For each region R in the divided third column, we label it with the permutation obtained by taking the associated permutation from column 1, and replacing O_{56} in it with the associated permutation from column 2. For example, the second region from the top in the third column is associated with $(O_1, O_2, O_3, O_{56}, O_{78}, O_4)$ from column 1 and (O_5, O_6) from column 2, and is labeled by combining them. Column three represents a division of flow among permutations of all the operators, yielding a final routing that divides t units of flow proportionally according to this division. The resulting routing allocates $\frac{1}{4}t$ flow to each of four permutations. The same approach would be used to incorporate the routing for K_{78} into the overall routing.

Note that the MTTC algorithm, like RouteChains, works by repeatedly combining individual operators into new superoperators. In the above example, O_5 and O_6 are combined to form the superoperator O_{56}. Although O_{56} is not an *l*-superoperator, it shares an important property with the *l*-superoperators: there is a routing for its component operators (namely K_{56}) that will saturate both of them. Therefore, when routing K'' is combined with K_{56} (to obtain a routing that uses O_5 and O_6, rather than O_{56}), the resulting routing either saturates both O_5 and O_6, or neither of them. The same is true for O_7 and O_8.

3.5.2 *MTTC algorithm description.* The MTTC algorithm is a recursive procedure that works by repeatedly eliminating forks in the precedence graph. We describe the steps in the MTTC algorithm here. The algorithm uses two procedures, *CombineRoutings* and *AggregateChains*, described below.

MTTC Algorithm: Inputs

1. Rate limits $r_1, \ldots, r_n$ (all > 0), and selectivities $p_1, \ldots, p_n$ (all strictly between 0 and 1) for a set of operators $\mathcal{O} = \{O_1, ..., O_n\}$,
2. A precedence graph G with vertex set $\mathcal{O}$, where G is a forest of trees.

MTTC Algorithm: Output

The algorithm returns an optimal routing K for the MTTC instance defined by the inputs.

MTTC Algorithm: Execution

1. Base Case: If G is a forest of chains, apply the MTTC algorithm for chains (Section 3.4) and return the solution.
2. Otherwise, let O_i be a fork of G whose subtrees are all chains.
3. Let S denote the set of descendants of O_i, and let I_S denote the induced MTTC instance restricted to the operators in S. The precedence graph G_S of I_S is thus a forest of chains.
4. Make the chains in I_S proper (Section 3.4.1), and call AggregateChains(I_S) to get a partition $(A_1, \cdots, A_m)$ of the operators of I_S, and routings $K_{A_1}, \cdots, K_{A_m}$ corresponding to the sets of the partition. For each K_{A_i}, let $\tau(K_{A_j})$ denote its throughput.
5. Create m new operators, $\mathcal{A}_1, \cdots, \mathcal{A}_m$ corresponding to the A_i's. For each $\mathcal{A}_i$, define its rate limit $\rho(\mathcal{A}_i)$ to be $\tau(K_{A_i})$, and its selectivity $\sigma(\mathcal{A}_i)$ to be the product of the selectivities of the operators in A_i.
6. Construct a new precedence graph G' from G by replacing the chains below O_i with the single chain $(\mathcal{A}_1, \ldots, \mathcal{A}_m)$. Thus G' has one fewer fork than G.
7. Let I' be the resulting new MTTC instance, having precedence graph G'.
8. Recursively solve I'. Let K' be the routing returned by the recursive call.
9. Use CombineRoutings to combine the K_{A_i} with K'. Return the resulting routing.

AggregateChains: AggregateChains is used in fork elimination, to replace a set of chains emanating from a fork by a single chain. In the example presented above in Section 3.5.1, to eliminate a fork we ran RouteChains repeatedly on the chains emanating from the fork, each time removing the operators in a saturated suffix. For efficiency, AggregateChains does something slightly different. It first uses a procedure, described below, to identify the operators in a saturated suffix of some optimal routing (without actually computing the routing); it then runs RouteChains just on the operators in that saturated suffix to produce a (saturating) routing just for those operators. As in the example, it then removes the operators in the suffix, and repeats. For each saturated suffix, AggregateChains outputs the set of operators in it, and the routing computed for it.

Formally, AggregateChains takes as input an instance I of the MTTC problem whose precedence graph G is a forest of proper chains. AggregateChains first sorts the operators in I in descending order of their rate limits. It (re)numbers them $O_1, \ldots, O_n$ so that $r_1 \geq \ldots \geq r_n$. It then executes the following recursive procedure on I. It computes the value

$$F^* = min_{k \in \{1,\ldots,n\}} \frac{\sum_{i=k}^{n} r_i(1-p_i)}{(\prod_{j=1}^{k-1} p_j)(1-\prod_{i=k}^{n} p_i)}.$$

It forms the set $C_{j^*} = \{O_{j^*}, \ldots, O_n\}$, where j^* is the largest value of j achieving the minimum value F^*. Since the chains of G are proper, the operators in C_{j^*} correspond to the operators in (possibly empty) suffixes of the chains of G. AggregateChains runs RouteChains just on the (sub)chains of operators in C_{j^*} to produce a routing K_{j^*}.

By Lemma 3.4, there is a routing for I with precedence constraints removed that has C_{j^*} as a saturated suffix. Therefore, the induced instance on C_{j^*}, with precedence constraints removed, has a routing that saturates all its operators. By Lemmas 3.6 and 3.2, K_{j^*} is optimal for I and saturates all the operators in C_{j^*}.

After computing K_{j^*}, AggregateChains then removes the operators in C_{j^*} from I. If no operators remain in the chains, AggregateChains outputs the one-item list A_1 where $A_1 = C_{j^*}$, together with routing $K_1 = K_{j^*}$. Otherwise, AggregateChains executes a recursive call on the remaining operators in the chains (and the induced precedence graph) to produce a partition of the operators $D = (A_1, \ldots, A_{m-1})$, together with a corresponding list $K_{A_1}, \ldots, K_{A_{m-1}}$ of saturating routings for the operators in each of the A_i. It then sets $A_m = C_{j^*}$, appends it to the end of D, appends K_{j^*} to the end of the associated list of routings, and outputs the resulting partition $D = (A_1, \ldots, A_m)$ and list of routings $K_{A_1}, \ldots, K_{A_m}$.

CombineRoutings: CombineRoutings is used to combine the routings $K_{A_1}, \ldots, K_{A_m}$ with K'. The approach is described fully in [Condon et al. 2009] and has been illustrated with an example above (Figure 4 (iii); Section 3.5.1). We describe it briefly here. It begins by dividing the real interval $[0, 1]$ once for K', and once for each K_{A_i}. The division for a routing divides $[0, 1]$ into as many subintervals as there are permutations used in the routing, each of these permutations is associated with an interval, and the length of each subinterval is equal to the proportion of flow the routing sends along the corresponding permutation. It superimposes these divisions to produce a new division that yields a combined routing for K' and the K_{A_i}, in a manner analogous to what was shown in the example. If P_{A_i} is the number of permutations used in K_{A_i}, and $P_{A'}$ is the number of permutations used in $P_{A'}$, then the number of permutations used in the combined routing is at most $P_{A'} + (\sum_{i=1}^{m} P_{A_i}) - m$.

3.5.3 *Analysis.* We now prove that the MTTC algorithm outputs an optimal routing.

The idea behind the MTTC algorithm is the same as the idea behind the chains algorithm: to produce a routing with a saturated suffix. Again, this may be impossible due to precedence constraints.

Let k be the number of recursive calls performed by the MTTC algorithm (including the initial call). Let $I_1, \ldots, I_k$ be the inputs to the k recursive calls. For convenience, define $I_0 = I_1$.

Let O_i^j denote the ith operator in I_j, the input to the jth recursive call of the MTTC algorithm. O_i^j is either a trivial superoperator containing one original operator, or was formed by recursively combining operators in recursive calls $1, \ldots, j-1$. We say that an operator O_i^j *contains* the original operators that were recursively combined to create it. We also say that it contains the intermediate operators that were combined during the recursive combination (including itself).

During the running of the MTTC algorithm, rate limits of operators are reduced. Suppose that each time we reduce the rate limit of an operator O_i^j by a factor c during a recursive call, we also reduce the rate limits of the original operators contained in O_i^j by the same factor c. At the end of the algorithm, each original operator has a final, reduced rate limit.

Let r_i^j denote the rate limit of O_i^j at the start of the jth call. Let s_i^j denote the multiplicative factor by which the rate limit of O_i^j is reduced in the jth call. If the rate limit of O_i^j is not reduced in the jth call, then $s_i^j = 1$.

We show that the routing output by the algorithm has a saturated suffix *with respect to the final reduced rate limits of the original operators*, and hence is optimal for those reduced rate limits. We then show that the routing is also optimal with respect to the original rate limits.

We begin by proving that certain rate limit reductions do not affect the total throughput attainable.

Given a precedence graph G and an operator O_i in G, define an *upper descendant set* of O_i to be a subset S of the descendants of O_i, not including O_i itself, such that for any operator O_j in S, S contains all operators lying on the subpath in G from O_i down to O_j (excluding O_i itself).

LEMMA 3.9. *Let I be an input instance for the MTTC algorithm. Let O_i be an operator in I. Let S be an upper descendant set of O_i, and let I_S be the MTTC instance induced from I by keeping just the operators in S, and the associated induced precedence graph. Suppose the maximum throughput attainable for I_S is more than $r_i p_i$. If we reduce the rate limits of a subset of the operators in S, such that the maximum throughput attainable on I_S is still at least $r_i p_i$, then the maximum throughput attainable on I is unchanged by those rate limit reductions.*

Proof. Let K_S be an optimal routing for I_S, *after* the rate reductions are performed on S. Since K_S is optimal, the throughput achieved by it is at least $r_i p_i$. We will use K_S below.

Now define a new MTTC problem I^∞, which is equal to the original I, except that we modify the rate limits of the operators in S to be ∞.

Clearly the maximum throughput attainable on I^∞ is at least as large as that attainable on I.

Consider an optimal routing K^∞ for I^∞. Without loss of generality, we can assume that in each permutation used by K^∞, the operators in S appear in some order immediately after operator O_i. If not, we can move them forward in the permutation to appear immediately after O_i, preserving their relative order: this can ony reduce the flow into other operators, so rate limits are still obeyed, and because S is an upper descendant set, precedence constraints are obeyed also.

Define a new (super)operator O_S whose selectivity is the product of the selectivities of the opeators in S, with infinite rate limit. In each permutation of K^∞, remove the operators in S (which appear consecutively) and replace them with O_S. This yields a modified routing K_1^∞ that uses O_S instead of S. Since O_S always appears immediately after O_i in K_1^∞, the total amount of flow entering O_S in K_1^∞ is at most $r_i p_i$.

Since the throughput achieved by routing K_S is at least $r_i p_i$, we can combine routing

K_1^∞ with routing K_S to obtain a new routing for I that achieves the same throughput as K^∞ and respects the reduced rate limits of operators in S. Thus the reduction in rate limits does not affect the maximum throughput attainable on I. □

We now prove an important property of the new operators $\mathcal{A}_1, \ldots, \mathcal{A}_m$ created as a result of the call to AggregateChains in Step 4 of the MTTC algorithm.

LEMMA 3.10. *The new operators $\mathcal{A}_1, \ldots, \mathcal{A}_m$ created in a recursive call of the MTTC algorithm have the property that for each $i \in \{2, \ldots, m\}$, $\tau(K_{A_{i-1}})\sigma(A_{i-1}) \geq \tau(K_{A_i})$ (i.e., $\mathcal{A}_{i-1}$ can saturate $\mathcal{A}_i$). The computed routings $K_{A_1}, \ldots, K_{A_m}$ are saturating routings for the induced MTTC instances on $A_1, \ldots, A_m$ respectively (with respect to the rate limits of the operators in the A_i when the K_{A_i} are computed).*

Proof. Consider the call to AggregateChains that was used to create $A_1, \ldots, A_m$. The chains input to AggregateChains were made proper prior to the call. In what follows, when we talk about issues involving rate limits (such as optimality of routing, saturating routings, or proper chains), we mean with respect to the rate limits of the operators at the start of AggregateChains.

By Lemma 3.2, for each $i \in \{1, .., m\}$, there exists a routing K'_i that is optimal for the induced MTTC instance on the operators in $A_1 \bigcup \ldots \bigcup A_i$ with precedence constraints removed, such that A_i is a saturated suffix of K'_i. Let $\tau(K'_i)$ denote the throughput of K'_i.

The second part of this lemma, which says that $K_{A_1}, \ldots, K_{A_m}$ are saturating routings for $A_1, \ldots, A_m$, follows directly from the argument in the description of AggregateChains. We expand on that argument here to prove the first part of this lemma.

Now let $i \in \{2, \ldots, m\}$. Since A_i is a saturated suffix of K'_i, routing K'_i sends all flow first through the operators in $A_1 \bigcup \ldots \bigcup A_{i-1}$ and then through the operators in A_i, which it saturates. The amount of flow reaching A_i in routing K'_i is $\tau(K'_i)\sigma(A_1)\sigma(A_2)\ldots\sigma(A_{i-1})$, and thus $\tau(K'_i)\sigma(A_1)\sigma(A_2)\ldots\sigma(A_{i-1}) = \tau(K_{A_i})$. Similarly, $\tau(K'_{i-1})\sigma(A_1)\sigma(A_2)\ldots\sigma(A_{i-2}) = \tau(K_{A_{i-1}})$.

If we remove from K'_i the portion of the routing that goes through A_i, we are left with a routing through $A_1 \bigcup \ldots \bigcup A_{i-1}$ that still achieves throughput $\tau(K'_i)$. Since K'_{i-1} is optimal, $\tau(K'_{i-1}) \geq \tau(K'_i)$. It follows that

$$\begin{aligned} \tau(K_{A_{i-1}})\sigma(A_{i-1}) &= \tau(K'_{i-1})\sigma(A_1)\sigma(A_2)\ldots\sigma(A_{i-1}) \\ &\geq \tau(K'_i)\sigma(A_1)\sigma(A_2)\ldots\sigma(A_{i-1}) \\ &= \tau(K_{A_i}) \end{aligned}$$

which completes the proof. □

Before proceeding, we define some additional notation.

Let $j' \geq j$. If in call j', the operator containing O_i^j is $O_{i'}^{j'}$, then we define $S(O_i^j, j') = s_{i'}^{j'}$, i.e., $S(O_i^j, j')$ is the multiplicative factor by which the rate limit of the operator containing O_i^j is reduced in call j'. Thus $S(O_i^j, j) = s_i^j$.

For l such that $j \leq l \leq k$, define $R(O_i^j, l) = r_i^j \prod_{j'=j}^{l} S(O_i^j, j')$. Thus $R(O_i^j, l)$ reflects the rate reduction performed on O_i^j in call j, and all rate reductions performed on operators containing O_i^j in calls $j+1$ through l.

Define K^j to be the routing returned by the jth recursive call.

The following lemma states that in the jth recursive call, for each operator O_i^j, there is a saturating routing on the original operators contained within it. Here saturation is defined with respect to the rate limits that reflect the reductions performed in calls $1, \ldots, j-1$.

LEMMA 3.11. *For each operator O_i^j in the jth recursive call of the MTTC algorithm, there exists a routing K_i^j on the set of original operators contained in O_i^j with the following properties:*

—*K_i^j obeys the precedence constraints in G.*

—*The throughput achieved by K_i^j is r_i^j.*

—*For each original operator O_l^1 contained in O_i^j, the amount of flow reaching O_l^1 in K_i^j is $R(O_l^1, j-1)$.*

Proof. The proof is by induction on j. It trivially holds for $j = 1$. Assume it holds for $j-1$, where $j \geq 2$. We show it holds for j.

Consider an operator O_i^j from call j. Suppose first that O_i^j was formed from operators below the eliminated fork in call $j-1$, i.e., O_i^j corresponds to some $\mathcal{A}_z$ created in call $j-1$ by combining the operators in A_z. Saturating routing K_{A_z} was computed for the operators of A_z. Let $O_{i'}^{j-1}$ be an operator in A_z.

The initial rate limit $r_{i'}^{j-1}$ of $O_{i'}^{j-1}$ is reduced by a factor of $s_{i'}^{j-1}$ in call $j-1$. By Lemma 3.10, routing K_{A_z} constructed for $\mathcal{A}_z$ is saturating for its component operators, implying that the amount of flowing reaching $O_{i'}^{j-1}$ in K_{A_z} is $r_{i'}^{j-1} s_{i'}^{j-1}$, its rate limit after the reduction.

By induction, there is a routing $K_{i'}^{j-1}$ for the original operators contained in $O_{i'}^{j-1}$ achieving throughput $r_{i'}^{j-1}$, such that for each O_l^1 contained in $O_{i'}^{j-1}$, the amount of flow reaching it in $K_{i'}^{j-1}$ is $R(O_l^1, j-2)$. Scaling $K_{i'}^{j-1}$ by a factor of $s_{i'}^{j} - 1$ reduces its throughput to $r_{i'}^{j-1} s_{i'}^{j-1}$, and reduces the amount of flow reaching contained operator O_l^1 to $s_{i'}^{j-1} R(O_l^1, j-2) = R(O_l^1, j-1)$. It follows that applying CombineRoutings to K_{A_z} and the $K_{i'}^{j}$, to incorporate scaled versions of the $K_{i'}^{j}$ into K_{A_z}, yields a routing with the desired properties.

If O_i^j was not formed from operators below the eliminated fork in call $j-1$, then it is the same as an operator from call $j-1$ whose rate limit was not reduced in that call. In this case, the existence of the desired routing follows immediately from the inductive assumption. □

The final recursive call of the algorithm computes a routing K^k for a forest of chains, after those chains are made proper. By Lemma 3.8 K^k has a saturated suffix. We want to show that each operator O_i^k in the saturated suffix has a special property: The routing K output by the MTTC algorithm saturates all original operators in O_i^k, if we define saturation with respect to the final reduced rate limits of the original operators (reflecting all reductions made to containing operators in calls $1, \ldots, k$). We do this by proving the following technical lemma. We get the desired result by taking $j = 1$ in the lemma.

LEMMA 3.12. *The following holds for each operator O_i^j in the jth recursive call of the MTTC algorithm: if K^k saturates the operator in I_k containing O_i^j (where saturation is defined respect to the rate limit of that operator at the end of call k), then in K^j, the amount of flow reaching operator O_i^j is exactly $R(O_i^j, k)$.*

Proof. The proof is by backwards induction on j. For the base case, let $j = k$. If O_i^k is an operator in I_k, and its rate limit is reduced by a factor of s_i^k during that iteration, then if routing K^k saturates O_i^k (with respect to O_i^k's final rate limit), the amount of flow reaching O_i^k in K^k is clearly $R(O_i^k, k)$.

Suppose the lemma is true for j. We show it is true for $j-1$. There are two types of operators in I_{j-1}: those that are in the chains below the fork that is removed in call $j-1$, and those that are not. We argue that the lemma holds for both types.

Consider recursive call $j-1$. Let O_i^{j-1} be an operator in call $(j-1)$ that is contained in a saturated operator of K^k.

Suppose first that O_i^{j-1} is in a chain below the eliminated fork, and is incorporated into an operator $\mathcal{A}_z$ which becomes operator O_l^j in I_j. By Lemma 3.10, routing K_{A_z} constructed for $\mathcal{A}_z$ is saturating, so the amount of flowing reaching O_i^{j-1} in K_{A_z} is $r_i^{j-1} s_i^{j-1}$. By induction, in routing K^j for I_j, the amount of flow reaching O_l^j is $R(O_l^j, k)$.

The throughput of K_{A_z} is $\tau(K_{A_z}) = r_l^j$. Let $t = R(O_l^j, k)/r_l^j$, i.e., t is the product of the factors by which the operators containing O_l^j are reduced in calls $j, \ldots, k$. Thus $R(O_i^j, k) = r_i^{j-1} s_i^{j-1} t$. In call $j-1$, CombineRoutings incorporates a scaled version of K_{A_z} into K^j to produce K^{j-1}. Since in K^j, $R(O_l^j, k)$ amount of flow reaches O_l^j, routing K^{j-1} sends a total of $R(O_l^j, k)$ flow through the operators in A_z, using K_{A_z} scaled down by a factor of $R(O_l^j, k)/\tau(K_{A_z})$. Since in K_{A_z}, the amount of flow reaching O_i^{j-1} is $r_i^{j-1} s_i^{j-1}$, the amount of flow reaching O_i^{j-1} in K^{j-1} is $r_i^{j-1} s_i^{j-1} R(O_l^j, k)/\tau(K_{A_z}) = r_i^{j-1} s_i^{j-1} t = R(O_i^{j-1}, k)$.

Now suppose that O_i^{j-1} is not in one of the chains below the eliminated fork. In this case, its rate limit not reduced in call $j-1$, and it is not combined with other operators during that call. It follows easily by induction that the amount of flow reaching O_i^{j-1} in K^{j-1} is $R(O_i^{j-1}, k)$. □

LEMMA 3.13. *Let S be a saturated suffix of the operators in K^k. Let S_j be the set of operators in I_j contained in the operators of S. Then in every permutation used in K^j, the operators in S_j form a suffix of the permutation.*

Proof. Using the recursive construction of K^j from K^k, it is easy to see that each permutation π used in K^j can be derived from a permutation π' in K^k by replacing each O_i^k in π' by the operators from I^j contained in it (listed in the appropriate order). Since the operators of S appear at the end of every permutation used by K^k, the lemma follows. □

We now prove correctness of the MTTC algorithm.

LEMMA 3.14. *The MTTC algorithm outputs an optimal routing.*

Proof: Let I be the input instance. The algorithm clearly outputs a routing satisfying all rate and precedence constraints of I. We show that the routing achieves maximum throughput.

Let $\tilde{I}_1, \ldots, \tilde{I}_k$ be the same as the initial input instance $I = I_1$, except that in $\tilde{I}_j$ the rate limit of $O_i (= O_i^1)$ is replaced with $R(O_i^1, j)$. Define $\tilde{I}_0 = I$. Thus the rate limits of the operators in $\tilde{I}_j$ are greater than or equal to the corresponding operators in $\tilde{I}_{j+1}$.

Note that the operators of each of the $\tilde{I}_j$ are precisely the original operators of the input instance I. In contrast, the operators in I_j, the input to the jth recursive call of the MTTC algorithm on input I, may be superoperators containing many of the original operators.

The proof is in two parts. We first show that the final routing output by the MTTC algorithm on input I achieves maximum throughput with respect to the rate limits in $\tilde{I}_k$. We then show that the maximum throughput attainable is the same for each of the $\tilde{I}_j$, implying that the final output routing is also optimal for $I_0 = I$, which proves the lemma.

Part 1: The final output routing achieves maximum throughput with respect to the rate limits of $\tilde{I}_k$.

Consider the final recursive call of the algorithm. The input I_k to this call consists of a forest of chains. The chains are first made proper, which may entail reducing some of the rate limit. These reductions are reflected in $\tilde{I}_k$. A routing is then computed for the chains, using RouteChains. By Lemma 3.8, this routing has a saturated suffix S.

Let S_1 be the set of original operators contained in the operators of S. Since the final routing output by the algorithm is $K^1 = K$, by Lemma 3.12 the amount of flow reaching each operator $O_i^1 \in S_1$ in routing K is $R(O_i^1, k)$. By Lemma 3.13, the operators in S_1 are a suffix in every permutation used by K. Thus S_1 is a saturated suffix of K with respect to the rate limits of $\tilde{I}_k$, and hence K is optimal with respect to those rate limits.

Part 2: For all $j \in \{1, \ldots, k\}$, the maximum throughput attainable on $\tilde{I}_j$ is the same as it is for $\tilde{I}_{j-1}$.

The statement clearly holds if $\tilde{I}_{j-1} = \tilde{I}_j$, i.e., if no rate limits are reduced in call j. Suppose not.

Consider the execution of call j. In call j, a fork is removed from the precedence graph. Consider the chains below that fork. Since $\tilde{I}_{j-1} \neq \tilde{I}_j$, some of the chains below the fork are not proper at the start of call j, and the rate limits of some of the operators in them are reduced in call j.

We call an operator in call j a “bottleneck operator” if it is in a chain below the removed fork but does not have its rate limit reduced in call j, but its successor in the chain does have its rate limit reduced in call j.

We claim that (1) each bottleneck operator in call j is a trivial superoperator consisting of only one original operator and (2) the rate limit of the bottleneck operator was not reduced in previous calls (i.e., its rate limit has the same value in $\tilde{I}_1, \ldots, \tilde{I}_{j-1}$). To prove the claim, note that operators are only combined by the AggregateChains procedure. If a chain $\mathcal{A}_1, \cdots, \mathcal{A}_m$ is formed during a call to AggregateChains, then by Lemma 3.10 each $\mathcal{A}_i$ in the chain can saturate its successor. Further, each time the rate limit of an operator is reduced, it becomes able to exactly saturate its successor. Since at the start of call j, each bottleneck operator cannot saturate its successor, the claim follows.

Now consider a bottleneck operator O_b in a chain of I_j. Consider the maximal subchain below O_b whose rate limits were reduced. After the reduction in rate limits, it is still possible to route $r_b p_b$ flow down that subchain without violating the rate limits of any of the operators in the subchain, where r_b and p_b are the rate limit and selectivity of the bottleneck operator. Let S denote the set of original operators contained in the operators in the subchain. Note that S is an upper descendant set of O_b in the original precedence graph (as defined right before the statement of Lemma 3.9). For each operator O_i^j in the subchain, let K_i^j be the routing on the original operators contained in O_i^j whose existence is guaranteed by Lemma 3.11. Using the K_i^j, one can convert the routing sending $r_b p_b$ flow down the subchain into a routing through the original operators in S that achieves throughput $r_b p_b$, by scaling each of the K_i^j by s_i^j. Further, since K_i^j sends at most $R(O_l^1, j-1)$ flow into each original operator O_l^1, the converted routing sends at most $s_i^j R(O_l^1, j-1) = R(O_l^1, j)$

flow into operator O_l^1. Thus if we take the MTTC instance $\tilde{I}_{j-1}$, and reduce the rate limits of all O_l^1 in S to $R(O_l^1, j)$, the throughput attainable on the induced instance consisting only of operators in S is at least $r_b p_b$. It follows from Lemma 3.9, that reducing the rate limits of operators O_l^1 in S from $R(O_l^1, j-1)$ to $R(O_l^1, j)$ in $\tilde{I}_{j-1}$ does not reduce the throughput attainable. Applying this argument successively to all bottleneck operators, it follows that the maximum throughput attainable on $\tilde{I}_j$ is the same as it is for $\tilde{I}_{j-1}$. □

THEOREM 3.2. *There is an algorithm that solves the MTTC problem that runs in time $O(n^3)$ and outputs a routing using fewer than $4n-3$ distinct permutations.*

Proof. Optimality was shown in Lemma 3.14. Here we analyze the running time and number of permutations.

We can view the creation of the (super)operators used in the MTTC algorithm as a hierarchical process, where new operators in the jth call are created out of sets of operators from the $(j-1)$th call. We can represent this as a forest Z of trees, where the roots of the trees are the operators in the final recursive call of the MTTC algorithm, each node in the tree corresponds to a (super)operator in the algorithm, and the children of an operator are the operators that were combined to create it. If an operator in the jth call is the same as an operator in a later call, i.e., contains the same original operators, we only represent that operator once in this tree (i.e., we collapse chains in which nodes have only one child), so each internal node has at least two children.

We consider a call to RouteChains to be non-trivial if the input to the call consists of more than one operator.

Claim 1: The amount of time spent in non-trivial calls to RouteChains is $O(n^2 \log n)$.

Each non-trivial call to RouteChains corresponds to the creation of a new operator from a set of two or more operators. Thus the internal nodes of Z correspond exactly to the non-trivial calls to RouteChains. The number of leaves in Z is n, and since the internal nodes of each tree all have at least two children, the number of internal nodes is at most $n-1$. Thus the number of edges in Z is at most $2n-2$.

If an internal node in Z has m children, then the corresponding call to RouteChains is on m operators, and takes time $O(m^2 \log m)$. If m_N denotes the number of children of node N, then the total time spent running RouteChains is upper bounded by $\sum_{N \in \mathcal{N}} c m_N^2 \log m_N$, for some constant c, where $\mathcal{N}$ is the set of internal nodes in Z.

The total number of children of all the internal nodes is equal to the number of edges in the forest Z. Thus $\sum_{N \in \mathcal{N}} m_N \leq 2n-2$, and hence $\sum_{N \in \mathcal{N}} m_N^2 \leq (2n-2)^2$. It follows that $\sum_{N \in \mathcal{N}} c m_N^2 \log m_N$ is $O(n^2 \log n)$, which proves the first claim.

Claim 2: The number of permutations used in the output routing is at most $4n-3$.

Consider the recursive construction of the output routing. In the final recursive call of the algorithm, a routing K^k is constructed on the forest of chains. Let n' be the number of (super) operators in that forest of chains. By Theorem 3.1, the number of permutations used in K^k is at most $4n'-3$.

Routing K^k is passed back through the recursive calls. At each call, CombineRoutings incorporates into it the routings K_{A_i}. If P_{A_i} is the number of permutations used in K_{A_i}, then incorporating K_{A_i} increases the number of permutations in the constructed routing by at most $P_{A_i} - 1$.

Let κ be the set of all K_{A_i} that are incorporated into K^k as it is passed back through the recursive calls. Thus the total number of permutations used in the final routing is at most

$(4n' - 3) + \sum_{K_{A_i} \in \kappa}(P_{A_i} - 1)$. Note that if K_{A_i} is a trivial routing through one operator, then the contribution to the sum is 0. Let κ' denote the set of K_{A_i} that are non-trivial.

Each $K_{A_i} \in \kappa'$ is produced by a non-trivial call to RouteChains. We can thus associate each K_{A_i} with a unique internal node N of Z corresponding to the associated new operator $\mathcal{A}_i$. The size of set A_i is equal to m_N, the number of children of N. By Theorem 3.1, the number of permutations in K_{A_i} is at most $4m_N - 3$.

It follows that $\sum_{K_{A_i} \in \kappa}(P_{A_i} - 1) \leq 4E - 4V_{\mathcal{N}}$, where E is the number of edges of the forest Z, and $V_{\mathcal{N}}$ is the number of internal nodes of Z.

The number of leaf nodes in Z is n. Since n' is the number of operators in the final recursive call, forest Z consists of n' trees. Thus $E = n + V_{\mathcal{N}} - n'$ and hence $(4n' - 3) + \sum_{K_{A_i} \in \kappa}(P_{A_i} - 1) \leq 4n - 3$, proving Claim 2.

Since the number of operators decreases by at least one in each recursive call of the MTTC algorithm, the total number of recursive calls is at most $n - 1$. Outside of the time spent in CombineChains and non-trivial calls to RouteChains, it is easy to see that the amount of time spent in each recursive call is $O(n^2)$.

It remains to bound the time spent running CombineChains. When CombineChains is called, it is used to combine a routing K' with routings K_{A_i}. Since each K_{A_i} is produced by a call to RouteChains, it uses at most $4n' - 3$ permutations, where we now define n' to be the number of operators in K_{A_i}. Since the routing output by the algorithm uses $O(n)$ permutations, K' also uses $O(n)$ permutations. It follows that, using appropriate data structures, each call to CombineChains can be implemented to run in time $O(n^2)$.

Thus the total running time of the algorithm is $O(n^3)$. □

3.6 Reducing the Number of Permutations used in a Routing

Finally, we present a general algorithm for reducing the number of permutations used in the routing to at most n, together with the justification of its correctness. A routing with smaller number of permutations would not only be easier to integrate into a traditional query processor, but would also lead to lower per-tuple overhead. This algorithm uses a standard technique from linear programming that is used in converting an optimal solution to a basic optimal solution (see, e.g., Dantzig and Thapa [2003]). The algorithm runs in polynomial time as long as the initial number of permutations is polynomial in n. Further, we note that the algorithm presented below works whether the operators are selective or non-selective, or a mixture of those.

Suppose we have an optimal routing that uses permutations $\pi_1, \ldots, \pi_t$, where $t > n$. Let a_j denote the amount of flow routed along permutation π_j. Our goal is to obtain a routing that uses at most n permutations, without reducing the total amount of flow.

Write down the n rate constraints, but for each permutation π not used in the routing, set the corresponding flow variable to 0. Add an additional slack variable s_i to each of the rate constraints to change it from an inequality to an equality. Add the constraint $s_i \geq 0$ for each of the slack variables s_i.

This gives us a new LP with equality constraints, rather than inequality constraints, and only t flow variables. Here we designate the flow variables as x_i's, rather than as f_π's.

New LP: Maximize

$$F = \sum_{j=1}^{t} x_j$$

subject to the constraints:

(1) $s_i + \sum_{j=1}^{t} x_j g(\pi_j, i) = r_i$ for all $i \in \{1, \ldots, n\}$

(2) $x_j \geq 0$ for all $j \in \{1, \ldots, t\}$

(3) $s_i \geq 0$ for all $i \in \{1, \ldots, n\}$

Let x^* be the vector of length $t + n$ such that $x_j^* = a_j$ for all $j \in \{1, \ldots, t\}$, and $x_{t+i}^* = r_i - \sum_{j=1}^{t} a_j g(\pi_j, i)$ for all $i \in \{1, \ldots, n\}$. Since the original routing was optimal, x^* is an optimal solution to this new LP.

Let $Ax = b$ be the matrix representation of all the equality constraints of the above LP. The matrix A is $n \times (t + n)$.

Since $t > n$, the first t columns of A are not all linearly independent. Let A' be the submatrix of A consisting just of the first t columns of A. Because of the linear dependence, there is a t-dimensional vector y such that y is non-zero, and $A'y = 0$. Find such a vector using Gaussian elimination. Add an additional n zero entries to y to form a $(t+n)$-dimensional vector y^*. The vector y^* has the property that $\sum_{j=1}^{t} y_j = 0$. (Otherwise, since all entries of x^* are positive, one could add a small multiple of y^* to x^* in the direction that increases $\sum_{j=1}^{t} x_j$ and increase the value of F, contradicting the optimality of x^*.) Find the $j \in \{1, \ldots, t\}$ that minimizes $x_j^*/|y_j^*|$. Let $\alpha = x_j^*/y_j^*$ for that minimizing j. Let z^* denote the vector whose first t components are equal to $x^* + \alpha y^*$, and whose last n components are equal to 0. By linearity, z^* is an optimal solution of the new LP, and it sets at least one of the t flow variables to 0. The flow assignment defined by first t entries of z^* gives a new optimal routing to the original problem, using at least one fewer permutation than the original routing.

The above procedure can be repeated until all columns of A corresponding to flow variables are linearly independent. Thus the number of permutations used can be reduced to n (or fewer).

We included the slack variables s_i in the above discussion to facilitate the justification of correctness. However, the s_i are never actually used in constructing z^*, and the procedure can be carried out without introducing them.

Each iteration of the above procedure can be carried out in time $O(n^3)$. Thus to reduce $O(n)$ permutations to n permutations takes time $O(n^4)$.

4. MTTC: NON-SELECTIVE OPERATORS

In this section, we discuss how the previous algorithm for tree-structured precedence constraints can be extended to handle non-selective operators, which may have selectivities larger than 1. We define the *generalized* MTTC problem to be the generalization of the MTTC problem in which operators may be either selective ($0 < p_i < 1$) or non-selective ($p_i \geq 1$).

We divide the non-selective operators into two types. Those with $p_i = 1$ we call *preserving* operators, and those with $p_i > 1$ we call *expanding* operators.

We begin by considering the case in which all operators are expanding. We show that the algorithm presented in the previous section can be directly used to find the optimal solution in that case. We then develop an optimal algorithm that handles a mixture of selective and non-selective operators, in the case where the precedence graph is restricted to be a forest of chains.

4.1 All Expanding Operators

If all operators are expanding ($p_i > 1$), then we construct an equivalent problem with only selective operators as follows:

—The selectivity p_i of an operator O_i is replaced by $1/p_i$.

—All precedence constraints are reversed.

After solving this new problem (which only contains selective operators), we reverse each of the permutations in the resulting routing to obtain a routing for the original problem. It is easy to see that, because the solution to the new problem is optimal, the resulting routing is optimal for the original problem.

Note that the precedence graph for the new problem may be an "inverted tree". The approach used in Section 3.5 (wherein we replace forests of chains by a single chain) can be extended to handling such precedence graphs.

Define an "inverted fork" to be a node that has more than one parent in the precedence graph. The MTTC algorithm described in Section 3 can be used to solve such instances. The only difference is that, instead of eliminating forks bottom-up, we instead eliminate the "inverted forks" top-down. We call this the MTTC-INV algorithm. We can prove an analogous statement to Theorem 3.2.

THEOREM 4.1. *When run on an MTTC instance I whose precedence graph is an inverted tree and which contains only selective operators, the MTTC-INV algorithm runs in time $O(n^3)$ and outputs an optimal routing for I using fewer than $4n - 3$ distinct permutations.*

Proof: The proof is almost the same as the proof of the analogous theorem for ordinary trees. The only real difference is in justifying the reductions in rate limits when chains are made proper.

Consider running the MTTC-INV algorithm on an instance I with precedence graph G. Each time a chain is made proper, it is either done immediately prior to eliminating a fork (i.e., prior to running CombineChains) or immediately prior to finding a routing on the chains in the final recursive call.

In either case, the chain that is being made proper can easily be shown to consist of a prefix of composite nodes (possibly empty) produced by a run of CombineChains, followed by a suffix (possibly empty) consisting of original nodes from G. By the analysis of CombineChains, the chain of composite nodes is already proper. Suppose the suffix is non-empty. We argue that if a rate reduction is performed on the first node in the suffix, then this rate reduction does not reduce the maximum throughput attainable (for the underlying, original problem instance I). It follows that any rate reductions to the other original operators in the suffix will not reduce the maximum throughput attainable, since all flow into these nodes must first pass through the operators above them in the suffix.

Note first that, from the above, it immediately follows that all rate reductions performed by the algorithm are performed on original nodes; the rate limits of new, composite operators are never reduced.

Let $\mathcal{B}_1, \ldots, \mathcal{B}_m$ denote the chain of operators in the prefix of the current chain, and let $B_1, \ldots, B_m$ denote the underlying sets of original nodes. Let O_{frst} denote the first operator in the suffix. Let $N = \bigcup_{i=1}^{m} B_i$. If the rate limit of O_{frst} is reduced when the chain is made proper, then it is re-set to be the product of the rate limit of $\mathcal{B}_m$ times its selectivity. We use the following claim.

Claim: In any feasible routing for G, the most flow that could ever reach O_{frst} is the product of the maximum-throughput attainable using just the operators in N, times $\sigma(N)$.

Recall that $\sigma(N)$ denotes the product of the selectivities of the operators in N. To prove the claim, consider a modified instance produced by eliminating the rate constraints for all operators in G other than O_{frst} and the operators in N (i.e., setting them equal to infinity). Consider a feasible routing K_{maxA} for this modified instance that maximizes the total amount of flow into O_{frst}. Because the modified rate constraints are less restrictive than the original constraints, the total amount of flow into O_{frst} in K_{maxA} is an upper bound on the maximum amount of flow into O_{frst} that would be attainable under the original constraints. Let $S = \{O_{frst}\} \bigcup N$. In routing K_{maxA}, if there is a permutation in which an operator not in S appears immediately after an operator in N, the order of these two operators can be switched in the permutation without affecting the total amount of flow into O_{frst}, and without violating any rate constraints in the modified instance. Thus we may assume without loss of generality that in routing K_{maxA}, all permutations begin with operators not in S, followed by the operators in N, followed immediately by O_{frst}. The total amount of flow that reaches the operators in N, over all these permutations, cannot be more than the maximum amount of flow that could be routed through just the operators in N. Before reaching O_{frst}, this flow is reduced by $\sigma(N)$. The claim follows.

By Lemma 3.4 and the description of CombineChains, it can be shown inductively that B_m is a saturated suffix of an optimal routing through the operators in N (where saturation is defined with respect to any rate reductions already performed on the operators in N), and the throughput achieved by this optimal routing (and hence by any optimal routing) is the rate limit of the composite operator $\mathcal{B}_m$ times $1/\prod_{i=1}^{m-1} \sigma(B_i)$. It immediately follows from the claim that the maximum total flow into O_{frst} in any feasible routing is at most the rate limit of $\mathcal{B}_m$ times $\sigma(B_m)$. But $\sigma(B_m)$ is the selectivity of $\mathcal{B}_m$, so this is precisely the value to which O_{frst}'s rate limit was re-set. Since total flow into O_{frst} cannot exceed this value, the re-setting does not reduce the maximum throughput attainable. □

4.2 Mixture of Selective and Non-Selective Operators

If the problem instance contains both selective and non-selective operators, the problem is more complex. We present an algorithm for the restricted case in which the precedence graph is a forest of chains (which includes the case where there are no precedence constraints).

The algorithm is based on the following lemma. We define c to be the function which assigns a value to each operator O_i depending on whether it is selective, preserving, or expanding, as follows: $c(O_i) = 1$ if $p_i < 1$, $c(O_i) = 2$ if $p_i = 1$, and $c(O_i) = 3$ if $p_i > 1$. We say that there is a precedence relation between O_i and O_j if one of these two operators is an ancestor of the other in the precedence graph.

LEMMA 4.1. *Given an instance I of the generalized MTTC problem containing both* *selective and non-selective operators, there exists an optimal solution K satisfying the following property: for all O_i, O_j such that $c(O_i) > c(O_j)$,*

—***(A)*** *If O_i and O_j have no precedence relation between them, then O_j does not immediately follow O_i in any permutation used in the routing.*

—***(B)*** *If O_j is the only child of O_i, then O_j immediately follows O_i in every permutation used in the routing.*

Proof: We first prove that there exists an optimal solution K with no violations of **(B)**. We use that to prove that there exists an optimal solution K with no violations of **(A)** or **(B)**.

Part I: We prove by contradiction that there exists an optimal solution K with no violations of **(B)**. Let K be an optimal solution to the problem instance I that minimizes the number of violations of **(B)**. Suppose there exists a permutation π used in the solution that violates **(B)**, i.e., $\exists O_i, O_j$ such that $c(O_i) > c(O_j)$, O_j is the only child of O_i, but O_j does not immediately follow O_i in π.

Note that O_j must appear somewhere after O_i in π (since O_j is the child of O_i). Let $O'_1, \cdots, O'_l$ denote the operators between O_i and O_j. There can be no precedence relation between O'_k and O_i or O_j for any k. This follows from O_j being the only child of O_i.

Let $p' = p'_1 \times \cdots \times p'_l$ denote the combined selectivity of these l operators. We replace the part $(O_i, O'_1, \cdots, O'_l, O_j)$ of π with:

—$O'_1, \cdots, O'_l, O_i, O_j$, if $p' < 1$ (i.e., we move O_i to the right of O'_l).

—$O_i, O_j, O'_1, \cdots, O'_l$, if $p' \geq 1$ (i.e., we move O_j to the left of O'_1).

Call the new routing K'. It is easy to see that the flow into each of the operators either decreases or remains constant after this replacement, so K' still obeys all rate constraints, and is an optimal routing. Further, we can show that K' did not introduce any new violations of **(B)**. Note that a new violation could only occur between two neighboring operators in π that were no longer neighbors after the replacement. In the first case, O'_l and O_j are the only such neighbors, but O_j is not an only child of O'_l, since its parent is O_i. In the second, the only such neighbors are O_i and O'_1, but O'_1 is not an only child of O_i, since O_j is the only child of O_i. Thus K' has one fewer violation of **(B)** than K, which contradicts the minimality property of K.

Part II: Let K be an optimal solution to the problem instance I with no violations of **(B)**. Consider a penalty function that charges a cost for each violation of **(A)** in K. More particularly, for each violation of **(A)** involving operators O_i and O_j occurring in a permutation π used in K, the penalty function charges a cost of t, where O_i is the t^{th} operator in permutation π. Thus violations occuring later in a permutation incur a larger cost than violations occuring earlier. The value of the penalty function on K is the sum of all these costs.

Assume that K is such that the value of the penalty function is minimized. By definition, K has no violations of **(B)**. We show by contradiction that it has no violations of **(A)**.

Suppose there exists a permutation π used in K that violates **(A)**, i.e., $\exists O_i, O_j$ such that $c(O_i) > c(O_j)$, O_j immediately follows O_i in π, and there is no precedence relation between O_i and O_j. Modify π by moving operator O_i past O_j, and then past each operator to its right, one position at a time, until either (1) there is no operator to the right of O_i, or (2) the operator to its right has an equal or higher c value, or (3) there is a precedence relation between O_i and the operator to its right. More formally, let $O'_1, \cdots, O'_l$ be the longest contiguous subsequence of π beginning at the operator to the right of O_j, with the property that $c(O_i)$ is strictly greater than the value of c on any of the operators in this subsequence, and there are no precedence relations between O_i and any of the operators in this subsequence. (This subsequence may be empty.) In π, replace $O_i, O_j, O'_1 \cdots, O'_l$ with $O_j, O'_1, \cdots, O'_l, O_i$. Let K' denote the new routing.

Because $c(O_i)$ is only moved past operators with strictly lower c values, the amount of flow reaching each of the operators in K' is no more than the amount of flow reaching each of the operators in K. Thus K' is still an optimal solution of instance I.

We now argue that K' still has no violations of **(B)** and that the value of the penalty function on K' is lower than the value of the penalty function on K. This is a contradiction, which completes the proof.

There are three cases.

Case 1: After moving O_i in π, there is no operator to its right.

In this case, it is clear that there is no new violation of **(B)**. Moving O_i eliminates the violation of **(A)** caused by O_i and O_j, but may introduce one new violation between O_j and the operator that was the predecessor of O_i in π. Such a new violation has a lower cost than the eliminated violation, so the value of the penalty function is reduced.

Case 2: $c(O_i) \leq c(O'_{l+1})$, where $C(O'_{l+1})$ is the operator to the right of O_i after moving O_i in π.

Let O'' be the predecessor of O_i after O_i is moved. Thus $O'' = O'_l$ if the subsequence is non-empty, and $O'' = O_j$ otherwise. Moving O_i to the right of O'_l could only introduce a new violation of **(B)** involving O'' and O'_{l+1}. But a violation does not occur since $c(O'') < c(O_i) \leq c(O_{l+1})$.

As in Case 1, moving O_i eliminates the violation of **(A)** caused by O_i and O_j, and may cause a new lower-cost violation involving O_j. The only other possible new violation would have to involve O_i, but no such violation occurs since $c(O'') < c(O_i) \leq c(O'_{l+1})$. So again, the value of the penalty function is reduced.

Case 3: $c(O_i) > c(O'_{l+1})$ and there is a precedence relation between O_i and O'_{l+1}.

Define O'' as in Case 2. Moving O_i to the right of O'_l could only introduce a new violation of of **(B)** involving O'' and O'_{l+1}. But a violation does not occur since there can be no precedence relation in this case between O'' and O'_{l+1}: if there were, it would imply that O'' and O_i were both ancestors of O_{l+1} in the precedence graph, but had no precedence relation between them, which is impossible with tree-structured constraints.

As in Cases 1 and 2, the violation of **(A)** caused by O_i and O_j is eliminated, and there may be a new lower cost violation involving O_j. There is no new violation involving O_i, since there is a precedence relation between O_i and its new successor. Again, the penalty function is reduced.

□

4.2.1 *Case: Forest of Chains.* We use the above lemma to give an algorithm for selective and non-selective operators in the case that the precedence graph is a forest of chains (which includes the case where there are no precedence constraints).

The algorithm reduces the problem to three problems on forests of chains: one involving only selective operators, one involving only preserving operators, and one involving only expanding operators. Algorithms for the first and third of these problems were given above. The algorithm for the second is trivial. If an MTTC instance has only preserving operators, then one can simply order the operators in any order π that obeys the precedence constraints, and send r_{min} flow along it, where r_{min} is the minimum rate limit of an operator. This routing is clearly optimal.

The algorithm for forests of chains is as follows. Let I denote a problem instance such that the precedence graph is a forest of chains (or there are no precedence constraints).

- **Pre-processing Step:** If there is a parent-child pair, O_i, O_j, such that $c(O_i) > c(O_j)$, then replace the two operators with a new operator O_{ij} with selectivity $p_i p_j$ and rate limit $\min(r_i, r_j/p_i)$. From Lemma 4.1 (B), this does not preclude attainment of the optimal solution.
- Repeat until there are no such pairs. In the resulting problem, I', there are no pairs of operators O_i and O_j such that $c(O_i) > c(O_j)$ and O_i precedes O_j in a chain.
- Split the problem into three problems, I'_{sel}, I'_{pre}, and I'_{exp}, containing the selective, preserving, and expanding operators respectively. Each of these subproblems also contains the induced precedence constraints.
- From Lemma 4.1, we know that there exists an optimal solution such that in every permutation used, the selective operators ($\in I'_{sel}$) precede the preserving operators ($\in I'_{pre}$), which precede the expanding operators ($\in I'_{exp}$).
- Solve the three problems separately to obtain routings K'_{sel}, K'_{pre}, and K'_{exp}.
- Combine the three routings in a fashion similar to CombineRoutings (Section 3.5). Each resulting permutation will have all operators from I'_{sel} before the operators in I'_{pre}, and all operators from I'_{pre} before all operators from I'_{exp}.

The correctness follows from Lemma 4.1.

4.2.2 *Case: Arbitrary Tree-Structured Precedence Constraints.* Given the above algorithm for forests of chains, a natural approach to solving the problem for arbitrary tree-structured precedence constraints is to try and apply the procedure described in Section 3.5 to turn the algorithm for chains into an algorithm for trees.

There is one immediate difficulty that is easily overcome. AggregateChains computes a quantity whose denominator may be zero when the instance has a mixture of selective and non-selective operators. Use of this quantity can be avoided, at the expense of additional computation, by running RouteChains repeatedly on the chains emanating from the fork, as done in the example in Section 3.5.1. (More efficient fixes are also likely to exist.)

We conjecture that the resulting algorithm returns an optimal routing; however, we have been unable to prove this so far.

5. THE MAX-THROUGHPUT PROBLEM AND MINIMIZING TOTAL WORK ON A SINGLE PROCESSOR

In Section 2, we discussed the problem of finding an optimal sequential filter ordering for executing a pipelined filter ordering query on a single processor, when the goal is to minimize the *total work*. We will call this the *classical selection ordering problem*. Abstractly, the inputs to this problem are a list of n positive-valued costs, $c_1, \ldots, c_n$, and n positive-valued selectivities $p_1, \ldots, p_n$. Using the notation in Section 3, the problem is to find the permutation π of the operators that minimizes $\sum_{i=1}^{n} c_i g(\pi, i)$.

The classical selection ordering problem is related to the max-throughput problem through the dual of the max-throughput LP. Here we use duality to prove a general result about the complexity of finding exact solutions to the classical selection ordering problem, and to the max-throughput problem with selective queries. We show that under any class of precedence constraints, these two problems are polynomially equivalent.

Our proof relies on the ellipsoid algorithm, which solves linear programs in polynomial time [Khachiyan 1979; Bland et al. 1981]. The key requirement of the ellipsoid algorithm is existence of a polynomial-time *separation oracle* which, given a specific assignment to the LP variables, tells us whether the assignment satisfies the constraints of the LP, and if not, returns a constraint that is violated.

Recall that the MTTC problem is the max-throughput problem with precedence constraints, where the precedence graph is restricted to be a forest of rooted trees. More generally, for any class C of directed graphs, we can define the max-throughput problem with precedence constraints, where the precedence graph is restricted to be in C. We call this the Max-Throughput(C) problem. We require that the output to the Max-Throughput(C) problem be given explicitly, as a list of permutations receiving non-zero flow, together with the amount of flow assigned to each of those permutations. [3]

The classical selection ordering problem with precedence constraints, where the precedence graph is restricted to be from C, is defined in the analogous way. We call this the Classical-SO(C) problem.

We now prove the following theorem, which formally establishes the polynomial equivalence between max-throughput and classical selection ordering.

THEOREM 5.1. *Let C be a class of directed graphs. There is a polynomial-time algorithm for the Classical-SO(C) problem if and only if there is a polynomial-time algorithm for the Max-Throughput(C) problem.*

Proof. The LP defining the Max-Throughput(C) problem is the same as the LP given in Section 3.1 for the MTTC problem, except that G is in C.

The dual LP is as follows, and defines the dual problem Dual-Max-Throughput(C). Note that the inputs to both problems are just the r_i's, the p_i's, and G.

Dual-Max-Throughput(C) LP: Given $r_1, \ldots, r_n > 0$, $p_1 \ldots, p_n > 0$, and a precedence graph G on $\{O_1, \ldots, O_n\}$ from the class C, find an assignment to the variables y_i, for all $i \in \{1, \ldots, n\}$, minimizing

$$F = \sum_{i=1}^{n} r_i y_i$$

subject to the constraints:

(1) $\sum_{i=1}^{n} g(\pi, i) y_i \geq 1$ for all $\pi \in \phi_G(n)$ and

(2) $y_i \geq 0$ for all $i \in \{1, \ldots, n\}$.

We first show how a polynomial-time algorithm for the Classical-SO(C) problem can be used to construct a polynomial-time algorithm for the Max-Throughput(C) problem.

[3] Condon et al. [2009] actually presented two algorithms for solving the max-throughput problem without precedence constraints. One outputs an implicit representation of a routing that may route positive flow through an exponential number of different permutations. That algorithm does not meet the condition we impose here.

Suppose we have a polynomial-time algorithm for the former problem.

Consider the Dual-Max-Throughput(C) LP. This LP may contain up to $n!$ constraints (of type 1), and hence it would seem that it cannot be solved directly using the polynomial-time LP algorithms. However, this LP can indeed be solved in polynomial-time using the ellipsoid method, provided we have a polynomial-time separation oracle for it. We show how to implement this separation oracle in polynomial time. Given an assignment $y = (y_1, \ldots, y_n)$, the separation oracle needs to determine whether y obeys the constraints of the Dual-Max-Throughput(C) LP. If not, it must return a violated constraint. All constraints of type (1) have 1 on the right hand side. It therefore suffices to first find the permutation π^* obeying the constraints in G, such that $\sum_{i=1}^{n} g(\pi^*, i)y_i$ is minimized, and then to check whether $\sum_{i=1}^{n} g(\pi^*, i)y_i \geq 1$. If so, y is feasible, else we have found a violated constraint and can return it. If we view each y_i as a cost associated with O_i, then finding π^* is equivalent to solving the Classical-SO(C) problem, with costs y_i and selectivities p_i. Since we have assumed we can solve Classical-SO(C) in polynomial time, this means we can also solve the dual of the Max-Throughput(C) problem in polynomial time.

We now show how to use the ability to solve the dual problem to get a solution to the original Max-Throughput(C) problem. Using the ellipsoid algorithm, as just described, we first find a solution y^* to the dual LP. During our execution of the ellipsoid algorithm, we keep track of the violated constraints returned by the execution of our separation oracle. Let ϕ' be the set of permutations corresponding to these violated constraints. If these were the only constraints of type (1) in the dual LP, y^* would still be optimal (because our execution of the ellipsoid algorithm could proceed in the same way). It follows that in the original Max-Throughput(C) problem, we can eliminate all flow variables for permutations not in ϕ', yielding an LP for the original Max-Throughput(C) problem with only a polynomial number of variables. This LP can be written down explicitly and solved using a polynomial-time LP algorithm.

Conversely, assume there is an algorithm for solving the Max-Throughput(C) problem in polynomial time that outputs a polynomial-size routing. We use this to solve the Classical-SO(C) problem. This direction of the proof has three steps:

(1) Proving that a polynomial-time algorithm for solving the Max-Throughput(C) problem implies a polynomial-time algorithm for the Dual-Max-Throughput(C) problem.

(2) Proving that a polynomial-time algorithm for solving the Dual-Max-Throughput(C) problem implies that a separation oracle for the Dual-Max-Throughput(C) LP can be simulated in polynomial time.

(3) Proving that a polynomial-time simulation of a separation oracle for the Dual-Max-Throughput(C) LP implies a polynomial-time algorithm for the Classical-SO(C) problem.

Step (2) follows immediately from a general result which states (essentially) that a polynomial-time optimization oracle for a set of linear inequalities implies a polynomial-time separation oracle for that set (Corollary 14.1b in [Schrijver 1986]). Note that this is the reverse of the well-known implication (which we exploited above) that a polynomial-time separation oracle implies a polynomial-time optimization oracle.

For Step (1), consider an instance I^D of the Dual-Max-Throughput(C) problem. Let I be the associated Max-Throughput(C) instance. The Max-Throughput(C) LP associated with I may have degenerate optimal solutions. Using the perturbation technique of Megiddo and Chandrasekaran [1989], we could remove degeneracy from this LP by adding a polynomially small quantity ϵ_i (computable in polynomial time) to the right hand side of each of the rate constraints; this is equivalent to adding ϵ_i to each rate limit r_i. Call the modified MaxThroughput(C) instance I'. We run the algorithm for solving the Max-Throughput(C) problem on I'; since the algorithm runs in polynomial-time, the algorithm outputs a routing using at most a polynomial number of permutations. This routing corresponds to an optimal solution x^* for the Max-Throughput(C) LP corresponding to I'. Let $Ax \leq r'$ denote the rate constraints of this LP. We assume that the columns of A corresponding to the non-zero entries of x^* are linearly independent. (If not, the standard linear programming technique discussed in Section 3.6, for converting an optimal solution into a basic optimal solution, can be used to obtain an optimal solution with fewer non-zero entries, such that the corresponding columns of A are linearly independent. This can be done in time polynomial in n.) Let m be the number of non-zero entries of x^*.

Since the max-throughput instance is not degenerate, exactly m of its rate constraints are tight under solution x^*. Let D be the $m \times m$ submatrix of A, whose rows correspond to those m tight rate constraints, and whose columns correspond to the non-zero entries of x^*. It follows from complementary slackness and non-degeneracy that D is invertible, and that the unique optimal solution to the dual of the LP for I' can be obtained by finding the m-vector y' satisfying $D^T y' = \vec{1}$ (where $\vec{1}$ denotes the column vector of 1's) to obtain values for the variables of the dual corresponding to the m tight rate constraints, and setting all other variables of the dual to 0. Further, this solution is not only optimal for the dual of I', but also for the dual of the original I.

The proof of (3) is essentially a reduction from the decision version of a problem to a search (minimization) version of the problem. Consider an instance of the Classical-SO(C) problem. The problem is to find the permutation π of C that minimizes $\sum_{i=1}^{n} y_i g(\pi, i)$, where y is the given cost vector. The decision version of this problem is to determine, given a number k, whether there exists a permutation π in C such that $\sum_{i=1}^{n} y_i g(\pi, i) \leq k$. This decision problem can easily be solved by using a separation oracle for the Dual-Max-Throughput(C) LP to determine whether vector $z = (1/k)y$ is feasible, that is, whether $\sum_{i=1}^{n} z_i g(\pi, i) \geq 1$ for each permutation π in C. For any π, the value of the associated expected cost, $\sum_{i=1}^{n} y_i g(\pi, i)$ is upper bounded by $(\sum_{i=1}^{n} y_i)(\prod_{j \in N} p_j)$, where $N = \{j | p_j \geq 1\}$. For any two permutations, π' and π'', if the expected costs associated with these two permutations are not equal, then the difference between them is at least $d := \prod_{k \in S} p_j$, where $S = \{k | p_k < 1\}$. Let Q be the least multiple of d that is greater than $(\sum_{i=1}^{n} y_i)(\prod_{j \in N} p_j)$. To find the cost associated with the optimal permutation, it therefore suffices use the separation oracle to perform binary search on the set of values $\{0, d, 2d, 3d, \ldots, Q\}$, to find the integer j such that the expected cost associated with every permutation in C is greater than jd, but the same is not true of $(j+1)d$. During execution of this binary search, the separation oracle is queried with the vector $(1/((j+1)d))y$, and outputs a violated constraint. From this violated constraint, we can determine the associated permutation π, which has minimum expected cost. The number of calls made to the separation oracle in performing the binary search is polynomial in the number of bits representing the y_i and p_i. It follows that the running time of the above procedure is polynomial in the size of the Classical-SO(C) instance. □

Ibaraki and Kameda [1984] showed that the classical selection ordering problem can be solved in polynomial time in the presence of tree-structured precedence constraints. From the proof of the above theorem, we thus have an algorithm that uses the ellipsoid method to solve the max-throughput problem in polynomial-time in the presence of tree-structured precedence constraints. This proves the following corollary.

COROLLARY 5.1. *There is a polynomial-time algorithm for solving Max-Throughput(C), where C is the class of directed graphs corresponding to tree-structured precedence constraints.*

Since Max-Throughput(C) instances can contain both selective and non-selective operators, the algorithm used to prove the corollary can handle a broader class of problems than the algorithms we gave in Sections 3.3 and 4 (which can't handle arbitrary tree-structured constraints when there is a mixture of selective and non-selective operators). The advantage of those algorithms is that they are combinatorial, and more efficient.

Ibaraki and Kameda [1984] also showed that the classical selection ordering problem is NP-hard in the presence of arbitrary precedence constraints. We thus have the following corollary:

COROLLARY 5.2. *When C is the class of all directed graphs, Max-Throughput(C) is NP-hard.*

Acknowledgment

This work was supported by NSF Grants IIS-0546136, ITR-0205647, and CCF-0917153. We thank Collette Coullard for her invaluable help with the proof of Theorem 5.1, and Kamesh Munagala for illuminating discussions that led us to conjecture that theorem. We also thank the reviewers for their many excellent comments. Some of the results presented here appeared in preliminary form in a prior conference paper [Deshpande and Hellerstein 2008].

REFERENCES

AVNUR, R. AND HELLERSTEIN, J. M. 2000. Eddies: Continuously adaptive query processing. In *SIGMOD '00: Proceedings of the ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM, 261–272.

BABU, S., MOTWANI, R., MUNAGALA, K., NISHIZAWA, I., AND WIDOM, J. 2004. Adaptive ordering of pipelined stream filters. In *SIGMOD '04: Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM Press, 407–418.

BLAND, R. G., GOLDFARB, D., AND TODD, M. J. 1981. The ellipsoid method: A survey. *Operations Research 29(6)*, 1039–1091.

BURGE, J., MINGALA, K., AND SRIVASTAVA, U. 2005. Ordering pipelined query operators with precedence constraints. Tech. Rep. 2005-40, Stanford University.

CHAUDHURI, S., DAYAL, U., AND YAN, T. W. 1995. Join queries with external text sources: Execution and optimization techniques. In *SIGMOD '95: Proceedings of the ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*. ACM Press, 410–422.

CONDON, A., DESHPANDE, A., HELLERSTEIN, L., AND WU, N. 2006. Flow algorithms for two pipelined filter ordering problems. In *PODS '06: Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*. ACM, 193–202.

CONDON, A., DESHPANDE, A., HELLERSTEIN, L., AND WU, N. 2009. Algorithms for distributional and adversarial pipelined filter ordering problems. *ACM Transactions on Algorithms 5,* 2.

DANTZIG, G. AND THAPA, M. 2003. *Linear Programming 2: Theory and Extensions*. Springer.

DESHPANDE, A., GUESTRIN, C., HONG, W., AND MADDEN, S. 2005. Exploiting correlated attributes in acquisitional query processing. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering, 5-8 April 2005, Tokyo, Japan*. IEEE Computer Society, 143–154.

DESHPANDE, A. AND HELLERSTEIN, L. 2008. Flow algorithms for parallel query optimization. In *ICDE '08: Proceedings of the 24th International Conference on Data Engineering, April 7-12, 2008, Cancún, México*. IEEE, 754–763.

ETZIONI, O., HANKS, S., JIANG, T., KARP, R. M., MADANI, O., AND WAARTS, O. 1996. Efficient information gathering on the internet. In *FOCS '96: Proceedings of the 37th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 234–243.

FEIGE, U., LOVÁSZ, L., AND TETALI, P. 2004. Approximating min-sum set cover. *Algorithmica 40,* 4, 219–234.

GOLDMAN, R. AND WIDOM, J. 2000. WSQ/DSQ: A practical approach for combined querying of databases and the web. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM, 285–296.

IBARAKI, T. AND KAMEDA, T. 1984. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst. 9,* 3, 482–502.

KAPLAN, H., KUSHILEVITZ, E., AND MANSOUR, Y. 2005. Learning with attribute costs. In *STOC' 05: The Thirty-Seventh Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*. ACM Press, 356–365.

KHACHIYAN, L. G. 1979. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady 20(1)*, 191–194.

KODIALAM, M. 2001. The throughput of sequential testing. In *Proceedings of the Conference on Integer Programming and Combinatorial Optimization (IPCO), LNCS 2081*. Springer-Verlag, 280–292.

KRISHNAMURTHY, R., BORAL, H., AND ZANIOLO, C. 1986. Optimization of nonrecursive queries. In *VLDB '86: Proceedings of the Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan*. Morgan Kaufmann, 128–137.

LAWLER, E. L. 1978. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Math. 2*, 75–90.

LIU, Z., PARTHASARATHY, S., RANGANATHAN, A., AND YANG, H. 2008. A generic flow algorithm for shared filter ordering problems. In *PODS '08: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems, June 9-11, 2008, Vancouver, BC, Canada*. ACM, 79–88.

MEGIDDO, N. AND CHANDRASEKARAN, R. 1989. On the epsilon-perturbation method for avoiding degeneracy. *Operations Research Letters 8*, 305–308.

SCHRIJVER, A. 1986. *Theory of Linear and Integer Programming*. Wiley-Interscience.

SHAYMAN, M. AND FERNANDEZ-GAUCHERAND, E. 2001. Risk-sensitive decision-theoretic diagnosis. *IEEE Transactions on Automatic Control 46*, 1166–1171.

SIMON, H. AND KADANE, J. 1975. Optimal problem-solving search: All-or-none solutions. *Artificial Intelligence 6*, 235–247.

SRIVASTAVA, U., MUNAGALA, K., AND WIDOM, J. 2005. Operator placement for in-network stream query processing. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems, Baltimore, Maryland, USA, June 13-15, 2005*. ACM, 250–258.

SRIVASTAVA, U., MUNAGALA, K., WIDOM, J., AND MOTWANI, R. 2006. Query optimization over web services. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 355–366.

Received February 2010; revised September 2010; accepted November 2010