Algorithms for Distributional and Adversarial Pipelined Filter Ordering Problems *

Anne Condon[1], Amol Deshpande[2], Lisa Hellerstein[3], and Ning Wu[3]

(1) Department of Computer Science, University of British Columbia
condon@cs.ubc.ca

(2) Department of Computer Science, University of Maryland, College Park
amol@cs.umd.edu

(3) Department of Computer and Information Science, Polytechnic University
hstein,wning@cis.poly.edu

September 11, 2007

*Lisa Hellerstein and Ning Wu were supported by NSF grant ITR-0205647. Lisa Hellerstein was supported by the Othmer Institute for Interdisciplinary Studies. Amol Deshpande is supported by the NSF CAREER Award (Grant IIS-0546136). Anne Condon was supported by a grant from the Natural Sciences and Engineering Research Council of Canada and by the Peter Wall Institute for Advanced Studies.

Abstract

Pipelined filter ordering is a central problem in database query optimization. The problem is to determine the optimal order in which to apply a given set of commutative filters (predicates) to a set of elements (the tuples of a relation), so as to find, as efficiently as possible, the tuples that satisfy all of the filters. Optimization of pipelined filter ordering has recently received renewed attention in the context of environments such as the web, continuous high-speed data streams, and sensor networks. Pipelined filter ordering problems are also studied in areas such as fault detection and machine learning under names such as learning with attribute costs, minimum-sum set cover, and satisficing search. We present algorithms for two natural extensions of the classical pipelined filter ordering problem: (1) a *distributional type* problem where the filters run in parallel and the goal is to maximize throughput, and (2) an *adversarial type* problem where the goal is to minimize the expected value of *multiplicative regret*. We present two related algorithms for solving (1), both running in time $O(n^2)$, which improve on the $O(n^3 \log n)$ algorithm of Kodialam. We use techniques from our algorithms for (1) to obtain an algorithm for (2).

1 Introduction

Pipelined filter ordering is a central problem in database query optimization. The problem is to determine the optimal order in which to apply a given set of commutative filters (predicates) to a set of elements (the tuples of a relation), so as to find, as efficiently as possible, the tuples that satisfy all of the filters. Optimization of conjunctive selection queries reduces to pipelined filter ordering, as does optimization of certain commonly occuring join queries (specifically, those posed against a so-called *star* schema [23, 28]). Pipelined filter ordering problems are also studied in other domains such as fault detection and machine learning (see e.g. Shayman et al. [25] and Kaplan et al. [19]) under names such as learning with attribute costs [19], minimum-sum set cover [12], and satisficing search [26]. Since our interest in filter ordering was motivated by the problem of ordering database selection queries, we discuss our work in this context, although it can be interpreted more generally.

Recently, the problem of pipelined filter ordering has received renewed attention in the context of environments such as the web [6, 11, 16], continuous high-speed data streams [1, 3], and sensor networks [10]; query optimization in these environments presents significantly different challenges than those posed in traditional database systems. In this paper, we present efficient algorithms for two pipelined filter ordering problems. The first problem was studied previously in a different context by Kodialam [20]. We present two algorithms for solving this problem, both achieving better running time than Kodialam's. One algorithm outputs a sparse solution; the other algoithm does not, but we present it because it leads naturally to our methods for minimizing multiplicative regret. We use techniques from these two algorithms to obtain an algorithm for the second pipelined filter ordering problem. Some of the results in this paper appeared in an earlier conference version [9].

Pipelined filter ordering problems can be partitioned into two types. In the "distributional" type, assumptions are made about the probability that tuples will satisfy a given filter, and optimization is with respect to expected behavior. Probabilities may be learned from the statistics maintained on the tuples, or on-the-fly while tuples are being processed [1, 3]. In the "adversarial" type, the goal is to optimize with respect to worst-case assumptions, such as when an adversary controls which tuples satisfy which filters.

The classical pipelined filter ordering problem [18, 21] is a distributional type of problem. A cost and probability are given for each filter – the cost of applying the filter to a tuple, and the probability that the tuple satisfies the filter and is not eliminated. The event that a tuple satisfies a filter is independent of whether the tuple satisfies other filters. The problem is to find the ordering ("pipeline") of filters that yields minimum expected cost for processing a tuple. A simple polynomial-time solution is to order the filters in non-decreasing order of the ratio $c_i/(1-p_i)$, where p_i is the probability associated with filter i and c_i is the cost of applying filter i (cf. Garey [14], Ibaraki and Kameda [18], Krishnamurthy et al. [21], and Simon and Kadane [26]).

Other pipelined filter ordering problems, both distributional and adversarial, have been studied recently. We discuss this work in more detail in Section 2.

The first piplined filter ordering problem we consider is a distributional type problem in a parallel or distributed environment. The second is an adversarial type problem in a centralized environment.

• **Problem 1: Distributional type, parallel environment:** Our interest in this problem is motivated by two increasingly prevalent scenarios: (1) massively parallel database systems and (2) web-based structured information sources such as IMDB and Amazon. In both, selection queries (i.e. conjunctions of predicates, or filters) may be processed in parallel as follows. For each predicate of the query, there is a distinct operator (processor) dedicated to evaluating that predicate. Each tuple in the input relation is routed from operator to operator, until it is found to satisfy all predicates of the query and is output, or until it is found not to satisfy a predicate, in which case it is discarded. Each tuple can be routed individually, so that different tuples can have different routes. At any moment, each operator can evaluate its predicate on at most one tuple, and each tuple can be evaluated by at most one processor; but the n different operators can work in parallel on n different tuples. The problem, then, is to determine how best to route each tuple.

In solving this problem, we assume that the selectivity of each operator O_i, i.e. the probability p_i that a tuple satisfies O_i's predicate, is known, and that the selectivities lie strictly between 0 and 1 (that is, cannot be 0 or 1). The assumption that selectivities are not equal to 0 or 1 is justified, given our motivation, because if the outcome of applying a predicate to a tuple is known definitively in advance, there is no need to evaluate the predicate on the tuple. (Our solutions can easily be generalized to handle selectivities that are 0 and 1, but the exposition is significantly simplified by removing these cases.) We also assume that each operator O_i has a known rate limit r_i on the expected number of tuples it can process per unit time. This formulation is equivalent to one in which r_i is defined to be the maximum, rather than the expected, number of tuples that O_i can process in unit time, and excess tuples are queued; see the discussion of Kodialam's results in Section 2.

We assume that the event that a tuple satisfies a predicate is independent of whether the tuple satisfies any of the other predicates, and of the events that other tuples satisfy any predicates. Our goal is to route tuples so as to maximize the throughput of tuple handling, subject to the constraint that the expected number of tuples processed by each operator O_i per unit time does not exceed r_i. We call this the *max-throughput* problem.

Kodialam [20] gave algorithms that, given an instance of the max-throughput problem (i.e. the selectivities and rate limits of each of the n operators), find (a) the value of the maximum throughput, and (b) an optimal routing scheme. His algorithms run in time $O(n^2)$ and $O(n^3 \log n)$ respectively; they exploit the polymatroid structure of a certain space associated with the problem instance, and build on a constructive proof of the Caratheodory representation theorem. We present an algorithm for (a) that runs in linear time if the rate limits $\{r_i\}$ are given in sorted order and two algorithms for (b) that run in $O(n^2)$ time. Kodialam's algorithm for (b) outputs a sparse routing scheme, that is, a scheme which routes tuples along at most n distinct orderings of the operators. A sparse solution is desirable as it would would be easier to integrate it into a database query processor, and would result in lower per-tuple overhead in practice. The first of our algorithms for (b) outputs a sparse scheme, but the second does not. A straightforward version of the second algorithm routes tuples along a potentially exponential number of orderings; we also present a modification of the algorithm that reduces the number of orderings to be less than n^2, but still do not obtain a sparse solution, in general. Our main motivation for presenting the second algorithm is that it is based on a somewhat different technique than the first algorithm, which is useful in solving the adversarial type filter ordering problem addressed in the second part of this paper. Interestingly, there are special cases of the max throughput algorithm in which the second algorithm does output a sparse solution, but that solution is very different from the solution output by the first algorithm.

Our algorithms are conceptually simpler than Kodialam's; we use flow-based algorithms for (b) and our analysis of the algorithms provides the basis for our linear-time algorithm for (a).[1]

A naive strategy for routing the tuples would be to send them all along the route in which operators are ordered in decreasing order of their rate limits. A simple argument shows that this strategy maximizes throughput under the restriction that a common single ordering must be used to route all tuples [27]. We show that allowing individual tuples to be routed via different orderings can improve on the throughput achievable using the optimal single ordering, but the improvement is at most a factor of n, and this factor is tight.

• **Problem 2: Adversarial type, single tuple:** Our second result pertains to a new, adversarial type of problem. We focus on the problem of routing a single tuple through the operators, where a cost c_i is associated with each operator O_i. If a tuple is processed by operators $O_{i_1}, O_{i_2}, \ldots, O_{i_k}$ before being eliminated by O_{i_k}, then the total cost of processing the tuple is $c_{i_1} + \ldots + c_{i_k}$. Had the tuple been routed to O_{i_k} first, it would have incurred a cost of only c_{i_k}. The *multiplicative regret* is $\frac{c_{i_1}+\ldots+c_{i_k}}{c_{i_k}}$, the ratio of the actual cost incurred in processing the tuple, to the minimum possible cost that could be incurred under an optimal routing of that tuple.

The problem is to choose a (randomized) routing of the tuple so as to minimize the expected multiplicative regret, under the following assumptions. We assume that the set of filters which will eliminate the tuple is determined (in secret) by an adversary before a routing is chosen for the tuple. The goal of the adversary is to maximize the expected multiplicative regret induced by the tuple routing. The adversary (who may make random choices) will know the strategy used in determining the randomized routing of the tuple, and can choose the set of filters accordingly. We thus have a classical zero-sum game between two players – the routing player and the adversary – and the problem is to determine the optimal strategy of the routing player. We call this the *game theoretic multiplicative regret* (GTMR) problem. Our algorithm for the GTMR problem is based on the same flow techniques that we use for the max-throughput problem and runs in time $O(n^2)$. There is an algorithm that finds the value of the optimal solution to the GTMR problem in linear time, given the costs of the operators in sorted order. The proof and the algorithm are analogous to those presented for the max-throughput problem.

In what follows, we actually use an equivalent formulation of the GTMR problem, in which we restrict the adversary to choose exactly one filter to eliminate the tuple. The equivalence follows

[1]The conference version of this paper [9] had substantive errors in its presentation of the algorithm for (a), which we correct here. Also, the first algorithm for (b), achieving a sparse solution, did not appear in the conference version. The second algorithm for (b) did appear in the conference version, but this paper reduces the number of orderings along which tuples are routed from exponential to less than n^2.

from that fact that the restriction does not disadvantage the adversary. It is easy to show that it is not in the interest of the adversary to cause the tuple to satisfy all filters (because then the multiplicative regret is 1, which is the minimum possible), nor to choose more than one filter to eliminate a tuple (because if S is the set of filters that eliminate the tuple, removing all but the lowest cost filter in S can only increase multiplicative regret).

From a practical point of view, assumption of such a powerful adversary is not well motivated, since real-world data tends not to have worst-case properties. However, from a theoretical perspective, our analysis provides insight into worst-case behavior of pipelined filter ordering with costs. We note that the assumption of such an adversary is standard in on-line optimization problems, in which the goal is to minimize the competitive ratio (which is a type of multiplicative regret). The GTMR problem is not a proper on-line problem, however, since it takes only a single input, rather than a sequence of inputs.

A naive strategy for minimizing multiplicative regret routes the tuples through the operators in increasing order of their costs. As noted by Kaplan et al. [19], this strategy incurs a multiplicative regret of at most n. How much worse is this strategy than the optimal strategy returned by our GTMR algorithm? If all costs are equal, then the adversary will cause the optimal strategy to have (expected) multiplicative regret $(n+1)/2$, and the naive strategy to have multiplicative regret of n. We show that, for any set of costs, the naive strategy achieves multiplicative regret that is within a factor 2 of the expected multiplicative regret achieved by the optimal strategy. We also show that variants of the GTMR problem, in which the goal is to minimize additive regret or total cost, rather than multiplicative regret, have simple linear-time algorithms, assuming sorted input.

Following a discussion of related work in Section 2, we present our results on solving the max-throughput problem in Section 3. We present our first algorithm for the max-throughput problem, and its analysis, in Section 3.1. In Section 3.2 we present our linear-time algorithm for computing the optimal value of the maximum throughput. We present our second algorithm in Section 3.3, and its analysis in Section 3.4. In Section 3.5 we describe how to reduce the number of permutations used by the second algorithm to $O(n^2)$, and in Section 3.6 we compare the output of the two algorithms in the special case of equal rate limits. In Section 3.7 we show that the naive strategy for the max-throughput problem achieves a solution that is within a factor of n of optimal. We present our results on the game theoretic multiplicative regret problem and its variants in Section 4. In Section 4.1 we present the description of the algorithm for the GTMR problem, in Section 4.2 we analyze the performance of the naive strategy for solving the GTMR problem, and in Section 4.3 we give algorithms for solving versions of the game theory problem with other types of regret.

2 Related work

As discussed above, Kodialam [20] previously gave an algorithm for the max-throughput problem, but with higher running time than the algorithms given in this paper. He first introduces a problem variant that takes queueing delays into account. Note that our formulation of the max-throughput problem implicitly assumes that an operator O_i can sometimes process tuples at a rate that exceeds its limit r_i, since a solution only guarantees that the *expected* rate of tuples arriving at O_i will not exceed r_i. Kodialam's queueing-theory formulation imposes a limit on *maximum*, rather than expected, rates, with excess tuples at operators buffered in queues. Following early work of Coffman and Mitrani [7] and Gelenbe and Mitrani [15], Kodialam reduces the queueing-theory formulation to a problem that is equivalent to our formulation of the max-throughput problem. His reduction implies that if K is an optimal routing scheme for our formulation with max throughput F then, for any $F^* < F$, there is a routing scheme K^* for the queueing-theoretic formulation (where K^* is easily obtained by scaling K appropriately) with throughput F^*.

Several other variants of the pipelined filter ordering problem have been studied recently. One such problem is as follows: given a list L of tuples and for each, the subset of filters which it satisfies, and a cost for applying each filter, find the ordering π of the filters which minimizes the sum of the costs of evaluating all tuples in L using π. This problem is NP-hard, and significant effort has been invested in development of approximation algorithms [4, 8, 12, 22].

Other recent papers have addressed on-line variants of pipelined filter ordering [19, 22]. In these settings, tuples arrive one at a time, and the operators each have an associated cost. Tuples are processed sequentially. In the standard version of the on-line problem, the goal is to minimize the ratio, over the worst-case sequence of tuples, between the cost paid on that sequence, and the cost that would have been paid if all tuples in the sequence had been processed according to the single ordering π incurring minimum total cost on this sequence. This ratio is a type of multiplicative regret, but the regret is with respect to a sequence of tuples, rather than a single tuple.

Etzioni et al. [11] studied a web-query problem with some similarities to the max-throughput problem. There are m queries and n information sources. Consulting a source has a time cost and a dollar cost, and yields the answer to a query with a certain probability (independent of whether other sources provide the answer). Multiple sources can be consulted at the same time. The goal is to answer all m queries while minimizing the sum of the time and dollar cost. They provide an approximation algorithm for this problem.

Srivastava et al. [5, 27] recently studied yet other variations, motivated by query processing over web services. In their most general version [5], as in our problem, queries are handled by operators, each operator has an associated selectivity, operators can run in parallel, and the goal is to maximize the rate at which queries are handled. Their problem is more general than ours in one aspect, namely that there are precedence constraints on the order in which queries can be handled by processors, but less general in another, namely that all queries are handled in the same order. The authors give an efficient greedy algorithm to determine the optimal order in which queries can be handled by operators, and use network flow techniques at each step of the greedy algorithm to determine which operator to add next to the optimal order.

In *generalized maximum flow* problems, the amount of flow may change as it travels through a network (cf. Fleischer [13]). Although the flow problems studied in this paper also have this property, the requirement that flow pass through all operators (if not eliminated along the way) does not arise in generalized maximum flow problems. For some flow problems, decisions about flow routing can be made locally at nodes of the network, independently of other nodes [2]. An interesting question is whether there are efficient distributed local algorithms for the pipelined filter ordering problems of this paper. The methods of Plotkin et al. [24] and Grigoriadis [17] for finding approximate solutions for fractional packing problems may apply to our problems; however these methods do not provide exact solutions and cannot yield faster algorithms than we obtain.

3 The max-throughput problem

We first formally define the max-throughput problem via a linear program and present an example problem. We then describe our two algorithms for solving this problem.

We will frequently refer to permutations of sets and introduce our notation here. Let π be a permutation of a set S of size l. We represent π as a sequence $(s_1, \ldots, s_l)$ of the elements of S. For $k \in \{1, \ldots, l\}$ we use $\pi(k)$ to denote the kth element of the sequence, s_k. For $s \in S$, $\pi^{-1}(s)$

denotes the position of s in the sequence, that is, $\pi^{-1}(s) = i$ such that $s = s_i$. Suppose that π_1 and π_2 are permutations over disjoint sets S_1 and S_2. Then $\pi_1\pi_2$ denotes the permutation of $S_1 \cup S_2$ corresponding to the sequence formed by concatenating the sequences representing π_1 and π_2. We let $\pi^* = (n, n-1, \ldots, 1)$, since we need to refer to this permutation frequently. An instance of the max-throughput problem is a list of n *selectivities* (probabilities) $p_1, \ldots, p_n$, and n *rate limits* $r_1, \ldots, r_n$. The p_i are real values that lie strictly between 0 and 1, and the r_i are non-negative real values. Let $\phi(n)$ be the set of all $n!$ permutations of $\{1, \ldots, n\}$. For $i \in \{1, \ldots, n\}$, and permutation $\pi \in \phi(n)$, let $g(\pi, i)$ denote the probability that a tuple sent according to permutation π reaches selection operator O_i without being eliminated. Thus if $\pi^{-1}(i) = 1$ then $g(\pi, i) = 1$, and if $\pi^{-1}(i) > 1$ then $g(\pi, i) = p_{\pi(1)}p_{\pi(2)} \ldots p_{\pi(m-1)}$, where $m = \pi^{-1}(i)$. Define $n!$ real-valued variables f_π, one for each permutation $\pi \in \phi(n)$, where each f_π represents the number of tuples routed along permutation π per unit time. We call the f_π *flow variables*. The *max-throughput* problem is to find an optimal solution F, and corresponding assignment K to the flow variables, in the following linear program. We refer to the constraints of the first type in the linear program as *rate constraints*. If an assignment K satisfies the ith rate constraint with equality, we say that operator O_i is *saturated*.

Max-throughput LP: Given $r_1, \ldots, r_n > 0$ and $p_1 \ldots, p_n \in (0, 1)$, maximize

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

subject to the constraints

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

$$f_\pi \geq 0 \text{ for all } \pi \in \phi(n)$$

For example, let $n = 2$, $p_1 = p_2 = 1/2$, $r_1 = 2$, and $r_2 = 3$. If all tuples are sent to O_2 first and then to O_1, only 3 tuples per unit time can be processed. That is, if we set $f_{1,2} = 0$, then the maximum possible value of F is 3. Also, since $p_2 = 1/2$, this solution results in an expected rate of 3/2 tuples per unit time arriving at O_1, which is below the rate limit $r_1 = 2$ of O_1. A different routing allows more tuples to be processed, namely sending 8/3 tuples per unit time along route O_2, O_1, and 2/3 tuples per unit time along route O_1, O_2 (i.e. $f_{2,1} = 8/3$ and $f_{1,2} = 2/3$).

3.1 First algorithm to calculate an optimal routing for the max-throughput problem

3.1.1 Introduction to the algorithm

We begin by giving an informal introduction to our routing algorithm. We view the problem of routing tuples as one of constructing a flow through the operators. The capacity of each operator is its rate limit, and the amount of flow sent along a path through the operators is equal to the number of tuples sent along that path per unit time. We treat an operator having selectivity p as outputing exactly p times the amount of flow into it, although this is actually the *expected* flow output. However, our arguments apply also to expectation.

Consider first the special case in which there are two operators, O_1 and O_2, with selectivities p_1 and p_2, and rate limits r_1 and r_2, such that $r_1 p_1 = r_2$. For example, let $r_1 = 2, r_2 = 1$, and $p_1 = 1/2$. In this special case, it makes sense to route all of the flow through O_1 first. Specifically, if r_1 units of flow are routed through O_1 then O_1 is saturated; also this causes $r_2 = r_1 p_1$ units of flow to be routed through O_2, so that O_2 is also saturated. As we will show below (Corollary 3.1), any routing that saturates all of the operators is guaranteed to be optimal. In contrast, any solution that routes some flow through O_2 first will fail to saturate O_1, and can be shown not be optimal.

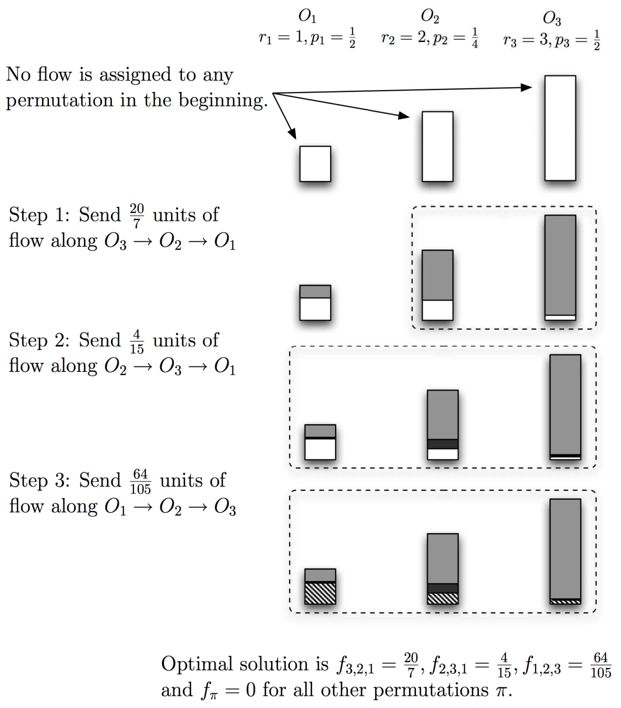
Now consider the general case where there are n operators. We construct a flow incrementally. Order the operators from n to 1 in decreasing order of their rate limits. Imagine pushing flow through the operators according to permutation $\pi^* = (n, n-1, \ldots, 1)$ that is, in the order $O_n, \ldots, O_1$. (Intuitively, it makes sense to send flow along this permutation, since tuples are eliminated as they pass through operators, and we'd like to eliminate as many tuples as we can before the tuple flow reaches the operators with low rate limits.) Suppose we continuously increase the amount of flow being pushed, beginning from zero, while monitoring the "residual capacity" of each operator, i.e., the difference between its rate limit and its load (the current rate of tuples arriving at that operator). Consider two adjacent operators, O_{i+1} and O_i. Initially, at zero flow, the residual capacity of O_{i+1} is greater than the residual capacity of O_i. As we increase the amount of flow, the residual capacity of each operator decreases continuously, with the residual capacity of O_{i+1} decreasing at a faster rate than that of O_i. We stop increasing the flow when one of the following stopping conditions is satisfied: (1) for some $i, 1 \leq i \leq n$, O_i becomes saturated, or (2) for some $i, 1 \leq i < n$, the residual capacity of O_i times its selectivity p_i becomes equal to the residual capacity of O_{i+1}. (Algorithmically, we do not increase the flow continuously, but instead directly calculate the amount of flow which triggers the stopping condition.)

We show that if stopping condition (1) above holds when the flow increase is stopped, the constructed flow is optimal. If stopping condition (2) holds for some i when the flow increase is stopped, our special case above suggests that we should order O_{i+1} immediately after O_i in routing any additional flow; in this way, if one becomes saturated, then the other will become saturated at exactly the same time. Thus, we keep the current flow, then replace O_i and O_{i+1} by a single "mega-operator" $O_{i,i+1}$ with rate limit equal to the residual capacity of O_i and selectivity equal to the product $p_i p_{i+1}$. We can solve the resulting smaller problem recursively (no resorting of operators is needed for the recursive call). Any flow that is routed through $O_{i,i+1}$ in the solution to the smaller problem is actually routed though O_i, immediately followed by O_{i+1}, in the final solution.

3.1.2 Example of first max-throughput algorithm

Suppose we have 3 operators, O_3, O_2, O_1 with rate limits $r_3 = 3, r_2 = 2$, and $r_1 = 1$, and selectivities $p_1 = p_3 = 1/2$ and $p_2 = 1/4$ (see Figure 1). If we send an amount t of flow along permutation O_3, O_2, O_1, then 1/2 of it will reach O_2 and 1/8 will reach O_1. Stopping condition (2) becomes true for O_2 when $(3 - t) = (2 - t/2)/4$, that is, when $t = 20/7$. Stopping condition (2) becomes true for O_3 when $(2 - t/2) = (1 - t/8)/2$, that is, when $t = 24/7$. Since 20/7 is the smaller of these quantities, and no operator becomes saturated while the flow is increased from 0 to 20/7, the stopping condition is reached at 20/7 tupes per unit time. Thus for our initial flow, we send 20/7 units along permutation O_3, O_2, O_1, causing the operators to have residual capacities 1/7 $(= 3 - 20/7)$, 4/7 $(= 2 - 20/14)$, and 9/14 $(= 1 - 20/56)$.

To augment this flow, we have a second stage, where we recursively solve the problem in which O_3 and O_2 are merged to form a mega-operator $O_{2,3}$ with selectivity 1/8 (that is, the product of the



O_1
$r_1 = 1, p_1 = \frac{1}{2}$
O_2
$r_2 = 2, p_2 = \frac{1}{4}$
O_3
$r_3 = 3, p_3 = \frac{1}{2}$
No flow is assigned to any permutation in the beginning.
Step 1: Send $\frac{20}{7}$ units of flow along $O_3 \rightarrow O_2 \rightarrow O_1$
Step 2: Send $\frac{4}{15}$ units of flow along $O_2 \rightarrow O_3 \rightarrow O_1$
Step 3: Send $\frac{64}{105}$ units of flow along $O_1 \rightarrow O_2 \rightarrow O_3$
Optimal solution is $f_{3,2,1} = \frac{20}{7}, f_{2,3,1} = \frac{4}{15}, f_{1,2,3} = \frac{64}{105}$ and $f_\pi = 0$ for all other permutations π.

Figure 1: **An example illustrating the first max-throughput algorithm.**

selectivities of O_2 and O_3) and rate limit (capacity) 4/7 (the rate residual capacity of O_2). Operator O_1 is included with capacity 9/14 and selectivity 1/2. Now, consider sending an increasing amount t of flow along permutation $O_{2,3}$, O_1. Stopping condition (2) holds when $(4/7-t) = (9/14-t/8)/2$, that is, when $t = 4/15$ and again, this flow can be achieved before either operator is saturated. We therefore augment the initial flow with 4/15 units sent along permutation $O_{2,3}, O_1$; translated, this means that 4/15 units are routed along O_2, O_3, O_1. The residual capacity of $O_{2,3}$ is then $4/7 - 4/15 = 32/105$ and the residual capacity of O_1 is $9/14 - 1/30 = 64/105$.

In the third and final stage, we solve the max-throughput problem with a single mega-operator $O_{1,2,3}$, which has residual capacity 64/105. This mega-operator becomes saturated when the flow is 64/105; this translates to a flow of 64/105 routed through O_1, O_2, O_3.

Together, the flows constructed in the above three stages yield the following optimal solution to the max-throughput LP for the given input instance: $f_{3,2,1} = 20/7$, $f_{2,3,1} = 4/15$, $f_{1,2,3} = 64/105$, and $f_\pi = 0$ for all other permutations π. This flow saturates all operators, although in general, this may not be the case.

3.1.3 Algorithm description

Algorithm 1 is a recursive algorithm that takes as input the selectivities $p_1, \ldots, p_n$ and rate limits $r_1, \ldots, r_n$, of operators $O_1, \ldots, O_n$, such that each p_i is a real number with $0 < p_i < 1$, each rate limit r_i is a non-negative real, and $r_i p_i \leq r_{i+1}$ for all i, $1 \leq i \leq n-1$. The algorithm constructs an assignment, represented as a set $K = \{(\pi, f_\pi) \mid f_\pi > 0\}$, to the flow variables of the corresponding max-throughput LP.

Before the initial call to the algorithm, the condition $r_i p_i \leq r_{i+1}$ for all i, $1 \leq i \leq n-1$, can be satisfied by numbering the operators (in advance) in decreasing order of their rate limits, that is, so $r_i \leq r_{i+1}$ for all i, $1 \leq i \leq n-1$, since $r_i \leq r_{i+1}$ implies $r_i p_i \leq r_{i+1}$.

The algorithm works as follows. It initializes K to be the empty set. For each operator O_i, it determines s_i, the smallest (non-negative) amount of flow through the operators in the order specified by $\pi^* = (n, \ldots, 1)$ which would cause either stopping condition (1) or (2) to hold for i. An amount s_i of flow, sent through the operators in the order π^*, causes stopping condition (1) to hold for i if O_i becomes exactly saturated by that flow. More precisely, since sending s_i amount of flow in the order π^* results in $s_i g(\pi^*, i)$ amount of flow reaching operator O_i, s_i satisfies stopping condition (1) when $s_i g(\pi^*, i) = r_i$, or equivalently when

$$s_i = r_i / g(\pi^*, i). \tag{1}$$

For $i, 1 \leq i \leq n-1$, s_i satisfies stopping condition (2) for i when $r_{i+1} - s_i g(\pi^*, i+1) = (r_i - s_i g(\pi^*, i)) p_i$, or equivalently when

$$s_i = \frac{r_{i+1} - r_i p_i}{g(\pi^*, i+1) - g(\pi^*, i) p_i}. \tag{2}$$

Thus, when $1 \leq i < n$, we set s_i to be the minimum of quantities on the right hand side of Equalities (1) and (2), and we set s_n to be the quantity on the right hand side of Equality (1). All quantities are non-negative, given our preconditions on the inputs.

The algorithm then determines the index, x, of the first operator for which a stopping condition would hold if the flow were to be increased from 0 (breaking ties arbitrarily). Namely, $x \leftarrow \text{argmin}_i \{s_i \mid 1 \leq i \leq n\}$.

If the flow s_x is greater than 0, then amount of flow s_x is routed through the operators in the order given by permutation π^*, i.e. (π^*, s_x) is added to K.

Then, if s_x satisfies stopping condition (2) for x (note that s_x may be 0), the algorithm subtracts $s_x g(\pi^*, i)$ from the rate limit of each operator $O_i \neq O_{x+1}$, to obtain new rate limits r'_i for these operators, sets the new selectivity of each O_i, $i \neq x, x+1$ to be $p'_i = p_i$, and sets the new selectivity of O_x to be $p_x p_{x+1}$. The algorithm deletes O_{x+1}, so O_x effectively becomes a mega-operator, which is the merge of O_x and O_{x+1}. Then, the problem is solved recursively with the resulting $n-1$ operators, using the new rate limits and selectivities.

Finally, routes are added to the solution K in the following way. First, the solution K'' output by the recursive call is adjusted, to renumber the operator indices and then to insert $x+1$ right after x in each permutation. For example, if $n = 5$ and $x = 2$, then operator 3 is removed in the recursive call, leaving operators $1, 2, 4$ and 5. However, a permutation π'' in the solution K'' refers to these, in order, as $1, 2, 3$ and 4. Permutation π' renames 3 and 4 back to 4 and 5. Thus, if $\pi'' = (3, 1, 2, 4)$ then $\pi' = (4, 1, 2, 5)$. Operator 3 is inserted in π' just after operator 2, to yield permutation $\pi^+ = (4, 1, 2, 3, 5)$. The flow $f_{\pi''}$ assigned to permutation π'' in solution K'' is now assigned to permutation π^+, and $(\pi^+, f_{\pi''})$ is added to K.

Each recursive call can be completed in $O(n)$ time; the recursion depth is at most n, and so the total running time is $O(n^2)$. Each recursive call adds at most one additional permutation to the solution. Hence the number of permutations in the solution is at most n, and so the solution is sparse.

3.1.4 Correctness

Lemma 3.1 *If inputs $p_1, \ldots, p_n, r_1, \ldots, r_n$ to Algorithm SolveMaxThroughput1 (Algorithm 1) satisfy the preconditions, and a recursive call to SolveMaxThroughput1 is made, then the preconditions also hold for the inputs to the recursive call.*

```
algorithm SolveMaxThroughput1($p_1, \ldots, p_n, r_1, \ldots, r_n$)
 input:          $n$ selectivities $p_1, \ldots, p_n$; $n$ rate limits $r_1, \ldots, r_n$
 preconditions:  $n \geq 1$, $r_i \geq 0$ and $0 < p_i < 1$ for all $i, 1 \leq i \leq n$
                 $r_i p_i \leq r_{i+1}$ for all $i, 1 \leq i \leq n-1$
 output:         optimal solution $K$ to max-throughput problem
                 for the given input parameters

 $\pi^* \leftarrow (n, n-1, \ldots, 1)$;
 for (each $i \in \{1, \ldots, n-1\}$)
 {
   // determine the smallest value at which a stopping condition holds for $O_i$
   $s_i \leftarrow \min\{r_i / g(\pi^*, i), \frac{r_{i+1} - r_i p_i}{g(\pi^*, i+1) - g(\pi^*, i) p_i}\}$
 }
 $s_n \leftarrow r_n / g(\pi^*, n)$;
 $x \leftarrow \text{argmin}_i \{s_i \mid 1 \leq i \leq n\}$; // break ties arbitrarily

 if ($s_x \neq 0$) { $K \leftarrow \{(\pi^*, s_x)\}$ } else { $K \leftarrow$ empty set }
 if ($x < n$ and $s_x = \frac{r_{x+1} - r_x p_x}{g(\pi^*, x+1) - g(\pi^*, x) p_x}$) // $s_x$ satisfies stopping condition (2) for $x$
 {
   // update flows and selectivities and solve subproblem recursively
   for (each $i \in \{1, \ldots, x, x+2, \ldots, n\}$) { $r'_i \leftarrow r_i - s_x g(\pi^*, i)$ }
   for (each $i \in \{1, \ldots, x-1, x+2, \ldots, n\}$) { $p'_i \leftarrow p_i$ }; $p'_x \leftarrow p_x p_{x+1}$;
   $K'' \leftarrow$ SolveMaxThroughput1($p'_1, \ldots, p'_x, p'_{x+2} \ldots, p'_n, r'_1, \ldots, r'_x, r'_{x+2}, \ldots, r'_n$);

   // renumber entries in permutations of $K''$
   $K' \leftarrow \{(\pi', f_{\pi'}) \mid (\pi'', f_{\pi''}) \in K''\}$ where $f_{\pi'} = f_{\pi''}$ and $\pi'$ is given by
         $\pi'(i) = \begin{cases} \pi''(i), & \text{if } \pi''(i) \leq x \\ \pi''(i) + 1, & \text{if } \pi''(i) > x \end{cases}$

   // insert $x+1$ to obtain $K$
   $K \leftarrow K \cup \{(\pi^+, f_{\pi^+}) \mid (\pi', f_{\pi'}) \in K'\}$ where $f_{\pi^+} = f_{\pi'}$ and $\pi^+$ is given by
         $\pi^+(i) = \begin{cases} \pi'(i), & \text{if } i \leq (\pi')^{-1}(x) \\ x+1, & \text{if } i = (\pi')^{-1}(x) + 1 \\ \pi'(i-1), & \text{if } i > (\pi')^{-1}(x) + 1 \end{cases}$
 }
 return $K$
```

Algorithm 1: Pseudocode of first algorithm for calculating an optimal routing for the max-throughput problem.

Proof. If a recursive call is made, then n must be at least 2, and so in the recursive call, the number of operators, $n-1$, must be at least 1. Each selectivity parameter, p_i, satisfies the precondition that it lies strictly between 0 and 1. Since each selectivity parameter, p'_i, in the recursive call is either p_i or, when $i = x$, the product $p_x p_{x+1}$, the p'_i also must lie strictly between 0 and 1.

We next consider the preconditions on the rate limits. Since for any i, $1 \le i \le n$, $g(\pi^*, i) > 0$, the quantity $r_i - sg(\pi^*, i)$ is continuously decreasing as s increases from 0. Thus, we have that

$$r_i - s_x g(\pi^*, i) \ge r_i - s_i g(\pi^*, i) \ge 0, \tag{3}$$

where the first inequality follows since $s_x \le s_i$ and the second by our choice of s_i.

Similarly, since for any i, $1 \le i < n$, $g(\pi^*, i+1) > g(\pi^*, i)$ (by the precondition that all selectivities lie strictly between 0 and 1), the quantity $(r_{i+1} - sg(\pi^*, i+1)) - (r_i - sg(\pi^*, i))p_i$ is non-negative when $s = 0$, and also is continuously decreasing as s increases from 0. Thus, we also have that

$$(r_{i+1} - s_x g(\pi^*, i+1)) - (r_i - s_x g(\pi^*, i))p_i \ge (r_{i+1} - s_i g(\pi^*, i+1)) - (r_i - s_i g(\pi^*, i))p_i \ge 0,$$

and so

$$(r_{i+1} - s_x g(\pi^*, i+1)) \ge (r_i - s_x g(\pi^*, i))p_i. \tag{4}$$

Now, to see that $r'_i \ge 0$ for any $i, 1 \le i \le n, i \ne x+1$, note that

$$r'_i = r_i - s_x g(\pi^*, i) \ge 0, \text{ (by Inequality (3))}. \tag{5}$$

To see that $r'_i p'_i \le r'_{i+1}$ for any $i, 1 \le i \le n$ with $i \ne x$ and $i \ne x+1$, note that if $i \ne x$,

$$\begin{aligned} r'_i p'_i &= (r_i - s_x g(\pi^*, i))p_i \\ &\le r_{i+1} - s_x g(\pi^*, i+1) \text{ (by Inequality (4))} \\ &= r'_{i+1}. \end{aligned}$$

Finally, to see that $r'_x p'_x \le r'_{x+2}$ in the case that $x + 2 \le n$, note that

$$\begin{aligned} r'_x p'_x &= (r_x - s_x g(\pi^*, x))p_x p_{x+1} \\ &\le (r_{x+1} - s_x g(\pi^*, x+1))p_{x+1} \text{ (by Inequality (4))} \\ &\le r_{x+2} - s_x g(\pi^*, x+2) \text{ (again, by Inequality (4))} \\ &= r'_{x+2}. \end{aligned}$$

□

Lemma 3.2 *On any input $(p_1, \ldots, p_n, r_1, \ldots, r_n)$ satisfying the preconditions, the solution K constructed by Algorithm 1 is a feasible solution of the max-throughput LP.*

Proof. The proof is a very straightforward induction on n, the number of operators. In the base case, when $n = 1$, there is only one permutation, namely π^*, where $f_{\pi^*} = r_1 \ge 0$, and it can easily be seen that this flow assignment satisfies the flow constaint.

For the induction step, let $n > 1$. First note that the flow assigned to permutation π^* must be non-negative by construction. Also, if a recursive call is made then by Lemma 3.1 the preconditions must hold in the recursive call and so by induction the flow assigned to any permutation in any K'' must be non-negative. Since the values of flows in K are either flows in K'' or the flow to π^*, the LP constraints that $f_\pi \geq 0$ are satisfied by K.

It remains to show that the rate constraints of the LP are satisfied. By construction, for each $(\pi'', f_{\pi''}) \in K''$, there is a corresponding flow assignment (π^+, f_{π^+}) in K, where $f_{\pi''} = f_{\pi^+}$ and π^+ is obtained from π'' by first renumbering, to obtain π', and then inserting $x+1$ after x in π'. Rather than reasoning about solution K'', we reason instead using K', since numbering of operators in K' agrees with numbering in K.

Let $g'(\pi', i)$ denote the probability that a tuple, sent according to permutation π', reaches selection operator O_i without being eliminated, for the instance of the max-throughput problem in the recursive call, namely when the selectivities are $p'_1, \ldots, p'_x, p'_{x+2}, \ldots, p'_n$. Thus if $(\pi')^{-1}(i) = 1$ then $g(\pi', i) = 1$, and if $(\pi')^{-1}(i) = m > 1$ then $g(\pi', i) = p_{\pi'(1)} p_{\pi'(2)} \cdots p_{\pi'(m-1)}$. By the induction hypothesis, the solution K' satisfies the rate constraints for the subproblem of the recursive call, and so for $i \in \{1, \ldots, x, x+2, \ldots, n\}$,

$$\sum_{(\pi', f_{\pi'}) \in K'} f_{\pi'} g'(\pi', i) \leq r'_i. \tag{6}$$

Also, for all $i \in \{1, \ldots, x, x+2, \ldots, n\}$,

$$g'(\pi', i) = g(\pi^+, i), \tag{7}$$

by the definition of selectivities $p'_i, i \neq x+1$. Therefore, for $i \in \{1, \ldots, x, x+2, \ldots, n\}$ we have that

$$\begin{aligned}
\sum_{\pi \in \phi(n)} f_\pi g(\pi, i) &= \sum_{(\pi^+, f_{\pi^+}) \in K} f_{\pi^+} g(\pi^+, i) \\
&= \sum_{(\pi', f_{\pi'}) \in K'} f_{\pi'} g'(\pi', i) + s_x g(\pi^*, i) \\
&\qquad \text{(by Equation (7) and construction of } K \text{ from } K'\text{)} \\
&\leq r'_i + s_x g(\pi^*, i) \text{ (by Inequality (6))} \\
&= r_i - s_x g(\pi^*, i) + s_x g(\pi^*, i) \text{ (by the definition of } r'_i\text{)} \\
&= r_i. \qquad (8)
\end{aligned}$$

Also,

$$\begin{aligned}
\sum_{\pi \in \phi(n)} f_\pi g(\pi, x+1) &= \sum_{(\pi^+, f_{\pi^+}) \in K} f_{\pi^+} g(\pi^+, x+1) \\
&= \sum_{(\pi', f_{\pi'}) \in K'} f_{\pi'} g'(\pi', x) p_x + s_x g(\pi^*, x+1) \\
&\qquad \text{(by Equation (7) and construction of } K \text{ from } K'\text{)} \\
&\leq r'_x p_x + s_x g(\pi^*, x+1) \text{ (by Inequality (6))} \\
&= (r_x - s_x g(\pi^*, x)) p_x + s_x g(\pi^*, x+1) \text{ (by the definition of } r'_x\text{)} \\
&= r_{x+1} - s_x g(\pi^*, x+1) + s_x g(\pi^*, x+1) \qquad (9) \\
&\qquad \text{(since } s_x \text{ satisfies stopping condition (2) for } x\text{)} \\
&= r_{x+1}. \qquad (10)
\end{aligned}$$

□

Let K be a feasible solution to the max-throughput LP with n operators. We say that K has the *saturated-suffix* property if for some non-empty "witness" subset $Q \subseteq \{1 \ldots n\}$, (1) O_i is saturated, i.e. $\sum_{\pi \in \phi(n)} f_\pi g(\pi, i) = r_i$, if and only if $i \in Q$, and (2) for any flow variable f_π, $f_\pi > 0$ implies that in permutation π, the elements of $\bar{Q} = \{1 \ldots n\} \backslash Q$ precede the elements of Q.

Next, we show that the solution constructed by our first max-throughput algorithm (Algorithm 1) has the saturated-suffix property.

Lemma 3.3 *If K is the output of the SolveMaxThroughput1 algorithm (Algorithm 1), then K has the saturated-suffix property with witness $Q = \{1, \ldots, q\}$, for some q where $1 \leq q \leq n$.*

Proof. The proof is by induction on n. When $n = 1$, $Q = \{1\}$ trivially satisfies the conditions of the saturated-suffix property.

Let $n > 1$ and suppose the lemma is true for instances with $n - 1$ operators. We first consider part (1) of the saturated-suffix property. If no recursive call is made, then K is either the empty set or $K = \{(\pi^*, s_x)\}$. If K is the empty set, then at least one operator must have rate limit equal to 0, and by the preconditions on the input, the set of operators with rate limit equal to 0 must be numbered from 1 to q for some q. If $K = \{(\pi^*, s_x)\}$, let q be the highest index of an operator that is saturated by K. We claim that all operators with indices in $\{1, \ldots, q\}$ are saturated. If not, let $j < q$ be the largest index such that O_j is not saturated. Since O_{j+1} is saturated, we have that

$$r_{j+1} - s_x g(\pi^*, j + 1) = 0. \tag{11}$$

By Inequality (4), it must also be that

$$r_{j+1} - s_x g(\pi^*, j + 1) \geq (r_j - s_x g(\pi^*, j))p_j. \tag{12}$$

But then Equation (11) and Inequality (12) together imply that $r_j - s_x g(\pi^*, j)p_j \leq 0$, contradicting the assumption that O_j is not saturated.

Next we consider part (1) of the saturated-suffix property if a recursive call is made. By the induction hypothesis, the solution K' satisfies the saturated-suffix property. Let Q' be the set of operators which are saturated by solution K', with respect to rates r'_i. Thus, for each $i \in Q'$,

$$\sum_{(\pi', f_{\pi'}) \in K'} f_{\pi'} g'(\pi', i) = r'_i. \tag{13}$$

Let $Q = Q'$ if $x \notin Q'$ and let $Q = Q' \cup \{x + 1\}$ otherwise. We claim that Q witnesses the fact that K satisfies the saturated-suffix property. In the derivation leading to Equation (8), the single inequality can be replaced by an equality if $i \in Q'$, by (13) above, showing that

$$\sum_{\pi \in \phi(n)} f_\pi g(\pi, i) = r_i. \tag{14}$$

Similarly, in the derivation leading to Equation (10), the inequality can be replaced by an equality if and only if O_x is saturated, and thus O_{x+1} is saturated if and only if O_x is. Finally, no operator in $\{1, \ldots, x, x + 2, \ldots, n\} \backslash Q$ is saturated, because in the derivation leading to Equation (8), the single inequality can be replaced by a strict inequality (that is, "$\leq$" can be replaced by "$<$"), since the operators in $\{1, \ldots, x, x + 2, \ldots, n\} \backslash Q$ are not saturated in K'.

We now show that part (2) of the saturated-suffix property holds, namely that for any (π, f_π) in K, the elements of $\bar{Q} = \{1, \dots, n\} \backslash Q$ precede the elements of Q in π; we also show that $Q = \{1, \dots, q\}$ for some q. There are two types of permutations in K: the permutations π^+ derived from permutations in K' constructed in the recursive call, and the permutation π^*. By induction, the elements in $\bar{Q}' = \{1, \dots, x, x+2, \dots, n\} \backslash Q'$ precede the elements of Q' in each permutation of K'. Moreover, the elements of Q' are either $\{1, \dots, q\}$ for some $q \leq x$, or $\{1, \dots, x, x+2, \dots, q\}$ for some $q \geq x+2$. Since $x+1$ is adjacent to x in each permutation π^+ in K and the order of operators is otherwise unchanged, and since $x+1$ is in Q if and only if x is, the elements of $\bar{Q} = \{1, \dots, n\} \backslash Q$ precede the elements of Q in each permutation π^+. Moreover, $Q = \{1, \dots, q\}$ for some q. Finally, part (2) holds for the permutation π^* since π^* orders indices from highest to lowest. $\square$

The following lemma upper bounds the optimal value of the objective function of the max-throughput problem, in terms of an arbitrary subset Q of the operators.

Lemma 3.4 *Let F^* be the optimal value of the objective function in the max-throughput problem. Let $Q \subseteq \{1, \dots, n\}$. Then*

$$F^* \leq \frac{\sum_{i \in Q} r_i (1 - p_i)}{(\prod_{j \notin Q} p_j)(1 - \prod_{i \in Q} p_i)}.$$

Proof. Consider a modification of the max-throughput LP in which we remove all rate constraints for operators not in Q, i.e. we remove all constraints $\sum_{\pi \in \phi(n)} f_\pi g(\pi, i) \leq r_i$ where $i \notin Q$. Let F' be the optimal value of the objective function for this modified max-throughput LP. Clearly $F^* \leq F'$.

Consider an optimal solution K' to the modified LP. Let $K' = \{f'_\pi \mid \pi \in \phi(n)\}$. Let $\phi'(n)$ be the set of permutations $\pi \in \phi(n)$ such that the elements of $\bar{Q} = \{1 \dots n\} \backslash Q$ precede the elements of Q in π. Because operators O_j such that $j \notin Q$ have no rate constraints, and because each such operator has selectivity $p_j < 1$ and thus eliminates some of the amount of flow that passes through it, we may assume without loss of generality that if $f'_\pi > 0$, then $\pi \in \phi'(n)$.

Since K' satisfies the rate constraints for all $i \in Q$, it follows that for $i \in Q$, $\sum_{\pi \in \phi(n)} f'_\pi g(\pi, i) \leq r_i$. If we multiply both sides of this inequality by $(1 - p_i)$ and sum over all $i \in Q$, we get that

$$\sum_{i \in Q} \sum_{\pi \in \phi(n)} f'_\pi g(\pi, i)(1 - p_i) \leq \sum_{i \in Q} r_i (1 - p_i). \tag{15}$$

We now rewrite the left hand side of Equation (15). Exchanging summation order and bringing out f'_π shows it equals $\sum_{\pi \in \phi(n)} f'_\pi \sum_{i \in Q} g(\pi, i)(1 - p_i)$. By assumption, $f'_\pi = 0$ for all $\pi \notin \phi'(n)$. Let $\pi \in \phi'(n)$. Consider $\sum_{i \in Q} g(\pi, i)(1 - p_i)$. For all $i \in Q$, $g(\pi, i)$ equals $(\Pi_{j \notin Q} p_j)$ times the product of all p_m such that $m \in Q$ and $\pi(m) < \pi(i)$. Thus

$$\sum_{i \in Q} g(\pi, i)(1 - p_i) = (\Pi_{j \notin Q} p_j) \sum_{i \in Q} (\Pi_{m \in Q : \pi(m) < \pi(i)} \, p_m)(1 - p_i).$$

To simplify this expression, we use the fact that

$$\sum_{i \in Q} (\Pi_{m \in Q : \pi(m) < \pi(i)} \, p_m)(1 - p_i) = (1 - \Pi_{i \in Q} p_i).$$

Hence

$$\sum_{i \in Q} g(\pi, i)(1 - p_i) = (\prod_{j \notin Q} p_j)(1 - \prod_{i \in Q} p_i)$$

and it follows [2] that

$$\sum_{\pi\in\phi(n)} f'_\pi(\prod_{j\notin Q} p_j)(1-\prod_{i\in Q} p_i) \le \sum_{i\in Q} r_i(1-p_i).$$

Thus

$$F' = \sum_{\pi\in\phi(n)} f'_\pi \le \frac{\sum_{i\in Q} r_i(1-p_i)}{(\prod_{j\notin Q} p_j)(1-\prod_{i\in Q} p_i)}.$$

□

We are now ready to prove the following important lemma, which shows that any feasible solution that has the saturated-suffix property is optimal.

Lemma 3.5 *If feasible solution K to the max-throughput LP has the saturated-suffix property, and Q is the set of operators saturated by K, then K is an optimal solution to the max-throughput problem and the value F of the objective function achieved by K is*

$$\frac{\sum_{i\in Q} r_i(1-p_i)}{(\prod_{j\notin Q} p_j)(1-\prod_{i\in Q} p_i)}.$$

Proof. Consider the assignment to the f_π under solution K. We have that $\sum_{\pi\in\phi(n)} f_\pi g(\pi,i) = r_i$, for $i \in Q$. Multiplying both sides of this equation by $(1-p_i)$ and summing over all $i \in Q$ we get that

$$\sum_{i\in Q}\sum_{\pi\in\phi(n)} f_\pi g(\pi,i)(1-p_i) = \sum_{i\in Q} r_i(1-p_i). \tag{16}$$

This is analogous to Equation (15), except with equality instead of inequality. By the same arguments as in the proof of Lemma 3.4, it follows that

$$F = \sum_{\pi\in\phi(n)} f_\pi = \frac{\sum_{i\in Q} r_i(1-p_i)}{(\prod_{j\notin Q} p_j)(1-\prod_{i\in Q} p_i)}.$$

By Lemma 3.4, this value is at least as large as the optimal value of the max-throughput LP. Therefore, it is equal to the optimal value. □

We note the following corollary to Lemma 3.5, which follows immediately since any solution saturating all operators trivially satisfies the saturated-suffix property.

Corollary 3.1 *If a feasible solution K to the max-throughput LP saturates all operators, then K is optimal.*

Finally, we put the previous results together to show that the algorithm is correct.

Theorem 3.1 *Algorithm 1, the SolveMaxThroughput1 algorithm, finds an optimal solution K to the max-throughput LP*

[2]In fact, the correctness of the next equation can also be shown as follows. For $i \in Q$, the expected number of tuples processed by O_i per unit time is at most r_i, and the expected number eliminated is $r_i(1-p_i)$. Thus the two sides of the equation both express the total expected number of tuples per unit time eliminated by operators O_i where $i \in Q$.

Proof. Lemma 3.2 shows that the SolveMaxThroughput1 algorithm outputs a feasible solution to the max-throughput LP. Lemma 3.3 shows that it outputs a solution to the max-throughput LP that satisfies the saturated-suffix property. Lemma 3.5 shows that a feasible solution that satisfies the saturated-suffix property is an optimal solution. □

We remark that our algorithm can be adapted to output a succinct description of the routing scheme in $O(n \log n)$ time. Roughly, this is because a heap data structure can be used to manage the s_i values; the recursive call can be changed to an iterative scheme, and can avoid the need to update rate limits and all selectivities except for one per iteration; and the change in "current" permutation from one iteration to the next can be represented succinctly. Since the details complicate the exposition, and since $\Theta(n^2)$ time is needed in any case to write down the full routing scheme, we do not present the details here.

3.2 Algorithm to compute the value of the maximum throughput

A simple algorithm for computing the value of the maximum throughput can be easily derived from the following lemma:

Lemma 3.6 *The optimal value of the objective function for the max-throughput problem, when $r_1 \leq r_2, \ldots \leq r_n$, is the minimum of*

$$F_q = \frac{\sum_{i=1}^{q} r_i(1-p_i)}{(\prod_{j=q+1}^{n} p_j)(1-\prod_{i=1}^{q} p_i)}.$$

over all $q \in \{1, \ldots, n\}$.

Proof. By Lemma 3.4, each of the values F_i is an upper bound on the value of the maxmum throughput. By Lemmas 3.3 and 3.5, our max-throughput algorithm finds an optimal solution having the saturated-suffix property, with the saturated suffix being $Q = \{1, \ldots, q\}$ for some q. Further, since the value of that solution is F_q for some q, the optimal value of the maximum throughput is equal to F_q for some q. The lemma follows. □

The max-throughput problem with the rates $r_1, r_2, \ldots, r_n$ in sorted order can thus be solved by simply computing $F_1, F_2, \ldots F_n$ in turn, and outputing their minimum. Each value F_{q+1} can be calculated in constant time, given the numerator and denominator of F_q. Thus if the rate limits are given in sorted order, this algorithm finds the value of the maximum throughput in linear time.

3.3 A second algorithm for solving the max-throughput problem

We now present our second algorithm for solving the max-throughput problem. As mentioned previously, the approach used in this algorithm is also useful in our solution to the GTMR problem.

3.3.1 Introduction to the second max-throughput algorithm

Consider the special case of the max-throughput problem in which operators all have the same rate limit (capacity). For example, let O_1, O_2 be two operators with selectivities p_1 and p_2, and with equal rate limits r. If we send x units of flow along permutation $(1, 2)$, and y units along permutation $(2, 1)$, then O_1 receives $x + p_2 y$ units and O_2 receives $y + p_1 x$ units. If $x = r(1-p_2)/(2-p_1p_2)$ and $y = r(1-p_1)/(2-p_1p_2)$, then $x + p_2 y = y + p_1 x = r$ and both operators are saturated. Further, for any $0 \leq q \leq 1$, if we send qx units along permutation $(1, 2)$ and qy units along permutation $(2, 1)$, the residual capacities of O_1 and O_2 remain equal. Below we give a closed-form expression that

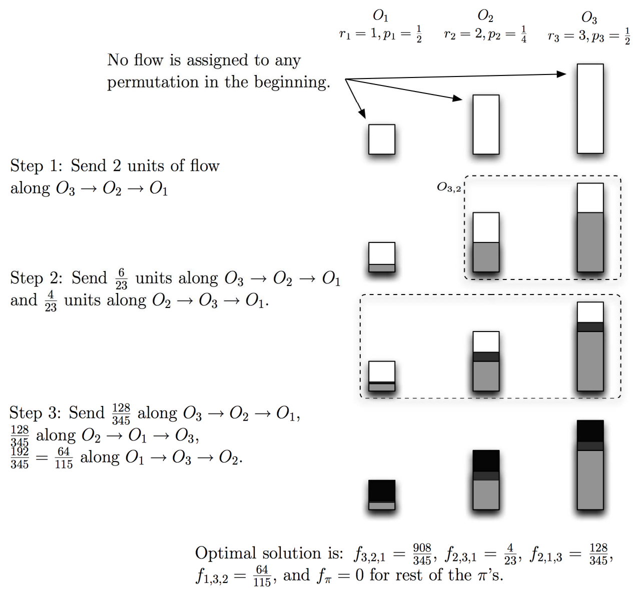

O_1
$r_1 = 1, p_1 = \frac{1}{2}$
O_2
$r_2 = 2, p_2 = \frac{1}{4}$
O_3
$r_3 = 3, p_3 = \frac{1}{2}$
No flow is assigned to any permutation in the beginning.
Step 1: Send 2 units of flow along $O_3 \rightarrow O_2 \rightarrow O_1$
$O_{3,2}$
Step 2: Send $\frac{6}{23}$ units along $O_3 \rightarrow O_2 \rightarrow O_1$ and $\frac{4}{23}$ units along $O_2 \rightarrow O_3 \rightarrow O_1$.
Step 3: Send $\frac{128}{345}$ along $O_3 \rightarrow O_2 \rightarrow O_1$, $\frac{128}{345}$ along $O_2 \rightarrow O_1 \rightarrow O_3$, $\frac{192}{345} = \frac{64}{115}$ along $O_1 \rightarrow O_3 \rightarrow O_2$.
Optimal solution is: $f_{3,2,1} = \frac{908}{345}$, $f_{2,3,1} = \frac{4}{23}$, $f_{2,1,3} = \frac{128}{345}$, $f_{1,3,2} = \frac{64}{115}$, and $f_\pi = 0$ for rest of the π's.

Figure 2: **An example illustrating the second max-throughput algorithm**

generalizes the above routing for the special case of $n > 2$ operators with equal rate limits; it gives a way to route flow so as to ensure that the operators continue to have equal residual capacity, if they start off with equal residual capacity.

We use the solution for the above special case as the basis for our second max-throughput algorithm. In the second algorithm, as in the first, we construct the flow routing in stages, recursively. In each recursive call, we partition the operators into equivalence classes according to their rate limits. Conceptually, we view each equivalence class as a mega-operator, with rate limit equal to the rate limit of its constituent operators. We order these mega-operators in decreasing order of the rate limits, and (conceptually) send a continuously increasing amount of flow through the mega-operators in this order. The twist is how we route flow within the mega-operator: when it reaches a mega-operator, we divide it as dictated by the solution to the special case, so as to preserve the property that the operators within a mega-operator have equal residual capacity. We continue increasing the flow amount until either (1) some operator becomes saturated or (2) the residual capacity of the operators in one mega-operator (equivalence class) becomes equal to the residual capacity of operators in another mega-operator. When one of the stopping conditions is reached, we add the current flow to the solution, and then recurse *on the original set of operators*, with rate limits equal to their residual capacities.

Note that the residual capacity of an operator in a mega-operator may decrease more slowly than it would if all flow were sent directly to that operator, because some flow may first be filtered through other operators in the mega-operator. This needs to be taken into account in determining when a stopping condition is reached. We discuss this in more detail below, but first give an example.

3.3.2 Example

Suppose we have 3 operators, O_3, O_2, O_1 with rate limits $r_3 = 3, r_2 = 2$, and $r_1 = 1$, and selectivities $p_1 = p_3 = 1/2$ and $p_2 = 1/4$ (see Figure 2). If we send flow along permutation O_3, O_2, O_1, then 1/2 of it will reach O_2 and 1/8 will reach O_1. Thus, O_3 and O_2 achieve equal residual capacity when 2 units of flow are sent, O_2 and O_1 achieve equal residual capacity when 8/3 units are sent, and the minimum amount of flow needed to saturate an operator is 3 units. Therefore, the stopping condition is reached at 2 units, and so for our initial flow, we send 2 units along permutation O_3, O_2, O_1, causing the operators to have residual capacities 1, 1, and 3/4.

To augment this flow, we have a second stage, where we solve the problem in which O_3, O_2 and O_1 have rate limits 1, 1, and 3/4 respectively. Replace O_3 and O_2 with a mega-operator $O_{3,2}$ having selectivity $1/2 * 1/4 = 1/8$. Consider sending flow along permutation $O_{3,2}$, O_1, dividing any flow into mega-operator $O_{3,2}$ so that 3/5 of it is sent along permutation O_3, O_2, and 2/5 along permutation O_2, O_3; this equalizes the load on O_3 and O_2. Under this division, t units of flow sent into $O_{3,2}$ decrease the capacity of O_2 and O_3 each by 7/10 t. Since the rate limits of O_2 and O_3 are 1, they therefore become saturated when 10/7 units are sent along $O_{3,2}$, O_1. Any flow sent along $O_{3,2}$, O_1 is reduced by a factor of $1/2 * 1/4 = 1/8$ before reaching O_1, so O_1 is saturated when 6 units of flow are sent along $O_{3,2}$, O_1. The residual capacities of the operators in $O_{3,2}$, and operator O_1 equalize at t units, where $1 - 7/10\ t = 3/4 - 1/8t$, that is, $t = 10/23$. Thus the stopping condition is reached at 10/23 units, when the residual capacities of the operators are equalized at 16/23. We therefore augment the initial flow with 10/23 units sent along permutation $O_{2,3}, O_1$; translated, this means 6/23 units along O_3, O_2, O_1, and 4/23 along O_2, O_3, O_1.

In the third and final stage, we solve the max-throughput problem in which operators O_3, O_2, O_1 have equal rate limits of 16/23. Using our closed-form expression (cf. Lemma 3.7) to equalize the load on the three operators, we divide the flow so 2/7 of it is sent along permutation O_3, O_2, O_1, 2/7 along permutation O_2, O_1, O_3, and 3/7 along permutation O_1, O_3, O_2. Under this division, 15/28 of the flow arrives at each operator. Thus sending $t = 448/345$ units saturates the operators, since $16/23 = 15/28t$. We augment the flow from the first two stages with 448/345 units of flow divided as just described.

Together, the flows constructed above yield the following optimal solution to the max-throughput LP for the given input instance: $f_{3,2,1} = 908/345$, $f_{2,3,1} = 4/23$, $f_{2,1,3} = 128/345$, $f_{1,3,2} = 64/115$, and $f_\pi = 0$ for all other permutations π. This flow saturates all operators, although in general, this may not be the case. Since the number of permutations used in routing the flow may be exponential in the number of operators, our algorithm outputs a compact representation of the flow, rather than giving the values of the non-zero f_π .

3.4 Correctness and further details of the second algorithm

The following lemma gives the closed-form expression for a routing that equalizes the load on n operators. This is used to route flow through the operators in a mega-operator.

Lemma 3.7 *Let ρ_1 be the permutation $1, \ldots, n$ and for $j \in [2 \ldots n]$, let ρ_j be permutation $j, j+1, \ldots, n, 1, 2, \ldots, j-1$, that is, the permutation obtained by performing $j-1$ left cyclic shifts on ρ_1. Let $t > 0$. Let*

$$f_{\rho_j} = \frac{1 - p_{j-1}}{n - \sum_{k=1}^n p_k} t \text{ for all } j \in [1 \ldots n].$$

(where $p_0 = p_n$) and let $f_\pi = 0$ for all other $\pi \in \phi(n)$. Then $\sum_{\pi \in \phi(n)} f_\pi = t$ and for all $i \in [1 \ldots n]$,

$$\sum_{\pi \in \phi(n)} f_\pi g(\pi, i) = \frac{1 - \prod_{k=1}^n p_k}{n - \sum_{k=1}^n p_k} t. \tag{17}$$

Proof. Clearly $\sum_{\pi \in \phi(n)} f_\pi = t$. Let i be in the range $[1 \ldots n]$. For the given assignment to the flow variables, $\sum_{\pi \in \phi(n)} f_\pi g(\pi, i) = \sum_{j=1}^{n} \frac{(1-p_{j-1})}{n - \sum_{k=1}^{n} p_k} g(\rho_j, i)t$. Recall that by definition, $g(\rho_j, i) = p_j \ldots p_{i-1}$ when $j \leq i$ and $g(\rho_j, i) = p_j \ldots p_n p_1 \ldots p_{i-1}$ when $j > i$. Thus, by cancellation of terms,

$$\sum_{j=1}^{n} \frac{(1-p_{j-1})}{n - \sum_{k=1}^{n} p_k} g(\rho_j, i)t = \frac{1}{n - \sum_{k=1}^{n} p_k} t - \frac{\prod_{k=1}^{n} p_k}{n - \sum_{k=1}^{n} p_k} t.$$

□

In each recurisve call of the second max-throughput algorithm, the operators are partitioned into sets (mega-operators) $E_m, \ldots, E_1$, and we push flow through the operators using a routing that obeys two properties. First, it only sends flow along permutations in which it goes first through the operators in E_m, then through operators in E_{m-1}, then through the operators in E_{m-2}, and so on. Second, for any set E_i in the partition, the amount of incoming flow is the same for all the operators in E_i. We rely on the following technical lemma (which follows from Lemma 3.7).

Lemma 3.8 *Let $E_m, \ldots, E_1$ be a partition of the set of operators $\{O_n, \ldots, O_1\}$ such that for $i \in [1 \ldots m]$, there exist indices $b(i) \leq c(i)$ such that $E_i = \{O_{c(i)}, O_{c(i)-1}, \ldots, O_{b(i)}\}$. Let $\beta(i)$ be the set of $|E_i| = c(i) - b(i) + 1$ permutations which are the left cyclic shifts of the permutation $c(i), \ldots, b(i)$. Let P be the set of $\prod_{l=1}^{m} |E_l|$ permutations $\pi_m \ldots \pi_1$ where each $\pi_i \in \beta(i)$. Suppose we probabilistically route a total of t tuples through the operators, dividing them among the permutations in P, such that the expected rate of tuples routed via permutation $\pi_m \ldots \pi_1$ is*

$$\frac{\prod_{l=1}^{m}(1 - p_{z(\pi_l)})}{\sum_{\pi_m \ldots \pi_1 \in P} \prod_{l=1}^{m}(1 - p_{z(\pi_l)})} t$$

where $z(\pi_l)$ is the last element of the permutation π_l. Then for all $i \in [1 \ldots m]$, the expected rate of tuples arriving at any operator in E_i is $t\xi(i)$ tuples per unit time, where

$$\xi(i) = p_n p_{n-1} \ldots p_{c(i)+1} \frac{1 - \prod_{j=b(i)}^{c(i)} p_j}{|E_i| - \sum_{j=b(i)}^{c(i)} p_j}. \qquad (18)$$

We now describe our second max-throughput algorithm in detail. Pseudocode is given in Algorithm 2. Assume that $r_n \geq r_{n-1} \geq \ldots \geq r_1$. The algorithm is recursive, and $r_n \geq r_{n-1} \geq \ldots \geq r_1$ holds in the recursive calls also.

We partition the operators into equivalence classes $E_m, \ldots, E_1$, where operators are in the same class if they have the same rate limit. Denote the rate limits of the operators in $E_m, \ldots, E_1$ by $R_m, \ldots, R_1$ respectively. Assume the E_i satisfy $R_m > R_{m-1} > \ldots > R_1$.

We will use the following notation. Suppose that we send tuples through the system at a rate of t tuples per unit time, according to the method of Lemma 3.8. Then for every E_i, tuples arrive at each operator in E_i at a rate of $t\xi(i)$ tuples per unit time (where $\xi(i)$ is as defined in Equation (18) of Lemma 3.8). Let $R'_m = R'_m(t), \ldots, R'_1 = R'_1(t)$ denote the residual capacities of the operators in $E_m, E_{m-1}, \ldots, E_1$ respectively, and let $r'_n = r'_n(t), \ldots, r'_1 = r'_1(t)$ denote the residual capacities of the individual operators $O_n, \ldots, O_1$. Then at $t = 0$, $R'_m > \ldots > R'_1$ and $r'_n \geq \ldots \geq r'_1$. As t increases, each R'_i (and r'_j) decreases continuously.

Next, we set $\hat{t}$ to be the smaller of (1) $\frac{r_1}{\xi(1)}$ or (2) the minimum of $\frac{R_i - R_{i-1}}{\xi(i) - \xi(i-1)}$, taken over all $i \in [2 \ldots m]$, where ξ is as defined in Equation (18) of Lemma 3.8. The first quantity is the smallest (positive) value of t at which the residual capacity of O_1 becomes 0, and the second quantity denotes the value of t at which the values R'_i and R'_{i-1} become equal. Thus $\hat{t}$ is the value of t that meets the stopping condition described in Section 3.3.1. Let $\hat{R}_m, \ldots, \hat{R}_1$ and $\hat{r}_n, \ldots, \hat{r}_1$ denote the values of $R'_m, \ldots, R'_1$ and $r'_n, \ldots, r'_1$ respectively when $t = \hat{t}$.

We claim that $\hat{R}_m \geq \ldots \geq \hat{R}_1$. Suppose not. Then $\hat{R}_i < \hat{R}_{i-1}$ for some i. Since at $t = 0$, $R'_i > R'_{i-1}$, both quantities decrease continuously as t increases, and $R'_i < R'_{i-1}$ at $t = \hat{t}$, there must be a value of t that is less than $\hat{t}$ for which $R'_i = R'_{i-1}$. But this contradicts our choice of $\hat{t}$. We have thus shown that $\hat{R}_m \geq \ldots \geq \hat{R}_1$ and hence $\hat{r}_n \geq \ldots \geq \hat{r}_1$.

Let K be the assignment to the flow variables induced by routing $\hat{t}$ tuples per unit time according to Lemma 3.8. To do our computation in polynomial time, we represent K succinctly, as the pair consisting of the partition $E_m, \ldots, E_1$ and the value $\hat{t}$ (from which, using Lemma 3.8, we can determine K).

If $\hat{t}$ equals quantity (1), namely $\frac{r_1}{\xi(1)}$, then we output K. Otherwise, it must be that (2) $<$ (1), and we recursively run the algorithm with selectivities $p_1, \ldots, p_n$ and rate limits $\hat{r}_1, \ldots, \hat{r}_n$. Note that for any j, k, if $r_j = r_k$, then $\hat{r}_j = \hat{r}_k$. Further, for at least one j, $r_j \neq r_{j+1}$, but $\hat{r}_j = \hat{r}_{j+1}$. Thus the equivalence classes E_i in each recursive call are formed by merging equivalence classes from the previous call, and the total number of equivalence classes decreases in each recursive call.

Let K' be the solution returned by the recursive call. We output K'', the solution to the LP which is obtained by setting each flow variable f_π to the sum of its value in K and K'. We can represent K'' succinctly as the concatenation of the representations of K and K'.

This completes the description of the algorithm. The number of equivalence classes decreases in each recursive call, so the number of recursive calls is at most $n - 1$. The time per recursive call is $O(n)$. Therefore, the algorithm runs in time $O(n^2)$.

It remains to prove that the algorithm outputs an optimal solution to the max-throughput LP. In the final recursive call, since $\hat{R}_m \geq \ldots \geq \hat{R}_1$, there is a maximum i such that $\hat{R}_i = \hat{R}_{i-1} = \ldots = \hat{R}_1 = 0$, and no other $\hat{R}_j$ is equal to 0. Let $O_q, O_{q-1}, \ldots, O_1$ be the operators in $E_i, E_{i-1}, \ldots, E_1$. Let $Q = \{1, \ldots, q\}$. Then in the final solution to the original max-throughput problem, constructed from all the recursive calls, Q is the set of indices of operators with residual capacity 0. Also, since the partitions in each recursive call are formed by merging sets of the partition in the previous call, tuples are only routed along permutations in which the operators indexed by elements of Q appear at the end (in some order). It follows that the solution obeys the saturated-suffix property, and hence, by Lemma 3.5 is optimal.

The output of the algorithm is a list of some n' pairs $(P_1, \hat{t}_1), \ldots, (P_{n'}, \hat{t}_{n'})$, where the P_i's are the partitions of operators and the $\hat{t}_i$'s are the $\hat{t}$ values. To use this representation in order to actually route tuples in a distributed environment, first calculate the sum $T = \sum_{i=1}^{n'} \hat{t}_i$. Send T tuples per unit time using the following procedure to route each tuple. For each tuple, first randomly choose an $i \in [1 \ldots n']$, with probability proportional to $\hat{t}_i$. For the chosen i, suppose $P_i = E_m, \ldots, E_1$. For each $j \in [1 \ldots m]$, randomly choose a permutation π_j from the permutations in $\beta(j)$ (defined in Lemma 3.8), with probability proportional to $1 - p_{z(\pi_j)}$, where $z(\pi_j)$ is the last element of the ordering π_j. Then route the tuple via permutation $\pi_m \ldots \pi_1$.

3.5 Reducing the number of permutations in the second algorithm

The potentially exponential number of permutations used by the second algorithm can be reduced to $O(n^2)$. In each recursive call of the second algorithm, flow $\hat{t}$ constructed in that iteration is routed through equivalence classes $E_m, \ldots, E_1$ by first directing it through a permutation of (the operators in) E_m, then a permutation of E_{m-1}, and so forth, as described in Lemma 3.8. An exponential number of routings are possible, in general, because flow might be routed along any of

```
**algorithm** SolveMaxThroughput2($p_1, \ldots, p_n, r_1, \ldots, r_n$)
 **input:**  $n$ selectivities $p_1, \ldots, p_n$; $n$ rate limits $r_1 \leq \ldots \leq r_n$
 **output:** compact representation of solution to max-throughput problem
          for the given input parameters

1. // form the equivalence classes $E_m, \ldots, E_1$;
   $E_1 = \{O_1\}$;
   $m = 1$; //$m$ is number of equivalence classes
   $R_1 = r_1$;

   **for** $(k = 2; k \leq n; k{+}{+})$
     **if** $(r_k \neq r_{k-1})\{$
       $m{+}{+}$;
       $E_m = \{O_k\}$;
       $R_m = r_k$;
     }
     **else** {           //$(r_k == r_{k-1})$
       $E_m = E_m \bigcup \{O_k\}$;
     }

2. //calculate $\hat{t}$ using the following steps
   **for** $(i = 1; i \leq m; i{+}{+})$ {           //for each equivalence class $E_i$
     $c(i)$ = highest index of an operator in $E_i$;
     $b(i)$ = lowest index of an operator in $E_i$;
     $\xi(i) = p_n p_{n-1} \ldots p_{c(i)+1} \frac{1-\prod_{j=b(i)}^{c(i)} p_j}{|E_i|-\sum_{j=b(i)}^{c(i)} p_j}$;
   }
   $\hat{t}_1 = \frac{r_1}{\xi(1)}$;
   $\hat{t}_2 = \min_{i \in [2 \ldots m]} \left( \frac{R_i - R_{i-1}}{\xi(i) - \xi(i-1)} \right)$;
   $\hat{t} = \min(\hat{t}_1,\ \hat{t}_2)$;

3. // calculate the residual capacity for each operator $O_k$
   **for** $(k = 1; k \leq n; k{+}{+})\{$
     $j$ = index of the equivalence class $E_j$ containing operator $O_k$;
     $\hat{r}_k = r_k - \xi(j)\hat{t}$;
   }

4. $K = (\{E_m, \ldots, E_1\}, \hat{t})$;
   **if** $(\hat{r}_1 == 0)$ //residual capacity of equivalence class $E_1$ is 0
     return $K$
   **else** {
     $K'$ = SolveMaxThroughput2($p_1, \ldots, p_n, \hat{r}_1, \hat{r}_2, \ldots, \hat{r}_n$);
     Return $K \circ K'$ (i.e. the concatenation of $K$ and $K'$)
   }
```

Algorithm 2: Pseudocode of second algorithm for the max-throughput problem.

the $\prod_{i=1}^{m} |E_i|$ routes produced by choosing one of the $|E_i|$ cyclic permutations used to route flow through each E_i.

However, it is not necessary to allow each of these possible routings; we show how to route the $\hat{t}$ flow from a given iteration among $O(n)$ permutations instead. The crucial observation is as follows. For each operator O_i, let t_i be the amount of the $\hat{t}$ flow arriving at operator O_i under the current routing of the $\hat{t}$ flow. Then any alternative routing of the $\hat{t}$ flow that also results in t_i flow arriving at each operator O_i can be substituted for the original routing: the original routing of the $\hat{t}$ flow is part of a solution to the max-throughput problem that has the saturated-suffix property, and substituting the alternative routing of the $\hat{t}$ flow preserves this property and thus is still optimal.

For each equivalence class E_i, let $m_i = |E_i|$ and let $\pi_{i,1}, \ldots, \pi_{i,m_i}$ be the m_i cyclic permutations used to route flow through operators in E_i. For j in $\{1, \ldots, m_i\}$, let $q_{i,j} \in [0, 1]$ be such that the fraction of the $\hat{t}$ flow sent through E_i along permutation $\pi_{i,j}$ is $q_{i,j}\hat{t}$. Note that $\sum_{j=1}^{m_i} q_{i,j} = 1$. Let $s_{i,0} = 0$, and let $s_{i,j} = \sum_{k=1}^{j} q_{i,k}$ for each j in $\{1, \ldots, m_i\}$. Each permutation $\pi_{i,j}$ has associated with it an interval $I_{i,j} = [s_{i,j-1}, s_{i,j}]$; the intervals $I_{i,j}$ form a partition of the interval $[0, 1]$ where the length $s_{i,j} - s_{i,j-1}$ of $I_{i,j}$ is equal to $q_{i,j}$. Let $S = \bigcup_{i=1}^{m} \{s_{i,j} | 1 \leq j \leq m_i\}$. Let $s_0 = 0$, $s_1, \ldots, s_r$ be the elements of S in sorted order, and let I_k be the interval $[s_{k-1}, s_k]$ for $1 \leq k \leq r$. Then for each i where $1 \leq i \leq m$, interval I_k is a subinterval of $I_{i,j}$ for some j; let $\rho_{i,k}$ be the permutation of E_i associated with the $I_{i,j}$ containing I_k. For each k, $1 \leq k \leq r$, we route $(s_k - s_{k-1})\hat{t}$ flow along the route which sends it first through E_m via the permutation $\rho_{m,k}$, then through E_{m-1} via the permutation $\rho_{m-1,k}$, and so forth.

Figure 3 gives a simple example. The equivalence class E_2 contains two operators, O_5 and O_4, and so the two cyclic permutations $\pi_{2,1} = (5, 4)$ and $\pi_{2,2} = (4, 5)$ are used to route flow through the operators. These routes are depicted as $O_5 \rightarrow O_4$ and $O_4 \rightarrow O_5$ in the figure. Since the selectivities of O_5 and O_4 are equal, the fraction of flow sent along each of the routes $O_5 \rightarrow O_4$ and $O_4 \rightarrow O_5$ should be 1/2; that is $q_{2,1} = q_{1,2} = 1/2$. In the figure, this is indicated by the dashed line which separates the two cyclic permutations, and which lies half way between the line marked 0 and the line marked 1. Similarly, the cyclic permutations used to route flow through operators in E_1 are depicted as $O_3 \rightarrow O_2 \rightarrow O_1$, $O_2 \rightarrow O_1 \rightarrow O_3$, and $O_1 \rightarrow O_3 \rightarrow O_2$. Each should route one third of the flow though E_2, as depicted by the two equi-spaced dashed lines between 0 and 1, which separate the three cyclic permutations. The flow through E_2 followed by E_1 can be routed according to the fractions given in the solution on the right side of Figure 3; this routing ensures that the correct fraction, namely one half, of the flow through E_2 is routed through the two cyclic permutations of the operators in E_2, and similarly one third of the flow through E_1 is routed through the three cyclic permutations of operators given for E_1.

It is easy to verify that, for every $1 \leq i \leq m$ and $1 \leq j \leq |E_i|$, this new allocation of the $\hat{t}$ flow results in the same amount of flow being sent through E_i along permutation $\pi_{i,j}$ as was sent in the original routing. Thus the new allocation also results in the same amount of flow being sent to each operator as in the original routing. Moreover, the total number of permutations (of all the operators) used in the new routing is the number of intervals I_k, which is $(\sum_{i=1}^{m} |E_i| - 1) + 1 = n - m + 1$.

Since there are at most n recursive iterations of the algorithm, and the number m of equivalence classes decreases in each, using the alternative routing in each iteration results in a solution that routes flow along at most $n(n-1)/2 < n^2$ distinct permutations.

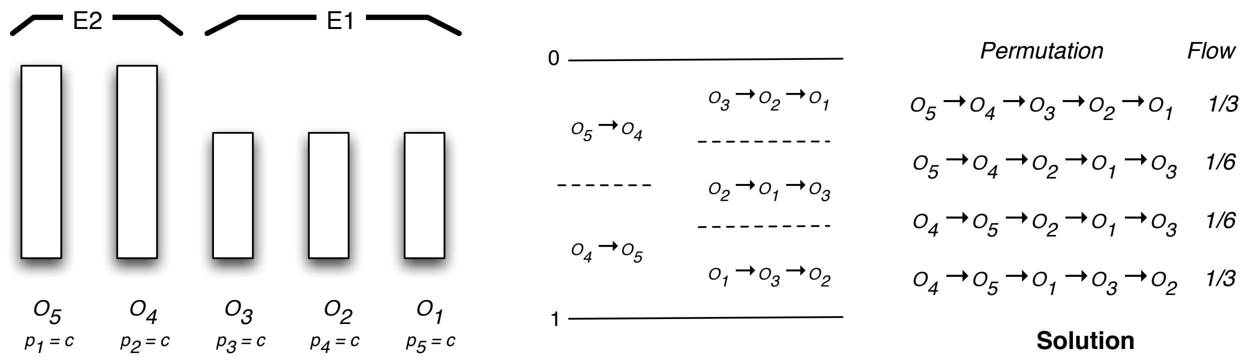

E2
E1
o_5 o_4 o_3 o_2 o_1
$p_1 = c$ $p_2 = c$ $p_3 = c$ $p_4 = c$ $p_5 = c$
0
1
$o_5 \to o_4$
$o_4 \to o_5$
$o_3 \to o_2 \to o_1$
$o_2 \to o_1 \to o_3$
$o_1 \to o_3 \to o_2$
Permutation
Flow
$o_5 \to o_4 \to o_3 \to o_2 \to o_1$ 1/3
$o_5 \to o_4 \to o_2 \to o_1 \to o_3$ 1/6
$o_4 \to o_5 \to o_2 \to o_1 \to o_3$ 1/6
$o_4 \to o_5 \to o_1 \to o_3 \to o_2$ 1/3
Solution

Figure 3: **With two equivalence classes E2 and E1, with 2 and 3 operators respectively, at most four permutations are sufficient to establish the required flow, instead of six.**

3.6 Comparison of the routes used by the two algorithms for the max-throughput problem

The first and second max-throughput algorithms can output qualitatively different solutions to the same problem instance. For example, consider a max-throughput instance in which we have n operators, each with the same rate limit $r = 1$, and the same selectivity $p = 1/2$. The second algorithm will output a solution with flow routed along n cyclic shifts of a single permutation. The first algorithm will not output a solution with this property. It is easy to show that in each recursive call, the first algorithm will merge the first two operators together (which switches their order in future permutations). Thus, for example, if the initial permutation is $(4, 3, 2, 1)$, the other permutations will be $(3, 4, 2, 1)$, $(2, 3, 4, 1)$, and $(1, 2, 3, 4)$.

3.7 Comparison of naive vs. optimal strategies for the max-throughput problem

We now consider the naive solution for the max-throughput problem, which sends all flow through the operators in decreasing order of their rate limits, and compare it to the optimal solution.

Lemma 3.9 *Consider an instance of the max-throughput problem. Let F_{Naive} be the throughput achieved by routing all flow through the operators in decreasing order of their rate limits. Let F^* be the throughput achieved by the optimal routing. Then*

$$\frac{F^*}{F_{Naive}} \leq n.$$

Proof. The linear program for the max-throughput problem has only n constraints, and so there is an optimal solution to the max-throughput problem which sends tuples along at most n distinct routes. One of these routes, say π, must account for at least $\frac{1}{n}$th of the total throughput, F. Thus, if F_π is the throughput that would be achieved if all tuples were sent along route π, then $F_\pi \geq F^*/n$. Moreover, F_π must be at most the throughput of the Naive strategy, since it is the optimal strategy when flow can be sent along a single ordering. Thus $F_{Naive} \geq F_\pi \geq F^*/n$. □

We show now that the factor of n in the above lemma cannot be improved. Consider first a version of the max-throughput problem in which selectivities are allowed to be equal to 0 (rather than strictly greater than 0). If the selectivity of every operator is equal to 0, and the rate limits of the processors are equal to some value r, then using the same route for each tuple yields throughput of r, while allowing different routes for the tuples enables a throughput of nr, that is, n times larger.

As we have defined the max-throughput problem, selectivites cannot equal 0; however by making the selectivity of each operator be arbitrarily close to 0, we can obtain a factor that is arbitrarily close to n.

4 The game theoretic multiplicative regret (GTMR) problem and variants

We begin by giving a formal definition of this problem. An instance of the GTMR problem is a list of positive real costs $c_1, \ldots, c_n$. Let $h(\pi, i) = c_{\pi(1)} + c_{\pi(2)} + \ldots + c_{\pi(m)}$ where $m = \pi^{-1}(i)$. Let $\phi(n)$ denote the set of all permutations of $\{1, \ldots, n\}$. The GTMR problem is given by the minimax formulation below. The f_π denote the probability of choosing to route the tuple through the operators according to the ordering specified by permutation π.

Game theoretic multiplicative regret:
Given $c_1, \ldots, c_n > 0$, minimize

$$\max_{i \in \{1,\ldots,n\}} \sum_{\pi \in \phi(n)} f_\pi h(\pi, i)/c_i$$

subject to the constraints

$$\sum_{\pi \in \phi(n)} f_\pi = 1$$

$$f_\pi \geq 0 \text{ for all } \pi \in \phi(n)$$

For example, consider an instance of the GTMR problem with $c_1 = c_2 = c_3$. The intuitive strategy of choosing a random routing (uniformly) is optimal. An alternative optimal strategy is to choose the orderings 1,2,3; 2,3,1; and 3,1,2 with equal probability.

A contrasting example is when the costs are $c_1 = 2, c_2 = 2$, and $c_3 = 8$. If the adversary selects O_1 or O_2 to eliminate the tuple, then routing the tuple to O_3 first is bad – it results in an expected multiplicative regret of at least 5 $(= (8 + 2)/2)$. In fact, it can be shown that the only optimal strategy for the routing player is to choose one of the orderings 1,2,3 and 2,1,3, each with probability 1/2, yielding expected multiplicative regret of 3/2. Note that both these orderings have 3 in the last position. Finally, suppose $c_1 = c_2 = 2$ and $c_3 = 7$. In this case, one optimal strategy is to choose from the permutations 1,2,3; 1,3,2; and 2,1,3, with probabilities 23/57, 2/57, and 32/57, respectively.

4.1 Algorithm to calculate an optimal routing for the GTMR problem

We relate the max-throughput problem and the GTMR problem by studying an (artificial) problem that we call the *cumulative cost limit problem.* The solution to this problem has many similarities to the solution to the max-throughput problem.

In the cumulative cost limit problem, we again have n operators $O_1, O_2, \ldots, O_n$ with costs, and we need to decide how many tuples per unit time to route along each permutation. However, in this problem there is also a cost limit d_i associated with each operator O_i. Tuples cannot be eliminated by operators, and the processing of each tuple is deterministic. Costs are cumulative, so that when a tuple arrives at an operator O_i, the amount that must be paid for O_i to process it is c_i plus the sum of all costs c_j associated with operators O_j that have already processed that tuple. Operators have no limit on the *number* of tuples they can process per unit time. Instead, they are limited by their cumulative cost limit d_i, which is an upper bound on the total amount that can be paid for using that operator per unit time. The problem is to route the tuples so as to maximize the rate of tuples that can be processed, subject to the cumulative cost limits.

Formally, the *cumulative cost limit problem* is given by the linear program below, where $\phi(n)$ denotes the set of permutations of $\{1,\ldots,n\}$ as before, and $h(\pi,i) = \sum_{j=1}^{m} c_{\pi(j)}$, where $m = \pi^{-1}(i)$.

Cumulative cost limit LP: Given $c_1,\ldots,c_n > 0$ and $d_1,\ldots,d_n > 0$, maximize

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

subject to the constraints

$$\sum_{\pi\in\phi(n)} f_\pi h(\pi,i) \le d_i \text{ for all } i \in \{1,\ldots,n\}$$

$$f_\pi \ge 0 \text{ for } \pi \in \phi(n)$$

Our max-throughput algorithms were based on the fact that any feasible solution satisfying the saturated suffix property is optimal. In the next lemma, we prove that an analogous saturated-*prefix* property guarantees optimality for the cummulative cost limit problem.

Lemma 4.1 *If a feasible solution to the cumulative cost limit LP has the property that for some non-empty subset $Q \subseteq \{1,\ldots,n\}$, (1) the cumulative cost limit constraint for operator O_i is tight iff $i \in Q$ and (2) for any flow variable f_π, $f_\pi > 0$ implies that in permutation π, the elements of Q precede the elements of $\bar{Q}$, then the feasible solution is optimal and the value of the objective function under the feasible solution is*

$$\frac{\sum_{i\in Q} c_i d_i}{(\sum_{i\in Q} c_i^2) + (\sum_{i,j\in Q, i<j} c_i c_j)} \tag{19}$$

Proof. Consider a feasible solution satisfying the conditions of the lemma. It specifies the rate at which tuples should be sent along each permutation. Let $C_Q = \sum_{j\in Q} c_j$. For each $i \in Q$, the cumulative cost limit constraint for O_i is tight, i.e. $\sum_{\pi\in\phi(n)} f_\pi h(\pi,i) = d_i$. Multiplying both sides of this constraint by $\frac{c_i}{C_Q}$ and summing over all $i \in Q$, we get that

$$\sum_{i\in Q}\sum_{\pi\in\phi(n)} f_\pi h(\pi,i)\frac{c_i}{C_Q} = \sum_{i\in Q}\frac{d_i c_i}{C_Q}. \tag{20}$$

Exchanging the order of the summations on the left hand side of the equation shows it is equal to

$$\sum_{\pi\in\phi(n)} f_\pi \sum_{i\in Q} h(\pi,i)\frac{c_i}{C_Q}. \tag{21}$$

Let $\phi'(n)$ denote the permutations of $\phi(n)$ in which the elements of Q precede the elements of $\bar{Q}$. By assumption, $f_\pi = 0$ for all $\pi \notin \phi'(n)$. Let $\pi \in \phi'(n)$. Consider $\sum_{i\in Q} h(\pi,i)\frac{c_i}{C_Q}$. The quantity $h(\pi, i)$ is equal to c_i plus the sum of the c_j such that j precedes i in π. For any $j, k \in Q$ such that $j \neq k$, either j precedes k in π and c_j is an element of the sum $h(\pi, k)$, or k precedes j in π, and c_k is an element of the sum $h(\pi, j)$. It follows that $\sum_{i\in Q} h(\pi,i)\frac{c_i}{C_Q} = U_Q/C_Q$, where $U_Q = (\sum_{i\in Q} c_i^2) + (\sum_{i,j\in Q, i<j} c_i c_j)$. Hence $\sum_{\pi\in\phi(n)} f_\pi U_Q/C_Q = \sum_{i\in Q}\frac{d_i c_i}{C_Q}$. It immediately follows that $F = \sum_{\pi\in\phi(n)} f_\pi = \frac{\sum_{i\in Q} d_i c_i}{U_Q}$.

We now show that the value of F cannot be larger than this value, for any feasible solution to the cumulative cost limit LP. Consider a modified version of the cumulative cost limit LP in which we eliminate all cost limit constraints for selection operators O_j such that $j \notin Q$. Consider an optimal solution to this modified problem which assigns values f'_π to each of the variables f_π. Let $F' = \sum_{\pi\in\phi(n)} f'_\pi$ be the value of the objective function. Clearly F' is an upper bound on the maximum possible value of the objective function for the original cumulative cost limit LP. Let $\phi'(n)$ be as defined above. Because selection operators O_j such that $j \notin Q$ have no cost limit constraints, and because each operator can only increase the cumulative amount of cost that will be passed on to subsequent operators, we may assume without loss of generality that if $f'_\pi > 0$, then $\pi \in \phi'(n)$. If we take the cost limit constraints for O_i where $i \in Q$, multiply both sides of each by $\frac{c_i}{C_Q}$, and add the resulting inequalities, we get that $\sum_{i\in Q}\sum_{\pi\in\phi(n)} f'_\pi h(\pi,i)\frac{c_i}{C_Q} \leq \sum_{i\in Q}\frac{d_i c_i}{C_Q}$. The same argument as above shows that $F' \leq \frac{\sum_{i\in q} d_i c_i}{U_Q}$. □

The approach we used in our first max-throughput algorithm does not, however, seem to work here. In that algorithm, we merge two operators O_{i+1} and O_i when $r_{i+1} = r_i p_i$, and send all subsequent flow so it goes to O_{i+1} immediately after going to O_i. In this way, we guarantee that both O_i and O_{i+1} will become saturated at the same time in the future (if they do, in fact, become saturated); the guarantee is based on the fact that the total amount of future flow into O_{i+1} will be exactly p_i times the total amount of future flow into O_i. However, in the cummulative cost limit problem it isn't clear how to ensure simultaneous saturation by placing O_{i+1} permanently in front of O_i, because the amount of future flow into (i.e. cost incurred by) O_{i+1} is not a fixed amount times the amount of future flow into O_i.

However, the approach used in our second max-throughput algorithm works quite directly for the cummulative cost limit problem, because we can prove the following "load balancing" lemma. It specifies a way to route t tuples per unit time so as to ensure that each operator has the same cumulative cost per unit time.

Lemma 4.2 *Let ρ_1 be the permutation $1, \ldots, n$ and for $j \in [2 \ldots n]$, let ρ_j be permutation $j, j+1, \ldots, n, 1, 2, \ldots, j-1$, that is, the permutation obtained by performing $j-1$ left cyclic shifts on ρ_1. Let $t > 0$. Suppose we send a total of t tuples per unit time through the operators, using the routing which sets f_{ρ_i} to $t\frac{c_i}{\sum_{j=1}^n c_j}$ for all $i \in \{1, \ldots, n\}$, and sets $f_\pi = 0$ for all other $\pi \in \phi(n)$. Then the amount that must be paid for every operator per unit time is $\frac{(\sum_{j=1}^n c_j^2) + (\sum_{1\leq i<j\leq n} c_i c_j)}{\sum_{j=1}^n c_j} t$.*

Proof. Let $i \in \{1, \ldots, n\}$. The quantity $\sum_{\pi\in\phi(n)} f_\pi h(\pi, i)$ is equal to $t\sum_{j=1}^n \frac{c_j}{\sum_{k=1}^n c_k} h(\rho_j, i)$, where $h(\rho_j, i)$ is equal to c_i plus the sum of the c_k such that k precedes i in ρ_j. Expanding the term $h(\rho_j, i)$ and multiplying out, each permutation ρ_j contributes to the expression the term $(c_j^2/\sum_{j=1}^n c_j)t$ and terms $(c_j c_k/\sum_{j=1}^n c_j)t$ for all k such that k precedes i in ρ_j. Consider any $j, k \in \{1, \ldots, n\}$ such that $j < k$ and $j, k \neq i$. If $j < i < k$, then j precedes i in ρ_k but k does not precede i in ρ_j. If, on the other hand, $i < j$ or $i > k$, then k precedes i in ρ_j, but j does not precede i in ρ_k. It follows that the sum of the terms of the expression is

$t\frac{(\sum_{j=1}^{n} c_j^2)+(\sum_{1\le i<j\le n} c_i c_j)}{\sum_{j=1}^{n} c_j}$. □

As in the max-throughput problem, we use this lemma to prove a technical lemma that gives a method of routing tuples so that, given a partition $E_m, \ldots, E_1$, we only send tuples along permutations in which operators in E_m are first, E_{m-1} are next, and so on, and such that the cumulative cost per unit time for an operator to process the tuples within any given set in the partition is the same for all operators.

With the technical lemma, we have the same building blocks that we had for the max-throughput algorithm, and we can essentially run the same algorithm to solve the cumulative cost limit problem (with the routing method for keeping costs equal, a different calculation for computing $\hat{t}$, and with the operators ordered in increasing cost limit order, rather than in decreasing rate limit order). Lemma 4.1 proves that the solution computed by the algorithm is optimal.

We now describe how the algorithm for the cumulative cost limit problem can be used to solve the GTMR problem. In standard routing problems with limits on the capacity of edges (or nodes), congestion minimization and throughput maximization are closely related. Congestion is the maximum, over all edges, of the *relative load* of an edge, the amount of flow through the edge divided by the capacity of the edge. If there is a routing of k flow units that achieves 5% congestion, then scaling the 5% congestion routing by a factor of 20 yields throughput of $20k$ (with 100% congestion). Comparison of the GTMR LP to the cost limit LP reveals that the GTMR problem is the congestion minimization problem corresponding to the cumulative cost limit (max-throughput) problem, and flow achieving minimum (cost) congestion can be scaled to achieve maximum throughput. In the next lemma, we formally reduce the GTMR problem to the cumulative cost limit problem. In what follows, for any assignment A of values to the flow variables f_π, $\pi \in \phi(n)$, let $f_\pi(A)$ denote the value assigned to f_π by A.

Lemma 4.3 *Let I_{mult} be an instance of the GTMR problem with costs $c_1, \ldots, c_n > 0$. Let I_{cost} be the instance of the cumulative cost limit problem with costs $c_1, c_2, \ldots, c_n$ and cumulative cost limits $d_1 = c_1, d_2 = c_2, \ldots, d_n = c_n$. Let K be the optimal solution to I_{cost}, and let F be the value of the objective function achieved by K. Let L be the assignment to flow variables f_π such that for each $\pi \in \phi(n)$, $f_\pi(L) = f_\pi(K)/F$. Then L is an optimal solution for I_{mult}.*

Proof. Since K is an optimal solution to I_{cost}, F is the maximum value of the objective function for I_{cost}. We show that L is an optimal solution for I_{mult}.

Since $\sum_{\pi\in\phi(n)} f_\pi(K) = F$, $\sum_{\pi\in\phi(n)} f_\pi(K)/F = 1$. Hence L satisfies the constraints of the GTMR problem. Since K maximizes the value of the objective function for the instance I_{cost} of the cumulative cost limit problem, there must be at least one i such that $\sum_{\pi\in\phi(n)} f_\pi(K)h(\pi, i) = c_i$ and hence $\sum_{\pi\in\phi(n)}(f_\pi(K)/F)h(\pi, i)/c_i = \frac{1}{F}$. Also, for every i,
$\sum_{\pi\in\phi(n)}(f_\pi(K)/F)h(\pi, i)/c_i \le \frac{1}{F}$.

Let H be the value of the objective function achieved by L for problem I_{mult}. That is, H is the maximum of $\sum_{\pi\in\phi(n)}(f_\pi(K)/F)h(\pi, i)/c_i$ over all i. Thus $H = \frac{1}{F}$.

Suppose L is not an optimal solution to the instance I_{mult} of the GTMR problem. Then there exists some other solution $\tilde{L}$ that is optimal. Let $\tilde{H}$ be the value of the objective function achieved by $\tilde{L}$. Thus $\tilde{H} < H$.

Let $\tilde{K}$ be the assignment to the flow variables such that $f_\pi(\tilde{K}) = F f_\pi(\tilde{L})$ for all $\pi \in \phi(n)$. The value of the objective function for I_{cost} achieved by $\tilde{K}$ is $\sum_{\pi\in\phi(n)} f_\pi(\tilde{K}) = \sum_{\pi\in\phi(n)} F f_\pi(\tilde{L}) = F$ because $\sum_{\pi\in\phi(n)} f_\pi(\tilde{L}) = 1$.

Since $\tilde{H} < H$, under solution $\tilde{L}$, for all i,

$$\sum_{\pi \in \phi(n)} f_\pi(\tilde{L}) h(\pi, i)/c_i \leq \tilde{H} < H = \frac{1}{F},$$

and thus $\sum_{\pi \in \phi(n)} F f_\pi(\tilde{L}) h(\pi, i) < c_i$.

Therefore, $\sum_{\pi \in \phi(n)} f_\pi(\tilde{K}) h(\pi, i) < c_i$. That is, $\tilde{K}$ is a feasible solution to I_{cost} such that none of the constraints are tight. It follows that there is a feasible solution $\tilde{M}$ to I_{cost} such that the value of the objective function under $\tilde{M}$ is greater than F. But this contradicts that F is the maximum possible value of the objective function for I_{cost}. □

The above reduction, together with the algorithm for the cumulative cost limit problem, yield an $O(n^2)$ algorithm for solving the GTMR problem. We note that, as in the case of the second algorithm for the max-throughput problem, the algorithm for the GTMR problem can output a solution using an exponential number of permutations, but can be modified to use $O(n^2)$ permutations. We do not currently have an algorithm for the GTMR problem that outputs a sparse solution, using n permutations.

4.2 Comparison of naive vs. optimal strategies for the GTMR problem

We now show that, for any set of costs, the naive strategy, which orders operators in increasing order of their costs, achieves multiplicative regret that is within a factor 2 of the expected multiplicative regret achieved by the optimal strategy.

Consider the GTMR problem, with costs $0 < c_1 \leq \ldots \leq c_n$. Let $Naive$ be the deterministic strategy for the routing player that orders the operators in increasing order of their costs. Let Opt be the optimal strategy for the routing player that we obtain in Lemma 4.3. Let $v_{GTMR}(Naive)$ be the expected multiplicative regret when the routing player uses strategy $Naive$ and the adversary uses the best (possibly randomized) strategy against $Naive$. Similarly, let $v_{GTMR}(Opt)$ be the expected multiplicative regret when the routing player uses Opt and the adversary uses the best (possibly randomized) strategy against Opt. Note that

$$v_{GTMR}(Naive) = \max_k \frac{\sum_{i=1}^k c_i}{c_k}.$$

This is because $\frac{\sum_{i=1}^k c_i}{c_k}$ is the multiplicative regret when the routing player uses $Naive$ and the adversary chooses operator k to discard the tuple; thus the best strategy of the adversary against $Naive$ maximizes this value.

Also, it is the case that

$$v_{GTMR}(Opt) = \max_k \frac{\sum_{i=1}^k c_i^2 + \sum_{1 \leq i < j \leq k} c_i c_j}{\sum_{i=1}^k c_i^2}. \tag{22}$$

Briefly, this follows from two inequalities. To obtain the first, consider the strategy of the adversary that chooses O_i (to be the operator which discards the tuple) with probability $\frac{c_i^2}{\sum_{i=1}^k c_i^2}$ if $i \in \{1, \ldots, k\}$ and with probability 0 otherwise. It can be shown that for any permutation π, if the routing player routes the tuple (deterministically) according to π, the expected multiplicative regret is at least $\frac{\sum_{i=1}^{k} c_i^2 + \sum_{1\le i<j\le k} c_i c_j}{\sum_{i=1}^{k} c_i^2}$ against this strategy of the adversary. Hence,

$$\max_k \frac{\sum_{i=1}^{k} c_i^2 + \sum_{1\le i<j\le k} c_i c_j}{\sum_{i=1}^{k} c_i^2} \le v_{GTMR}(Opt).$$

Second, Lemmas 4.1 and 4.3, together with the algorithm of Lemma 4.2, imply that for some k,

$$v_{GTMR}(Opt) = \frac{\sum_{i=1}^{k} c_i^2}{(\sum_{i=1}^{k} c_i^2) + (\sum_{1\le i<j\le k} c_i c_j)}.$$

Hence $v_{GTMR}(Opt) \le \max_k \frac{\sum_{i=1}^{k} c_i^2}{(\sum_{i=1}^{k} c_i^2)+(\sum_{1\le i<j\le k} c_i c_j)}$. Combining both inequalities yields (22).

Lemma 4.4

$$\frac{v_{GTMR}(Naive)}{v_{GTMR}(Opt)} \le 2.$$

Proof. Let $m = \arg\max_k \frac{\sum_{i=1}^{k} c_i}{c_k}$, that is, m is the value of k that maximizes the multiplicative regret under the *Naive* strategy. Thus $v_{GTMR}(Naive) = \frac{\sum_{i=1}^{m} c_i}{c_m}$. Note that

$$\begin{aligned} v_{GTMR}(Opt) &= \max_k \frac{\sum_{i=1}^{k} c_i^2 + \sum_{1\le i<j\le k} c_i c_j}{\sum_{i=1}^{k} c_i^2} \\ &\ge \frac{\sum_{i=1}^{m} c_i^2 + \sum_{1\le i<j\le m} c_i c_j}{\sum_{i=1}^{m} c_i^2} \\ &= \frac{(\sum_{i=1}^{m} c_i)^2 + \sum_{i=1}^{m} c_i^2}{2\sum_{i=1}^{m} c_i^2}. \end{aligned}$$

Therefore,

$$\begin{aligned} \frac{v_{GTMR}(Naive)}{v_{GTMR}(Opt)} &\le \left(\frac{\sum_{i=1}^{m} c_i}{c_m}\right)\left(\frac{2\sum_{i=1}^{m} c_i^2}{(\sum_{i=1}^{m} c_i)^2 + \sum_{i=1}^{m} c_i^2}\right) \\ &\le \left(\frac{\sum_{i=1}^{m} c_i}{c_m}\right)\left(\frac{2c_m\sum_{i=1}^{m} c_i}{(\sum_{i=1}^{m} c_i)^2}\right) = 2. \end{aligned}$$

□

4.3 Other game theoretic formulations

The *game theoretic additive regret* (GTAR) problem is analogous to the GTMR problem, except that the goal is to minimize the *difference* between the cost paid and the cost that would have been paid under the least cost routing. The minimax formulation of the GTAR problem is obtained from the minimax formulation of the GTMR problem by replacing $h(\pi, i)/c_i$ by $h(\pi, i) - c_i$.

The *game theoretic total cost* (GTTC) problem seeks to minimize the total cost paid, rather than a regret function, and its minimax formulation is derived from the minimax formulation of the GTMR problem by replacing $h(\pi, i)/c_i$ by $h(\pi, i)$.

A simple linear time algorithm for the GTAR problem follows directly from the following lemma:

Lemma 4.5 *Let ρ_1 be the permutation $1, \ldots, n$ and let ρ_j be permutation $j, j+1, \ldots, n, 1,$ 2, $\ldots, j-1$ for $j \in [2 \ldots n]$. That is, ρ_j is the permutation obtained by performing $j-1$ left cyclic shifts on ρ_1. The assignment to the variables f_π which sets f_{ρ_i} to $\frac{c_{i-1}}{\sum_{j=1}^{n} c_j}$ for all $i \in \{1, \ldots, n\}$ and sets $f_\pi = 0$ for all other $\pi \in \phi(n)$, is a solution to the GTAR problem.*

Proof. Let

$$v_{GTAR}(Opt) = \frac{\sum_{0<i<j\leq n} c_i c_j}{\sum_{j=1}^{n} c_j}.$$

It can easily be verified that in the two-person game defined by the GTAR problem, if the routing player chooses π according to the mixed strategy given in the statement of the lemma, the expected additive regret incurred will be $v_{GTAR}(Opt)$, no matter which operator is chosen by the adversary to eliminate the tuple. Thus $v_{GTAR}(Opt)$ is an upper bound on the optimal value of the objective function of the GTAR problem. On the other hand, if for each i, the adversary chooses O_i as the discarding operator with probability $\frac{c_i}{\sum_{j=1}^{n} c_j}$, then no matter what strategy the routing player uses, the expected additive regret incurred will also be $v_{GTAR}(Opt)$, proving that $v_{GTAR}(Opt)$ is also a lower bound on the optimal value of the objective function of the GTAR problem. It follows that $v_{GTAR}(Opt)$ is equal to the optimal value of the objective function of the GTAR problem, and the strategy given in the lemma is optimal for the routing player. $\square$

A very similar lemma and proof hold for the GTTC problem, leading to a linear time algorithm for that problem also. The solution for GTTC assigns value $\frac{c_i}{\sum_{j=1}^{n} c_j}$ to each f_{ρ_i}.

We note that the GTTC and GTAR problems, like the GTMR problem, restrict the adversary to choose exactly one operator to eliminate the tuple. As discussed earlier, under the multiplicative regret measure used in the GTMR problem, this restriction doesn't disadvantage the adversary. Similarly, it does not disadvantage the adversary under the cost difference measure used in the GTAR problem. However, with respect to the total cost measure used in the GTTC problem, the restriction is a disadvantage, because the adversary would be better off making the tuple satisfy all operators. When a tuple satisfies all operators, though, there is no filter ordering problem to solve, because all orderings yield the same total cost. Therefore, to motivate the GTTC problem, one can begin by assuming that at least one operator eliminates the tuple. Under that assumption, using the total cost measure, the restriction that *exactly* one operator eliminates a tuple does not disadvantage the adversary.

Acknowledgments

We thank Joseph M. Hellerstein for illuminating discussions and for initiating this work. We thank Boris Aronov, Richard Van Slyke, and Shiyan Hu for their insights and for a conjecture that led to a linear-time algorithm for the dual of the GTMR problem. We thank Kamesh Munagala for useful background information and Rachel Pottinger for valuable feedback on an earlier draft.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, volume 29, pages 261–272, 2000.

[2] B. Awerbuch and F. T. Leighton. A simple local-control approximation algorithm for multicommodity flow. In *Proceedings of the 34th IEEE Symposium on Foundations of Computer Science*, pages 459–468. IEEE Computer Society, 1993.

[3] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 407–418. ACM Press, 2004.

[4] A. Bar-Noy, M. Bellare, M.M. Halldórsson, H. Shachnai, and T. Tamir. On chromatic sums and distributed resource allocation. *Inform. and Comput.*, 140(2):183–202, 1998.

[5] J. Burge, K. Mungala, and U. Srivastava. Ordering pipelined query operators with precedence constraints. Technical Report 2005-40, Stanford University, 2005.

[6] S. Chaudhuri, U. Dayal, and T. W. Yan. Join queries with external text sources: Execution and optimization techniques. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 410–422. ACM Press, 1995.

[7] E. Coffman and I. Mitrani. A characterization of waiting time performance realizable by single server queues. *Operations Research*, 20:810–821, 1980.

[8] E. Cohen, A. Fiat, and H. Kaplan. Efficient sequences of trials. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 737–746. ACM Press, 2003.

[9] A. Condon, A. Deshpande, L. Hellerstein, and N. Wu. Flow algorithms for two pipelined filter ordering problems. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM Press, 2006.

[10] A. Deshpande, C. Guestrin, S. Madden, and W. Hong. Exploiting correlated attributes in acquisitional query processing. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 143–154. IEEE Computer Society, 2005.

[11] O. Etzioni, S. Hanks, T. Jiang, R. M. Karp, O. Madani, and O. Waarts. Efficient information gathering on the internet. In *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science*, pages 234–243. IEEE Computer Society, 1996.

[12] U. Feige, L. Lovász, and P. Tetali. Approximating min-sum set cover. *Algorithmica*, 40(4):219–234, 2004.

[13] L. Fleischer and K. Wayne. Fast and simple approximation schemes for generalized flow. *Mathematical Programming*, 91(2):215–238, 2002.

[14] M. Garey. Optimal task scheduling with precedence constraints. *Discrete Mathematics*, 4:37–56, 1973.

[15] E. Gelenbe and I. Mitrani. *Analysis and synthesis of computer systems.* Academic Press, 1980.

[16] R. Goldman and J. Widom. WSQ/DSQ: A practical approach for combined querying of databases and the web. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 285–296. ACM, 2000.

[17] M.D. Grigoriadis and L.G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM Journal on Optimization*, 4:86–107, 1994.

[18] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[19] H. Kaplan, E. Kushilevitz, and Y. Mansour. Learning with attribute costs. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing (STOC)*, pages 356–365. ACM Press, 2005.

[20] M.S. Kodialam. The throughput of sequential testing. In *Proceedings of the 8th International IPCO Conference on Integer Programming and Combinatorial Optimization*, volume 2081, pages 280–292. Lecture Notes in Computer Science, Springer-Verlag, London, UK, 2001.

[21] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *Proceedings of the 12th International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 128–137. Morgan Kaufmann Publishers Inc., San Fransisco, CA, USA, 1986.

[22] K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. In *Proceedings of the Tenth International Conference on Database Theory (ICDT)*, volume 3363, pages 83–98. Lecture Notes in Computer Science, Springer-Verlag, Berlin / Heidelberg, 2005.

[23] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, 1995.

[24] S. A. Plotkin, D. B. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20(2):257–301, 1995.

[25] M.A. Shayman and E. Fernandez-Gaucherand. Risk-sensitive decision-theoretic diagnosis. *IEEE Transactions on Automatic Control*, 46:1166–1171, 2001.

[26] H. Simon and J. Kadane. Optimal problem-solving search: All-or-none solutions. *Artificial Intelligence*, 6:235–247, 1975.

[27] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 355–366, 2006.

[28] A. Weininger. Efficient execution of joins in a star schema. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of data*, pages 542–545, 2002.