

Max-Throughput for (Conservative) k -of- n Testing

Lisa Hellerstein*

Özgür Özkan†

Linda Sellie‡

June 4, 2018

Abstract

We define a variant of k -of- n testing that we call *conservative k -of- n testing*. We present a polynomial-time, combinatorial algorithm for the problem of maximizing throughput of conservative k -of- n testing, in a parallel setting. This extends previous work of Kodialam and Condon et al., who presented combinatorial algorithms for parallel pipelined filter ordering, which is the special case where $k = 1$ (or $k = n$) [4, 5, 8]. We also consider the problem of maximizing throughput for *standard k -of- n testing*, and show how to obtain a polynomial-time algorithm based on the ellipsoid method using previous techniques.

1 Introduction

In *standard k -of- n testing*, there are n binary tests, that can be applied to an “item” x . We use x_i to denote the value of the i^{th} test on x , and treat x as an element of $\{0, 1\}^n$. With probability p_i , $x_i = 1$, and with probability $1 - p_i$, $x_i = 0$. The tests are independent, and we are given p_1, \dots, p_n . We need to determine whether at least k of the n tests on x have a value of 1, by applying the tests sequentially to x . Once we have enough information to determine whether this is the case, that is, *once we have observed k tests with value 1, or $n - k + 1$ tests with value 0*, we do not need to perform further tests.¹

We define *conservative k -of- n testing* the same way, except that we continue performing tests until we have either observed k tests with value 1, or have performed all n tests. In particular, we do not stop testing when we have observed $n - k + 1$ tests with value 0.

There are many applications where k -of- n testing problems arise, including quality testing, medical diagnosis, and database query optimization. In quality testing, an item x manufactured by a factory is tested for defects. If it has at least k defects, it is discarded. In medical diagnosis, the item x is a patient; patients are diagnosed with a particular disease if they fail at least k out of n special medical tests. A database query may ask for all tuples x satisfying at least k of n given predicates (typically $k = 1$ or $k = n$).

For $k = 1$, standard and conservative k -of- n testing are the same. For $k > 1$, the conservative variant is relevant in a setting where, for items failing fewer than k tests, we need to know *which* tests they failed. For example, in quality testing, we may want to know which tests were failed by items failing fewer than k tests (i.e. those not discarded) in order to repair the associated defects.

Our focus is on the MAXTHROUGHPUT problem for k -of- n testing. Here the objective is to maximize the throughput of a system for k -of- n testing in a parallel setting where each test is performed by a separate “processor”. In this problem, in addition to the probabilities p_i , there is a *rate limit* r_i associated with the processor that performs test i , indicating that the processor can only perform tests on r_i items per unit time.

*Polytechnic Institute of NYU. This research is supported by the NSF Grant CCF-0917153. hstein@poly.edu

†Polytechnic Institute of NYU. This research supported by US Department of Education Grant P200A090157. ozgurozkan@gmail.com

‡Polytechnic Institute of NYU. This research is supported by a CIFellows Project postdoc, sponsored by NSF and the CRA. sellie@mac.com

¹In an alternative definition of k -of- n testing, the task is to determine whether at least k of the n tests have a value of 1. Symmetric results hold for this definition.

MAXTHROUGHPUT problems are closely related to MINCOST problems [6, 9]. In the MINCOST problem for k -of- n testing, in addition to the probabilities p_i , there is a cost c_i associated with performing the i^{th} test. The goal is to find a testing strategy (i.e. decision tree) that minimizes the expected cost of testing an individual item. There are polynomial-time algorithms for solving the MINCOST problem for standard k -of- n testing [1, 3, 10, 11].

Kodialam was the first to study the MAXTHROUGHPUT k -of- n testing problem, for the special case where $k = 1$ [8]. He gave a $\mathcal{O}(n^3 \log n)$ algorithm for the problem. The algorithm is combinatorial, but its correctness proof relies on polymatroid theory. Later, Condon et al. studied the problem, calling it “parallel pipelined filter ordering”. They gave two $\mathcal{O}(n^2)$ combinatorial algorithms, with direct correctness proofs [5].

Our Results. In this paper, we extend the previous work by giving a polynomial-time combinatorial algorithm for the MAXTHROUGHPUT problem for conservative k -of- n testing. Our algorithm can be implemented to run in time $\mathcal{O}(n^2)$, matching the running time of the algorithms of Condon et al. for 1-of- n testing. More specifically, the running time is $\mathcal{O}(n(\log n + k) + o)$, where o varies depending on the output representation used; the algorithm can be modified to produce different output representations. We discuss output representations below.

The MAXTHROUGHPUT problem for standard k -of- n testing appears to be fundamentally different from its conservative variant. We leave as an open problem the task of developing a polynomial time *combinatorial* algorithm for this problem. We show that previous techniques can be used to obtain a polynomial-time algorithm based on the ellipsoid method. This approach could also be used to yield an algorithm, based on the ellipsoid method, for the conservative variant.

Output Representation For the type of representation used by Condon et al. in achieving their $\mathcal{O}(n^2)$ bound, $o = \mathcal{O}(n^2)$. A more explicit representation has size $o = \mathcal{O}(n^3)$. We also describe a new, more compact output representation for which $o = \mathcal{O}(n)$.

In giving running times, we follow Condon et al. and consider only the time taken by the algorithm to produce the output representation. We note, however, that different output representations may incur different post-processing costs when we want to use them to implement the routings. For example, the compressed representation has $o = \mathcal{O}(n)$, but it requires spending $\mathcal{O}(n)$ time in the worst case to extract any permutation of megaprocessors stored by the megaprocessor representation. We can reduce this complexity to $\mathcal{O}(\log n)$ using persistent search trees [13]. In contrast, the explicit $\mathcal{O}(n^3)$ representation gives direct access to the permutations. In practice, the choice of the best output representation can vary depending on the application and the setting.

For ease of presentation, in our pseudocode we use the megaprocessor representation, which is also used by Condon et al. [5] in their Equalizing Algorithm.

2 Related Work

Deshpande and Hellerstein studied the MAXTHROUGHPUT problem for $k = 1$, when there are precedence constraints between tests [6]. They also showed a close relationship between the exact MINCOST and MAXTHROUGHPUT problems for k -of- n testing, when $k = 1$. Their results can be generalized to apply to testing of other functions.

Liu et al. [9] presented a generic, LP based method for converting an approximation algorithm for a MINCOST problem, into an approximation algorithm for a MAXTHROUGHPUT problem. Their results are not applicable to this paper, where we consider only exact algorithms.

Polynomial-time algorithms for the MINCOST problem for standard k -of- n testing were given by Salloum, Breuer, Ben-Dov, and Chang et al. [1, 3, 10–12].

The problem of how to best order a sequence of tests, in a sequential setting, has been studied in many different contexts, and in many different models. See for example [9] and [5] for a discussion of related work on the filter-ordering problem (i.e. the MINCOST problem for $k = 1$) and its variants, and [14] for a general survey of sequential testing of functions.

3 Problem Definitions

A *k-of-n testing strategy* for tests $1, \dots, n$ is a binary decision tree T that computes the *k-of-n* function, $f : \{0, 1\}^n \rightarrow \{0, 1\}$, where $f(x) = 1$ if and only if x contains fewer than k 0's. Each node of T is labeled by a variable x_i . The left child of a node labeled with x_i is associated with $x_i = 0$ (i.e., failing test i), and the right child with $x_i = 1$ (i.e., passing test i). Each $x \in \{0, 1\}^n$ corresponds to a root-to-leaf path in the usual way, and the label at the leaf is $f(x)$.

A *k-of-n* testing strategy T is *conservative* if, for each root-to-leaf path leading to a leaf labeled 1, the path contains exactly n non-leaf nodes, each labeled with a distinct variable x_i .

Given a permutation π of the n tests, we define $T_k^c(\pi)$ to be the conservative strategy described by the following procedure: *Perform the tests in order of permutation π until at least k 0's have been observed, or all tests have been performed, whichever comes first. Output 0 in the first case, and 1 in the second.*

Similarly, we define $T_k^s(\pi)$ to be the following standard *k-of-n* testing strategy: *Perform the tests in order of permutation π until at least k 0's have been observed, or until $n - k + 1$ 1's have been observed, whichever comes first. Output 0 in the first case, and 1 in the second.*

Each test i has an associated probability p_i , where $0 < p_i < 1$. Let D_p denote the product distribution on $\{0, 1\}^n$ defined by the p_i 's; that is, if x is drawn from D_p , then $\forall i, \Pr[x_i = 1] = p_i$ and the x_i are independent. We use $x \sim D_p$ to denote a random x drawn from D_p . In what follows, when we use an expression of the form $\Pr[\dots]$ involving an item x , we mean the probability with respect to D_p .

3.1 The MinCost problem

In the MINCOST problem for standard *k-of-n* testing, we are given n probabilities p_i and costs $c_i > 0$, for $i \in \{1, \dots, n\}$, associated with the tests. The goal is to find a *k-of-n* testing strategy T that minimizes the expected cost of applying T to a random item $x \sim D_p$. The cost of applying a testing strategy T to an item x is the sum of the costs of the tests along the root-to-leaf path for x in T .

In the MINCOST problem for conservative *k-of-n* testing, the goal is the same, except that we are restricted to finding a *conservative* testing strategy.

For example, consider the MINCOST 2-of-3 problem with probabilities $p_1 = p_2 = 1/2$, $p_3 = 1/3$ and costs $c_1 = 1$, $c_2 = c_3 = 2$. A standard testing strategy for this problem can be described procedurally as follows: *Given item x , begin by performing test 1. If $x_1 = 1$, follow strategy $T_2^s(\pi_1)$, where $\pi_1 = (2, 3)$. Else if $x_1 = 0$, follow strategy $T_1^s(\pi_2)$, where $\pi_2 = (3, 2)$.*

Under the above strategy, which can be shown to be optimal, evaluating $x = (0, 0, 1)$ costs 5, and evaluating $x' = (1, 1, 0)$ costs 3. The expected cost of applying this strategy to a random item $x \sim D_p$ is $3\frac{5}{6}$.

Because the MINCOST testing strategy may be a tree of size exponential in the number of tests, algorithms for the MINCOST problem may output a compact representation of the output strategy.

The Algorithm for the MinCost Problem. In the literature, versions of the MINCOST problem for 1-of- n testing are studied under a variety of different names, including pipelined filter ordering, selection ordering, and satisficing search (cf. [5]).

The following is a well-known, simple algorithm for solving the MINCOST problem for standard 1-of- n testing (see e.g. [7]): First, sort the tests in increasing order of the ratio $c_i/(1 - p_i)$. Next, renumber the tests, so that $c_1/(1 - p_1) < c_2/(1 - p_2) < \dots < c_n/(1 - p_n)$. Finally, output the sorted list $\pi = (1, \dots, n)$ of tests, which is a compact representation of the strategy $T_1^s(\pi)$ (which is the same as $T_1^c(\pi)$).

The above algorithm can be applied to the MINCOST problem for conservative *k-of-n* testing, simply by treating π as a compact representation of the conservative strategy $T_k^c(\pi)$. In fact, that strategy is optimal for conservative *k-of-n* testing: it has minimum expected cost among all conservative strategies. This follows immediately from a lemma of Boros et al. [2]².

²The lemma of Boros et al. actually proves that the corresponding decision tree is *0-optimal*. A decision tree computing a function f is 0-optimal if it minimizes the expected cost of testing an random x , *given that $f(x) = 0$* . In conservative *k-of-n* testing, where f is the *k-of-n* function, the cost of testing x is the same for all x such that $f(x) = 1$. Thus the problem of finding a min-cost conservative strategy for *k-of-n* testing is essentially equivalent to the problem of finding a 0-optimal decision tree

3.2 The MaxThroughput problem

The MAXTHROUGHPUT problem for k -of- n testing is a natural generalization of the MAXTHROUGHPUT problem for 1-of- n testing, first studied by Kodialam [8]. We give basic definitions and motivation here. For further information about this problem, including information relevant to its application in practical settings, see [4, 5, 8].

In the MAXTHROUGHPUT problem for k -of- n testing, as in the MINCOST problem, we are given the probabilities p_1, \dots, p_n associated with the tests. Instead of costs c_i for the tests, we are given *rate limits* $r_i > 0$. The MAXTHROUGHPUT problem arises in the following context. There is an (effectively infinite) stream of items x that need to be tested. Every item x must be assigned a strategy T that will determine which tests are performed on it. Different items may be assigned to different strategies. Each test is performed by a separate “processor”, and the processors operate in parallel. (Imagine a factory testing setting.) Item x is sent from processor to processor for testing, according to its strategy T . Each processor can only test one item at a time. We view the problem of assigning items to strategies as a flow-routing problem.

Processor O_i performs test i . It has rate limit (capacity) r_i , indicating that it can only process r_i items x per unit time.

The goal is to determine how many items should be assigned to each strategy T , per unit time, in order to maximize the number of items that can be processed per unit time, the throughput of the system. The solution must respect the rate limits of the processors, in that the expected number of items that need to be tested by processor O_i per unit time must not exceed r_i . We assume that tests behave according to expectation: if m items are tested by processor O_i per unit time, then mp_i of them will have the value 1, and $m(1 - p_i)$ will have the value 0.

Let \mathcal{T} denote the set of all k -of- n testing strategies and \mathcal{T}_c denote the set of all conservative k -of- n testing strategies. Formally, the MAXTHROUGHPUT problem for standard k -of- n testing is defined by the linear program below. The linear program defining the MAXTHROUGHPUT problem for conservative k -of- n testing is obtained by simply replacing the set of k -of- n testing strategies \mathcal{T} by the set of conservative k -of- n testing strategies \mathcal{T}_c .

We refer to a feasible assignment to the variables z_T in the LP below as a *routing*. We call constraints of type (1) *rate constraints*. The value of F is the *throughput* of the routing. We define $g(T, i)$ as the probability that test i will be performed on an item x that is tested using strategy T , when $x \sim D_p$. For $i \in \{1, \dots, n\}$, if $\sum_{T \in \mathcal{T}} g(T, i) z_T = r_i$, we say that the routing *saturates* processor O_i .

We will refer to the MAXTHROUGHPUT problems for standard and conservative k -of- n testing as the “SMT(k) problem” and the “CMT(k) problem”, respectively.

As a simple example, consider the following CMT(k) problem (equivalently, SMT(k) problem) instance, where $k = 1$ and $n = 2$: $r_1 = 1$, $r_2 = 2$, $p_1 = 1/2$, $p_2 = 1/4$. There are only two possible strategies, $T_1(\pi_1)$, where $\pi_1 = (1, 2)$, and $T_1(\pi_2)$, where $\pi_2 = (2, 1)$. Since all flow assigned to $T_1(\pi_1)$ is tested by O_1 , $g(T_1(\pi_1), 1) = 1$; this flow continues on to O_2 only if it passes test 1, which happens with probability $p_1 = 1/2$, so $g(T_1(\pi_1), 2) = 1/2$. Similarly, $g(T_1(\pi_2), 2) = 1$ while $g(T_1(\pi_2), 1) = 1/4$, since $p_2 = 1/4$. Consider the routing that assigns $F_1 = 4/7$ units of flow to strategy $T_1(\pi_1)$, and $F_2 = 12/7$ units to strategy $T_1(\pi_2)$. Then the amount of flow reaching O_1 is $4/7 \cdot g(T_1(\pi_1), 1) + 12/7 \cdot g(T_1(\pi_2), 1) = 1$, and the amount of flow reaching O_2 is $4/7 \cdot g(T_1(\pi_1), 2) + 12/7 \cdot g(T_1(\pi_2), 2) = 2$. Since $r_1 = 1$ and $r_2 = 2$, this routing saturates both processors. By the results of Condon et al. [5], it is optimal.

MaxThroughput LP:

Given $r_1, \dots, r_n > 0$ and $p_1, \dots, p_n \in (0, 1)$, find an assignment to the variables z_T , for all $T \in \mathcal{T}$, that maximizes

$$F = \sum_{T \in \mathcal{T}} z_T$$

subject to the constraints:

computing the k -of- n function. The lemma of Boros et al. also applies to a more general class of functions f that include the k -of- n functions.

- (1) $\sum_{T \in \mathcal{T}} g(T, i) z_T \leq r_i$ for all $i \in \{1, \dots, n\}$ and
- (2) $z_T \geq 0$ for all $T \in \mathcal{T}$

where $g(T, i)$ denotes the probability that test i will be performed on an item x that is tested using strategy T , when $x \sim D_p$.

4 The Algorithm for the CMT(k) problem

We begin with some useful lemmas. The algorithms of Condon et al. [5] for maximizing throughput of 1-of- n testing rely crucially on the fact that saturation of all processors implies optimality. We show that the same holds for conservative k -of- n testing.

Lemma 1. *Let R be a routing for an instance of the CMT(k) problem. If R saturates all processors, then it is optimal.*

Proof. Each processor O_i can test at most r_i items per unit time. Thus at processor O_i , there are at most $r_i(1 - p_i)$ tests performed that have the value 0. Let f denote the k -of- n function.

Suppose R is a routing achieving throughput F . Since F items enter the system per unit time, F items must also leave the system per unit time. An item x such that $f(x) = 0$ does not leave the system until it fails k tests. An item x such that $f(x) = 1$ does not leave the system until it has had all tests performed on it. Thus, per unit time, in the entire system, the number of tests performed that have the value 0 must be $F \cdot M$, where $M = (k \cdot \Pr[x \text{ has at least } k \text{ 0's}] + \sum_{j=0}^{k-1} j \cdot \Pr[x \text{ has exactly } j \text{ 0's}])$.

Since at most $r_i(1 - p_i)$ tests with the value 0 can occur per unit time at processor O_i , $F \cdot M \leq \sum_{i=1}^n r_i(1 - p_i)$. Solving for F , this gives an upper bound of $F \leq \sum_{i=1}^n r_i(1 - p_i)/M$ on the maximum throughput. This bound is tight if all processors are saturated, and hence a routing saturating all processors achieves the maximum throughput. \square

In the above proof, we rely on the fact that every routing with throughput F results in the same number of 0 test values being generated in the system per unit time. Note that this is not the case for *standard* testing, where the number of 0 test values generated can depend on the routing itself, and not just on the throughput of that routing. We now give a simple counterexample showing that, in fact, saturation does not imply optimality for the SMT(k) problem. Consider the MAXTHROUGHPUT 2-of-3 testing instance where $p_1 = 1/2, p_2 = 1/4, p_3 = 3/4$, and $r_1 = 2, r_2 = 1\frac{3}{4}, r_3 = 1\frac{3}{4}$.

The following is a 2-of-3 testing strategy: *Given item x , perform test 1. If $x_1 = 1$, follow strategy $T_1^s(\pi_1)$, where $\pi_1 = (2, 3)$. Else if $x_1 = 0$, follow strategy $T_1^s(\pi_2)$, where $\pi_1 = (3, 2)$.*

Assigning 2 units of flow to this strategy saturates the processors: O_1 is saturated since it receives the 2 units entering the system, O_2 is saturated since it receives $1 = 2 \cdot p_1$ units from O_1 and $3/4 = 2 \cdot p_3 \cdot (1 - p_1)$ items from O_1, O_2 . Similarly, O_3 is saturated since it receives $1 = 2 \cdot (1 - p_1)$ units from O_1 and $3/4 = 2 \cdot (1 - p_3) \cdot p_1$ units from O_1, O_3 .

We show that the routing is not optimal by giving a different routing with higher throughput. The routing uses two strategies. The first is as follows: *Given item x , perform test 1. If $x_1 = 1$, follow strategy $T_2^s(\pi_1)$, where $\pi_1 = (3, 2)$. Else, if $x_1 = 0$ follow strategy $T_1^s(\pi_1)$, where $\pi_2 = (2, 3)$.* The second strategy used by the routing is $T_2^s(\pi_3)$, where $\pi_3 = (3, 2, 1)$. Assigning $F = 1\frac{1}{2}$ units to the first strategy uses $1\frac{1}{2}$ units of the capacity of O_1 , $15/16 = 1\frac{1}{2} \cdot (1 - p_1) + 1\frac{1}{2} \cdot p_1 \cdot (1 - p_3)$ units of the capacity of O_2 , and $15/16 = 1\frac{1}{2} \cdot (1 - p_1) + 1\frac{1}{2} \cdot (1 - p_1) \cdot p_2$ of the capacity of O_3 . This leaves O_2 and O_3 with residual capacity more than $3/4 < 1 + 3/4 - 15/16$, and O_1 with residual capacity $1/2 = 2 - 1\frac{1}{2}$. We can then assign $3/4$ additional units to the second strategy without violating any of the rate constraints, for a routing with total throughput $2\frac{1}{4}$. (The resulting routing is not optimal, but illustrates our point.)

The routing produced by our algorithm for the CMT(k) problem uses only strategies of the form $T_k^c(\pi)$, for some permutation π of the tests (in terms of the LP, this means $z_T > 0$ only if $T = T_k^c(\pi)$ for some π). We call such a routing a *permutation routing*. We say that it has a *saturated suffix* if for some subset Q of

the processors (1) R saturates all processors in Q , and (2) for every strategy $T_k^c(\pi)$ used by R , the processors in Q (in some order) must form a suffix of π .

With this definition, and the above lemma, we are now able to generalize a key lemma of Condon et al. to apply to conservative k -of- n testing. The proof is essentially the same as theirs; we present it below for completeness.

Lemma 2. (*Saturated Suffix Lemma*) *Let R be a permutation routing for an instance of the $\text{CMT}(k)$ problem. If R has a saturated suffix, then R is optimal.*

Proof. If R saturates all processors, then the previous lemma guarantees its optimality. If not, let L denote the set of processors not saturated by R . Imagine that we removed the rate constraints for each processor in L . Let R' be an optimal routing for the resulting problem. We may assume that on any input x , R' performs the tests in L in some fixed arbitrary order (until and unless k tests with value 0 are obtained), prior to performing any tests in Q . This assumption is without loss of generality, because if not, we could modify R' to first perform the tests in L without violating feasibility, since the processors in L have no rate constraints, and performing their tests first can only decrease the load on the other processors. Thus the throughput attained by R' is $T_R \cdot \frac{1}{p_L}$, where T_R denotes the maximum throughput achievable just with the processors in Q , and p_L is the probability that a random x will have the value 0 for fewer than k of the tests in L (i.e. it will not be eliminated by the tests in L).

Routing R also routes flow first through L , and then through Q . Since it saturates the processors in Q , by the previous lemma, it achieves maximum possible throughput with those processors. It follows that R achieves the same throughput as R' , and hence is optimal for the modified instance where processors in L have no rate constraints. Since removing constraints can only increase the maximum possible throughput, it follows that R is also optimal for the original instance. \square

4.1 The Equal Rates Case

We begin by considering the $\text{CMT}(k)$ problem in the special case where the rate limits r_i are equal to some constant value r for all processors. Condon et al. presented a closed-form solution for this case when $k = 1$ [5]. The solution is a permutation routing that uses n strategies of the form $T_1(\pi)$. Each permutation π is one of the n left cyclic shifts of the permutation $(1, \dots, n)$. More specifically, for $i \in \{1, \dots, n\}$, let $\pi_i = (i, i+1, \dots, n, 1, 2, \dots, i-1)$, and let $T_i = T_1^c(\pi_i)$. The solution assigns $r(1 - p_{i-1})/(1 - p_1 \cdots p_n)$ units of flow to each T_i (where p_0 is defined to be p_n). By simple algebra, Condon et al. verified that the solution saturates all processors. Hence it is optimal.

The solution of Condon et al. is based on the fact that for the 1-of- n problem, assigning $(1 - p_{i-1})$ flow to each T_i equalizes the load on the processors. Surprisingly, this same assignment equalizes the load for the k -of- n problem as well. Using this fact, we obtain a closed-form solution to the $\text{CMT}(k)$ problem.

Lemma 3. *Consider an instance of the $\text{CMT}(k)$ problem. For $i \in \{1, \dots, n\}$, let T_i be as defined above. Let $X_{a,b} = \sum_{\ell=a}^b (1 - x_\ell)$ and let $\alpha = \sum_{t=1}^k \Pr[X_{1,n} \geq t]$. Any routing that assigns a total of t units of flow to the strategies T_i , such that the fraction of the total that is assigned to each T_i is $(1 - p_{i-1})/\sum_{j=1}^n (1 - p_{j-1})$, will cause each processor's residual capacity to be reduced by $t\alpha/\sum_{j=1}^n (1 - p_j)$ units. If all processors have the same rate limit r , then the routing that assigns $r(1 - p_{i-1})/\alpha$ units of flow to strategy T_i saturates all processors.*

Proof. We begin by considering the routing in which $(1 - p_{i-1})$ units of flow are assigned to each T_i . Consider the question of how much flow arrives per unit time at processor O_1 , under this routing. For simplicity, assume now that $k = 2$. Thus as soon as an item has failed 2 tests, it is discarded. Let $q_i = (1 - p_i)$.

Of the q_n units assigned to strategy T_1 , all q_n arrive at processor O_1 . Of the q_{n-1} units assigned to strategy T_n , all q_{n-1} arrive at processor O_1 , since they can fail either 0 or 1 test (namely test n) beforehand.

Of the q_{n-2} units assigned to strategy T_{n-1} , the number reaching processor O_1 is $q_{n-2}\beta_{n-1}$, where β_{n-1} is the probability that an item fails either 0 or 1 of tests $n-1$ and n . Therefore, $\beta_{n-1} = 1 - q_{n-1}q_n$.

More generally, for $i \in \{1, \dots, n\}$, of the q_{i-1} units assigned to T_i , the number reaching processor O_1 is $q_{i-1}\beta_i$, where β_i is the probability that a random item fails a total of 0 or 1 of tests $i, i+1, \dots, n$. Thus, $\beta_i = \Pr[X_{i,n} = 0] + \Pr[X_{i,n} = 1]$. It follows that the total flow arriving at processor O_1 is $\sum_{i=1}^n (q_{i-1}\Pr[X_{i,n} = 0]) + \sum_{i=1}^n (q_{i-1}\Pr[X_{i,n} = 1])$.

Consider the second summation, $\sum_{i=1}^n (q_{i-1}\Pr[X_{i,n} = 1])$. We claim that this summation is equal to $\Pr[X_{1,n} \geq 2]$, which is the probability that x has *at least* two x_i 's that are 0. To see this, consider a process where we observe the value of x_n , then the value of x_{n-1} and so on down towards x_1 , stopping if and when we have observed exactly two 0's. The probability that we will stop at some point, having observed two 0's, is clearly equal to the probability that x has *at least* two x_i 's that are set to 0. The condition $\sum_{j=i}^n (1 - x_j) = 1$ is satisfied when exactly 1 of x_n, x_{n-1}, \dots, x_i has the value 0. Thus $q_{i-1}\Pr[X_{i,n} = 1]$ is the probability that we observe exactly one 0 in x_n, \dots, x_i , and then we observe a second 0 at x_{i-1} . That is, it is the probability that we stop after observing x_{i-1} . Since the second summation takes the sum of $q_{i-1}\Pr[X_{i,n} = 1]$ over all i between 1 and n , the summation is precisely equal to the probability of stopping at some point in the above process, having seen two 0's. This proves the claim.

An analogous argument shows that the first summation, $\sum_{i=1}^n (q_{i-1}\Pr[X_{i,n} = 0])$, is equal to $\Pr[X_{1,n} \geq 1]$.

It follows that the amount of flow reaching processor O_1 is $\Pr[X_{1,n} \geq 1] + \Pr[X_{1,n} \geq 2]$. This expression is symmetric in the processor numbers, so the amount of flow reaching every O_i is equal to this value. Thus the above routing causes all processors to receive the same amount of flow.

Scaling each assignment in the above routing by a constant factor scales the amount of flow reaching each processor by the same factor. In the above routing, the fraction of total flow assigned to each T_i is $q_{i-1} / \sum_{j=1}^n q_j$, so each unit of input flow sent along the T_i results in each processor receiving $(\Pr[X_{1,n} \geq 1] + \Pr[X_{1,n} \geq 2]) / \sum_{j=1}^n q_j$ units. Thus any routing that assigns a total of t units of flow to the strategies T_i , such that the fraction assigned to each T_i is $q_{i-1} / \sum_{j=1}^n q_j$, will cause each processor to receive $t(\Pr[X_{1,n} \geq 1] + \Pr[X_{1,n} \geq 2]) / \sum_{j=1}^n q_j$ units.

Thus if all processors have the same rate limit r , the routing that assigns $rq_{i-1} / (\Pr[X_{1,n} \geq 1] + \Pr[X_{1,n} \geq 2])$ units to each strategy T_i will saturate all processors.

The above argument for $k = 2$ can easily be extended to arbitrary k . The corresponding proportional distribution of flow for arbitrary k assigns a $q_{i-1} / \sum_{j=1}^n q_j$ fraction of the total flow to strategy T_i , and each unit of input flow sent along the T_i according to these proportions results in $\alpha / \sum_{j=1}^n q_j$ units reaching each processor. The saturating routing for arbitrary k , when all processors have rate limit r , assigns rq_{i-1} / α units of flow to strategy T_i . \square

4.2 The Equalizing Algorithm of Condon et al.

Our algorithm for the CMT(k) problem is an adaptation of one of the two MAXTHROUGHPUT algorithms, for the special case where $k = 1$, given by Condon et al. [5]. We begin by reviewing that algorithm, which we will call the *Equalizing Algorithm*. Note that when $k = 1$, it only makes sense to consider strategies that are permutation routings, since an item can be discarded as soon as it fails a single test.

Consider the CMT(k) problem for $k = 1$. View the problem as one of constructing a flow of items through the processors. The capacity of each processor is its rate limit, and the amount of flow sent along a permutation π (i.e., assigned to strategy $T_1^c(\pi)$) is equal to the number of items sent along that path per unit time. Sort the tests by their rate limits, and re-number them so that $r_n \geq r_{n-1} \geq \dots \geq r_1$. Assume for the moment that all rate limits r_i are distinct.

The Equalizing Algorithm constructs a flow incrementally as follows. Imagine pushing flow along the single permutation $(n, \dots, 1)$. Suppose we continuously increase the amount of flow being pushed, beginning from zero, while monitoring the "residual capacity" of each processor, i.e., the difference between its rate limit and the amount of flow it is already receiving. (For the moment, do not worry about exceeding the rate limit of a processor.)

Consider two adjacent processors, i and $i-1$. As we increase the amount of flow, the residual capacity of each decreases continuously. Initially, at zero flow, the residual capacity of i is greater than the residual capacity of $i-1$. It follows by continuity that the residual capacity of i cannot become less than the residual

capacity of $i - 1$ without the two residual capacities first becoming equal. We now impose the following stopping condition: increase the flow sent along permutation $(n, \dots, 1)$ until either (1) some processor becomes saturated, or (2) the residual capacities of at least two of the processors become equal. The second stopping condition ensures that when the flow increase is halted, permutation $(n, \dots, 1)$ still orders the processors in decreasing order of their residual capacities. (Algorithmically, we do not increase the flow continuously, but instead directly calculate the amount of flow which triggers the stopping condition.)

If stopping condition (1) above holds when the flow increase is stopped, then the routing can be shown to have a saturated suffix, and hence it is optimal.

If stopping condition (2) holds, we keep the current flow, and then augment it by solving a new MAX-THROUGHPUT problem in which we set the rate limits of the processors to be equal to their residual capacities under the current flow (their p_i 's remain the same).

We solve the new MAXTHROUGHPUT problem as follows. We group the processors into equivalence classes according to their rate limits. We then replace each equivalence class with a single megaprocessor, with a rate limit equal to the residual capacities of the constituent processors, and probability p_i equal to the product of their probabilities. We then essentially apply the procedure for the case of distinct rate limits to the megaprocessors. The one twist is the way in which we translate flow sent through a megaprocessor into flow sent through the constituent processors of that megaprocessor; we route the flow through the constituent processors so as to equalize their load. We accomplish this by dividing the flow proportionally between the cyclic shifts of a permutation of the processors, using the proportional allocation of Lemma 3. We thus ensure that the processors in each equivalence class continue to have equal residual capacity. Note that, under this scheme, the residual capacity of a processor in a megaprocessor may decrease more slowly than it would if all flow were sent directly to that processor (because some flow may first be filtered through other processors in the megaprocessor) and this needs to be taken into account in determining when the stopping condition is reached.

We illustrate the Equalizing Algorithm on the following CMT(k) problem where $k = 1$ and $n = 3$ (since $k = 1$ this is also an SMT(k) problem, where $k = 1$ and $n = 3$). Suppose we have 3 processors, O_1, O_2, O_3 with rate limits $r_1 = 3, r_2 = 14$, and $r_3 = 18$, and probabilities $p_1 = 1/8, p_2 = 1/2$ and $p_3 = 1/3$. When flow is sent along O_3, O_2, O_1 , after 6 units of flow is sent we achieve a stopping condition with O_3 and O_2 having the same residual capacity of 12; the residual capacity of O_1 is 2.

Our algorithm then performs a recursive call where the processors O_3 and O_2 are combined into a megaprocessor $O_{2,3}$ with associated probability $p_{2,3} = 1/2 \cdot 1/3 = 1/6$. Within megaprocessor $O_{2,3}$, flow will be routed by sending $3/7$ of it along permutation O_3, O_2 , and the remaining $4/7$ along permutation O_2, O_3 ; we observe that for one unit of flow sent through $O_{2,3}$ the amount of capacity used by each processor is $3/7 + 2/7 = 5/7$. Using this internal routing for megaprocessor $O_{2,3}$, the algorithm sends flow along $O_{2,3}, O_1$; after 12 units of flow, we reach a stopping condition when O_1 is saturated. Even though O_2 and O_3 are not saturated (they have $12 - 12 \cdot 5/7$ residual capacity left) the flows constructed as described provide optimal throughput.

The Equalizing Algorithm, implemented in a straightforward way, outputs a representation of the resulting routing that consists of a sequence of pairs of the form $((E_m, \dots, E_1), \hat{t})$, one for each recursive call. We call this a *megaprocessor representation*. The list (E_m, \dots, E_1) represents the permutation of megaprocessors E_i along which flow is sent during that call. Each E_i is given by the subset of original processors contained in it, and $\hat{t} > 0$ is a real number that denotes the amount of flow to be sent along (E_m, \dots, E_1) . Of course, flow coming into each megaprocessor should be routed so as to equalize the load on each of its constituent processors. The size of this representation is $\mathcal{O}(n^2)$. Interpreted in a straightforward way, the representation corresponds to a routing that sends flow along an exponential number of different permutations of the original processors.

Condon et al. describe a combinatorial method to reduce the number of such permutations used to be $\mathcal{O}(n^2)$ [5]. After such a reduction, the output can be represented explicitly as a set of $\mathcal{O}(n^2)$ pairs of the form (π, t) , one for each permutation π that is used, indicating that $t > 0$ amount of flow should be sent along permutation π . We call such a representation a *permutation representation*. The size of this permutation representation, given explicitly, is $\mathcal{O}(n^3)$. (Hellerstein and Deshpande describe a linear algebraic method for

reducing the number of permutations to be at most n , yielding an explicit representation of size $\mathcal{O}(n^2)$, but at the cost of higher time complexity [6].)

We also describe a variant of the megaprocessor representation called the *compressed representation*, where the algorithm outputs only the first permutation explicitly, and the outputs the sequence of merges, yielding a representation of size $\mathcal{O}(n)$.

4.3 An Equalizing Algorithm for the CMT(k) problem

In this section, we prove the following Theorem by presenting an algorithm. We will give an outline of the algorithm as well as its pseudocode. We will then describe how to achieve the running time stated in the Theorem.

Theorem 4. *There is a combinatorial algorithm for solving the CMT(k) problem that can be implemented to run in time $\mathcal{O}(n(\log n + k) + o)$, where the value of o depends on the output representation. For the megaprocessor representation, $o = \mathcal{O}(n^2)$, for the permutation representation, $o = \mathcal{O}(n^3)$, and for the compressed representation, $o = \mathcal{O}(n)$.*

Algorithm Outline We extend the Equalizing Algorithm of Condon et al., to apply to arbitrary values of k . Again, we will push flow along the permutation of the processors $(n, \dots, 1)$ (where $r_n \geq r_{n-1} \geq \dots \geq r_1$) until one of the two stopping conditions is reached: (1) a processor is saturated, or (2) two processors have equal residual capacity. Here, however, we do not discard an item until it has failed k tests, rather than discarding it as soon as it fails one test. To reflect this, we divide the flow into k different types, numbered 0 through $k - 1$, depending on how many tests its component items have failed. Flow entering the system is all of type 0.

When m units of flow of type τ enters a processor O_i , $p_i m$ units pass test i , and $(1 - p_i)m$ units fail it. So, if $\tau < k - 1$, then of the m incoming units of type τ , $(1 - p_i)m$ units will exit processor O_i as type $\tau + 1$ flow, and $p_i m$ will exit as type τ flow. Both types will be passed on to the next processor in the permutation, if any. If $\tau = k - 1$, then $p_i m$ units will exit as type τ flow and be passed on to the next processor, and the remaining $(1 - p_i)m$ will be discarded.

Algorithmically, we need to calculate the minimum amount of flow that triggers a stopping condition. This computation is only slightly more complicated for general k than it is for $k = 1$. The key is to compute, for each processor O_i , what fraction of the flow that is pushed into the permutation will actually reach processor O_i (i.e. we need to compute the quantity $g(T_k^c(\pi), i)$ in the LP.)

If stopping condition (2) holds, we keep the current flow, and augment it by solving a new MAXTHROUGHPUT problem in which we set the rate limits of the processors to be equal to their residual capacities under the current flow (their p_i 's remain the same). To solve the new MAXTHROUGHPUT problem, we again group the processors into equivalence classes according to their rate limits, and replace each equivalence class with a single megaprocessor, with a rate limit equal to the rate limit of the constituent processors, and probability p_i equal to the product of their probabilities.

We then want to apply the procedure for the case of distinct rate limits to the megaprocessors. To do this, we need to translate flow sent into a megaprocessor into flow sent through the constituent processors of that megaprocessor, so as to equalize their load. We do this translation separately for each type of flow entering the megaprocessor. Note that flow of type τ must be discarded as soon as it fails an additional $k - \tau$ tests. We therefore send flow of type τ into the constituent processors of the megaprocessor according to the proportional allocation of Lemma 3 for $(k - \tau)$ -of- n' testing, where n' is the number of constituent processors of the megaprocessor. We also need to compute how much flow of each type ends up leaving the megaprocessor (some of the incoming flow of type τ entering the megaprocessor may, for example, become outgoing flow of type $\tau + n'$), and how much its residual capacity is reduced by the incoming flow.

We give a more detailed description of the necessary computations in the pseudocode, which we discuss next. However, the pseudocode does not contain all the implementation details, and is not optimized for efficiency. It also gives the output using a megaprocessor representation. Following presentation of the

pseudocode, we discuss how to implement it to achieve the running times stated in Theorem 4 for the different output representations.

Pseudocode The main part of the pseudocode is presented below as Algorithm 1. The following information will be helpful in understanding it.

At each stage of the algorithm, the processors are partitioned into equivalence classes. The processors in each equivalence class constitute a megaprocessor. Each equivalence class consists of a contiguous subsequence of processors in the sorted sequence O_n, \dots, O_2, O_1 . We use m to denote the number of megaprocessors (equivalence classes). The processors in each equivalence class all have the same residual capacity. In Step 1 of the algorithm, we partition the processors into equivalence classes according to their rate limits; two processors are in the same equivalence class if and only if they have the same rate limit. We use E_i to denote both the i^{th} equivalence class and the i^{th} megaprocessor. In some of our examples, we denote a megaprocessor containing processors $\{O_i, O_{i+1}, \dots, O_j\}$ by $O_{i,i+1,\dots,j}$.

In Step 2, we compute the amount of flow \hat{t} that triggers one of the two stopping conditions. In order to do this, we need to know the rate at which the residual capacity of each processor within an equivalence class E_i will be reduced when flow is sent down the megaprocessors in the order E_m, \dots, E_1 . We use $\xi(i)$ to denote the amount by which the residual capacity of the processors in E_i is reduced when one unit of flow is sent in that order.

The equation for $\xi(i)$ follows from the preceding lemmas and discussion. We use $f_j(z)$ to denote the amount of flow of type j that would reach processor z , if one unit of flow were sent down the permutation O_n, \dots, O_1 , where these are the original processors, not the megaprocessors. This is precisely equal to the probability that random item x has exactly j 0's in tests $n, \dots, z+1$. We compute the value of $f_j(z)$ for all z and j in a separate initialization routine, given below. The key here is noticing that if you send one unit of flow down the megaprocessors E_m, \dots, E_1 , the amount of flow reaching megaprocessor E_i is precisely $f_j(c(i))$, where $c(i)$ is the highest index of a processor in E_i ; the amount of flow reaching the megaprocessor depends only on how many 0's have been encountered in test $n, \dots, c(i)+1$, and not on the order used to perform those tests.

The quantity \hat{t}_1 is the amount of flow sent down E_m, \dots, E_1 that would cause saturation of the processors in E_1 . The quantity \hat{t}_2 is the minimum amount of flow sent down E_m, \dots, E_1 that would cause the residual capacities of two megaprocessors to equalize. The stopping condition holds at the minimum of these two quantities.

MAXTHROUGHPUT Initialization

```

 $f_j(z) \leftarrow 0, \forall z \in \{1, \dots, n\}, \forall j \in \{0, \dots, k-1\};$ 
 $f_0(1) \leftarrow 1;$ 
for ( $z \leftarrow 2; z \leq n; z \leftarrow z+1$ ) do
  for ( $j \leftarrow 0; j \leq k-1; j \leftarrow j+1$ ) do
     $f_j(z) \leftarrow q_{z-1}f_{j-1}(z-1) + p_{z-1}f_j(z-1);$ 
return SolveMaxThroughput( $p_1, \dots, p_n, r_1, \dots, r_n$ );

```

Example We illustrate our algorithm for the CMT(k) problem on the following example. Let $k = 2$ and $n = 4$. Suppose the probabilities are $p_1 = p_2 = p_3 = 1/2$, $p_4 = 3/4$, and the rate limits are $r_1 = r_2 = 12$, $r_3 = r_4 = 10$.

Our algorithm first combines processors with same rate limits into megaprocessors; thus we combine O_1 and O_2 into megaprocessor $O_{1,2}$ with rate limit 12. It routes flow through this megaprocessor by sending a $1/2$ fraction of the flow in the order O_1, O_2 , and sending the other $1/2$ fraction in the order O_2, O_1 . Similarly, O_3 and O_4 have the same rate limit, so they are combined into a megaprocessor $O_{3,4}$ with rate limit 10, where a $1/3$ fraction of the flow is sent along O_3, O_4 , and the other $2/3$ fraction is sent along O_4, O_3 .

Our megaprocessor $O_{1,2}$ has a higher rate limit than $O_{3,4}$, consequently our algorithm routes flow in the order $O_{1,2}, O_{3,4}$. We now show that the stopping condition is reached after sending 6 units of flow along this

Algorithm 1 SolveMaxThroughput($p_1, \dots, p_n, r_1, \dots, r_n$)

Input: n selectivities p_1, \dots, p_n ; n rate limits $r_1 \leq \dots \leq r_n$

Output: representation of solution to the MAXTHROUGHPUT problem for the given input parameters

```
1. // form the equivalence classes  $E_m, \dots, E_1$ ;  
   Let  $1 \leq \ell_1 < \dots < \ell_{m+1} = n + 1$  such that, for all  $y, y' \in [\ell_{i-1}, \ell_i)$  and  $z, z' \in [\ell_i, \ell_{i+1})$ , where  $i \in [2, m]$ ,  
   we have  $r_y = r_{y'} < r_z = r_{z'}$   
   Then, for  $i \in [1, m]$ ,  $E_i = \{O_z \mid \ell_i \leq z < \ell_{i+1}\}$ , and  $R_i \leftarrow r_{\ell_i}$ .  
  
2. // calculate  $\hat{t}$  using the following steps;  
   for ( $i \leftarrow 1; i \leq m; i \leftarrow i + 1$ ) do  
      $c(i) \leftarrow$  highest index of a processor in  $E_i$ ;  
      $b(i) \leftarrow$  lowest index of a processor in  $E_i$ ;  
     Recall that  $X_{a,b} = \sum_{\ell=a}^b (1 - x_\ell)$   
      $\xi(i) \leftarrow \sum_{j=0}^{k-1} f_j(c(i)) \cdot \left( \sum_{v=1}^{k-j} \Pr[X_{b(i),c(i)} \geq v] \right) / \sum_{t=b(i)}^{c(i)} (1 - p_t)$ ;  
      $\hat{t}_1 \leftarrow \frac{R_1}{\xi(1)}$ ;  
      $\hat{t}_2 \leftarrow \min_{i \in [2, \dots, m]} \left( \frac{R_i - R_{i-1}}{\xi(i) - \xi(i-1)} \right)$ ;  
      $\hat{t} \leftarrow \min(\hat{t}_1, \hat{t}_2)$ ;  
  
3. // calculate the residual capacity for each processor  $O_\ell$ ;  
   for ( $\ell \leftarrow 1; \ell \leq n; \ell \leftarrow \ell + 1$ ) do  
      $j \leftarrow$  index of the equivalence class  $E_j$  containing processor  $O_\ell$ ;  
      $r'_\ell \leftarrow r_\ell - \xi(j)\hat{t}$ ;  
  
4. // store new flow and recurse if needed  
    $K \leftarrow ((E_m, \dots, E_1), \hat{t})$ ;  
   if ( $r'_1 == 0$ ) then // residual capacity of equivalence class  $E_1$  is 0  
     return  $K$ ;  
   else  
      $K' \leftarrow \text{SolveMaxThroughput}(p_1, \dots, p_n, r'_1, \dots, r'_n)$ ;  
     return  $K \circ K'$ ; // i.e. the concatenation of  $K$  and  $K'$ 
```

route.

The 6 units of flow decreased the capacity of processors O_1 , and O_2 in $O_{1,2}$ by 6, since $k = 2$ and thus flow cannot be discarded before it has been subject to at least two tests.

We now calculate the reduction of capacity in O_3 and O_4 caused by the 6 units of flow sent through $O_{1,2}, O_{3,4}$. Flow leaving $O_{1,2}$ has a $1/4$ probability of having failed both processors in $O_{1,2}$ and exiting the system; for flow that stays in the system to be tested by $O_{3,4}$, it has a $1/4$ chance of having passed the test of both processors; it has a $1/2$ chance of having passed the test of one processor and having failed the test of the other processor. Thus, of the 6 units of flow sent into $O_{1,2}$, $1/4 \cdot 6 = 3/2$ units are passed on to $O_{3,4}$ as type 0 flow, and $1/2 \cdot 6 = 3$ units of flow are passed on to $O_{3,4}$ as type 1 flow.

Of the $3/2$ units of type 0 flow, entering $O_{3,4}$, all of it must undergo both test 3 and test 4, since flow is not discarded until it has failed two tests. Thus that flow reduces the capacity of both O_3 and O_4 by $3/2$ units.

Of the 3 units of type 1 flow entering $O_{3,4}$, $1/3$ is tested first by O_3 , and then by O_4 only if it passes test 3 (which it does with probability $1/2$). The remaining $2/3$ is tested first by O_4 , and then by O_3 only if it passes test 4 (which it does with probability $3/4$). Thus of the 3 units of type 1 flow, $3 \cdot (1/3 + 2/3 \cdot 3/4) = 5/2$ units reach O_3 , and $3 \cdot (2/3 + 1/3 \cdot 1/2) = 5/2$ units reach O_4 . Hence the $3 + 3/2$ total units of flow entering $O_{3,4}$ reduce the capacities of both O_3 and O_4 by $5/2 + 3/2 = 4$.

We have thus shown that the 6 units of flow sent first to $O_{1,2}$ and then to $O_{3,4}$, cause the residual capacities of O_1 and O_2 to be $12 - 6 = 6$, and the residual capacities of O_3 and O_4 to be $10 - 4 = 6$. Thus the residual capacities of all processors equalize, as claimed.

At this point our algorithm constructs a new megaprocessor, by combining the processors in $O_{1,2}$ with the processors in $O_{3,4}$. All the processors in the resulting megaprocessor, $O_{1,2,3,4}$, have a residual capacity of 6. Using the proportional allocation of Lemma 3 to route flow sent into $O_{1,2,3,4}$, we assign $1/7$ of the flow into $O_{1,2,3,4}$ to permutation $\pi_1 = \{1, 2, 3, 4\}$, $2/7$ to permutation $\pi_2 = \{2, 3, 4, 1\}$, $2/7$ to permutation $\pi_3 = \{3, 4, 1, 2\}$, and $2/7$ to permutation $\pi_4 = \{4, 1, 2, 3\}$. By sending a total of 7 units of flow through $O_{1,2,3,4}$ according to this allocation, we send 1, 2, 2, and 2 units respectively along the four permutations, achieving the saturating routing given in Lemma 3.

Our final routing achieves a throughput of $6 + 7 = 13$ which is optimal.

Achieving the running time. Let us first consider the running time of the algorithm excluding the computation of $\xi(i)$ and the time it takes to construct the output representation K . It is easy to see that the algorithm makes at most $n - 1$ recursive calls, because megaprocessors can only be merged a total of $n - 1$ times. Excluding the computation of $\xi(i)$, the time spent in each recursive call is clearly $\mathcal{O}(n)$. However, we can implement the algorithm so as to ensure this time is $\mathcal{O}(\log n)$, as follows. First, the maintenance of the equivalence classes can be handled in $\mathcal{O}(1)$ time per merge by simply taking a union of the sets of adjacent processors in each megaprocessor, instead of recomputing these sets from scratch.

Second, we do not need to compute the residual capacity of each megaprocessor at every recursive call. In fact, for all megaprocessors except the first one, we only need enough information about its residual capacity to allow us to compute $\hat{t}_2 \leftarrow \min_{i \in [2, \dots, m]} \left(\frac{R_i - R_{i-1}}{\xi(i) - \xi(i-1)} \right)$. This suggests that for each megaprocessor i where $i \geq 2$, we keep the quantity Q_i , where $Q_i = \left(\frac{R_i - R_{i-1}}{\xi(i) - \xi(i-1)} \right)$ instead of R_i . The megaprocessors can be stored in a priority queue, according to their Q_i values.

Consider any i where E_i or E_{i-1} are not involved in a merge. Then

$$\frac{(R_i - \xi(i)\hat{t}) - (R_{i-1} - \xi(i-1)\hat{t})}{\xi(i) - \xi(i-1)} = \frac{R_i - R_{i-1}}{\xi(i) - \xi(i-1)} - \hat{t}.$$

Thus following the merge, Q_i is decreased by the same amount \hat{t} for all such i . Therefore, instead of updating the Q_i for these i in the priority queue, we can keep their current values, and maintain the sum of the \hat{t} values computed so far; this can be subtracted from Q_i if its updated value is needed. We do need to remove the two merged megaprocessors from the priority queue, insert the information about the resulting new megaprocessor, and update the Q_i values for megaprocessors i such that E_i or E_{i-1} were involved in

a merge. Note that we need to change the Q_i values for such megaprocessors due to the change in the $\xi(\cdot)$ value of the newly formed megaprocessor. The above operations can be performed in time $\mathcal{O}(\log n)$ time per merge, using the priority queue.

Therefore, the running time of the algorithm excluding the computation of $\xi(i)$ is $\mathcal{O}(n \log n + o)$ where o is the time required to construct the output. In the pseudocode, the output is computed using the megaprocessor representation. Since there are at most n recursive calls, there are at most n pairs $((E_m, \dots, E_1), \hat{t})$ in the output, and therefore $o = \mathcal{O}(n^2)$.

If one chose to convert this representation to a permutation representation, using the combinatorial method of Condon et al. [5], then the value of o would be $\mathcal{O}(n^3)$.

Consider instead the following more compact output representation, which we call the compressed representation. Suppose the algorithm outputs the initial permutation, then outputs the sequence of merges performed, together with the \hat{t} values associated with the merges. In this case, we have $o = \mathcal{O}(n)$.

We will next show that the computation of $\xi(i)$ throughout the algorithm can be performed in $\mathcal{O}(nk)$ total time.

Computing $\xi(i)$. Let $E_k^{(i)}$ be the k^{th} megaprocessor in the recursive call associated with the i^{th} merge. Let $b(i, k)$ be the lowest index of a processor in $E_k^{(i)}$, and let $c(i, k)$ be the highest index of a processor in $E_k^{(i)}$. Let $|E_k^{(i)}|$ denote the *size* of that megaprocessor, that is, the number of processors in it. Thus $|E_k^{(i)}| = c(i, k) - b(i, k) + 1$.

Let $h(i)$ and $h(i) + 1$ be the indices of the megaprocessors merged by the i^{th} merge (i.e. $E_{h(i)}^{(i)}$ and $E_{h(i)+1}^{(i)}$ are the megaprocessors merged by the i^{th} merge).

Observe that at iteration i after megaprocessor $E_{h(i)}^{(i)}$ is merged with $E_{h(i)+1}^{(i)}$ we only recompute $\xi(h(i))$. After the merge, we need to compute

$$\xi(h(i)) = \sum_{j=0}^{k-1} f_j(c(i, h(i) + 1)) \cdot \left(\sum_{v=1}^{k-j} \Pr[X_{b(i, h(i)), c(i, h(i)+1)} \geq v] \right) / \sum_{t=b(i, h(i))}^{c(i, h(i)+1)} (1 - p_t)$$

Consider the denominator $\sum_{t=b(i, h(i))}^{c(i, h(i)+1)} (1 - p_t)$ in the above expression. It is the sum of the failure probabilities of all processors contained in $E_{h(i)}^{(i)}$ and $E_{h(i)+1}^{(i)}$. To enable this computation to be performed in constant time per recursive call, we simply store, with each megaprocessor, the sum of the failure probabilities of all processors in it. In each recursive call, it only takes constant time to update this information. Recall that $f_j(c(i, h(i) + 1))$ for all $j \in \{0, \dots, k-1\}$ are computed in the initialization procedure. Let

$$D_j = \sum_{v=1}^{k-j} \Pr[X_{b(i, h(i)), c(i, h(i)+1)} \geq v].$$

Given D_j for $j \in \{0, \dots, k-1\}$, we can compute $\xi(h(i))$ in $\mathcal{O}(k)$ time. Observe that

$$D_j = D_{j+1} + \Pr[X_{b(i, h(i)), c(i, h(i)+1)} \geq k - j].$$

Therefore, given $\Pr[X_{b(i, h(i)), c(i, h(i)+1)} \geq v]$ for $v \in \{1, \dots, k\}$, we can compute $\{D_0, \dots, D_{k-1}\}$ in $\mathcal{O}(k)$ time. Finally, observe that

$$\Pr[X_{b(i, h(i)), c(i, h(i)+1)} \geq v] = \Pr[X_{b(i, h(i)), c(i, h(i)+1)} \geq v+1] + \Pr[X_{b(i, h(i)), c(i, h(i)+1)} = v].$$

Therefore, given $\Pr[X_{b(i, h(i)), c(i, h(i)+1)} = v]$ for $v \in \{1, \dots, k\}$, we can compute $\Pr[X_{b(i, h(i)), c(i, h(i)+1)} \geq v]$ for all $v \in \{1, \dots, k\}$ in $\mathcal{O}(k)$ time. To enable these computations, we store, with each megaprocessor, the values $\Pr[X_{b,c} = v]$ for all $v \in \{1, \dots, k\}$, where b and c are respectively the lowest and highest indices of the processors contained in that megaprocessor. We will analyze below the total cost of keeping these values updated.

We denote by \mathcal{C} the total cost of computing $\xi(\cdot)$ throughout the algorithm, using the implementation described. Since we will have to compute $\xi(\cdot)$ at most n times throughout the algorithm, by the arguments above, \mathcal{C} is bounded by $\mathcal{O}(nk)$ plus the cost of computing $\Pr[X_{b(i,h(i)),c(i,h(i)+1)} = v]$ for all $i \in \{1, \dots, n-1\}$ and $v \in \{0, \dots, k\}$. Let us denote the cost of computing $\Pr[X_{b(i,h(i)),c(i,h(i)+1)} = v]$ by $\mathcal{C}_{i,v}$. Therefore, $\mathcal{C} = \mathcal{O}(nk) + \sum_{i=1}^{n-1} \sum_{v=0}^k \mathcal{C}_{i,v}$. We show that $\mathcal{C} = \mathcal{O}(nk)$ by proving $\sum_{i=1}^{n-1} \sum_{v=0}^k \mathcal{C}_{i,v} = \mathcal{O}(nk)$.

Lemma 5. $\sum_{i=1}^{n-1} \sum_{v=0}^k \mathcal{C}_{i,v} = \mathcal{O}(nk)$.

Proof. Let $E_{\min}^{(i)} = E_{h(i)}^{(i)}$ if $|E_{h(i)}^{(i)}| \leq |E_{h(i)+1}^{(i)}|$ and $E_{\min}^{(i)} = E_{h(i)+1}^{(i)}$ otherwise. Recall that when we need to compute $\Pr[X_{b(i,h(i)),c(i,h(i)+1)} = v]$, we have already computed and stored $\Pr[X_{b(i,h(i)),c(i,h(i))} = v]$ and $\Pr[X_{b(i,h(i)+1),c(i,h(i)+1)} = v]$ for all $v \in \{0, \dots, k\}$.

Since $\Pr[X_{b,c} = v] = 0$ for any b, c if $v > c - b + 1$, we can compute $\Pr[X_{b(i,h(i)),c(i,h(i)+1)} = v]$ using the following equality:

$$\Pr[X_{b(i,h(i)),c(i,h(i)+1)} = v] = \sum_{j=\max(0, v-|E_{h(i)+1}^{(i)}|)}^{\min(v, |E_{h(i)}^{(i)}|)} \Pr[X_{b(i,h(i)),c(i,h(i))} = j] \cdot \Pr[X_{b(i,h(i)+1),c(i,h(i)+1)} = v-j] \quad (1)$$

Thus, we perform at most one multiplication and one addition for each term in Equation 1, yielding

$$\mathcal{C}_{i,v} < 2 \cdot \min(v+1, |E_{\min}^{(i)}| + 1). \quad (2)$$

We can now bound $\sum_{i,v} \mathcal{C}_{i,v}$ as follows. Each time two megaprocessors $E_{h(i)}^{(i)}$ and $E_{h(i)+1}^{(i)}$ merge, we charge the cost of computing $\Pr[X_{b(i,h(i)),c(i,h(i)+1)} = v]$, for all $v \in \{0, \dots, k\}$ to the processors in the smaller of the two megaprocessors, distributing the cost evenly among the processors in the megaprocessor. Thus, we charge $(\sum_{v=0}^k \mathcal{C}_{i,v})/|E_{\min}^{(i)}|$ to each processor $O_i \in E_{\min}^{(i)}$.

Let $\kappa_j(i, v)$ denote how much of the cost of computing $\Pr[X_{b(i,h(i)),c(i,h(i)+1)} = v]$ we charge to O_j during the i^{th} merge. Let $\kappa_j(v)$ denote how much of the cost of computing $\Pr[X_{b(i,h(i)),c(i,h(i)+1)} = v]$ for all $i \in \{1, \dots, k-1\}$ we charge to processor O_j . Let κ_j denote the total amount we charge to processor O_j . In other words,

$$\kappa_j(i, v) = \begin{cases} \mathcal{C}_{i,v}/|E_{\min}^{(i)}| & \text{if } O_j \in E_{\min}^{(i)}, \\ 0 & \text{otherwise.} \end{cases} \quad (3)$$

$$\kappa_j(v) = \sum_{i=1}^{n-1} \kappa_j(i, v) \quad (4)$$

$$\kappa_j = \sum_{v=0}^k \kappa_j(v) = \sum_{i=1}^{n-1} \sum_{v=0}^k \kappa_j(i, v) \quad (5)$$

Then, we have $\sum_{i,v} \mathcal{C}_{i,v} = \sum_{j=1}^n \kappa_j$. We will bound $\sum_{i,v} \mathcal{C}_{i,v}$ by proving an upper bound on κ_j .

Consider any processor O_j . We will show that $\kappa_j(v) = \mathcal{O}(1)$. Let $i'(z) = i'(j, v, z)$ be the index of the merge in which processor O_j is charged for the cost of computing $\Pr[X_{b,c} = v]$ for any b, c for the z^{th} time. Formally, let

$$i'(z) = i'(j, v, z) = \begin{cases} \ell & \text{if } \exists \ell \in [1, n-1] \text{ s.t. } \kappa_j(\ell, v) > 0 \wedge |\{t \mid t < \ell, \kappa_j(t, v) > 0\}| = z-1 \\ 0 & \text{otherwise} \end{cases}$$

By Equation 2, $\mathcal{C}_{i,v} < 2|E_{\min}^{(i)}| + 2$, which implies $\kappa_j(i, v) < 4$ by Equation 3. Observe that by the definition of $E_{\min}^{(i)}$ and $i'(z)$, if $i'(2) > 0$, we have

$$|E_{\min}^{(i'(2))}| \geq |E_{h(i'(1))}^{(i'(1))}| + |E_{h(i'(1))+1}^{(i'(1))}| \geq v. \quad (6)$$

Also by the definition of $E_{\min}^{(i)}$, if $i'(z) > 0$, we have

$$|E_{\min}^{(i'(z))}| \geq 2 \cdot |E_{\min}^{(i'(z-1))}|. \quad (7)$$

Combining all these facts, and letting $Z = \max(x : i'(x) > 0)$, we have

$$\begin{aligned}
\kappa_j(v) &= \sum_{i=1}^{n-1} \kappa_j(i, v) && \text{by definition} \\
&\leq 4 + \sum_{i=2}^{n-1} \kappa_j(i, v) && \kappa_j(i, v) < 4 \\
&= 4 + \sum_{z=2}^Z \kappa_j(i'(z), v) && \text{by definition} \\
&= 4 + \sum_{z=2}^Z \frac{\mathcal{C}_{i'(z), v}}{|E_{\min}^{(i'(z))}|} && \text{by Equation 3} \\
&< 4 + \sum_{z=2}^Z \frac{2v+2}{|E_{\min}^{(i'(z))}|} && \text{by Equation 2} \\
&\leq 4 + \sum_{z=2}^Z \frac{2v+2}{2^{z-2} \cdot |E_{\min}^{(i'(2))}|} && \text{by Equation 7} \\
&< 6 + \sum_{z=2}^Z \frac{2v}{2^{z-2} \cdot |E_{\min}^{(i'(2))}|} && |E_{\min}^{(i'(2))}| \geq 2 \\
&\leq 6 + \sum_{z=2}^Z \frac{2}{2^{z-2}} && \text{by Equation 6} \\
&< 6 + 4 \\
&= 10.
\end{aligned}$$

Thus, we have $\kappa_j(v) = \mathcal{O}(1)$. This yields $\kappa_j = \sum_{v=0}^k \kappa_j(v) = \mathcal{O}(k)$. Since $\sum_{j=1}^n \kappa_j \leq n \cdot \max_{i \in \{1, \dots, n\}} \kappa_i$, we have,

$$\sum_{i=1}^{n-1} \sum_{v=0}^k \mathcal{C}_{i,v} = \sum_{j=1}^n \kappa_j \leq n \cdot \mathcal{O}(k) = \mathcal{O}(nk)$$

□

Therefore, by Lemma 5, we have $\mathcal{C} = \mathcal{O}(nk) + \sum_{i=1}^{n-1} \sum_{v=0}^k \mathcal{C}_{i,v} = \mathcal{O}(nk)$. Thus, our algorithm runs in total time $\mathcal{O}(n(\log n + k) + o)$.

5 An Ellipsoid-Based Algorithm for the $\text{SMT}(k)$ problem

There is a simple and elegant algorithm that solves the MINCOST problem for standard k -of- n testing, due to Salloum, Breuer, and (independently) Ben-Dov [1, 10, 11]. It outputs a strategy compactly represented by two permutations, one ordering the processors in increasing order of the ratio $c_i/(1-p_i)$, and the other in increasing order of the ratio c_i/p_i . Chang et al. and Salloum and Breuer later gave modified versions of this algorithm that output a less compact, but more efficiently evaluable representation of the same strategy [3, 12].

We now show how to combine previous techniques to obtain a polynomial-time algorithm for the $\text{SMT}(k)$ problem based on the ellipsoid method. The algorithm uses a technique of Despande and Hellerstein [6].

They showed that, for 1-of- n testing, an algorithm solving the MINCOST problem can be combined with the ellipsoid method to yield an algorithm for the MAXTHROUGHPUT problem. In fact, as we see in the proof below, their approach is actually a generic one, and can be applied to the testing of other functions.

The ellipsoid-based algorithm for k -of- n testing makes use of the dual of the LP for the CMT(k) problem, which is as follows:

Dual of Max-Throughput LP: Given $r_1, \dots, r_n > 0, p_1, \dots, p_n \in (0, 1)$, find an assignment to the variables y_i , for all $i \in \{1, \dots, n\}$, minimizing

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

subject to the constraints:

- (1) $\sum_{i=1}^n g(\pi, i) y_i \geq 1$ for all $T \in \mathcal{T}_c$,
 - (2) $y_i \geq 0$ for all $i \in \{1, \dots, n\}$.
-

Theorem 6. *There is a polynomial-time algorithm, based on the ellipsoid method, for solving the SMT(k) problem.*

Proof. The approach of Deshpande and Hellerstein works as follows. The input consists of the p_i and the r_i , and the goal is to solve the MAXTHROUGHPUT LP in time polynomial in n . The number of variables of the MAXTHROUGHPUT LP is not polynomial, so the LP cannot be solved directly. Instead, the idea is to solve it by first using the ellipsoid method to solve the dual LP. The ellipsoid method is run using an algorithm that simulates a separation oracle for the dual in time polynomial in n . During the running of the ellipsoid method, the violated constraints returned by the separation oracle are saved in a set M . Each constraint of the dual corresponds to an ordering T . When the ellipsoid method terminates, a modified version of the MAXTHROUGHPUT LP is generated, which includes only the variables z_T corresponding to orderings T in M (i.e. the other variables z_T are set to 0). This modified version can then be solved directly using a polynomial-time LP algorithm. The resulting solution is an optimal solution for the original MAXTHROUGHPUT LP.

The above approach requires a polynomial-time algorithm for simulating the separation oracle for the dual. Deshpande and Hellerstein's method for simulating the separation oracle relies on the following observations. In the dual LP for the MAXTHROUGHPUT 1-of- n testing problem, there are $n!$ constraints corresponding to the $n!$ permutations of the processors. The constraint for permutation π is $\sum_{i=1}^n g(T_1(\pi), i) y_i \leq 1$. If one views y as a vector of costs, where the cost of i is y_i , then $\sum_{i=1}^n g(T, i) y_i$ is the expected cost of testing an item x using ordering T . Thus one can determine the ordering T that minimizes $\sum_{i=1}^n g(T, i) y_i$ by solving the MINCOST problem with probabilities p_1, \dots, p_n and cost vector y . (Liu et al.'s approximation algorithm for generic MAXTHROUGHPUT also relies on this observation [9].)

If the MINCOST ordering T has expected cost less than 1, then the constraint it corresponds to is violated. Otherwise, since the right hand side of each constraint is 1, y obeys all constraints. Thus simulating the separation oracle for the dual on input y can be done by first running the MINCOST algorithm (with probabilities p_i and costs y_i) to find a MINCOST ordering T . Once T is found, the values of the coefficients $g(T, i)$ are calculated. These are used to calculate $\sum_{i=1}^n g(T, i) y_i$, the expected cost of T . If this value is less than 1, then the constraint $\sum_{i=1}^n g(T, i) y_i \leq 1$ is returned.

To apply the above approach to MAXTHROUGHPUT for standard k -of- n testing, we observe that in the dual LP for this problem, there is a constraint, $\sum_{i=1}^n g(T, i) y_i \leq 1$, for every possible strategy T . We can simulate a separation oracle for the dual on input y by running a MINCOST algorithm for standard k -of- n testing. We also need to be able to compute the $g(T, i)$ values for the strategy output by that algorithm. The algorithm of Chang et al. for the MINCOST standard k -of- n testing problem is suitable for this purpose, as it can easily be modified to output the $g(T, i)$ values associated with its output strategy T [3]. \square

References

- [1] Yosi Ben-Dov. Optimal testing procedure for special structures of coherent systems. *Management Science*, 27:1410–1420, 1981.
- [2] Endre Boros and Tonguç Ünlüyurt. Diagnosing double regular systems. *Annals of Mathematics and Artificial Intelligence*, 26(1-4):171–191, 1999.
- [3] Ming-Feng Chang, Weiping Shi, and W. Kent Fuchs. Optimal diagnosis procedures for k-out-of-n structures. *IEEE Transactions on Computers*, 39(4):559–564, 1990.
- [4] Anne Condon, Amol Deshpande, Lisa Hellerstein, and Ning Wu. Flow algorithms for two pipelined filter ordering problems. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, PODS 2006, Chicago, Illinois, USA*, pages 193–202. ACM, 2006.
- [5] Anne Condon, Amol Deshpande, Lisa Hellerstein, and Ning Wu. Algorithms for distributional and adversarial pipelined filter ordering problems. *ACM Transactions on Algorithms*, 5:24:1–24:34, March 2009.
- [6] Amol Deshpande and Lisa Hellerstein. Parallel pipelined filter ordering with precedence constraints. *ACM Transactions on Algorithms*, 8(4):41, 2012.
- [7] Michael R. Garey. Optimal task sequencing with precedence constraints. *Discrete Mathematics*, 4:37–56, 1973.
- [8] Murali S. Kodialam. The throughput of sequential testing. In *Proceedings of Integer Programming and Combinatorial Optimization, 8th International IPCO Conference, Utrecht, The Netherlands, June 13-15, 2001*, volume 2081 of *Lecture Notes in Computer Science*, pages 280–292. Springer, 2001.
- [9] Zhen Liu, Srinivasan Parthasarathy, Anand Ranganathan, and Hao Yang. A generic flow algorithm for shared filter ordering problems. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pages 79–88. ACM, 2008.
- [10] Salam Salloum. *Optimal testing algorithms for symmetric coherent systems*. PhD thesis, University of Southern California, 1979.
- [11] Salam Salloum and Melvin A. Breuer. An optimum testing algorithm for some symmetric coherent systems. *Journal of Mathematical Analysis and Applications*, 101(1):170 – 194, 1984.
- [12] Salam Salloum and Melvin A. Breuer. Fast optimal diagnosis procedures for k-out-of-n:G systems. *IEEE Transactions on Reliability*, 46(2):283 –290, June 1997.
- [13] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.
- [14] Tonguç Ünlüyurt. Sequential testing of complex systems: a review. *Discrete Applied Mathematics*, 142(1-3):189 – 205, 2004.