Content Placement via the Exponential Potential Function Method

David Applegate, Aaron Archer, Vijay Gopalakrishnan,
Seungjoon Lee, and K.K. Ramakrishnan

AT&T Shannon Research Laboratory, 180 Park Avenue, Florham Park, NJ 07932
{david,aarcher,gvijay,slee,kkrama}@research.att.com

Abstract. We empirically study the exponential potential function (EPF) approach to linear programming (LP), as applied to optimizing content placement in a video-on-demand (VoD) system. Even instances of modest size (e.g., 50 servers and 20k videos) stretch the capabilities of LP solvers such as CPLEX. These are packing LPs with block-diagonal structure, where the blocks are fractional uncapacitated facility location (UFL) problems. Our implementation of the EPF framework allows us to solve large instances to 1% accuracy 2000x faster than CPLEX, and scale to instances much larger than CPLEX can handle on our hardware.

Starting from the packing LP code described by Bienstock [4], we add many innovations. Our most interesting one uses priority sampling to shortcut lower bound computations, leveraging fast block heuristics to magnify these benefits. Other impactful changes include smoothing the duals to obtain effective Lagrangian lower bounds, shuffling the blocks after every round-robin pass, and better ways of searching for OPT and adjusting a critical scale parameter. By documenting these innovations and their practical impact on our testbed of synthetic VoD instances designed to mimic the proprietary instances that motivated this work, we aim to give a head-start to researchers wishing to apply the EPF framework in other practical domains.

Keywords: exponential potential function, approximate linear programming, Dantzig-Wolfe decomposition, priority sampling, content placement, video-on-demand

1 Introduction

Exponential potential function (EPF) methods for approximately solving linear programs (LPs) have a long history of both theory and implementation. These methods are most attractive when the constraint matrix has a large block-diagonal piece plus some coupling constraints, particularly if there is a fast oracle for optimizing linear functions over each block (called a *block optimization*). While the main focus has been on multicommodity flow (MCF) problems, Bienstock's implementation for generic packing LPs [4, Ch. 4] has been effective also for some network design problems, and Müller, Radke and Vygen [23] have produced an effective code for VLSI design.

Using Bienstock's book as a starting point, we adapted the EPF method to solve very large LPs arising from the content placement problem in a video-on-demand (VoD) system. After several major algorithmic improvements and some careful engineering, our code is able to solve instances with 2.5 billion variables and constraints to 1% accuracy in about 2.5 minutes. The largest instance CPLEX can solve on the same hardware is 1/50th as big, taking 3 to 4 hours. Most of our algorithmic insights apply to the EPF method in general, although some are specific to our VoD problem. Our goals are two-fold: to describe our innovations and their practical impact, and to give empirical insight into the workings of the EPF method, as a guidepost to future implementers.

Our orientation is thoroughly empirical: we explore and report what works well in practice, and do not aim to improve theoretical running times. Our implementation is based on the FPTASes in the literature, but we view broken proofs as an acceptable price to pay for substantial improvements in the performance of our code on our testbed. Interior-point LP codes take a similar tack, using parameter settings with better empirical performance than the ones that lead to provable polynomial-time convergence [15].

Related work. Building on the *flow deviation* method of Fratta, Gerla and Kleinrock [12], Shahrokhi and Matula [28] used an EPF to establish the first combinatorial FPTAS for the min-congestion MCF (a.k.a. *max concurrent flow*) problem. The myriad ensuing advances fall mainly into three lines. Fleischer [11], Garg and Könemann [13], Mądry [22], and the references therein contain improvements and extensions to other versions of MCF. Plotkin, Shmoys and Tardos [25] extended the MCF ideas to general packing and covering LPs; Koufogiannakis and Young [20] and the references therein contain further generalizations in this direction. Grigoriadis and Khachiyan [16] generalized in the direction of convex optimization; Müller, Radke, and Vygen [23] and the references therein extend this line of work. The survey by Arora, Hazan and Kale [2] draws connections between these algorithms and multiplicative weight update methods in other areas such as machine learning. All of these running times depend on the approximation error ϵ as ϵ^{-2}, and Klein and Young [19] give strong evidence that this is a lower bound for Dantzig-Wolfe methods like these. Building on work of Nesterov [24], Bienstock and Iyengar [6] obtained an algorithm with ϵ^{-1} dependence. However, their block optimizations solve separable quadratic programs rather than LPs. In practice, the improved ϵ dependence may or may not outweigh the increased iteration complexity.

In the literature, most implementations of the EPF framework are for MCF [3, 8, 14, 17, 18, 21, 27]. They show that the EPF method can outperform general-purpose LP solvers such as CPLEX by 2 or 3 orders of magnitude, but good performance requires careful implementation. Here, the blocks are ordinary flow problems, often shortest paths. Müller, Radke and Vygen [23] address a very different problem domain: VLSI design. Their blocks are fractional Steiner tree problems. Bienstock's code solves mixed packing and covering problems [5], although his published description covers only packing problems [4, Ch. 4]. Focus-

ing on generality and minimizing iterations, he solves blocks as generic LPs using CPLEX. He reports success with both MCF and network design problems. His code departs strongly from the theory by relying heavily on a *bootstrap* method (a.k.a. *procedure NX*), and using the block iterations mainly as an elaborate form of column generation for the bootstrap. Extending this idea, Bienstock and Zuckerberg [7] solve very large open-pit mining LPs (millions of variables and constraints) *to optimality* using *only* the bootstrap.

Our contributions. Bienstock's primary argument for the bootstrap is the strong lower bounds it yields. Our initial implementation replicated Bienstock, and although the bootstrap produced excellent lower bounds as advertised, we found that its time and memory requirements posed our primary barrier to scalability, so we got rid of it. We found that smoothing the ordinary EPF duals to use as Lagrangian multipliers gave good lower bounds. Each Lagrangian lower bound is expensive, as it requires a full pass of block optimizations. Our most interesting contribution is a method for shortcutting these lower bound passes, by extending the priority sampling techniques of [10, 31], combined with cached block solutions and judicious use of block heuristics. Other empirically important contributions include a better way of adjusting an important scale parameter δ over the course of the run, replacing the usual binary search with a radically different method for searching for OPT, and smart chunking strategies to exploit parallelism. While round-robin [4, 11, 26] and uniform random [16, 25] block selection strategies have been proposed in the past, we found round-robin with a random shuffle after each pass to be dramatically more powerful. All of these techniques apply to the EPF framework in general.

In our VoD application, the blocks are fractional uncapacitated facility location (UFL) problems. We use a greedy dual heuristic and a primal heuristic based on the local search algorithm of Charikar and Guha [9], for the *integer* UFL problem. This is insane for two reasons: we're using an approximation algorithm for an NP-hard problem as a heuristic for its LP relaxation, and the integrality gap means we may get worse solutions. Regardless, it is incredibly effective. Our primal and dual heuristics prove each other optimal most of the time, with only small gaps otherwise. Our primal heuristic is 30x to 70x faster than CPLEX, and our dual heuristic is 10x to 30x faster, where the speedup grows with network size. Our heuristic block solves are so fast that their running time is on par with the mundane data manipulation that surrounds them.

Section 2 describes the EPF framework. Section 3 describes our VoD model and our testbed of LPs. Section 4 compares our running times to CPLEX, and breaks them down by major components. Section 5 describes our key improvements to the basic framework, and demonstrates their practical impact on our testbed. Our end result is a code achieving a 2000x speedup over CPLEX, solving to 1% accuracy for the largest instances that CPLEX fits into memory on our system. The gap grows with problem size, and we easily solve instances that are 50x larger.

2 EPF Framework

We first discuss the general principles of the EPF framework, and then describe our algorithm as one instantiation. The EPF framework is a Dantzig-Wolfe decomposition method that uses exponential penalty functions in two capacities: first to define a potential function that encodes relaxed feasibility problems, and second to define Lagrange multipliers for computing lower bounds. Consider this LP:

$$\min cz \text{ s.t. } Az \leq b, z \in F^1 \times \cdots \times F^K \subseteq \mathbb{R}^n, \tag{1}$$

where each F^k is a polytope, $A = (a_{ij})$ is an $m \times n$ matrix, $c \in \mathbb{R}_n$ and $b \in \mathbb{R}^m$. Let OPT denote its optimum. Solution $z \in F$ is *ϵ-feasible* if $Az \leq (1+\epsilon)b$ (i.e., it violates each coupling constraint by at most $1+\epsilon$), and it is *ϵ-optimal* if $cz \leq (1+\epsilon)\text{OPT}$. Given constant ϵ, we aim for an ϵ-feasible, ϵ-optimal solution.

Dantzig-Wolfe decomposition takes advantage of fast algorithms for optimizing linear objective functions over each F^k. In this paper, we assume that the coupling constraints constitute a *packing* problem, i.e., $a_{ij} \geq 0$ and $b_i > 0$. Let the set R index the rows of A, let $a_i \in \mathbb{R}_n$ denote row i of A, let index 0 refer to the objective, and define $R^* = R \cup \{0\}$. Given a row vector of Lagrange multipliers $\lambda \in \mathbb{R}_{R^*}$ with $\lambda \geq 0$ and $\lambda_0 > 0$, define $c(\lambda) = c + \frac{1}{\lambda_0}\lambda A$. Whenever $k \in \mathcal{B} := \{1, \ldots, K\}$, a superscript k denotes the portion of an object corresponding to block k, e.g., z^k, A^k, F^k, or $c^k(\cdot)$. Define

$$LR^k(\lambda) = \min_{z^k \in F^k} c^k(\lambda) z^k, \tag{2}$$

and $LR(\lambda) = \sum_{k \in \mathcal{B}} LR^k(\lambda) - \frac{1}{\lambda_0}\lambda_R b$, where the notation λ_R means to restrict vector $\lambda \in \mathbb{R}_{R^*}$ to its R components (i.e., exclude λ_0). Standard duality arguments show that $LR(\lambda) \leq \text{OPT}$. All of our lower bounds derive from this fact.

The heart of the EPF method addresses feasibility problems, so we will guess a value B for OPT and consider the problem FEAS(B), wherein we replace the objective in (1) with the constraint $cz \leq B$. A solution is ϵ-feasible for FEAS(OPT) iff it is ϵ-feasible, ϵ-optimal for (1). With OPT unknown, we must search on B.

Define $\alpha(\delta) = \frac{\gamma \log(m+1)}{\delta}$, where $\gamma \geq 1$ and δ is a scale factor that evolves over the course of the algorithm. Let $r_i(z) = \frac{1}{b_i} a_i z - 1$ be the relative infeasibility of constraint i, and define aliases $a_0 := c$ and $b_0 := B$ so that $r_0(z) = \frac{1}{B} cz - 1$. Define $\delta_c(z) = \max_{i \in R} r_i(z)$ as the max relative infeasibility over the coupling constraints, and $\delta(z) = \max(\delta_c(z), r_0(z))$. Let $\Phi_i^\delta(z) = \exp(\alpha(\delta) r_i(z))$ define the potential due to constraint i, and $\Phi^\delta(z) = \sum_{i \in R^*} \Phi_i^\delta(z)$ define the overall potential function we aim to minimize (for fixed δ). If z is feasible for (1) then each $r_i(z) \leq 0$ so $\Phi^\delta(z) \leq m+1$, whereas if even one constraint i has $r_i(z) \geq \delta$, then $\Phi^\delta(z) > \Phi_i^\delta(z) \geq (m+1)^\gamma \geq m+1$. Thus, minimizing $\Phi^\delta(z)$ either finds a δ-feasible z, or proves that none exists.

Algorithm 1 EPF framework

Parameters: approximation tolerance $\epsilon > 0$, exponent factor $\gamma \approx 1$, smoothing parameter $\rho \in [0, 1)$, chunk size $s \in \mathbb{N}$
Initialize: solution $z \in F$, LB = valid lower bound on OPT, UB $\leftarrow \infty$, objective target $B \leftarrow$ LB, smoothed duals $\bar{\pi} = \pi^\delta(z)$, scale parameter $\delta = \delta(z)$, number of chunks $N_{\text{ch}} = \lceil K/s \rceil$
for Pass $= 1, 2, \ldots$ **do**
  Select a permutation σ of the blocks $\mathcal{B}$ uniformly at random, and partition $\mathcal{B}$ into chunks $C_1, \ldots, C_{N_{\text{ch}}}$, each of size s, according to σ.
  for chunk $C = C_1, \ldots, C_{N_{\text{ch}}}$ **do**
    for each block $k \in C$ **do**
      optimize block: $\hat{z}^k \leftarrow \arg\min_{z^k \in F^k} c^k(\pi^\delta(z))$
      compute step size: $\tau^k \leftarrow \arg\min_{\tau \in [0,1]} \Phi^\delta(z + \tau(\hat{z}^k - z^k))$
      take step in this block: $z^k \leftarrow z^k + \tau^k(\hat{z}^k - z^k)$
      Save $\hat{z}^k$ for possible use in shortcutting step 15 later.
    shrink scale if appropriate: $\delta = \min(\delta, \delta(z))$
    if $\delta_c(z) \leq \epsilon$ (i.e., z is ϵ-feasible) and $cz <$ UB **then** UB $\leftarrow cz$, $z^* \leftarrow z$
    if UB $\leq (1+\epsilon)$LB **then** return z^*
    $\bar{\pi} \leftarrow \rho\bar{\pi} + (1-\rho)\pi^\delta(z)$
  lower bound pass: LB $\leftarrow \max(\text{LB}, LR(\bar{\pi}))$, $B \leftarrow$ LB
  if UB $\leq (1+\epsilon)$LB **then** return z^*

The plan is to minimize $\Phi^\delta(z)$ via gradient descent. Let $\pi_i^\delta(z) = \Phi_i^\delta(z)/b_i$ for $i \in R^*$, and $g(z) = \pi_0^\delta(z)c + \pi_R^\delta(z)A$. The gradient of the potential function is

$$\nabla\Phi^\delta(z) = \alpha(\delta)g(z) = \alpha(\delta)\pi_0^\delta(z)c\left(\pi^\delta(z)\right), \tag{3}$$

a positive scalar times $c\left(\pi^\delta(z)\right)$. By gradient descent, we mean to move z along some segment so as to decrease $\Phi^\delta(z)$ at maximum initial rate. More precisely, defining $z(\tau) = (1-\tau)z + \tau\hat{z}$, we choose $\hat{z} \in F$ to minimize the directional derivative $\frac{d}{d\tau}\Phi^\delta(z(\tau))|_{\tau=0} = \nabla\Phi^\delta(z)(\hat{z} - z) = \alpha(\delta)\pi_0^\delta(z)c\left(\pi^\delta(z)\right)(\hat{z} - z)$. This is equivalent to solving the optimization problem (2) with $\lambda = \pi^\delta(z)$, once for each block $k \in \mathcal{B}$. Thus, solving the Lagrangian relaxation of (1) with this choice of multipliers serves twin purposes: giving a primal search direction and a lower bound on OPT. If we require only a search direction, we can optimize just a single block k and step in that block, leaving the others fixed. This block iteration is the fundamental operation of the EPF method.

Our full algorithm is extremely intricate. For clarity of exposition, we begin our description with a high-level overview in pseudocode as Algorithm 1. Our approach departs from that of both the theory and previous experimental work in several key ways. Some of these departures are evident in the pseudocode, but most are embedded in how we implement key steps. We now outline them briefly, deferring details to Section 5.

Instead of locating OPT via binary search on B, we use Lagrangian lower bounds directly and employ an *optimistic-B* strategy, setting $B \leftarrow$ LB. This departs strongly from Bienstock [4], whose lower bounds come from his bootstrap

procedure. The scale parameter δ is critical because it appears in the denominator of the exponentials, and strongly affects the step sizes τ^k. Earlier work changed δ by discrete factors of 2, which we found to be quite disruptive to convergence. Instead, we lower δ gradually as $\delta(z)$ falls. Continuous δ works harmoniously with optimistic-B, since it avoids the spikes in $\delta(z)$ associated with decreases in B during binary search.

The theory suggests using the dual weights $\pi^\delta(z)$ in step 15, but we discovered that the smoothed duals $\bar{\pi}$ yield stronger and more consistent lower bounds. Although we cannot justify it, replacing $\pi^\delta(z)$ with $\bar{\pi}$ in the block solves (step 7) improves iteration performance, even though $\hat{z} - z$ is no longer a gradient direction. Fast block heuristics dramatically speed up steps 7 and 15, and we can partially parallelize the chunk iteration (steps 6–10), both with minimal impact on iteration performance. The simple idea of shuffling the round-robin order for each pass (step 4) has a surprisingly dramatic impact on iteration performance. We solve some knapsack problems to initialize z and LB in step 2 (details omitted for space). Finally, we use $\gamma \leftarrow 1$, $s \leftarrow 120$, and $\rho \leftarrow 0.001^{1/N_{\text{ch}}}$.

3 VoD Model, Testbed and Machine Environment

Our VoD model, introduced in [1], begins with a set of *video hub offices* (VHOs), each serving video requests from users in a metropolitan area. High-speed links connect the VHOs, allowing them to satisfy a local request by streaming the video from a remote VHO. Given a demand forecast, we desire an assignment of video content to VHOs that minimizes total network traffic while respecting link bandwidths and VHO disk capacities. Variable y_i^k indicates whether to place video k at VHO i, and x_{ij}^k denotes what fraction of requests for video k at VHO j should be served from VHO i. The blocks are fractional UFL problems. In reality, the y_i^k variables should be binary, and we have an effective heuristic that rounds an LP solution, one video at a time. In practice, this tends to blow up the disk and link utilizations by about 1%, and the objective by about 1% to 2%, relative to the LP solution. Therefore, we focus on finding an ϵ-feasible, ϵ-optimal solution to the LP, with $\epsilon = 1\%$. Since the content placement would be re-optimized weekly, we can afford to let the optimization run for several hours. Disk and link capacity are treated as fixed at this time scale. Optimizing those over a longer planning horizon is another interesting problem, but not our present focus.

We conducted experiments on a testbed of 36 synthetic instances, consisting of all combinations of 3 real network topologies (Tiscali, Sprint, and Ebone, taken from Rocketfuel [29]), 6 library sizes (ranging from 5k to 200k videos), and 2 disk size scenarios (small disk/large bandwidth, and vice versa). Our testbed is available at `http://www.research.att.com/~vodopt`. These non-proprietary instances were designed to mimic the salient features of the proprietary instances that motivated this work. In each instance, we set the uniform link bandwidth just slightly larger than the minimum value that makes the instance feasible. We ran our experiments on a system with two 6-core 2.67GHz Intel Xeon X5650

Table 1. Running time, memory usage , and number of passes. Each row aggregates 6 instances (i.e., 3 networks and 2 disk types).

library	CPLEX		Block (100 seeds)			
size	time (s)	mem (GB)	time (s)	# passes	mem (GB)	speedup
5,000	894.47	10.15	1.39	15.37	0.11	644x
10,000	2062.10	19.36	1.77	9.29	0.18	1168x
20,000	5419.57	37.63	2.62	6.85	0.33	2071x
50,000			5.44	5.94	0.77	
100,000			10.45	5.57	1.52	
200,000			20.03	5.25	3.02	
1,000,000			98.61	5.07	15.00	

CPUs (12 cores total) and 48GB of memory. Our code is written in C++, compiled with the GNU g++ compiler v4.4.6. For our exact (not heuristic) block optimizations, we use CPLEX version 12.3.

4 Top-Line Results

Table 1 compares the running time and memory usage of our code with $\epsilon = 1\%$ to the CPLEX parallel barrier code for solving the same instances to optimality. The results reported for CPLEX reflect a single run. The results for our code use 100 different random seeds, since there is some variability across seeds, owing primarily to the block shuffling (Section 5.3). For this experiment only, we included instances with one million videos to emphasize scalability. In 38 instances out of 42, the running time's standard deviation is within 10% of its mean. In 33 instances, the number of passes has standard deviation within 5% of its mean. We take the arithmetic mean over the 100 random seeds for each instance, then take the geometric mean over the 6 instances of each library size, and report these numbers. The memory footprint and running time of our code both scale about linearly with library size (modulo a constant offset). The CPLEX memory footprint also scales linearly, but its running time is decidedly superlinear. For the largest instances that CPLEX could solve, our code is 2000x faster.

Both our code and CPLEX ran on all 12 cores. Our code achieves an 8x parallel speedup on its block solves, but only 4x overall. CPLEX achieves a 3x parallel speedup.

Our code's total running time over these 4200 runs breaks down as follows. Block solves (step 7) account for 24.2%, line search (step 8) for 3.4%, and the remainder of block iterations (steps 9-14) for 22.7%. In the LB pass (step 15, see Section 5.1), heuristic LB passes account for 17.5% and exact CPLEX passes for 0.7%. Initializing z and LB (step 2) accounts for 17.0%, and various overheads (e.g., reading instance data, data structure setup, etc.) for the remaining 14.4%.

5 Key Algorithmic Improvements

We now describe each of our key algorithmic ideas in some detail, and report experiments quantifying their impact. Unless specified otherwise, each experiment involves running with 10 different random seeds on each of the 36 instances. In reporting our data, we took an arithmetic mean over the 10 runs on each instance. When aggregating further, we then use a geometric mean across instances, to cope with the differing scales.

5.1 Shortcutting the Lower Bound Passes

Since step 15 is executed only once per primal pass, these LB passes account for at most half of the block solves. We can cut this further, by using the statistical technique of *priority sampling* to abort an LB pass early if we have high confidence that finishing it will not yield a useful bound.

Priority sampling. Duffield, Lund and Thorup's priority sampling is a non-uniform sampling procedure that yields low-variance unbiased estimates for weighted queries, where the query need not be known at the time the sample is computed [10]. We describe it in the abstract before explaining how to apply it in our context. Given a set of items $i \in I$, non-negative item weights w_i, and a sample size N, priority sampling selects a random sample $S_N \subseteq I$ with $|S_N| = N$ and weight estimators $\hat{w}_i$ for $i \in S_N$ with the following properties:

- **The estimator is unbiased [10]:** for all query vectors $f \in [0,1]^I$,

$$\sum_{i \in I} f_i w_i = E\left[\sum_{i \in S_N} f_i \hat{w}_i\right] . \tag{4}$$

- **The estimator is nearly optimal:** in a sense formalized by Szegedy [30], it has lower variance than the best unbiased estimator using $N-1$ samples.
- **The estimator comes with confidence bounds [31]:** Given error tolerance $\xi > 0$, there are readily computable lower and upper confidence bounds $LC(\xi)$ and $UC(\xi)$ (depending also on w, f, and the random draw) such that $\Pr[LC(\xi) > \sum_{i \in I} f_i w_i] \leq \xi$ and $\Pr[UC(\xi) < \sum_{i \in I} f_i w_i] \leq \xi$. The confidence bounds assume adversarial w and f; the probability is over the random draw.
- **The estimator (4) evaluates f_i and $\hat{w}_i$ only for $i \in S_N$:** In the applications that originally motivated priority sampling, I and w_i were presented in a stream and priority sampling limited the memory required for $\hat{w}$; f was supplied afterwards, but given explicitly. In our application, we can easily store all of w, but f is given implicitly and is expensive to compute. Priority sampling allows us to compute f_i only for $i \in S_N$.
- **The samples can be nested:** The random draw establishes a permutation of the elements such that S_N consists of the first N, regardless of N.

The cited papers [10, 30, 31] consider only the binary case $f \in \{0,1\}^I$. However, all properties except the confidence bounds generalize to $f \in [0,1]^I$. We use the confidence bounds even though they are only conjectured.

Combining priority sampling with solution pools. We estimate $LR(\bar{\pi}) = \sum_{k\in\mathcal{B}} LR^k(\bar{\pi}) - \frac{1}{\bar{\pi}_0}\bar{\pi}_R b$ by sampling the blocks, so $I = \mathcal{B}$. Given any pool of solutions P^k for block k, $\overline{LR}^k(\bar{\pi}) := \min_{z^k\in P^k} c(\bar{\pi})\, z^k$ is an upper bound on $LR^k(\bar{\pi})$. Two solutions are readily available: z^k and the $\hat{z}^k$ we saved in step 10. Let $w_k := \overline{LR}^k(\bar{\pi})$, $f_k := LR^k(\bar{\pi})/\overline{LR}^k(\bar{\pi})$, and $\xi = 5\%$, so the quantity to estimate is

$$\sum_{k\in I} f_k w_k = \sum_{k\in\mathcal{B}} \frac{LR^k(\bar{\pi})}{\overline{LR}^k(\bar{\pi})}\overline{LR}^k(\bar{\pi}) = \sum_{k\in\mathcal{B}} LR^k(\bar{\pi}), \tag{5}$$

We always feed the priority sampling routine a target lower bound T. For $\ell = 0, s, 2s, \ldots$ we compute $LR^k(\bar{\pi})$ for the s new blocks in S_ℓ and check whether $UC(\xi) < T + \frac{1}{\bar{\pi}_0}\bar{\pi}_R b$. If so, we predict that $LR(\bar{\pi}) < T$, and exit without a bound. Otherwise we continue with the next chunk of s blocks. Eventually, we either exit with no bound or compute $LR(\bar{\pi}) \geq T$.

Supposing $LR(\bar{\pi}) \geq T$, then each computation of $UC(\xi)$ nominally causes us to mistakenly terminate early (a false-positive error) with a distinct probability of ξ. This suggests that the overall false-positive rate for step 15 could be higher than ξ. However, these errors are highly correlated, and the empirical false positive rate is precisely zero!

We also leverage our block heuristics in step 15. When UB $= \infty$, we call routine WeakLB, in which we solve blocks using the dual heuristic to get a lower bound, possibly weaker than $LR(\bar{\pi})$, and use an aggressive target $T >$ LB to encourage either an early exit or a substantial increment to LB. Once UB $< \infty$, we instead call StrongLB with $T =$ UB$/(1+\epsilon)$, hoping to terminate. In this case, we use our block heuristics to reduce the number of (very expensive) exact block solves required. We first use the primal heuristic to generate a tighter upper bound on $LR(\bar{\pi})$, exiting early if $UC(\xi) < T$. Then we run the dual heuristic for all blocks; whenever it matches the primal heuristic, they both equal $LR^k(\bar{\pi})$. Then we exactly solve the remaining blocks to close their gaps, using the confidence bounds to exit early as appropriate. The dual heuristic already gives a (weak) bound, and the exact block solves strengthen it as we go along. In this case, StrongLB returns a bound even when it exits early.

Computational results for priority sampling. To measure how effectively priority sampling terminates LB passes, each time we ran a sampling LB pass, we also ran the LB pass to completion to determine the true result (but didn't use that value in the run). As a result, we could divide the LB passes into two cases, *useless* LB passes in which the true result was $< T$, and *useful* LB passes, in which the true result was $\geq T$. Among the 360 runs in this experiment, 264 terminated as a result of a useful StrongLB pass; the others were able to terminate based on a LB obtained from a previous useful WeakLB pass or useless StrongLB pass that returned a bound.

For a useless pass, we measure the effectiveness by how many block solves were avoided by terminating the pass early. Priority sampling avoided 87.4% of

Table 2. Performance of block heuristics. "Average error" measures relative error between the heuristic solution and the optimum. "Fraction opt" and "Fraction within 1%" are the fraction of the solutions with relative error at most 10^{-6} and 1%, respectively.

	primal heuristic			dual heuristic		
	Tiscali	Sprint	Ebone	Tiscali	Sprint	Ebone
average error	0.19%	0.30%	0.36%	0.12%	0.24%	0.13%
fraction opt	82.3%	76.8%	77.8%	78.8%	68.2%	77.0%
fraction within 1%	93.5%	90.3%	88.6%	96.3%	91.6%	95.8%
heuristic time/block	49 μs	31 μs	19 μs	85 μs	57 μs	35 μs
CPLEX time/block	3384 μs	1305 μs	610 μs	2499 μs	930 μs	405 μs

the block solves in WeakLB, 69.8% in the primal heuristic portion of StrongLB, and 94.8% in the (very expensive) exact portion of StrongLB.

For useful LB passes, the danger is incorrectly terminating the pass. Surprisingly, for the 1915 useful LB passes in this experiment, *none* were incorrectly shortcut. This is strong empirical evidence that the worst-case error bounds above are extremely pessimistic. Even using $\xi := 50\%$ would have caused only 18 passes to be incorrectly shortcut!

5.2 Block Heuristics

For block iterations, we need not actually compute a gradient direction; it suffices to compute a search direction that improves the potential Φ^δ. Thus, a fast primal heuristic can create significant savings. The block solves in LB passes need not be exact either, so we use a dual heuristic. In addition, when the primal heuristic fails to find an improving direction, we make a second attempt, using the greedy dual heuristic to provide an alternate warm start for the primal heuristic.

Table 2 illustrates the performance of the block heuristics, split by network since that determines the size of the UFL instances. The results are for 1 random seed, and are arithmetic means for the instances on that network in the testbed. Our heuristics are 11x to 70x faster than CPLEX and find optimal solutions in the majority of cases, with small error otherwise. While the detail is not shown here, the small errors by the heuristic solutions cause a modest increase in the number of passes. Specifically, when compared to the number of passes using exact block solutions, our primal and dual heuristics respectively lead to 16% and 1% more passes on average.

5.3 Block Shuffling

Bienstock's code uses round-robin passes [4]. We found that shuffling the blocks randomly after each pass improves iteration performance sharply. We compare our strategy with the following three static block ordering strategies: sorting videos by increasing or decreasing total request volume, or shuffling them randomly into a fixed order. All three require >40x more passes (45.2, 54.9, and 46.5, respectively) than our main code. Moreover, the demand-sorted versions

Table 3. Varying chunk size and line search method.

	bundled τ search			sequential τ search					
chunk size	12	24	48	12	48	96	120	192	240
thread loading	0.64	0.72	0.79	0.63	0.78	0.84	0.85	0.87	0.88
average τ	0.53	0.49	0.46	0.49	0.49	0.48	0.48	0.46	0.45
no. passes	9.25	10.28	11.52	7.26	7.29	7.37	7.45	7.67	7.97

failed to converge in some instances. We do not understand why the effect of shuffling is so dramatic.

5.4 Chunking

Our code groups s blocks into each chunk. Instead of solving the blocks sequentially as written in Algorithm 1, we solve them in parallel. Larger chunks enable better parallelism. The tradeoff is that block k cannot react to the change in $c^k(\pi^\delta(z))$ as other blocks in the same chunk update z in step 9. Another relevant aspect is how to perform line search for a given chunk. One way is to bundle all blocks in the chunk, select a single step size τ, and move them all by τ. Another way is to compute τ^k and update z^k sequentially for each block k, so that each step size adapts to the ones taken before, although the step directions do not. Step 9 actually updates two sets of values: $r_i(z)$ and z. Our code updates the former sequentially, so the next τ^k can benefit, but does the latter in parallel after computing all τ^k. The blocks within the chunk are visited in random order, according to σ from step 4.

Table 3 shows that bundled τ search achieves better thread-loading as s increases (defined as average utilization of each thread during parallel block solves), but the average step size drops noticeably, leading to more passes and poorer overall performance, even for $s = 48$. Sequential τ search shows the same trend for thread-loading, with the benefit tailing off by $s = 120$, but the decrease in step size and increase in passes are modest through $s = 120$, the value used in our main code. Empirically, the slightly outdated search direction does not significantly worsen the iteration counts until s is quite large.

5.5 Smoothing the Duals

Empirically, using the smoothed duals $\bar{\pi}$ instead of $\pi^\delta(z)$ causes the sequence of LBs computed to be both stronger and nearly monotone. This lessens the penalty for incorrectly aborting an LB pass, freeing us to be more aggressive about our shortcuts. For 85.2% of the passes, the LB we would obtain by disabling shortcutting and calling WeakLB exceeds the LB from the previous pass, and for 82.0% of the passes the LB is the largest we have seen so far. Compared to using $\pi^\delta(z)$ for the LB pass, using $\bar{\pi}$ results in 19% fewer passes on average. Using $\bar{\pi}$ for primal block iterations results in an average 19% further reduction in passes. We discovered this curious last item by mistake and cannot motivate or explain it.

6 Summary and Future Work

We implemented the EPF framework for approximate linear programming, and applied it to LPs that optimize content placement for a VoD service. We describe design choices and innovations that led to significant, sometimes dramatic, improvements in the performance of the code on our testbed. It would be interesting to learn whether these techniques are equally effective when applying the EPF method in other domains.

Other experimenters [4, 8, 14, 21] have reported a better experimental iteration dependence on ϵ than the $O(\epsilon^{-2})$ predicted by theory. It would be interesting to see how our code compares.

We have discovered a method for solving our UFL block LPs via the simplex method, while handling all but $O(n)$ of the $O(n^2)$ variables and constraints combinatorially. Our method bears a resemblance to Wunderling's kernel simplex method [32]. In future work, we intend to implement this method and use it as a replacement for CPLEX in the few cases where we require an exact block solver.

Acknowledgments: We thank Dan Bienstock, Cliff Stein, David Shmoys, Jochen Könemann and David J. Phillips for useful discussions, and especially Bienstock for providing us with his EPF code for comparison. We also thank Mikkel Thorup for showing us how to generalize priority sampling to make it apply in our scenario.

References

1. Applegate, D., Archer, A., Gopalakrishnan, V., Lee, S., Ramakrishnan, K.K.: Optimal content placement for a large-scale VoD system. CoNEXT. (2010) 4:1–12
2. Arora, S., Hazan, E., Kale, S.: The multiplicative weights update method: a meta-algorithm and applications. Theor. Comput. **8** (2012) 121–164
3. Batra, G., Garg, N., Gupta, G.: Heuristic improvements for computing maximum multicommodity flow and minimum multicut. In Brodal, G.S., Leonardi, S., eds.: Algorithms - ESA 2005. Volume 3669 of LNCS. Springer, Heidelberg (2005) 35–46
4. Bienstock, D.: Potential Function Methods for Approximately Solving Linear Programming Problems: Theory and Practice. Kluwer, Boston (2002)
5. Bienstock, D.: Personal communication (2011)
6. Bienstock, D., Iyengar, G.: Approximating fractional packings and coverings in $O(1/\epsilon)$ iterations. SIAM J. Comput. **35**(4) (2006) 825–854
7. Bienstock, D., Zuckerberg, M.: Solving LP relaxations of large-scale precedence constrained problems. In Eisenbrand, F., Shepherd, F.B., eds.: IPCO. Volume 6080 of LNCS. Springer, Heidelberg (2010) 1–14
8. Borger, J.M., Kang, T.S., Klein, P.N.: Approximating concurrent flow with unit demands and capacities: an implementation. In Johnson, D.S., McGeoch, C.C., eds.: Network Flows and Matching: First DIMACS Implementation Challenge. American Mathematical Society (1993) 371–386
9. Charikar, M., Guha, S.: Improved combinatorial algorithms for facility location problems. SIAM J. Comput. **34**(4) (2005) 803–824
10. Duffield, N.G., Lund, C., Thorup, M.: Priority sampling for estimation of arbitrary subset sums. J. ACM **54**(6) (2007)

11. Fleischer, L.: Approximating fractional multicommodity flow independent of the number of commodities. SIAM J. Discrete Math. **13**(4) (2000) 505–520
12. Fratta, L., Gerla, M., Kleinrock, L.: The flow deviation method: An approach to store-and-forward communication network design. Networks **3**(2) (1973) 97–133
13. Garg, N., Könemann, J.: Faster and simpler algorithms for multicommodity flow and other fractional packing problems. SIAM J. Comput. **37**(2) (2007) 630–652
14. Goldberg, A., Oldham, J., Plotkin, S., Stein, C.: An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow. In Bixby, R.E., Boyd, E.A., Ríos-Mercado, R.Z., eds.: IPCO. Volume 1412 of LNCS. Springer, Heidelberg (1998) 338–352
15. Gondzio, J.: Interior point methods 25 years later. Eur. J. Oper. Res. **218** (2012) 587–601
16. Grigoriadis, M.D., Khachiyan, L.G.: Fast approximation schemes for convex programs with many blocks and coupling constraints. SIAM J. Optimiz. **4**(1) (1994) 86–107
17. Grigoriadis, M.D., Khachiyan, L.G.: An exponential-function reduction method for block-angular convex programs. Networks **26** (1995) 59–68
18. Jang, Y.: Development and implementation of heuristic algorithms for multicommodity flow problems. PhD thesis, Columbia University (1996)
19. Klein, P.N., Young, N.E.: On the number of iterations for Dantzig-Wolfe optimization and packing-covering approximation algorithms. In Cornuéjols, G., Burkard, R.E., Woeginger, G.J., eds.: IPCO. Volume 1610 of LNCS. Springer, Heidelberg (1999) 320–327
20. Koufogiannakis, C., Young, N.E.: Beating simplex for fractional packing and covering linear programs. FOCS. (2007) 494–504
21. Leong, T., Shor, P., Stein, C.: Implementation of a combinatorial multicommodity flow algorithm. In Johnson, D.S., McGeoch, C.C., eds.: Network Flows and Matching: First DIMACS Implementation Challenge. American Mathematical Society (1993) 387–406
22. Mądry, A.: Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. STOC. (2010) 121–130
23. Müller, D., Radke, K., Vygen, J.: Faster min-max resource sharing in theory and practice. Math. Program. Comput. **3** (2011) 1–35
24. Nesterov, Y.: Smooth minimization of non-smooth functions. Math. Program. Ser. A **103** (2005) 127–152
25. Plotkin, S.A., Shmoys, D.B., Éva Tardos: Fast approximation algorithms for fractional packing and covering problems. Math. Oper. Res. **20** (1995) 257–301
26. Radzik, T.: Fast deterministic approximation for the multicommodity flow problem. Math. Program. **77** (1997) 43–58
27. Radzik, T.: Experimental study of a solution method for the multicommodity flow problem. ALENEX '00. (2000) 79–102
28. Shahrokhi, F., Matula, D.W.: The maximum concurrent flow problem. J. ACM **37**(2) (1990) 318–334
29. Spring, N., Mahajan, R., Wetherall, D.: Measuring ISP topologies with Rocketfuel. SIGCOMM. (2002)
30. Szegedy, M.: The DLT priority sampling is essentially optimal. STOC. (2006) 150–158
31. Thorup, M.: Confidence intervals for priority sampling. SIGMETRICS. (2006) 252–263
32. Wunderling, R.: The kernel simplex method. Talk at ISMP (August 2012)