Implementation of the fastpc algorithm: data structures and distributions

Stephen V. Cole
Monik Khare
Neal E. Young
University of California, Riverside

September 29, 2011

1 Introduction

In [4], Koufogiannakis and Young give a randomized algorithm for solving pure packing and covering linear programs. Here, a packing problem is of the form $\max\{a \cdot x : Mx \leq b, x \geq 0\}$, and a covering problem is of the form $\min\{a \cdot \hat{x} : M\hat{x} \geq b, \hat{x} \geq 0\}$, where the entries in the constraint matrix M are non-negative. They build on an algorithm by Grigoriadis and Khachiyan [2] to simultaneously build up primal and dual solutions until a $(1 \pm O(\epsilon))$ ratio exists between them, which by weak duality implies $(1 \pm O(\epsilon))$-approximate solutions. The primal and dual solution vectors are initialized to zero and then are built up in a series of rounds. In each round, one primal and one dual variable are simultaneously chosen and incremented by a small amount. The primal variable is chosen from a distribution determined by the current dual solution vector, and vice versa. The authors show that their algorithm returns a $(1 \pm O(\epsilon))$-approximate solution in running time $O(n + (r + c)\log(n)/\epsilon^2)$ with high probability. We assume a reasonable degree of familiarity with this algorithm, which we refer to as "fastpc," in the current work; its pseudocode is shown in Figure **??**.

Koufogiannakis and Young implemented the algorithm in C++ for the restricted case of 0/1 input matrices, and their results confirm asymptotic superiority to the Simplex algorithm as implemented in the Gnu Linear Programming Kit (GLPK). We present a generalized implementation for input matrices with coefficients in $[0, \infty)$. As suggested in [4], this is primarily accomplished by employing the non-uniform-increments scheme of Garg and Konemann [1] and by maintaining approximate, rather than exact, constraint values.

The fastpc algorithm contains many operations on floating-point data, and assumes these operations can be carried out with arbitrary precision. However, this is not possible on any physical hardware machine; some approximation of floating-point values is inherent in representing them with a finite number of bits. In this work, we present a practical data structure and a modified version of the fastpc algorithm which together maintain the algorithm's correctness and allow for a straightforward implementation while bounding the possible error due to this

approximation.

The rest of this paper is organized as follows. In Section 2 we briefly discuss the fastpc algorithm as given in [4]. Section 3 describes our computation model for the modified algorithm. A modified sampling procedure is introduced and analyzed in Sections 4. In Section 6 we introduce our modifications to the fastpc algorithm, and we analyze the performance and correctness of this algorithm in Section 7. Conclusions and future work are discussed in Section 8.

2 FASTPC algorithm

The FASTPC algorithm given in [4] is shown in Figure **??**.

3 Computation model

Floating-point operations in the fastpc algorithm that could cause precision loss occur in two main categories: those that maintain the probability pseudo-distribution vectors, and those involved in random sampling. Here we introduce our design decisions that impact the probability vectors, and we discuss the random sampling procedure in Section 4.

The fastpc algorithm maintains dynamic probability pseudo-distributions p, $\hat{p}$, $p \times \hat{u}$, and $\hat{p} \times u$ (a pseudo-distribution is a vector of probability values (items) that do not add up to one, i.e. they have not been normalized).

Item weights as given in fastpc are powers of $(1+\epsilon)$ for items in p and $p \times \hat{u}$, and powers of $(1-\epsilon)$ for items in $\hat{p}$ and $\hat{p} \times u$. We modify this scheme by rounding ϵ to satisfy $(1+\epsilon) = 2^{1/k}$ for some integer k. Then items in all probability vectors are maintained as powers of $(2^{1/k})$, with exponents in $\hat{p}$ and $\hat{p} \times u$ being nonpositive and decreasing. We show how to compensate for this change in Section 7. To simplify calculations and help prevent precision loss, we store all item weights by their exponent only. Since $k = \log_{1+\epsilon} 2$, k can easily be represented in a 32-bit machine word for reasonable values of ϵ (for example, $k < 70$ when $\epsilon \geq 0.01$).

Also, by the method described in [4], and originally from [5], we can scale all input coefficients in M to be in a bounded range. Any solution to the scaled problem will be a feasible solution to the original problem, at the cost of an extra ϵ-factor in the runtime. To properly initialize the vectors u and $\hat{u}$, we round all input coefficients to their nearest power of $(1+\epsilon)$.

Thus, all arithmetic involving the probability vectors is done in the integer regime, at the cost of an extra ϵ-factor in the approximation guarantee.

4 Sampling data structure

We now introduce a practical data structure for storing the probability vectors used by fastpc. We implement a structure called a "sampler" for this purpose, drawing heavily on the schemes for theoretical data structures previously proposed by Hagerup et al. in [3] and Matias et al. in

[6]. Specifically, we employ the rejection method when sampling and the clustering of items with similar weights into buckets.

The sampler provides the following interface to fastpc:

1. `create_sampler(int` n`, int[] expts)`: create sampler containing n items with initial exponents from `expts`.
2. `sample(sampler` s`)`: return some s_i or *REJECT*. Each item is chosen with probability proportional to its weight; the sampler returns *REJECT* with some probability according to the rejection method.
3. `norm(sampler` s`)`: return total weight contained in s, including the weight associated with rejection, as a (mantissa, exponent) pair. Because the total weight in a sampler can exceed the storage capacity of a standard double-precision word, which includes 11 bits for the exponent, we simulate floating-point words by storing a mantissa and exponent in separate words, allowing 32 or 64 bits of storage for the exponent.
4. `increment_exponent(sampler` s`, int` i`)`; `decrement_exponent(sampler` s`, int` i`)`: increase/decrease exponent of s_i by 1.
5. `remove(sampler` s`, int` i`)`: remove s_i from s.

In order to meet the runtime guarantee of fastpc, all sampler operations must be performed in amortized $O(1)$ time. In addition, the probability of a `sample()` call returning *REJECT* must be constant and < 1. In section 4.3 we show a worst-case running time of $O(\log n)$ per operation, but an empirically observed time of $O(1)$.

We introduce our sampling data structure in three stages: first for the special case where all item weights are whole powers of 2, then for arbitrary item weights but with no runtime guarantee, and finally for arbitrary item weights with an amortized $\log n$ runtime guarantee per operation.

4.1 Special Case: Whole Powers of Two

We first present a sampler suited to handle item weights that are whole powers of 2. This is the special case in which the probability of rejection is 0.

For each exponent whose value is held by at least one item, we maintain a bucket of all items currently having that exponent. To calculate the norm, we multiply the number of items in each bucket by the appropriate power of 2 and sum the results. To sample, we first choose a random integer between 0 and the norm of the sampler. We then scan through the buckets from largest exponent to smallest until the cumulative weight of the scanned buckets exceeds the random integer. Finally, we choose an item uniformly from the last bucket scanned and return it. To increment (decrement) an item's exponent, we remove the item from the bucket of its old exponent and add it to the bucket of its new exponent.

4.2 General Case

We next describe a sampler suited to handle arbitrary item weights. Our strategy will be to consider all item weights to be their next-highest power of 2 and use the rejection method to account for their actual weights.

We calculate the sampler norm exactly as in the previous scheme. Note that the norm could now be larger than the sum of all item weights; the difference between the two is the probability of rejection. We sample as in the previous scheme, with the addition of a rejection step. Once we have chosen an item, we accept and return it with probability equal to its weight over the next-highest power of two. This quantity is $\geq 1/2$. If we do not accept the chosen item, we return *REJECT*. To increment (decrement) an item's exponent, we increment (decrement) its stored exponent and move it to a new bucket if necessary.

4.3 Full Scheme

We now introduce a modification to the previous data structure that both ensures correctness and gives a $\log n$ worst-case running time guarantee.

We first calculate a certain threshold exponent t. We consider all items with exponent less than t to be in a single bucket which we call *Small* , and we refer to all other items collectively as *Large* . We set t to be the highest exponent such that the total weight in *Large* is at least the total weight in *Small* . In other words, we set t such that at least half the sampler's total weight is in *Large* . An illustration of the full sampler is shown in Figure 1.

To calculate the norm, we first calculate the mantissa. We consider all bucket exponents relative to t rather than to 0; for example, all item weights in *Small* are counted as $2^0 = 1$, all item weights in the bucket with exponent $t+1$ are counted as $2^1 = 2$, etc. This technique ensures an upper bound of $4n$ on the mantissa value (see Appendix 9.1 for details), which implies that the mantissa can be stored accurately as a 64-bit integer as long as $n < 2^{62}$. This is significantly larger than any problem size would require in the foreseeable future. We return the (mantissa, t) pair as the sampler norm.

When sampling, we consider the total weight in the sampler to consist only of the mantissa, and we consider all item exponents relative to t. Note that because *Small* can contain items with arbitrarily small weights relative to 2^t, the probability of rejection in *Small* could be arbitrarily high. However, since *Large* contains $\geq 1/2$ the total weight and the probability of rejection in *Large* is $\leq 1/2$, then the overall probability of rejection in the sampler is $\leq 1/4$, which is essential for satisfying the performance requirement specified in our discussion of the interface.

By our choice of t, *Large* is guaranteed to contain $\leq \log n$ buckets. It is this structural feature which produces the worst-case time bound on all sampler operations. Constant time per operation is achieved if a constant fraction of the weight in a sampler is contained in its first $O(1)$ buckets. In practice, we find this condition to hold for a variety of problem inputs.

The dense clustering of items in the first few buckets in practice motivates our deviation from the theoretical data structures of [3] and [6]. We group items into buckets much as in these

Figure 1: The full sampler scheme. The entire gray shaded area indicates the weight that is included in the mantissa. True item weights are represented by hatched bars, and the probability of rejection consists of all solid gray regions.

structures, but they impose an additional recursive hierarchy onto the buckets. Such a hierarchy guarantees good worst-case performance, since sampler operations can drill down through the hierarchy to a bucket rather than linearly traversing all buckets as our sampler does. However, drilling down through a bucket hierarchy also introduces overhead into each sampler operation. Since the weight of the samplers is densely concentrated into just a few buckets in our application, we find that the overhead incurred by a recursive bucket structure is more expensive than a linear-traversal scheme.

5 Sampling operations

We now discuss the probabilistic sampling operations used in the fastpc algorithm and show that these operations can be performed with small bounded precision loss.

Two sampling operations are required by the fastpc algorithm: sampling uniformly from the unit range [0,1] in lines ??? of fastpc and choosing an item from a sampler with probability proportional to its weight (which is the goal of the random-pair() subroutine). This second

operation is accomplished simply by invoking the `sample()` function of the appropriate samplers, the mechanics of which is described in the previous section.

Lines ??? of fastpc require a uniform random sample in the range [0,1] and a comparison of the quantities $M_{ij'}\delta_{i'j'}$ and $M_{i'j}\delta_{i'j'}$ to that sampled value. Because these quantities could be arbitrarily small, the granularity of a given machine's built-in sampling procedure may not be sufficient to ensure a fair sample and comparison. We therefore use a custom sampling procedure similar in spirit to the bucket-selection procedure of the `sample()` function when the value of either of these quantities is $< 1/2$. When a quantity is $>= 1/2$, we simply use the machine's built-in random sampling procedure and perform the comparison in a straightforward way. We describe the procedure for the comparison test when $M_{ij'}\delta_{i'j'} < 1/2$, and note that the test for $M_{i'j}\delta_{i'j'}$ is identical. Conceptually, this sampling procedure divides the [0,1] interval into smaller intervals with boundaries determined by successive powers of 1/2. We then iteratively select an interval by repeatedly sampling in [0,1] uniformly and moving to the next interval only if the sampled value is $> 1/2$. If the selected interval's lower boundary is greater than $M_{ij'}\delta_{i'j'}$, then the procedure terminates and returns that $M_{ij'}\delta_{i'j'}$ is smaller than the sampled value. Similarly, if during the iteration process the current interval's upper boundary is smaller than $M_{ij'}\delta_{i'j'}$, then the procedure terminates and returns that $M_{ij'}\delta_{i'j'}$ is greater than the sampled value. If $M_{ij'}\delta_{i'j'}$ lies between the boundaries of the selected interval, we normalize that interval to the [0,1] interval, adjusting $M_{ij'}\delta_{i'j'}$ accordingly, and recursively repeat the procedure. The probability of reaching any interval diminishes exponentially with the size of the interval, and so this procedure is expected to terminate in a constant number of iterations w.h.p.

We now show that the overall impact of these approximate sampling operations on the solution generated by fastpc is less than a $(1+\epsilon)$-factor.

To analyze the impact of the random-pair() subroutine using our `sample()` function on the overall modified algorithm, we consider the probability γ_{ij} of returning a particular (i,j) pair in an exact-arithmetic model and the probability γ'_{ij} of returning the corresponding (i,j) pair in our computational model using the rejection method. We first note that the weight $w'(sampler)$ of any sampler in our regime could differ from its weight $w(sampler)$ in the exact-arithmetic regime by at most a factor of $(1+\epsilon)$. We next observe that since the entire random-pair() procedure is restarted upon rejection, γ'_{ij} does not depend on any rejections that may occur during a call to random-pair() and can be calculated as though no rejections happen. Therefore, γ'_{ij} is proportional to $w'(i)/w'(sampler)$ and $w'(j)/w'(sampler)$. Now, since the weight of any individual item $w'(i)$ (or, equivalently, $w'(j)$) in our modified samplers also differs from its weight $w(i)$ in the exact regime by at most a factor of $(1+\epsilon)$, then the ratios of their weights clearly also differ by at most a $(1+\epsilon)$ factor. This implies that $\gamma'_{ij} \leq (1+\epsilon)\gamma$.

6 Modified FASTPC algorithm

To accommodate our arithmetic model and our sampling implementation, we introduce the following changes to the fastpc algorithm:

1. We replace the value of ϵ given as a problem parameter with a new ϵ that satisfies $(1+\epsilon)^k = 2$ for some integer k.
2. Instead of maintaining $\hat{p}_j$ as $(1-\epsilon)^{\hat{y}_j}$, we maintain it as $(1+\epsilon)^{-\hat{y}_j}$.
3. We modify the sampling procedure presented in fastpc; details are given in Section 4.
4. We change line ??? of fastpc so that y_i and its samplers are only updated if $\left(\frac{1}{1+\epsilon}\right) M_{ij'}\delta_{i'j'} \geq z$. This implies that $E[\Delta y] = \left(\frac{1}{1+\epsilon}\right) E[\Delta M x] = \left(\frac{1}{1+\epsilon}\right) \alpha M\hat{p}/|\hat{p}|$. We calculate $\hat{y}$ as in [4], so that $E[\Delta\hat{y}] = E[\Delta M^T\hat{x}] = \alpha M^T p/|p|$.

   With this modification, the analysis of correctness and running time from [4] still holds, with an additional additive ϵ-error in the approximation guarantee. Details are given in Section 7.

7 Analysis of modified FASTPC

We now follow the analysis in [4], Lemmas 3 and 4, to show that with probability at least $1-\frac{3}{rc}$ the revised algorithm returns a feasible $(1-O(\epsilon))$-approximate solution. We use the same potential function as in [4], namely $\Phi = |p||\hat{p}|$. We first show that it is a super-Martingale. If p' and $\hat{p}'$ denote p and $\hat{p}$ after a given round of the main loop of the algorithm, then

$$\begin{aligned}
|p'| &= \sum_i p_i(1+\epsilon)^{\Delta y_i} \\
&= \sum_i p_i(1+\epsilon\Delta y_i) \\
&= |p| + \epsilon p^T\Delta y. \\
\text{Also, } |\hat{p}'| &= \sum_j \hat{p}_j(1+\epsilon)^{-\Delta\hat{y}_j} \\
&= \sum_j \hat{p}_j(1-\frac{\epsilon}{1+\epsilon})^{\Delta\hat{y}_j} \text{ since } \frac{1}{1+\epsilon} = 1-\frac{\epsilon}{1+\epsilon} \\
&= \sum_j \hat{p}_j(1-\frac{\epsilon}{1+\epsilon}\Delta\hat{y}_j) \\
&= |\hat{p}| - \frac{\epsilon}{1+\epsilon}\hat{p}^T\Delta\hat{y}.
\end{aligned}$$

Multiplying these equations gives

$$\Phi' = |p'||\hat{p}'| \leq |p||\hat{p}| + \epsilon|\hat{p}|p^T\Delta y - \frac{\epsilon}{1+\epsilon}|p|\hat{p}^T\Delta\hat{y}.$$

Taking expectations and substituting for $E[\Delta y]$ and $E[\Delta\hat{y}]$ gives

$$\begin{aligned}
|p'||\hat{p}'| &\leq |p||\hat{p}| + \frac{\epsilon\left(\frac{1}{1+\epsilon}\right)|\hat{p}|p^T\alpha M\hat{p}}{|\hat{p}|} - \frac{\epsilon|p|\hat{p}^T\alpha M^T p}{(1+\epsilon)|p|} \\
&= |p||\hat{p}| + \epsilon\left(\frac{1}{1+\epsilon}\right)\alpha p^T M\hat{p} - \frac{\epsilon}{1+\epsilon}\alpha\hat{p}^T M^T p \\
&= |p||\hat{p}|
\end{aligned}$$

By Wald's equation $E[\Phi]$ when the algorithm terminates is at most its initial value rc. Applying the Markov bound, with probability at least $1 - \frac{1}{rc}$, at termination $\max_i p_i \max_j \hat{p}_j \leq |p||\hat{p}| \leq (rc)^2$. By the argument in [4], then, Lemma 3 holds, i.e. $max_i y_i \leq N$ and $min_j \hat{y}_j \geq N(1-2\epsilon)$.

Lemma 4 shows that $y \approx Mx$ and $\hat{y} \approx M^T\hat{x}$. Since we have introduced an $O(\epsilon)$-factor adjustment to y, this is where we incur an extra ϵ-factor in the performance guarantee. To prove the lemma we use the Chernoff bound for random stopping times given in Lemma 10.

For part (1) we substitute $\Delta(\left(\frac{1}{1+\epsilon}\right))M_i x$ for x_t and Δy_i for y_t in Lemma 10. The proof holds as in [4], with the conclusion that with probability at least $1 - \frac{3}{rc}$, $\frac{|x^*|}{|\hat{x}^*|} = \frac{\min_j M_j^T \hat{x}}{\max_i M_i x} \geq 1 - ???\epsilon$.

8 Conclusions and future work

9 Appendix

9.1 Upper Bound on Sampler Norm

Because of the way item weights are calculated internally, the sampler norm can be upper-bounded by $4n$, where n is the number of items in the sampler. Without loss of generality, we show this by induction for the sampler p. First note that if t were equal to the maximum exponent of any item in p, then all items would be in *Small* and $|p|$, the norm of p, would be n. Now given any value of t, consider decrementing t by k to t', which transfers one bucket from *Small* to *Large* . Let s_t be the number of items in *Small* relative to t, and likewise for $s_{t'}$. Let $|p|'$ be the norm of p calculated using t' as the base exponent. Then we know that $|p|' \leq 2 \cdot |p|$, since moving t by k doubles the weights of all items in *Large* and at most doubles the weight of any item in *Small* by adding it to *Large* . But by our calculation of t, it must be the case that $|p| \leq 2 \cdot s_t$, else we would have already found the final value of t and would not need to decrement it further. Clearly $s_t \leq n$, and so we have that $|p|' \leq 4 \cdot s_t \leq 4 \cdot n$.

References

[1] N. Garg and J. Koenemann. Faster and Simpler Algorithms for Multicommodity Flow and other Fractional Packing Problems. *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 300–309, 1998.

[2] Michael D. Grigoriadis and Leonid G. Khachiyan. A sublinear-time randomized approximation algorithm for matrix games. *Operations Research Letters*, 18(2):53–58, 1995.

[3] T. Hagerup, K. Mehlhorn, and J. I. Munro. Optimal algorithms for generating discrete random variables with changing distributions. *Lecture Notes in Computer Science*, 700:253–264, 1993. Proceedings 20th International Conference on Automata, Languages and Programming.

[4] C. Koufogiannakis and N.E. Young. Beating simplex for fractional packing and covering linear programs. *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, pages 494–504, 2007.

[5] Michael Luby and Noam Nisan. A parallel approximation algorithm for positive linear programming. *Proceedings of the 25th annual ACM symposium on Theory of Computing*, pages 448–457, 1993.

[6] Y. Matias, J.S. Vitter, and W.C. Ni. Dynamic generation of discrete random variates. *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 361–370, 1993.