UNIVERSITY OF CALIFORNIA
RIVERSIDE

Topics in Approximation Algorithms

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Monik Khare

March 2013

The Dissertation of Monik Khare is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am sincerely grateful to my very supportive and understanding advisor, Prof. Neal E. Young, without whose help I would not have been able to write this thesis. He patiently helped me with my research and showed all great qualities one expects in a mentor. He let me work independently, but when needed, he sat with me for long discussions stretching over multiple hours trying to solve problems. His immense knowledge of the research topics was certainly helpful to me in making progress in my research. I was really fortunate to have him as my PhD advisor. I would also like to thank the other members of my dissertation committee, Prof. Marek Chrobak and Prof. Stefano Lonardi, for insightful feedback about my work and guiding me in the right directions.

I would like to thank my colleagues and friends Steve Cole and Li Yan. I worked with Steve on some research projects during the early days of my PhD. He often motivated me with his hard work and discipline, and it was really a pleasure to work with him. He is also a good friend and an amazing ex-roommate. I have talked to Li about many interesting problems, related to our research and otherwise, throughout my stay at UCR. I would also like to thank him for helping me improve my programming skills.

I am grateful to Prof. C. Ravishankar and Jonathan Dautrich, who gave me the opportunity to work on the BCOE Interactive Course Plan System. I enjoyed working on this project, which not only was a great learning experience, but also provided financial support. I also enjoyed puzzling over interesting problems, both in and out of the project, with Jonathan.

I would like to thank all my friends for their love and support. Coffee breaks with Akshay Morye and tennis sessions with Vijay Nagarajan significantly contributed to the rise in my happiness levels during my stay in Riverside. Special thanks to my friends Sudheer Sahu (my career advisor), Dushyant Mukkamala, Peter Lonjers, Casey Czechowski, Nathan Diep, Arman Yousefi, Pradyumna Goli, Jessica Ricci, Amarjeet Singh and Vikrant Verma.

Finally, and most importantly, I would like to thank my entire family. I would have not been able to accomplish this without their love and support. My wife, Gargi Kulkarni, has been a constant source of love, support, and encouragement, and a huge amount of credit for the completion of this thesis goes to her. Special thanks to my in-laws for their unwavering love and support.

To my parents, Mrs. Rita Khare and Mr. S. P. Khare,

my wife, Gargi Kulkarni,

and

my daughter, Myra

ABSTRACT OF THE DISSERTATION

Topics in Approximation Algorithms

by

Monik Khare

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2013
Dr. Neal E Young, Chairperson

This thesis focuses on approximation and online algorithms for a few different problems.

1. There have been continued improvements in the approximation algorithms for packing and covering problems. Some recent algorithms, including Fastpc [45], have provable good worst-case running times, but it is not known how they perform in practice compared to the simplex and interior-point algorithms that are widely used to solve these problems. We present an empirical comparison of these algorithms, using our own implementation of Fastpc [2] and CPLEX implementations of simplex and interior-point methods. We use a variety of inputs for this experimental study. We find that Fastpc is slower for small problem instances, but its performance, relative to CPLEX algorithms, improves as the instances get bigger. Our experiments show that for reasonably large inputs Fastpc performs better than the CPLEX algorithms.

2. We give deterministic algorithms for some variants of online file caching by reducing the problems to online covering. The variants considered in this study include one or both of the following features: (i) a *rental cost* for each slot occupied in

the cache, and (ii) *zapping* a file by paying a cost so that the *zapped* file does not occupy any space in the cache and does not incur any retrieval cost. The rental cost is motivated by the idea of energy efficient caching where caching systems can save power by turning off slots not being used to store files [15]. Our approach is based on the online covering algorithm by Koufogiannakis and Young [46]. We give deterministic lower bounds for these variants, which are tight within constant factors of the upper bounds for most of the cases. We also give randomized algorithms and lower bounds for the variants with rental cost.

3. We introduce online Huffman coding. In Huffman coding, the symbols are drawn from a probability distribution and revealed one by one, and the goal is to find a minimum cost prefix-code for the symbols. In the online version, the algorithm has to assign a codeword to a symbol when it is revealed for the first time. We propose an online greedy algorithm and show that it is constant-competitive for online Huffman coding. We also show a lower bound of 10/9 on the competitive ratio of any deterministic online algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Approximation algorithms are algorithms that compute approximate solutions to optimization problems. These algorithms usually run in polynomial time and provide a guarantee on the quality of the solution (i.e. cost of the solution respect to the optimal cost). The *approximation ratio* for an algorithm is defined as follows. Let ALG(I) be the cost of the algorithm ALG on input I and let OPT(I) be the cost of the optimal solution. ALG is an α-approximation algorithm for a maximization (or, minimization) problem if, for each input I, $\mathrm{ALG}(I) \geq \mathrm{OPT}(I)/\alpha$ (or, $\mathrm{ALG}(I) \leq \alpha \cdot \mathrm{OPT}(I)$). The approximation ratio is a measure of the quality of the solution in the worst case scenario.

There has been a lot of research on designing and analyzing approximation algorithms for various optimization problems [28, 32, 33, 34, 45, 53]. Approximation algorithms are sometimes used even for problems where polynomial time algorithms exist but are too slow for large instances. In some cases, approximation algorithms are used to quickly compute a near-optimal solution which can be used by an exact algorithm as a starting point to compute the optimal solution. For instance, the barrier algorithm in CPLEX, a commercial tool for solving linear programs, uses the interior-point method to compute an approximate solution, and then uses primal-simplex to start with the approximate solution and compute the optimal solution.

We also study some problems in the *online* setting where the input is revealed step-by-step. An *online* algorithm has to make a decision at each step without any knowledge of the input revealed at any future steps. Online algorithms are analyzed using the competitive-analysis framework [55] where the cost of the solution computed by the algorithm is compared to the optimal offline solution (the entire input is known in advance). Analogous to the approximation ratio for approximation algorithms, the *competitive ratio* for online algorithms is defined as follows. The input is defined as a sequence σ, where $\sigma(t)$ denotes the input revealed at time t. Let $\mathrm{ALG}(\sigma)$ be the cost of algorithm ALG on input sequence σ, and let $\mathrm{OPT}(\sigma)$ be the corresponding optimal offline cost. ALG is an α-competitive online algorithm for a maximization (or, minimization) problem if, for every input sequence σ, $\mathrm{ALG}(\sigma) \geq \mathrm{OPT}(\sigma)/\alpha + c$ (or, $\mathrm{ALG}(\sigma) \leq \alpha \cdot \mathrm{OPT}(\sigma) + c$), where c is a constant independent of the request sequence.

There has been substantial amount of growth in the research on online algorithms in the last two decades [5, 7, 25, 38, 55, 62, 65]. They are of great interest also because of their many practical applications, including resource management in operating systems, online advertising, and network routing.

Many techniques that are used to design and analyze approximation algorithms can be used to design and analyze online algorithms. Such approaches include greedy, primal-dual, and randomized rounding.

This thesis focuses on approximation and online algorithms for a few problems. This chapter presents a brief overview of these problems and summarizes some key results. We describe these problems, the related literature, and our work in more detail in the respective chapters.

1.1 Empirical study of algorithms for packing and covering

Explicitly given packing and covering linear programs are of the form $\max\{a \cdot x : Mx \leq b, x \geq 0\}$ and $\min\{a \cdot x : Mx \geq b, x \geq 0\}$, respectively, where all entries in the vectors a and b and in the matrix M are non-negative.

Various algorithms for solving linear programs, including the well known simplex algorithm [21], can be used to solve these packing and covering linear programs as well. These algorithms don't have good worst-case running times, but are still used widely as they are believed to work well in practice. There has been a lot of research on approximation algorithms for these problems. The fast asymptotic running times of some recent approximation algorithms, including FASTPC [45], motivate this empirical study. We evaluate FASTPC to determine if these approximation algorithms have come closer to matching or improving the performance of the traditional algorithms, such as simplex and interior-point algorithms[39], in practice. FASTPC [45] computes an $(1 \pm \epsilon)$-solution for any $\epsilon > 0$ and runs in $[n + (r + c) \log n/\epsilon^2]$ time with a high probability, where r is the number of rows (constraints), c is the number of columns (variables), and n is the number of non-zeros in the constraint matrix M. For dense inputs, where r and c are $O(\sqrt{n})$, the running time of FASTPC is linear in the number of non-zeros.

We evaluate the performance of a these algorithms for solving pure packing or covering linear programs on a variety of inputs. We use CPLEX for the implementations of primal-simplex, dual-simplex, and interior-point (barrier) algorithms. We refer to these algorithms as the CPLEX algorithms. We use our own implementation of FASTPC [2].

Koufogiannakis and Young [45] present empirical results comparing FASTPC to primal-simplex, but their work has certain limitations. These limitations include

(a) only random and relatively small input instances and (b) using a simpler and faster implementation of FASTPC that handles only 0/1 coefficients. We address these as well as a few other limitations (refer to Section 2.1.2) in our study. We use set cover benchmark inputs [3, 61], DIMACS inputs [1], random inputs, and finally, inputs generated by simulating X-Ray tomography on 2-dimensional images. We report various new findings. We observe that FASTPC is faster than CPLEX algorithms on some input types, when the inputs are reasonably large. For random and tomography inputs, FASTPC catches up with CPLEX algorithms at around $r + c > 17000$ when density > 0.15, and around $r + c > 32000$ when $0.01 \leq$ density ≤ 0.15, and around $r + c > 77000$ when density < 0.01.

The barrier algorithm (CPLEX implementation) runs in two phases. (a) In the pre-crossover phase, it runs an interior-point algorithm to compute a near-optimal solution, and (b) in the post-crossover step, it runs simplex to start with the near-optimal solution and compute the optimal solution. We propose and analyze the hypothetical hybrid algorithm that uses FASTPC instead of the interior-point algorithm in the first phase of the barrier method.

We present this work in Chapter 2 of this thesis.

1.2 File caching with rental cost and zapping

The *file caching* problem is defined as follows. Given a cache of size k (a positive integer), the goal is to minimize the total retrieval cost for the given sequence of requests to files. A file f has size $size(f)$ (a positive integer) and retrieval cost $cost(f)$ (a non-negative number) for bringing the file into the cache. A *miss* or *fault* occurs when the requested file is not in the cache and the file has to be retrieved into the cache by paying the retrieval cost, and some other files may have to be removed (*evicted*) from the cache so that the total size of the files in the cache does not exceed k. Paging is the special case when each file has size 1 and the retrieval cost for each file is 1. *Bit model* and *fault model* are special cases of the file caching problem where, for each file f, $cost(f) = size(f)$ and $cost(f) = 1$, respectively.

We study the following variants of the online file caching problem. ***Caching with Rental Cost*** **(or** ***Rental Caching*****)**: There is a rental cost λ (a positive number) for each file in the cache at each time unit. The goal is to minimize the sum of the retrieval costs and the rental costs. ***Caching with Zapping***: A file can be *zapped* by paying a zapping cost N. Once a file is zapped, it does not use any space in the cache and all future requests to it don't incur any cost. The goal is to minimize the sum of the retrieval costs and the zapping costs. We study these two variants and also the variant which combines these two (rental caching with zapping). We present deterministic and randomized lower and upper bounds in the competitive-analysis framework.

Koufogiannakis and Young [46] give a Δ-approximation algorithm for covering with submodular cost, where Δ is the maximum number of variables in any constraint. Their algorithm is Δ-competitive for the online case, where the constraints are revealed one at a time. Many online problems, including the ones we study in this work, can be reduced to online covering and the online covering algorithm in [65] can be applied. We study and explore this approach for caching with rental cost and zapping and its variants. We find that this approach can be extended to these problems, but in some cases the algorithms thus derived are sub-optimal. We are able to apply modifications to the online covering algorithm to yield improved competitive ratios in some cases. We discuss these scenarios and the modifications. We also present randomized lower and upper bounds for these problems. For most cases, the deterministic and randomized bounds shown in this work are tight within constant factors. Table 3.1 summarizes our results.

In our work, we give deterministic online algorithms for variants of file caching with rental cost and zapping by reducing them to online covering programs. There has been some research on the online primal-dual approach to give deterministic and randomized algorithms for the file caching problems and its variants [5, 7, 8, 13, 23]. The FASTPC algorithm also uses a primal-dual analysis, thought it is not an online algorithm.

We present this work in detail in Chapter 3.

1.3 Online Huffman Coding

We introduce *Online Huffman Coding.* The online Huffman coding problem is an online version of the Huffman coding problem. Huffman coding is defined as follows. Given a probability distribution $P = \{ p_1, p_2, \cdots, p_n \}$ on $[n]$ and an encoding alphabet Σ, find a minimum cost prefix-free code (or prefix code) over Σ. The cost of a prefix code χ is given by $\sum_{i=1}^{n} p_i|\chi_i|$, where χ_i is the codeword assigned to i and $|\chi_i|$ is the length of χ_i. We assume that $0 < p_i \leq 1$ for each i. Huffman [35] gives optimal algorithm for this problem. In the online version, the symbols are drawn from a probability distribution P and are revealed one by one. The algorithm does not have any knowledge of the probability distribution or the future sequence of symbols, and has to assign codewords *online.*

The online Huffman coding problem is different from the *adaptive Huffman coding* problem. In adaptive Huffman coding, the code is built dynamically as the symbols are revealed one by one. As the symbols are revealed, the algorithm maintains the weights and the corresponding Huffman code. Unlike Online Huffman Coding, the codeword assignment maybe updated at any step when a symbol is revealed [24, 27, 43, 59, 60].

We study the online Huffman coding problem for binary codes in the competitive analysis framework. We show a lower bound of 10/9 on the competitive ratio of any deterministic algorithm. We present a greedy algorithm for online Huffman coding and analyze it. We show that the algorithm is 7-competitive. We also show that the algorithm is $(1 + o(1))$-competitive asymptotically.

We present this work in Chapter 4 of this thesis.

Chapter 2

Empirical study of algorithms for packing and covering

2.1 Introduction

In this study we focus on algorithms for solving explicitly given packing and covering linear programs. Explicitly given packing and covering linear programs are of the form $\max\{a \cdot x : Mx \leq b, x \geq 0\}$ and $\min\{a \cdot x : Mx \geq b, x \geq 0\}$, respectively, where all entries in the vectors a and b and in the matrix M are non-negative. Various important problems can be specified as explicitly given packing and covering linear programs, for instance, variants of multicommodity flow problems and the fractional set cover problem.

We use the following notation throughout this chapter:

- r : number of rows (constraints)
- c : number of columns (variables)
- n : number of non-zeros in the constraint matrix

2.1.1 Traditional algorithms for solving linear programs

Traditional algorithms for solving linear programs, like simplex algorithm and ellipsoid method, can be used to solve packing and covering linear programs as well. There are many algorithms for solving linear programs. We briefly discuss some of these well known algorithms in this section.

The Simplex method was proposed by Dantzig [21]. The simplex algorithm doesn't have a good worst-case bound, but is considered to work well in practice. In 1982 Smale [56] showed that for fixed number of constraints the number of pivots required to solve a linear program grows linearly in number of variables on the average. In 2004 Kelner and Spielman [40] proposed a randomized implementation of the simplex algorithm that runs in polynomial time.

The ellipsoid method by Kozlov et al. [47] has a polynomial running time for solving convex optimizations, but in practice is slower than the interior-point and simplex algorithms. The worst case running time of ellipsoid method is $O(c^6L)$, where L is a measure of the bit complexity of the input that is known to be polynomial in the input size.

In 1984 Karmarkar [39] published a polynomial time interior-point algorithm for general linear programs. The running time of this algorithm is $O(c^{3.5}L)$, where L is the measure of the bit complexity of the input. This running time is an improvement by a factor of $O(c^{2.5})$ over the ellipsoid algorithm and, more importantly, the performance in practice is competitive with the simplex algorithm. Karmarkar's algorithm also inspired more research on the interior-point algorithms.

In 1989 Vaidya [58] presented an algorithm to solve linear programming problems. The algorithm performs a total of $O(rc^2L + cM(r)L)$ operations, where L is the measure of the bit complexity of the input and $M(r)$ is the time to invert an $r \times r$ matrix. Since the current fastest algorithm for computing the inverse of an $m \times m$ matrix runs in $O(r^{2.376})$ time [20], Vaidya's algorithm is faster than Karmarkar's algorithm when $r \in O(c^{1.05})$.

The simplex algorithm is used very extensively for solving linear programs, but there is very limited literature available on how it performs in practice. Interior point methods are also used widely, but again, not much information is available on their performance in practice.

2.1.2 Approximation algorithms for packing and covering

There has also been a long line of research focused on developing fast approximation algorithms for solving packing and covering problems [28, 32, 33, 34, 45, 53]. These approximation algorithms, unlike the traditional algorithms listed above, are developed using techniques like Lagrangian relaxation. Lagrangian relaxation is a very general and powerful technique and has been used for developing algorithms in many different areas [6, 10, 57].

Over the years, there have been continued improvements in approximation algorithms for packing and covering problems. Some of the recent algorithms, including Fastpc [45], have provable worst-case running times that are superior to the worst-case running times bounds for the traditional algorithms for linear programming, but it is not known if these approximation algorithms work as well in practice. There isn't much information available in the literature on how these approximation algorithms compare to the traditional algorithms like simplex and interior-point algorithms in practical scenarios. In this work, we do an empirical study of the Fastpc approximation algorithm, proposed by Koufogiannakis and Young [45], and compare its performance primal-simplex, dual-simplex, and interior-point algorithms. Our experiments include various types of inputs (refer to Section 2.2.4) to explore if FASTPC can match or improve the performance of these traditional algorithms for any particular classes of inputs.

FASTPC computes a $(1+\epsilon)$ approximate solution (or, $(1\pm\epsilon)$-solution) for any $\epsilon > 0$, and with a high probability runs in $O(n+(r+c)\log n/\epsilon^2)$, where n is the number of non-zeros, r is the number of rows, and c is the number of columns in the input matrix. Koufogiannakis and Young [45] also present some experimental results where they evaluate the primal-simplex algorithm (GLPK) and a preliminary implementation of FASTPC. They observed that FASTPC was significantly faster than the primal-simplex algorithm even for moderately large inputs (about 2000x2000). Their work had certain limitations that we address in our study.

- They evaluated only the primal-simplex algorithm. Their study didn't include dual-simplex and interior-point algorithms.
- Inputs used were relatively small (up to 4000x4000, 8000x2000). We include much larger inputs (e.g. 23000x23000, 100,000x5000).
- Only a few inputs were sparse. The smallest density used was 0.008 and only a small number of inputs had such small densities. Our study includes many inputs that are sparse ($d < 0.005$).
- Their implementation handled inputs with 0/1 coefficients only. The implementation we use in our study is more complex and handles all pure packing and covering inputs with non-negative real coefficients. Their simplified implementation is faster as compared to our implementation by constant factors.
- All the inputs used in their study were random. It is believed that CPLEX runs

faster on structured problems. Our study includes various types of inputs (refer to Section 2.2.4).

- They used GLPK for their experiments. We use CPLEX, which we found to be faster than GLPK by constant factors, for the implementations of primal-simplex, dual-simplex, and interior-point algorithms.

The rest of the chapter is structured as follows. In the next section we talk about the experimental setup for this study. Then we present the results of these experiments and our conclusions.

2.2 Experimental setup

2.2.1 Machine specifications

All experiments were performed using Lenovo ThinkStation E30 series with Intel Xeon E3-1220 Processor (3.10GHz 8MB) and 8 GB RAM [4 $\times$ 2GB ECC DDR3 PC3-10600 SDRAM (1333MHz uDIMM)].

2.2.2 Fastpc

We implemented the FASTPC algorithm using C++ [2]. As opposed to the preliminary implementation that handles inputs with only 0/1 coefficients, our FASTPC implementation works for all pure packing and covering inputs with non-negative real coefficients.

Cole et al. [19] provide key details related to the implementation of FASTPC we use for our experiments. The algorithm maintains a pseudo-distribution (a distribution that isn't normalized) and samples from it. The two key issues that come up with this random sampling scheme are efficiency and precision. Cole et al. [19] propose an efficient data structure for this random sampling scheme. This data structure maintains the values approximately. This requires some modifications in the algorithm so that the approximation guarantee still holds. They also present the modified Fastpc algorithm and the modified analysis of the approximation ratio.

2.2.3 CPLEX

We used ILOG CPLEX Optimization Studio Academic Research Edition (version 12.3.0, with academic license from IBM) for implementations of the primal-simplex, dual-simplex, and barrier methods. We refer to these three as the CPLEX algorithms.

2.2.4 Inputs

We compare the algorithms on variety of inputs. The following are the classes of inputs that we used for our experiments.

- **Random:** The coefficients were chosen uniformly from a range.
- **DIMACS:** DIMACS [1] graphs were used to create vertex cover instances on bipartite graphs. For a given graph, a bipartite graph can be constructed as follows. For every vertex u in the original graph, create two vertices u_1 & u_2. For every edge (u, v) in the original graph, create edges (u_1, v_2) and (u_2, v_1). The linear program used for experiments corresponds to the vertex cover in this bipartite graph.
- **Set packing:** The benchmark instances from [3] were used to generate packing linear programs. We also generated larger instances using the generator described in Xu et al. [61].

- **Tomography:** X-Ray tomography image reconstruction can be formulated as a mixed packing covering linear program. We simulated X-Ray tomography on 2-dimensional images and used the data to generate instances of pure packing linear program.

All the DIMACS and set packing inputs are highly asymmetric (e.g. $r = 100000$ and $c = 1000$). For all these instances, the number of variables is much smaller than the number of constraints. Tomography inputs are also asymmetric. The random inputs we generated have some asymmetric instances and some symmetric ones.

The *density* of an input is $d = \frac{n}{rc}$. For sparse inputs, $r + c$ is $\Omega(n)$ and for dense inputs $r + c$ is $O(\sqrt{n})$. All the DIMACS and set packing inputs are very sparse. Random inputs used in this study are a mix of sparse, medium dense, and dense inputs. The tomography inputs are medium dense.

2.2.5 Value of ϵ

The value of ϵ used for our experiments is 0.01.

2.3 Results

In this section we present our findings. First, we present the observed lower bounds for the CPLEX algorithms (primal-simplex, dual-simplex, and barrier) for various input types. We give these bounds for computing an $(1 \pm \epsilon)$-solution as well as for computing the optimal solution and we compare the performance of these algorithms. Next, we compare FASTPC to the CPLEX algorithms to compute $(1 \pm \epsilon)$-solution for these inputs.

All the plots in this chapter are semi-log (log-lin) plots, unless stated otherwise.

2.3.1 CPLEX algorithms

For any Gaussian-elimination based method to solve system of linear equations, the best known bound is $\tilde{O}(\min(r,c)^3(||M|| + ||b||))$ [22], where $\tilde{O}$ hides polylogarithmic factors and $||D||$ represents the highest absolute value in matrix D. The simplex algorithm, in each iteration, uses Gaussian elimination to transform the linear program from one configuration to the other. The CPLEX implementation of the barrier method runs in two phases. In the first phase the interior-point algorithm computes an approximate solution $O(c^{3.5}L)$ (L is the bit complexity of the input) and in the second phase primal-simplex starts with the approximate solution computed in the first phase and finishes with the optimal solution. In general, these algorithms, due to their dependence on the Gaussian elimination, should take at least $\Omega(\min(r,c)^3)$ for dense matrices. For sparse matrices, these algorithms should take at least $\Omega(\min(r,c)^2\delta)$, where δ is the maximum number of non-zeros in any row. We study the experimental running times for each algorithm on different types of inputs to see how these algorithms perform in practice with respect to these bounds. We are also interested in finding out if there are particular types of inputs where these algorithms perform much worse than these lower bounds. We study these running times to compute $(1 \pm \epsilon)$-solutions as well as for computing exact solutions.

In this section, the approximate solutions correspond to $\epsilon = 0.01$. The primal and dual simplex algorithms do not have the knowledge of the approximation ratio of the solution at any stage of the execution. The time taken to reach a particular approximation ratio can be computed only with the knowledge of the optimal solution. We determined the time for these algorithms to compute the approximate solution for a given ϵ by using the logs.

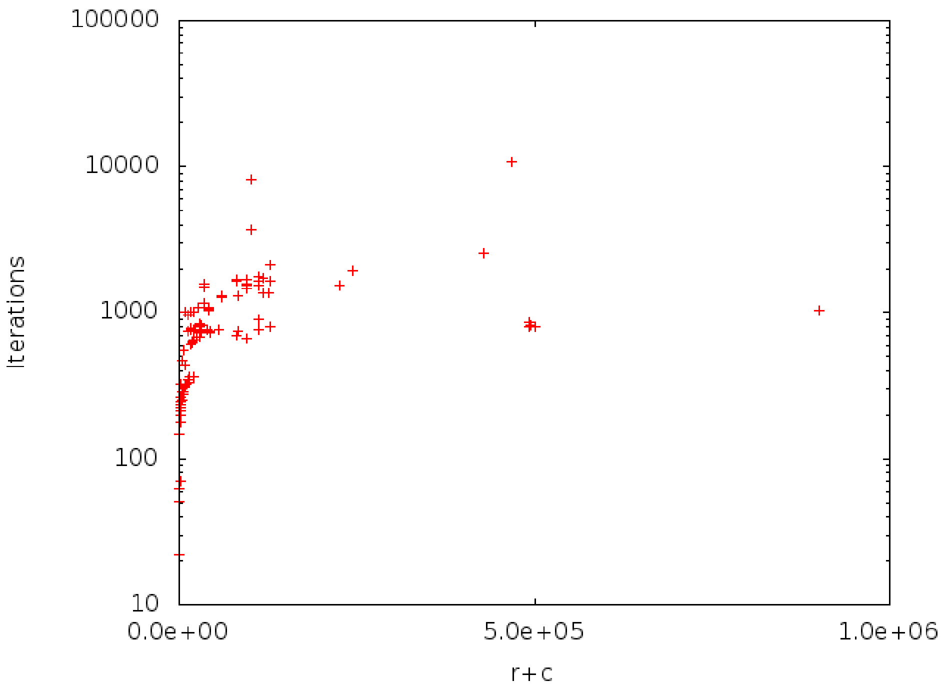
2.3.1.1 Set packing and DIMACS inputs

Table 2.1 summarizes the observed lower bounds on the number of iterations, time per iteration, and total time for CPLEX algorithms for set packing and DIMACS inputs. We show the plots supporting these observed lower bounds in Figure 2.1 and Figure 2.2 for primal-simplex, Figure 2.3 and Figure 2.4 for dual-simplex, and Figure 2.5 and Figure 2.6 for the barrier method.

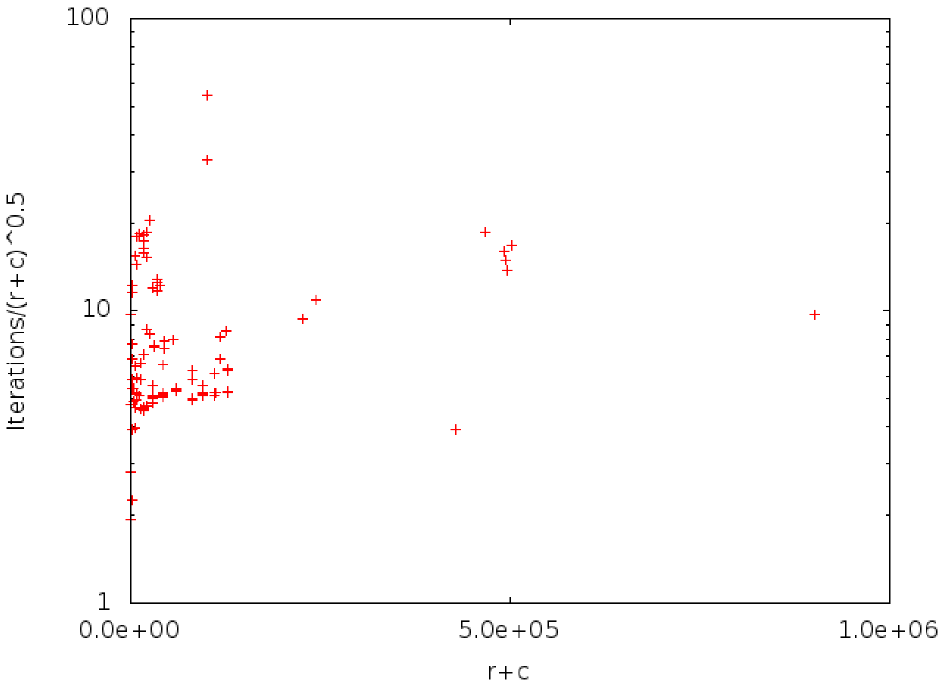
For almost all of these instances, dual-simplex was found to be faster than primal-simplex and barrier methods by constant factors (Figure 2.7). For these inputs, all three algorithms took $\Omega((r{+}c)^1.5)$ time to compute an $(1{\pm}\epsilon)$-solution. For computing exact solutions, dual-simplex was found to be faster than primal-simplex and barrier algorithms by a factor of $O((r+c)^0.5)$.

Algorithm		$(1 \pm \epsilon)$-**solution**	**Exact solution**
primal-simplex	Iterations	$O(1)$	$(r+c)^{0.5}$
	Time/Iteration	$(r+c)^{1.5}$	$(r+c)^{1.5}$
	Time	$(r+c)^{1.5}$	$(r+c)^{2}$
dual-simplex	Iterations	$(r+c)^{0.5}$	$(r+c)^{0.5}$
	Time/Iteration	$(r+c)$	$(r+c)$
	Time	$(r+c)^{1.5}$	$(r+c)^{1.5}$
barrier	Iterations	$3-6$	$5-10$
	Time/Iteration	$(r+c)^{1.5}$	$(r+c)^{2}$
	Time	$(r+c)^{1.5}$	$(r+c)^{2}$

Table 2.1: Observed lower bounds for CPLEX algorithms on set packing and DIMACS inputs

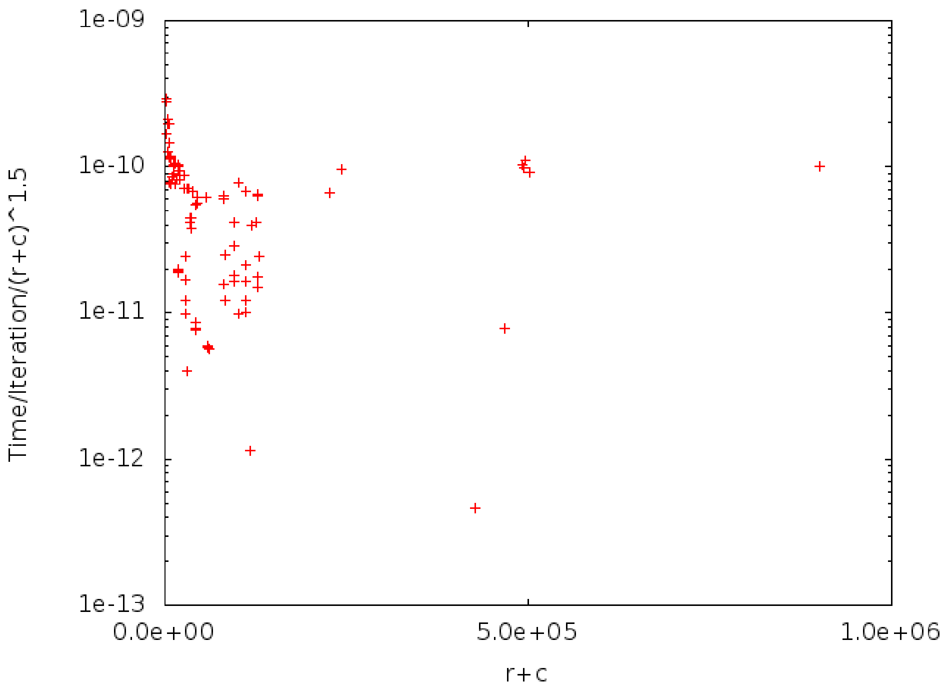


(a) $(1 \pm \epsilon)$-solution

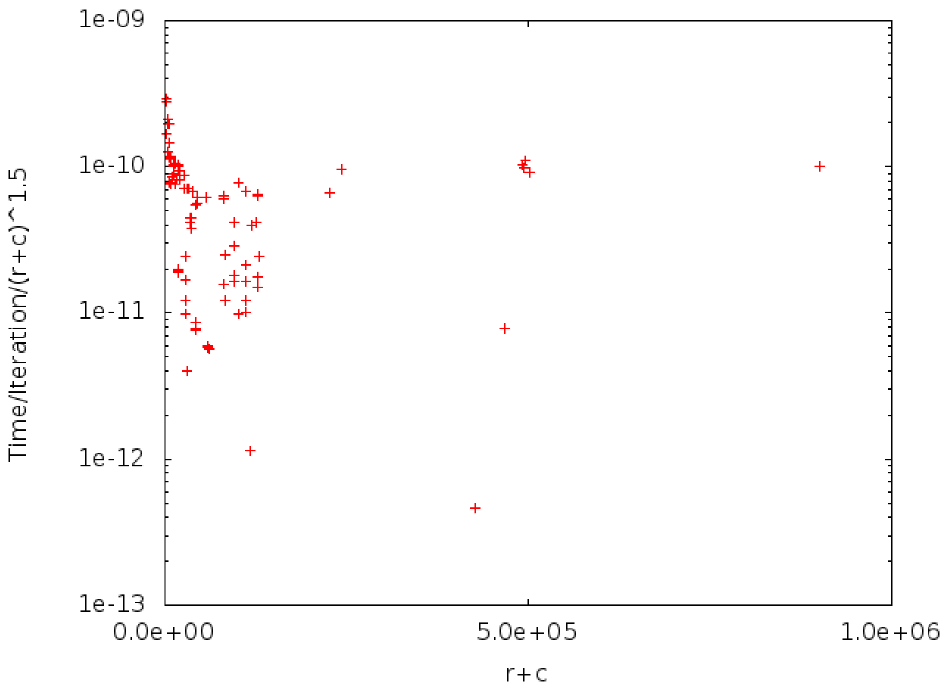


(b) Exact solution

Figure 2.1: Iterations for primal-simplex on set packing and DIMACS inputs

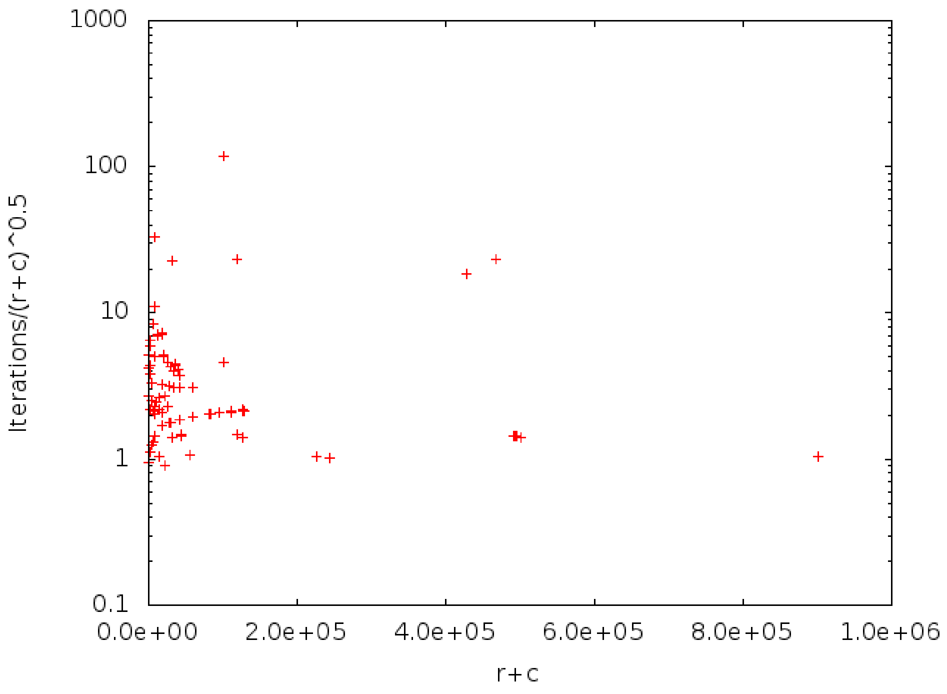


(a) $(1 \pm \epsilon)$-solution

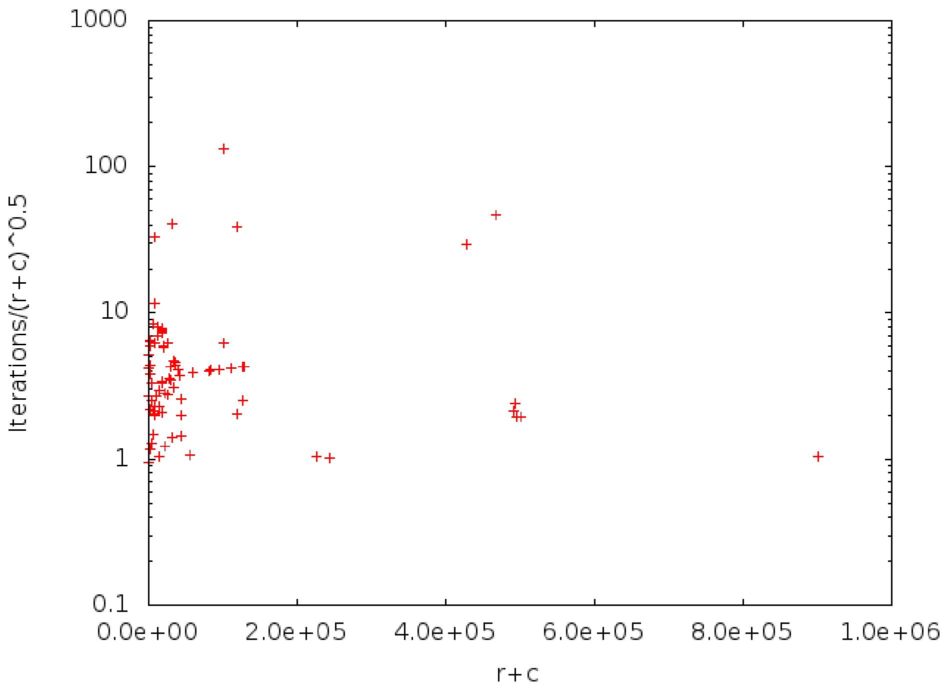


(b) Exact solution

Figure 2.2: Time per iteration for primal-simplex on set packing and DIMACS inputs

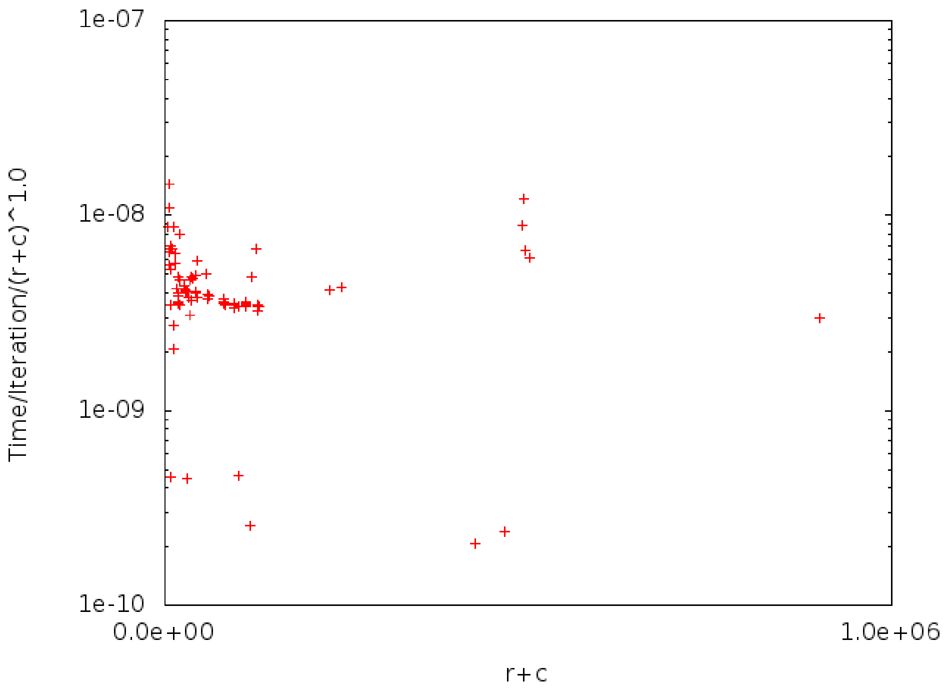


(a) $(1 \pm \epsilon)$-solution

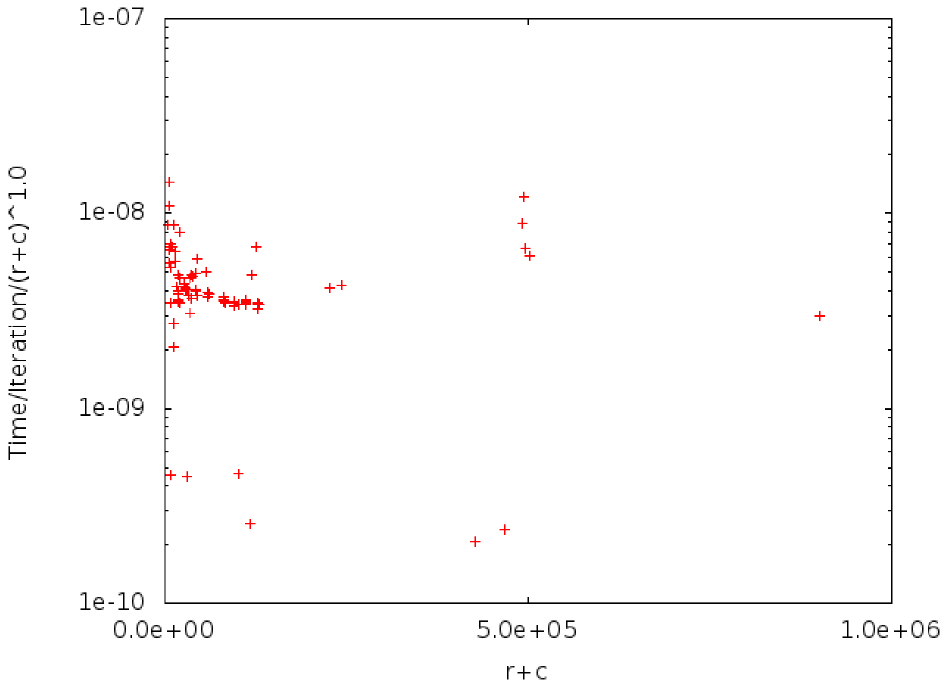


(b) Exact solution

Figure 2.3: Iterations for dual-simplex on set packing and DIMACS inputs

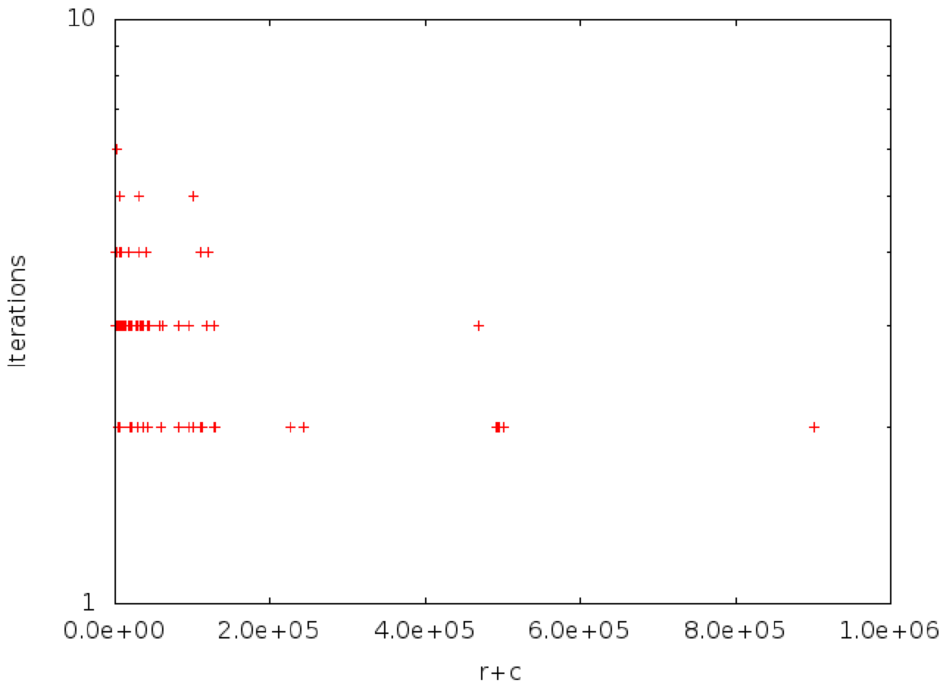


(a) $(1 \pm \epsilon)$-solution

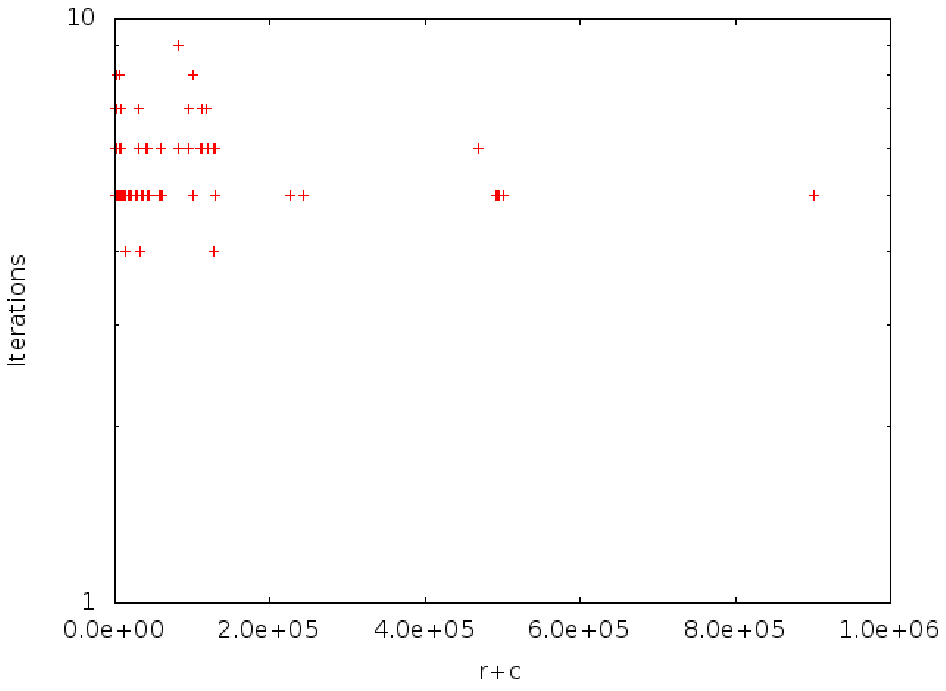


(b) Exact solution

Figure 2.4: Time per iteration for dual-simplex on set packing and DIMACS inputs

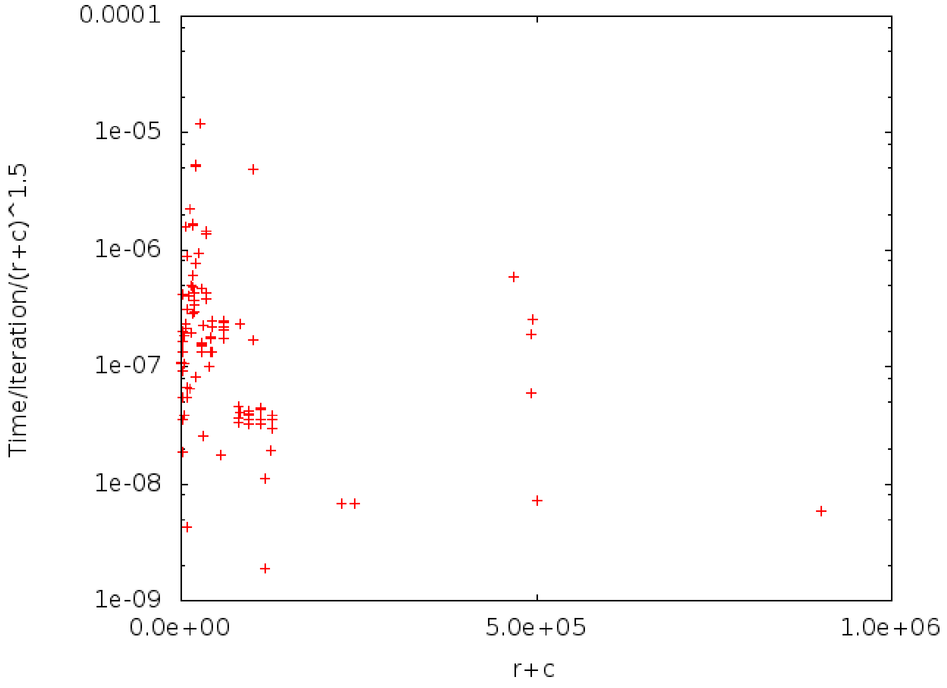


(a) $(1 \pm \epsilon)$-solution

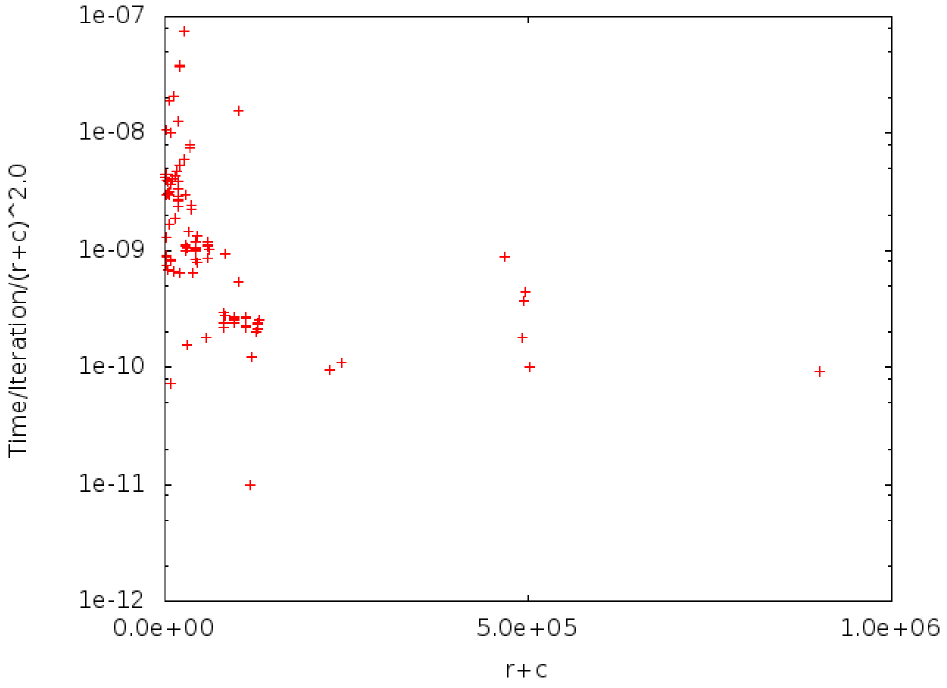


(b) Exact solution

Figure 2.5: Iterations for barrier-simplex on set packing and DIMACS inputs



(a) $(1 \pm \epsilon)$-solution



(b) Exact solution

Figure 2.6: Time per iteration for barrier-simplex on set packing and DIMACS inputs

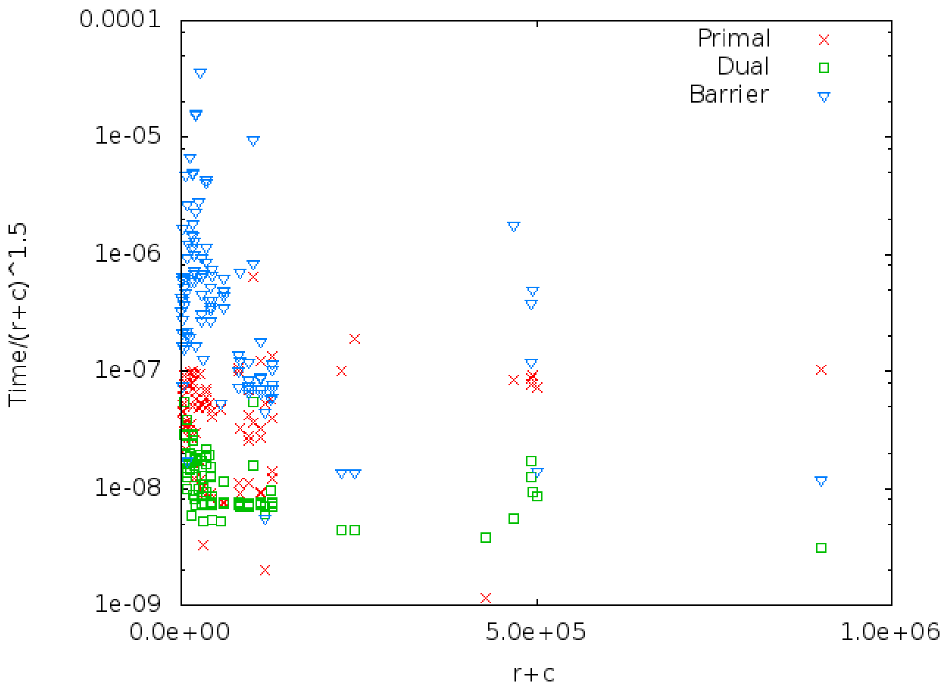


Figure 2.7: Comparison of CPLEX algorithms to compute $(1 \pm \epsilon)$-solution for $\epsilon = 0.01$ on set packing and DIMACS inputs

2.3.1.2 Random and tomography inputs

Table 2.2 summarizes the observed lower bounds on the number of iterations, time per iteration, and total time for CPLEX algorithms for set packing and DIMACS inputs. We show the plots supporting these observed lower bounds in Figures 2.8, 2.9, 2.10, and 2.11 for primal-simplex, Figures 2.12, 2.13, 2.14, and 2.15 for dual-simplex, and Figures 2.16 and 2.17 for the barrier method.

We observe that the algorithms, including FastPC, are slower on these inputs than on set packing and DIMACS inputs. We don't know the reason behind this behavior. One possible explanation is that even though the set packing and DIMACS inputs are considered to be hard instances for computing integer solutions, they may not be hard for computing fractional solutions.

Unlike the set packing and DIMACS instances, the random and tomography inputs had varied densities. Experiments show that all three algorithms perform better for sparse inputs than dense inputs of the same size. For sparse inputs, all three algorithms took $\Omega((r+c)^3)$ time to compute $(1 \pm \epsilon)$-solution, but primal and dual simplex were found to be slower than barrier on dense instances. Barrier performed better than primal and dual simplex algorithms for computing approximate solutions. For computing exact solutions, dual-simplex performed better than the other two algorithms by a factor of $(r+c)^{0.5}$ on dense inputs. On sparse inputs, primal and dual simplex were much faster than barrier.

Algorithm		$(1 \pm \epsilon)$**-solution**		**Exact solution**	
		$d < 0.01$	$d \geq 0.01$	$d < 0.01$	$d \geq 0.01$
primal-simplex	**Iterations**	$(r+c)^{1.5}$	$(r+c)^{2.5}$	$(r+c)^{1.5}$	$(r+c)^{2.5}$
	Time/Iteration	$(r+c)^{1.5}$	$(r+c)^{2.5}$	$(r+c)^{1.5}$	$(r+c)^{2.5}$
	Time	$(r+c)^3$	$(r+c)^5$	$(r+c)^3$	$(r+c)^5$
dual-simplex	**Iterations**	$(r+c)^{1.5}$	$(r+c)^2$	$(r+c)^{1.5}$	$(r+c)^2$
	Time/Iteration	$(r+c)^{1.5}$	$(r+c)^{2.5}$	$(r+c)^{1.5}$	$(r+c)^{2.5}$
	Time	$(r+c)^3$	$(r+c)^{4.5}$	$(r+c)^3$	$(r+c)^{4.5}$
barrier	**Iterations**	$5-9$	$5-9$	$(r+c)^3$	$(r+c)^3$
	Time/Iteration	$(r+c)^3$	$(r+c)^3$	$(r+c)^2$	$(r+c)^2$
	Time	$(r+c)^3$	$(r+c)^3$	$(r+c)^5$	$(r+c)^5$

Table 2.2: Observed lower bounds for the primal-simplex and the dual-simplex algorithms on random and tomography inputs

2.3.2 Fastpc

The experimental runs show that our implementation of FASTPC is slower (by constant factor of around 4) than the simpler implementation (only 0/1 coefficients) used in [45]. The running time of FASTPC is well predicted by the expression $[1.2n + 5(r + c)\log n/\epsilon^2]$ (appropriately scaled) for the random and tomography instances (Figure 2.19b). For set packing and DIMACS instances, the behavior isn't well predicted for small instances, but as the instances grow bigger, the running time starts to match this expression (Figure 2.19a). [45] report that the average time per operation was not constant and it grew with large instances by up to a factor of two. We also observe the same phenomenon in our experimental runs, but we do not understand why this happens.

For dense problems where r and c are $O(\sqrt{n})$, the running time is $O(n + \sqrt{n}\log(n)/\epsilon^2)$ which is linear in the size of the input even for very small ϵ. In our experiments, the running times for dense matrices were dominated by the second term in the expression above.

2.3.3 CPLEX vs Fastpc

We give the running times of all algorithms for set packing and DIMACS inputs in Table A.1 and random and tomography inputs in Table A.2. If an algorithm did not finish computing the solution (approximate or optimal), the time is indicated by a "-". In some cases if an algorithm had been running for a long time and was not making enough progress, the execution was stopped. For large random and tomography instances primal and dual simplex performed very poorly and were not included in some of the final runs with very large inputs.

Note that our results are significantly different from the empirical results in [45]. As pointed out in Section 2.1.2, the implementation used in their study is simpler and faster by constant factors. We find that for the sizes considered in their study, our implementation of Fastpc does not perform better than CPLEX algorithms. This is also because our implementation is compared to the CPLEX implementations of simplex and interior-point algorithms, while their experiments used GLPK, which is considerably slower than CPLEX. Moreover, our experiments included inputs which were very sparse. We find that the CPLEX algorithms perform better on sparse inputs and the gap between Fastpc and CPLEX algorithms is higher. A large number of sparse instances contribute to a bigger gap between the performance of Fastpc and that of CPLEX algorithms as compared to the previous study.

We observe that all CPLEX algorithms are faster than Fastpc on set packing and DIMACS inputs. All three CPLEX algorithms compute the $(1 \pm \epsilon)$-solution within at most 6 minutes and the optimal solution in under an hour on each input. The set packing and DIMACS instances were designed to make the computation of the integer optimal solution hard. It is possible that computing fractional solutions for these instances is easy for the algorithms considered in this study. In such cases, where CPLEX algorithms compute the solutions quickly, it is unlikely that Fastpc will perform better. We used the generator for set packing inputs to create larger instances, but CPLEX algorithms were still fast on those inputs. We were not able to generate inputs large enough where CPLEX algorithms are slow (take more than a few hours for computing approximate solutions). We can't conclude based on this data if Fastpc will eventually perform better than CPLEX algorithms for larger instances of these input types.

There is a different trend for random and tomography instances. We see that Fastpc competes better with the CPLEX algorithms for these instances compared to the DIMACS and set packing inputs. FASTPC catches up with CPLEX algorithms at around $r + c > 17000$ when density > 0.15, and around $r + c > 32000$ when $0.01 \leq$ density ≤ 0.15, and around $r + c > 77000$ when density < 0.01 (Figure 2.22).

2.3.4 Hybrid Algorithm

The barrier algorithm runs in two phases. In the first phase (pre-crossover) the interior-point algorithm computes an approximate solution and in the second phase (post-crossover) primal-simplex starts with the approximate solution computed in the first phase and finishes with the optimal solution. Since FASTPC is faster than barrier for reasonably large instances, this motivates the possibility of a faster hybrid algorithm that uses FASTPC instead of interior-point algorithm for the pre-crossover phase. It first runs FASTPC to compute a $(1 - \epsilon)$-approximate solution for small ϵ then finishes (as in the barrier algorithm) by crossing over and running iterations of primal-simplex until an optimal solution is found.

Is it possible that by an appropriate choice of ϵ this hypothetical hybrid algorithm could be faster than the existing barrier algorithm?

In the existing barrier algorithm the time is spent roughly equally in the pre-crossover and post-crossover phases. To substantially reduce the running time, the proposed hybrid algorithm would have to find, in the first phase, a $(1 + \epsilon)$-approximate solution with ϵ substantially smaller than is found by the current barrier algorithm at crossover.

In all of our current trials the crossover occurs when ϵ is somewhere between 10^{-6} and 10^{-9} (Figure 2.23). This is substantially smaller than $1/(r + c)$. But, even running FASTPC to get a $1+1/(r+c)$-approximate solution requires more than $\Omega((r+c)^3)$ time. Thus, at least for these instances the proposed hybrid algorithm would be much slower than barrier.

There seem to be two possible scenarios concerning much larger instances:

- The value of ϵ required at crossover in order to substantially reduce the running time of the second stage is less than $1/(r+c)$ even for large instances. In this case the proposed hybrid algorithm will not be faster even for much larger inputs.
- Alternatively, the required value of ϵ remains constant in the range $10^{-6} - 10^{-9}$ even for much larger inputs. In this case, the proposed hybrid algorithm could eventually be faster than the barrier algorithm for sufficiently large inputs.

Both of these scenarios are consistent with our experimental data. In short, the proposed hybrid algorithm would not be faster for inputs up to the size about $r + c = 10^6$ to 10^9. For larger inputs we can not conclude from our data whether the hybrid algorithm could be faster.

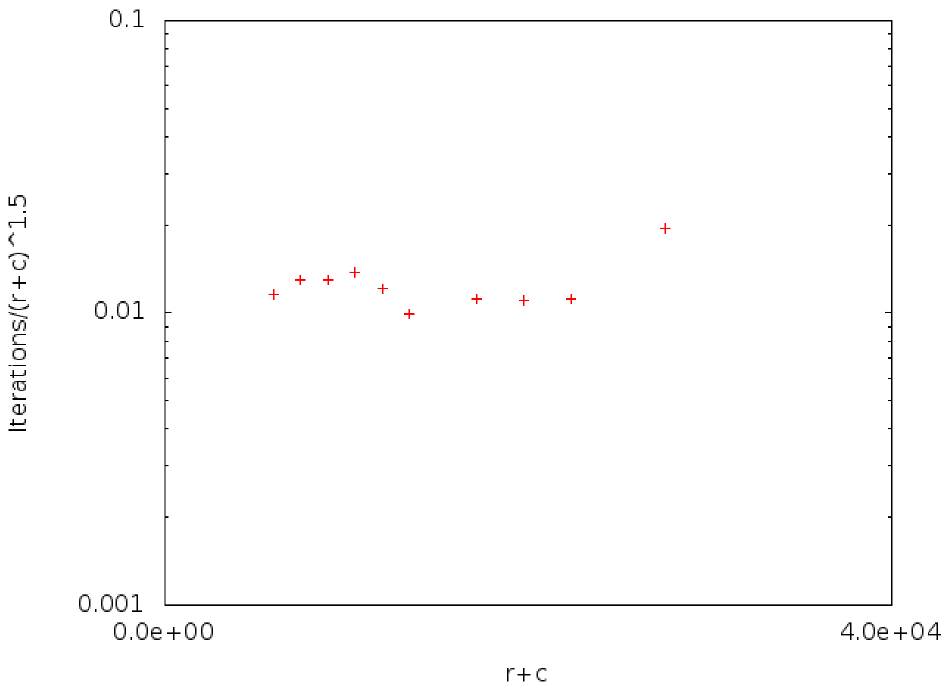
2.4 Conclusions and future directions

The observed lower bounds for the CPLEX and FASTPC algorithms suggest that for reasonably large instances FASTPC will outperform the CPLEX algorithms for random and tomography inputs. We were able to see the crossover point for dense inputs in our experiments and we also predict the crossover points for sparse inputs using the models for FASTPC and barrier algorithms.

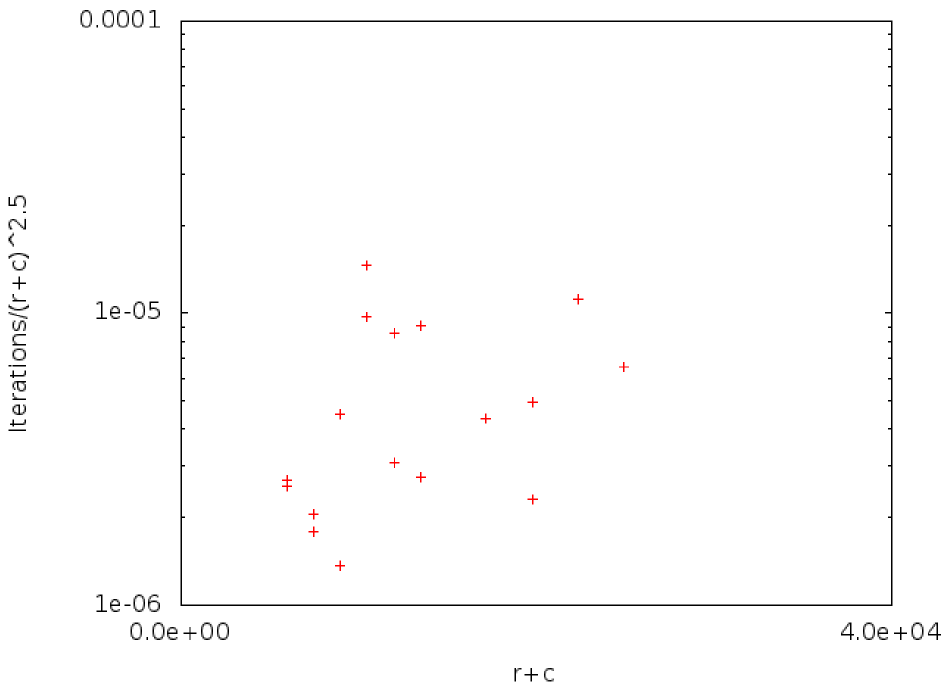
For set packing and DIMACS inputs, we can not conclude from this data that FASTPC will perform better than CPLEX algorithms for large instances. If CPLEX algorithms are slow (asymptotically) on larger instances of these input types, we may see FASTPC catching up with them. We find that all CPLEX algorithms compute the fractional solutions on DIMACS and set packing inputs very quickly, compared to random and tomography inputs. This is true even for the larger instances we created using the generator described in [61]. It may also be possible to tweak this model or use a different model to generate instances where these algorithms are relatively slower in computing fractional solutions.

We don't study the memory usage of these algorithms in this study. It would be an interesting follow-up to this work.

In some recent work, Bienstock and Iyengar [11] and Chudak and Eleutério [18] give algorithms where the dependence on ϵ is proportional to $O(1/\epsilon)$. The running time of the algorithm in [18] is $[c^{1.5}(r+c)/\epsilon + c^2r]$. For a fixed or moderately small ϵ, this algorithm is slower than FASTPC but, for smaller values of ϵ it may be faster. It may be interesting to investigate if this approach used in [11, 18] can be modified to achieve better running times with the same dependence on ϵ. It would also be interesting to study how it compares to the algorithms studied in this work. If it is faster than the interior-point algorithm for very small values of ϵ, the hybrid algorithm where the pre-crossover step uses this algorithm may be faster than the barrier algorithm.

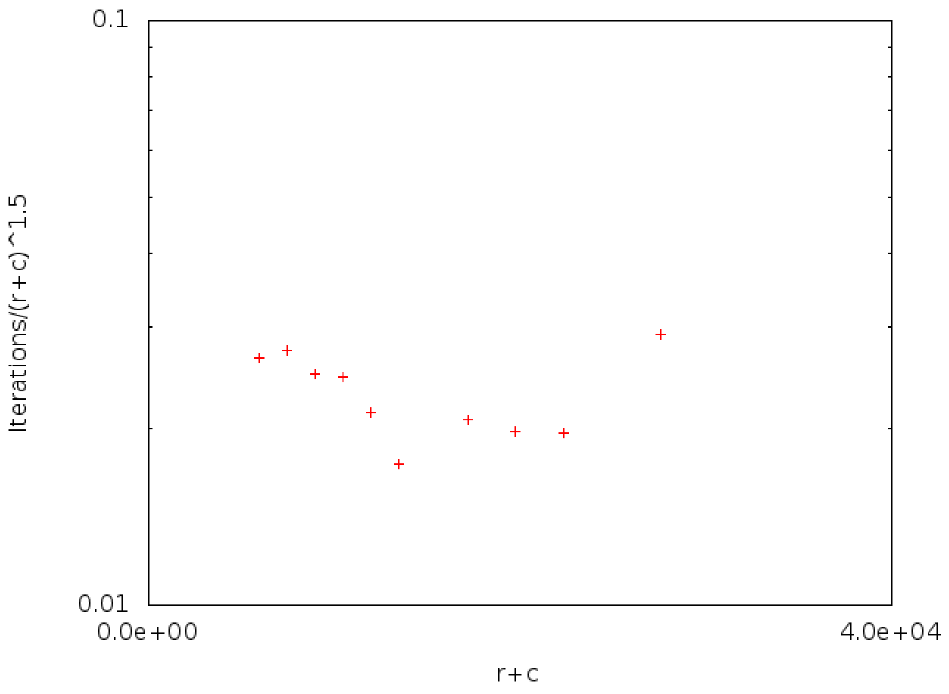


(a) $d < 0.01$

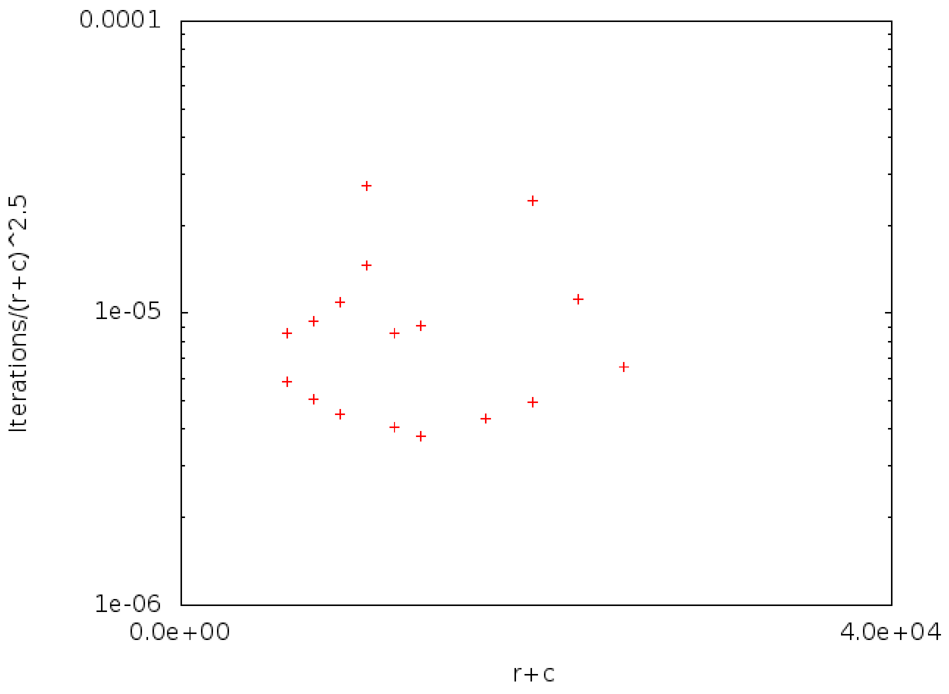


(b) $d \geq 0.01$

Figure 2.8: Iterations for primal-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution

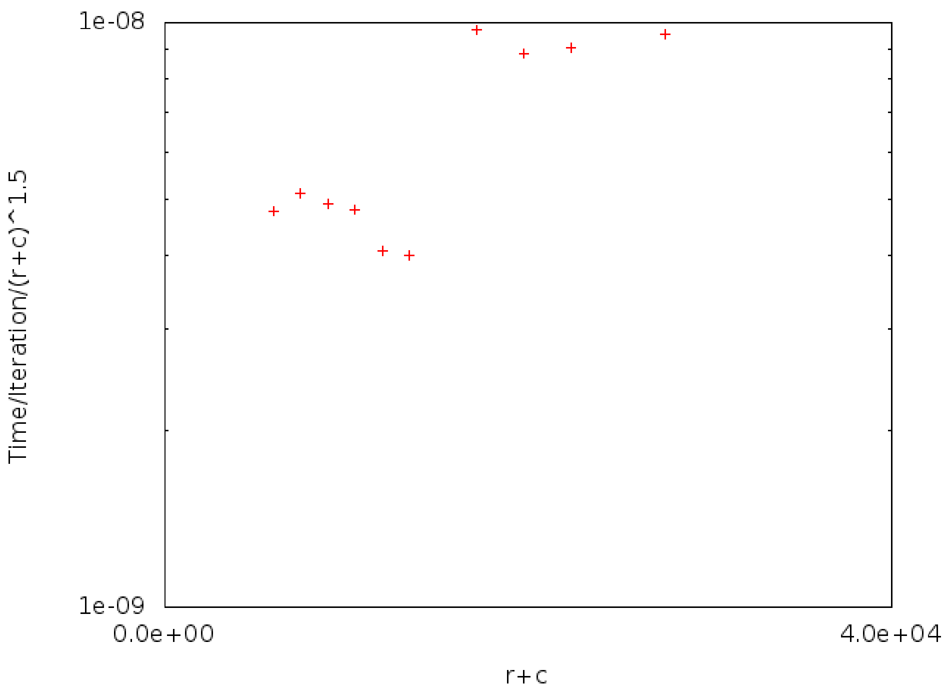


(a) $d < 0.01$

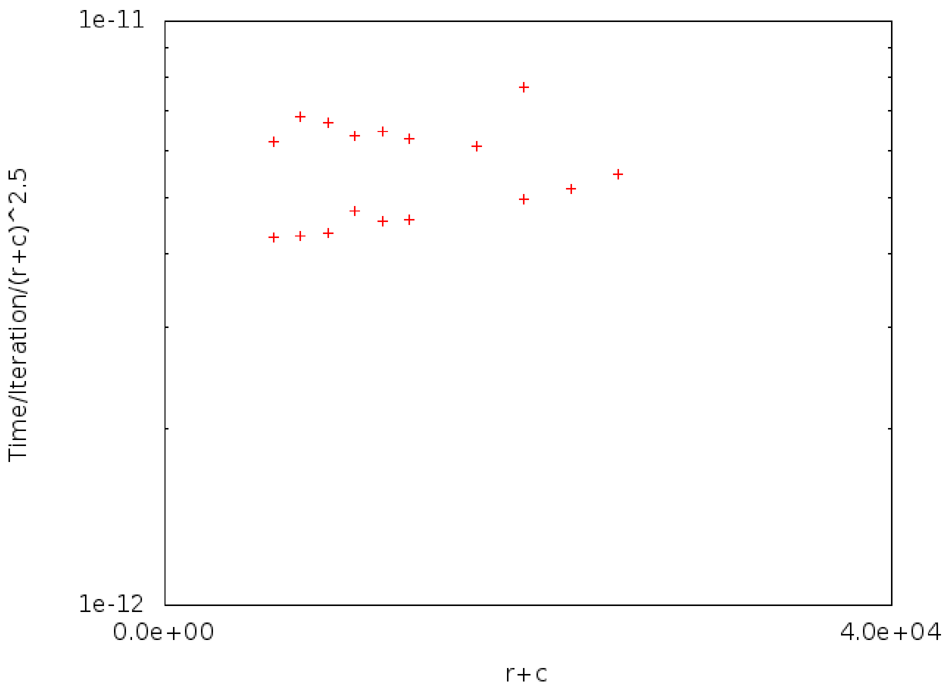


(b) $d \geq 0.01$

Figure 2.9: Iterations for primal-simplex on random and tomography inputs for exact solution

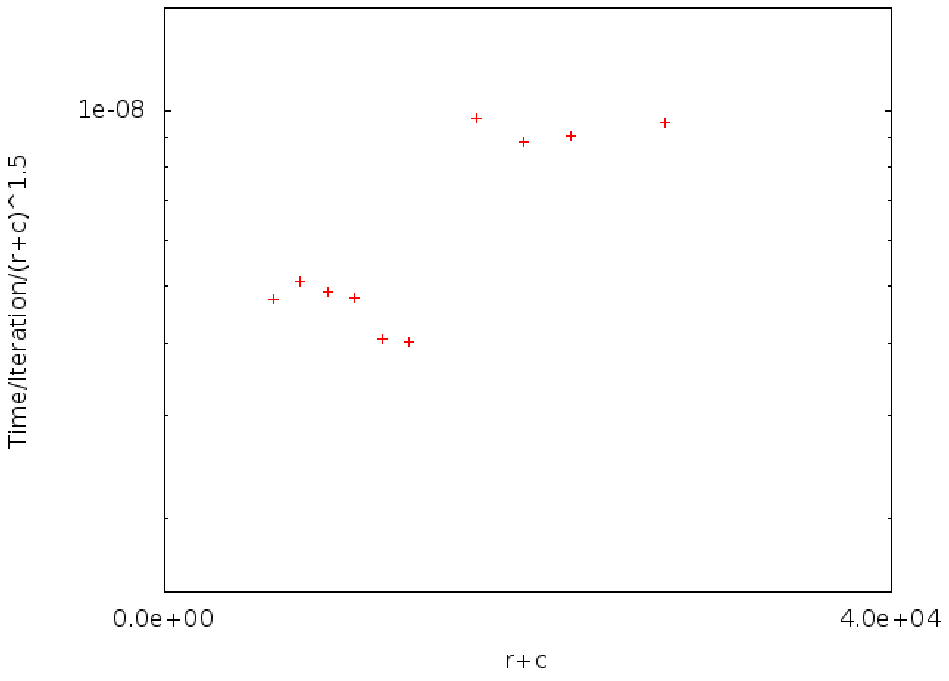


(a) $d < 0.01$

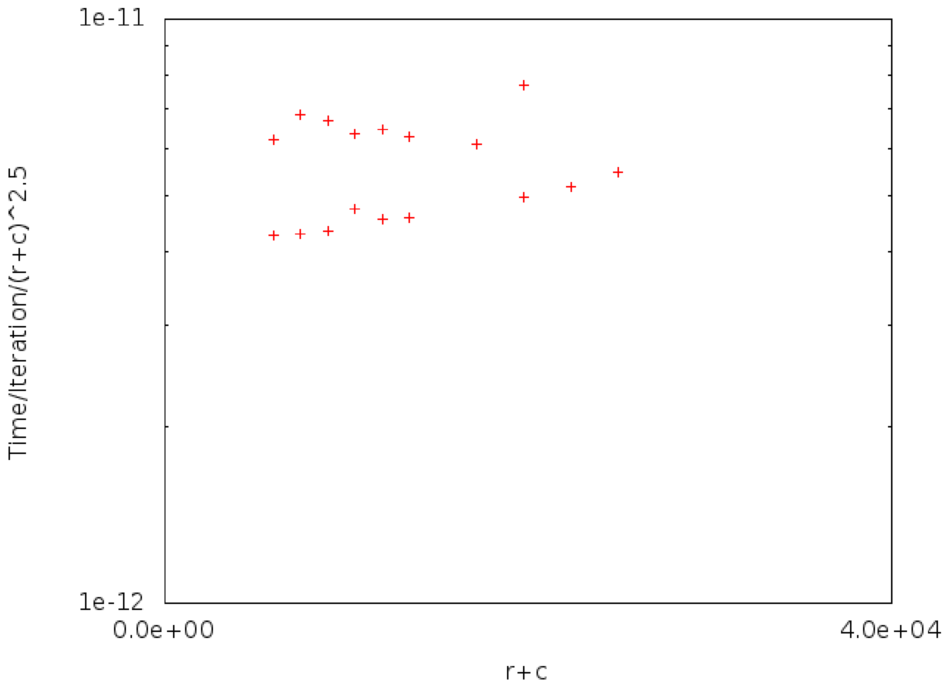


(b) $d \geq 0.01$

Figure 2.10: Time per iteration for primal-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution

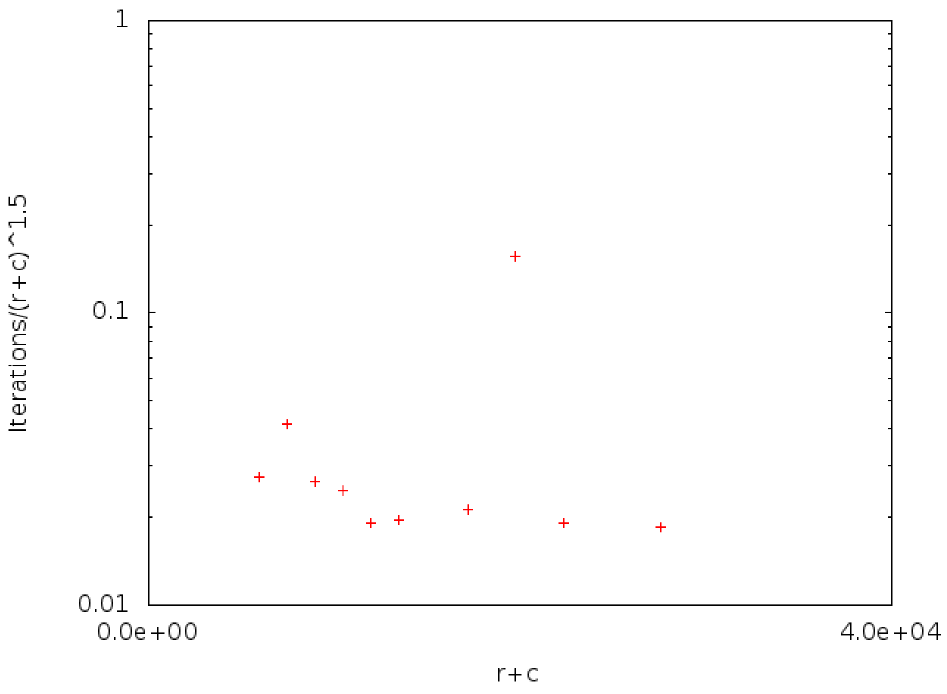


(a) $d < 0.01$

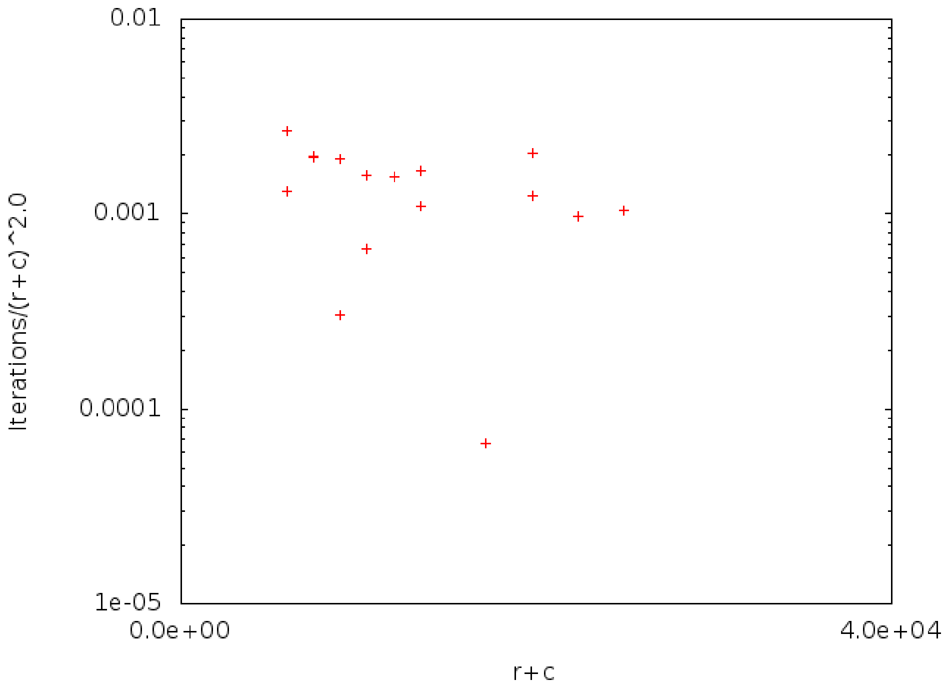


(b) $d \geq 0.01$

Figure 2.11: Time per iteration for primal-simplex on random and tomography inputs to compute exact solution

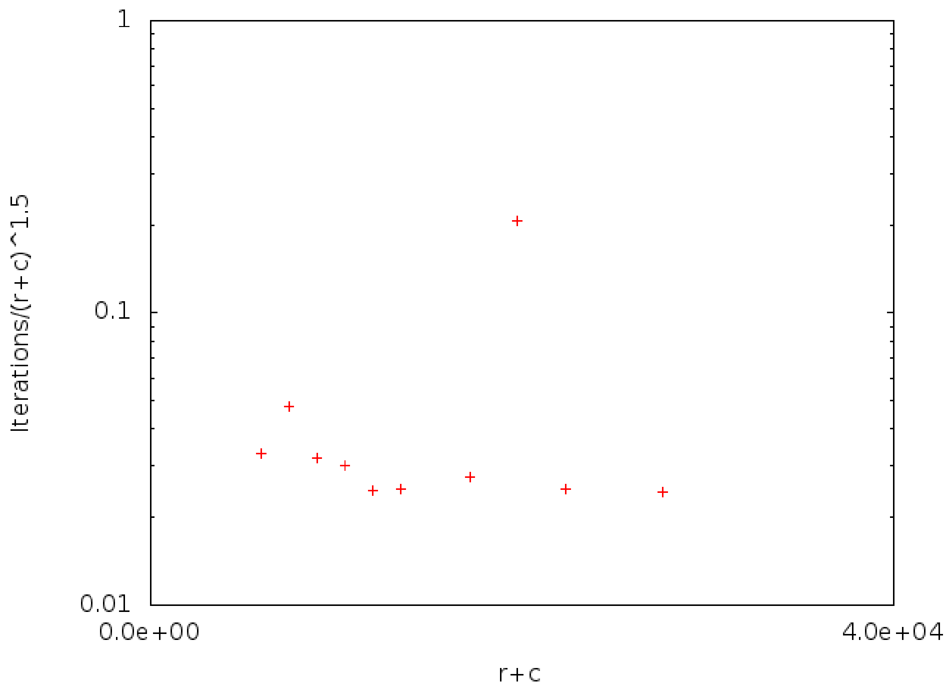


(a) $d < 0.01$

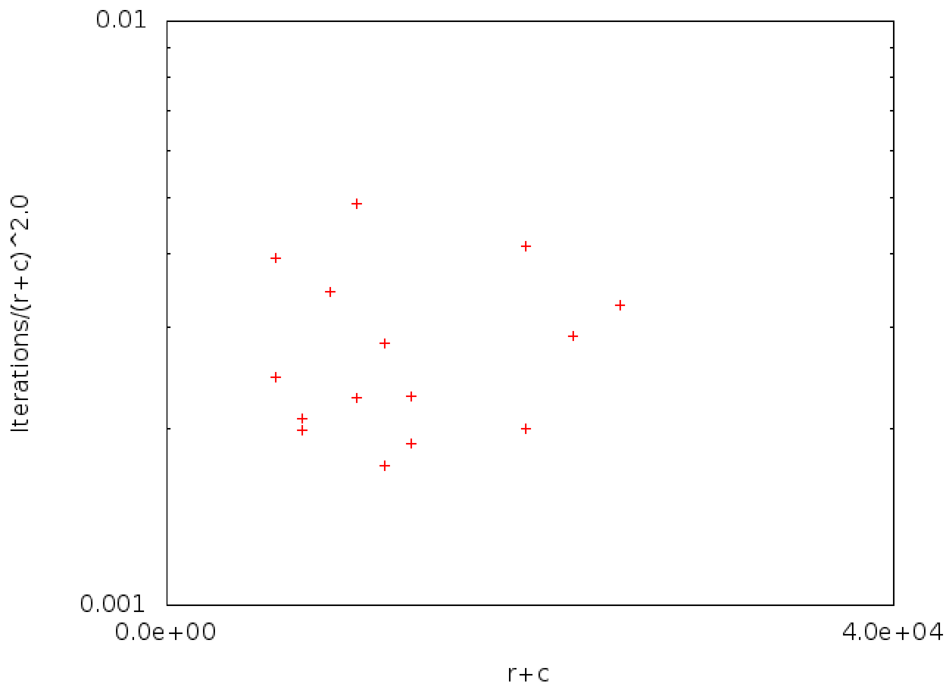


(b) $d \geq 0.01$

Figure 2.12: Iterations for dual-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution

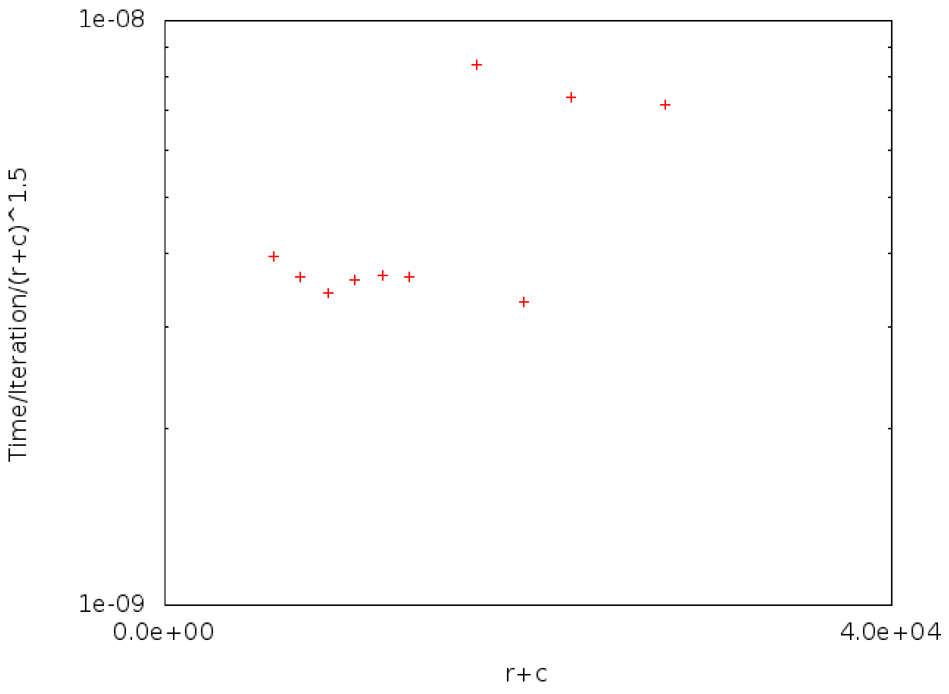


(a) $d < 0.01$

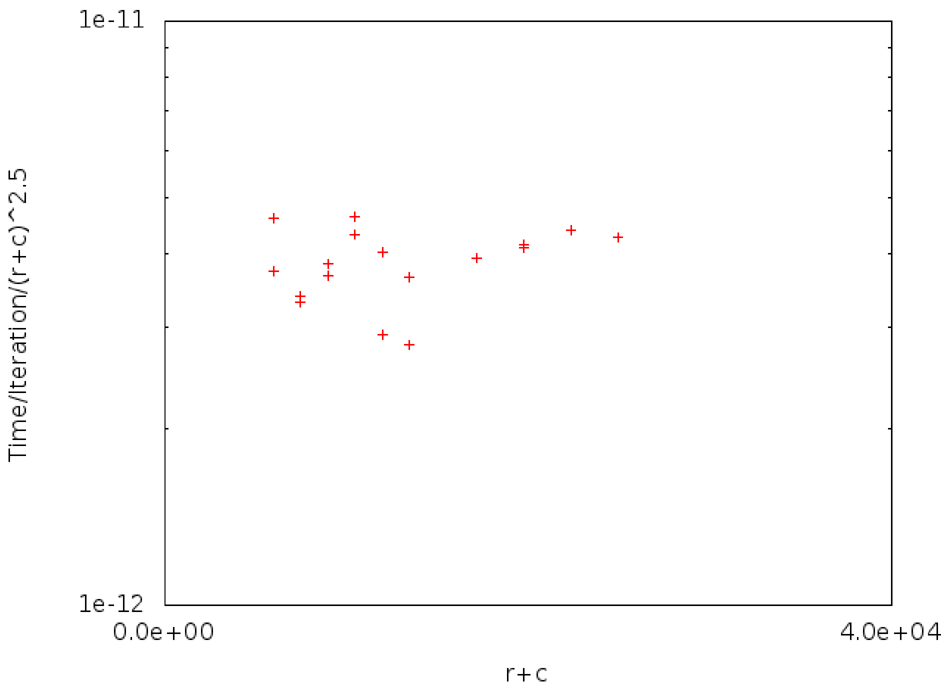


(b) $d \geq 0.01$

Figure 2.13: Iterations for dual-simplex on random and tomography inputs for exact solution

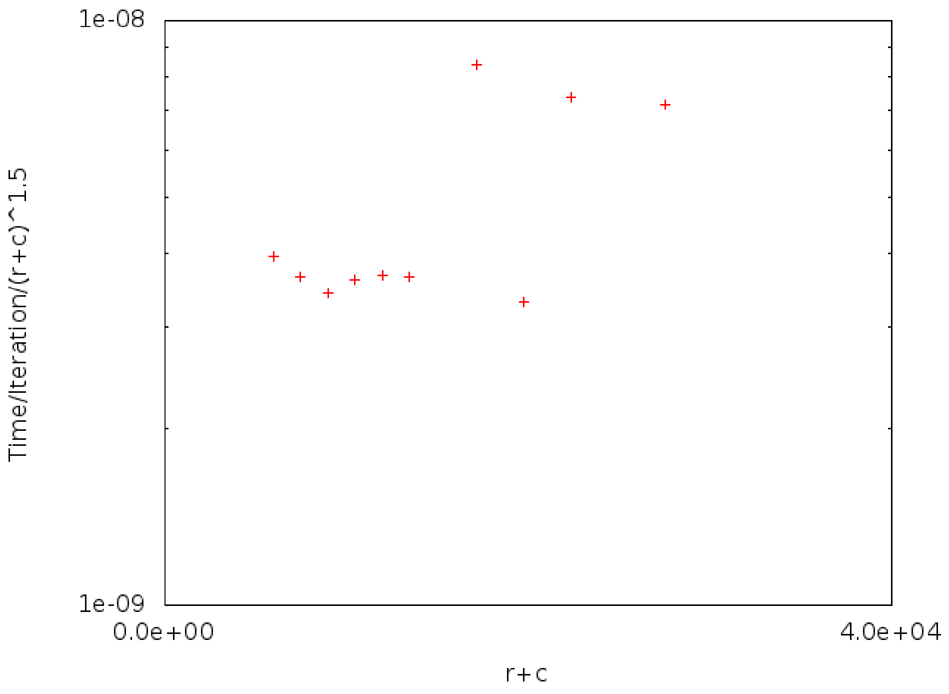


(a) $d < 0.01$

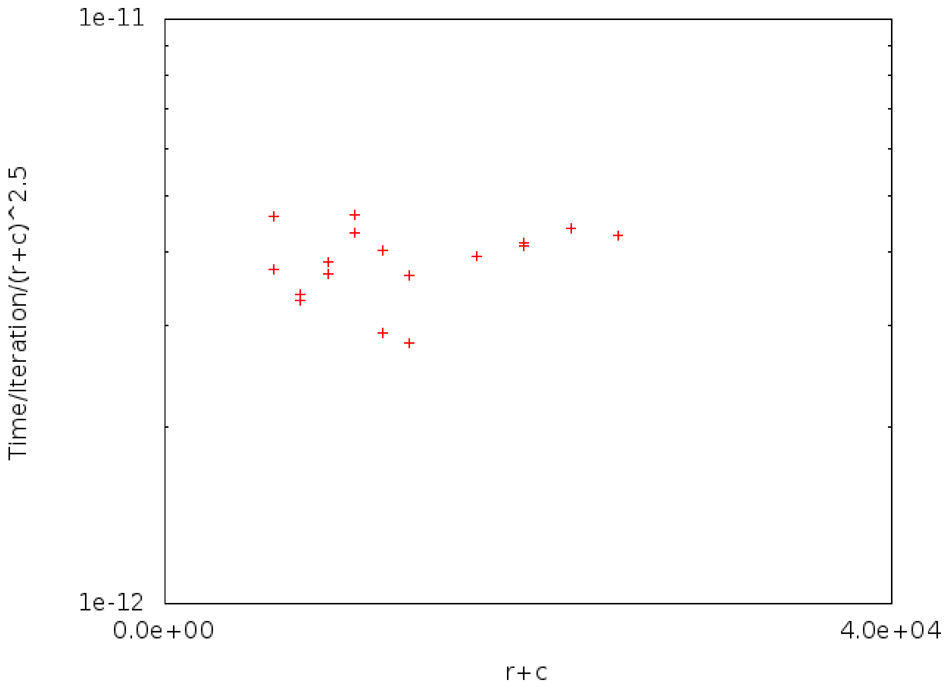


(b) $d \geq 0.01$

Figure 2.14: Time per iteration for dual-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution

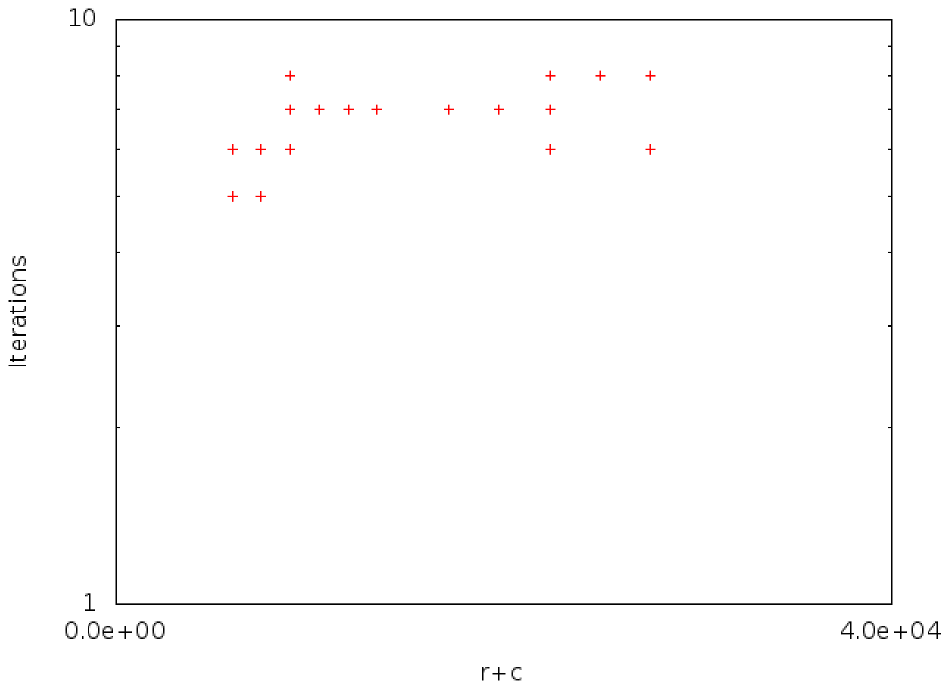


(a) $d < 0.01$

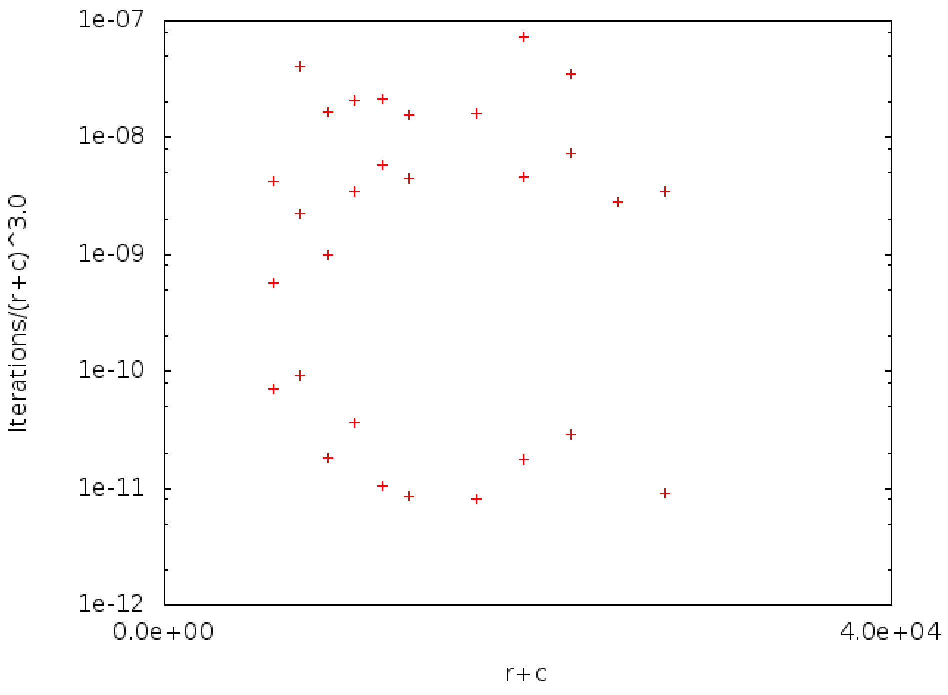


(b) $d \geq 0.01$

Figure 2.15: Time per iteration for dual-simplex on random and tomography inputs to compute exact solution

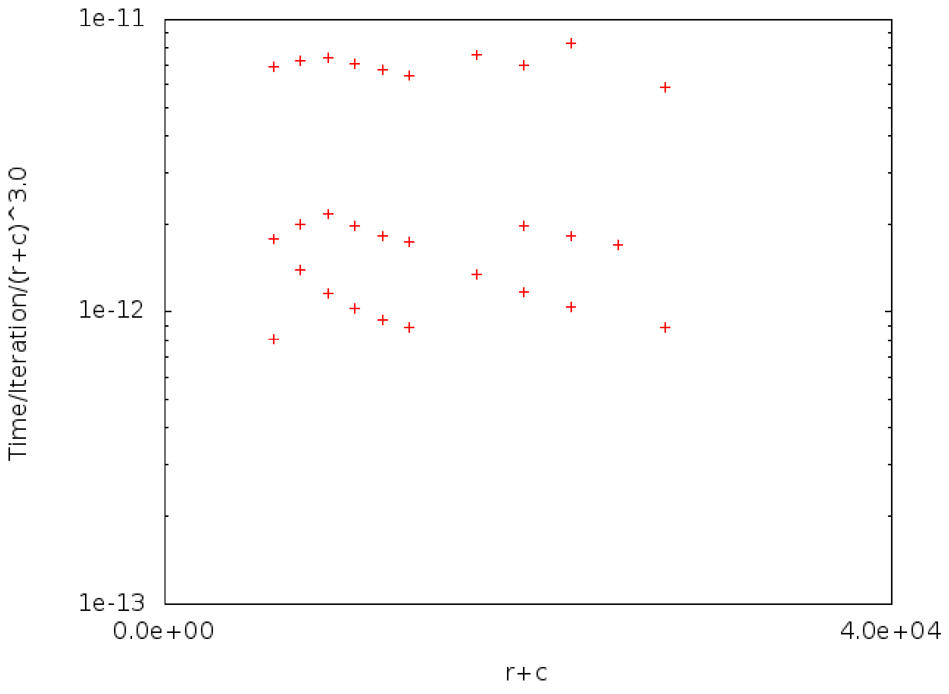


(a) $(1 \pm \epsilon)$-solution

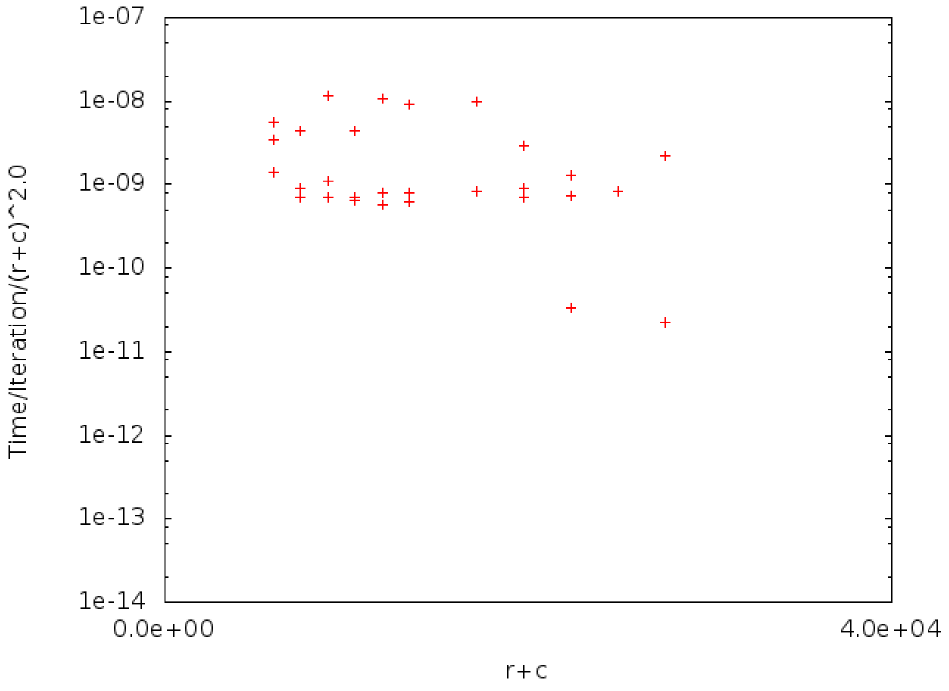


(b) Exact solution

Figure 2.16: Iterations for barrier-simplex on set packing and DIMACS inputs

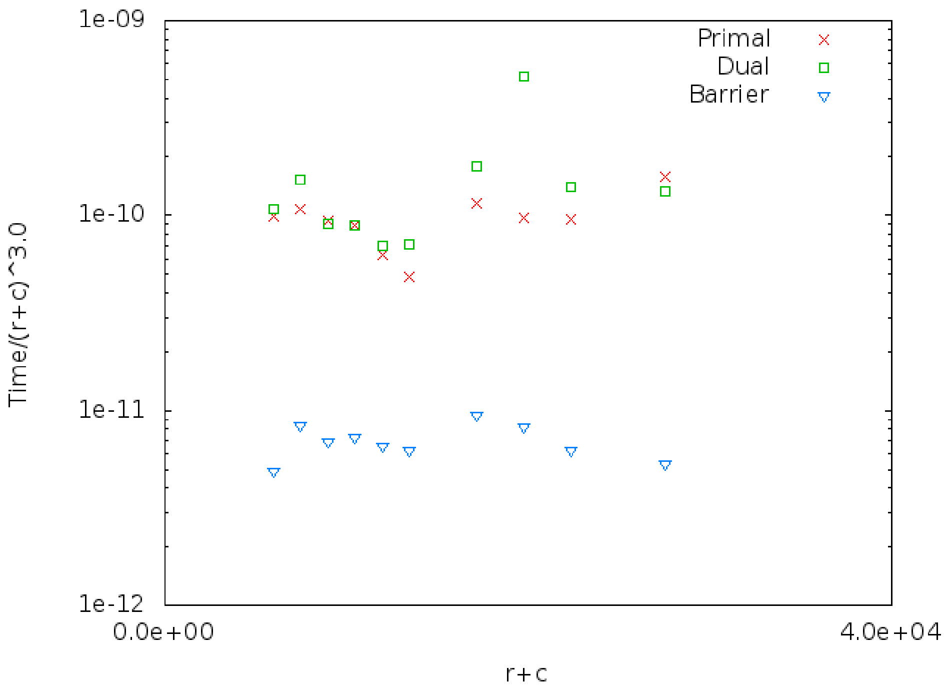


(a) $(1 \pm \epsilon)$-solution

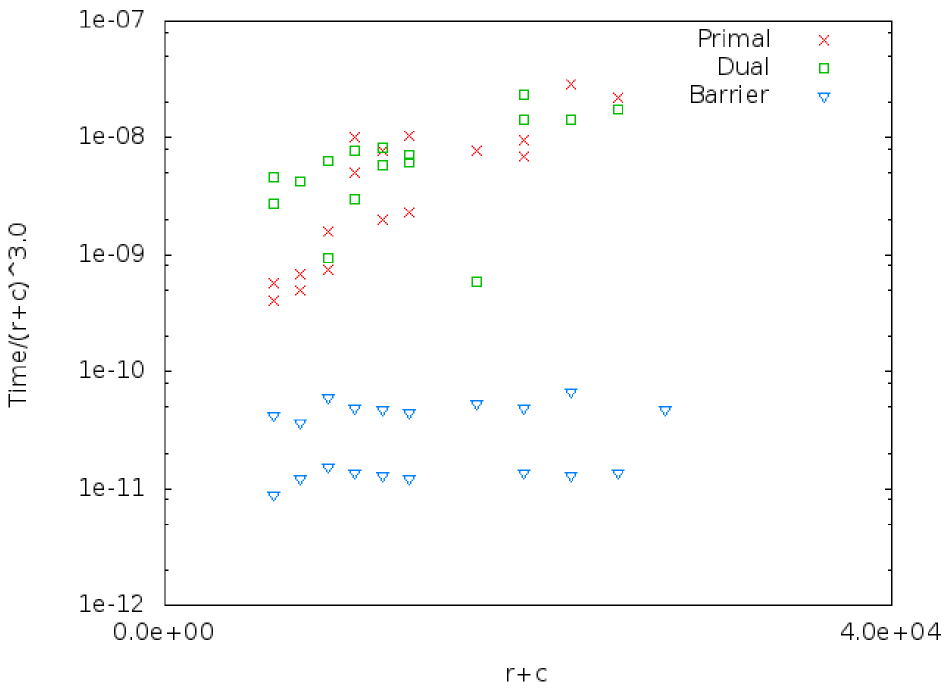


(b) Exact solution

Figure 2.17: Time per iteration for barrier-simplex on set packing and DIMACS inputs

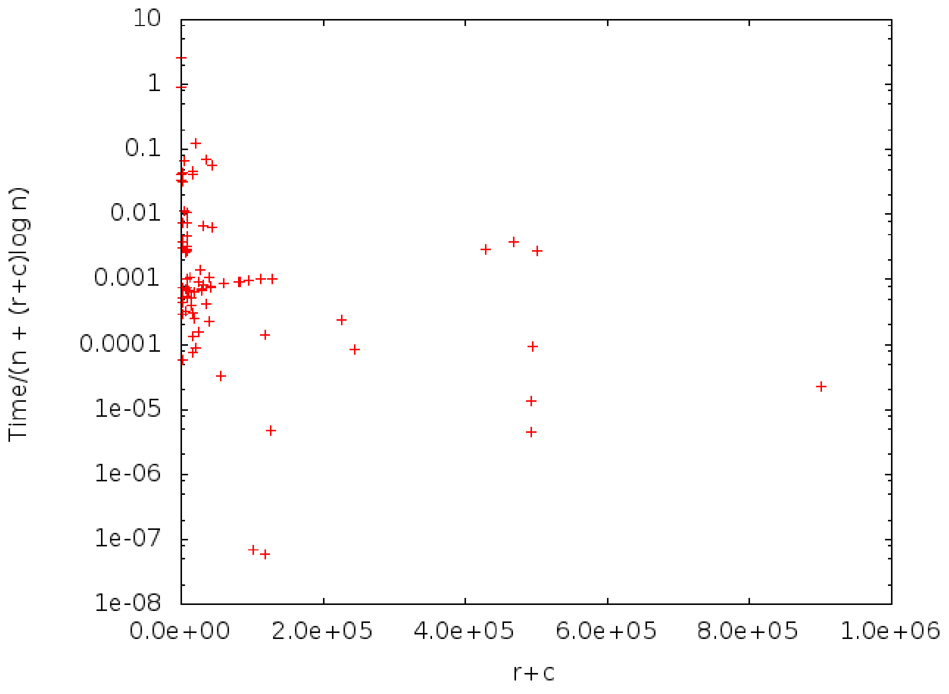


(a) $d < 0.01$

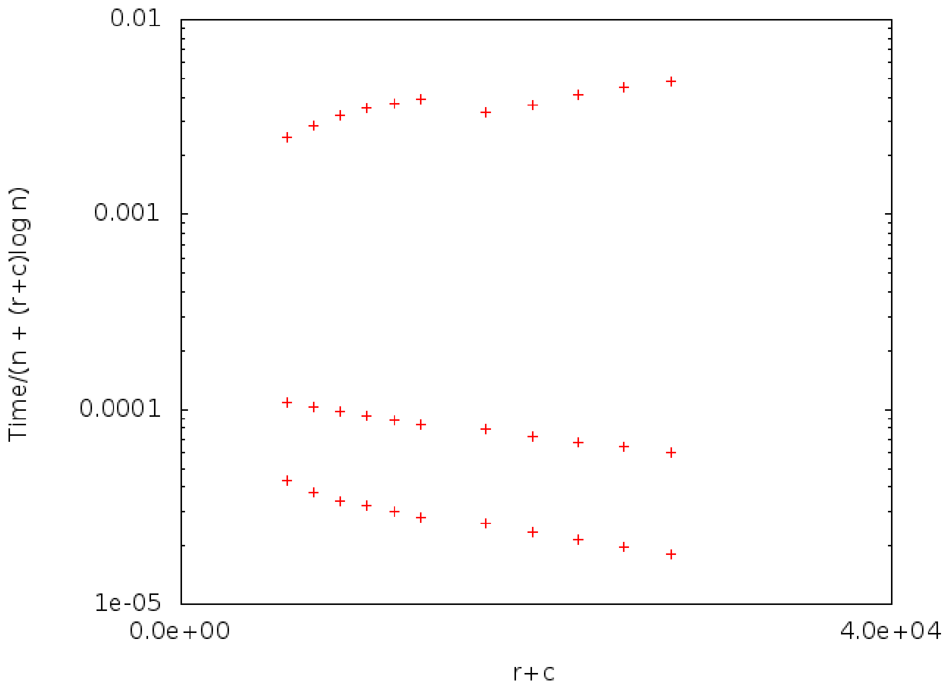


(b) $d \geq 0.01$

Figure 2.18: Comparison of CPLEX algorithms to compute $(1 \pm \epsilon)$-solution for $\epsilon = 0.01$ on random and tomography inputs

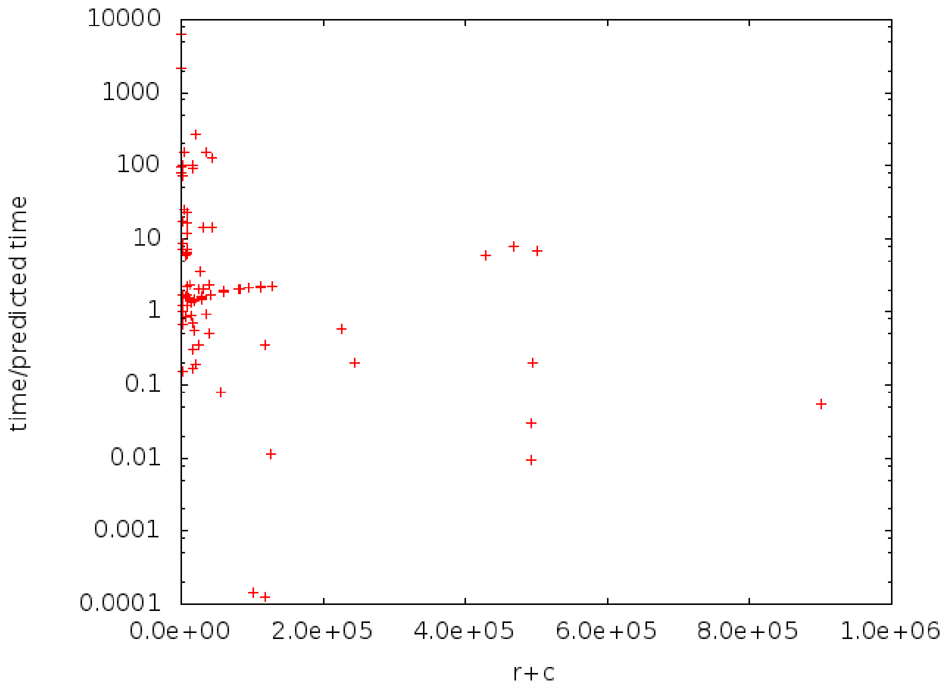


(a) Set packing and DIMACS inputs

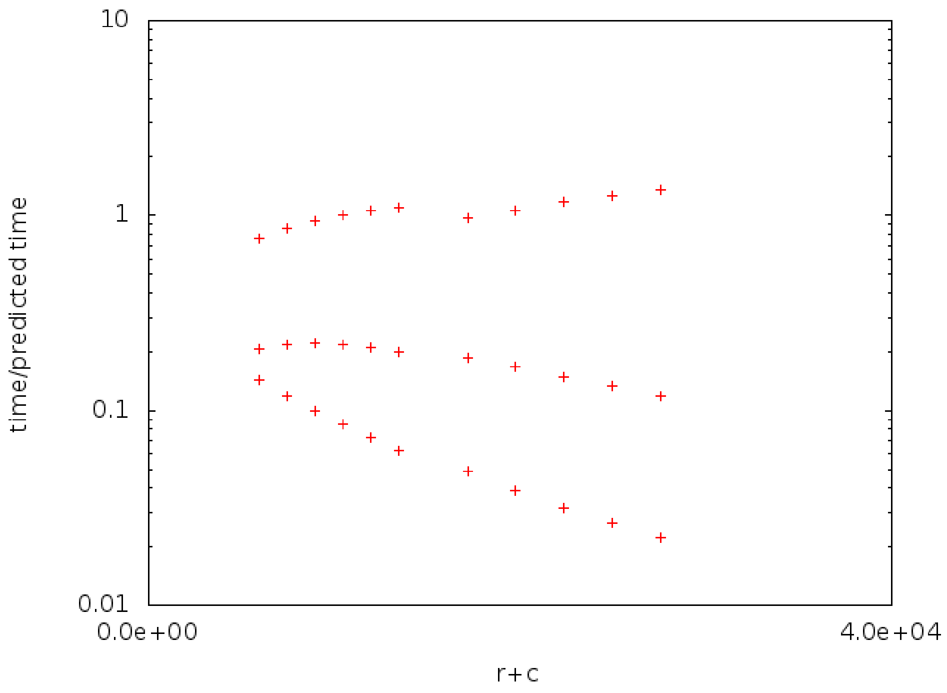


(b) Random and tomography inputs

Figure 2.19: Running time for FASTPC for $\epsilon = 0.01$



(a) Set packing and DIMACS inputs



(b) Random and tomography inputs

Figure 2.20: Time / predicted time for FastPC for $\epsilon = 0.01$

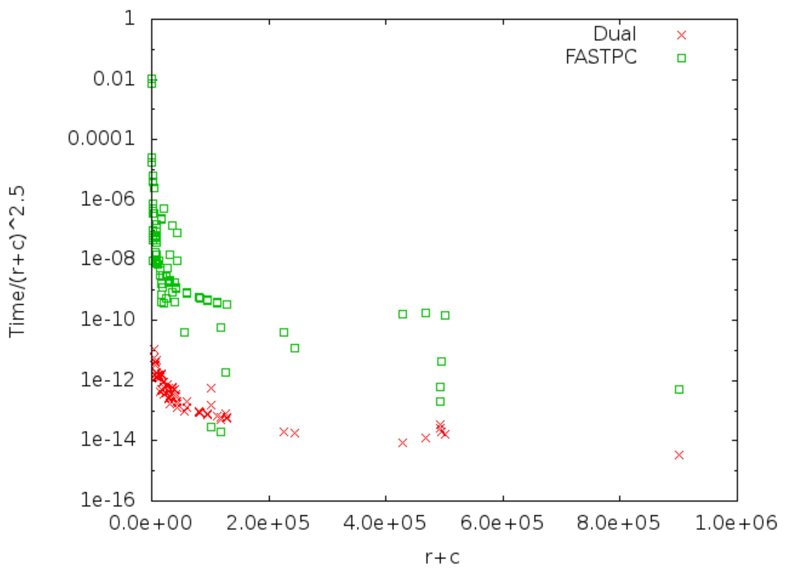
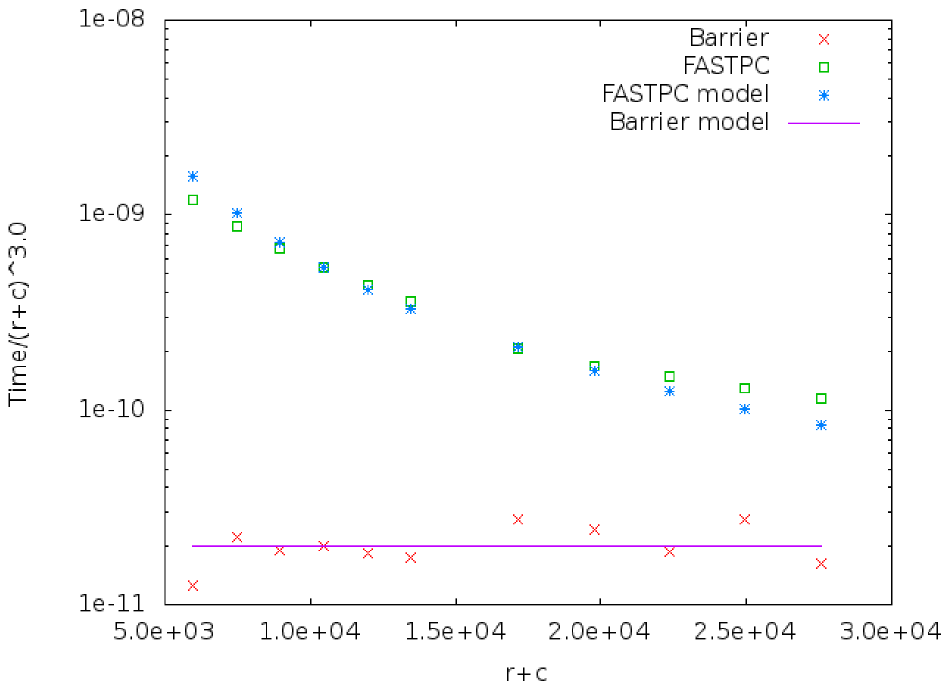
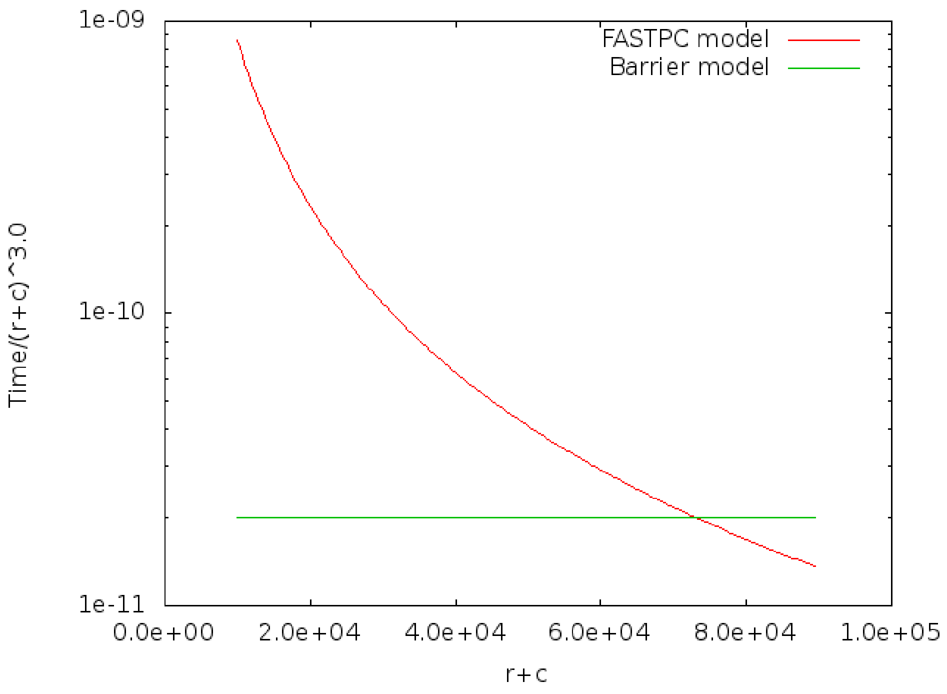


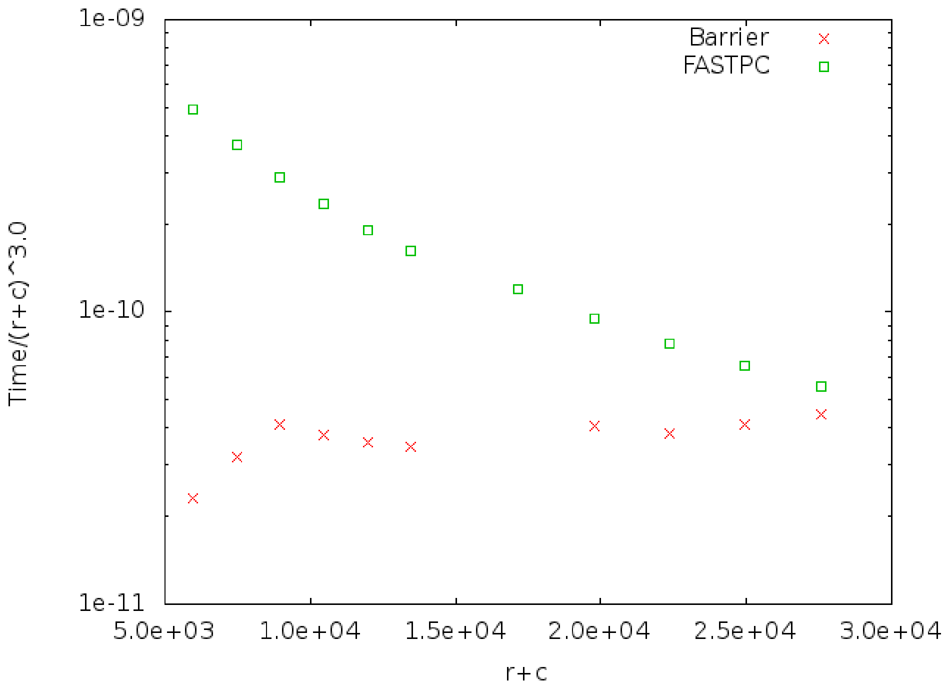
Figure 2.21: FASTPC vs dual-simplex on set packing and DIMACS inputs



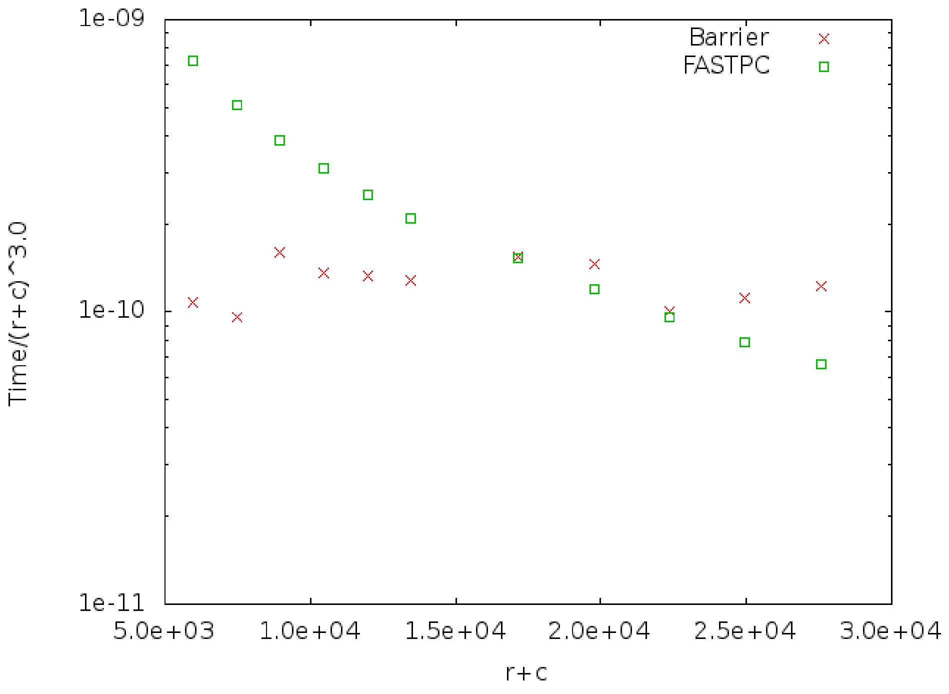
(a) Actual data and models for density < 0.01



(b) Prediction using models for density < 0.01



(c) $0.01 \leq$ Density < 0.15



(d) Density ≥ 0.15

Figure 2.22: FASTPC vs barrier on random and tomography inputs

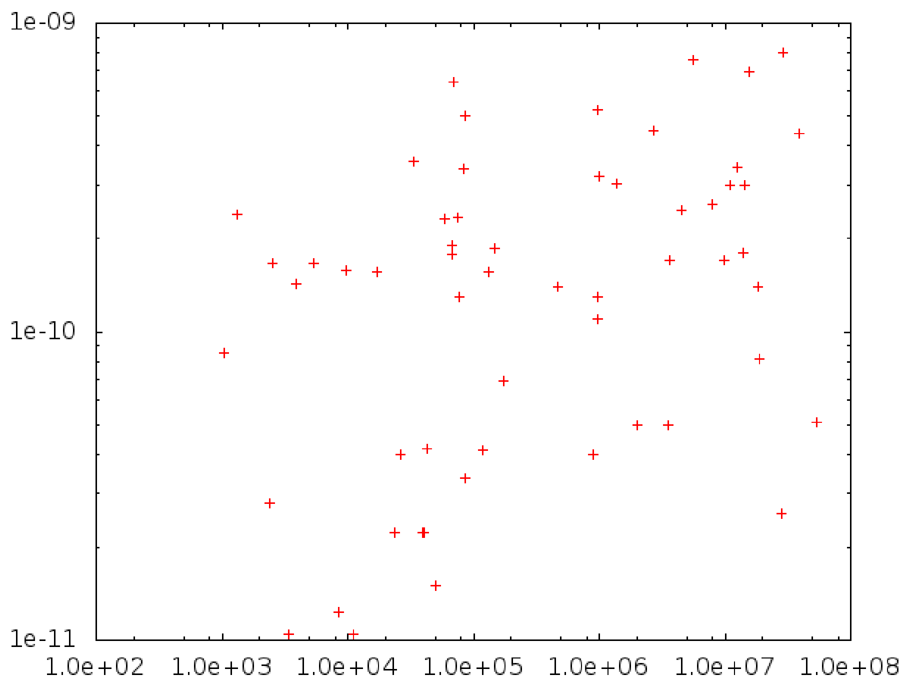


Figure 2.23: Value of ϵ at crossover (log-log scale)

Chapter 3

File caching with rental cost and zapping

3.1 Introduction

3.1.1 Background

The *file caching* (or *generalized caching*) problem is defined as follows. Given a cache of size k (a positive integer), the goal is to minimize the total retrieval cost for the given sequence of requests to files. A file f has size $size(f)$ (a positive integer) and retrieval cost $cost(f)$ (a non-negative number) for bringing the file into the cache. A *miss* or *fault* occurs when the requested file is not in the cache and the file has to be brought into the cache by paying the retrieval cost. When a file is retrieved into the cache, some other files may have to be removed (*evicted*) from the cache so that the total size of the files in the cache does not exceed k. Weighted caching (or weighted paging) is the special case when each file has size 1. Paging is the special case when each file has size 1 and the retrieval cost for each file is 1.

An algorithm is *online* if its response for each request is independent of all future requests. Let ALG(σ) be the cost of algorithm ALG on request sequence σ, and let OPT(σ) be the corresponding optimal offline cost. ALG is α-competitive if, for every request sequence σ, $\mathrm{ALG}(\sigma) \leq \alpha \cdot \mathrm{OPT}(\sigma) + c$, where c is a constant independent of the request sequence.

In this work, we study the following variants of the file caching problem in the online setting using the competitive-analysis framework [41].

Caching with Rental Cost (or Rental Caching): There is a rental cost λ (a positive number) for each slot used by a file in the cache at each time step. The goal is to minimize the sum of the retrieval costs and the rental costs.

The rental cost for file f that is in the cache for d time is $size(f) \cdot d$. Chrobak [15] proposes the rental caching problem and also presents some preliminary results. The rental caching problem is motivated by the idea of energy efficient caching. Caching systems can save power by turning off the memory blocks that are not being used to store any files. Rental caching models this by charging a rental cost for keeping each file in the cache. See [49] for specific applications.

In Section 3.3.2 we show that the variant of rental caching where the cache has infinite size is closely related to the ski-rental problem. The *ski-rental* problem is the following. A pair of skis can be rented by paying \$$\lambda$ per day, or can be bought for the remainder of the ski season by paying \$$B$. It is not known in advance when the season is going to end and the goal is to minimize the total money spent for the entire season [38].

Weighted rental caching (or, *weighted rental paging*) is a special case of the rental caching where each file has size 1 and *rental paging* is a special case where each file has size 1 and the retrieval cost for each file is 1. Similarly, rental caching for the cases of bit and fault models have $\mathit{cost}(f) = 1$ and $\mathit{cost}(f) = \mathit{size}(f)$, respectively, for each file f.

Caching with Zapping: Any file can be *zapped* by paying the *zapping* cost N (a positive number) at any time step. Once a file is zapped, it doesn't take up any space in the cache and any future requests to it do not incur any retrieval or rental cost. The goal is to minimize the sum of the retrieval costs and the zapping costs.

We study file caching, as well as the special cases of weighted caching, paging, bit model, and fault model, where we have at least one of rental cost and zapping.

Weighted caching with zapping (or, *weighted paging with zapping*) is a special case of the rental caching where each file has size 1. *paging with zapping* is a special case where each file has size 1 and the retrieval cost for each file is 1. Caching with zapping for the cases of bit and fault models have $\mathit{cost}(f) = 1$ and $\mathit{cost}(f) = \mathit{size}(f)$, respectively, for each file f.

These variants generalize the file caching problem. File caching is a special case of rental caching where the rental cost is 0. Similarly, caching is a special case of caching with zapping where the cost of zapping is arbitrarily large. We also study the variant which combines these two variants: ***rental caching with zapping***. In all these variants, we allow time steps with no requests.

3.1.2 Previous work

In 1985 Sleator and Tarjan [55] introduce the competitive-analysis framework. In [55] they show that well-known paging rules like LRU, FIFO, and FWF are k-competitive and that k is the best ratio any deterministic online algorithm can achieve for the paging problem.

Fiat et al. [25] initiate the competitive analysis of paging algorithms in the randomized setting. They show a lower bound of H_k, where H_k is the k_{th} harmonic number, for any randomized algorithm. They give a $2H_k$-competitive randomized marking algorithm. Achlioptas et al. [4] show that the tight competitive ratio of the randomized marking algorithm is $2H_k - 1$. McGeoch and Sleator [52] and Achlioptas et al. [4] give optimal H_k-competitive randomized algorithms for paging.

For weighted caching, Chrobak et al. [17] give a tight k-competitive deterministic algorithm. For the randomized case, Bansal et al. [7] give a tight $O(\log k)$-competitive primal-dual algorithm.

For file caching, Irani [37] shows that the offline problem is NP-hard. For the online case, Irani [37] gives results for the *bit model* ($cost(f) = size(f)$ for each file f) and *fault model* ($cost(f) = 1$ for each file f). She shows that LRU is $(k+1)$-competitive for both models. Cao and Irani [14] extend the result to file caching. Young [65] independently gives the LANDLORD algorithm and shows that it is $(\frac{k}{k-h+1})$-competitive for the file caching problem. Irani [37] gives an $O(\log^2 k)$-competitive randomized algorithm for bit and fault models. Bansal et al. [7] give an $O(\log k)$-competitive randomized algorithm for both the models, and an $O(\log^2 k)$-competitive randomized algorithm for the general case.

3.1.2.1 Methodology

Young [62] uses a primal-dual analysis to give a $\left(\frac{k}{k-h+1}\right)$-competitive deterministic online algorithm for weighted caching. Bansal et al. [7, 8], Buchbinder and Naor [13] use a primal-dual approach to give randomized algorithms for paging, weighted caching, and file caching. In a recent work, Adamaszek et al. [5] build on their online primal-dual approach to give an $O(\log k)$-competitive for file caching. In another recent work Epstein et al. [23] show that this online primal-dual approach can be extended to *Caching with Rejection.* Caching with rejection is a variant of file caching where a request to a file that is not in the cache can be declined by paying a rejection penalty. In this variant, each request is specified as a pair (f, r), where f is the file requested and r is the rejection penalty. Note that caching with rejection is different from caching with zapping. In caching with zapping, a file can be zapped at any time, while in caching with rejection a file can be rejected only when it is requested. Moreover, a rejected file can incur a retrieval cost or a rejection penalty again in the future, while the zapped file does not incur any cost after it is zapped.

Koufogiannakis and Young [46] give a deterministic greedy Δ-approximation algorithm for any covering problem with a submodular and non-decreasing objective function, and with arbitrary constraints that are closed upwards, such that each constraint has at most Δ variables. They show that their algorithm is Δ-competitive for the online version of the problem where the constraints are revealed one at a time. Many online caching and paging problems reduce to this online covering problem, and consequently, their algorithm generalizes many classical deterministic algorithms for these problems. These include LRU and FWF for paging, BALANCE and GREEDY DUAL for weighted caching, LANDLORD (a.k.a. GREEDY DUAL SIZE) for file caching, and algorithms for CONNECTION CACHING [46]. We study this approach and extend it to give deterministic online algorithms for the variants of online file caching studied in this work.

3.1.3 Our contributions

We study rental caching, caching with zapping, and rental caching with zapping. We present deterministic and randomized lower and upper bounds for these new variants of paging, weighted caching, and file caching in the online setting. We use the approach in [46] to give deterministic algorithms for these online problems. While this approach is general, it doesn't necessarily give optimal online algorithms. The direct application of this approach yields sub-optimal algorithms in some of the cases we study in this paper. We describe these scenarios and also the appropriate modifications to the algorithm to achieve better competitive ratios. Table 3.1 summarizes our results.

Problem			**Lower Bound**	**Upper Bound**	**Gap**
Rental paging, Rental caching: bit model, Rental caching: fault model	Det	$\lambda \geq \frac{1}{k}$	$2 - \lambda$	2	$O(1)$
		$\frac{1}{k^2} \leq \lambda < \frac{1}{k}$	$\frac{k+k\lambda}{1+k^2\lambda}$	$1 + \frac{1}{k\lambda}$	
		$\lambda < \frac{1}{k^2}$		k	
	Rand	$\lambda \geq \frac{1}{k}$		$\frac{e}{e-1}$	$O(1)$
		$\lambda < \frac{1}{k}$	$\frac{H_k+k^2H_k\lambda}{1+k^2H_k\lambda}$	$H_k + \frac{e}{e-1}$	$O(H_k)$
Weighted rental caching, Rental caching:	Det	$\lambda \geq \frac{1}{k}$	$2 - \lambda$	k	$O(k)$
		$\lambda < \frac{1}{k}$	$\frac{k+k\lambda}{1+k^2\lambda}$		$O(1)$
	Rand	$\lambda \geq \frac{1}{k}$		$\frac{e}{e-1}$	$O(1)$
		$\lambda < \frac{1}{k}$	$\frac{H_k+k^2H_k\lambda}{1+k^2H_k\lambda}$	$O(\log k)$	$O(\log k)$
Paging with zapping, Weighted caching with zapping, Caching with zapping	Det		$\min(N-1, \frac{2Nk+N-(k+1)}{N+2k})$	$\min(N, 2k+1)$	$O(1)$
Rental paging with zapping, Rental caching with zapping: bit model, Rental caching with zapping: fault model	Det	$\lambda \geq \frac{1}{k}$	$2 - \lambda$	3	$O(1)$
		$\frac{1}{k^2} \leq \lambda < \frac{1}{k}$	$\frac{k+k\lambda}{1+k^2\lambda}$	$1 + \frac{2}{k\lambda}$	
		$\lambda < \frac{1}{k^2}$		$2k+1$	
Weighted rental caching with zapping, Rental caching with zapping	Det	$\lambda \geq \frac{1}{k}$	$2 - \lambda$	$2k+1$	$O(k)$
		$\lambda < \frac{1}{k}$	$\frac{k+k\lambda}{1+k^2\lambda}$		$O(1)$

Table 3.1: Competitive ratios for the variants with rental cost and zapping. In column 2, "Det" represents Deterministic and "Rand" represents Randomized.

For rental paging and for rental caching for bit and fault models, the deterministic upper and lower bounds in this paper are tight within constant factors. For the randomized case, the lower and upper bounds are tight within constant factors when λ is $O(\frac{1}{k^2 H_k})$ and when $\lambda \geq \frac{1}{k}$. For weighted rental paging and for rental caching, the upper and lower bounds are tight within constant factors when $\lambda < \frac{1}{k}$ for the deterministic case, and when λ is $O(\frac{1}{k^2 H_k})$ or when $\lambda \geq \frac{1}{k}$ for the randomized case. The bounds for the variants with rental cost are within constant factors of the bounds for the variants without rental cost when $\lambda \leq \frac{1}{k^2}$ for the deterministic case and when λ is $O(\frac{1}{k^2 H_k})$ for the randomized case. For higher values of $\lambda \geq \frac{1}{k}$ we show constant lower bounds and matching upper bounds.

For rental paging with zapping and for rental caching with zapping for bit and fault models, the deterministic upper and lower bounds in this paper are tight within constant factors. Weighted caching with zapping and caching with zapping, the deterministic lower and upper bounds in this paper are tight within constant factors when $\lambda \leq \frac{1}{k}$.

3.1.4 Other work on rental paging

Lopez-Ortiz and Salinger [49], in an independent work, study the rental paging problem. They give a deterministic polynomial time algorithm for the offline problem by reducing it to interval weighted interval scheduling. They show that any *conservative* or *marking* algorithm is k-competitive and that the bound is tight. An algorithm is *conservative* if it incurs at most k faults on any consecutive subsequence of requests that contains at most k distinct pages. A *marking* algorithm marks each page when it is requested, and when it is required to evict a page, it evicts an unmarked page. If there are no unmarked pages, it first unmarks all the pages.

For any online algorithm A for paging, define the algorithm A_d for rental paging as follows. A_d behaves like A with the modification that any page in the cache that has not been requested for d steps is evicted. They define a class of online algorithms M_d, where M is any conservative or marking algorithm. They show an upper bound of 2 on the competitive ratio of $M_{\frac{1}{\lambda}}$ when $\lambda > \frac{1}{k}$, which matches the presented in upper bound in this work. They show an upper bound of $\max(k, \frac{(k+1)}{1+\lambda(k-1)})$ on the competitive ratio of $M_{\frac{1}{\lambda}}$ when $\lambda \leq \frac{1}{k}$. Their bound is at least k when $\lambda < \frac{1}{k}$ and is weaker than the upper bound we present in this work when $\frac{1}{k^2} < \lambda < \frac{1}{k}$. In particular, when λ is $\Omega(\frac{1}{k})$, there is a gap of $\Omega(k)$ between their bound and the bounds shown in this work.

Their deterministic lower bound on the competitive ratio for rental paging matches the lower bound in this work.

They also present experimental results for the performance of various LRU, LRU$_{\frac{1}{\lambda}}$, FWF, FWF$_{\frac{1}{\lambda}}$, FIFO, FIFO$_{\frac{1}{\lambda}}$, and the optimal offline algorithm.

They present results only for rental paging and not for weighted rental paging or rental caching. They do not study the rental paging problem in the randomized setting.

3.2 Online covering approach

In this section, we give an example of the online covering approach from [46]. We use this approach, with modifications in some cases, to give deterministic algorithms for the variants of paging and caching problems in this work.

Koufogiannakis and Young [46] give a Δ-approximation algorithm for Submodular Cost Covering.

Definition 1. ***Submodular Cost Covering****: An instance is a triple $(c, \mathcal{C}, U)$, where*

- *The cost function $c : \bar{\mathbb{R}}^n_{\geq 0} \to \bar{\mathbb{R}}_{\geq 0}$ is submodular, continuous, and non-decreasing.*

- *The constraint set* $\mathcal{C} \subseteq 2^{\bar{\mathbb{R}}^n_{\geq 0}}$ *is a collection of covering constraints, where constraint* $S \in \mathcal{C}$ *is a subset of* $\bar{\mathbb{R}}^n_{\geq 0}$ *that is closed upwards and under limit.*

- *For each* $j \in [n]$*, the domain* U_j *(for variable* x_j*) is any subset of* $\bar{\mathbb{R}}_{\geq 0}$ *that is closed under limit.*

Here $\bar{\mathbb{R}}_{\geq 0}$ denotes $\mathbb{R}_{\geq 0} \cup \{\infty\}$. c is submodular if, $c(x)+c(y) \geq c(x \wedge y)+c(x \vee y)$, where $c(x \wedge y)$ and $c(x \vee y)$ are component-wise minimum and maximum, respectively, of x and y.

In online version of this problem, the constraints are revealed one at a time in any order. Whenever the algorithm, *Online-Covering*, gets a constraint that is not yet satisfied, it raises all variables in the constraint at the rates inversely proportional to their coefficients in the cost function, until the constraint is satisfied. This algorithm is Δ-competitive, where Δ is the maximum number of variables in any constraint. We use the following result from [46].

Theorem 2. Online-Covering *is* Δ*-competitive for online submodular cost covering.*

We omit the proof of this theorem, but we illustrate this approach and give a proof for the competitive ratio for this algorithm for the case of paging. We define the following notation.

- f_t : file requested at time t

- t' : time of next request to the file requested at time t

- x_t : indicator variable for the event that the file requested at time t was evicted before t'

- $R(t)$: set of times of the most recent request to each file until and including time t

- $\mathcal{Q}(t)$: represents every minimal set of files, which were requested before time t, such that for any $Q \in \mathcal{Q}(t)$, $Q \cup \{t\}$ violates the cache-size constraint. Thus, for paging, we define $\mathcal{Q}(t)$ as $\{Q \subseteq R(t) - \{t\} : |Q| = k\}$.
- T : time of last request

We formulate paging as the following covering problem (Paging-CP):

$$\begin{aligned} \min \quad & \sum_{t=1}^{T} x_t \\ \text{s.t.} \quad \forall t, \forall Q \in \mathcal{Q}(t) : \quad & \sum_{s \in Q} \lfloor x_s \rfloor \geq 1 \end{aligned}$$

Each constraint represents the following. At time t, when f_t is requested, for any subset Q of $\mathcal{Q}(t)$, it must be true that at least one of the files in Q is evicted to make space for f_t. Clearly, any feasible solution to the paging problem, is a feasible solution to Paging-CP. In particular, any optimal solution to the paging problem is a feasible solution to Paging-CP.

Now we describe the algorithm for paging. At each time step the algorithm may get multiple constraints. The algorithm considers the constraints in arbitrary order. When it gets a constraint that is not yet satisfied, it raises each variable in the constraint at unit rate until the constraint is satisfied. Whenever a variable reaches 1, the algorithm evicts the corresponding file from the cache. If the algorithm gets a constraint that is already satisfied, the algorithm does not do anything. We say the algorithm *uses* a constraint, if it isn't already satisfied when the algorithm considers it and raises the variables in the constraint, as described above, to satisfy it.

Each constraint in Paging-CP has exactly k variables. Now we show that this algorithm is k-competitive using the following potential function.

$$\phi = \sum_t \max(x_t^* - x_t, 0)$$

Here x^* denotes the value of x in the optimal solution. Initially, $\phi = \text{OPT}$ and $\text{ALG} = 0$. When the algorithm gets a constraint that is not satisfied, it raises each variable in the constraint at rate 1. So, the cost of the algorithm increases at the rate k. Also, ϕ decreases at unit rate because there is at least one variable x_s in the constraint such that $x_s < x_s^*$ (otherwise the constraint would already be satisfied). Thus, the algorithm maintains the invariant $\text{ALG}/k + \phi \leq \text{OPT}$. Since $\phi \geq 0$, it must be true that $\text{ALG} \leq k \cdot \text{OPT}$.

For the variants studied in this work, we use the approach outlined above, but with modifications in some cases. When we use the algorithm without any modifications, we omit the proofs for the competitive ratio. For these cases, the competitive ratio is the maximum number of variables in any constraint the algorithm uses. If we apply any modifications, we present complete proofs.

3.3 File caching with rental cost and zapping

3.3.1 Deterministic algorithms via online covering

We first consider rental caching with zapping. We present the deterministic algorithm, *Rent-Zap-Caching-CP*, for this problem by reducing it to online covering. Our algorithm is based on the greedy online covering algorithm outlined in Section 3.2. We use the notation defined in Section 3.2. In addition, we define the following indicator variable to account for renting and zapping files.

- $y_{t,s}$: indicator variable for the event that the file requested at time t pays the rental cost at time $s < t'$

- z_f: indicator variable for the event that the file f has been zapped

The following is the formulation of rental caching with zapping as a covering program:

$$\begin{aligned}
&\min && \sum_{t=1}^{T}\Big(cost(f_t)\cdot x_t + \lambda \sum_{t\le s<t'} size(f_t)\cdot y_{t,s}\Big) + N\sum_{f\in F} z_f \\
\text{s.t.} \quad & \forall t, \forall Q \in Q(t): && \Big(\sum_{s\in Q}\min(\lfloor x_s\rfloor + \lfloor z_{f_s}\rfloor, 1)\Big) + \lfloor z_{f_t}\rfloor \ge 1 && (I) \\
& \forall t, t\le s<t': && \lfloor y_{t,s}\rfloor + \lfloor x_t\rfloor + \lfloor z_{f_s}\rfloor \ge 1 && (II)
\end{aligned}$$

The first set of constraints (I) enforces the cache size at time t (similar to the constraints in Paging-CP), and the second set of constraints (II) says that at time s the file is being rented, has been evicted, or has been zapped. We call them *cache-size* constraints and *rent-evict-zap* constraints, respectively.

For each request, *Rent-Zap-Caching-CP* gets some cache-size constraints and some rent-evict-zap constraints. It considers the rent-evict constraints before the cache-size constraints. When there are multiple constraints of the same type (cache-size or rent-evict-zap), the algorithm considers them in arbitrary order. Whenever it gets a constraint that is not satisfied, it raises all variables in the constraint at rates inversely proportional to their coefficients, until the constraint is satisfied. When *Rent-Zap-Caching-CP* gets a rent-evict-zap constraint that is not yet satisfied, it raises x_t at rate $\frac{1}{cost(f_t)}$, raises $y_{t,s}$ at rate $\frac{1}{size(f_t)\lambda}$, and raises z_{f_t} at rate $\frac{1}{N}$ for each x_t, $y_{t,s}$, and z_{f_t} in the constraint. When it gets a cache-size constraint, it raises each x_s in the constraint at rate $\frac{1}{cost(f_s)}$.

Theorem 3. Rent-Zap-Caching-CP *is* $(2k+1)$*-competitive for rental caching with zapping.*

Proof. Each file has size at least 1, and hence, $|Q| \leq k$. Thus, each cache-size constraint has at most $2k + 1$ variables. Each rent-evict-zap constraint has exactly 3 variables. Theorem 2 implies that *Rent-Zap-Caching-CP* is $(2k + 1)$-competitive. □

Corollary 4. Rent-Zap-Caching-CP *is* $(2k + 1)$*-competitive for caching with zapping.*

Theorem 5. Rent-Zap-Caching-CP *is* $\max(2, k)$*-competitive for rental caching.*

Proof. For rental caching, there are no variables for zapping. Following the argument in the proof of Theorem 3, each cache-size constraint has at most k variables. Each rent-evict constraint has exactly 2 variables. Thus, *Rent-Zap-Caching-CP* is $\max(2, k)$-competitive for rental caching. □

3.3.2 Rental caching with infinite cache

We observe that for paging and for bit and fault models, when there is rental cost (with or without zapping), this algorithm achieves a better competitive ratio when $\lambda \geq \frac{1}{k}$. The key idea is that the rental cost forces eviction of files in way that the cache-size constraints are never violated. We present our analysis in this section and in Section 3.3.3. We present an algorithm for $\frac{1}{k^2} < \lambda < \frac{1}{k}$ that achieves a better competitive ratio than the algorithms in Section 3.3.1.

Consider the special case of rental caching where the cache has infinite size (i.e. $k = \infty$). This is equivalent to the rental caching problem without any size constraints. Even though there is no size constraint, this problem is still interesting because there is a rental cost for keeping files in the cache. However, if a file is evicted to save the rental cost, it will incur the retrieval cost on its next request.

Theorem 6. *If there is an* α*-competitive algorithm* $\mathrm{ALG}_{\mathrm{SR}}$ *for ski-rental, there is an* α*-competitive algorithm for rental caching with infinite cache.*

Proof. Fix any file f. We define phases as follows. A phase starts with a request to the file f and ends at the time step just before the next request to f. When a phase starts, the file f must be present in the cache and the earliest it can be evicted from the cache is at the next time step. Each phase is similar to the ski-rental problem, but there is one difference. In this problem, unlike ski-rental, the algorithm cannot evict (buy) the file at the very first step. It has to wait for at least one step before it can evict the file.

For each phase, excluding the first step, we define the *rent-or-evict* problem as follows. There is a file f in the cache. The cost of keeping a file in the cache is $\lambda size(f)$ per time step and the cost of evicting it is $cost(f)$. The algorithm doesn't know when the phase ends and at each step has to decide if it rents the file or if it evicts it.

Rent-or-evict is equivalent to ski-rental, where the cost of renting skis for a day is $\lambda size(f)$ and the cost of buying skis for the season is $cost(f)$. At each step, the algorithm either buys the skis (evicts the file) or rents the skis (rents the file). The algorithm does not know when the season (phase) ends. Given an α-competitive algorithm for ski-rental, the algorithm ALG_∞ for rental caching with infinite cache does the following. For a request to file f_t at time t, ALG_∞ brings the file into the cache. Starting at the next time step, it simulates $\mathrm{ALG}_{\mathrm{SR}}$ on f_t to decide how long to keep the file in the current phase. If $\mathrm{ALG}_{\mathrm{SR}}$ buys f at any time step during the phase, ALG_∞ evicts it from the cache at the beginning of that step. The total rental cost of ALG_∞ is same as the total rental cost of $\mathrm{ALG}_{\mathrm{SR}}$ and the total eviction cost of ALG_∞ is equal to the total cost of buying for $\mathrm{ALG}_{\mathrm{SR}}$.

Let $\mathrm{OPT}_{\mathrm{SR}}$ be the optimal cost of the ski-rental instance. In a phase, ALG_∞ cost is at most $\lambda size(f) + \alpha \cdot \mathrm{OPT}_{\mathrm{SR}}$ and the optimal cost is $\mathrm{OPT}_\infty = \lambda size(f) + \mathrm{OPT}_{\mathrm{SR}}$. The the competitive ratio of this algorithm is at most $\frac{\lambda size(f) + \alpha \cdot \mathrm{OPT}_{\mathrm{SR}}}{\lambda size(f) + \mathrm{OPT}_{\mathrm{SR}}}$. Since $\alpha > 1$, the competitive ratio is at most α. □

Corollary 7. *There is a 2-competitive deterministic algorithm for rental caching with infinite cache.*

Proof. The 2-competitive deterministic algorithm for ski-rental [38] and Theorem 6 together imply that ALG_∞ is 2-competitive. □

Corollary 8. *There is a* $(\frac{e}{e-1})$*-competitive randomized algorithm for rental caching with infinite cache.*

Proof. The $(\frac{e}{e-1})$-competitive randomized algorithm for ski-rental [38] and Theorem 6 together imply that ALG_∞ is $(\frac{e}{e-1})$-competitive. □

Theorem 9. *If* $\mathrm{ALG}_{\mathrm{SR}}$ *is an* α*-competitive optimal algorithm for ski-rental, then there is an* α*-competitive algorithm for rental paging when* $\lambda \geq \frac{1}{k}$.

Proof. Any optimal algorithm for ski-rental buys the skis in at most $\frac{1}{\lambda}$ days. Thus, by simulating $\mathrm{ALG}_{\mathrm{SR}}$ on each phase, ALG_∞ keeps each file in the cache for $1 + \frac{1-\lambda}{\lambda} = \frac{1}{\lambda}$ steps from its latest request. So, ALG_∞ uses a cache size of at most $\frac{1}{\lambda}$. When $\lambda \geq \frac{1}{k}$, ALG_∞ uses a cache of size at most k. Thus, the cache-size constraints are never violated and ALG_∞ is α-competitive for rental paging. □

Theorem 9 implies the following two corollaries.

Corollary 10. *When* $\lambda \geq \frac{1}{k}$*, there is a 2-competitive deterministic algorithm for rental paging.*

Corollary 11. *When* $\lambda \geq \frac{1}{k}$*, there is a* $(\frac{e}{e-1})$*-competitive randomized algorithm for rental paging.*

3.3.3 Improvements for high rental cost

For any $\gamma > 0$, we define *Rent-Zap-Caching-CP*$_\gamma$ as the algorithm that behaves like *Rent-Zap-Caching-CP* with the following modification. Whenever *Rent-Zap-Caching-CP*$_\gamma$ gets a rent-evict-zap constraint, it raises $y_{t,s}$ in the constraint at rate $\frac{\gamma}{size(f_t)\lambda}$. Note that *Rent-Zap-Caching-CP*$_1$ and *Rent-Zap-Caching-CP* are the same algorithm.

We define $\eta = \min_t \frac{\lambda size(f_t)}{\gamma cost(f_t)} k$. We show that when this quantity is at least 1, the cache-size constraints are not used by the algorithm.

Claim 12. *When* Rent-Zap-Caching-CP$_\gamma$ *considers a rent-evict-zap constraint for the file requested at time t, either f_t is evicted or zapped, or x_t increases by $\frac{size(f_t)\lambda}{cost(f_t)\gamma}$.*

Proof. Whenever *Rent-Zap-Caching-CP*$_\gamma$ considers a rent-evict-zap constraint, it raises x_t at rate $\frac{1}{cost(f_t)}$ and $y_{t,s}$ at rate $\frac{\gamma}{size(f_t)\lambda}$. Thus, when $y_{t,s}$ goes from 0 to 1, x_t increases by $\frac{size(f_t)\lambda}{cost(f_t)\gamma}$. It is possible that x_t or z_t reach 1 before $y_{t,s}$ reaches 1, implying the file is evicted or zapped, respectively. □

Claim 13. *When $\eta \geq 1$,* Rent-Zap-Caching-CP$_\gamma$ *does not use any cache-size constraints.*

Proof. We claim that, at any given time, if all the rent-evict constraints are satisfied, the cache-size constraints are satisfied too. We prove this by showing that each file is evicted within k steps from its latest request, by using just the rent-evict-zap constraints corresponding to the file in the cache.

For each file f, $size(f) \geq 1$. Therefore, $size(f)\lambda \geq \frac{1}{k}$.

Fix a file in the cache and let t be the time of the latest request to this file. If f_t is zapped by time s, we are done. Lets assume it is not zapped. Let $d = s - t$. Claim 12 implies that at time s the value of x_t is $\frac{size(f_t)\lambda}{cost(f_t)\gamma} d$. This value is 1 in at most $d = k$

steps. Thus, the file is evicted in at most k steps. Note that if z_{f_t} is 1 before x_t is 1, the file still does not use any slots in the cache. Thus, the cache-size constraints are never violated and the algorithm uses only the rent-evict-zap constraints. □

Claim 14. *η is at least 1, when $\lambda \geq \frac{1}{k}$ and $\gamma = 1$ for (a) rental paging with zapping and for rental caching with zapping for the cases of (b) bit model and (c) fault model.*

Proof. Plugging in $\gamma = 1$ in the expression for η gives $\eta = \min_t(\frac{\lambda size(f_t)}{cost(f_t)}k)$. Now consider the following cases.

(a) For paging, $cost(f_t) = size(f_t) = 1$. Thus, $\eta = k\lambda \geq 1$.

(b) For bit model, $cost(f_t) = size(f_t)$. Thus, $\eta = k\lambda \geq 1$.

(c) For fault model, $cost(f_t) = 1$. Thus, $\eta = size(f_t)k\lambda \geq k\lambda \geq 1$.

This proves the claim. □

Theorem 15. *When $\lambda \geq \frac{1}{k}$,* Rent-Zap-Caching-CP *is 3-competitive for (a) rental paging with zapping and for rental caching with zapping for the cases of (b) bit model and (c) fault model.*

Proof. Claims 13 and 14 together imply *Rent-Zap-Caching-CP* uses only the rent-evict-zap constraints. Each of these constraints has exactly 3 variables. Thus, using Theorem 2, RentalZappingPagingCILP is 3-competitive. □

Theorem 16. *When $\lambda \geq \frac{1}{k}$,* Rent-Zap-Caching-CP *is 2-competitive for (a) rental paging and for rental caching for the cases of (b) bit model and (c) fault model.*

Proof. Claims 13 and 14 together imply *Rent-Zap-Caching-CP* uses only the rent-evict-zap constraints. Each rent-evict-zap constraint has exactly 2 variables. Thus, by Theorem 2, the algorithm is 2-competitive for paging and for bit and fault models. □

Claim 17. *η is at least 1, when $\frac{1}{k^2} < \lambda < \frac{1}{k}$ and $\gamma = k\lambda$ for (a) rental paging with zapping and for rental caching with zapping for the cases of (b) bit model and (c) fault model.*

Proof. $\eta = \min_t(\frac{size(f_t)}{cost(f_t)})$ when $\gamma = k\lambda$. Now consider the following cases.

(a) For paging, $cost(f_t) = size(f_t) = 1$. Thus, $\eta = 1$.

(b) For bit model, $cost(f_t) = size(f_t)$. Thus, $\eta = 1$.

(c) For fault model, $cost(f_t) = 1$. Thus, $\eta = size(f_t) \geq 1$.

Thus, η is at least 1 in all three cases. □

Theorem 18. *When $\frac{1}{k^2} < \lambda < \frac{1}{k}$,* Rent-Zap-Caching-CP$_{k\lambda}$ *is $(1 + \frac{2}{k\lambda})$-competitive for (a) rental paging with zapping and for rental caching with zapping for the cases of (b) bit model and (c) fault model.*

Proof. Claims 13 and 17 together imply that the algorithm uses only the rent-evict-zap constraints.

Now we show that this modified algorithm is $(1 + \frac{1}{k\lambda})$-competitive. We use the following potential function for our proof:

$$\phi = \sum_{t=1}^{T} \Big(cost(f_t) \max(x_t^* - x_t, 0) + \sum_{t \leq s < t'} \lambda size(f_t) \max(y_{t,s}^* - y_{t,s}, 0) \Big) + \sum_{f \in F} N \max(z_f^* - z_f, 0)$$

Consider the rent-evict-zap constraint at time s for the file whose most recent request was at time t. When the algorithm raises the variables in this constraint, the cost of the algorithm increases at the rate $(2 + \gamma)$. Also, ϕ decreases at the rate $\min(1, \gamma)$. Thus, the algorithm maintains the invariant ALG$/(2 + \gamma) + \phi/(\min(1, \gamma)) \leq$ OPT. (It is true initially, because ALG $= 0$ and $\phi =$ OPT.) Since, $\phi \geq 0$, this implies that ALG $\leq \frac{2+\gamma}{\min(1,\gamma)}$OPT. Also, $\gamma = k\lambda \leq 1$. So, ALG $\leq (1 + \frac{2}{k\lambda})$OPT. □

Theorem 19. *When $\frac{1}{k^2} < \lambda < \frac{1}{k}$,* Rent-Zap-Caching-CP$_{k\lambda}$ *is $(1 + \frac{1}{k\lambda})$-competitive for (a) rental paging and for rental caching for the cases of (b) bit model and (c) fault model.*

Proof. Claims 13 and 17 together imply that the algorithm uses only the rent-evict-zap constraints. The proof for the competitive ratio of the modified algorithm is similar to the proof for Theorem 18. There is no variables for zapping. In this case, ALG increases at rate $(1 + \gamma)$ and ϕ decreases at rate at least $\min(1, \gamma)$. This implies that the invariant $\text{ALG}/(1 + \gamma) + \phi/(\min(1, \gamma)) \leq \text{OPT}$ holds. Since $\gamma = k\lambda \leq 1$ and $\phi \geq 0$, $\text{ALG} \leq (1 + \frac{1}{k\lambda})\text{OPT}$. □

3.3.4 Deterministic and randomized meta algorithms

In this section we present deterministic and randomized online algorithms for rental caching with or without zapping.

Theorem 20. *If there is an α-competitive algorithm* ALG_{SR} *for ski-rental, and a β-competitive algorithm* ALG_{C} *for file caching with (or without) zapping, then there is an $(\alpha + \beta)$-competitive algorithm for rental caching with (or, respectively without) zapping. If* ALG_{C} *and* ALG_{SR} *are both deterministic,* ALG_{RM} *is a deterministic online algorithm, otherwise it is a randomized online algorithm.*

We present the algorithm ALG_{RM}. If ALG_{C} is an algorithm for file caching, ALG_{RM} is an algorithm for rental caching. If ALG_{C} is an algorithm for caching with zapping, then ALG_{RM} is an algorithm for rental caching with zapping.

ALG_{RM} uses ALG_{SR} and ALG_{C} to generate a solution for rental caching. On an input sequence σ and cache size k, ALG_{RM} does the following. It simulates ALG_{C} on the input sequence σ and cache $\mathbb{C}_1$ of size k. In parallel, it simulates ALG_∞ on the request sequence σ and cache $\mathbb{C}_2$ of infinite size. ALG_∞ in turn simulates ALG_{SR} on each request, as described in Section 3.3.2. At any time, the cache of $\mathrm{ALG}_{\mathrm{RM}}$ contains the intersection of the files present in caches $\mathbb{C}_1$ and $\mathbb{C}_2$. When $\mathrm{ALG}_{\mathrm{C}}$ zaps a file, $\mathrm{ALG}_{\mathrm{RM}}$ zaps it.

Claim 21. *The total size of the items in the cache of* $\mathrm{ALG}_{\mathrm{RM}}$ *never exceeds* k*.*

Proof. Total size of all items in the cache of $\mathrm{ALG}_{\mathrm{C}}$ is at least the total size of all items in the cache of $\mathrm{ALG}_{\mathrm{RM}}$. This proves our claim, because $\mathrm{ALG}_{\mathrm{C}}$ maintains the invariant that the total size of items in the cache is at most k. □

We use $E[A]$ to denote the expected cost of algorithm A.

Claim 22. $E[\mathrm{ALG}_{\mathrm{RM}}] \leq E[\mathrm{ALG}_{\mathrm{SR}}] + E[\mathrm{ALG}_{\mathrm{C}}]$

Proof. $\mathrm{ALG}_{\mathrm{RM}}$ evicts a file, when at least one of $\mathrm{ALG}_{\mathrm{SR}}$ and $\mathrm{ALG}_{\mathrm{C}}$ evicted the file. For each eviction, we charge the cost of eviction for $\mathrm{ALG}_{\mathrm{RM}}$ to the algorithm that evicted the file, breaking ties arbitrarily. We charge the rental cost of $\mathrm{ALG}_{\mathrm{RM}}$ to the rental cost of $\mathrm{ALG}_{\mathrm{SR}}$. If $\mathrm{ALG}_{\mathrm{C}}$ is an algorithm for file caching with zapping, we charge the zapping cost of $\mathrm{ALG}_{\mathrm{RM}}$ to the zapping cost of $\mathrm{ALG}_{\mathrm{C}}$. This proves our claim. □

Also, $E[\mathrm{ALG}_{\mathrm{SR}}] \leq \alpha \cdot \mathrm{OPT}_{\mathrm{SR}} \leq \alpha \cdot \mathrm{OPT}$, and $E[\mathrm{ALG}_{\mathrm{C}}] \leq \beta \cdot \mathrm{OPT}_{\mathrm{C}} \leq \beta \cdot \mathrm{OPT}$, where $\mathrm{OPT}_{\mathrm{SR}}$ denotes the optimal cost for rental caching with infinite cache, $\mathrm{OPT}_{\mathrm{C}}$ denotes the optimal cost for caching (rental caching with zapping), and OPT denotes the optimal cost for rental caching (rental caching with zapping). So, $E[\mathrm{ALG}_{\mathrm{RM}}] \leq (\alpha + \beta)\mathrm{OPT}$, and hence, $\mathrm{ALG}_{\mathrm{RM}}$ is $(\alpha + \beta)$-competitive algorithm for rental caching (rental caching with zapping).

Theorem 20 implies the following corollaries.

Corollary 23. *The* 2*-competitive deterministic online algorithm for ski-rental [38] and* *the k-competitive deterministic online algorithm for file caching [55], give a $(k+2)$-competitive deterministic online algorithm for rental caching.*

Corollary 24. *The $(\frac{e}{e-1})$-competitive randomized online algorithm for ski-rental [38] and the $O(\log k)$-competitive randomized online algorithm for file caching [4, 52], give an $O(\log k)$-competitive randomized online algorithm for rental caching.*

Corollary 25. *The 2-competitive deterministic online algorithm for ski-rental [38] and the $(2k+1)$-competitive deterministic online algorithm for caching with zapping (Corollary 4), give a $(2k+3)$-competitive deterministic online algorithm for rental caching with zapping.*

3.4 Lower bounds

3.4.1 Rental paging

Theorem 26. *The competitive ratio of any deterministic algorithm for rental paging is at least (a) $2-\lambda$ when $\lambda > \frac{1}{k}$, and (b) $\frac{k+k\lambda}{1+k^2\lambda}$ for any λ.*

Proof. (a) $\lambda > \frac{1}{k}$: The adversary uses one file and does the following. It requests the file at the first time step, forcing the algorithm to bring the file into the cache, and waits for the algorithm to evict the file. When the algorithm evicts the file, the adversary requests it again at the next step. We split the entire sequence into phases such that each request is the beginning of a phase. Fix a phase p. Let the length of the phase be L_p. Note that, $L_p > 1$ as the algorithm cannot evict it at the time of request. The total cost of the algorithm is $L_p\lambda + 1$. The optimal cost is $\min(1, (L_p+1)\lambda)$. The ratio is minimized when $L_p = \frac{1}{\lambda} - 1$, and this ratio is $(\frac{1}{\lambda} - 1)\lambda + 1 = 2 - \lambda$.

(b) The adversary requests files from the set $\{1, 2, 3, \cdots, k+1\}$. At each step, the adversary requests a file that is not present in the cache of the algorithm. The algorithm faults at each time step and pays at least λ at each step. OPT pays the rental cost to keep k items in the cache at each time step and faults once in k steps. So, the ratio is at least $\frac{k+k\lambda}{1+k^2\lambda}$. $\square$

Theorem 27. *The competitive ratio of any randomized algorithm for rental paging is at least* $\frac{H_k+k^2H_k\lambda}{1+k^2H_k\lambda}$ *when* $\lambda < \frac{1}{k}$.

Proof. The adversary requests files from a set of $k+1$ files. At each step, the adversary requests a file with uniform probability over all files except the file requested at the previous step. We split the request sequence into *phases* as follows. The first phase starts with the first request in the input sequence. Each phase is the maximal subsequence with containing at most k distinct requests, and starts immediately after the previous phase ends.

We now show that the expected length of each phase is kH_k. When i distinct files have been requested in a phase, the probability of requesting some file that has not been requested in the phase is $\frac{k-i}{k}$. Thus the expected length of a phase is $\left(\sum_{i=0}^{k-1}\frac{k}{k-i}\right) = kH_k$.

Assume that i distinct files have been requested in the phase and the algorithm has $p \leq k$ files in the cache. At the next time step, the algorithm faults with a probability $\frac{k-p+1}{k}$ and pays a rental cost $p\lambda$. So, the expected cost of the algorithm is $1+\frac{1}{k}-p(\frac{1}{k}-\lambda)$. When $\frac{1}{k} > \lambda$, the cost is minimized when $p = k$. So, it pays $\frac{1}{k}$ eviction cost and $k\lambda$ rental cost at each step. OPT pays the same rental cost, but faults once in each phase. So, for each phase, the ratio is at least $\frac{H_k+k^2H_k\lambda}{1+k^2H_k\lambda}$. $\square$

3.4.2 Paging with zapping

Theorem 28. *For paging with zapping, the competitive ratio of any deterministic algorithm is at least* $\min(N-1, \frac{2Nk+N-(k+1)}{N+2k})$.

Proof. The adversary maintains a set of $k+1$ *active* files at all times. Every time a file is zapped by the algorithm, it is replaced in the active set by a new file that has never been requested before. At each time step, the adversary requests a file from the active set that is not present in the cache of the algorithm. We define a *zap-phase* as follows. A zap-phase ends every time the algorithm zaps a file and the following request marks the beginning of the next zap-phase. The first zap-phase starts with the first request of the input sequence. We define a *round* as follows. The first round starts with the first request of the input sequence. A round ends when the algorithm has zapped all of those $k+1$ files that were requested in the first $k+1$ time steps in that round (some other files may have been zapped too). The adversary discards all files from the active set at the end of a round and never requests a file that has been requested in any of the previous rounds. The total number of files zapped in a round is at least $k+1$. The adversary repeats the process for a large number of rounds.

Now we show the lower bound on the competitive ratio in each round. Consider any round. Let $T \geq k+1$ be the total number of files zapped by the algorithm in the round. Let Z_j be the length of zap-phase $j, 1 \leq j \leq T$.

Any deterministic algorithm faults at each time step and zaps a total of T files. So, the cost of any deterministic algorithm is at least, $ALG = NT + \sum_{j=1}^{T}(Z_j - 1) = (N-1)T + \sum_{j=1}^{T} Z_j$. Note that $\sum_{j=1}^{T} Z_j \geq T$.

Consider the offline algorithms (a) $\mathcal{F}_1$ that does not zap any files, and (b) $\mathcal{F}_2$ chooses one file from the set of the first $k+1$ files requested and zaps it at the $t = 1$.

Since $\mathcal{F}_1$ doesn't zap any files, in the first zap-phase it pays k to bring the first k files into the cache and then pays at most $\lceil \frac{Z_1-k}{k} \rceil$ in the remainder of the first phase. For any zap-phase $j > 1$, $\mathcal{F}_1$ pays at most $\lceil \frac{Z_j}{k} \rceil$.

$$\begin{aligned}
\mathcal{F}_1 &= (k + \lceil \frac{Z_1 - k}{k} \rceil) + \Big(\sum_{j=2}^{T} \lceil \frac{Z_j}{k} \rceil \Big) \\
&\leq (k + \frac{Z_1}{k}) + \Big(\sum_{j=2}^{T} \frac{Z_j}{k} + 1 \Big) \\
&= k + T - 1 + \sum_{j=1}^{T} \frac{Z_j}{k}
\end{aligned}$$

$\mathcal{F}_2$ incurs k faults in the first phase and 1 fault in each phase after that. It also pays N for zapping 1 file. The total cost of $\mathcal{F}_2$ is $k + (T-1) + N$.

Since $\min(\mathcal{F}_1, \mathcal{F}_2) \geq OPT$, the competitive ratio is at least

$$\frac{\sum_{j=1}^{T} Z_j + (N-1)T}{\min\left(k + T - 1 + \sum_{j=1}^{T} \frac{Z_j}{k}, k + T + N - 1\right)}$$

The ratio is minimized for fixed k, N and T, when $k+T-1+\sum_{j=1}^{T} \frac{Z_j}{k} = k+T+N-1$. Simplifying gives, $\sum_{j=1}^{T} Z_j = Nk$. So, the ratio is at least

$$\frac{Nk + (N-1)T}{N + k + T - 1}$$

Differentiating w.r.t. T gives,

$$\frac{(N-1)(N+k+T-1) - (Nk + (N-1)T)}{(N+k+T-1)^2}$$

The numerator is non-negative when $(N-1)^2 \geq k$ and is negative otherwise. Thus, for $(N-1)^2 \geq k$, the ratio is minimized when $T = k+1$. Otherwise, the ratio is minimized when T is large and this ratio is at least $N-1$. So, the competitive ratio is at least

$$\min\Big(N - 1, \frac{2Nk + N - (k+1)}{N + 2k}\Big)$$

□

3.5 Conclusions and further directions

The gaps in the bounds shown in this paper are given in 3.1. For the deterministic case, there is a gap of $O(k)$ for the variants of caching and weighted caching with rental cost, when $\lambda \geq \frac{1}{k}$. For the randomized case, there is a gap of $O(\log k)$ for all variants with rental cost and no zapping, when $\frac{1}{k^2 H_k} \leq \lambda < \frac{1}{k}$.

We do no give any randomized lower or upper bounds for variants with zapping but no rental cost. It would be interesting to explore these lower bounds and algorithms.

The models in this work assume uniform rental cost and uniform zapping cost in this study. Note that, in our model for rental caching, the total rental cost depends only on the size of a file. A natural extension would be to consider models with (arbitrary) non-uniform rental and zapping costs.

The online covering algorithm by Koufogiannakis and Young [46] generalizes many deterministic algorithms for online paging and caching problems. We use this approach for all the new variants studied in this work. The algorithms thus derived may not be optimal, as we show for the case of rental paging (or caching) and also for rental paging (or caching) with zapping. For the problems in this work, we were able to apply simple modifications to achieve upper bounds within constant factors of the lower bounds.

The primal-dual approach in [5, 7, 8, 13, 23] is a powerful framework for deriving randomized algorithms for online caching problems. It would be interesting to investigate if the approach can be used to give randomized algorithms for the variants studied in this work.

Chapter 4

Online Huffman Coding

4.1 Introduction

Huffman Coding was introduced by Huffman [35] in 1952. The Huffman Coding problem is the following. Given a probability distribution $P = \{ p_1, p_2, \cdot, p_n \}$ on $[n]$ and an encoding alphabet Σ, find a minimum cost prefix-free code (or prefix code) over Σ. A prefix code is one where no codeword is a prefix of any other codeword. The cost of a prefix code χ is given by $\sum_{i=1}^{n} p_i|\chi_i|$, where χ_i is the codeword assigned to i and $|\chi_i|$ is the length of χ_i. We assume that $0 < p_i \leq 1$ for each i. Huffman [35] give optimal algorithm for this problem.

In this work, we introduce the *Online Huffman Coding* problem. In online Huffman coding the symbols are sampled from a probability distribution P with replacement and are revealed one by one. The algorithm does not have any knowledge of the probability distribution or the future sequence of symbols, and has to assign codewords *online*. That is, whenever a symbol is seen for the first time, the algorithm must assign a codeword to it. The goal is to minimize the cost of the code.

Online Huffman coding is different from the *Adaptive Huffman Coding* problem.

In Adaptive Huffman Coding, the code is built dynamically as the symbols are revealed one by one. As the symbols are revealed, the algorithm maintains the weights and the corresponding Huffman code. Unlike online Huffman coding, the codeword assignment maybe updated as the process continues. Faller [24] and Gallager [27] independently introduce Adaptive Huffman Coding and give algorithms for it. Knuth [43] extend this work and make significant improvements to the algorithm. Vitter [59, 60] gives a new algorithm for Adaptive Huffman Coding.

We study online Huffman coding, for binary codes, in the competitive analysis framework. In this work, we consider the following forms of competitiveness. We say an algorithm ALG is *α-competitive* if, for every input σ, $\text{ALG}(\sigma) \leq \alpha \cdot \text{OPT}(\sigma)$. We say an algorithm ALG is *α-competitive asymptotically*, if there exists a constant c such that, for every input σ, $\text{ALG}(\sigma) \leq \alpha \cdot \text{OPT}(\sigma) + c$, or if $\text{ALG}(\sigma) \leq \alpha \cdot OPT(\sigma) + o(OPT(\sigma))$.

In Section 4.2 we present a lower bound of 10/9 on the competitive ratio of any deterministic algorithm. In Section 4.3 we present the Greedy-Huffman algorithm for online Huffman coding and analyze it. We show that the algorithm is 7-competitive. We also show that the algorithm is $(1 + o(1))$-competitive asymptotically.

We define the *Online Coding* problem as the variant of online Huffman coding where the prefix-free constraint is dropped.

We define *Slot Assignment* as follows. Given a probability distribution $P = \{p_1, p_2, p_3, \cdots, p_n\}$ on $[n]$ and n slots (named 1 through n) with costs $\{c_1, c_2, c_3, \cdots, c_n\}$ (the cost of j_{th} slot is c_j), find a minimum cost one-to-one assignment of slots to items. The cost of assigning slot j to item i is $p_i c_j$, and the cost of the assignment is the sum of the costs. In the online version of the problem, a slot must be assigned to an item when it is revealed for the first time.

Online coding is a special case of online slot assignment, where each slot corresponds to a node in the code tree, and the cost of the slot is equal to the length of the codeword (i.e. the length of the path from the root to the node). We present the algorithm GREEDY-CODING for slot assignment and show that it is optimal among all online algorithms for slot assignment.

4.2 Lower Bound (non-asymptotic)

In this section, we show a lower bound for the case when $n = 4$. Consider the following two distributions with four symbols: (a) $p_1 = p_2 = p_3 = p_4 = 0.25$, and (b) $p_1 = 1 - 3\epsilon, p_2 = p_3 = p_4 = \epsilon$. Figure 4.1 shows corresponding optimal Huffman codes, with optimal costs 2 and $1(1 - 3\epsilon) + 2\epsilon + 3\epsilon + 3\epsilon = 1 + 5\epsilon$, respectively.

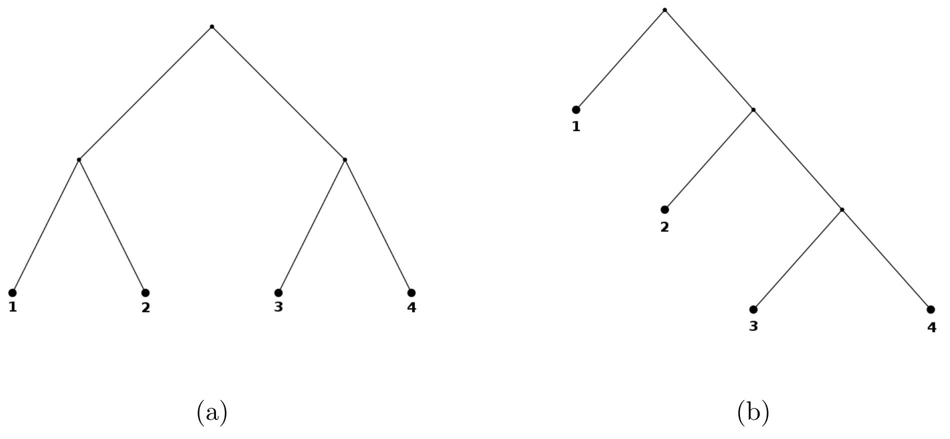


Figure 4.1: Optimal Huffman code for distributions (a) and (b), respectively

Theorem 29. *For any online algorithm, the competitive ratio is at least* 10/9.

Proof. The adversary chooses distributions (a) and (b) with probabilities p_a and $1 - p_a$, respectively. When the adversary chooses the distribution (b), it chooses the symbol with the highest weight with uniform probability over the four symbols. Thus, irrespective of the distribution used, the first symbol revealed to the algorithm is uniformly distributed all four symbols. Thus, when the first symbol is revealed, the algorithm does not know which distribution is being used. So, the algorithm assigns the first symbol a codeword probabilistically independent of the distribution used by the algorithm.

Let p be the probability that algorithm assigns a codeword of length 1 to the first symbol. With probability $(1 - p)$ it assigns a codeword of length more than 1.

If distribution (a) is used, the cost of the algorithm is at least 2.25 with probability p and 2 with probability $(1-p)$. Thus, expected ratio is at least $1.125p+(1-p) = 1 + 0.125p$.

If distribution (b) is used, the cost of the algorithm is at least $1 + 5\epsilon$ with probability p and is at least $2(1 - 3\epsilon)$ with probability $(p - 1)$. Thus, the expected ratio is at least $p + (1 - p)(2 - O(\epsilon)) = 2 - p - O(\epsilon)$. This ratio tends to $2 - p$ as ϵ tends to 0.

Thus, the expected ratio for the algorithm is $\max_{p_a}\{p_a(0.125p+1)+(1-p_a)(2-p)\}$.

This is the value of a 2-player zero-sum game with payoff matrix

$$\begin{bmatrix} 1.125 & 1 \\ 1 & 2 \end{bmatrix}$$

The value of this game is 10/9. Thus, any algorithm has a competitive ratio at least 10/9. □

4.3 Greedy-Huffman Algorithm

Now we present the algorithm Greedy-Huffman for online Huffman coding. Our algorithm is based on the approach given by Golin et al. [31] for a PTAS for the Huffman Coding with Letter Costs (HULC) problem. To understand this greedy assignment, we first explain the assignment without the prefix-free constraint and then translate it to the assignment with the prefix-free constraint.

We refer to online Huffman coding without the prefix-free constraint as the *Online Coding* problem. Here is the algorithm GREEDY-CODING for online coding: when a symbol is revealed for the first time, assign the shortest unassigned word of $\{0,1\}^*$ as a codeword, breaking ties by using the lexicographic order among codewords of equal length.

To construct a prefix-free code, GREEDY-HUFFMAN applies the following modification. When a symbol is revealed for the first time, if GREEDY-CODING were to assign a word χ_i, then GREEDY-HUFFMAN instead assigns the word $\chi_{i'} = pair(enc(|\chi_i|))01\chi_i$, where $|\chi_i|$ is the length of χ_i, $enc(l)$ is the binary encoding of the integer l, and $pair(x)$ replaces each 0 and each 1 in x by 00 and 11 respectively. The resulting code is prefix free [31].

Let GC be the expected cost of the GREEDY-CODING algorithm, and GH be the expected cost of the GREEDY-HUFFMAN algorithm. Let OPT be the optimal *offline* cost of online Huffman coding. Any prefix code has cost greater than or equal to the entropy of the distribution, $\mathcal{H} = -\sum_i p_i \log_2 p_i$ [54]. That is, OPT $\geq \mathcal{H}$.

Claim 30. GC $\leq$ OPT

Proof. Let the r.v. n_i be the number of distinct symbols sampled including symbol i when i is seen for the first time. GREEDY-CODING assigns to i a codeword of length $\lfloor \log_2 n_i \rfloor$, for an expected cost of

$$\mathrm{GC} = E\Big[\sum_i p_i \lfloor \log_2 n_i \rfloor\Big] \leq E\Big[\sum_i p_i \log_2 n_i\Big] \leq \sum_i p_i \log_2 E[n_i]$$

by concavity of the logarithm function.

For $j \neq i$, the probability that the j_{th} symbol appears before the i_{th} symbol is

$p_j/(p_j+p_i) < p_j/p_i$, so $E[n_i] \le 1 + \sum_{j\neq i} \frac{p_j}{p_i} = 1 + (1-p_i)/p_i = 1/p_i$. Thus

$$\mathrm{GC} \le \sum_i p_i \log_2 \frac{1}{p_i} = \mathcal{H}.$$

□

Theorem 31. $\mathrm{GH} \le \mathrm{OPT} + 4 + 2\log_2(\mathrm{OPT}+1)$. *Thus,* Greedy-Huffman *is asymptotically* $(1+o(1))$*-competitive.*

Proof. Let ℓ_i denote the length of the codeword in GC for symbol i, so that $\mathrm{GC} = \sum_i p_i \ell_i$. The corresponding codeword in GH has length $\ell_i + 2 + 2\lceil \log_2(\ell_i+1)\rceil \le \ell_i + 4 + 2\log_2(\ell_i+1)$. This, the concavity of the logarithm, and $\log(z+1) \le 1 + \log z$ for $z \ge 1$ imply that

$$\begin{aligned}
\mathrm{GH} &\le \sum_i p_i \big[\ell_i + 4 + 2\log_2(\ell_i+1)\big] \\
&= \sum_i p_i \ell_i + 4 + 2\log_2 \sum_i p_i(\ell_i+1) \\
&= \mathrm{GC} + 4 + 2\log_2(\mathrm{GC}+1) \\
&\le \mathrm{OPT} + 4 + \log_2(\mathrm{OPT}+1).
\end{aligned}$$

□

Theorem 32. Greedy-Huffman *is* 7*-competitive.*

Proof. Follows from Theorem 31 and $z + 4 + 2\log_2(z+1) < 7z$ for $z \ge 1$. □

4.4 Greedy-Coding is optimal for slot assignment

In this section, we show that Greedy-Coding is optimal for slot assignment, in the sense that its competitive ratio is minimum among all algorithms for slot assignment. If the probabilities and costs are sorted, the optimal cost is $\mathrm{OPT} = \sum_{i\in[n]} p_i c_{n-i}$.

Online coding is a special case of online slot assignment, where each slot corresponds to a node in the code tree, and the cost of the slot is equal to the length of the codeword (i.e. the length of the path from the root to the node).

Given P, consider the following random experiment. Sample items repeatedly and independently from P until each item has been sampled at least once. A *strategy* is a function that assigns a slot to an item when it is sampled for the first time. Note that Greedy-Coding is the strategy that assigns the minimum cost available slot.

We show that Greedy-Coding is optimal among all strategies. For our proof, we use the following explicit game in the extensive form to define all strategies. We first describe the explicit game, and then describe the underlying rooted tree.

The game has two players: {*Nature*, *Algorithm*}. *Nature* starts the game and then the players alternate turns. *Nature*, on its turn, plays an item from S according to the probability distribution P. If the item played by *Nature* has already been played, we say the move is a *redundant* move and the item is a *redundant* item. In this case, *Algorithm* has only one option, that is to do nothing (not assign any slots) and pass the turn back to *Nature*. Otherwise, the item played has not been played earlier. *Algorithm* assigns any one of the *available* slots. A slot is said to be *available* if it has not been assigned by *Algorithm* yet. The game ends when all slots have been assigned. The payoff at a leaf is equal to the total cost of the slot assignments along the path from the root to that leaf.

Now we describe the tree for the game. Let V be the set of all nodes in the tree and let L be the set of leaves. $V \setminus L$ can be partitioned into disjoint sets V_1 and V_2 such that, for any node in V_1, *Nature* moves, and for any node in V_2, *Algorithm* moves. We define the level of any node in this tree as its distance from the root. Thus, root is at level 0, its children at level 1, and so on. For any node x at level k, if k is odd, $x \in V_1$, otherwise, $x \in V_2$. When *Algorithm* assigns the final slot (no more slots are available), the corresponding child is a leaf node.

Each node in V_1 has n edges (to its children), one for each item *Nature* can play. Nodes in V_2 can be of two types.

- If the last move by *Nature* was redundant, the *Algorithm* node has just one edge, which corresponds to not assigning any slot.
- If the last move by *Nature* was not redundant, the *Algorithm* node has an edge for each available slot.

Each node in the tree corresponds to a partial sequence of items played and slots assigned. For any node x, I_x denotes the set of (distinct) items played by *Nature* to reach node x, and J_x denotes the set of slots assigned by *Algorithm* thus far.

Two distinct nodes $x, y \in V_2$ are *equivalent* iff $I_x = I_y$. In other words, the set of items sampled to reach x or y is the same, but the sequence to reach x is different from the sequence to reach y. Note that the equivalence of nodes is independent of the slot assignment.

A *strategy* is the specification of how player 2 moves on its turns. Note that, there is a one to one correspondence between the strategies in the game and the strategies for slot assignment. Any strategy for the game is a valid strategy for slot assignment.

Definition 33. *A* ***deterministic strategy*** *specifies one move for each node in* V_2.

Definition 34. *A behavioral strategy is* ***unbiased*** *if it specifies the same probability distribution for all children of each node* V_1.

Definition 35. *A behavioral strategy is* ***stateless*** *if, for any two equivalent nodes, the strategy specifies the same probability distribution.*

Note that Greedy-Coding is stateless and unbiased.

Claim 36. *If there is a randomized c-competitive strategy A, then there is a randomized c-competitive strategy A' that is behaviorally unbiased.*

Proof. We obtain A' from A as follows. Given any input sequence, A' first renames the n items according to a random permutation Π, then applies A to the renamed sequence. Running A' on any distribution P' is the same as running A on a distribution P obtained by randomly permuting P'. Clearly $\mathrm{OPT}(P) = \mathrm{OPT}(P')$. Thus, A' is c-competitive:

$$\begin{aligned}
A'(P') &= E_P[A(P)] \\
&\leq E_P[c \cdot \mathrm{OPT}(P)] \\
&= E_P[c \cdot \mathrm{OPT}(P')] \\
&= c \cdot \mathrm{OPT}(P')
\end{aligned}$$

We claim that A' is a behaviorally unbiased. We can implement A' as follows. Whenever A' receives an item s_i, A' feeds $\Pi(i)$ to A (in its simulation of A). If s_i is non-redundant, then A responds by assigning $\Pi(i)$ a slot j, making A' assign s_i slot j. Instead of choosing Π at the beginning, A' can choose $\Pi(i)$ only when s_i is seen for the first time. A' chooses $\Pi(i)$ uniformly at random from the remaining unassigned names in S.

A random node v is said to be in state M, if $I_v = M$. If a node is in state M and the next non-redundant request s_i is received, A' assigns a slot to s_i randomly from a distribution that depends only on M, but not on s_i (since the input $\Pi(i)$ to A is distributed from a distribution that is independent of s_i).

For any given node $v \in V_1$, let $x, y \in V_2$ be any two of its non-redundant children. Clearly, $J_x = J_y$, and by the reasoning above, the strategy A' specifies the same strategy for x and y.

Thus, A' is behaviorally unbiased. □

Claim 37. *Let A be a c-competitive behaviorally unbiased strategy. Then there is a c-competitive strategy that is behaviorally stateless and unbiased.*

Proof. We modify A to make it stateless without increasing its cost. We focus on the nodes in V_2.

Any $M \subseteq S$ induces an equivalence class of nodes E_M. Formally, a node $v \in V_2$ belongs to the equivalence class E_M, if $I_v = M$.

We fix a subset M. Consider all nodes in the equivalence class, E_M. For each node v in E_M, let $T(v)$ denote the subtree rooted at v. Note that, for any two equivalent nodes v and v', the two trees $T(v)$ and $T(v')$ are the same tree (call it T'), although the strategies for $T(v)$ and $T(v')$ may differ. We would like to replace, for each node v in E_M, the strategy for $T(v)$ by the minimum cost strategy for T'.

Some of the nodes are descendants of others, but only along paths with redundant moves. Note that each redundant move by *Nature* is followed by a move by *Algorithm* to pass the turn back to *Nature* (without assigning any slots). So, the redundant moves do not change the state of the game or the cost of a strategy. We do the following modification. For each node in E_M, define the *non-redundant subtree* $N(v)$ to be the subtree obtained from $T(v)$ by deleting the subtrees rooted at the redundant children of v. That is, remove any children w such that (v, w) is a redundant edge.

The non-redundant subtrees of $N(v)$ and $N(v')$, for any two distinct nodes v, $v' \in E_M$, are disjoint. Our goal is to select a single strategy Q' for a subtree $N(v')$ for some v', and replace all strategies for every tree $N(v)$ for every $v \in E$ by that Q'.

Define the $cost(N(v))$ to be the expected cost of a random leaf, for the given behaviorally unbiased strategy, conditioned on ending in one of the leaves of $N(v)$. Let $P(v)$ denote the probability of ending in some leaf of $N(v)$. Then, conditioned on ending in some leaf of some $N(v)$, the expected cost of A is $C = \sum_{v \in E_M} P(v) cost(N(v))$. There are infinitely many nodes in E_M, so there might not be one with minimum $cost(N(v))$, but there must still be at least one v' in E such that $N(v') \leq \frac{C}{\sum_{v \in E_M} P(v)}$. Take the strategy for this $N(v')$ and replace the strategy of $N(v)$ for every v in E_M. This leads to a valid overall strategy, and also does not increase the cost.

The modification described above ensures that, for any two nodes x and y in E_M, the modified strategy specifies the same probability distribution for x and y. Also, if the strategy is behaviorally unbiased before the above modification, it remains behaviorally unbiased after the modification.

We repeat the above operation for each of the 2^n subsets M of S. Note that when we do the operation for a subset M_i and then later for M_j, the operation for M_j preserves the property that, for any two nodes in the equivalence class for M_i, the probability distribution on the children is the same. Thus, after repeating the operation for all subsets M_j, the resulting strategy is stateless.

This proves that, for every behaviorally unbiased strategy, there is a behaviorally unbiased and stateless strategy that is at least as good. Since there are only finitely many stateless strategies, this proves the theorem. $\square$

Claim 38. GREEDY-CODING *algorithm has the minimum cost among all behaviorally unbiased and stateless strategies.*

Proof. A behaviorally unbiased and stateless strategy is a function g mapping each subset M of S to a probability distribution over the slots not yet assigned, where $g(M)$

is the probability distribution that the strategy uses to choose a slot that is assigned to the first non-redundant item sampled when in state M. A node x is said to be in state M, if $I_x = M$. If the move by *Nature* was redundant, *Algorithm* just passes the turn back to *Nature*. If the latest move is not redundant, let the item played by *Nature* be s_i. *Algorithm* specifies the probability distribution $g(M) = \{g_1(M), g_2(M), \cdots, g_n(M)\}$, where $g_j(M)$ is the probability of choosing slot j. If slot j has already been assigned (in an earlier step), $g_j(M) = 0$, otherwise $0 \leq g_j(M) \leq 1$, and the cost of choosing slot j is $p_i g_j(M) c_j$. The total expected cost given *Nature* played s_i in the previous move, therefore, is $p_i \sum_j g_j(M) c_j$. Note that the strategy does not know p_i. If h is the slot minimizing c_h for available slots, then the probability distribution for greedy choice (used by GREEDY-CODING) is defined as follows: $g_h(M) = 1$ and $g_j(M) = 0$ for $j \neq h$. We use the notation $g(M) = h$ to denote such a probability distribution.

To show that GREEDY-CODING is optimal, we give a proof by induction on n. The base case $n = 1$ is trivial (one item and one slot to be assigned).

For the induction step ($n \geq 2$), fix any optimal behaviorally unbiased and stateless strategy g. Note that, once the first item i is sampled and assigned a slot k, the remaining problem is equivalent to an instance with items $S\ i$ (with normalized probabilities $p/(1 - p_i)$) and all slots except k (with costs inherited from the original problem). By induction, for each of these subproblems, greedy is optimal. We denote the empty set with ϕ. Let us assume that, in g, the only non-greedy choice is the first. Let h be the slot minimizing c_h, that is $c_h \leq c_{k \neq h}$. If $g(\phi) = h$, we are done. So, we assume $g(\phi) \neq h$, and for all subsequent steps, the strategy matches the greedy choice (for the subproblems as defined above).

Let k be the slot minimizing c_k for $k \neq h$. That is, k is the slot with the minimum cost among all slots except h.

There must be a slot $m \neq h$ such that $g_(\phi) > 0$. The strategy chooses this slot with probability $g_m(\phi)$. After choosing m as the first slot, by the induction reasoning outlined above, $g(i) = h$ for each $1 \leq i \leq n$. We convert g as follows: $g_h(\phi) = g_h(\phi) + g_m(\phi)$ and set $g_m(\phi) = 0$.

This modification decreases the expected cost by

$$\sum_i p_i^2(c_m - c_h)g_m(\phi)$$

because the probability of choosing item i first is p_i, and if i is chosen, the savings is $p_i(c_m - c_h)g_m(\phi)$. Here, $g_m(\phi)$ is the probability before the modification is applied to the strategy.

This modification increases the expected cost by

$$\sum_i \sum_{j \neq i} p_i p_j p_j (c_k - c_h) g_m(\phi)$$

because the probability of choosing item i first, then item $j \neq i$, is $p_i p_j$, and the increase in cost when that happens is $p_j(c_k - c_h)$ (k is the slot assigned by the greedy choice in the second step). The increase can be simplified to

$$\sum_j p_j^2(1 - p_j)(c_k - c_h)g_m(\phi)$$

Since $c_k \leq c_m$ and $p_j > 0$ (otherwise, s_j will not be chosen), the increase is strictly smaller than the decrease, contradicting the optimality of g. $\square$

Theorem 39. Greedy-Coding *is optimal for slot assignment.*

Proof. Claims 36, 37, and 38 together imply that Greedy-Coding is optimal for online slot assignment. $\square$

4.5 Conclusions and further directions

We give lower bound and upper bounds for online Huffman coding. The bounds we show for online Huffman coding are tight within constant factors.

We show that Greedy-Coding is optimal among all online algorithms for slot assignment. We show the optimality, but don't give any lower or upper bounds on the competitive ratio of Greedy-Coding for greedy coding and slot assignment in this work. In an unpublished follow-up to this work, Young and Mathieu show that Greedy-Coding is $O(\log^2 n)$-competitive for slot assignment.

Bibliography

[1] http://mat.gsia.cmu.edu/color/instances.html.

[2] https://code.google.com/p/fastpc/.

[3] http://www.nlsde.buaa.edu.cn/∼kexu/benchmarks/set-benchmarks.htm.

[4] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1-2):203–218, 2000.

[5] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke. An $o(logk)$-competitive algorithm for generalized caching. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1681–1689. SIAM, 2012.

[6] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: A meta-algorithm and applications. *Theory OF Computing*, 8:121–164, 2012.

[7] N. Bansal, N. Buchbinder, and J. Naor. A primal-dual randomized algorithm for weighted paging. In *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*, pages 507–517. IEEE, 2007.

[8] N. Bansal, N. Buchbinder, and J.S. Naor. Randomized competitive algorithms for generalized caching. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 235–244. ACM, 2008.

[9] J.L. Bentley, D.D. Sleator, R.E. Tarjan, and V.K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[10] D. Bienstock. *Potential function methods for approximately solving linear programming problems: theory and practice*, volume 53. Kluwer Academic Pub, 2002.

[11] D. Bienstock and G. Iyengar. Approximating fractional packings and coverings in $O(1/\epsilon)$ iterations. *SIAM Journal on Computing*, 35(4):825–854, 2006.

[12] A. Borodin and R. El-Yaniv. Online computation and competitive analysis. 1998. *Cambridge University Ptess.*

[13] N. Buchbinder and J.S. Naor. Online primal-dual algorithms for covering and packing. *Mathematics of Operations Research*, 34(2):270–286, 2009.

[14] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, volume 193, 1997.

[15] M. Chrobak. Sigact news online algorithms column 17. *ACM SIGACT News*, 41 (4):114–121, 2010.

[16] M. Chrobak and J. Noga. Competitive algorithms for multilevel caching and relaxed list update. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 87–96. Society for Industrial and Applied Mathematics, 1998.

[17] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan. New results on server problems. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 291–300. Society for Industrial and Applied Mathematics, 1990.

[18] F. Chudak and V. Eleutério. Improved approximation schemes for linear programming relaxations of combinatorial optimization problems. *Integer Programming and Combinatorial Optimization*, pages 191–219, 2005.

[19] S.V. Cole, M. Khare, and N.E. Young. Implementation of the fastpc algorithm: data structures and distributions. 2011.

[20] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic programming. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[21] G.B. Dantzig. Linear programming and its extensions. *University of California and the Rand Corporation*, 1963.

[22] W. Eberly, M. Giesbrecht, P. Giorgi, A. Storjohann, and G. Villard. Solving sparse rational linear systems. In *Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 63–70. ACM, 2006.

[23] L. Epstein, C. Imreh, A. Levin, and J. Nagy-György. On variants of file caching. *Automata, Languages and Programming*, pages 195–206, 2011.

[24] N. Faller. An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pages 593–597, 1973.

[25] A. Fiat, R.M. Karp, M. Luby, L.A. McGeoch, D.D. Sleator, and N.E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[26] L.K. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM Journal on Discrete Mathematics*, 13(4):505–520, 2000.

[27] R. Gallager. Variations on a theme by huffman. *Information Theory, IEEE Transactions on*, 24(6):668–674, 1978.

[28] N. Garg and J. Koenemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM Journal on Computing*, 37(2): 630–652, 2007.

[29] CR Glassey and RM Karp. On the optimality of huffman trees. *SIAM Journal on Applied Mathematics*, pages 368–378, 1976.

[30] A.V. Goldberg. A natural randomization strategy for multicommodity flow and related algorithms. *Information Processing Letters*, 42(5):249–256, 1992.

[31] M.J. Golin, C. Mathieu, and N.E. Young. Huffman coding with letter costs: A linear-time approximation scheme. *SIAM Journal on Computing*, 41(3):684–713, 2012.

[32] M.D. Grigoriadis and L.G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM Journal on Optimization*, 4(1):86–107, 1994.

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

[34] M.D. Grigoriadis and L.G. Khachiyan. Approximate minimum-cost multicommodity flows in $(\epsilon^2 KNM)$ time. *Mathematical Programming*, 75(3):477–482, 1996.

[35] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[36] FK Hwang. Generalized huffman trees. *SIAM Journal on Applied Mathematics*, pages 124–127, 1979.

[37] S. Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3):384–409, 2002.

[38] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[39] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM, 1984.

[40] J.A. Kelner and D.A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 51–60. ACM, 2006.

[41] M. Khare and N.E. Young. Caching with rental cost and nuking. *Arxiv preprint arXiv:1208.2724*, 2012.

[42] P. Klein and N. Young. On the number of iterations for Dantzig-Wolfe optimization and packing-covering approximation algorithms. *Lecture Notes in Computer Science*, 1610:320–327, 1999.

[43] D.E. Knuth. Dynamic huffman coding. *Journal of algorithms*, 6(2):163–180, 1985.

[44] S.G. Kolliopoulos and N.E. Young. Approximation algorithms for covering/packing integer programs. *Journal of Computer and System Sciences*, 71(4):495–505, 2005.

[45] C. Koufogiannakis and N.E. Young. Beating simplex for fractional packing and

covering linear programs. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 494–504, 2007.

[46] Christos Koufogiannakis and Neal E. Young. Greedy Δ-approximation algorithm for covering with arbitrary constraints and submodular cost. *Algorithmica*, 2012.

[47] MK Kozlov, SP Tarasov, and LG Khachiyan. The polynomial solvability of convex quadratic programming. *USSR Comput. Math. Math. Phys.*, 20(5):223–228, 1980.

[48] L.L. Larmore and D.S. Hirschberg. A fast algorithm for optimal length-limited huffman codes. *Journal of the ACM (JACM)*, 37(3):464–473, 1990.

[49] A. Lopez-Ortiz and A. Salinger. Minimizing cache usage in paging. 2012.

[50] Z. Lotker, B. Patt-Shamir, and D. Rawitz. Rent, lease or buy: Randomized algorithms for multislope ski rental. *arXiv preprint arXiv:0802.2832*, 2008.

[51] W.W. Lu and MP Gough. A fast-adaptive huffman coding algorithm. *Communications, IEEE Transactions on*, 41(4):535–538, 1993.

[52] L.A. McGeoch and D.D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1):816–825, 1991.

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

[54] C.E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[55] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[56] S. Smale. On the average number of steps of the simplex method of linear programming. *Mathematical Programming*, 27(3):241–262, 1983.

[57] M.J. Todd. The many facets of linear programming. *Mathematical Programming*, 91(3):417–436, 2002.

[58] P.M. Vaidya. A new algorithm for minimizing convex functions over convex sets. *Mathematical Programming*, 73(3):291–341, 1996.

[59] J.S. Vitter. Design and analysis of dynamic huffman codes. *Journal of the ACM (JACM)*, 34(4):825–845, 1987.

[60] J.S. Vitter. Algorithm 673: dynamic huffman coding. *ACM Transactions on Mathematical Software (TOMS)*, 15(2):158–167, 1989.

[61] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. A simple model to generate hard satisfiable instances. *arXiv preprint cs/0509032*, 2005.

[62] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

[63] N.E. Young. Randomized rounding without solving the linear program. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 170–178. Society for Industrial and Applied Mathematics, 1995.

[64] N.E. Young. Sequential and parallel algorithms for mixed packing and covering. *IEEE Symposium on Foundations of Computer Science*, 2001.

[65] Neal E. Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002.

Appendix A

Packing and covering run times

In Table A.1 and Table A.2, P_ϵ and P_{OPT} are the times taken by primal-simplex for computing approximate and optimal solutions, respectively. Similar, D_ϵ and D_{OPT} are the times for dual-simplex, and B_ϵ and B_{OPT} are the times for barrier. F_ϵ is the time then by FASTPC. We use "$-$" to indicate that the algorithm didn't finish computing the solution. The first column in tables represents the input type. DIMACS inputs are denoted by D, set packing inputs by S, random inputs by R, and tomography inputs by T.

	r	c	d	P_ϵ	P_{OPT}	D_ϵ	D_{OPT}	B_ϵ	B_{OPT}	F_ϵ
D	1548	256	0.0078	-	-	-	-	-	-	569
D	37414	1728	0.0012	-	-	-	-	-	-	118
D	1972	276	0.0072	0	0	0	0	0	0	87
D	7082	422	0.0047	-	-	-	-	-	-	277
D	17376	850	0.0024	-	-	-	-	-	-	57
D	1624	174	0.0115	0	0	0	0	0	0	9
D	7850	394	0.0051	-	-	-	-	-	-	709

Table A.1 – continued from previous page

	r	c	d	P_ϵ	P_{OPT}	D_ϵ	D_{OPT}	B_ϵ	B_{OPT}	F_ϵ
D	7832	368	0.0054	-	-	-	-	-	-	52
D	23308	992	0.0020	-	-	-	-	-	-	48
D	27938	1242	0.0016	-	-	-	-	-	-	270
D	1016	160	0.0125	-	-	-	-	-	-	34
D	142	46	0.0435	0	0	0	0	0	0	3481
D	8200	422	0.0047	-	-	-	-	-	-	275
D	1160	72	0.0278	0	0	0	0	0	0	5
D	7892	370	0.0054	-	-	-	-	-	-	277
D	640	50	0.0400	0	0	0	0	0	0	214
D	7946	372	0.0054	-	-	-	-	-	-	97
D	472	94	0.0213	0	0	0	0	0	0	201
D	1204	148	0.0135	0	0	0	0	0	0	4
S	3688	4000	0.0005	0	0	0	0	0	0	827
D	40	22	0.0909	0	0	0	0	0	0	311
S	61476	40000	0.0001	4	20	6	7	97	160	0
S	78025	40000	0.0001	0	0	1	1	1	4	0
D	490000	2000	0.0010	129	1684	18	28	176	921	35
D	1472	250	0.0160	0	0	0	0	0	0	1
D	38190	770	0.0026	1	4	0	0	10	19	541
D	55794	500	0.0080	2	5	0	0	2	10	29
D	2912	128	0.0156	0	0	0	0	0	0	24
S	27843	595	0.0034	0	0	0	0	5	14	253

Table A.1 – continued from previous page

	r	c	d	P_ϵ	P_{OPT}	D_ϵ	D_{OPT}	B_ϵ	B_{OPT}	F_ϵ
S	320069	108000	0.0000	1	1	4	7	-	-	18479
D	2552	240	0.0083	0	0	0	0	0	0	215
D	19606	900	0.0022	1	2	0	0	142	238	22
D	43266	600	0.0033	1	3	0	0	12	25	3812
D	16744	392	0.0051	0	0	0	0	12	20	8745
S	126164	1534	0.0013	2	6	1	2	13	86	1915
S	126083	1534	0.0013	2	5	1	2	17	82	1903
D	42750	600	0.0033	1	2	0	0	19	36	32970
S	127012	1534	0.0013	7	8	1	2	12	77	1928
S	126556	1534	0.0013	23	24	1	2	10	77	1934
S	110039	1400	0.0014	4	5	1	2	12	73	1598
S	109677	1400	0.0014	1	3	1	2	9	68	1604
S	109602	1400	0.0014	16	18	1	2	9	61	1606
S	125983	1534	0.0013	22	23	1	2	19	85	1908
S	109380	1400	0.0014	1	2	1	2	11	70	1593
S	109402	1400	0.0014	4	4	1	2	24	72	1600
S	17832	450	0.0044	0	0	0	0	5	14	147
S	17828	450	0.0044	0	0	0	0	4	12	147
S	17810	450	0.0044	0	0	0	0	5	13	147
D	16338	900	0.0022	1	1	0	0	33	55	29
S	94290	1272	0.0016	3	3	1	2	7	51	1323
S	27932	595	0.0034	0	0	0	0	7	14	255

Table A.1 – continued from previous page

	r	c	d	P_ϵ	P_{OPT}	D_ϵ	D_{OPT}	B_ϵ	B_{OPT}	F_ϵ
D	10384	288	0.0069	0	0	0	0	4	6	88
S	80036	1150	0.0017	8	8	1	1	8	33	1064
S	94227	1272	0.0016	4	5	1	1	12	49	1326
D	242550	1000	0.0040	88	244	2	2	6	126	366
S	94128	1272	0.0016	1	3	1	1	9	52	1309
S	80073	1150	0.0017	9	9	1	1	6	45	1065
S	94309	1272	0.0016	7	7	1	1	8	54	1316
S	94228	1272	0.0016	3	3	1	1	7	47	1314
S	81069	1150	0.0017	3	3	1	1	57	134	1085
S	80259	1150	0.0017	1	2	1	1	11	40	1070
S	80852	1150	0.0017	1	1	1	1	10	39	1083
S	41096	760	0.0026	0	0	0	0	8	24	426
S	41264	760	0.0026	0	0	0	0	15	35	428
S	41315	760	0.0026	0	0	0	0	15	35	431
S	58580	945	0.0021	0	0	1	1	17	61	704
S	58625	945	0.0021	0	0	0	1	23	80	707
S	59187	945	0.0021	0	0	0	1	31	62	709
S	58246	945	0.0021	0	0	0	1	24	69	693
D	5472	192	0.0104	0	0	0	0	1	1	46
S	58550	945	0.0021	0	0	0	1	22	64	704
D	499652	2000	0.0020	111	1664	13	18	21	535	26146
S	41606	760	0.0026	0	0	0	0	10	34	429

Table A.1 – continued from previous page										
	r	c	d	P_ϵ	P_{OPT}	D_ϵ	D_{OPT}	B_ϵ	B_{OPT}	F_ϵ
S	41620	760	0.0026	0	0	0	0	10	30	429
D	11468	900	0.0022	0	1	0	0	26	45	156
S	27848	595	0.0034	0	0	0	0	14	37	250
D	24916	1000	0.0040	1	4	0	0	456	763	570
S	28144	595	0.0034	0	0	0	0	4	13	254
S	27857	595	0.0034	0	0	0	0	5	14	258
D	4680	256	0.0078	0	0	0	0	0	0	607
D	25280	512	0.0039	1	1	0	0	36	61	306
S	17795	450	0.0044	0	0	0	0	9	17	146
S	17875	450	0.0044	0	0	0	0	7	13	147
D	117724	1000	0.0040	8	16	1	1	7	38	283
D	8452	256	0.0078	0	0	0	0	0	1	75
D	20720	450	0.0044	1	1	0	0	21	36	32471
D	898898	2000	0.0020	398	3555	12	12	45	1695	391
S	371848	96000	0.0000	114	135	7	15	2405	4833	26809
D	13922	250	0.0160	0	0	0	0	3	4	109
D	16336	900	0.0022	0	1	0	0	33	56	66
D	491660	2000	0.0010	114	1509	25	43	565	1887	107
D	31336	500	0.0080	1	2	0	0	12	19	409
S	19147	12000	0.0002	0	0	0	0	2	3	2492
D	33500	900	0.0022	1	2	0	0	89	152	190
D	16520	900	0.0022	1	2	0	0	10	17	9673

Table A.1 – continued from previous page										
	r	c	d	P_ϵ	P_{OPT}	D_ϵ	D_{OPT}	B_ϵ	B_{OPT}	F_ϵ
D	224874	1000	0.0040	42	121	2	2	6	94	946
D	493416	2000	0.0010	135	1585	14	19	740	2324	734
D	125248	1000	0.0040	9	21	2	3	9	47	10
D	34686	900	0.0022	2	2	0	0	25	45	32383
D	7110	1000	0.0040	0	0	0	0	0	1	527
D	6436	500	0.0080	0	0	0	0	7	12	31
D	5880	200	0.0100	0	0	0	0	1	2	177
D	1510	190	0.0105	0	0	0	0	0	0	737
D	16526	900	0.0022	1	2	0	0	8	11	16
D	4224	162	0.0123	0	0	0	0	0	1	3238
D	13312	338	0.0059	0	0	0	0	7	12	65

Table A.1: Running times for CPLEX algorithms on set packing and DIMACS inputs

	r	c	d	P_ϵ	P_{OPT}	D_ϵ	D_{OPT}	B_ϵ	B_{OPT}	F_ϵ
R	13270	14287	0.4000	3173	-	7352	-	2563	-	1384
R	13270	14287	0.1000	1544	-	1782	-	933	-	1168
R	12270	12687	0.0009	1021	-	6322	-	427	-	1992
R	11270	11087	0.4000	-	-	-	-	1118	-	1078
R	12270	12687	0.4000	-	-	-	-	1742	-	1226
R	9270	7887	0.1000	-	-	-	-	-	-	603
T	178030	9025	0.0086	-	-	-	-	-	-	16627

Table A.2 – continued from previous page

	r	c	d	P_ϵ	P_{OPT}	D_ϵ	D_{OPT}	B_ϵ	B_{OPT}	F_ϵ
T	353079	9300	0.0088	-	-	-	-	-	-	12
T	320779	7600	0.0098	-	-	-	-	-	-	9
T	490076	16800	0.0065	-	-	-	-	1130	30902	24
R	10270	9487	0.4000	-	-	-	-	1122	589251	919
R	9270	7887	0.4002	5024	29137	4748	26971	779	58843	769
R	12270	12687	0.1000	-	-	-	-	641	68751	1024
R	6325	7145	0.4000	71810	71816	50370	57475	314	15661	508
R	5725	6245	0.4001	37989	37992	28238	31722	227	11990	433
R	3925	3545	0.1001	547	1346	4737	5082	13	124	155
T	178030	9025	0.0086	5	9	0	9	1	9	16348
T	347404	9025	0.0090	8	18	0	18	0	18	12
R	11270	11087	0.1000	965205	965250	476371	1424136	428	428138	875
R	5125	5345	0.4001	32185	32186	24773	76996	157	5251	355
R	10270	9487	0.1000	219499	219511	324895	527417	315	29010	735
R	4525	4445	0.4001	1450	11589	1854	2902	116	1830	279
R	6325	7145	0.0017	335	488	497	636	43	100	884
R	3325	2645	0.0999	227	493	2530	3752	5	39	105
R	5725	6245	0.0019	303	420	338	434	32	77	744
R	5125	5345	0.0022	285	374	282	346	23	56	620
T	254099	6230	0.0120	6	34	95	181	3	24	8716
R	4525	4445	0.0027	183	239	178	216	14	33	490
R	3925	3545	0.0034	119	155	169	194	9	25	366

Table A.2 – continued from previous page

	r	c	d	P_ϵ	P_{OPT}	D_ϵ	D_{OPT}	B_ϵ	B_{OPT}	F_ϵ
R	6325	7145	0.1000	15952	21942	43499	90139	84	3524	394
R	3925	3545	0.4000	758	4020	4715	4717	40	1818	213
R	5725	6245	0.0999	9708	12749	39532	71508	61	2355	327
R	5125	5345	0.1000	16096	45224	9616	33145	44	782	268
R	3325	2645	0.3999	313	1040	1520	2877	23	119	154
R	10270	9487	0.0013	2229	4023	11852	15646	187	465	1297
R	13270	14287	0.0008	10142	17912	8539	11256	341	998	2375
R	4525	4445	0.1000	3084	3084	12325	22087	30	174	209
R	9270	7887	0.0015	1717	2977	2631	3412	139	351	1035
R	11270	11087	0.0011	3182	5992	4748	6253	210	619	1650
T	52409	210	0.0567	1	1	2	2	1	5	958
R	3325	2645	0.0045	55	70	60	73	3	8	256

Table A.2: Running times for CPLEX algorithms on random and tomography inputs