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

Dissertation Committee:

Dr. Neal E Young, Chairperson
Dr. Marek Chrobak
Dr. Stefano Lonardi
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 $\frac{10}{9}$ on the competitive ratio of any deterministic online algorithm.
Contents

List of Figures xi
List of Tables xii

1 Introduction 1
  1.1 Empirical study of algorithms for packing and covering . . . . . . . . . . 3
  1.2 File caching with rental cost and zapping . . . . . . . . . . . . . . . . 4
  1.3 Online Huffman Coding . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2 Empirical study of algorithms for packing and covering 8
  2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
    2.1.1 Traditional algorithms for solving linear programs . . . . . . . . 9
    2.1.2 Approximation algorithms for packing and covering . . . . . . . 10
  2.2 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
    2.2.1 Machine specifications . . . . . . . . . . . . . . . . . . . . . . . 12
    2.2.2 FASTPC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
    2.2.3 CPLEX . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
    2.2.4 Inputs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
    2.2.5 Value of $\epsilon$ . . . . . . . . . . . . . . . . . . . . . . . . . 14
  2.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
    2.3.1 CPLEX algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . 15
      2.3.1.1 Set packing and DIMACS inputs . . . . . . . . . . . . . 16
      2.3.1.2 Random and tomography inputs . . . . . . . . . . . . 23
    2.3.2 FASTPC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
    2.3.3 CPLEX vs FASTPC . . . . . . . . . . . . . . . . . . . . . . . . . 25
    2.3.4 Hybrid Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . 27
  2.4 Conclusions and future directions . . . . . . . . . . . . . . . . . . . . . 28

3 File caching with rental cost and zapping 47
  3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
    3.1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
    3.1.2 Previous work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
      3.1.2.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . 50
    3.1.3 Our contributions . . . . . . . . . . . . . . . . . . . . . . . . . . 51
    3.1.4 Other work on rental paging . . . . . . . . . . . . . . . . . . . . 53
  3.2 Online covering approach . . . . . . . . . . . . . . . . . . . . . . . . . . 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>File caching with rental cost and zapping</td>
<td>57</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Deterministic algorithms via online covering</td>
<td>57</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Rental caching with infinite cache</td>
<td>59</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Improvements for high rental cost</td>
<td>62</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Deterministic and randomized meta algorithms</td>
<td>65</td>
</tr>
<tr>
<td>3.4</td>
<td>Lower bounds</td>
<td>67</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Rental paging</td>
<td>67</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Paging with zapping</td>
<td>69</td>
</tr>
<tr>
<td>3.5</td>
<td>Conclusions and further directions</td>
<td>71</td>
</tr>
<tr>
<td>4</td>
<td>Online Huffman Coding</td>
<td>72</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>72</td>
</tr>
<tr>
<td>4.2</td>
<td>Lower Bound (non-asymptotic)</td>
<td>74</td>
</tr>
<tr>
<td>4.3</td>
<td>GREEDY-HUFFMAN Algorithm</td>
<td>75</td>
</tr>
<tr>
<td>4.4</td>
<td>GREEDY-CODING is optimal for slot assignment</td>
<td>77</td>
</tr>
<tr>
<td>4.5</td>
<td>Conclusions and further directions</td>
<td>85</td>
</tr>
<tr>
<td>A</td>
<td>Packing and covering run times</td>
<td>93</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Iterations for primal-simplex on set packing and DIMACS inputs . . . 17  
2.2 Time per iteration for primal-simplex on set packing and DIMACS inputs 18  
2.3 Iterations for dual-simplex on set packing and DIMACS inputs . . . . 19  
2.4 Time per iteration for dual-simplex on set packing and DIMACS inputs 20  
2.5 Iterations for barrier-simplex on set packing and DIMACS inputs . . . 21  
2.6 Time per iteration for barrier-simplex on set packing and DIMACS inputs 22  
2.7 Comparison of CPLEX algorithms to compute $(1 \pm \epsilon)$-solution for $\epsilon = 0.01$ on set packing and DIMACS inputs 23  
2.8 Iterations for primal-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution 30  
2.9 Iterations for primal-simplex on random and tomography inputs for exact solution 31  
2.10 Time per iteration for primal-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution 32  
2.11 Time per iteration for primal-simplex on random and tomography inputs to compute exact solution 33  
2.12 Iterations for dual-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution 34  
2.13 Iterations for dual-simplex on random and tomography inputs for exact solution 35  
2.14 Time per iteration for dual-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution 36  
2.15 Time per iteration for dual-simplex on random and tomography inputs to compute exact solution 37  
2.16 Iterations for barrier-simplex on set packing and DIMACS inputs . . . 38  
2.17 Time per iteration for barrier-simplex on set packing and DIMACS inputs 39  
2.18 Comparison of CPLEX algorithms to compute $(1 \pm \epsilon)$-solution for $\epsilon = 0.01$ on random and tomography inputs 40  
2.19 Running time for FASTPC for $\epsilon = 0.01$ 41  
2.20 Time / predicted time for FASTPC for $\epsilon = 0.01$ 42  
2.21 FASTPC vs dual-simplex on set packing and DIMACS inputs 43  
2.22 FASTPC vs barrier on random and tomography inputs 45  
2.23 Value of $\epsilon$ at crossover (log-log scale) 46  
4.1 Optimal Huffman code for distributions (a) and (b), respectively 74
List of Tables

2.1 Observed lower bounds for CPLEX algorithms on set packing and DIMACS inputs .......................... 16
2.2 Observed lower bounds for the primal-simplex and the dual-simplex algorithms on random and tomography inputs ................................................................. 24
3.1 Competitive ratios for the variants with rental cost and zapping. In column 2, “Det” represents Deterministic and “Rand” represents Randomized. 52
A.1 Running times for CPLEX algorithms on set packing and DIMACS inputs 98
A.2 Running times for CPLEX algorithms on random and tomography inputs 100
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 $\text{ALG}(I)$ be the cost of the algorithm ALG on input $I$ and let $\text{OPT}(I)$ be the cost of the optimal solution. ALG is an $\alpha$-approximation algorithm for a maximization (or, minimization) problem if, for each input $I$, $\text{ALG}(I) \geq \text{OPT}(I)/\alpha$ (or, $\text{ALG}(I) \leq \alpha \cdot \text{OPT}(I)$). 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 $\sigma$, where $\sigma(t)$ denotes the input revealed at time $t$. Let $\text{ALG}(\sigma)$ be the cost of algorithm ALG on input sequence $\sigma$, and let $\text{OPT}(\sigma)$ be the corresponding optimal offline cost. ALG is an $\alpha$-competitive online algorithm for a maximization (or, minimization) problem if, for every input sequence $\sigma$, $\text{ALG}(\sigma) \geq \text{OPT}(\sigma)/\alpha + c$ (or, $\text{ALG}(\sigma) \leq \alpha \cdot \text{OPT}(\sigma) + c$), 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 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 $0.01 \leq \text{density} \leq 0.15$, 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 $\text{size}(f)$ (a positive integer) and retrieval cost $\text{cost}(f)$ (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 $f$, $cost(f) = \text{size}(f)$ 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 $\lambda$ (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 $\Delta$-approximation algorithm for covering with submodular cost, where $\Delta$ is the maximum number of variables in any constraint. Their algorithm is $\Delta$-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, though 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 \( \Sigma \), find a minimum cost prefix-free code (or prefix code) over \( \Sigma \). The cost of a prefix code \( \chi \) is given by \( \sum_{i=1}^{n} p_i |\chi_i| \), where \( \chi_i \) is the codeword assigned to \( i \) and \( |\chi_i| \) is the length of \( \chi_i \). 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 $O(c^6 L)$, 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 $O(rc^2 L + cM(r)L)$ 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 × 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 \( c = 100 \)). 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 \( \epsilon \)

The value of \( \epsilon \) 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 $\delta$ 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 $\epsilon$ 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 $O((r+c)^{0.5})$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$(1 \pm \epsilon)$-solution</th>
<th>Exact solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>primal-simplex</td>
<td>Iterations $O(1)$</td>
<td>$(r+c)^{0.5}$</td>
</tr>
<tr>
<td></td>
<td>Time/Iteration $(r+c)^{1.5}$</td>
<td>$(r+c)^{1.5}$</td>
</tr>
<tr>
<td></td>
<td>Time $(r+c)^{1.5}$</td>
<td>$(r+c)^2$</td>
</tr>
<tr>
<td>dual-simplex</td>
<td>Iterations $(r+c)^{0.5}$</td>
<td>$(r+c)^{0.5}$</td>
</tr>
<tr>
<td></td>
<td>Time/Iteration $(r+c)$</td>
<td>$(r+c)$</td>
</tr>
<tr>
<td></td>
<td>Time $(r+c)^{1.5}$</td>
<td>$(r+c)^{1.5}$</td>
</tr>
<tr>
<td>barrier</td>
<td>Iterations $3 - 6$</td>
<td>$5 - 10$</td>
</tr>
<tr>
<td></td>
<td>Time/Iteration $(r+c)^{1.5}$</td>
<td>$(r+c)^2$</td>
</tr>
<tr>
<td></td>
<td>Time $(r+c)^{1.5}$</td>
<td>$(r+c)^2$</td>
</tr>
</tbody>
</table>

Table 2.1: Observed lower bounds for CPLEX algorithms on set packing and DIMACS inputs
Figure 2.1: Iterations for primal-simplex on set packing and DIMACS inputs
Figure 2.2: Time per iteration for primal-simplex on set packing and DIMACS inputs
Figure 2.3: Iterations for dual-simplex on set packing and DIMACS inputs
Figure 2.4: Time per iteration for dual-simplex on set packing and DIMACS inputs
Figure 2.5: Iterations for barrier-simplex on set packing and DIMACS inputs
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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$\epsilon$</th>
<th>Exact solution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$(1 \pm \epsilon)$-solution</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$d &lt; 0.01$</td>
<td>$d \geq 0.01$</td>
</tr>
<tr>
<td>primal-simplex</td>
<td>$(r + c)^{1.5}$</td>
<td>$(r + c)^{2.5}$</td>
</tr>
<tr>
<td>Iterations</td>
<td>$(r + c)^{1.5}$</td>
<td>$(r + c)^{1.5}$</td>
</tr>
<tr>
<td>Time/Iteration</td>
<td>$(r + c)^{2.5}$</td>
<td>$(r + c)^{2.5}$</td>
</tr>
<tr>
<td>Time</td>
<td>$(r + c)^3$</td>
<td>$(r + c)^3$</td>
</tr>
<tr>
<td>dual-simplex</td>
<td>$(r + c)^{1.5}$</td>
<td>$(r + c)^{2.5}$</td>
</tr>
<tr>
<td>Iterations</td>
<td>$(r + c)^{1.5}$</td>
<td>$(r + c)^{1.5}$</td>
</tr>
<tr>
<td>Time/Iteration</td>
<td>$(r + c)^{2.5}$</td>
<td>$(r + c)^{2.5}$</td>
</tr>
<tr>
<td>Time</td>
<td>$(r + c)^3$</td>
<td>$(r + c)^3$</td>
</tr>
<tr>
<td>barrier</td>
<td>$5 - 9$</td>
<td>$(r + c)^3$</td>
</tr>
<tr>
<td>Iterations</td>
<td>$5 - 9$</td>
<td>$(r + c)^3$</td>
</tr>
<tr>
<td>Time/Iteration</td>
<td>$(r + c)^3$</td>
<td>$(r + c)^3$</td>
</tr>
<tr>
<td>Time</td>
<td>$(r + c)^3$</td>
<td>$(r + c)^3$</td>
</tr>
</tbody>
</table>

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 $\epsilon$. 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 $\leq 0.15$, 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 $\epsilon$ 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 $\epsilon$ 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 $\epsilon$ substantially smaller than is found by the current barrier algorithm at crossover.

In all of our current trials the crossover occurs when $\epsilon$ 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 $\epsilon$ 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 $\epsilon$ 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 $\epsilon$ 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 $\epsilon$, this algorithm is slower than Fastpc but, for smaller values of $\epsilon$ 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 $\epsilon$. 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 $\epsilon$, the hybrid algorithm where the pre-crossover step uses this algorithm may be faster than the barrier algorithm.
Figure 2.8: Iterations for primal-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution
Figure 2.9: Iterations for primal-simplex on random and tomography inputs for exact solution
Figure 2.10: Time per iteration for primal-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution
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
Figure 2.13: Iterations for dual-simplex on random and tomography inputs for exact solution
Figure 2.14: Time per iteration for dual-simplex on random and tomography inputs to compute $(1 \pm \epsilon)$-solution
Figure 2.15: Time per iteration for dual-simplex on random and tomography inputs to compute exact solution
Figure 2.16: Iterations for barrier-simplex on set packing and DIMACS inputs
Figure 2.17: Time per iteration for barrier-simplex on set packing and DIMACS inputs
Figure 2.18: Comparison of CPLEX algorithms to compute \((1 \pm \epsilon)\)-solution for \(\epsilon = 0.01\) on random and tomography inputs
Figure 2.19: Running time for FASTPC for $\epsilon = 0.01$
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
Figure 2.22: FASTPC vs barrier on random and tomography inputs

(c) $0.01 \leq \text{Density} < 0.15$

(d) $\text{Density} \geq 0.15$
Figure 2.23: Value of $\epsilon$ 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 $\text{ALG}(\sigma)$ be the cost of algorithm ALG on request sequence $\sigma$, and let $\text{OPT}(\sigma)$ be the corresponding optimal offline cost. ALG is $\alpha$-competitive if, for every request sequence $\sigma$, $\text{ALG}(\sigma) \leq \alpha \cdot \text{OPT}(\sigma) + c$, 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 $\lambda$ (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 $\text{size}(f) \cdot d$. 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 \( \text{cost}(f) = 1 \) and \( \text{cost}(f) = \text{size}(f) \), 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 \( \text{cost}(f) = 1 \) and \( \text{cost}(f) = \text{size}(f) \), 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 ($\text{cost}(f) = \text{size}(f)$ for each file $f$) and fault model ($\text{cost}(f) = 1$ 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 $\left(\frac{k}{k-h+1}\right)$-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 \(\Delta\)-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 \(\Delta\) variables. They show that their algorithm is \(\Delta\)-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.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rental paging, bit model, Rental caching:</td>
<td>Det</td>
<td>Lower Bound</td>
<td>Upper Bound</td>
</tr>
<tr>
<td>fault model</td>
<td></td>
<td>$\frac{\lambda}{k^2} \leq \lambda &lt; \frac{1}{k}$</td>
<td>$2 - \lambda$</td>
</tr>
<tr>
<td></td>
<td>Rand</td>
<td>$\lambda \leq \frac{1}{k}$</td>
<td>$2(\lambda + 1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda &lt; \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda \leq \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td>Weighted rental caching, Weighted caching</td>
<td>Det</td>
<td>$\lambda \geq \frac{1}{k}$</td>
<td>$2 - \lambda$</td>
</tr>
<tr>
<td>with zapping</td>
<td></td>
<td>$\lambda &lt; \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td></td>
<td>Rand</td>
<td>$\lambda &lt; \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda \leq \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda \leq \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td>Caching with zapping</td>
<td>Det</td>
<td>$\lambda \geq \frac{1}{k}$</td>
<td>$2 - \lambda$</td>
</tr>
<tr>
<td></td>
<td>Rand</td>
<td>$\lambda &lt; \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda \leq \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda \leq \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td>Paging with zapping</td>
<td>Det</td>
<td>$\lambda \geq \frac{1}{k}$</td>
<td>$2 - \lambda$</td>
</tr>
<tr>
<td></td>
<td>Rand</td>
<td>$\lambda &lt; \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda \leq \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\lambda \leq \frac{1}{k}$</td>
<td>$\frac{k + k\lambda}{1 + k^2\lambda}$</td>
</tr>
</tbody>
</table>

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 deter-
ministic 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 $\lambda$ is $O\left(\frac{1}{k^2 H_k}\right)$ 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 $\lambda$ is $O\left(\frac{1}{k^2 H_k}\right)$ 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 $\lambda$ is $O\left(\frac{1}{k^2 H_k}\right)$ 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 $M\frac{d}{k}$ when $\lambda > \frac{1}{k}$, which matches the presented in upper bound in this work. They show an upper bound of $\max\left(k, \frac{(k+1)}{1+\lambda(k-1)}\right)$ on the competitive ratio of $M\frac{d}{k}$ 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 $\lambda$ is $\Omega\left(\frac{1}{k}\right)$, 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$_\frac{d}{k}$, FWF, FWF$_\frac{d}{k}$, FIFO, FIFO$_\frac{d}{k}$, 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 $\Delta$-approximation algorithm for Submodular Cost Covering.

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

- The cost function $c : \mathbb{R}_{\geq 0}^n \rightarrow \mathbb{R}_{\geq 0}$ is submodular, continuous, and non-decreasing.
• The constraint set \( C \subseteq 2^{\mathbb{R}_{\geq 0}^n} \) is a collection of covering constraints, where constraint \( S \in C \) is a subset of \( \mathbb{R}_{\geq 0}^n \) that is closed upwards and under limit.

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

Here \( \mathbb{R}_{\geq 0} \) denotes \( \mathbb{R}_{\geq 0} \cup \{\infty\} \). \( c \) is submodular if, \( c(x) + c(y) \geq c(x \land y) + c(x \lor y) \), where \( c(x \land y) \) and \( c(x \lor y) \) 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 \( \Delta \)-competitive, where \( \Delta \) is the maximum number of variables in any constraint. We use the following result from [46].

**Theorem 2.** Online-Covering is \( \Delta \)-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 \( t \)
Q(t) : represents every minimal set of files, which were requested before time t, such that for any Q ∈ Q(t), Q ∪ {t} violates the cache-size constraint. Thus, for paging, we define Q(t) as \{Q ⊆ R(t) − \{t\} : |Q| = k\}.

T : time of last request

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

\[
\min \sum_{t=1}^{T} x_t \\
\text{s.t. } \forall t, \forall Q \in Q(t) : \sum_{s \in Q} \lfloor x_s \rfloor \geq 1
\]

Each constraint represents the following. At time t, when \(f_t\) is requested, for any subset \(Q\) of \(Q(t)\), 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, \( \phi \) 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, \textit{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:

$$
\begin{align*}
\text{min} & \quad \sum_{t=1}^{T} \left( \text{cost}(f_{t}) \cdot x_{t} + \lambda \sum_{t \leq s < t'} \text{size}(f_{s}) \cdot y_{t,s} \right) + N \sum_{f \in F} z_{f} \\
\text{s.t.} & \quad \forall t, \forall Q \in Q(t) : \left( \sum_{s \in Q} \min(|x_{s}| + |z_{f_{s}}|, 1) \right) + |z_{f_{t}}| \geq 1 \quad (I) \\
& \quad \forall t, t \leq s < t' : \left| y_{t,s} \right| + |x_{t}| + |z_{f_{s}}| \geq 1 \quad (II)
\end{align*}
$$

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 $\frac{1}{\text{cost}(f_{t})}$, raises $y_{t,s}$ at rate $\frac{1}{\text{size}(f_{t})} \lambda$, 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 $\frac{1}{\text{cost}(f_{s})}$.

**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 a 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 $\alpha$-competitive algorithm ALG$_{SR}$ for ski-rental, there is an $\alpha$-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 \textit{rent-or-evict} problem as follows. There is a file $f$ in the cache. The cost of keeping a file in the cache is $\lambda \text{size}(f)$ per time step and the cost of evicting it is $\text{cost}(f)$. 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 $\lambda \text{size}(f)$ and the cost of buying skis for the season is $\text{cost}(f)$. 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 $\alpha$-competitive algorithm for ski-rental, the algorithm $\text{ALG}_\infty$ for rental caching with infinite cache does the following. For a request to file $f_t$ at time $t$, $\text{ALG}_\infty$ brings the file into the cache. Starting at the next time step, it simulates $\text{ALG}_{SR}$ on $f_t$ to decide how long to keep the file in the current phase. If $\text{ALG}_{SR}$ buys $f$ at any time step during the phase, $\text{ALG}_\infty$ evicts it from the cache at the beginning of that step. The total rental cost of $\text{ALG}_\infty$ is same as the total rental cost of $\text{ALG}_{SR}$ and the total eviction cost of $\text{ALG}_\infty$ is equal to the total cost of buying for $\text{ALG}_{SR}$.

Let $\text{OPT}_{SR}$ be the optimal cost of the ski-rental instance. In a phase, $\text{ALG}_\infty$ cost is at most $\lambda \text{size}(f)+\alpha \cdot \text{OPT}_{SR}$ and the optimal cost is $\text{OPT}_\infty = \lambda \text{size}(f)+\text{OPT}_{SR}$. The the competitive ratio of this algorithm is at most $\frac{\lambda \text{size}(f)+\alpha \cdot \text{OPT}_{SR}}{\lambda \text{size}(f)+\text{OPT}_{SR}}$. Since $\alpha > 1$, the competitive ratio is at most $\alpha$. \hfill \square
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 ALG_{SR} is an \(\alpha\)-competitive optimal algorithm for ski-rental, then there is an \(\alpha\)-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 ALG_{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 \(\alpha\)-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 $\frac{\gamma \text{size}(f_t)}{\text{size}(f_t)}$. Note that $Rent-Zap-Caching-CP_1$ and $Rent-Zap-Caching-CP$ are the same algorithm.

We define $\eta = \min_t \frac{\lambda \text{size}(f_t)}{\text{cost}(f_t)} k$. 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 $\frac{\text{size}(f_t) \lambda}{\text{cost}(f_t)} \gamma$.

**Proof.** Whenever $Rent-Zap-Caching-CP_\gamma$ considers a rent-evict-zap constraint, it raises $x_t$ at rate $\frac{1}{\text{cost}(f_t)}$ and $y_{t,s}$ at rate $\frac{\gamma \text{size}(f_t)}{\text{size}(f_t)}$. Thus, when $y_{t,s}$ goes from 0 to 1, $x_t$ increases by $\frac{\text{size}(f_t) \lambda}{\text{cost}(f_t)}$. 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. $\square$

**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 $f$, $\text{size}(f) \geq 1$. Therefore, $\text{size}(f) \lambda \geq \frac{1}{k}$.

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. Let $d = s - t$. Claim 12 implies that at time $s$ the value of $x_t$ is $\frac{\text{size}(f_t) \lambda}{\text{cost}(f_t)} d$. This value is 1 in at most $d = k$
steps. Thus, the file is evicted in at most \( k \) steps. Note that if \( z_{ft} \) 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. \( \eta \) 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 \( \eta \) gives \( \eta = \min_t \left( \frac{\lambda \cdot \text{size}(f_t)}{\text{cost}(f_t)} \right) k \). Now consider the following cases.

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

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

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

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

(a) For paging, \( \text{cost}(f_t) = \text{size}(f_t) = 1 \). Thus, \( \eta = 1 \).

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

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

Thus, \( \eta \) is at least 1 in all three cases. \( \square \)

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:

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

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, \( \phi \) decreases at the rate \( \min(1, \gamma) \).

Thus, the algorithm maintains the invariant \( \text{ALG}/(2 + \gamma) + \phi/(\min(1, \gamma)) \leq \text{OPT} \). (It is true initially, because \( \text{ALG} = 0 \) and \( \phi = \text{OPT} \).) Since, \( \phi \geq 0 \), this implies that \( \text{ALG} \leq \frac{2 + \gamma}{\min(1, \gamma)} \text{OPT} \). Also, \( \gamma = k\lambda \leq 1 \). So, \( \text{ALG} \leq (1 + \frac{2}{k\lambda}) \text{OPT} \). \( \square \)
**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 \(\phi\) 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 \(\phi \geq 0\), \(\text{ALG} \leq (1 + \frac{1}{k\lambda})\text{OPT}\). \(\square\)

### 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 \(\alpha\)-competitive algorithm \(\text{ALG}_{SR}\) for ski-rental, and a \(\beta\)-competitive algorithm \(\text{ALG}_C\) for file caching with (or without) zapping, then there is an \((\alpha + \beta)\)-competitive algorithm for rental caching with (or, respectively without) zapping.

*If \(\text{ALG}_C\) and \(\text{ALG}_{SR}\) are both deterministic, \(\text{ALG}_{RM}\) is a deterministic online algorithm, otherwise it is a randomized online algorithm.*

We present the algorithm \(\text{ALG}_{RM}\). If \(\text{ALG}_C\) is an algorithm for file caching, \(\text{ALG}_{RM}\) is an algorithm for rental caching. If \(\text{ALG}_C\) is an algorithm for caching with zapping, then \(\text{ALG}_{RM}\) is an algorithm for rental caching with zapping.

\(\text{ALG}_{RM}\) uses \(\text{ALG}_{SR}\) and \(\text{ALG}_C\) to generate a solution for rental caching. On an input sequence \(\sigma\) and cache size \(k\), \(\text{ALG}_{RM}\) does the following. It simulates \(\text{ALG}_C\) on the input sequence \(\sigma\) and cache \(C_1\) of size \(k\). In parallel, it simulates \(\text{ALG}_\infty\) on the request sequence \(\sigma\) and cache \(C_2\) of infinite size. \(\text{ALG}_\infty\) in turn simulates \(\text{ALG}_{SR}\) on
each request, as described in Section 3.3.2. At any time, the cache of $\text{ALG}_{\text{RM}}$ contains the intersection of the files present in caches $C_1$ and $C_2$. When ALG$_C$ zaps a file, ALG$_{\text{RM}}$ zaps it.

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

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

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

**Claim 22.** $E[\text{ALG}_{\text{RM}}] \leq E[\text{ALG}_{\text{SR}}] + E[\text{ALG}_C]$

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

Also, $E[\text{ALG}_{\text{SR}}] \leq \alpha \cdot \text{OPT}_{\text{SR}} \leq \alpha \cdot \text{OPT}$, and $E[\text{ALG}_C] \leq \beta \cdot \text{OPT}_C \leq \beta \cdot \text{OPT}$, where $\text{OPT}_{\text{SR}}$ denotes the optimal cost for rental caching with infinite cache, $\text{OPT}_C$ denotes the optimal cost for caching (rental caching with zapping), and OPT denotes the optimal cost for rental caching (rental caching with zapping). So, $E[\text{ALG}_{\text{RM}}] \leq (\alpha + \beta)\text{OPT}$, and hence, ALG$_{\text{RM}}$ 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) $\frac{k+1}{1+k\lambda}$ for any $\lambda$.

**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, · · · , 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 $\lambda$ 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 $\frac{k + k\lambda}{1 + k^2 H_k \lambda}$.

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

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

**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) \(F_1\) that does not zap any files, and (b) \(F_2\) 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 $\lceil \frac{Z_j}{k} \rceil$.

$$\mathcal{F}_1 = (k + \lceil \frac{Z_1-k}{k} \rceil) + \left( \sum_{j=2}^{T} \lceil \frac{Z_j}{k} \rceil \right) \leq (k + \frac{Z_1}{k}) + \left( \sum_{j=2}^{T} \frac{Z_j}{k} + 1 \right) = k + T - 1 + \sum_{j=1}^{T} \frac{Z_j}{k}$$

$\mathcal{F}_2$ 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

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

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

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

\[\square\]
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 \( P = \{ p_1, p_2, \ldots, p_n \} \) on \([n]\) and an encoding alphabet \( \Sigma \), find a minimum cost prefix-free code (or prefix code) over \( \Sigma \). A prefix code is one where no codeword is a prefix of any other codeword. The cost of a prefix code \( \chi \) is given by \( \sum_{i=1}^{n} p_i |\chi_i| \), where \( \chi_i \) is the codeword assigned to \( i \) and \( |\chi_i| \) is the length of \( \chi_i \). 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 may be 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 $\alpha$-competitive if, for every input $\sigma$, $ALG(\sigma) \leq \alpha \cdot OPT(\sigma)$. We say an algorithm ALG is $\alpha$-competitive asymptotically if there exists a constant $c$ such that, for every input $\sigma$, $ALG(\sigma) \leq \alpha \cdot OPT(\sigma) + c$, or if $ALG(\sigma) \leq \alpha \cdot OPT(\sigma) + o(OPT(\sigma))$.

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 $P = \{p_1, p_2, p_3, \ldots, p_n\}$ on $[n]$ and $n$ slots (named 1 through $n$) with costs $\{c_1, c_2, c_3, \ldots, c_n\}$ (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 cor-
responds 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 \texttt{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](image)

Theorem 29. \textit{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 $\epsilon$ 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 \(\chi_i\), then GREEDY-HUFFMAN instead assigns the word \(\chi_i' = \text{pair}(\text{enc}(|\chi_i|))01\chi_i\), where \(|\chi_i|\) is the length of \(\chi_i\), \text{enc}(l)\) is the binary encoding of the integer \(l\), and \text{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, \(\text{OPT} \geq \mathcal{H}\).

Claim 30. \(\text{GC} \leq \text{OPT}\)

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

\[
\text{GC} = E\left[\sum_i p_i \lfloor \log_2 n_i \rfloor\right] \leq E\left[\sum_i p_i \log_2 n_i\right] \leq \sum_i p_i \log_2 E[n_i]
\]

by concavity of the logarithm function.

For \(j \neq i\), the probability that the \(j_{th}\) symbol appears before the \(i_{th}\) symbol is...
\[ p_j/(p_j + p_i) < p_j/p_i, \text{ so } E[n_i] \leq 1 + \sum_{j \neq i} \frac{p_j}{p_i} = 1 + (1-p_i)/p_i = 1/p_i. \] Thus

\[ \text{GC} \leq \sum_i p_i \log_2 \frac{1}{p_i} = H. \]

\[ \square \]

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

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

\[ \text{GH} \leq \sum_i p_i [\ell_i + 4 + 2 \log_2(\ell_i + 1)] \]

\[ = \sum_i p_i \ell_i + 4 + 2 \log_2 \sum_i p_i (\ell_i + 1) \]

\[ = \text{GC} + 4 + 2 \log_2(\text{GC} + 1) \]

\[ \leq \text{OPT} + 4 + \log_2(\text{OPT} + 1). \]

\[ \square \]

**Theorem 32.** \textsc{Greedy-Huffman} is 7-competitive.

**Proof.** Follows from Theorem 31 and \( z + 4 + 2 \log_2(z + 1) < 7z \) for \( z \geq 1 \). \[ \square \]

4.4 \textbf{Greedy-Coding is optimal for slot assignment}

In this section, we show that \textsc{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 \( \text{OPT} = \sum_{i \in [n]} p_i c_{n-i} \).
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: $\{\textit{Nature}, \textit{Algorithm}\}$. *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 $\Pi$, 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 $\text{OPT}(P) = \text{OPT}(P')$. Thus, $A'$ is $c$-competitive:

$$A'(P') = E_P[A(P)]$$

$$\leq E_P[c \cdot \text{OPT}(P)]$$

$$= E_P[c \cdot \text{OPT}(P')]$$

$$= c \cdot \text{OPT}(P')$$

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 $\Pi$ 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. \( \square \)

**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 $\text{cost}(N(v))$ 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 $C = \sum_{v \in E_M} P(v)\text{cost}(N(v))$. There are infinitely many nodes in $E_M$, so there might not be one with minimum $\text{cost}(N(v))$, 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.

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 $g(M) = \{g_1(M), g_2(M), \ldots, g_n(M)\}$, 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 $\phi$. 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 $h$. 83
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

\[
\sum_i \sum_{j \neq i} p_ip_jp_j(c_k - c_h)g_m(\phi)
\]
because the probability of choosing item \( i \) first, then item \( j \neq i \), is \( p_ip_j \), 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.** \textsc{Greedy-Coding} is optimal for slot assignment.

**Proof.** Claims 36, 37, and 38 together imply that \textsc{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


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


Appendix A

Packing and covering run times

In Table A.1 and Table A.2, \( P_\epsilon \) and \( P_{OPT} \) are the times taken by primal-simplex for computing approximate and optimal solutions, respectively. Similar, \( D_\epsilon \) and \( D_{OPT} \) are the times for dual-simplex, and \( B_\epsilon \) and \( B_{OPT} \) are the times for barrier. \( F_\epsilon \) 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.

<table>
<thead>
<tr>
<th></th>
<th>r</th>
<th>c</th>
<th>d</th>
<th>( P_\epsilon )</th>
<th>( P_{OPT} )</th>
<th>( D_\epsilon )</th>
<th>( D_{OPT} )</th>
<th>( B_\epsilon )</th>
<th>( B_{OPT} )</th>
<th>( F_\epsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>1548</td>
<td>256</td>
<td>0.0078</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>569</td>
</tr>
<tr>
<td>D</td>
<td>37414</td>
<td>1728</td>
<td>0.0012</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>118</td>
</tr>
<tr>
<td>D</td>
<td>1972</td>
<td>276</td>
<td>0.0072</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>87</td>
</tr>
<tr>
<td>D</td>
<td>7082</td>
<td>422</td>
<td>0.0047</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>277</td>
</tr>
<tr>
<td>D</td>
<td>17376</td>
<td>850</td>
<td>0.0024</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>57</td>
</tr>
<tr>
<td>D</td>
<td>1624</td>
<td>174</td>
<td>0.0115</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>D</td>
<td>7850</td>
<td>394</td>
<td>0.0051</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>709</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>r</td>
<td>c</td>
<td>d</td>
<td>$P_e$</td>
<td>$P_{OPT}$</td>
<td>$D_e$</td>
<td>$D_{OPT}$</td>
<td>$B_e$</td>
<td>$B_{OPT}$</td>
<td>$F_e$</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>7832</td>
<td>368</td>
<td>0.0054</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>23308</td>
<td>992</td>
<td>0.0020</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>27938</td>
<td>1242</td>
<td>0.0016</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1016</td>
<td>160</td>
<td>0.0125</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>142</td>
<td>46</td>
<td>0.0435</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3481</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>8200</td>
<td>422</td>
<td>0.0047</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>275</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1160</td>
<td>72</td>
<td>0.0278</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>7892</td>
<td>370</td>
<td>0.0054</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>277</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>640</td>
<td>50</td>
<td>0.0400</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>214</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>7946</td>
<td>372</td>
<td>0.0054</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>472</td>
<td>94</td>
<td>0.0213</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1204</td>
<td>148</td>
<td>0.0135</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>3688</td>
<td>4000</td>
<td>0.0005</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>827</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>40</td>
<td>22</td>
<td>0.0909</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>311</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>61476</td>
<td>40000</td>
<td>0.0001</td>
<td>4</td>
<td>20</td>
<td>6</td>
<td>7</td>
<td>97</td>
<td>160</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>78025</td>
<td>40000</td>
<td>0.0001</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>490000</td>
<td>2000</td>
<td>0.0010</td>
<td>129</td>
<td>1684</td>
<td>18</td>
<td>28</td>
<td>176</td>
<td>921</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1472</td>
<td>250</td>
<td>0.0160</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>38190</td>
<td>770</td>
<td>0.0026</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>55794</td>
<td>500</td>
<td>0.0080</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>2912</td>
<td>128</td>
<td>0.0156</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>27843</td>
<td>595</td>
<td>0.0034</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>14</td>
<td>253</td>
</tr>
</tbody>
</table>
Table A.1 – continued from previous page

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>c</td>
<td>d</td>
<td>$P_t$</td>
<td>$P_{OPT}$</td>
<td>$D_t$</td>
<td>$D_{OPT}$</td>
<td>$B_t$</td>
<td>$B_{OPT}$</td>
<td>$F_t$</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>320069</td>
<td>108000</td>
<td>0.0000</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>18479</td>
</tr>
<tr>
<td>D</td>
<td>2552</td>
<td>240</td>
<td>0.0083</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>215</td>
</tr>
<tr>
<td>D</td>
<td>19606</td>
<td>900</td>
<td>0.0022</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>142</td>
<td>238</td>
<td>22</td>
</tr>
<tr>
<td>D</td>
<td>43266</td>
<td>600</td>
<td>0.0033</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>25</td>
<td>3812</td>
</tr>
<tr>
<td>D</td>
<td>16744</td>
<td>392</td>
<td>0.0051</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>20</td>
<td>8745</td>
</tr>
<tr>
<td>S</td>
<td>126164</td>
<td>1534</td>
<td>0.0013</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>13</td>
<td>86</td>
<td>1915</td>
</tr>
<tr>
<td>S</td>
<td>126083</td>
<td>1534</td>
<td>0.0013</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>17</td>
<td>82</td>
<td>1903</td>
</tr>
<tr>
<td>D</td>
<td>42750</td>
<td>600</td>
<td>0.0033</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>36</td>
<td>32970</td>
</tr>
<tr>
<td>S</td>
<td>127012</td>
<td>1534</td>
<td>0.0013</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>77</td>
<td>1928</td>
</tr>
<tr>
<td>S</td>
<td>126556</td>
<td>1534</td>
<td>0.0013</td>
<td>23</td>
<td>24</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>77</td>
<td>1934</td>
</tr>
<tr>
<td>S</td>
<td>110039</td>
<td>1400</td>
<td>0.0014</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>73</td>
<td>1598</td>
</tr>
<tr>
<td>S</td>
<td>109677</td>
<td>1400</td>
<td>0.0014</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>68</td>
<td>1604</td>
</tr>
<tr>
<td>S</td>
<td>109602</td>
<td>1400</td>
<td>0.0014</td>
<td>16</td>
<td>18</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>61</td>
<td>1606</td>
</tr>
<tr>
<td>S</td>
<td>125983</td>
<td>1534</td>
<td>0.0013</td>
<td>22</td>
<td>23</td>
<td>1</td>
<td>2</td>
<td>19</td>
<td>85</td>
<td>1908</td>
</tr>
<tr>
<td>S</td>
<td>109380</td>
<td>1400</td>
<td>0.0014</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>11</td>
<td>70</td>
<td>1593</td>
</tr>
<tr>
<td>S</td>
<td>109402</td>
<td>1400</td>
<td>0.0014</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>24</td>
<td>72</td>
<td>1600</td>
</tr>
<tr>
<td>S</td>
<td>17832</td>
<td>450</td>
<td>0.0044</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>14</td>
<td>147</td>
</tr>
<tr>
<td>S</td>
<td>17828</td>
<td>450</td>
<td>0.0044</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>12</td>
<td>147</td>
</tr>
<tr>
<td>S</td>
<td>17810</td>
<td>450</td>
<td>0.0044</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>147</td>
</tr>
<tr>
<td>D</td>
<td>16338</td>
<td>900</td>
<td>0.0022</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>33</td>
<td>55</td>
<td>29</td>
</tr>
<tr>
<td>S</td>
<td>94290</td>
<td>1272</td>
<td>0.0016</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>51</td>
<td>1323</td>
</tr>
<tr>
<td>S</td>
<td>27932</td>
<td>595</td>
<td>0.0034</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>14</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>c</td>
<td>d</td>
<td>$P_t$</td>
<td>$P_{OPT}$</td>
<td>$D_t$</td>
<td>$D_{OPT}$</td>
<td>$B_t$</td>
<td>$B_{OPT}$</td>
<td>$F_t$</td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
<td>------</td>
<td>------</td>
<td>-------</td>
<td>-----------</td>
<td>-------</td>
<td>-----------</td>
<td>------</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>D</td>
<td>10384</td>
<td>288</td>
<td>0.0069</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>88</td>
</tr>
<tr>
<td>S</td>
<td>80036</td>
<td>1150</td>
<td>0.0017</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>33</td>
<td>1064</td>
</tr>
<tr>
<td>S</td>
<td>94227</td>
<td>1272</td>
<td>0.0016</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>49</td>
<td>1326</td>
</tr>
<tr>
<td>D</td>
<td>242550</td>
<td>1000</td>
<td>0.0040</td>
<td>88</td>
<td>244</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>126</td>
<td>366</td>
</tr>
<tr>
<td>S</td>
<td>94128</td>
<td>1272</td>
<td>0.0016</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>52</td>
<td>1309</td>
</tr>
<tr>
<td>S</td>
<td>80073</td>
<td>1150</td>
<td>0.0017</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>45</td>
<td>1065</td>
</tr>
<tr>
<td>S</td>
<td>94309</td>
<td>1272</td>
<td>0.0016</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>54</td>
<td>1316</td>
</tr>
<tr>
<td>S</td>
<td>94228</td>
<td>1272</td>
<td>0.0016</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>47</td>
<td>1314</td>
</tr>
<tr>
<td>S</td>
<td>81069</td>
<td>1150</td>
<td>0.0017</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>57</td>
<td>134</td>
<td>1085</td>
</tr>
<tr>
<td>S</td>
<td>80259</td>
<td>1150</td>
<td>0.0017</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>40</td>
<td>1070</td>
</tr>
<tr>
<td>S</td>
<td>80852</td>
<td>1150</td>
<td>0.0017</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>39</td>
<td>1083</td>
</tr>
<tr>
<td>S</td>
<td>41096</td>
<td>760</td>
<td>0.0026</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>24</td>
<td>426</td>
</tr>
<tr>
<td>S</td>
<td>41264</td>
<td>760</td>
<td>0.0026</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>35</td>
<td>428</td>
</tr>
<tr>
<td>S</td>
<td>41315</td>
<td>760</td>
<td>0.0026</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>35</td>
<td>431</td>
</tr>
<tr>
<td>S</td>
<td>58580</td>
<td>945</td>
<td>0.0021</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>61</td>
<td>704</td>
</tr>
<tr>
<td>S</td>
<td>58625</td>
<td>945</td>
<td>0.0021</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>23</td>
<td>80</td>
<td>707</td>
</tr>
<tr>
<td>S</td>
<td>59187</td>
<td>945</td>
<td>0.0021</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>31</td>
<td>62</td>
<td>709</td>
</tr>
<tr>
<td>S</td>
<td>58246</td>
<td>945</td>
<td>0.0021</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>24</td>
<td>69</td>
<td>693</td>
</tr>
<tr>
<td>D</td>
<td>5472</td>
<td>192</td>
<td>0.0104</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>46</td>
</tr>
<tr>
<td>S</td>
<td>58550</td>
<td>945</td>
<td>0.0021</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>22</td>
<td>64</td>
<td>704</td>
</tr>
<tr>
<td>D</td>
<td>499652</td>
<td>2000</td>
<td>0.0020</td>
<td>111</td>
<td>1664</td>
<td>13</td>
<td>18</td>
<td>21</td>
<td>535</td>
<td>26146</td>
</tr>
<tr>
<td>S</td>
<td>41606</td>
<td>760</td>
<td>0.0026</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>34</td>
<td>429</td>
</tr>
</tbody>
</table>
Table A.1 – continued from previous page

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

<table>
<thead>
<tr>
<th>r</th>
<th>c</th>
<th>d</th>
<th>$P_\epsilon$</th>
<th>$P_{OPT}$</th>
<th>$D_\epsilon$</th>
<th>$D_{OPT}$</th>
<th>$B_\epsilon$</th>
<th>$B_{OPT}$</th>
<th>$F_\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>224874</td>
<td>1000</td>
<td>0.0040</td>
<td>42</td>
<td>121</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>94</td>
</tr>
<tr>
<td>D</td>
<td>493416</td>
<td>2000</td>
<td>0.0010</td>
<td>135</td>
<td>1585</td>
<td>14</td>
<td>19</td>
<td>740</td>
<td>2324</td>
</tr>
<tr>
<td>D</td>
<td>125248</td>
<td>1000</td>
<td>0.0040</td>
<td>9</td>
<td>21</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>D</td>
<td>34686</td>
<td>900</td>
<td>0.0022</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>25</td>
<td>45</td>
</tr>
<tr>
<td>D</td>
<td>7110</td>
<td>1000</td>
<td>0.0040</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>6436</td>
<td>500</td>
<td>0.0080</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>5880</td>
<td>200</td>
<td>0.0100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>1510</td>
<td>190</td>
<td>0.0105</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>16526</td>
<td>900</td>
<td>0.0022</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>4224</td>
<td>162</td>
<td>0.0123</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>13312</td>
<td>338</td>
<td>0.0059</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>12</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>r</th>
<th>c</th>
<th>d</th>
<th>$P_\epsilon$</th>
<th>$P_{OPT}$</th>
<th>$D_\epsilon$</th>
<th>$D_{OPT}$</th>
<th>$B_\epsilon$</th>
<th>$B_{OPT}$</th>
<th>$F_\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>13270</td>
<td>14287</td>
<td>0.4000</td>
<td>3173</td>
<td>-</td>
<td>7352</td>
<td>-</td>
<td>2563</td>
<td>-</td>
</tr>
<tr>
<td>R</td>
<td>13270</td>
<td>14287</td>
<td>0.1000</td>
<td>1544</td>
<td>-</td>
<td>1782</td>
<td>-</td>
<td>933</td>
<td>-</td>
</tr>
<tr>
<td>R</td>
<td>12270</td>
<td>12687</td>
<td>0.0009</td>
<td>1021</td>
<td>-</td>
<td>6322</td>
<td>-</td>
<td>427</td>
<td>-</td>
</tr>
<tr>
<td>R</td>
<td>11270</td>
<td>11087</td>
<td>0.4000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1118</td>
<td>-</td>
</tr>
<tr>
<td>R</td>
<td>12270</td>
<td>12687</td>
<td>0.4000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1742</td>
<td>-</td>
</tr>
<tr>
<td>R</td>
<td>9270</td>
<td>7887</td>
<td>0.1000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T</td>
<td>178030</td>
<td>9025</td>
<td>0.0086</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>r</td>
<td>c</td>
<td>d</td>
<td>$P_c$</td>
<td>$P_{OPT}$</td>
<td>$D_c$</td>
<td>$D_{OPT}$</td>
<td>$B_c$</td>
<td>$B_{OPT}$</td>
<td>$F_c$</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-----------</td>
<td>-------</td>
<td>-----------</td>
<td>-------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>T</td>
<td>353079</td>
<td>9300</td>
<td>0.0088</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>12</td>
</tr>
<tr>
<td>T</td>
<td>320779</td>
<td>7600</td>
<td>0.0098</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9</td>
</tr>
<tr>
<td>T</td>
<td>490076</td>
<td>16800</td>
<td>0.0065</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1130</td>
<td>30902</td>
<td>24</td>
</tr>
<tr>
<td>R</td>
<td>10270</td>
<td>9487</td>
<td>0.4000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1122</td>
<td>589251</td>
<td>919</td>
</tr>
<tr>
<td>R</td>
<td>9270</td>
<td>7887</td>
<td>0.4002</td>
<td>5024</td>
<td>29137</td>
<td>4748</td>
<td>26971</td>
<td>779</td>
<td>58843</td>
</tr>
<tr>
<td>R</td>
<td>12270</td>
<td>12687</td>
<td>0.1000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>641</td>
<td>68751</td>
<td>1024</td>
</tr>
<tr>
<td>R</td>
<td>6325</td>
<td>7145</td>
<td>0.4000</td>
<td>71810</td>
<td>71816</td>
<td>50370</td>
<td>57475</td>
<td>314</td>
<td>15661</td>
</tr>
<tr>
<td>R</td>
<td>5725</td>
<td>6245</td>
<td>0.4001</td>
<td>37989</td>
<td>37992</td>
<td>28238</td>
<td>31722</td>
<td>227</td>
<td>11990</td>
</tr>
<tr>
<td>R</td>
<td>3925</td>
<td>3545</td>
<td>0.1001</td>
<td>547</td>
<td>1346</td>
<td>4737</td>
<td>5082</td>
<td>13</td>
<td>124</td>
</tr>
<tr>
<td>T</td>
<td>178030</td>
<td>9025</td>
<td>0.0086</td>
<td>5</td>
<td>9</td>
<td>0</td>
<td>9</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>T</td>
<td>347404</td>
<td>9025</td>
<td>0.0090</td>
<td>8</td>
<td>18</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>R</td>
<td>11270</td>
<td>11087</td>
<td>0.1000</td>
<td>965205</td>
<td>965250</td>
<td>476371</td>
<td>1424136</td>
<td>428</td>
<td>428138</td>
</tr>
<tr>
<td>R</td>
<td>5125</td>
<td>5345</td>
<td>0.4001</td>
<td>32185</td>
<td>32186</td>
<td>24773</td>
<td>76996</td>
<td>157</td>
<td>5251</td>
</tr>
<tr>
<td>R</td>
<td>10270</td>
<td>9487</td>
<td>0.1000</td>
<td>219499</td>
<td>219511</td>
<td>324895</td>
<td>527417</td>
<td>315</td>
<td>29010</td>
</tr>
<tr>
<td>R</td>
<td>4525</td>
<td>4445</td>
<td>0.4001</td>
<td>1450</td>
<td>11589</td>
<td>1854</td>
<td>2902</td>
<td>116</td>
<td>1830</td>
</tr>
<tr>
<td>R</td>
<td>6325</td>
<td>7145</td>
<td>0.0017</td>
<td>335</td>
<td>488</td>
<td>497</td>
<td>636</td>
<td>43</td>
<td>100</td>
</tr>
<tr>
<td>R</td>
<td>3325</td>
<td>2645</td>
<td>0.0999</td>
<td>227</td>
<td>493</td>
<td>2530</td>
<td>3752</td>
<td>5</td>
<td>39</td>
</tr>
<tr>
<td>R</td>
<td>5725</td>
<td>6245</td>
<td>0.0019</td>
<td>303</td>
<td>420</td>
<td>338</td>
<td>434</td>
<td>32</td>
<td>77</td>
</tr>
<tr>
<td>R</td>
<td>5125</td>
<td>5345</td>
<td>0.0022</td>
<td>285</td>
<td>374</td>
<td>282</td>
<td>346</td>
<td>23</td>
<td>56</td>
</tr>
<tr>
<td>T</td>
<td>254099</td>
<td>6230</td>
<td>0.0120</td>
<td>6</td>
<td>34</td>
<td>95</td>
<td>181</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>R</td>
<td>4525</td>
<td>4445</td>
<td>0.0027</td>
<td>183</td>
<td>239</td>
<td>178</td>
<td>216</td>
<td>14</td>
<td>33</td>
</tr>
<tr>
<td>R</td>
<td>3925</td>
<td>3545</td>
<td>0.0034</td>
<td>119</td>
<td>155</td>
<td>169</td>
<td>194</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>r</td>
<td>c</td>
<td>d</td>
<td>( P_e )</td>
<td>( P_{OPT} )</td>
<td>( D_e )</td>
<td>( D_{OPT} )</td>
<td>( B_e )</td>
<td>( B_{OPT} )</td>
<td>( F_e )</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----------</td>
<td>-------------</td>
<td>----------</td>
<td>-------------</td>
<td>----------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td>R</td>
<td>6325</td>
<td>7145</td>
<td>0.1000</td>
<td>15952</td>
<td>21942</td>
<td>43499</td>
<td>90139</td>
<td>84</td>
<td>3524</td>
</tr>
<tr>
<td>R</td>
<td>3925</td>
<td>3545</td>
<td>0.4000</td>
<td>758</td>
<td>4020</td>
<td>4715</td>
<td>4717</td>
<td>40</td>
<td>1818</td>
</tr>
<tr>
<td>R</td>
<td>5725</td>
<td>6245</td>
<td>0.0999</td>
<td>9708</td>
<td>12749</td>
<td>39532</td>
<td>71508</td>
<td>61</td>
<td>2355</td>
</tr>
<tr>
<td>R</td>
<td>5125</td>
<td>5345</td>
<td>0.1000</td>
<td>16096</td>
<td>45224</td>
<td>9616</td>
<td>33145</td>
<td>44</td>
<td>782</td>
</tr>
<tr>
<td>R</td>
<td>3325</td>
<td>2645</td>
<td>0.3999</td>
<td>313</td>
<td>1040</td>
<td>1520</td>
<td>2877</td>
<td>23</td>
<td>119</td>
</tr>
<tr>
<td>R</td>
<td>10270</td>
<td>9487</td>
<td>0.0013</td>
<td>2229</td>
<td>4023</td>
<td>11852</td>
<td>15646</td>
<td>187</td>
<td>465</td>
</tr>
<tr>
<td>R</td>
<td>13270</td>
<td>14287</td>
<td>0.0008</td>
<td>10142</td>
<td>17912</td>
<td>8539</td>
<td>11256</td>
<td>341</td>
<td>998</td>
</tr>
<tr>
<td>R</td>
<td>4525</td>
<td>4445</td>
<td>0.1000</td>
<td>3084</td>
<td>3084</td>
<td>12325</td>
<td>22087</td>
<td>30</td>
<td>174</td>
</tr>
<tr>
<td>R</td>
<td>9270</td>
<td>7887</td>
<td>0.0015</td>
<td>1717</td>
<td>2977</td>
<td>2631</td>
<td>3412</td>
<td>139</td>
<td>351</td>
</tr>
<tr>
<td>R</td>
<td>11270</td>
<td>11087</td>
<td>0.0011</td>
<td>3182</td>
<td>5992</td>
<td>4748</td>
<td>6253</td>
<td>210</td>
<td>619</td>
</tr>
<tr>
<td>T</td>
<td>52409</td>
<td>210</td>
<td>0.0567</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>R</td>
<td>3325</td>
<td>2645</td>
<td>0.0045</td>
<td>55</td>
<td>70</td>
<td>60</td>
<td>73</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

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