The School of Mathematics

THE UNIVERSITY
of EDINBURGH

Solver Benchmarking Using the Julia Language and Optimisation Modelling Language JuMP

by

Ilias Alexandros Parmaksizoglou

Dissertation Presented for the Degree of
MSc in Operational Research with Data Science

August 2021

Supervised by
Dr Emre Alper Yildirim

Abstract

The aim of this dissertation was to conduct an independent benchmarking for open-source and commercial optimisation solvers, using exclusively the Julia programming language and JuMP modelling language. A comprehensive study was conducted that focused exclusively on Linear Programming (LP) solvers that are compatible with the JuMP modelling language. A test set of 35 LP instances of challenging problems was gathered, in order to test overall quality of solvers and performance of the different algorithms that were used, i.e., Simplex Methods or Interior-Point Methods. Five scenarios with different settings were created and each scenario had key differences from the other that corresponded either to the algorithm or the number of threads used during benchmarking. Concrete conclusions were drawn after benchmarking under these five scenarios, regarding solvers' and algorithms' performance in the selected test set. These conclusions can provide insight for users and researchers, with respect to performance of a solver and quality of a solver's Julia interface.

Acknowledgments

Firstly, I would like to thank Dr Emre Alper Yildirim, for the overall impeccable cooperation that we had during the whole summer period that this dissertation took place. His guidance, help and advice, not only were crucial throughout the whole stages of this dissertation, but his commitment and willingness to help, made the whole learning experience fulfilling and rewarding.

I would also like to thank my Personal Tutor Dr. Lars Schewe, for his help throughout this MSc and his readiness to assist me with any matter that was arising, throughout this arguably difficult last year.

Finally, I would like to thank my friends and family for their love and support. They constantly provided me with a much needed emotional boost to continue during both the dissertation period and last year in general.

Own Work Declaration

I declare that this dissertation was written exclusively by myself and that any word, table or figure on the following pages is my own, except when stated otherwise.

Contents

List of Tables

List of Figures

List of Abbreviations

OR	Operational Research
LP	Linear Programming
MILP	Mixed Integer Linear Programming
QP	Quadratic Programming
SOCP	Second Order Cone Programming
SDP	Semi Definite Programming
NLP	Non Linear Programming
QCQP	Quadratically Constrained Quadratic Programming
IPM	Interior-Point Method
OS	Operating System

1 Introduction

Linear programming is widely prevalent within the field of Operational Research, as a significant part of real world practical problems has requirements that can be intuitively expressed through linear relationships. Applications of LP problems can be found in sectors such as transportation, telecommunication, logistics and production planning, with a few examples of LP problems being product mix, blending and network flow problems. However, with the world becoming more and more complicated as technology advances, the size of LP problems increases accordingly. Nowadays, typical LP problems may have hundreds of thousands of variables and constraints. Recent breakthroughs in computing power helps alleviate the problem of increased dimensionality in LP problems, however, even the fastest computer needs state of the art software to first model LP problems and secondly actually solve them.

Software to tackle LP problems are more commonly known as optimisation solvers. These software usually include powerful algorithms, like the Simplex Algorithm or an Interior-Point Method, in order to solve a variety of mathematical models, including LP problems. Due to the aforementioned increase in computing power of the recent years, as well as advances in parallel computing which have facilitated the use of optimisation solvers in a multi-threaded setting, solvers' performance has greatly improved. However, along with this general increase in solver performance, increased competition between optimisation solver manufacturers has also been exhibited.

Optimisation solvers are generally categorized between commercial solvers, i.e., solvers that require an acquired license and open-source solvers, i.e., solvers that can be accessed for free. Commercial solvers tend to be faster, more robust and often provide increased functionality and capabilities that open-source solvers can not. However, open-source solvers can be used completely free of charge and can be quite competitive in performance to commercial solvers. Hence, they can be a suitable alternative for users that do not expect to use a solver for significantly complicated instances.

This highlights that the selection process behind which solver to use is not trivial. Most commercial LP solvers are quite expensive, hence a fare comparison of solvers before acquiring one, would be a great tool for prospective users. Such a process, which is more commonly known as benchmarking, could pinpoint which solver has an edge over another one, whether performance is coupled with robustness for a solver and finally if an open-source solver could be a viable option for the user. During the last two decades there has been significant research in the area of benchmarking, in order to facilitate this selection process.

Apart from prospective users, benchmarking is an important tool for solver manufacturers as well. Through benchmarking, weaknesses of different optimisation solvers can be exposed that were previously unnoticed. This can lead to adjustments in the deployed algorithms and overall significant improvements. Finally, benchmarking also provides a clear initiative to the manufacturer for constant software improvement. This is evident, because the optimisation solver industry, although small, is significantly competitive and benchmarking constantly tries to answer the question of which solver is better.

In order to use an LP solver, the problem needs to be formulated with a modelling language in a computer environment, in a format that the solver can understand. Most commercial solvers have their own modelling language included with the provided software, but open-source solvers tend to rely on separate modelling language packages that are embedded in popular programming languages such as Python, MATLAB and Julia. Some example of this packages are Pyomo (Python) and JuMP (Julia). These modelling languages are also compatible with commercial solvers, hence they are very practical in benchmarking, as selected problems can be formulated once and executed by many different solvers. For the purposes of this dissertation, the JuMP modelling language will be used, which is a package of the Julia Language.

Julia is a high-level, high-performance and dynamic programming language, well suited for scientific computing. Although Julia is a flexible dynamic language, which tend to be slower in performance (e.g Python and MATLAB), Julia does not exhibit such gaps in performance being able to provide speeds comparable to traditional statically-typed languages, such as C and FORTRAN [1]. Julia has seen a significant increase of users in the last decade, with usage jumping by 87 percent to more than 24 million users in 2020. In general, many users are switching to Julia, due to the excellent low level performance it offers in the area of numerical analysis and computational science [2].

JuMP (Julia for Mathematical Programming) is a modeling language for mathematical optimisation embedded in Julia. JuMP takes advantage of advanced features of the Julia programming language to offer unique functionality, while achieving performance on par with commercial modeling tools for standard tasks. It currently supports a vast number of open-source and commercial solvers for a variety of problem classes, including linear, mixed-integer, second-order conic, semidefinite, and nonlinear programming [3].

The goal of this dissertation is to create an intelligible, well structured, fast and completely novel benchmarking framework, purely in Julia. Such a framework will exploit the increased performance that the Julia language offers and the ease of use that the JuMP modelling language exhibits, with respect to reading and formulating models, as well as interacting with optimisation solvers. A variety of the better known commercial (GuRoBi, CPLEX, Xpress, Mosek) and open-source (Clp, HiGHS, Tulip) optimisation solvers that were compatible with JuMP were selected for this framework. All these solvers were interfaced with JuMP and were thoroughly examined, in order to facilitate manipulation of core parameters such as deployed algorithms used in optimisation and number of threads allocated to the solver.

Afterwards, LP test cases which were deemed challenging for the solvers were selected to benchmark solvers under non-trivial instances. Different scenarios were carefully created and key assumptions were stated during the experimental design of this study, in order to create interesting settings to measure solver performance, under alternate circumstances. Such settings could be the performance of a solver using only one thread or using only a specific algorithm. Using these different scenarios, tangible results concerning both performance, accuracy and robustness will be extracted, presented and analyzed, in order to clarify weaknesses and strengths observed across all solvers.

The structure of this dissertation is presented in Figure 1. Firstly, a brief state-of-the-art review will be given, that will also contain the general rules and good practices that should be followed during the benchmarking process. Afterwards, the focus will be redirected to the explanation of the JuMP package and the benefits it provides in the area of benchmarking. The commercial and open-source solvers utilised in the experiments, along with the algorithms deployed by these solvers, will also be described concisely. Then, by taking into account the practices mentioned in the literature review, the methodology regarding the experimental design of the dissertation will be provided in depth, as well as the reasoning behind its step taken during the benchmarking process. An extensive analysis of the scenarios that were selected will follow, coupled with different ways of visualisation and presentation of the results, in order to facilitate extracting concrete conclusions. Finally, the main takeaways of this dissertation as well as suggestions for further research will be presented to the reader.

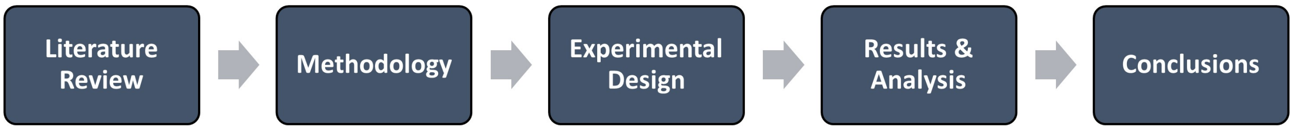

Figure 1: Outline of Dissertation

2 Literature Review

2.1 Solver Benchmarking

In general, benchmarking is a process where at least two products are fairly compared between each other, over a series of relevant performance measures. In the case of solver benchmarking, such products are usually different optimisation algorithms and/or solver modes. After executing instances of each different product, the generated metrics can indicate the better products. In solver benchmarking the predominant metric is the execution time of a solver. However, other measures such as the memory used by a solver and the number of iterations that the algorithm utilized, can also be significant metrics that separate good solvers from great.

When solver benchmarking is done responsibly the benefits can be significant. Strengths of different algorithms can be pinpointed and weaknesses can be exposed and fixed after taking benchmarking results into consideration. Additionally, it can help users decide which product suits their needs. However, solver benchmarking results can also be deceiving. Weaknesses of different implementations may be omitted and assumptions made by researchers can be stated as concrete fact and mislead potential users.

2.1.1 Four-Step Process

A four-step process has been proposed by Beiranvand et al. [4], as a framework for benchmarking optimisation algorithms. This framework provides guidelines for researchers to design experiments in a fair and well structured way. The referenced steps are:

- Clarify the reason for benchmarking
- Select the test set
- Perform the experiments
- Analyse and report the results

A brief analysis of each step will follow.

Step 1 - Clarify the reason for benchmarking

Stating the motivation behind conducting benchmarking experiments may seem like a very straightforward and intuitive step, but is one that is quite frequently omitted by researchers eager to start running instances, without first thinking of the specifics behind their research. Reasons can vary and can significantly alter the experimental design. For example, solver benchmarking can be problem focused, where tests are conducted for finding the solvers that provide the best or the fastest solution for a problem, but can also be solver focused, i.e., the best algorithm of a specific solver needs to be determined. Motivation for benchmarking can also be found under a non-performance focused scenario, such as the testing of a specialized software or programming language as a tool for benchmarking itself. Finally, this step also includes solver selection for benchmarking, a process equally important, as it will determine the technical expertise needed, for actually conducting the experiments in the next steps.

Step 2 - Select the test set

Selecting the test set for benchmarking is a widely researched topic and is definitely not a trivial step, as it can lead to severe flaws in benchmarking, if not done responsibly. Generally, test sets include real world problems, pre-generated problems and randomly-generated problems. The number of problems selected is also an important decision during this step. Including too few problems may not provide a good reflection on the performance of these solvers, while including too many may greatly improve the duration of the experiments. More important than the number of the selected problems is their variety. A test set containing only simple problems may find small gaps in performance, while a test set with very difficult problems where every solver fails, provides little to none information. Specifically to LP problems, variety can be measured by the inclusion of sparse and non-sparse problems and problems with more rows than columns and vice-versa.

Step 3 - Perform the experiments

Actually running the experiments is probably the simplest step, but is arguably the most time-consuming one. Gaining familiarity with all selected solvers and designing a robust framework that both runs all the problems in a serialised manner easily and extracts all the relevant information must be done carefully. Designing a framework that is capable of universally influence solver parameters is crucial, in order to maximize efficiency between different runs and minimise wasted time. The metrics behind performance evaluation must also be decided here. Such metrics revolve mainly between efficiency and reliability. Efficiency refers to the computational effort required to obtain a solution and is typically measured with running time, memory usage and number of fundamental evaluations, while reliability translates to robustness and the ability of a solver to terminate, i.e., solve a problem successfully.

Step 4 - Analyse and report the results

Finally, analysis, discussion and visualisation of the performed experiments is necessary. After some crucial, but basic statistics of the metrics acquired during step 3, presentation of results can be done with different tools such as tables, graphics and very often in solver benchmarking with performance profiles [5]. Performance profiles are cumulative distributions for any performance metric that was calculated, which can act as a tool for solver performance evaluation. Presenting the results properly is quite an important step, as it can both justify and exhibit to the reader, which solver is better.

2.1.2 Existing Research

Benchmarking research can be traced back to the early 1950s, however for the scope of this dissertation, emphasis will be given on research conducted in the last two decades. Mittelman et al. [6] have created the "Decision Tree for Optimisation Software", a repository of information for researchers interested in independent benchmarking, containing useful guidelines, software, test cases and publications. Mittelman's benchmarks are monthly updated and contain a significant number of both commercial and open-source solvers, however, since 2017 the number of benchmarked solvers is smaller [7]. Mittelman's benchmarks contain experiments on LP, MILP, QP , SDP, SOCP and NLP. Notable research, mostly on LP and MILP problems has also been conducted by Meindl et al. (2012) [8], Jablonsky (2015) [9], Anand et al. (2017) [10].

Significant research has also been done in standardizing the proper techniques behind the process of benchmarking, as already stated in [4], as well as in novel ways of reporting experimental results such as performance profiles [5]. Performance profiles have some limitations however, as noted by Gould et al. (2016) [11]. When it comes to reporting simple statistics, relating to solver performance, a lot of metrics have been considered [12], however the most common measure used is the shifted geometric mean [6]. Shifted geometric means are generally accepted by both solver manufacturers and researchers, as the most appropriate metric [13]. The main advantage of the shifted geometric mean, when compared to the arithmetic mean, is that it is less sensitive to very big outliers. Additionally, it is also less sensitive to very small outliers, which is the main problem with geometric means [14]. Alternate representation tools that have been suggested are accuracy profiles by Hare et al. [15], which are mostly used in NLP problems and showcase the proportion of problems that a solver can solve within a specified accuracy τ.

Finally, some focus must be given in the different file formats used in benchmarking. One of the most prevalent formats used, is also one of the oldest, the Mathematical Programming System (MPS) format. MPS is column-oriented format, where all different model components receive different names. MPS is developed for usage with punch cards, so its quite evident that it is neither compact nor easy to read and understand. With the emergence of algebraic modelling languages, such as AMPL and GAMS, its usage has rapidly declined, falling to below 1 % in 2011 (Figure 2). However, due to the fact that MPS is old, it is accepted by mostly all commercial and non-commercial solvers. Hence, it is very convenient to use for researchers in benchmarking, who want to test different solvers that may be unable to read a problem modelled in a more recent format.

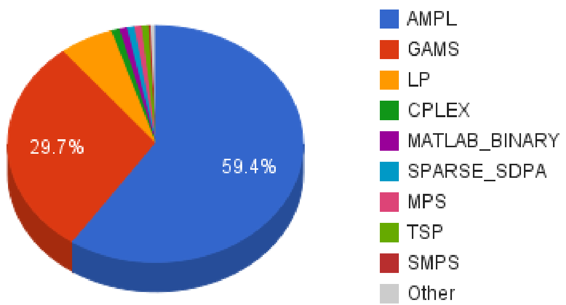

Figure 2: File format usage (2011)
Source: NEOS Server, Argonne National Laboratory

2.2 Julia Language

Julia usage inside the optimisation community has steadily grown in the last decades. Packages like JuMP [3], Optim [16] and JuliaDiff [17] are some examples of this growth. For the purposes of this dissertation, emphasis was given to the JuMP modelling language. JuMP, apart from an excellent modelling language, has also built-in functionalities with the MathOptInterface API [18], an abstraction layer designed to provide a unified interface for mathematical optimisation solvers, making it a powerful tool for solver benchmarking.

2.2.1 JuMP

JuMP is a modelling language for optimisation, built inside Julia. JuMP is comparable to other modelling languages such as GAMS and AMPL, although these are standalone software not built inside a specific programming language. More similar to JuMP are packages for programming languages, such as CVX for MATLAB and Pyomo for Python. Main advantages of JuMP is the access it provides to a significant list of optimisation solvers. Most commercial solvers, such as GuRoBi and CPLEX, have either a complete Julia interface, a wrapper for Julia (usually built around the C API) or direct communication with MathOptInterface. Finally, a huge advantage of JuMP, especially when compared with packages in Python and MATLAB, is its speed. JuMP is fast thanks to Julia's metaprogramming capabilities and provides performance comparable to special-purpose modeling languages such as AMPL.

In general, JuMP is very convenient for anyone that is engaged in classical constraint optimisation problems, due to its user-friendliness, significant speed, solver independence and active community. However, for anyone interested in black-box, derivative-free or unconstrained optimisation and multi-objective optimisation, JuMP is not the right tool, at least for now.

2.2.2 MathOptInterface

JuMP has been recently restructured to include MathOptInterface as its core architecture, replacing its oldest variation, MathOptFormat. Approximately 90 % of the code in JuMP has been re-written during this transition [18]. MathOptInterface was created to improve JuMP and is essentially the way of communication between JuMP and each different solver. Each variable, constraint and objective function created using JuMP is basically transformed to a MathOptInterface object, in order for the solver to interact with it. Capabilities that MathOptInterface provides to JuMP vary from being able to translate to solvers complex second-order cone programming problems to setting simple time limits of execution time to solvers.

For the purposes of this dissertation, MathOptInterface has been a great tool, as for the most part, it provided a way of universal communication between JuMP and each deployed solver. Alternating between different algorithms of solvers, changing the number of threads used in optimisation and also extracting relevant information, such as run time and Simplex iterations that each solver used, was straight-forward to do, due to MathOptInterface.

2.3 Solvers

As already stated, solvers are generally categorised in commercial solvers and open-source solvers. A quick review will follow of all solvers used for the purposes of this dissertation, dependent on this categorisation. The selection of solvers was done purely by their compatibility with JuMP and their occurrence in similar research on benchmarking solvers. In Figure 3, a concise representation of key characteristics of selected solvers is displayed, before further analysis in the following paragraphs.

Solvers						
Solver	Open-Source	Primal Simplex	Dual Simplex	Interior-Point Method	Crossover	Multi-Threading
Clp 1.17	✓	✓	✓	✓	✓	✗
FICO Xpress 8.7	✗	✓	✓	✓	✓	✓
GuRoBi 9.1.2	✗	✓	✓	✓	✓	✓
IBM ILOG CPLEX 20.1	✗	✓	✓	✓	✓	✓
HiGHS 1.0	✓	✗	✓	✓	✗	Linux only
Mosek 9.2.48	✗	✓	✓	✓	✓	✓
Tulip 0.8	✓	✗	✗	✓	✗	✗

Figure 3: Selected solvers' capabilities

2.3.1 Commercial Solvers

FICO Xpress

Xpress was developed in 1983 by Dash Optimisation as a pure LP solver, before being acquired by FICO in 2008. Currently, Xpress is an optimisation solver for linear programming, mixed integer linear programming, convex quadratic programming, quadratically constrained quadratic programming, second-order cone programming and non-linear programming. Xpress supports multi-threaded parallel processing, with the current stable release (8.11) maxing out at 8 threads. It can solve LP problems using the Primal Simplex, Dual Simplex and Barrier Interior-Point method. Xpress can communicate with other standard modeling languages, such as AMPL and GAMS and also has a supported API for MATLAB, C , C++ , Python and Java. Communication with Julia is achieved via a thin wrapper around the C API and MathOptInterface.

GuRoBi

Gurobi was founded in 2008 by Zonghao **Gu**, Edward **Ro**thberg and Robert **Bi**xby. Gu and Rothberg were part of the development team of CPLEX and Bixby was the founder of CPLEX. Gurobi supports linear programming, mixed integer linear programming, convex quadratic programming, quadratically constrained quadratic programming and all their mixed integer counter-parts. Gurobi has a soft-limit of 32 threads, when using multi-threaded algorithms, although even this limit can be superseded. It can solve LP problems using the Primal Simplex, Dual Simplex and Barrier Interior-Point method, but it also has a concurrent method where it can utilize multiple threads to execute multiple algorithms in parallel. It has both object-oriented interfaces for C++ and Java and Matrix-Oriented interfaces for Python MATLAB and R.

IBM ILOG CPLEX

CPLEX was created in 1988 by Robert Bixby, as an optimisation for the Simplex method using the C language. Nowadays, similarly to the previously mentioned solvers, supports linear programming, mixed integer linear programming, convex quadratic programming, and quadratically constrained quadratic programming. CPLEX supports multi-threading and parallel computing, but multi-threading can not be limited to a specific algorithm. Hence, when more than one threads are used the result of an LP problem could be achieved by any of its available algorithms such as the Primal/Dual Simplex or Barrier Interior-Point method, which run concurrently. Interfacing with C++, Java, and C# languages is done with a modeling layer called Concert, while interfacing with Julia and Python is achieved via the C API.

Mosek

Mosek is part of the Danish company Mosek ApS which was established in 1997 by Erling D. Andersen. Mosek is more well suited for solving large scale sparse problems, particular with its Interior-Point-method optimiser, although Mosek also supports Simplex methods. Hence, Mosek is primarily a software package for linear programming, conic quadratic and semi-definite programming, but also has functionalities for mixed integer programming, quadratic programming and non-linear programming. Mosek supports multi-threading and has an interface for C, Java, MATLAB, Python and R.

2.3.2 Open-Source Solvers

Clp

Clp is part of the COIN-OR (Computational Infrastructure for Operations Research) project and was developed in 2004 by John Forrest et al., as an open-source Simplex optimiser written in C++. Since then, Clp has grown and also has an Interior-Point method solver. Clp is purely a linear programming optimiser and can not support more complex problem structures such as mixed-integer programming. Additionally, there is no multi-threaded or concurrent method available as parallel computing is not supported by Clp. This solver is mainly designed to be used as a callable library for different programming languages with available interfaces in Python, Julia, Java etc. Clp is a foundation for a plethora of other solvers in the COIN-OR project such as Cbc , Bcp and SYMPHONY, which are focused in mixed-integer linear programming.

HiGHS

HiGHS is an open-source solver built on the high performance dual revised Simplex implementation and its parallel variant [19], developed by Julian Hall, Ivet Galabova, Qi Huangfu, and Lukas Schork.

Although a Dual Simplex optimiser first, it also has an Interior-Point method solver. HiGHS is both a parallel and a serial solver for large-scale sparse linear programming and mixed-integer programming, that supports multi-threading up to 8 threads under Linux, but not on Windows or MacOS. HiGHS was developed in C++ and has supported interfaces for languages such as C, C#, FORTRAN, Julia and Python.

Tulip

Tulip is an exclusively Interior-Point method open-source solver for linear programming, written in pure Julia. Tulip was created by Miguel F. Anjos, Andrea Lodi, Mathieu Tanneau [20] and it implements the homogeneous primal-dual Interior-Point algorithm with multiple centrality corrections, and therefore handles unbounded and infeasible problems. Tulip does not support crossover and it is not multi threaded at the moment, however external linear algebra libraries may exploit multiple threads. Being built on Julia, the recommended way of interacting with it is through JuMP and MathOptInterface, however a low level API that is currently under development is also available.

2.4 Algorithms

Most solvers for LP problems have either a pivot method, an Interior-Point method or both. The argument behind which method is better is a long and interesting one and this dissertation touches on it a little bit. However, the general consensus is that for the most practical and routine applications of linear programming the gaps in performance are small, but for specific types of LP problems one algorithm may have an edge over the other.

2.4.1 Simplex Methods

The most successful pivot method is arguably the Simplex method. Originally developed in 1947 by George Dantzig, the Simplex method exploits convexity of the feasible region of an LP problem to traverse a vertex to vertex path, from feasible basic solution to feasible basic solution, in order to reach an optimal basic solution. Simplex pivot methods require primal or dual feasibility of the actual basic solutions and work towards feasibility of both. A Primal Simplex method is initiated by a primal feasible basic solution and a Dual Simplex method is initiated by a dual feasible basic solution [21]. Although, the Simplex algorithm has an exponential worst-case complexity, as found by Klee and Minty [22], studies has shown that it has a polynomial-time average-case complexity under various probability distributions [23].

As mentioned in section 2.3, the Simplex method have plenty state-of-the-art, implementations. These software usually exploit, the Simplex method's flexibility to combine it with specialized heuristics and rules, in order to achieve optimised performance. Specifically in the last years, automatised dualisation and the Dual Simplex method have proven very efficient in practice, both for LP problems and for Mixed-Integer Linear Programming problems, as the dual method has been very effective, when used in Branch and Bound for solving LP relaxations of such formulations [21], that are constantly re-optimized after adding an additionally constraint, that Branch and Bound dictates.

2.4.2 Interior-Point Methods

The first polynomial Interior-Point method was created by Narendra Karmarkar in 1984. Karmarkar's algorithm was very efficient in practice and succeeded in test cases were the older Simplex method failed. An Interior-Point method finds an optimal solution by traversing the interior of the feasible region and not traversing vertex to vertex like the Simplex method. In general, an Interior-Point method tries to maintain primal and dual feasibility while working toward complementary slackness. Despite their polynomial complexity, Interior-Point methods would not be reliable without direct methods and techniques of factorization in linear algebra, which have different limitations as the problems increase in size [24]. Hence, Interior-Point methods can not be considered strongly polynomial [25] in practice.

Interior-Point solvers have been very efficient and although Simplex solvers typically compete quite well with them, IPM solvers are consistently more efficient for very-large scale problems, with millions of variables and high degeneracy. Main reason behind this computational success of IPMs is the significant research in efficient sparse matrix technology in the last decades, with symbolic factorization, sparse Cholesky and Bunch–Parlett factorization being indispensable tools for IPMs [21]. This fact has tied future development of IMPs with future advancements in linear algebra [24].

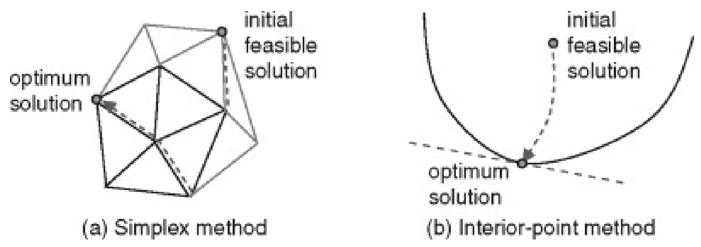
initial feasible solution
optimum solution
(a) Simplex method
initial feasible solution
optimum solution
(b) Interior-point method

Figure 4: Simplex & Interior-Point Method - Graphical Representation

Finally, an important part of an Interior-Point method is the Crossover to a basic solution. Unlike Simplex methods, an Interior-Point method may not automatically find a basic optimal solutions, i.e, a solution that lies on a vertex. Although acquiring a basic solution is not necessarily required, in practice basic solutions are preferred, as they provide a solution where m variables will have nonzero values, with m being the number of constraints, which is a convenient mathematical property.

In general, a non-basic solution found by an Interior-Point method is converted to a basic solution, by the use of Simplex-like "push" iterations, usually with the aid of a Simplex solver. In practice, Crossover may be unimportant and including it to an Interior-Point method increases total solution time [26] of an LP problem significantly. However, most solvers actually have Crossover activated by default as basic solutions tend to be better rounded and are especially useful for re-optimisation purposes.

3 Methodology

The following chapter will follow the framework presented in section 2.1.1 and aim to justify and explain all the decisions and assumptions taken, during the experimental design of this dissertation. As this chapter will include many of the technical aspects of the dissertation, references to external repositories (GitHub) are expected on this chapter.

3.1 Motivation

As previously mentioned, clarifying the reason behind benchmarking is an easy to omit step. Researchers can be anxious to start running computational experiments, without first settling on why they are interested in actually running them. Establishing a clear direction behind each step taken, can help researchers keep their goals clear and concise. The motivation behind the experiments provided in this dissertation is divided in three parts.

Firstly, a significant part of initiating a benchmarking study has been the sheer interest around Julia and more specifically JuMP. Julia has been steadily growing inside the scientific community in the last decade and mentions of its increased speed, flexibility and performance were certainly of interest to this study. Essentially, the possibility of a transition away from more specialised modelling software, e.g. AMPL, or a slower dynamic language based on a programming language, e.g. MATLAB, was deemed important to investigate. This notion was reinforced by the increased growth of Julia and more accurately JuMP, inside the optimisation community. The aim was to create a standardised framework for benchmarking around JuMP that could easily be reused and augmented in the future, irrespective of the problem. This framework would not be particularly judged around its performance and speed, as this would require a more comprehensive study and comparison with other modelling languages, but would need to be user-friendly and convenient as a tool for benchmarking. However, this does not mean that the performance of JuMP was completely not of interest to this study, as some focus was given on reporting the differences in execution time inside the solver and total execution time of the problem in JuMP.

Secondly, there was significant interest in creating a comprehensive study that includes benchmarks from both commercial and open-source solvers. More than 25 % of available solvers in JuMP, were open-source solvers, hence it was deemed appropriate to not exclude them from this process, despite the fact that performance was likely to be slower. Essentially, interest around open-source solvers was more focused on how robust these solvers were and if they were able to terminate within a specified time-limit. The idea was that the time-limits would be strict enough that if an open-source solver was able to terminate, it would be competitive to commercial solvers, despite the fact that the commercial solver would probably be faster. Finally, another aspect behind the solver selection process, would be to create an independent benchmark that includes Xpress and CPLEX, which has been excluded by the Mittelman benchmarks since 2017, at their request.

Finally, focusing this research on LP problems was something that was also decided relatively early-on. This was decided because LP problems are the most prevalent inside the OR community. However, it was necessary the created framework, was useful as a foundation for further studies in different OR problems, such as MILP. In fact, the decision to limit the study to LP problems was both practical and logical. On the practical side, including different types of problems would greatly augment the duration of the experiments and probably result in the exclusion of solvers. On the logical side, limiting the variety of problems, led to a more in-depth look in specific aspects of LP solvers that were considered of interest, such as multi-threading and different algorithms.

3.2 Test Cases Selection

Selecting the test cases, that would be included in the experiments was entwinwed with two decisions. The first decision was related to the quantity of the problems included, while the second decision was related to their variety.

The first part was settled relatively easy. Following the best practices for designing benchmarking experiments [4], it was evident that the selected tests should be more than 20, if the benchmark tested performance of different solvers and more than 100, if the benchmark tested performance of different versions of the solver. Since this study was focusing with the former case, the number of tests selected was deemed to be 35, which is comfortably higher than 20.

To ensure variety of test cases, significant attention was directed to existing research in benchmarking. Since in section 3.1, the decision to limit this study in LP problems was taken, current benchmarking test cases for LP problems were sought. Another aspect that was important to be addressed, before finalising a test set was a pragmatic one. As mentioned in [9], OR problems are usually divided in the following three categories:

- Easy - problems that can be solved with a solver within one hour
- Hard - problems that have been solved, but termination time is unpredictable
- Open - problems that the optimal solution is unknown

For the purposes of this dissertation, with a time-frame smaller than 6 weeks to complete the experiments, including any problem that was not in the easy category was deemed unrealistic. Hence, focus was directed to easy problems, although this does not implies that all solvers were able to solve these problems under one hour, but that at least one was able to.

The test cases selected were picked by Professor Mittelmans's [6] selection of LP problems for benchmarking. Most problems were either part of the most recent benchmark of Professor Mittelman for Simplex solvers or Interior-Point method solvers. Since the Mittelman benchmarks are the "gold standard" for independent benchmarking at the moment, the decision to pick problems from this library was deemed a well justified one. The 35 problems selected are presented in Figure 5 with relevant details such as number of rows, columns and non-zero values.

These problems are accessible for download in the repository of Professor Mittelman and Professor Csaba Meszaros [27], as well as in the official MIPLIB library. Downloading them is easy, as all relevant links are gathered in the Decision Tree for optimisation Software, that Professor Mittelman maintains. Since all problems are quite large, which stands to reason as they are used in benchmarking studies, a compression of these file was needed. These files were compressed using the .emps (extended-mps) format. The procedure to transform the .emps files back to the .mps has been reported in the NETLIB repository, another significant library for LP benchmarking. The procedure included compiling a C file, which was available in the NETLIB library, to create an executable. Then, translating the .emps file to .mps was possible using the executable and simple DOS commands.

Another important aspect of the included instances was that all problems were minimisation problems. This was not something that is explicitly stated on a problem formulated in the MPS format, as this information is generally not included in an .mps file, but is something that was corroborated after studying the repositories, where these problems were stored. Additionally, it is important to notice that all problems had a finite global optimum, i.e., no problem was unbounded and no problem was infeasible. This was not something that was necessarily sought after, but it was a consequence of picking the problems from Mittelman's library. The reason why Mittelman's library only includes finitely feasible problems in unclear, but the main assumption is that it enables easier comparison for matters corresponding to solvers' accuracy.

Finally, the convenience of the test cases being in the MPS format both for benchmarking and usage with JuMP, must also be briefly addressed. Original plans of this dissertation included creating a parser for translating problems from MPS to the JuMP modelling language. However, this was deemed

Problem	**Columns**	**Rows**	**Non-Zeros**
chromaticindex1024-7	73728	67583	270328
cont1	40398	160792	399991
cont11	80396	160792	439989
datt256_lp	262144	11077	1503732
ex10	17680	69608	1179680
fhnw-binschedule0_lp	319319	58085	1373718
fome13	97840	48568	334984
graph40-40_lp	102600	360900	1260900
irish-electricity	61728	104259	538809
L1_sixm1000obs	1426256	3082940	14262560
L1_sixm250obs	428032	986069	4280320
Linf_520c	69004	93326	566193
neos-3025225_lp	69846	91572	9357951
neos-5052403-cygnet	32868	38268	4898304
neos-5251015_lp	136971	486531	1955388
neos	36786	479119	1084461
neos3	6624	512209	1542816
ns1687037	43749	50622	1406739
ns1688926	16587	32768	1712128
nug08-3rd	20448	19728	139008
pds-100	505360	156243	1390539
physiciansched3-3	79555	266227	1062480
qap15	22275	6330	110700
rail4284	1092610	4284	12372358
rmine15_lp	42438	358395	879732
s100	364417	14733	2127672
s250r10	273142	10962	1572104
s82_lp	1690631	87878	7022608
savsched1	328575	295989	1846351
scpm1_lp	500000	5000	6250000
self	7364	960	1148845
square41	62234	40160	13628623
stat96v1	197472	5995	588798
stormG2_1000	1259121	528185	4228817
supportcase10	14770	165684	551152

Figure 5: Selected problem set with problem details

unnecessary, as JuMP's latest release (0.21.9), included functionality to easily transform .mps files to MathOptInterface objects that solvers can easily understand. Conversely, any problem modelled in JuMP modelling language can now easily be saved to the MPS format (if the problems is LP or MILP) for easier access of users in different software and modelling languages. This fact alleviated part of the computational load of this dissertation and made way for creating a more-comprehensive benchmark around LP solvers and algorithms.

3.3 Performed Experiments

In this section, the experimental design of this dissertation will be discussed and the general high-level framework of what constitutes a single "benchmark run" for the purposes of this study. Then, focus will be redirected to the technical aspects of making the experimental design possible, using Julia and JuMP.

Experimental design

In the early phases of this dissertation, including different OR problems such as MILP and QP problems was considered. However, this idea was abandoned relatively soon. Main reasoning of this decision was the flexibility that JuMP provided, in influencing solver parameters. Hence, original plans were abandoned in favour of providing a more comprehensive study of LP problems, where different algorithms and computational barriers could be tested under specifically determined rules.

This led to the importance of creating different scenarios of benchmarking, all relating to the same LP problems. After thoughtful consideration, five cases were created. These were mainly influenced by existing literature and curiosity of the researchers and partly by solver capabilities. The cases selected differed with respect to the algorithm used during benchmarking and/or different parameters set in the solver. The selected cases are:

- Case 1 - Automatic - No Thread Limit Mode
- Case 2 - Primal Simplex - With Thread Limit Mode
- Case 3 - Dual Simplex - With Thread Limit Mode
- Case 4 - Interior-Point Method - With Crossover & Thread Limit Mode
- Case 5 - Interior-Point Method - Without Crossover - With Thread Limit Mode

Case 1 is the most straight-forward and it measures solver performance if no parameter was changed by the user. The idea behind this case is to test solver performance, as the solver's manufacturer intended, with maximum available threads and whichever algorithm the solver selected. Case 2 aims to test the quality of the Primal Simplex algorithm, but with a thread limit of one thread used by each different solver. All the remaining cases were limited to one thread. The idea behind limiting the benchmark to one thread was initially influenced by the fact that not all solvers have mutli-threading capabilities. Hence, to really test the quality of the algorithm implementation and not the effect of more computing power, all remaining experiments were set to one thread. Additionally, even if allowing the use of more threads, the information regarding to how many threads were actually used was not available, but only the maximum available threads that could be used. Hence, the decision to limit the threads was a sensible one. Case 3 and Case 4, similarly to Case 2, intend to test the quality of the Dual Simplex method and standard Interior-Point method (barrier). However, Case 5 diverges a little bit from Case 4, by removing crossover. The decision to see the quality of an algorithm without crossover was also motivated by the fact that some solvers do not have crossover capabilities and including them in Case 4 would be unfair.

As highlighted in the paragraph above, not all solvers could be included in the different cases presented, as not all solvers have the functionality to perform the experiments under the specified scenarios. These limitations only affected the open-source solvers. Most exclusions from different scenarios were because of solver limitations, but some exclusions were made purely because of performance. These were Clp and HiGHS solvers were their Interior-Point Method was not tested, as these solvers are primarily simplex solvers. The solvers that were included in each different case were:

- Case 1 : Clp, Xpress, GuRoBi, HiGHS, CPLEX, Mosek, Tulip
- Case 2 : Clp, Xpress, GuRoBi, CPLEX, Mosek
- Case 3 : Clp, Xpress, GuRoBi, HiGHS, CPLEX, Mosek
- Case 4 : Xpress, GuRoBi, CPLEX, Mosek
- Case 5 : Xpress, GuRoBi, CPLEX, Mosek, Tulip

Additionally, some rules were established concerning the experimental design. The most significant was the inclusion of a universal time limit of solver termination, at 30 minutes. This was a practical decision, as the establishment of a longer time limit would greatly improve the duration of the experiments. Although smaller than the more common time limit of 1 hour in other studies, which is also the threshold that signifies whether a problem is easy or hard, most solvers managed to terminate successfully at the majority of problems with this time limit. Another rule that was ultimately scrapped was the inclusion of universal primal feasibility and dual feasibility tolerances. Although the framework for including this parameter exists in the code [28], a universal tolerance was never put in effect, as it would need a deeper understanding of each algorithm developed by each solver, which was unrealistic given the time-frame of this dissertation.

Some consideration must also be given on the reported metrics. For the pruporses of this dissertation, two metrics were included, solver time and Julia Time. Julia Time is similar to solver time, with Julia Time including solver time. Basically, Julia Time differs because it includes time spent on the solver interacting with MathOptInterace, reading the model and extracting required metrics. Differences between Julia Time and solver time were expected to be unimportant, but was not for some solvers in specific problems.

Finally, after thorough explanation of its parts, what constitutes a benchmark run can be defined. A benchmark run is a complete execution of all the test instances that were described in Section 3.2, using one compatible solver, under one of the described cases. During a benchmark run, all problems are executed serially, using the existing rules that have been set. For each benchmark run, the log of the solver is stored at an external file. After termination of a problem, either due to solution or enforcement of a rule, the reported metrics of interest and result of the problem are also stored at an external file. A benchmark run is only complete after successful completion of all problems in the Test set.

Technical Framework

Firstly, a quick note on the solvers that were used. The version of each solver included in the benchmark was the same as the versions mentioned in Figure 3. The commercial solvers required a separate license and installation of the product, as well as the download of a separate Julia Package. The open-source solvers that were used did not require a separate product installation and the necessary binaries were installed via their Julia package. All packages that related to a solver installation, along with instructions on how to install the solvers were part of the JuMP-dev repository [29] and were installed using the Julia Package manager (Pkg).

Secondly, a brief description of the hardware and software used (not relating to the solvers) for performing the experiments, is given. The experiments were executed on an Asus ROG Strix G513 laptop, with an AMD Ryzen 9 5900HX CPU at 4.3 GHz, with 8 Cores, 16 Threads and 16 GB RAM. The operating system used was Windows 10 Home (Build 19042). Scripting of the relevant code was done with Visual Studio Code and execution of the code was done directly via the Julia REPL. Finally, version-control was used (Git) and all relevant code is uploaded at a public repository [28] on GitHub.
It's clear from the Experimental Design that a significant part of the technical framework corresponds to extracting and influencing parameters via JuMP. These parameters were divided to two different categories. Parameters that could be set or extracted via MathOptInterface across all solvers and parameters that required studying the manuals of each different solver.

MathOptInterface theoretically provides functionality to set a solver time and a thread limit, across different solvers. However, implementing thread limit with MathOptInterface was buggy for some solvers, as stated by different solver manufacturers, hence this parameter was never set via MathOptInterface. MathOptInterface was more useful for extracting metrics from the solvers. Reporting Simplex iterations, barrier iterations, solution time, status of the problem and its objective value was done via MathOptInterface. The only exceptions to that was the Clp solver, where iterations could not be extracted via MathOptInterface and were extracted from the reported logs of the solver and the Tulip solver where, solver time was not available and Julia Time was only used instead. Julia Time was calculated using a simple time function native to the standard Julia library.

Influencing parameters in the solvers was heavily dependent on consulting the manuals of each solver and/or relevant GitHub repositories of the solvers. In general, setting a specific algorithm, number of threads, including crossover or not and influencing tolerances in the solver, required meticulous study of solver manuals. To set a parameter via JuMP the key which corresponds to a parameter was needed, along with the keyword that specified how to influence the parameter. For example, the GuRoBi solver changes the algorithm used with the parameter "Method" and for the Dual Simplex algorithm the keyword is 1. This was the information that was needed to influence the solver with JuMP, using the *set_attribute* function. For easier execution of a benchmark run across different solvers, dictionaries were created that stored information for all solvers, for a specific parameter. For example, a dictionary was created that included all the keywords that changed the algorithm to Dual Simplex, as well as a dictionary that stored all the keys that corresponded to changing the algorithm of the solver. All the dictionaries created, for the seven solvers that were included in this dissertation are available on GitHub [28].

After managing to standardize the process of influencing and extracting parameters using JuMP, which was a tedious process that required significant time reading the manuals of the solvers, actually performing the experiments could commence. There were 27 benchmark runs that were initiated in total, each for the test set of 35 problems described in Section 3.2. In total the experiments, took approximately three and a half weeks to conclude.

3.4 Visualisation & Analysis

Having completed all benchmark runs, its evident that a huge amount of information is stored. Analysis and proper representation of these results is necessary, in order to be able to extract concrete takeaways from the performed experiments. To facilitate the analysis of the results, tools from statistics will be used, as well as graphical tools such as performance profiles and relevant charts. This step is probably the most complicated one and it needs to be handled carefully, as it is easy to clatter the analysis with huge tables and irrelevant charts, that provide little merit in the extraction of information.

Since the experimental design of this dissertation was divided in five different cases, the presentation of the results will be also divided accordingly. It's evident that the tables with the reported results from each different case and solver can not be omitted, both for transparency of the benchmark and also for providing a more in depth look to specific problem performance, for the reader. However, it's true that tables alone provide little insight to the reader, with regards to performance. To remedy that, tables will be accompanied with simple statistics for the performed metrics.

It would be illogical to provided insight, by using averages of solver time, across all problems, as these metrics would obviously be heavily variant. Hence, the scaled and unscaled shifted geometric mean will be used instead, as a way to tackle this problem. The unscaled shifted geometric mean of the n nonnegative numbers $v[1], ...v[n]$ and a nonnegative shift sh, can be computed using the following formula.

$$SGM = \exp(\sum_{i=1}^{n} \frac{\ln(\max(1, v[i] + sh)}{n}) - sh$$

The nonnegative shift used was 10 seconds. To scale the shifted geometric mean, the fraction between the unscaled solver performance against the unscaled performance of the best performing solver was computed. This provided an insight regarding how much slower a solver is, in comparison to the best solver. Finally, the number of problems solved by each solver, during a specific case, was also provided, along the cumulative results table.

As already mentioned, performance profiles were created as a comparison tool for different solvers' performance, under a different scenario. However, performance profiles were also deemed useful for specific solver performance comparison, under different scenarios, e.g., Dual Simplex performance versus Primal Simplex. It's true that performance profiles provide less insight for each solver, as the number of solver grows [11], however they still provide a good insight on the number of problems that a solver can solve successfully, within margins of the best performing solver. All performance profiles generated, used the log_2 scale on the x axis, for better readability of the results.

Additionally, charts that showcased specific solver performance across all problems were created. For each different case, two baselines were computed from the best recorded metric and an average derived from the shifted geometric mean for each problem, regardless of the solver. Then, a the performance of the solver was plotted in a chart along these baselines. This aimed to present the divergence of a specific solver, from the best recorded performance and the average performance.

To generate these plots, tables and statistics, relevant code had to be developed. This was done in Python using Matplotlib and Pandas and not in Julia. The main reasoning behind this, was the more developed packages that Python offered for data analysis and visualisation. Although Julia had similar packages, likes Plots and CSV, the benefits that came with using Python packages that were significantly older and with a bigger community were many, especially when working under a small time-frame. The developed code is also available on this GitHub page [28].

Finally, it's evident that the amount of plots and tables that can be provided is huge. In chapter 4, the ones that were deemed most important will be presented and discussed. For any specific chart that is not included in this dissertation, cloning the repository from GitHub and generating it should be fairly straightforward and fill any potential gaps in the analysis.

4 Results & Analysis

This chapter is broken down in two parts. Firstly, the results of the experiments for each different case will be presented, along with some basic statistics and visualisation tools that will facilitate the extraction of information. Secondly, an extensive analysis of the performance of different algorithms and solvers used in the benchmark will follow.

For each different case presented in section 3.3, the cumulative table of reported solver time of any solver included in this case, will be exhibited. Additionally, the scaled and unscaled shifted geometric mean (shift of 10 seconds) and a report concerning the success rate of each solver in this case will be included. This report will provide information regarding:

- No. of Problems solved to Optimality
- No. of Problems terminated due to Time Limit
- No. of Problems that the solver failed

Each problem that was not solved optimally will contribute with a value of 1800 seconds to the geometric mean, equal to the time limit imposed in this experiments. Examples of solver failures were early terminations, numerical errors, loading errors and solutions that were closed to optimality, but ultimately did not converge with a feasibility tolerance of 10^{-4}.

Additionally, a performance profile that will include all solvers deployed in the examined case, will be presented for each case. More specialised performance profiles will be exhibited in section 4.2. Performance profiles are used as a visualisation tool that aims to provide context, regarding the amount of problems that could be completed successfully within specific fractions of the optimal time achieved, by the best performing solver.

Section 4.1 can not be considered a proper analysis of the experiments on its own, but it providesa thorough examination of the reported results. All results can be validated by the logs of the experiments and all the tables, plots and summary statistics can be generated, by using the code provided on GitHub [28]. Conversely, in section 4.2 a deeper analysis will be attempted, paired with main takeaways regarding solvers' performance, always within the context of the examined test set.

4.1 Examined Cases

4.1.1 Case 1 - Automatic - No Thread Limit Mode

Firstly, the more general case is presented, where no influence to the solvers' parameters is applied. As it is appropriate, all solvers are included in this case with no distinction for number of threads or algorithms used. Additionally, all solvers had crossover activated when using an Interior-Point method, apart from Tulip which does not have crossover functionality.

	Clp	**Xpress**	**GuRoBi**	**CPLEX**	**HiGHS**	**Mosek**	**Tulip**
Unscaled Geometric Mean	343.11	70.96	22.27	44.63	552.07	51.77	381.82
Scaled Geometric Mean	15.41	3.19	1.00	2.00	24.79	2.32	17.15
Optimal	25	33	35	34	21	31	20
Time Limit	6	2	0	0	7	3	6
Fail	4	0	0	1	7	1	9

Table 1: Geometric mean & Success Report for Case 1

It is evident from Table 1 that the best performing solver, both in terms of solver time and success rate, is GuRoBi. Second is CPLEX with a geometric mean a little bit over 2 times bigger than GuRoBi. Close third is Mosek with approximately 2.3 times slower performance than GuRoBi and

Xpress is fourth. Best performing open-source solver is Clp with a geometric mean 16 times greater than GuRobi and 25 problems solved in total out of 35. Last solver, with regards to geometric mean of solver time is HiGHS. However, HiGHS managed to solve more problems than Tulip, with HiGHS solving 21 problems and Tulip solving only 20 problems.

The cumulative table containing solver time reported by each solver is presented in Table 2. All problems that a time limit was imposed are marked with t and all problems that a solver failed are marked with f. Same notation will be followed in tables used for the next cases.

Problems	**Clp**	**Xpress**	**GuRoBi**	**CPLEX**	**HiGHS**	**Mosek**	**Tulip**
chromaticindex1024-7.mps	121	83	5	3	77	9	3
cont1.mps	218	71	20	110	t	55	14
cont11.mps	1299	1383	27	311	1297	f	18
datt256_lp.mps	91	658	343	406	t	217	18
ex10.mps	346	1796	8	14	88	17	t
fhnw-binschedule0_lp.mps	68	17	3	8	1581	23	19
fome13.mps	53	4	3	2	61	13	222
graph40-40_lp.mps	593	t	50	96	t	13	12
irish-electricity.mps	f	14	7	f	533	10	f
L1_sixm1000obs.mps	t	14	39	415	f	t	f
L1_sixm250obs.mps	t	3	7	7	f	49	f
Linf_520c.mps	20	198	20	29	f	t	t
neos-3025225_lp.mps	732	561	39	101	t	13	t
neos-5052403-cygnet.mps	306	17	9	30	123	6	170
neos-5251015_lp.mps	573	3	14	13	t	14	51
neos.mps	172	10	13	10	336	13	212
neos3.mps	t	16	4	3	1284	19	583
ns1687037.mps	931	771	26	508	387	t	f
ns1688926.mps	17	11	8	130	98	2	f
nug08-3rd.mps	139	15	20	12	247	116	t
pds-100.mps	f	19	30	36	28	55	f
physiciansched3-3.mps	t	t	46	55	307	89	f
qap15.mps	100	6	2	2	835	10	145
rail4284.mps	f	24	37	28	f	24	529
rmine15_lp.mps	379	272	156	133	684	126	t
s100.mps	809	221	15	20	t	10	507
s250r10.mps	245	20	9	8	190	6	f
s82_lp.mps	f	822	204	187	f	99	t
savsched1.mps	t	109	104	117	f	78	150
scpm1_lp.mps	9	112	39	55	967	47	634
self.mps	5	10	12	15	t	5	10
square41.mps	883	3	4	3	37	9	89
stat96v1.mps	t	10	19	16	26	1151	f
stormG2_1000.mps	159	51	33	85	f	34	1114
supportcase10.mps	78	40	12	9	402	11	1498

Table 2: Solver time reported for Case 1

Finally, the performance profile for all solvers used in Case 1 is presented in Figure 6. The performance profile shows that Mosek manages to solve the most problems faster than the other solvers, solving approximately 30 percent of the problems. However, due to the fact that Mosek fails to solve 4 problems in total, it's surpassed in the profile by GuRoBi and CPLEX firstly and then by Xpress as well (Xpress solved two more problems than Mosek, despite worst geometric mean). Similarly, when comparing open-source solvers, Tulip manages to solve the most problems faster, but as tau progress ultimately falls to last place because Tulip managed to solve successfully the least amount of problems.

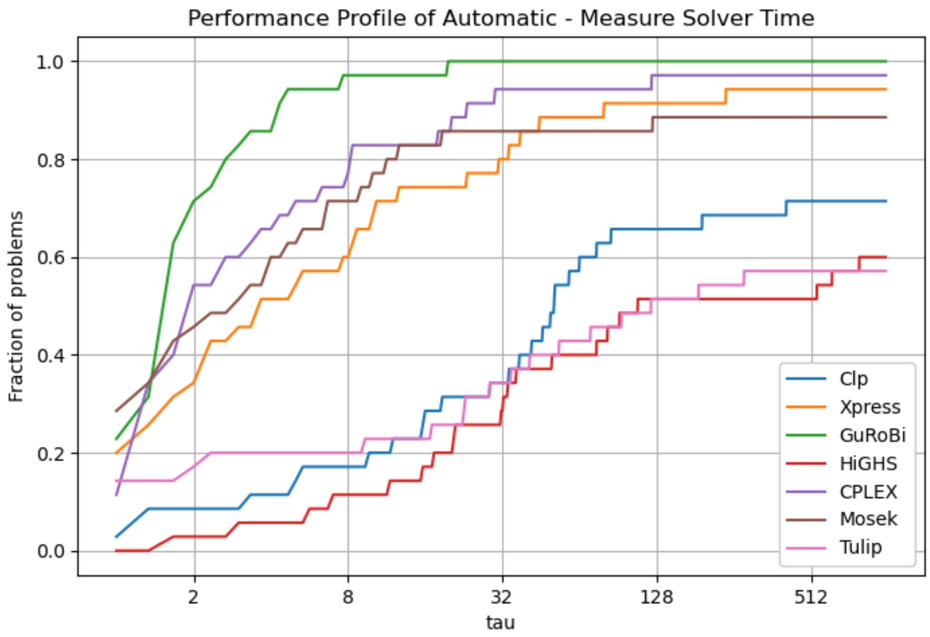

Figure 6: Performance Profile - Case 1

4.1.2 Case 2 - Primal Simplex - With Thread Limit Mode

After finishing the experiments in the default settings of the solvers, experimenting with different algorithms begun. Firstly, the Primal Simplex algorithm was tested after limiting thread usage to 1 thread, as already mentioned in Section 3.3.

	Clp	**Xpress**	**GuRoBi**	**CPLEX**	**Mosek**
Unscaled Geometric Mean	498.96	173.70	123.31	98.86	454.11
Scaled Geometric Mean	5.05	1.76	1.25	1.00	4.59
Optimal	24	31	29	31	21
Time Limit	8	4	4	4	12
Fail	3	0	2	0	2

Table 3: Geometric mean & Success Report for Case 2

The solvers included in Case 2 are Clp, Xpress, GuRoBi, CPLEX and Mosek. Recorded solver time, for each solver is listed in 4. As expected the average performance has diminished significantly, when compare with Case 1. In fact, by consulting 3 it's noticed that the best performing solver in Case 2 (CPLEX), reported an average performance that was 4.5 time worse, to the best performing solver in Case 1 (GuRoBi). This is also reflected in the success rate of all solvers, which was smaller when compared to Case 1 for all solvers.

Second to CPLEX in performance was GuRoBi, but GuRoBi solved less problems successfully than the third solver (Xpress). This exhibits that GuRoBi has a quite fast Primal Simplex optimiser, which lacks a little bit in robustness. Last two solvers are Mosek and Clp respectively, with Mosek providing a little bit better performance, but Clp outscoring Mosek in robustness by solving 3 more problems than Mosek and a total of 24 problems.

The performance profile, as reported in Figure 7, provides a similar picture to the results of Table 3. CPLEX is the best performing solver constantly in the profile, with GuRoBi being second in performance until surpassed by Xpress, due to Xpress solving more problems successfully. Similarly, Mosek tops Clp in the beginning, but as tau progress is outscored by Clp, as Clp's Primal Simplex optimiser proved more robust than Mosek's.

Problems	Clp	Xpress	GuRoBi	CPLEX	Mosek
chromaticindex1024-7.mps	145	129	4	2	236
cont1.mps	227	t	187	60	494
cont11.mps	1318	1296	f	959	t
datt256_lp.mps	513	1228	t	t	t
ex10.mps	717	48	12	13	73
fhnw-binschedule0_lp.mps	141	11	2	4	4
fome13.mps	107	7	6	5	982
graph40-40_lp.mps	587	t	84	34	t
irish-electricity.mps	f	164	f	63	419
L1_sixm1000obs.mps	t	43	71	1645	t
L1_sixm250obs.mps	t	4	7	6	64
Linf_520c.mps	739	1800	684	t	t
neos-3025225_lp.mps	716	370	371	1313	t
neos-5052403-cygnet.mps	247	1123	265	90	31
neos-5251015_lp.mps	586	250	t	1388	416
neos.mps	93	94	102	22	9
neos3.mps	t	487	21	2	6
ns1687037.mps	330	415	t	t	t
ns1688926.mps	280	26	19	529	t
nug08-3rd.mps	t	49	38	67	t
pds-100.mps	f	262	200	144	388
physiciansched3-3.mps	t	23	109	46	202
qap15.mps	98	6	5	5	t
rail4284.mps	1500	220	1207	63	62
rmine15_lp.mps	t	295	1063	987	1775
s100.mps	23	t	35	16	f
s250r10.mps	14	1617	6	24	f
s82_lp.mps	t	t	t	t	t
savsched1.mps	433	46	102	60	724
scpm1_lp.mps	1798	736	582	90	774
self.mps	f	29	25	17	1
square41.mps	971	6	5	6	t
stat96v1.mps	53	91	23	129	324
stormG2_1000.mps	t	175	651	86	694
supportcase10.mps	65	26	37	45	1353

Table 4: Solver time reported for Case 2

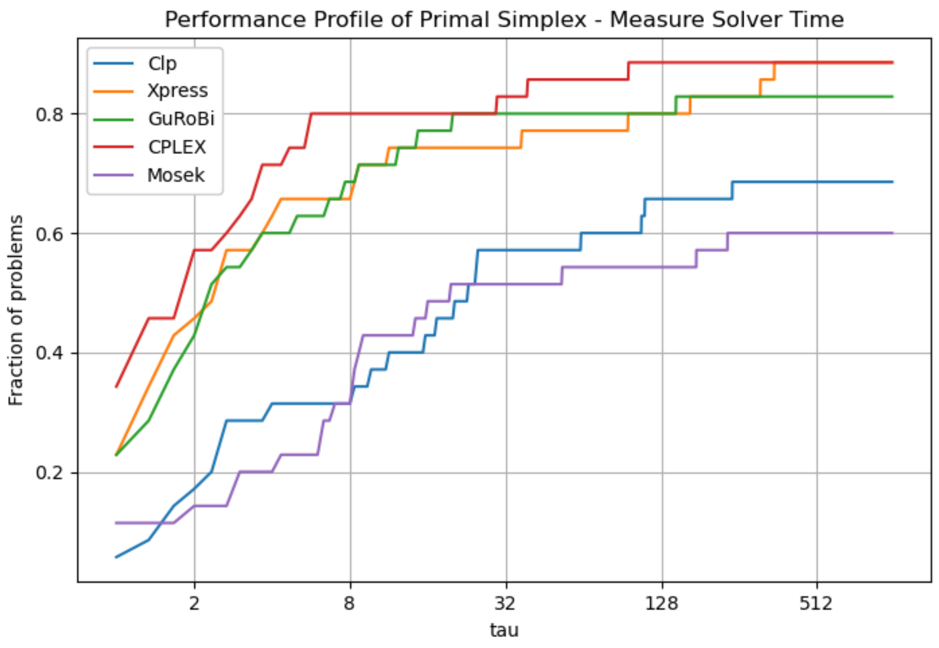

Figure 7: Performance Profile - Case 2

4.1.3 Case 3 - Dual Simplex - With Thread Limit Mode

Next case that was examined concerned the Dual Simplex Algorithm, once again with a Thread limit of maximum 1 thread. All solvers that were used in 4.1.2 were again used, with the addition of HiGHS solver. Performance concerning solver time is reported in Tables 6 and 5.

	Clp	Xpress	GuRoBi	CPLEX	HiGHS	Mosek
Unscaled Geometric Mean	493.14	77.09	48.92	64.53	552.07	496.84
Scaled Geometric Mean	10.08	1.58	1.00	1.32	11.28	10.16
Optimal	23	33	34	30	21	22
Time Limit	8	2	1	5	7	10
Fail	4	0	0	0	7	3

Table 5: Geometric mean & Success Report for Case 3

Although HiGHS performance is included in Case 3, it must be stated that the recorded performance is identical to the reported performance in Case 1. Despite the fact that HiGHS supports multi-threading, it currently is available only on Linux and the benchmarks were all executed on Windows 10. Additionally, although HiGHS has an Interior-Point Method optimiser, a quick check in HiGHS logs proved that all problems in Case 1 were solved by a Dual Simplex optimiser. Hence, it was deemed unnecessary to re-run the experiments, as Case 1 for HiGHS was both single-threaded and used a Dual Simplex method.

Problems	Clp	Xpress	GuRoBi	CPLEX	HiGHS	Mosek
chromaticindex1024-7.mps	28	95	3	2	77	187
cont1.mps	280	106	54	150	t	531
cont11.mps	t	693	189	t	1297	t
datt256_lp.mps	t	874	256	t	t	t
ex10.mps	470	1760	5	11	88	275
fhnw-binschedule0_lp.mps	652	17	2	6	1581	166
fome13.mps	54	5	5	4	61	108
graph40-40_lp.mps	1103	t	256	31	t	t
irish-electricity.mps	f	16	63	15	533	126
L1_sixm1000obs.mps	t	14	27	82	f	t
L1_sixm250obs.mps	t	3	6	5	f	268
Linf_520c.mps	1429	218	61	318	f	1282
neos-3025225_lp.mps	t	630	393	t	t	f
neos-5052403-cygnet.mps	187	18	34	111	123	61
neos-5251015_lp.mps	t	4	44	t	t	t
neos.mps	40	10	10	10	336	65
neos3.mps	1737	16	3	1	1284	1069
ns1687037.mps	934	1469	806	t	387	t
ns1688926.mps	17	8	5	5	98	t
nug08-3rd.mps	138	14	14	7	247	915
pds-100.mps	f	16	12	12	28	31
physiciansched3-3.mps	f	t	t	21	307	236
qap15.mps	1703	12	11	10	835	773
rail4284.mps	988	37	28	50	f	979
rmine15_lp.mps	359	272	345	123	684	1373
s100.mps	818	225	79	363	t	f
s250r10.mps	246	32	15	16	190	f
s82_lp.mps	f	968	1090	767	f	t
savsched1.mps	t	88	489	304	f	888
scpm1_lp.mps	510	198	178	346	967	t
self.mps	12	17	12	13	t	1
square41.mps	31	2	3	2	37	t
stat96v1.mps	t	13	13	10	26	64
stormG2_1000.mps	160	42	19	66	f	185
supportcase10.mps	108	40	16	6	402	73

Table 6: Solver time reported for Case 3

With regards to performance, GuRoBi was the fastest and more robust solver in Case 3, only failing in solving one problem within the time-limit. GuRoBi was closely followed by CPLEX and Xpress, by small margins. Xpress, although performing a little worse in the geometric mean, actually managed to solve three more problems than CPLEX, so it's possible that if a bigger time limit was imposed, Xpress would perform better than CPLEX. Interestingly, Clp was the forth solver in ranking, slightly edging out Mosek. This was the only instance where a commercial solver was surpassed by an open-source solver.

The performance profile, presented on Figure 8 shows that the fastest solver for the biggest percentage of problems, although by a small margin, was CPLEX closely followed by GuRoBi. Eventually, both GuRoBi and Xpress surpass CPLEX, as tau progress, because they showed more robustness. Clp follows and although HiGHS ultimately is surpassed by Mosek, HiGHS seriously challenges Mosek in approximately 50 percent of the problems in the benchmark.

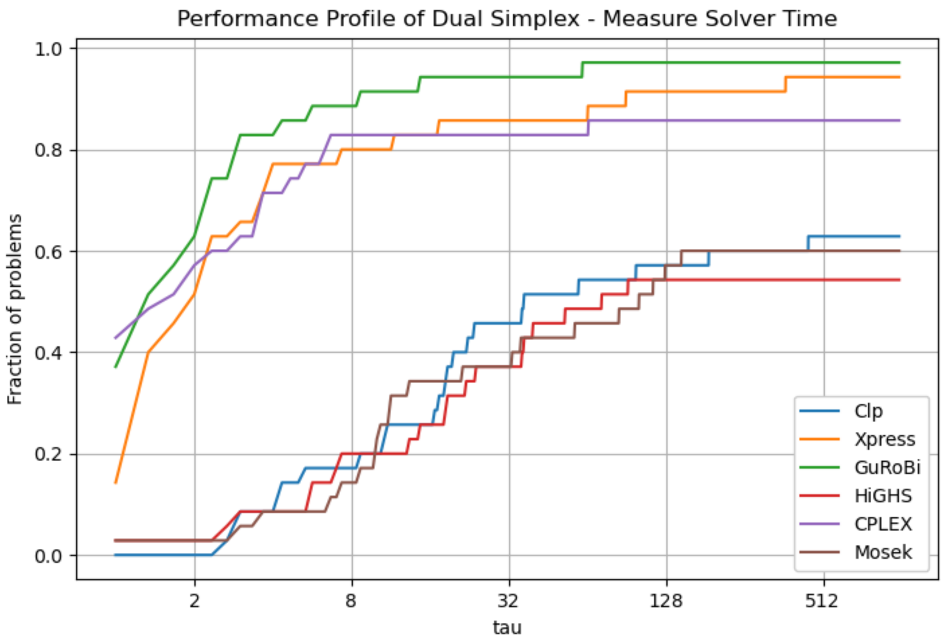

Figure 8: Performance Profile - Case 3

4.1.4 Case 4 - Interior-Point Method - With Crossover & Thread Limit Mode

In the next case, only Interior-Point method optimisers were tested using only one thread and crossover. Although, Clp and HiGHS have an Interior-Point method optimiser, both solvers are primarily Simplex solvers, hence these optimisers were not included in the next two cases. Additionally, Tulip is not included as it has no crossover functionality. Hence, in Case 4 only commercial solvers were tested.

	Xpress	**GuRoBi**	**CPLEX**	**Mosek**
Unscaled Geometric Mean	22.81	30.32	56.15	78.83
Scaled Geometric Mean	1	1.33	2.46	3.46
Optimal	35	34	32	31
Time Limit	0	1	2	3
Fail	0	0	1	1

Table 7: Geometric mean & Success Report for Case 4

Interestingly, the best performing solver was Xpress with a shifted geometric mean of 22.81 seconds, as stated in Table 7. This is way better than the overall performance of Xpress in Case 1 and only 1.025 longer performance than the best performing solver so far, GuRoBi in Case 1. This was peculiar and an effort to explain it will follow in Section 4.2. Xpress was also able to solve all 35 problems successfully. Close second was GuRoBi, with CPLEX and Mosek following. The performance profile 9 exhibited below, presents a similar description of the results reported in Table 7.

It's easy to notice that since all tested solvers are commercial, no huge gaps in performance can be observed, although it is easy to distinct the best performing solver.

Problems	Xpress	GuRoBi	CPLEX	Mosek
chromaticindex1024-7.mps	11	4	2	9
cont1.mps	10	14	28	421
cont11.mps	10	30	332	f
datt256_lp.mps	168	525	167	218
ex10.mps	34	36	16	38
fhnw-binschedule0_lp.mps	7	7	11	25
fome13.mps	3	3	3	29
graph40-40_lp.mps	39	36	194	15
irish-electricity.mps	3	150	f	16
L1_sixm1000obs.mps	122	t	t	t
L1_sixm250obs.mps	4	16	9	109
Linf_520c.mps	14	19	31	t
neos-3025225_lp.mps	37	33	142	31
neos-5052403-cygnet.mps	15	9	35	14
neos-5251015_lp.mps	4	8	7	17
neos.mps	11	14	16	27
neos3.mps	25	4	12	41
ns1687037.mps	45	30	t	t
ns1688926.mps	26	7	20	2
nug08-3rd.mps	30	28	53	253
pds-100.mps	56	55	166	218
physiciansched3-3.mps	22	76	188	265
qap15.mps	2	2	2	27
rail4284.mps	34	37	42	40
rmine15_lp.mps	66	128	216	618
s100.mps	14	15	23	20
s250r10.mps	8	9	10	8
s82_lp.mps	195	166	268	248
savsched1.mps	134	95	72	84
scpm1_lp.mps	42	37	81	113
self.mps	5	10	6	6
square41.mps	3	4	3	16
stat96v1.mps	28	19	116	94
stormG2_1000.mps	25	47	30	41
supportcase10.mps	10	10	22	28

Table 8: Solver time reported for Case 4

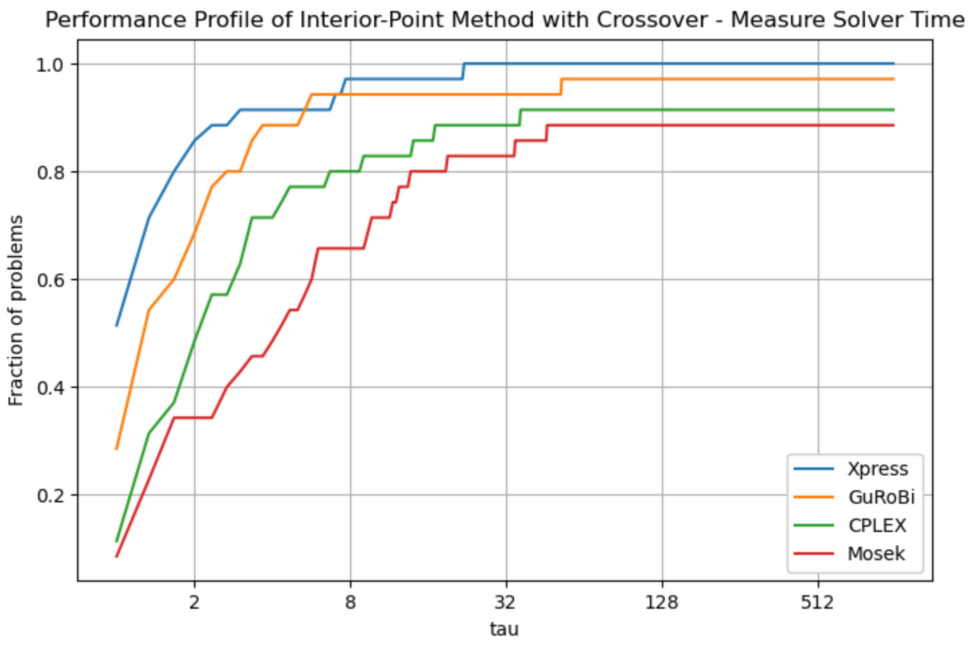

Figure 9: Performance Profile - Case 4

4.1.5 Case 5 - Interior-Point Methods Mode - No Crossover

In the final case, crossover was deactivated for all Interior-Point optimisers. This allowed to test a fair benchmark for Interior-Point method optimisers that included Tulip as well. Similarly with HiGHS in the Dual Simplex case, Tulip's solver time in this Case is exactly the same with the recorded solver time in Case 1, as Tulip is not a multi-threaded solver. The reported metrics are presented in Tables 9 and 10.

	Xpress	**GuRoBi**	**CPLEX**	**Mosek**	**Tulip**
Unscaled Geometric Mean	17.94	32.26	38.12	54.13	381.82
Scaled Geometric Mean	1.00	1.80	2.12	3.02	21.29
Optimal	33	30	31	31	20
Time Limit	0	1	1	1	6
Fail	2	4	3	3	9

Table 9: Geometric mean & Success Report for Case 5

In general, eliminating crossover to a basic solution seemed to improve solver time across all solvers. However, lack of crossover led to more instance failures, hence this is not strictly reflected to the geometric means of all solvers. For example GuRoBi's geometric mean is worse in Case 5 than in Case 4. Once again, Xpress performed better than all solvers and scored the lowest geometric mean across all cases with 17.94, but without managing to solve two problems successfully. GuRoBi followed, without being able to solve 5 problems (the most solver failures for GuRoBi) and then CPLEX and Mosek were respectively third and forth, each solving 31 problems. Finally, Tulip followed with significant slower performance than all solvers. This gap in performance is also reflected in the performance profile in Figure 9.

Problems	**Xpress**	**GuRoBi**	**CPLEX**	**Mosek**	**Tulip**
chromaticindex1024-7.mps	10	1	1	2	3
cont1.mps	3	3	3	7	14
cont11.mps	3	3	f	7	18
datt256_lp.mps	3	3	4	5	18
ex10.mps	18	31	9	35	t
fhnw-binschedule0_lp.mps	3	3	3	10	19
fome13.mps	2	2	2	22	222
graph40-40_lp.mps	5	1	1	6	12
irish-electricity.mps	3	7	21	14	f
L1_sixm1000obs.mps	f	t	t	t	f
L1_sixm250obs.mps	4	15	9	104	f
Linf_520c.mps	11	17	25	f	t
neos-3025225_lp.mps	23	16	132	29	t
neos-5052403-cygnet.mps	12	8	33	13	170
neos-5251015_lp.mps	4	f	7	15	51
neos.mps	11	13	15	26	212
neos3.mps	15	2	3	35	583
ns1687037.mps	11	25	39	f	f
ns1688926.mps	f	f	f	f	f
nug08-3rd.mps	3	2	4	175	t
pds-100.mps	54	52	162	217	f
physiciansched3-3.mps	19	f	183	268	f
qap15.mps	1	1	1	25	145
rail4284.mps	32	35	40	39	529
rmine15_lp.mps	63	96	206	607	t
s100.mps	14	15	23	19	507
s250r10.mps	10	8	10	8	f
s82_lp.mps	194	170	260	243	t
savsched1.mps	11	12	17	19	150
scpm1_lp.mps	39	36	76	106	634
self.mps	2	2	2	2	10
square41.mps	2	2	1	15	89
stat96v1.mps	4	f	f	13	f
stormG2_1000.mps	21	41	24	38	1114
supportcase10.mps	10	10	21	28	1498

Table 10: Solver time reported for Case 5

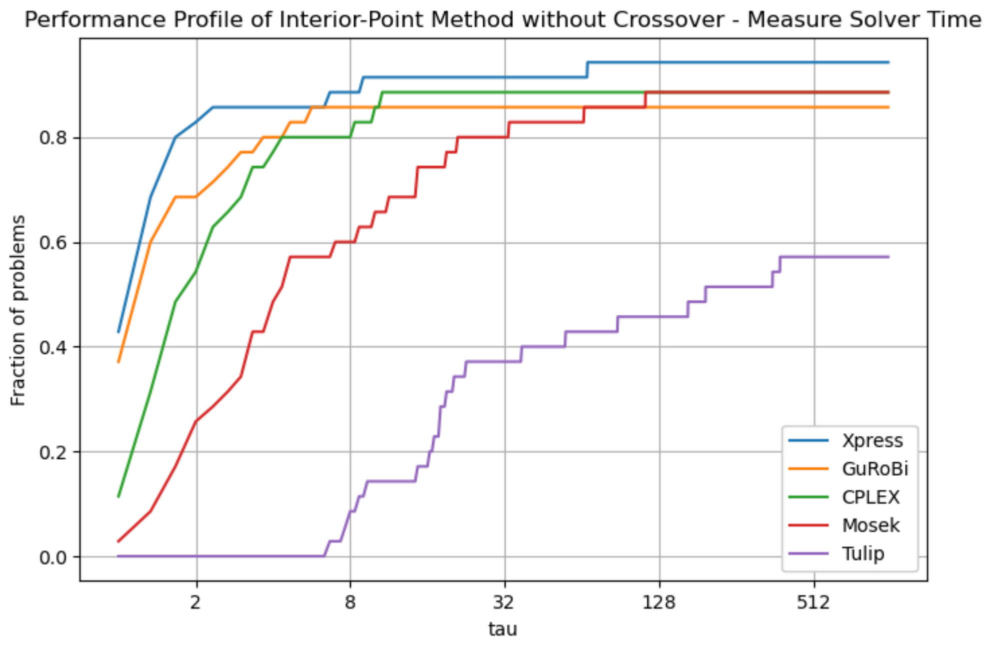

Figure 10: Performance Profile - Case 5

4.2 Discussion

In this chapter, a more thorough analysis of the results reported in Section 4.1 will be attempted. This analysis consist of two parts. First, an individual assessment of all solvers, included in the benchmarks will occur, followed by a comparison between the performance of Simplex methods vs Interior-Point. It is stated that all drawn conclusions relate to performance on the examined test set and generalisation of them could be misleading.

4.2.1 Performance of Solvers

Clp

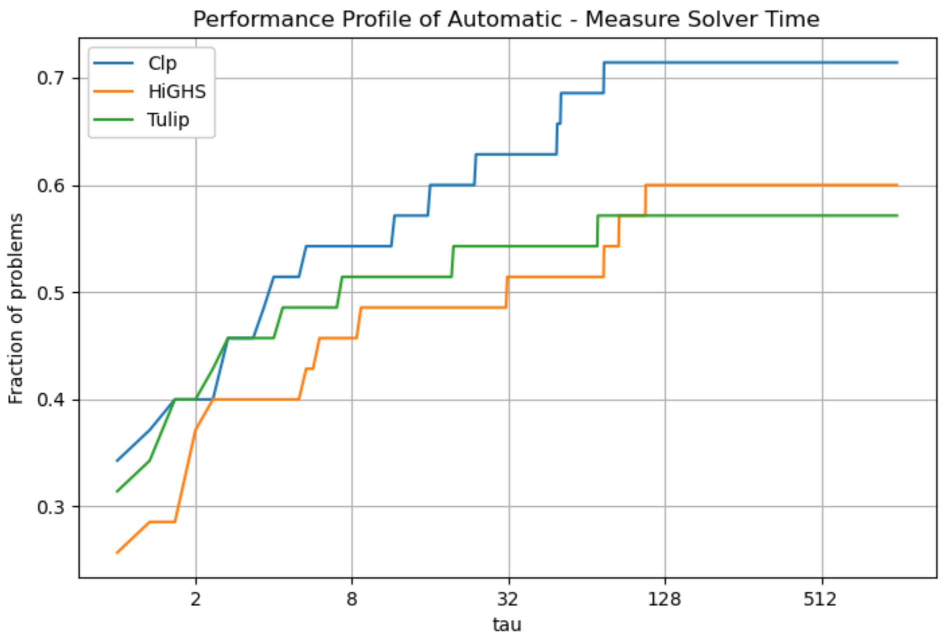

Figure 11: Performance Profile - Open Source Solvers - Case 1

Clp was clearly the best-performing open-source solver in this study. Although, the maximum problems that Clp was able to solve was 25 out of 35, this number would improve if a bigger time limit was set. Nonetheless, it solved the most problems out of the open-source solvers. This performance is validated by the geometric means reported and the modified performance profile below in Figure 11, which draws performance only from open-source solvers.

In general, Clp was not that competitive to commercial solvers. With the exception of Mosek's Dual Simplex optimiser, there was no other commercial solver that was surpassed by Clp, in any of the five examined cases. As Mosek is primarily an Interior-Point method optimiser, this showcases that the performance gap between commercial solvers and open-source solvers is significant.

Clp was the only open-source solver that utilized more than one algorithms in their default settings. Both Clp's Primal Simplex and Dual Simplex optimiser were being used and this enabled the solver to terminate successfully in more problems and overall better performance. As it can be observed in 12, a bigger fraction of problem were solved within small values of tau using default settings, because both algorithms were used by the solver, each when it was appropriate. This does not imply that the algorithms were used concurrently, as Clp is not multi-threaded, only that the selection process behind which algorithm to use was efficient.

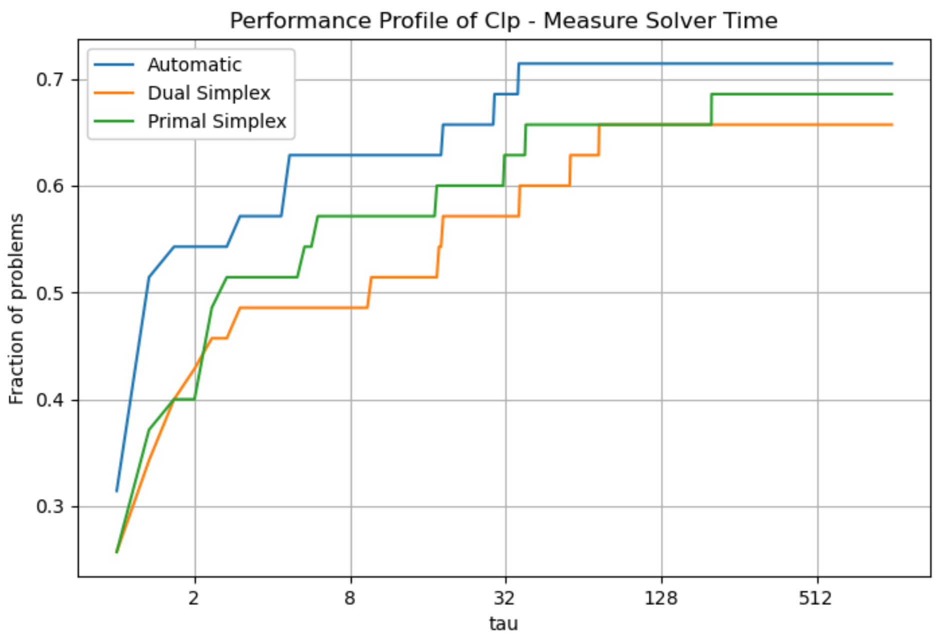

Figure 12: Performance Profile - Clp - All Cases

FICO Xpress

Xpress proved to be the solver with the most ambiguous results. During the experimental design, it was mainly assumed that with the exception of Case 5, where the computational load of computing the crossover would be lifted from the benchmarks, the best performing Case across solvers would be Case 1, as multi-threading and free algorithm selection would supposedly improve performance. This proved true, for all solvers but Xpress. As presented in the performance profile 13 and the reported results of Section 4.1, not only the Interior-Point method optimiser of Xpress was faster, but also more robust, as it completed successfully more problems.

The reason behind this gap in performance is not entirely unclear. Consultation of the logs of Case 1 showed that default settings of Xpress, stick to the Dual Simplex optimiser at all times. Even though more threads are available, Xpress does not seem to use different methods concurrently, but utilizes the extra threads to use the parallel version of Dual Simplex. Hence, the main assumption is that Xpress does not have a good selection strategy behind which method to use. This results in Dual Simplex being selected almost exclusively and thus influencing negatively, the expected performance.

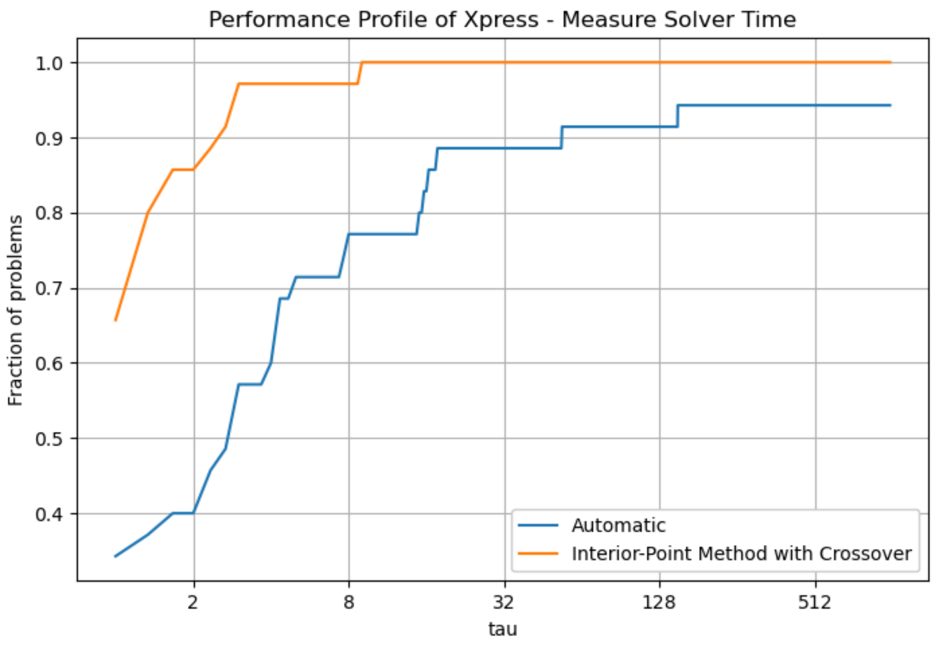

Figure 13: Xpress - Automatic versus Interior-Point Method with Crossover

Nonetheless, performance of the Interior-Point method optimiser of Xpress was excellent, reporting the best performance in both cases were Interior-Point methods were involved. Simplex optimisers were also quite good being comfortably, in the first three places of average reported solver time. The main problem of Xpress, seems to be its default settings. Visualizing performance of Xpress in each problem using default settings and using an Interior-Point method optimiser paints two different pictures, as exhibited in Figures 14 and 15.

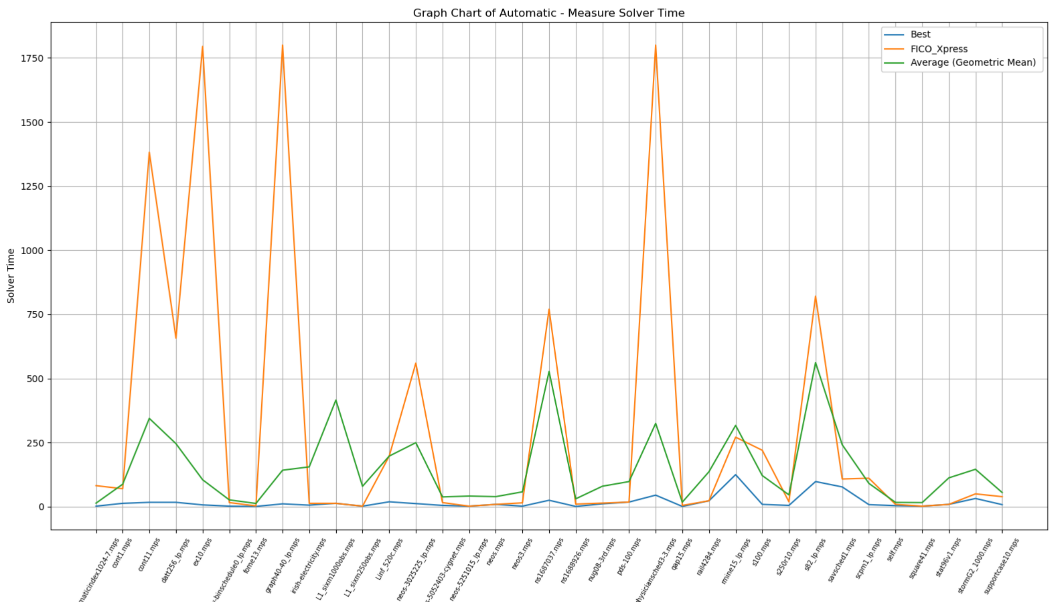

Figure 14: Xpress - Performance chart of Case 1 (Automatic) against baselines

Figure 14 exhibits a solver that even though is within the general area enclosed by the best performance reported by any solver in a problem and the geometric average, also has instances where performance is grossly far from the average baseline. Meanwhile, Figure 15 presents a solver that is rarely, if never, above the average reported performance in a problem and mostly sets the baseline for

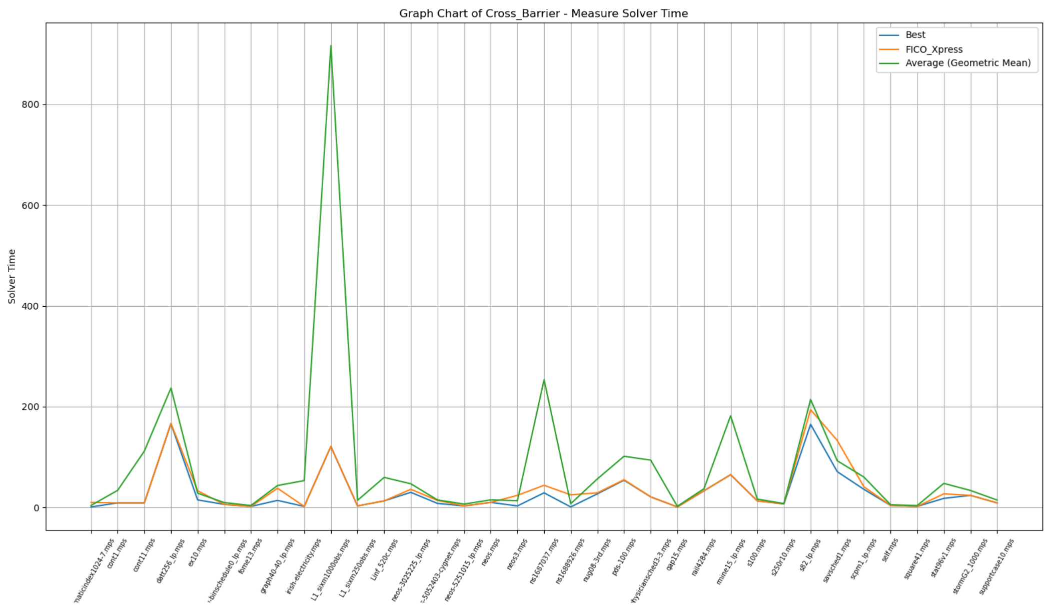

Figure 15: Xpress - Performance chart of Case 4 (IPM) against baselines

best performance across solvers.

<u>GuRoBi</u>

It's fair to say that GuRoBi was definitely the most consistent solver across all different cases. Being best performing solver, in Case 1 (Automatic) and Case 3 (Dual Simplex), GuRoBi never dropped anywhere below the second place across all cases, with regards to the metric of the shifted geometric mean. GuRoBi's default settings recorded the best performance across all solvers in cases 1-4.

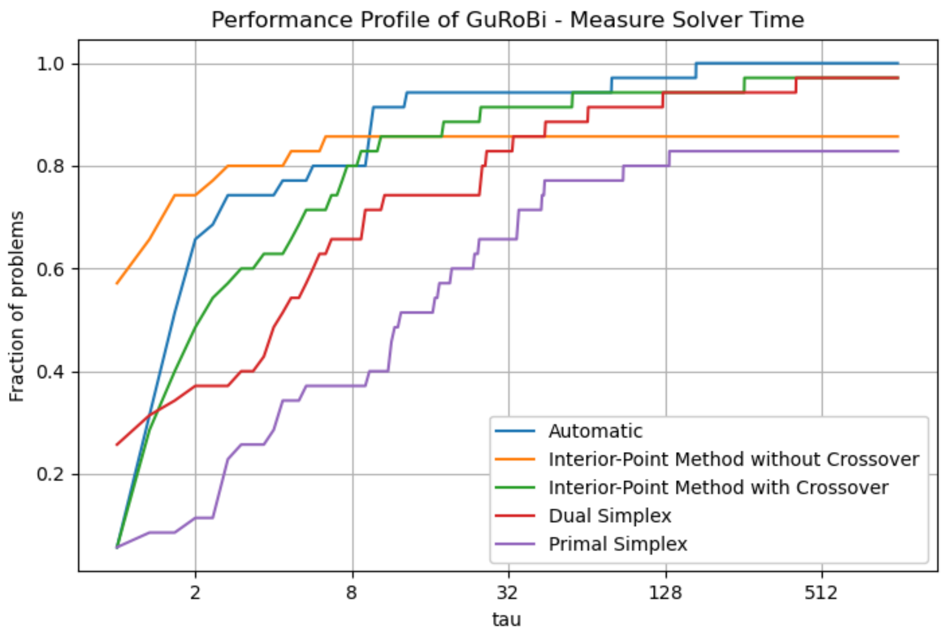

Figure 16: GuRoBi - Performance Profile of all cases

The performance profile in Figure 16, highlights this consistency that GuRoBi has across different methods used and Figure 17 exhibits the performance of a quality solver, that rarely fails to be above the baseline for average performance, in the metric of solver time.

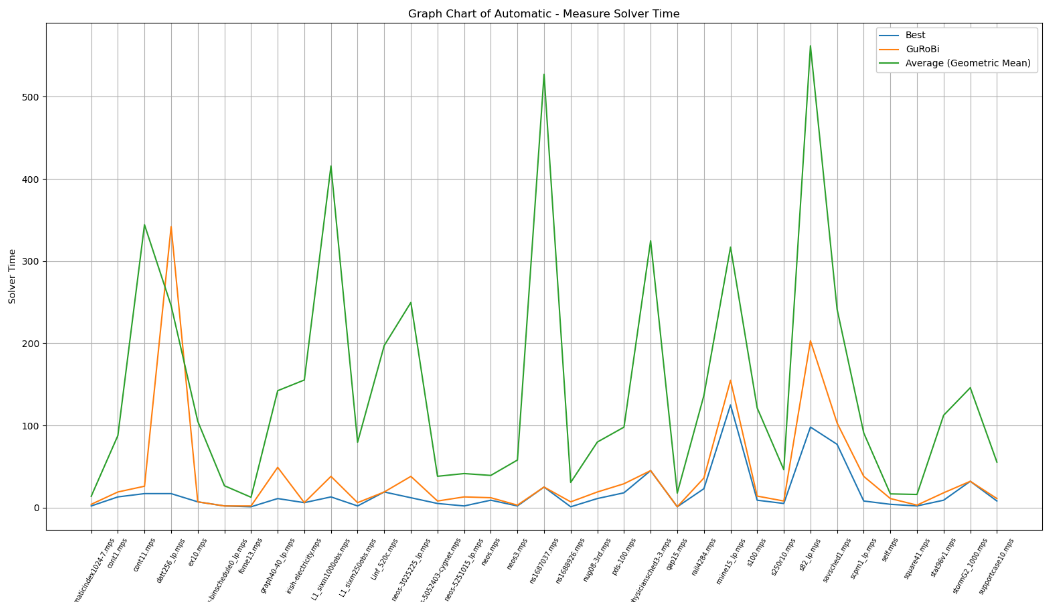

Figure 17: GuRoBi - Performance chart of Case 1 (Automatic) against baselines

This overall consistency across methods and overall speed that GuRoBi recorded in the selected test set, definitely supported the argument that GuRoBi is the fastest and more robust solver around.

HiGHS

HiGHS solver displayed performance that was not necessarily expected. Although ranking last in the metric of shifted geometric mean for both Case 1 and Case 3, this performance was significantly skewed by the number of fails that HiGHS exhibited. HiGHS had seven fails and seven problems that were stopped due to time limit. Interestingly, the seven fails were not product of numerical errors or early terminations, but all occurred due to time-out errors. Basically, in seven problems that were part of the test set, the solver never started running and the problem was terminated manually after a significant amount of time.

However, this was largely inconsistent based on separate reported results, recorded in Professor Mittelman's benchmarks [6]. This prompted further investigation, in an effort to explain this unexpected performance. Firstly, the benchmarks of the specific failed problems were re-initiated, on a different Laptop and different operating system (Linux). The displayed performance was identical, hence the possibility of the issue being related to the OS or the hardware dissipated.

Then, the HiGHS solver was installed in the same computer that all experiments were run, but this time outside of Julia and the problematical instances were tested. Using the solver directly presented performance similar to results reported by Mittelman. This performance is not listed here as it is out of the scope of this dissertation, where the use of benchmarking in Julia is tested. The list of problems that HiGHS failed due to time-out errors are listed in the appendix (Table 12). Hence, it is assumed that the main reason behind this gap in performance is mostly due to problems in the interface of HiGHS, between the solver and Julia.

Problems between the interface of HiGHS in Julia and solver interaction can also be explained by observing the reported metrics of Julia Time for the HiGHS solver. Even in problems where HiGHS terminated successfully, the overall time Julia spent to solve the problem was highly divergent from the solver time.

In Table 11, the geometric mean of HiGHS is listed for the metric of Julia Time, i.e., wall clock time of Julia being spent on a problem. It's clear that HiGHS solver time is almost two times bigger than the geometric mean of the solver time as reported in Table 1. In contrast, Clp which is also a mostly Simplex open-source solver, shows a very small increase of Julia Time (343.31) when compared to solver time (340.04).

	Clp	**Xpress**	**GuRoBi**	**CPLEX**	**HiGHS**	**Mosek**	**Tulip**
Unscaled Geometric Mean	343.31	74.10	22.47	45.30	969.65	52.70	380.84
Scaled Geometric Mean	15.28	3.30	1.00	2.02	43.14	2.34	16.95
Optimal	25	33	35	34	21	31	20
Time Limit	6	2	0	0	7	3	6
Fails	4	0	0	1	7	1	9

Table 11: Julia time for All solvers in Case 1

Even with this definitely skewed performance, it is interesting to observe performance of HiGHs against other solvers. The performance profile between the Dual Simplex algorithm of HiGHS and the Dual Simplex algorithm of Clp is presented in Figure 18. This performance profile is presented, because Clp is the solver that HiGHS bares the most similarities with.

The performance profile highlights that Clp manages to solve about 6 % more problems than HiGHS, which was already presented in tables 11 and 1. Additionally, Clp seems to solve most problems faster initially (where tau equals to one) and converges to the maximum problems it can solve within smaller fraction of the best recorded performance a little bit faster than HiGHS. However, as Clp did not experience similar time-out errors it would be interesting to check the performance profile on a reduced or alternate test set or entirely outside of Julia and JuMP.

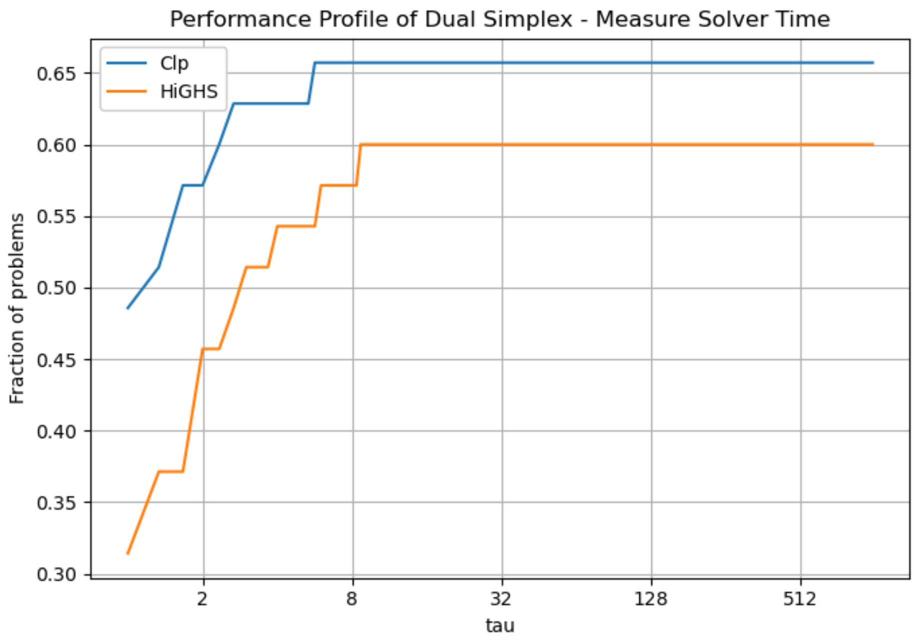

Figure 18: Clp vs HiGHS - Performance Profile for Case 3

IBM ILOG CPLEX

In a similar manner to GuRoBi, but arguably second overall, CPLEX also showed consistency as a solver.

As exhibited by the shifted geometric means in Section 4.1, CPLEX was the best Primal Simplex optimiser, the second best when using default settings or the Dual Simplex optimiser and third best

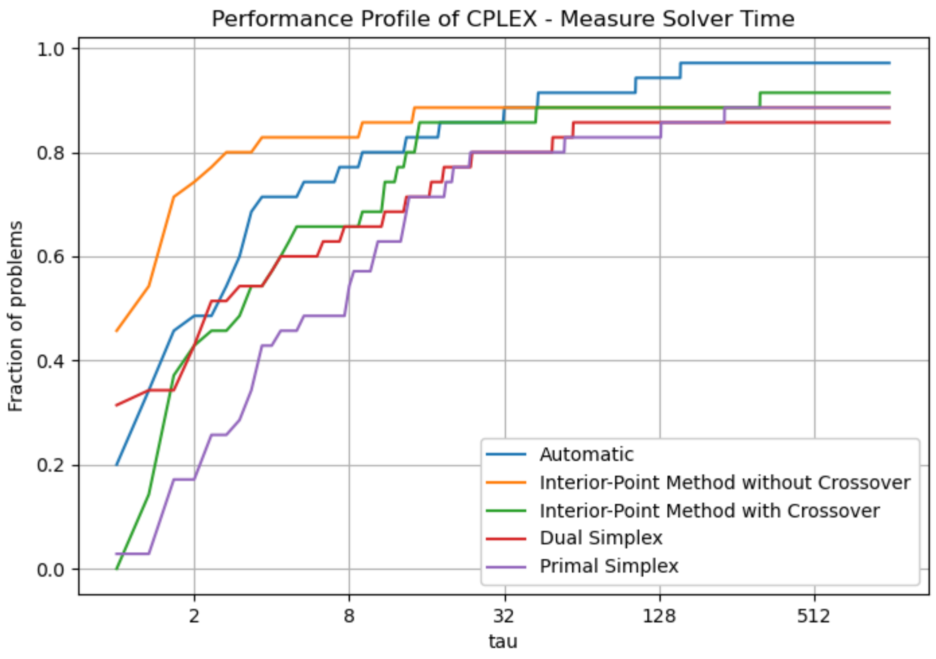

Figure 19: CPLEX - Performance Profile of all cases

Interior-Point method optimiser. In general, performance of CPLEX across different methods was not especially variant and heavily robust as seen in Figure 19.

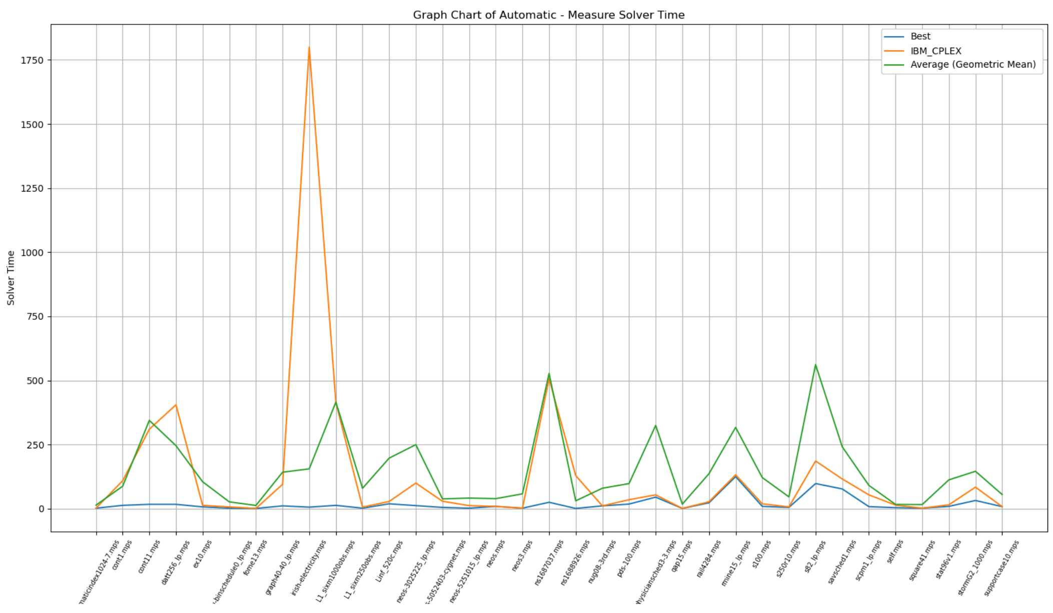

Figure 20: CPLEX - Performance chart of Case 1 (Automatic) against baselines

Additionally, Figure 20 showcases that utilizing CPLEX in its default settings will provide performance, close to the general optimal.

Since CPLEX and GuRoBi have been created by members of the same development team, it is interesting to provide a more in-depth look at how they compare. To achieve that, a performance profile of the two solvers will be provided, with a much smaller tau range, to signify the small performance differences between the two solvers. The performance profile provided corresponds to the solver time performed in Case 1, which was the best performing case for both solvers.

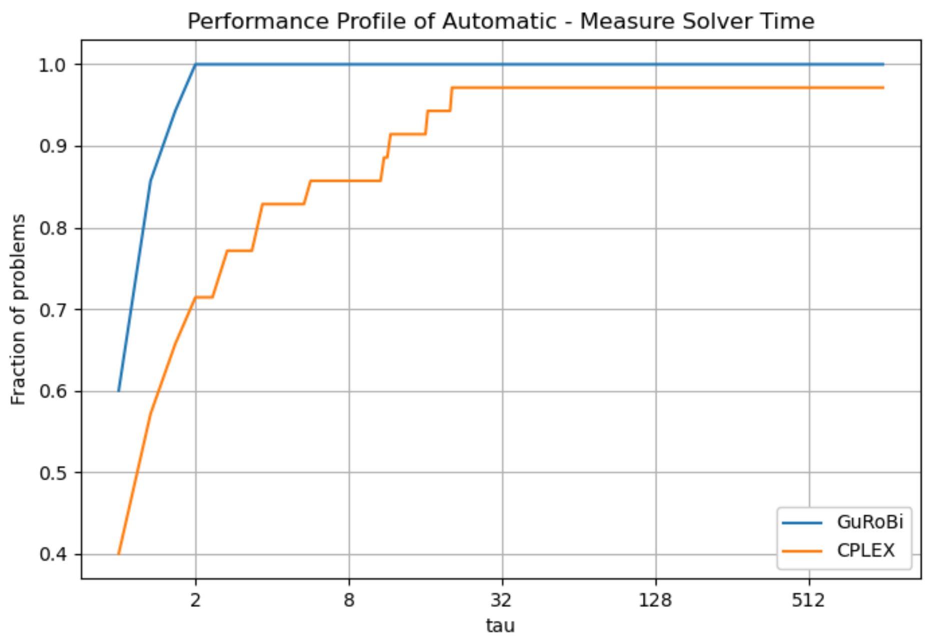

Figure 21: CPLEX vs GuRoBi - Performance Profile for Case 1

Arguably, with the smaller tau range applied the results may seem more dramatic than they really are. GuRoBi converges to solving all problems within approximately 2 times the best recorded solution for any proble, while CPLEX converges to solving 34/35 problem within approximately 25 times of the best solution for all problems.

However, it must be addressed that in performance profiles the size and difficulty of the problems are not reflected in the chart. Hence, despite the fact that CPLEX recorded a solution for a problem that was 25 times worse than the best solution provided by GuRoBi, this could be a very easy problem, that CPLEX solved within seconds, but GuRoBi solved within fractions of a second. Either way, Figure 21 showcases a clear winner, even if the differences in performance can be actually small.

<u>Mosek</u>

Mosek was overall the third best performing solver when using default settings, with performance quite close to CPLEX, which was second. Taking into account that Mosek solved 31 out of 35 problems, in comparison to CPLEX which solved 34 problems, it is clear that Mosek is really fast as the shifted geometric mean did not seem to be influenced that much by these four failures, due to fast recorded metrics in the rest of the problems.

In Figure 22 this performance is highlighted and it is shown that with the exception of the four problems that terminated due to time limit, performance was generally very close to the best recorded case.

Being primarily an Interior-Point method optimiser, Mosek performed really well in Cases 4 and 5, where Interior-Point methods were tested. However, it was outscored by all other commercial solvers, although not by huge margins.

The most problematic aspect of Mosek was by far its Simplex optimisers, as highlighted by the performance profile of Mosek across all examined cases in Figure 23. Mosek's Simplex optimisers proved to be less than effective and managed to solve a significant smaller number of problems than other commercial solvers and being steadily challenged by open-source solvers. In fact, Mosek's Dual Simplex method was actually outscored in geometric mean by Clp's, which further highlights that Mosek is probably better for users that are not interested in Simpmex Methods.

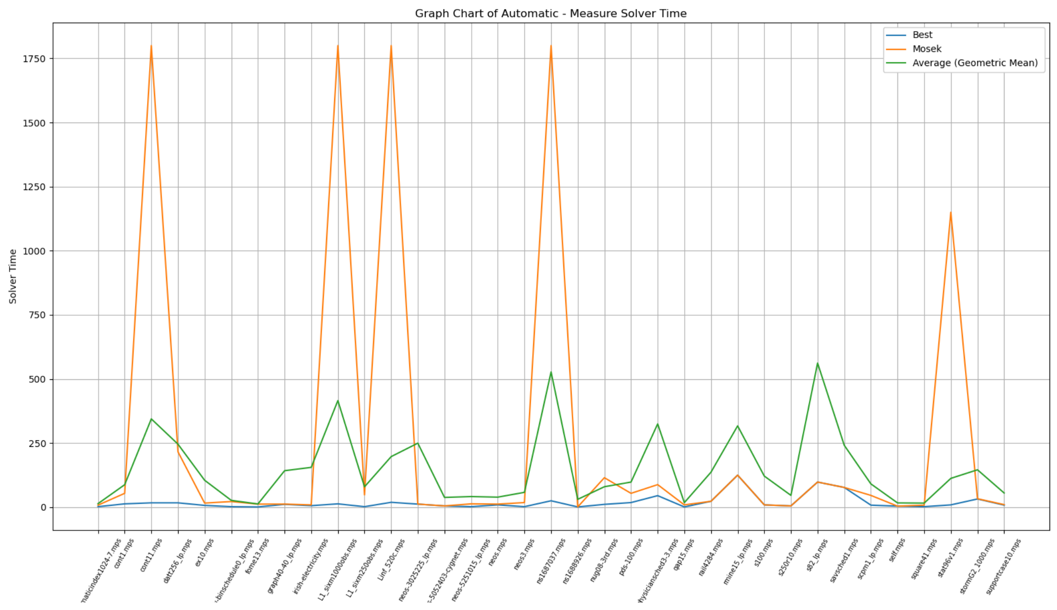

Figure 22: Mosek - Performance chart of Case 1 (Automatic) against baselines

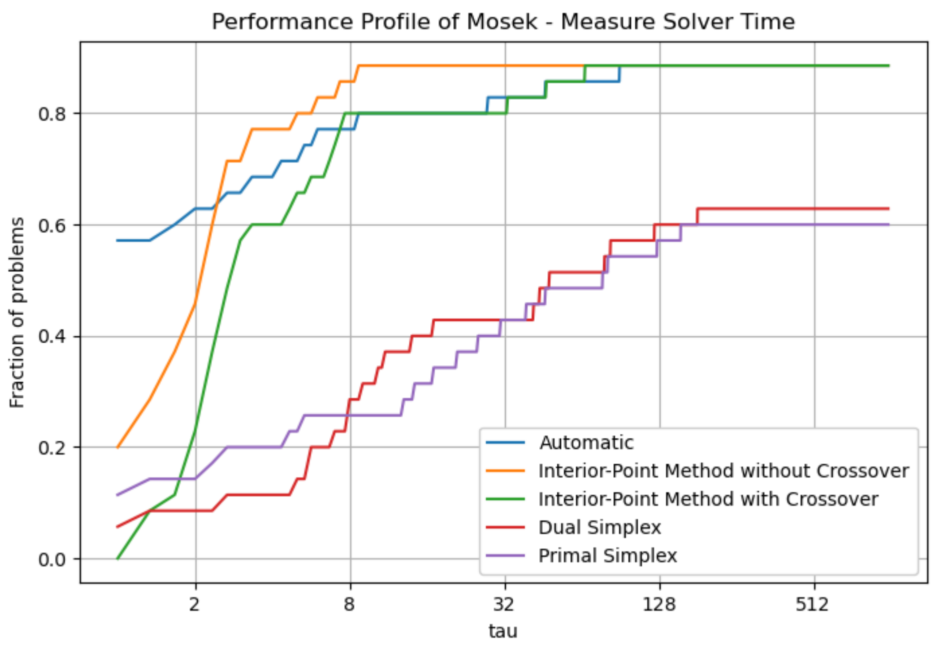

Figure 23: Mosek - Performance Profile of all cases

Tulip

Tulip as a solver had arguably problems with robustness. Tulip solved the least amount of problems with twenty problems out of thirty five problems. However, as it can be evidenced by the performance profile for open-source solvers in Figure 11, its speed is remarkable, despite the fact that this solver is quite new.

Tulip seems to challenge Clp within small fragments of tau, but slowly fails as tau progresses and eventually is surpassed by HiGHS. Additionally, as mentioned in Table 1, even though Tulip solved less problems than HiGHS, it actually recorded a better shifted geometric mean, which showcases the speed that Tulip exhibited in specific instances.

However, it must be addressed that the lack of crossover computation diminishes the computational load required by the solver. Getting a basic solution is not absolutely important for an LP problem, although in practice it has its benefits and Tulip does not offer this functionality. It's also possible that the lack of crossover, influences overall performance on Tulip as in a lot of instances that Tulip failed, the solver had terminated successfully, but reported slightly wrong results. Additionally, Tulip failed in some instances due to numerical errors arising in Linear Algebra packages. This issue generally occurred in problems, that were great in dimensionality and size of the .mps file.

Since Tulip has no crossover functionality, it was also included in Case 5 where all the Interior-Point Method optimisers were used, but without crossover. As all the solvers used in this case, apart from Tulip, were commercial solvers, the Figure in 24, shows a solver that is very far from the average performance. This was expected, as all commercial solvers displayed their best, even though not most robust, performance in Case 5.

However, it is not very reasonable to draw conclusions for Tulip comparatively to the performance of state-of-the-art commercial solvers, but as no other open-source Interior-Point methods solvers were used there was no other alternative.

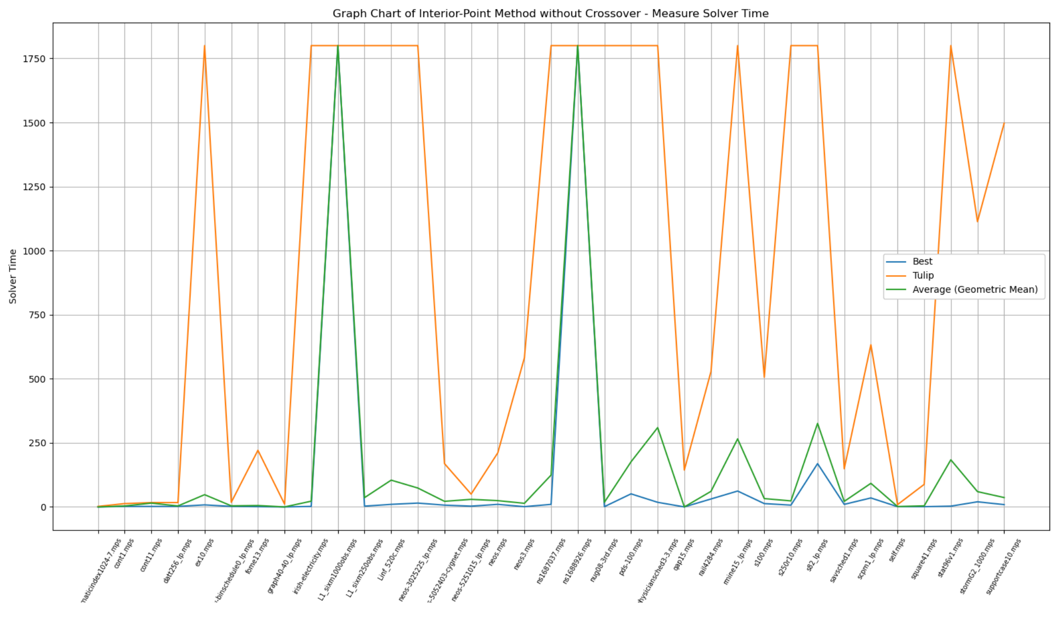

Figure 24: Tulip - Performance chart of Case 5 against baselines

4.2.2 Performance of Algorithms

Next it was attempted to find the best performing algorithm out of the two methods that were used across all solvers, Simplex Methods and Interior-Point Methods.

To achieve that, the analysis was divided in two sections. Firstly, the default settings of the two best and only actual concurrent solvers in the benchmark will be examined, GuRoBi and CPLEX. Even though Xpress and Mosek are also theoretically concurrent solvers, in practice both solvers only used the Dual Simplex and an Interior-Point method. Hence, for GuRoBi and CPLEX the logs will be consulted to determine for each problem, the algorithms that terminated faster, when the solver was initiated in a concurrent mode.

Secondly, a non-solver dependent analysis will follow. Taking the reported results from Cases 2,3,4, the best performed solver time in Cases 2 and 3 will represent the Simplex methods and Case 4 will represent Interior-Point Methods, regardless of which solver achieved it. Case 5 is excluded from this analysis, as removing crossover is providing an advantage to Interior-Point methods. Firstly, Simplex methods' performance will consist of both Dual Simplex and Primal Simplex solver times, but then further analysis that compares Dual Simplex and Primal Simplex alone, against Interior-Point methods will follow.

Examining the logs of the concurrent solvers of GuRoBi and CPLEX, made clear that the Interior-Point method managed to solve more problems than the Simplex methods when initiated in parallel. Specifically, GuRoBi solved 22 problems with an Interior-Point method, 12 problems with Dual Simplex and 1 problem with Primal Simplex. Similarly, CPLEX solved 22 problems with an Interior-Point method, 9 problems with Dual Simplex, 3 problems with Primal Simplex and failed at one problem. The table containing which problems were solved by which algorithm with the concurrent solvers of CPLEX and GuRoBi is located in the appendix (Table 13).

Although this seems to suggest a superiority of Interior-Point Methods, running an analysis independent of these two solvers presents a different picture. It's reminded that the metrics that were drawn from Cases 2,3,4, were all run on a single thread. Firstly, the case were Primal and Dual Simplex performances across solvers are combined to create an "any Simplex" method is reported. In this case Simplex beats Interior-Point methods in 17 out of 35 problems and IPM is faster in 16 out of 35 problems, while there are also two ties. This is reflected on the performance profile in Figure 25.

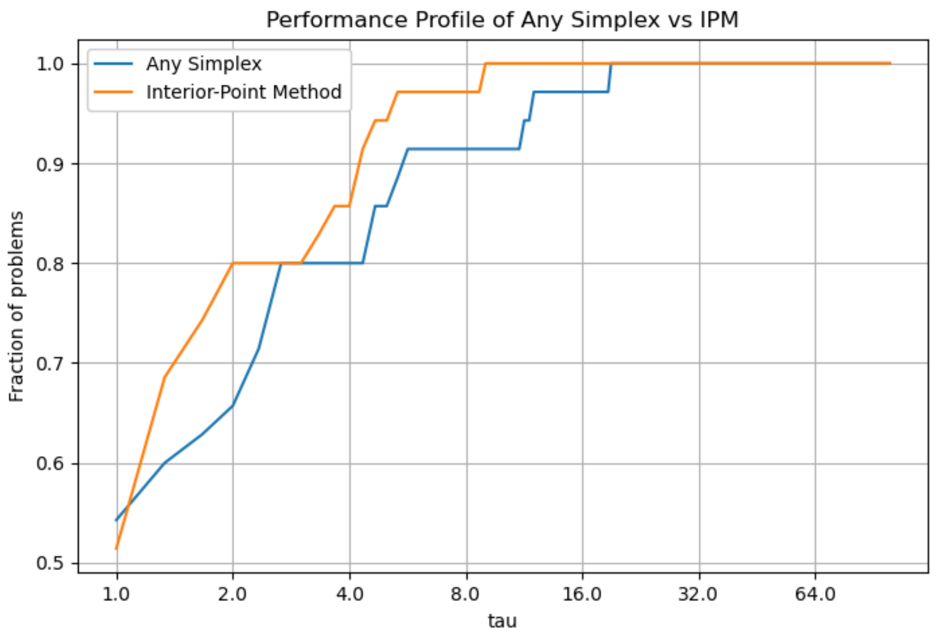

Figure 25: Any Simplex versus IPM - Performance Profile

Examining this performance profile, makes clear that the gap in performance is significantly smaller than what the results from the concurrent solvers suggested. However, even though the "Any Simplex" method manages to solve more problems faster, it is eventually surpassed in performance by Interior-Point Method solvers, as tau progresses. The differences however are minimal and could be unimportant, as the small advantage that IPM has could relate to problems that were relatively easy to solve for both methods or vice-versa.

A more significant gap in performance is noticed, if only the Dual Simplex optimisers combined are compared to the Interior-Point method optimisers. As displayed in the performance profile in Figure 26, here the Dual Simplex method is consistently below the chart drawn by the Interior-Point method. However, even though IPM seems to be definitely the winner, these differences are still not especially large. In total Interior-Point methods beat Dual Simplex in 18 problems, Dual Simplex beats IPM in 15 problems and there are also two ties.

A more dramatic case is presented in Figure 27, where the performance of the Primal Simplex method across solvers is presented against an Interior-Point Method. Primal Simplex solvers cumulatively only managed to solve 8 problems faster than all Interior-Point Method optimisers. This is reflected on the performance profile, along with the definite superiority of Interior-Point Method solvers against Primal Simplex solvers.

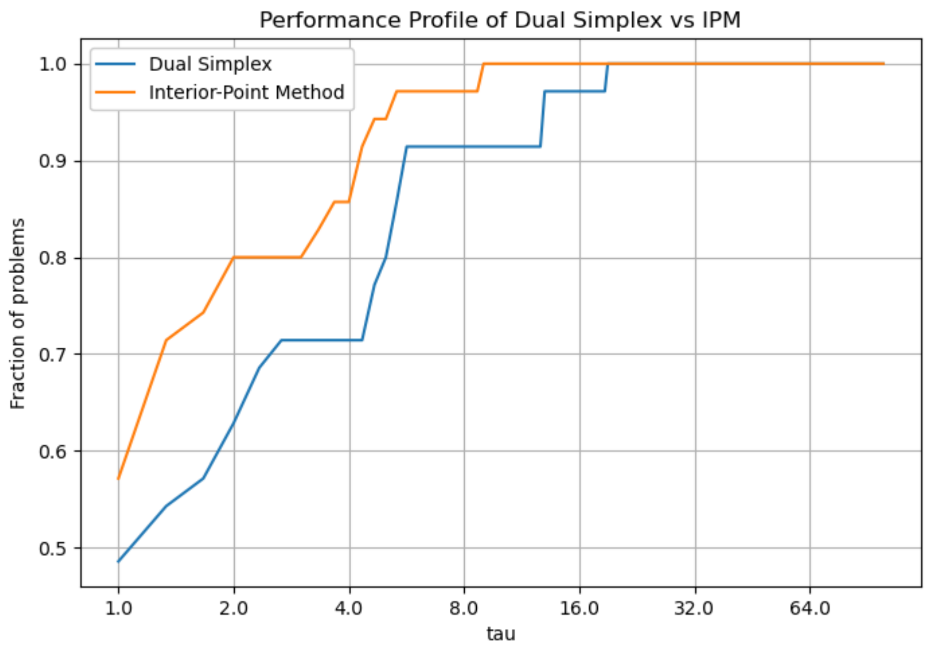

Figure 26: Dual Simplex versus IPM - Performance Profile

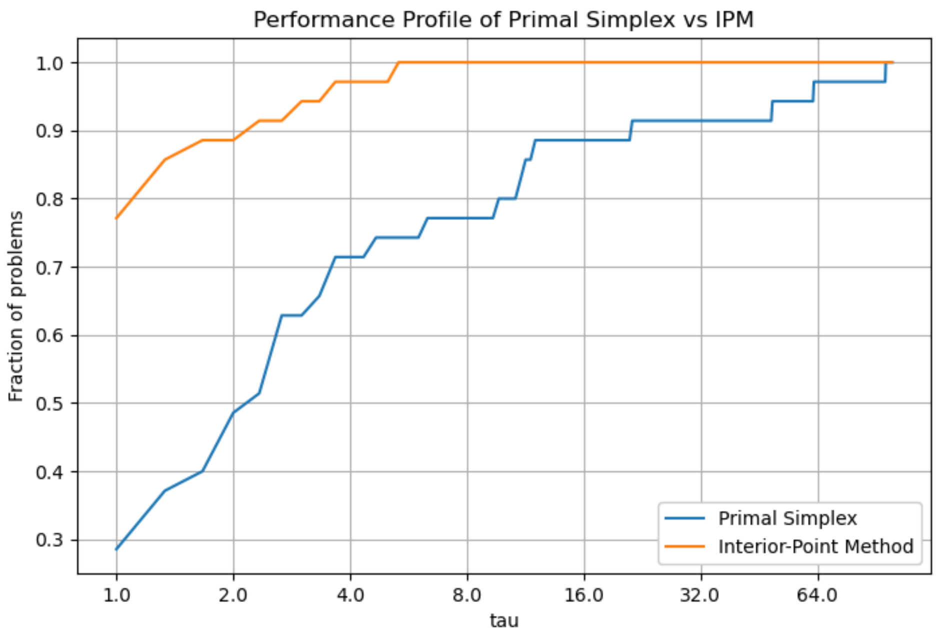

Figure 27: Primal Simplex versus IPM - Performance Profile

5 Conclusions

In this chapter, a brief summary of the main outcomes that were derived during the dissertation will be presented, as well as some suggestion for further research in the future, using this dissertation as a blueprint. It is noted that all conclusions that were drawn, concerning solver behaviour, relate to performance on the selected test set. Although the selection of the test set has been careful, it is still a relatively small list of instances that can only provide insight. Hence, it would be a mistake to generalise these outcomes outside the scope of this dissertation.

5.1 Main Outcomes

Firstly, it is the general belief of the research team that the advantages of using Julia and JuMP for solver benchmarking have been proven. JuMP specifically has been an excellent tool for solver interaction and influencing of solver parameters. The experimental design of this dissertation has been significantly facilitated by the user-friendliness and robustness of JuMP. Especially for benchmarking LP problems, where the universally accepted MPS format is widely used, serializing and automating the process has been very efficient and straight-forward.

The metrics that were recorded during benchmarking were the solver time and Julia Time. However, Julia Time did not prove to be a significant metric and can be probably omitted in future benchmarks with Julia, as the divergences between solver time and total problem execution have been minimal. However, even this metric provided some insight in the case of the HiGHS solver, as it was the only time that this metric was highly different than the recorded solver time. This helped in locating unexpected problems in this solver.

Regarding solver performance, it is evident from the reported metrics that the best overall performer in terms of solver time is GuRoBI. GuRoBi's default settings provided the best recorded metrics, when utilizing maximum available threads and utilizing many different methods concurrently. GuRobi also has the best performing Dual Simplex algorithm with usage of a single thread as a forced solver parameter. CPLEX has the best Primal Simplex optimiser and Xpress has the best Interior-Point method optimiser, both with restricted usage of maximum one thread.

In general, these three commercial solvers seem to dominate the examined cases for the selected test set. However, the remaining commercial solver, Mosek, manages to outperform Xpress on its default settings, hence it's performance can not be ignored overall. The reason behind the unexpected performance of Xpress, on its default settings, is assumed to be a bad selection strategy of which method to use, as the Interior-Point method of Xpress seems to be almost definitively superior than its Dual Simplex method. Mosek on the other side proved to be a significantly better Interior-Point method optimiser, than a Simplex optimiser, with its Simplex performance being closer to performance exhibited by open-source solvers.

Arguably, best performance from the examined open-source solvers, has been exhibited by Clp. Clp managed to take advantage of the Primal Simplex and Dual Simplex optimisers that it has and managed to solve more problems than the other open-source solvers and relatively faster. Clp's Dual Simplex method even performed better, on average than a commercial solver (Mosek). HiGHS solver seems to be burdened by issues of compatibility between the solver and Julia, as the performance that was exhibited inside Julia was not consistent with performance recorded when initiated outside Julia. Finally, Tulip showed some impressive performance, with regards to speed, on part of the test set, but overall it has issues that relate to rounding/numerical errors and early terminations to overcome, as it was the solver that failed in the most problems.

Drawing conclusions regarding the better algorithm is even harder to do, even when restricted to this specific test set. Interior-Point methods seem to take advantage of parallelisation better than Simplex methods and exhibit better performance on concurrent and multi-threaded solvers, by a significant margin. However, single threaded performance of Simplex Methods challenges Interior-Point methods very closely. In fact, the best recorded solver time across all solvers for every problem was more commonly found by a Simplex method, primal or dual, although by a small margin. Differentiating between primal and Dual Simplex shows that Interior-Point methods manage to solve more problems faster than the Dual Simplex, however not significantly more. On the other hand, Primal Simplex is heavily outscored by Interior-Point methods, on the selected test set.

5.2 Suggestions for Further Research

The first area that could be explored in future research concerns discrepancies in performance that JuMP may exhibit, compared to native environments of different solvers. This is reinforced by the discrepancies found in the HiGHS solver between solver and Julia interaction, but divergences could apply to more solvers, even if significantly smaller. It would be interesting to record performance outside of JuMP and observe if using JuMP sacrifices performance by a little or at all, as this could motivate extended usage of this modelling language.

Apart from JuMP, there are more modelling languages, distributed as standalone software, e.g AMPL, or embedded through a programming language, e.g. Pyomo in Python. Benchmarking outside the scope of finding the best solver, but to actually find the best modelling language in terms of speed could also prove useful. It's true that speed is not the main reason behind the selection of a modelling language, with main factors being compatibility with solvers and user-friendliness. However, performance can also be a factor that influences a significant part of the scientific community to switch to a new modelling language.

Additionally, one key conclusion that became apparent from reporting literature on benchmarking is that there is no general consensus regarding the best way to visualise and present the reported results. Performance profiles provide some valuable insight, but a lot of information seems to be lost and understated as the number of solvers grows. Additionally, measures like the geometric mean are used to present a more clear picture than the arithmetic mean, but there is still debate, about their accuracy. A comprehensive study around the proper metrics that can be used for the reporting of results in benchmarking can be of great value for interested researchers.

When it comes to issues relating closer to solver benchmarking the natural next step is to introduce more variety in the selected test set. This can be mainly broken down to two parts. Firstly, including more LP problems that have different characteristics is definitely a first step. For the purposes of this dissertation, only easy problems were used, i.e., problems that can be solved by at least one solver, within one hour. Including hard problems is a logical step to follow. Additionally, specialising the test set in denser or sparser LP problems could also provide some useful insights. Secondly, moving entirely out of LP problems and expanding benchmarking in Julia to MILP, SOCP, SDP and QP problems is definitely a very interesting avenue to follow, for further research.

References

[1] Jeff Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1 (2017), pp. 65–98. URL: https://doi.org/10.1137/141000671.

[2] John Russell. "Julia Update: Adoption Keeps Climbing; Is It a Python Challenger?" In: *HPCWire* (2021). URL: https://www.hpcwire.com/2021/01/13/julia-update-adoption-keeps-climbing-is-it-a-python-challenger/.

[3] Iain Dunning, Joey Huchette, and Miles Lubin. "JuMP: A Modeling Language for Mathematical Optimization". In: *SIAM Review* 59.2 (2017), pp. 295–320. DOI: 10.1137/15M1020575.

[4] Vahid Beiranvand, Warren Hare, and Yves Lucet. "Best practices for comparing optimization algorithms". In: *Optimization and Engineering* 18.4 (2017), pp. 815–848. ISSN: 1573-2924. DOI: 10.1007/s11081-017-9366-1. URL: https://doi.org/10.1007/s11081-017-9366-1.

[5] Elizabeth D. Dolan and Jorge J. Moré. "Benchmarking optimization software with performance profiles". In: *Mathematical Programming* 91.2 (2002), pp. 201–213. ISSN: 1436-4646. DOI: 10.1007/s101070100263. URL: https://doi.org/10.1007/s101070100263.

[6] Hans Mittelmann and P. Spellucci. "Decision Tree for Optimization Software". In: (2005). URL: http://plato.asu.edu/guide.html.

[7] Hans D. Mittelmann. "Benchmarking Optimization Software - a (Hi)Story". In: *SN Operations Research Forum* 1.1 (2020), p. 2. ISSN: 2662-2556. DOI: 10.1007/s43069-020-0002-0. URL: https://doi.org/10.1007/s43069-020-0002-0.

[8] B Meindl and Matthias Templ. "Analysis of commercial and free and open source solvers for linear optimization problems". In: (Aug. 2013).

[9] Josef Jablonsky. "Benchmarks for Current Linear and Mixed Integer Optimization Solvers". In: *Acta Universitatis Agriculturae et Silviculturae Mendelianae Brunensis* 63 (Jan. 2015), pp. 1923–1928. DOI: 10.11118/201563061923.

[10] Rimmi Anand, Divya Aggarwal, and Vijay Kumar. "A comparative analysis of optimization solvers". In: *Journal of Statistics and Management Systems* 20.4 (2017), pp. 623–635. DOI: 10.1080/09720510.2017.1395182. URL: https://doi.org/10.1080/09720510.2017.1395182.

[11] Nicholas Gould and Jennifer Scott. "A Note on Performance Profiles for Benchmarking Software". In: *ACM Transactions on Mathematical Software* 43 (Aug. 2016), pp. 1–5. DOI: 10.1145/2950048.

[12] John Mashey. "War of the benchmark means: time for a truce." In: *SIGARCH Computer Architecture News* 32 (Jan. 2004), pp. 1–14.

[13] Oliver Bastert and Timo Berthold. *A Note on fairbenchmarking*. https://community.fico.com/s/blog-post/a5Q2E000000Dt0JUAS/fico1421.

[14] Koch et al. "MIPLIB 2010". In: *Mathematical Programming Computation* 3.2 (2011), p. 103. ISSN: 1867-2957. DOI: 10.1007/s12532-011-0025-9. URL: https://doi.org/10.1007/s12532-011-0025-9.

[15] Warren Hare and Claudia Sagastizabal. "Benchmark of some nonsmooth optimization solvers for computing nonconvex proximal points". In: *Pacific Journal of Optimization* 2 (Mar. 2006), pp. 545–573.

[16] Patrick K. Mogensen and Asbjørn N. Riseth. "Optim: A mathematical optimization package for Julia". In: *Journal of Open Source Software* 3.24 (2018), p. 615. DOI: 10.21105/joss.00615. URL: https://doi.org/10.21105/joss.00615.

[17] J. Revels, M. Lubin, and T. Papamarkou. "Forward-Mode Automatic Differentiation in Julia". In: *arXiv:1607.07892 [cs.MS]* (2016). URL: https://arxiv.org/abs/1607.07892.

[18] Benoît Legat et al. *MathOptInterface: a data structure for mathematical optimization problems*. 2020. arXiv: 2002.03447 [math.OC]. URL: https://arxiv.org/abs/2002.03447.

[19] Q. Huangfu and J. A. J. Hall. "Parallelizing the dual revised simplex method". In: *Mathematical Programming Computation* 10.1 (2018), pp. 119–142. ISSN: 1867-2957. DOI: 10.1007/s12532-017-0130-5. URL: https://doi.org/10.1007/s12532-017-0130-5.

[20] Mathieu Tanneau, Miguel F. Anjos, and Andrea Lodi. "Design and implementation of a modular interior-point solver for linear optimization". en. In: *Mathematical Programming Computation* (Feb. 2021). ISSN: 1867-2957. DOI: 10.1007/s12532-020-00200-8. URL: https://doi.org/10.1007/s12532-020-00200-8 (visited on 03/07/2021).

[21] Tibor Illés and Tamás Terlaky. "Pivot versus interior point methods: Pros and cons". In: *European Journal of Operational Research* 140.2 (2002), pp. 170–190. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(02)00061-9. URL: https://www.sciencedirect.com/science/article/pii/S0377221702000619.

[22] V. Klee and G. Minty. "HOW GOOD IS THE SIMPLEX ALGORITHM". In: 1970.

[23] K. Borgwardt. "The Simplex Method: A Probabilistic Analysis". In: 1986.

[24] Jacek Gondzio. "Interior point methods 25 years later". In: *European Journal of Operational Research* 218.3 (2012), pp. 587–601. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2011.09.017. URL: https://www.sciencedirect.com/science/article/pii/S0377221711008204.

[25] John Fearnley and Rahul Savani. "The Complexity of the Simplex Method". In: *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*. STOC '15. Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 201–208. ISBN: 9781450335362. DOI: 10.1145/2746539.2746558. URL: https://doi.org/10.1145/2746539.2746558.

[26] Dongdong Ge et al. *From an Interior Point to a Corner Point: Smart Crossover*. 2021. arXiv: 2102.09420 [math.OC].

[27] István Maros and Csaba Mészáros. "A repository of convex quadratic programming problems". In: *Optimization Methods and Software* 11.1-4 (1999), pp. 671–681. DOI: 10.1080/10556789908805768. eprint: https://doi.org/10.1080/10556789908805768. URL: https://doi.org/10.1080/10556789908805768.

[28] Ilias Parmaksizoglou. *Julia Benchmarking*. https://github.com/iparmax/uoe_julia_benchmarking.

[29] Oscar Dowson. *JuMP-dev*. https://github.com/jump-dev.

Appendix

Problems	**HiGHS**
L1_sixm1000obs.mps	Loading Error
L1_sixm250obs.mps	Loading Error
Linf_520c.mps	Loading Error
rail4284.mps	Loading Error
s82_lp.mps	Loading Error
savsched1.mps	Loading Error
stormG2_1000.mps	Loading Error

Table 12: Problems where HiGHS solver never started

Problems	**GuRoBi**	**CPLEX**
chromaticindex1024-7.mps	Dual Simplex	Primal Simplex
cont1.mps	IPM	IPM
cont11.mps	IPM	IPM
datt256_lp.mps	Dual Simplex	IPM
ex10.mps	Dual Simplex	IPM
fhnw-binschedule0_lp.mps	Primal Simplex	Primal Simplex
fome13.mps	IPM	IPM
graph40-40_lp.mps	IPM	Dual Simplex
irish-electricity.mps	IPM	Fail
L1_sixm1000obs.mps	Dual Simplex	Dual Simplex
L1_sixm250obs.mps	Dual Simplex	Dual Simplex
Linf_520c.mps	IPM	IPM
neos-3025225_lp.mps	IPM	IPM
neos-5052403-cygnet.mps	IPM	IPM
neos-5251015_lp.mps	IPM	IPM
neos.mps	IPM	IPM
neos3.mps	Dual Simplex	Dual Simplex
ns1687037.mps	IPM	IPM
ns1688926.mps	Dual Simplex	Dual Simplex
nug08-3rd.mps	Dual Simplex	Dual Simplex
pds-100.mps	Dual Simplex	Dual Simplex
physiciansched3-3.mps	IPM	Dual Simplex
qap15.mps	IPM	IPM
rail4284.mps	IPM	IPM
rmine15_lp.mps	IPM	IPM
s100.mps	IPM	IPM
s250r10.mps	IPM	IPM
s82_lp.mps	IPM	IPM
savsched1.mps	IPM	Primal Simplex
scpm1_lp.mps	IPM	IPM
self.mps	IPM	IPM
square41.mps	Dual Simplex	Dual Simplex
stat96v1.mps	Dual Simplex	IPM
stormG2_1000.mps	Dual Simplex	IPM
supportcase10.mps	IPM	IPM

Table 13: Fastest Algorithms on Concurrent solver for CPLEX and GuRoBi