Binary Integer Programming in associative data models

Nils Fagerberg, Daniel Odenbrand
Binary Integer Programming in associative data models

Nils Fagerberg & Daniel Odenbrand

March 2016

Master’s thesis work carried out at Qlik, Lund.

Supervisors: Thore Husfeldt
              José Díaz López

Examiner: Krzysztof Kuchcinski
Abstract

The data visualization softwares Qlikview and Qlik Sense are based on an associative data model, and this thesis analyzes different tools and methods for solving 0-1 integer programs as well as examines their applicability to the computational engine behind these softwares. The first parts are dedicated to a theoretical background on mathematical optimization, linear programming and Qlik’s implementation of the associative data model. We tested two optimization packages, Gurobi and Microsoft Solver Foundation alongside an enumeration method we implemented in an external extension communicating with a Qlik Sense application via network. The test results showed some promise in terms of number of operations, so we created an implementation closer to the engine. While faster, using this implementation we were still unable to solve some of our more difficult problems within a reasonable time frame. An alternative heuristic for node traversal was also considered, suspecting that this would be more efficient on a different class of problems. In practice one of the heuristics, best-first search, was faster in general but we believe that it benefited from the data being autogenerated and that more optimization can be made to the alternative search method, in particular when augmenting the candidate solution. In the future, we do not believe that implicit enumeration will replace the traditional methods of solving 0-1 integer programs since Gurobi still performed the best on average, but it may have some exciting applications on a specific type of problems where, once they reach a certain size, traditional models would not stand a chance.

Keywords: Integer Programming, Gurobi, Microsoft Solver Foundation, Simplex Method, Qlik Sense, Implicit Enumeration
Acknowledgements

Special acknowledgements go to:

- José Díaz López, Lead Scientist at Qlik, Lund. Supervisor at Qlik.
- Thore Husfeldt, Ph.D, Professor at Faculty of Engineering, Lund University. Supervisor at LTH.
- Ola Nilsson, Sr Manager R&D at Qlik. Resource- and administration manager at Qlik.
- Lars Skage, Quality Analyst at Qlik. Miscellaneous support.

This thesis was created as an initiative of José Díaz López. The idea was to implement a feature in Qlik’s products, Qlikview and Qlik Sense, which optimizes an expression in regard to the underlying database, both user-provided input. Most of the work and studies was be done at Qlik in Lund and José was responsible for supervision and guidance. Thore Husfeldt made sure the project moved forward and provided opinions and feedback on the methods investigated. Ola Nilsson provided administrative support and guidance whenever José was out of office. An honorable mention goes to Lars Skage who helped us a great deal with the Qlik Load Script language so that we could run our benchmarks without running into major problems.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>2</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>3</td>
</tr>
<tr>
<td>Introduction</td>
<td>8</td>
</tr>
<tr>
<td><strong>1 An introduction to optimization and Integer Programming</strong></td>
<td>13</td>
</tr>
<tr>
<td>1.1 Optimization problems</td>
<td>13</td>
</tr>
<tr>
<td>1.2 Linear Programming and Integer (Linear) Programming</td>
<td>14</td>
</tr>
<tr>
<td>1.3 An example of a 0-1 Integer Program</td>
<td>14</td>
</tr>
<tr>
<td><strong>2 Algorithms for Linear- and Integer Programming</strong></td>
<td>17</td>
</tr>
<tr>
<td>2.1 Problem relaxations</td>
<td>17</td>
</tr>
<tr>
<td>2.2 The Simplex Method</td>
<td>17</td>
</tr>
<tr>
<td>2.3 Branch and Bound</td>
<td>19</td>
</tr>
<tr>
<td>2.3.1 Branching</td>
<td>20</td>
</tr>
<tr>
<td>2.3.2 Bounding</td>
<td>21</td>
</tr>
<tr>
<td>2.3.3 Cutting Planes</td>
<td>21</td>
</tr>
<tr>
<td>2.4 Implicit Enumeration</td>
<td>22</td>
</tr>
<tr>
<td>2.4.1 The algorithm</td>
<td>24</td>
</tr>
<tr>
<td>2.4.2 Improvements</td>
<td>24</td>
</tr>
<tr>
<td>2.4.3 Alternative search heuristic and economic variable representation</td>
<td>25</td>
</tr>
<tr>
<td>2.4.4 Representation</td>
<td>25</td>
</tr>
<tr>
<td>2.4.5 Fathoming and node ordering</td>
<td>26</td>
</tr>
<tr>
<td><strong>3 Related work</strong></td>
<td>27</td>
</tr>
<tr>
<td>3.1 Basics of Integer Programming</td>
<td>27</td>
</tr>
<tr>
<td>3.2 The Complexity of Integer Programming</td>
<td>27</td>
</tr>
<tr>
<td>3.3 Improvements</td>
<td>28</td>
</tr>
<tr>
<td>3.3.1 Evaluating the Impact of AND/OR Search on 0-1 Integer Linear Programming</td>
<td>28</td>
</tr>
</tbody>
</table>
## 3.3.2 Infeasibility Pruning ................................................. 28
## 3.4 Other takes on solving 0-1 ILPs ............................... 29
### 3.4.1 The Reduce and Optimize Approach ...................... 29
### 3.4.2 An algorithm for large scale 0-1 Integer Programming with application to airline crew scheduling .................. 29
### 3.4.3 Pivot-and-Fix; A Mixed Integer Programming Primal Heuristic ......................... 29
## 3.5 Linear Programming in Database ............................... 30
## 3.6 Other applications of ILPs ......................................... 30

### 4 Introducing the basics of Qlik’s internal structure ........ 31
#### 4.1 Loading data from a database source in Qlik ............... 31
#### 4.2 Qlik Internal Database ........................................... 31
#### 4.3 Inference Machine ................................................ 33
#### 4.4 From QIDB to doc state ........................................... 33
#### 4.5 The Hypercube ..................................................... 35
#### 4.6 The associative model in Qlik .................................. 35

### 5 Method ................................................................. 37
#### 5.1 Problem structure ................................................ 37
#### 5.2 Microsoft Solver Foundation ................................. 38
##### 5.2.1 Setting up the model ....................................... 39
##### 5.2.2 Results ...................................................... 39
#### 5.3 Gurobi ............................................................ 39
##### 5.3.1 Setting up the model ....................................... 41
##### 5.3.2 Results ...................................................... 41
#### 5.4 A first attempt at Implicit Enumeration ...................... 41
##### 5.4.1 Best-first search ............................................ 43
##### 5.4.2 Setting up the model ....................................... 44
##### 5.4.3 Results ...................................................... 44
#### 5.5 A first analysis ................................................... 45
#### 5.6 Implementing Implicit Enumeration in the QIX engine .... 48
##### 5.6.1 Qlik configurations ......................................... 49
##### 5.6.2 Implementation details .................................... 51
##### 5.6.3 Results ...................................................... 51
##### 5.6.4 Proving a point .............................................. 52
#### 5.7 Another method for Implicit Enumeration in the QIX engine 56
##### 5.7.1 Implementation details .................................... 56
##### 5.7.2 Results ...................................................... 59
#### 5.8 A comparison between the different QIX implementations 60

### 6 Discussion ............................................................ 67
#### 6.1 Measurement of success ......................................... 67
#### 6.2 Examining the results ............................................ 68
#### 6.3 The hash used with the best-first search heuristic .......... 68
#### 6.4 Number of branches as a unit of measurement .............. 68
#### 6.5 Fetching data from the hypercube ............................. 69
#### 6.6 The different problem difficulties ............................ 70
## CONTENTS

6.7 The randomness of the problems ........................................ 70
6.8 The smaller testcases .................................................. 70
6.9 Conclusions .................................................................. 71
   6.9.1 LP Solvers .......................................................... 72
   6.9.2 Enumeration ....................................................... 72

7 One qlick away from the future ........................................... 75
   7.1 Exploiting data dependencies ........................................ 75
   7.2 Don’t build the hypercube ........................................... 76
   7.3 AND/OR Search Tree ................................................ 76
      7.3.1 The idea ......................................................... 77
      7.3.2 The benefits .................................................. 78

Work Process .................................................................. 79

Bibliography .................................................................... 83
Introduction

Background

Qlik is a company which started in 1993 in Lund, Sweden. Their initial product - Qlikview - was designed for business insight by helping users present data visualizations that were easier to understand. A key feature is the color-coding system - selected values are highlighted in green, linked values are highlighted in white and excluded values are shaded out in grey [1].

In 2014 Qlik released its first version of Qlik Sense which is a product designed to support the user’s need to visualize and understand data. Just like in Qlikview, it is possible to distribute complete visualizations to the user, but at the same time the user may as well read local data or choose to visualize existing data in another way than already presented in the distributed visualization [2].

Compared to other existing data visualization tools, Qlikview allows you to play around with the data to find out what questions you need to ask, rather than knowing what question to ask and then receive the corresponding data. For example using regular SQL will always require a request from the user, and the user will receive only the relevant data and nothing else. In Qlik’s products you can play around with the entire database at once.

Motivation

When loading a database containing data of interest into Qlik’s products the database is read and stored in internal memory. Qlik’s engine, the underlying back-end code that makes all calculations (in the future, we will refer to it as QIX engine), then masks the internal memory database to a table of ones and zeros to see which posts are currently active and thus which data to be marked as green, white or grey. If you click (classic Qlik-pun for "click") on a post, it will turn green and the engine will infer what other values are to be colored white or grey. The inference is possible due to the QIX engine storing the data in an associative model rather than a relational model which is what most databases use today.
Suppose that the user could also specify a mathematical expression in addition to the database with the goal of optimizing this expression over a subset of the data model, that is, finding out what qlicks to make in order to optimize the given expression. This is a task mixing mathematical analysis with computer science, where one needs understanding of the problem definition and the methods for reaching the solution to be able to compare, analyze and improve different algorithms for solving these types of problems as well as a deep knowledge of memory management and data structures to achieve a robust and efficient solution. One also needs deep understanding of the QIX engine which requires reading of source code.

The specific optimization problem that this thesis is about to examine, 0-1 Integer Programming, is in fact NP-complete (it is possible to verify a given solution for this problem in polynomial time) and thus there does not yet exist an efficient algorithm for these problems. Hopefully we can let the QIX engine perform a lot of the work and exploit the fact that the problem is over the domain \{0, 1\}^n since you can only either select a post or not. Thus, investigating the relations between the bit-mask in the engine, the internal memory database and the optimization problem in question qualifies as a Master’s Thesis at Lund University’s Faculty of Engineering.

Goals

The main goal is to create an algorithm and implement it in the QIX engine which finds an optimal subset of the database according to a given expression. To reach this goal we will need to create relevant examples, generate data for them and compare existing 0-1 Integer Programming solvers to our own algorithm to find its strengths and weaknesses. We do not aspire to find a polynomial time algorithm for solving general 0-1 Integer Programs but instead find strengths in the QIX engine and use special structures that we can exploit in our examples.

Concrete sub-goals are:

- Choose a reasonable algorithm and generate and run tests to motivate further development and optimizations in data structures used.
- Implement the algorithm in the QIX engine and optimize it. Compare the implemented algorithm to commercial optimizers.
- Compare the algorithm to alternative solutions.
- Present future developments in this area. This step is especially important since time will most likely prevent us from doing all possible optimizations.

Disposition

The thesis will be divided into several chapters.

Chapters one through three present the underlying theory regarding optimization, linear programming and integer programming needed for this thesis. In chapter three specifically the reader can find further reading about related work.
Chapter four contains information regarding the underlying structures of Qlik’s products and is essential for understanding the selection of algorithm.

In chapter five we run tests and compare current commercial solvers to our own implementations. We try to analyze the results and find the pros and cons of our implementation and how we can take advantage of this.

Then, in chapter six, we draw our conclusions from this thesis and lastly we present future work in chapter seven.

Contributions

The work has not been separated in the sense that we have worked physically next to each other. Therefore both authors have reviewed and approved all content.
Chapter 1

An introduction to optimization and Integer Programming

Since this thesis is focusing on mathematical optimization, this chapter aims to explain to the reader what kind of problems we are trying to solve as well as introduce some popular methods currently used in the field.

1.1 Optimization problems

When solving an optimization problem, we are concerned with finding a best solution. Given an objective function $F$ defined on a set $\Omega$ with range $\Phi$, an element in $\Omega$ corresponding to such a solution can be denoted $\text{argopt}$. Then we arrive at the following definition:

**Definition 1.1.** An optimization problem is to find $\tilde{x}$ such that:

$$\tilde{x} = \text{argopt}_{x \in \Omega} F(x), \quad F : \Omega \rightarrow \Phi. \quad (1.1)$$

In the scope of this thesis, we will almost exclusively encounter functions having domains consisting of certain subsets of $\mathbb{R}^n$ and ranges $\mathbb{R}$. The best solution for a problem with such a function can then be interpreted as finding a minimum or maximum. We only require a definition for one of these, however, since a maximization problem can easily be rewritten as minimization by inverting the signs of both the objective function and the result (that is, "$\min F = -\max (-F)$").

When performing optimization in the real world we can usually constrain ourselves to a tighter definition to fit our problem formulation and the type of desired solutions, but a general definition is needed before we study special cases.
1.2 Linear Programming and Integer (Linear) Programming

Definition 1.2. A Linear Program (LP) is an optimization problem that can be written on the form

\[
\begin{align*}
\text{maximize} & \quad c^T x \\
\text{s.t.} & \quad Ax \leq b, \\
& \quad x_i \geq 0 \quad \text{for} \quad i = 1 \ldots n, \\
& \quad x \in \mathbb{R}^n,
\end{align*}
\]  

where \(c\) and \(b\) are column vectors containing \(n\) and \(m\) elements respectively, where \(n\) is the number of state variables (i.e. the size of \(x\)) and \(m\) is the number of constraints. Similarly, \(A\) is an \(m \times n\) matrix. Elements of \(c\), \(b\) and \(A\) are real numbers. An LP without solutions on the domain is called infeasible, while a particular assignment of variables \(\tilde{x}\) satisfying the constraints \(Ax \leq b\) is called a feasible solution to the LP.

When solving problems of this kind, stating the problem in a way that culminates into solving \(Ax = b\) rather than the inequality in 1.2 is usually preferred. This process is called standardization and is in practice done partly by introducing extra variables called slack variables representing the discrepancy within the inequality. For example, a constraint

\[x_1 + x_2 \leq 15\]

is rewritten by introducing the slack variable \(s := 15 - x_1 - x_2\), thus becoming

\[x_1 + x_2 + s = 15,\]

where, naturally, \(s \geq 0\). Depending on the problem, there can be more to the standardization process. See [4] for a more detailed explanation.

Definition 1.3. Linear Programming is a set of methods for solving Linear Programs.

Definition 1.4. Integer Programming is a special case of Linear Programming where extra constraints are added; specifically several of the variables are constrained to take integer values. The formulation of an Integer Program (IP) is almost identical to that of a Linear Program in 1.2 but forcing the vector \(x\) to contain only integer values i.e. replacing the constraint \(x \in \mathbb{R}^n\) with \(x \in \mathbb{Z}^n\). Methods for solving these are called Pure Integer Programming. If \(x\) only takes the values 0 and 1 the methods are called 0-1 Integer Programming, and if one or more element in \(x\) is constrained to be integer, the methods are called Mixed Integer Programming.

1.3 An example of a 0-1 Integer Program

To make our arguments more illustrative and provide more easily interpreted results, we will use the "warehouse problem" as a basis for discussion. The warehouse problem can be stated as follows: a number of warehouses with known delivery times stocks products
that are divided into distinct types. A customer places an order on a specific quantity of some of the products, and the company with the warehouses must match the demand while minimizing total delivery time. This can be modelled as a 0-1 IP as follows:

\[
\text{minimize } \quad c^T x \\
\text{s.t. } \quad Ax \geq b , \\
\text{and } \quad x \in \{0, 1\}^n
\]

with the following notations:

- \( n \) is the number of warehouses.
- \( m \) is the number of products included in the order, i.e. the number of constraints.
- \( c_i \) is the delivery time for warehouse \( i \). \( n \times 1 \) vector. In this case all \( c_i \geq 0 \) (since we cannot have a negative delivery time).
- \( x_i = 1 \) if we elect to send products from warehouse \( i \) and 0 otherwise. \( n \times 1 \) vector.
- \( a_{ji} \) is the quantity of product \( j \) stored in warehouse \( i \). \( m \times n \) matrix.
- \( b_i \) is the demand for product \( i \). \( m \times 1 \) vector.

Attempting to model this as an LP will fail in reality since we cannot deliver a fraction of the stock in a warehouse in a fraction of the time, we have to travel the entire distance regardless of how much we wish to transport. A regular IP model would also fail since it is likely to find a solution where the company should send multiples of the stock in a warehouse from that same warehouse, which is clearly impossible.

Additional constraints can also be added to represent some desirable qualities of a solution. For example, if we wish to involve no more than \( l \) warehouses, we can add the constraint \( \sum_{k=1}^{n} x_k \leq l \). In a similar fashion, by adding constraints we can force the inclusion or exclusion of certain combinations of warehouses.

The warehouse problem will later be used for testing methods and benchmarking.
1. An introduction to optimization and Integer Programming
Chapter 2
Algorithms for Linear- and Integer Programming

Since many problems can be formulated as LPs the number of methods developed to solve this type of problems are numerous. This chapter aims to introduce the most common approaches as well as others used and mentioned in this thesis.

2.1 Problem relaxations

A general approach for solving IPs is to first ignore some or all integrality constraints. Specifically, we want to solve the linear relaxation of the IP.

Definition 2.1. For each IP, we can construct an LP with the same coefficients, objective function and constraints omitting those concerning integrality. Such an LP is called a linear relaxation of the IP.

If we do not arrive at an integral solution immediately we can introduce new constraints in the system which makes the current nonintegral solution infeasible. This approach requires the solving of an LP in each step.

Another approach is constraint relaxation, where we evaluate a solution in advance of examining its feasibility, rather than solving the problem without the integrality of the variables. This doesn’t require any Linear Program to be solved, but it is likely that we have to perform more operations in some instances since we get little to no indication of whether we are heading in the right direction or not.

2.2 The Simplex Method

The simplex method is a popular algorithm used to solve linear programs. The general idea of this algorithm is to find a simple solution satisfying all constraints. Then this solution
is altered iteratively by doing a sequence of pivot operations each improving the objective function. Note that our explanation of the simplex method is insufficient to solve every LP. We encourage readers to read more about the simplex method in [4] or most any other book on optimization, but more knowledge is not necessary to understand this thesis.

After standardizing an LP on the form

\[
\begin{align*}
\text{maximize} & \quad c^T x \\
\text{s.t.} & \quad Ax \leq b, \\
& \quad x_i \geq 0 \quad \text{for} \quad i = 1 \ldots n, \\
\text{and} & \quad x \in \mathbb{R}^n,
\end{align*}
\]

we should have a set \( s_j, i = 1 \ldots m \) of slack variables. Then, finding a set of variables satisfying the constraints is trivial - we simply set all original state variables to zero, giving the solution \( s_j = b_j \). Such a solution is called a basic solution, from the idea that \( s_1 \ldots s_m \) constitutes a basis in the sense that a solution can be written using a combination of only these variables. Initially, this results in the objective function getting the value zero which probably isn’t a good solution. To remedy this, we switch the variables in the basis in such a way that we arrive at a basic solution with a higher valued objective function. This is done by reiterating the following procedure:

1. Select a variable such that, when incremented, results in a higher valued objective function. If no such value can be found, terminate.

2. Increment it as much as possible without violating any constraints.

3. "Pivot" on the chosen variable (eliminate the chosen variable from the other equations using Gauss elimination on the row setting the upper bound of the incrementation).

   - This leads to a different representation of the equation system, one with a higher valued objective function for the "new" basic solution.

   - The method for obtaining a new basic solution can informally be described as "set the slack variable corresponding to the previously modified equation to 0 and set the recently incremented variable to whatever it was incremented by. The other variables remain the same."

4. Return to step 1.

In theory, any variable can be chosen as long as it hasn’t been chosen in a previous step. In practice, it is usually step-efficient to select the one with the largest coefficient in the objective function. A small example of the simplex method can be found below.

\[
\begin{align*}
\text{maximize} & \quad z = 2x_1 + 3x_2 \\
\text{subject to} & \quad x_1 + x_2 \leq 5 \\
& \quad x_2 \leq 3 \\
& \quad x_1, x_2 \geq 0.
\end{align*}
\]
2.3 Branch and Bound

To solve this using the simplex method, introduce slack variables $s_1$ and $s_2$ such that the equations become

\[
\begin{align*}
-z + 2x_1 + 3x_2 &= 0 \quad (2.1) \\
x_1 + x_2 + s_1 &= 5 \quad (2.2) \\
x_2 + s_2 &= 3 \quad (2.3) \\
x_1, x_2, s_1, s_2 &\geq 0. \quad (2.4)
\end{align*}
\]

The basic solution is now given by $x_1 = 0, x_2 = 0, s_1 = 5, s_2 = 3$, giving $z = 0$. Incrementing either of $x_1$ or $x_2$ results in a higher value of $z$. We arbitrarily choose one of them, $x_2$. Since we require all variables to be nonnegative, the first constraint is violated if we choose $x_2 > 5$ while the second is violated if we choose $x_2 > 3$. Let $x_2 = 3$ and pivot on $x_2$ i.e. eliminate $x_2$ from row 2.2 and 2.1 by subtracting one and three row 2.3 respectively:

\[
\begin{align*}
x_1 + x_2 + s_1 - x_2 - s_2 &= 5 - 3 \iff x_1 + s_1 - s_2 = 2 \\
x_2 + s_2 &= 3 \\
-z + 2x_1 + 3x_2 - 3x_2 - 3s_2 &= 0 - 9 \iff -z + 2x_1 - 3s_2 = -9.
\end{align*}
\]

This gives the new basic solution $x_1 = 0, x_2 = 3, s_1 = 2, s_2 = 0$ giving $z$ a value of 9, which is an improvement. We now say that $x_2$ has entered the basis while $s_2$ has left the basis. In the equation containing $z$, we observe that incrementing $x_1$ leads to a greater valued objective function. Choosing $x_1 > 2$ violates the first constraint while no possible value of $x_1$ can violate the second constraint since there is no dependency. Choosing $x_1 = 2$ and pivoting gives the system

\[
\begin{align*}
x_1 + s_1 - s_2 &= 2 \\
x_2 + s_2 &= 3 \\
-z + 2x_1 - 3s_2 - 2x_1 - 2s_1 - 2s_2 &= -9 - 4 \iff -z - 2s_1 - 5s_2 = -13,
\end{align*}
\]

having the basic solution $x_1 = 2, x_2 = 3, s_1 = 0, s_2 = 0$ giving $z$ the value 13. As we can see in the expression $-z - 2s_1 - 5s_2 = -13$, no changes of any of the variables can result in a greater value of $z$, so this solution must be optimal. Note that the simplex method only finds one optimal solution and only provided one exists; there may be anywhere between zero and infinitely many solutions.

2.3 Branch and Bound

The Branch and Bound algorithm is a method for solving integer programs using the fact that we are able to solve the corresponding linear relaxation. For simplicity we will use "a solution’s value" rather than "the value of the solution’s objective function" throughout this section. We will also assume that we are talking about a maximization problem. For minimization, the results are analogous if we replace upper bounds with lower bounds.

We will need the following definitions:
Definition 2.2. For a real number $\xi$, $\lfloor \xi \rfloor$ is the largest integer $\leq \xi$. Thus $\xi = \lfloor \xi \rfloor + \epsilon$, $0 \leq \epsilon < 1$.

Definition 2.3. For a real number $\xi$, $\lceil \xi \rceil$ is the smallest integer $\geq \xi$. Thus $\lceil \xi \rceil = \xi + \epsilon$, $0 \leq \epsilon < 1$.

The following theorem will also prove useful:

Theorem 2.1. For a solution to a linear programming problem $S^*$ with value $z^*$ and domain $D$, and the solution to the same linear programming problem $S, z$, over a domain $D_s \subset D$ we have that $z \leq z^*$.

Proof. Given $z^*$ as the optimal value over $D$ and $z$ as the optimal value over $D_s$. Assume that $z > z^*$. Since $D_s \subset D$ we have that a feasible solution over $D_s$ is also a feasible solution over $D$. Thus $z$ is a feasible solution over $D$. The assumption $z > z^*$ contradicts the fact that $z^*$ is the optimal value over $D$ which gives us that $z \leq z^*$ must hold.

Specifically, the value of an optimal solution to an IP cannot exceed that of its linear relaxation since $\mathbb{Z} \subset \mathbb{R}$. Also worthy of note is that if the coefficients in the objective function are all integer, the value of an integer solution must be integer since the sum of products of integers is also integer. In this case, after solving a linear relaxation of an IP we can impose an even tighter bound on an optimal integer solution than the theorem promised - if $z^*$ is the linear relaxation's optimal solution value, then the value of an integer solution can be no greater than $z = \lfloor z^* \rfloor$ since that is the largest integer $\leq z^*$.

The approach of the Branch and Bound algorithm is as follows:

1. Solve the linear relaxation of the integer program e.g. by using the simplex method. This provides us with an upper bound on our optimal solution according to theorem 2.1.

2. If the optimal variable assignment of the linear relaxation are all integer then an optimal solution has been found and the branch terminates.

3. Otherwise, pick a variable that is not integer in the solution to the linear relaxation and branch (see 2.3.1) on that variable, creating two new subproblems.

4. Select an active (see 2.3.2) subproblem with the current highest upper bound on its solution and return to step 1.

2.3.1 Branching

Given a solution, $S = \{x_1^*, \ldots, x_n^*\}$, to the linear relaxation of an integer programming problem $P$ choose a variable $x_i^*$ which is not integer. This variable exists at the branching step since otherwise the branch would have already terminated. Create two new subproblems $P_1$ and $P_2$ which are identical to $P$ except that one has the added constraint $x_k \leq \lfloor x_k^* \rfloor$ and the other has $x_k \geq \lceil x_k^* \rceil$. For both of the subproblems it will now hold that $S$ is not a solution since the value of $x_i^*$ doesn’t satisfy the new constraint. No other feasible integer solutions become infeasible since the constraints round to the nearest integers. Due to this, if there exists an optimal integer solution to $P$, it is guaranteed to be found along either of the two branches.
For each of the two new branches we calculate a new solution to each corresponding linear relaxation and apply the same argument. The values of these solutions will then serve as upper bound for solutions along their respective branch, more on that in section 2.3.2.

2.3.2 Bounding

While searching for a maximum integer solution with branch and bound, we are only interested in the nodes with the current highest upper bound on a solution on which we haven’t yet branched. Recall that theorem 2.1 guarantees that the LP finds a solution which is more optimal than either of the branches’ solutions. Nodes with this quality are called active subproblems. A subproblem is not active if it:

1. has been used for branching.
2. is infeasible.
3. cannot generate a better solution than the current best solution by branching.
4. has only integer values in its solution.

If neither of these hold, the subproblem is active. Subproblems falling under 1 or 2 can be disregarded entirely, but problems falling under 3 can become active again if the bounds on our solution are altered while branching.

Whenever the upper bound is changed, an inactive problem in 4 could possibly be identified as an optimal solution. That is, if the value $z$ of the solution to such a problem is greater than or equal to the current upper bound $z^*$ in each active subproblem (or in the case of integer coefficients in the objective function, is equal to $\lfloor z^* \rfloor$), this solution can be identified as a maximum integer solution.

2.3.3 Cutting Planes

Another algorithm for solving integer programs is called The Cutting Plane Algorithm. This technique has the same basic ideas as the ones explained in 2.3—solve a linear problem and, if integral optimum is not reached, introduce new constraints rendering the current solution infeasible. The approach is the following:

1. Solve the linear relaxation of the integer program e.g. by using the simplex method.
2. If the optimal basic variables are all integer then an optimum has been found and the algorithm terminates.
3. Generate a cut, i.e. add a constraint which is satisfied by all integer solutions to the problem but not by the current solution to the linear relaxation.
4. Add this new constraint and go to step 1.
A cutting method

The Cutting Plane Technique is based on the following construction of a "cutting plane": consider an integer program on the following form:

\[
\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{n} c_i x_i \\
\text{s.t.} & \quad A x \leq b, \\
& \quad x \in \mathbb{Z}^n, \\
& \quad x \geq 0,
\end{align*}
\]

(2.5)

with an optimal linear solution that, for a basic variable \(x_k\), has a constraint:

\[
x_k + \sum_{i=1}^{n+m} a_i x_i = b
\]

(2.6)

where \(b\) is not integer. We can then write each \(a_i = [a_i] + a'_i\), for some \(0 \leq a'_i < 1\), and \(b = [b] + b'\), for some \(0 \leq b' < 1\) and rearrange the equation such that the left hand side contains only integers:

\[
x_k + \sum_{i=1}^{n+m} ([a_i] + a'_i) x_i = [b] + b'
\]

(2.7)

\[
\Leftrightarrow \quad x_k + \sum_{i=1}^{n+m} [a_i] x_i - [b] = b' - \sum_{i=1}^{n+m} a'_i x_i.
\]

Since the left hand side is a sum of integers the right hand side must sum up to an integer as well and since \(b'\) is fractional by definition and \(a'_i x_i \geq 0\), for \(i = 1 \ldots n + m\), we must have that the right hand side is 0 or negative:

\[
b' - \sum_{i=1}^{n+m} a'_i x_i \leq 0.
\]

(2.8)

The equation (2.8) is the cut. By construction the values \(x_i = 0\) in our current continuous feasible solution (they are not basic variables) so the constraint makes our current continuous solution infeasible but all other feasible integer solutions remain feasible [5].

### 2.4 Implicit Enumeration

The Implicit Enumeration Algorithm is also known as Balas Additive Algorithm, and is only applicable in the case of a 0-1 IP since we will receive too many branches otherwise. The general idea is to utilize a process similar to Branch and Bound to fix variables at either zero or one by relaxing the constraints.

The following restrictions come with this method:

- The objective function \(z = c^T x\) have only nonnegative \(c_i\), \(i = 1 \ldots n\) for a minimization problem and only negative \(c_i\) for a maximization problem.
• The \( m \) constraints are all on the form \( \sum_{i=1}^{n} a_i x_i \geq b \) for a minimization problem and on the form \( \sum_{i=1}^{n} c_i x_i \leq b \) for a maximization problem.

• An ordering \( G \) on \( x \) is imposed such that \( x_i < x_j \iff c_i \leq c_j, \forall i, j, i \neq j \) for a minimization problem and in the case of a maximization problem; \( x_i < x_j \iff c_i \geq c_j, \forall i, j, i \neq j \).

This seems to reduce generality a lot but we can easily convert various problems to this form in a manner much like the standardization of linear programs.

**Theorem 2.2.** Given an optimal solution, \( S \), for a minimization 0-1 IP we can do a variable substitution on the form \( x'_i := 1 - x_i \iff x_i = 1 - x'_i \) and retain optimum without invalidating any constraints.

Informally, this means introducing a variable that represents the opposite of \( x_i \). Choosing \( x'_i = 1 \) corresponds to excluding \( x_i \) from the solution, and vice versa. Proof of the theorem is given below:

**Proof.** Consider the objective function \( z \) of \( S \). The substitution on the above mentioned form for a \( x_i \) with corresponding \( c_i \) will lead to \( z \) containing the term \( c_i x_i = c_i(1 - x'_i) = -c_i x'_i + c_i \) instead of the term \( c_i x_i \). Recall that \( c_i \) is negative, so \( x'_i \) being preceded by \( -c_i \) means that it has a positive coefficient in the objective function. Thus, choosing \( x'_i = 0 \) results in \( z \) attaining a smaller value, just like choosing \( x_i = 1 \) would. The magnitude of the change is the same, \( |c_i| \), since \( x_i \in \{0, 1\} \), same as \( x'_i \). Hence, the objective function is unchanged by this substitution, but we also need to verify that the constraints aren’t affected.

It is enough to consider a constraint with two terms since all terms are independent. So, consider a constraint on the form \( a_i x_i + a_j x_j \geq b \). Substituting \( x_i = 1 - x'_i \) gives us \( a_i - a_i x'_i + a_j x_j \geq b \iff -a_i x'_i + a_j x_j \geq b - a_i \). With \( x'_i = 0 \) we get \( a_j x_j \geq b - a_i \) and with \( x'_i = 1 \) we get \( a_i x_i \geq b \). In the constraint before the substitution we had that \( x_i = 0 \) gave us \( a_i x_i \geq b \) and \( x_i = 1 \) gave us \( a_j x_j \geq b - a_i \). Thus \( x'_i = 0 \) after substitution gives the same constraints as \( x_i = 1 \) before substitution and the reverse holds for \( x'_i = 1 \), so constraints are not invalidated by this substitution.

**Theorem 2.2** works for maximization problems as well and the proof is analogous. The following transformations are also available:

• If we have a constraint with strict inequalities and all constants integer we can easily convert them to \( \leq \) constraints.

• If we have reversed constraints (\( \leq \) constraints when \( \geq \) constraints are required) we can just multiply the entire constraint by \(-1\).

• Equality constraints can be split into two separate constraints, e.g. \( x_i = b_i \implies x_i \leq b_i \land x_i \geq b_i \). Then we can use the previous proposition to again have all \( \geq \) constraints.

• The ordering is trivial to change.

From the restrictions above we can note the following structure in our problem:
• For a minimization problem we have all nonnegative coefficients in the objective function, so setting all variables to 0 will give the optimal value of $z$.

• If setting all variables to 0 violates one or more constraints, then setting the variable with the smallest index to 1 minimizes $z$ since the variables are ordered according to $G$.

### 2.4.1 The algorithm

Each step of this algorithm consists of two parts: First, fix one or more variables to be either 0 or 1. Typically this is done one variable at a time and we denote this variable $x_N$ ("x now", not to be confused with $x_n$, $x$ with index $n$). The other part is connected to the calculation of an upper bound for the best possible solution in this part of the subtree, and two cases are possible:

1. If $x_N = 1$, then the solution might be feasible, so the node is bound by $z = \sum_{j=1}^{N} c_j x_j$.

2. If $x_N = 0$, then things are a bit different. Later we will see that we need to calculate a bounding function value only for nodes that are currently infeasible. That means that currently one of the $\geq$ constraints is not yet satisfied. But setting the current variable to 0, this constraint will not change. Therefore we must set at least another variable to 1 and the cheapest one is $X_{N+1}$, so the bounding function value is $\sum_{j=1}^{N} c_j x_j + c_{N+1}$. This setting might provide a feasible solution.

To check whether the solution proposed by the bounding function is feasible we simply set all variables past $x_N$ (if $x_N = 1$) or $x_{N+1}$ (if $x_N = 0$) to 0 and check all constraints. If all of the constraints are satisfied then this is the best solution in this subtree and the rest of its descendants will not require further evaluation. This is all due to the sign imposition on all $c_i x_i$ that asserts that setting more variables to 1 will give a worse value on $z$. Then the feasible solution is compared to the current best solution and the value is saved if our new solution is better.

### 2.4.2 Improvements

We can consider further pruning of nodes such as infeasibility pruning. That is we can check whether a node cannot develop into a feasible solution no matter how the rest of the variables are set. If we have a minimization problem, and therefore only $\geq$ constraints, we can simply set the rest of the variables in each constraint to give the maximum possible value (that is $\forall j, x_j = 0$ if $a_{ji} < 0$ and $x_j = 1$ if $a_{ji} > 0$). If the left hand side is maximized and still doesn’t fulfill the constraint then it can not fulfill the constraint in any subtree of the current solution and there is no need to evaluate any descendants of this node further. Note that this does not imply anything other than the possible presence of feasible solutions since it is possible to consider both $x_i = 0$ and $x_i = 1$ for different constraints which is not allowed in a candidate solution.
The current solution is found through a depth-first search and a possible improvement to this is to implement a more suitable search heuristic. One candidate is best-first search, that is, evaluating the subtree of the node with the current best bounding value first, thus making it a greedy algorithm. Greedy algorithms is a class of algorithms based on making the best choice locally every time, see e.g. [6, ch.4] for a more thorough explanation. By doing this, we can terminate as soon as we find a feasible solution, again due to the imposition on the signs of $c_i x_i$. If we do this, however, the algorithm can no longer be called Balas Additive Algorithm[7, ch. 13].

2.4.3 Alternative search heuristic and economic variable representation

Best first search seems like a good option over depth first search since it is guaranteed to visit fewer nodes although there might be some other node visitation order that could be better or worse depending on the problem structure. We would like to present one such alternative.

As with the best first search we have a method that will (implicitly) enumerate all $2^n$ solution candidates and thus guarantee completion. Most of the time, though, we will only have to generate a small subset of all $2^n$ possibilities. The article from which this method is taken is written by Arthur M. Geoffrion for the United States Air Force[8]. Refer to the article by Geoffrion for details while this report will present only the most important parts.

2.4.4 Representation

Given a problem with $n$ variables we say that a partial solution $S$ is defined as an assignment of binary values to a subset of the $n$ variables. Any variable not assigned in a partial solution is called free. We denote $x_j = 1$ by $j$ and $x_j = 0$ by $-j$. Thus if $n = 5$ and $S = \{3, 5, -2\}$, then $x_3 = 1, x_5 = 1, x_2 = 0$, and $x_1$ and $x_4$ are free. A completion of a partial solution $S$ is $S$ itself together with arbitrary assignment to the free binary variables. If we can find a completion of a partial solution that is feasible we may save it as the incumbent whenever the cost function is more optimal than the previous incumbent. Alternatively we might be able to determine that $S$ has no feasible completion better than the incumbent. In either case the partial solution $S$ is now fathomed (we will present exactly how to fathom partial solutions later). All completions of a fathomed solution is implicitly enumerated.

To avoid redundancy and be certain that we only visit each node once we need to keep track of which nodes we have visited. Consider that we can fathom $S^{k_i}$ at iteration $k_i$. Then we will avoid redundancy if we choose $S^{k_i+1}$ to be exactly $S^{k_i}$ with its last element multiplied by $-1$ and underlined which represents taking the other branch on the last variable while simultaneously marking the first branch as fathomed. If then $S^{k_i+1}$ is also fathomed this is equivalent to $S^{k_i}$ less its last element being fathomed and we can choose $S^{k_i+2}$ to be $S^{k_i}$ less its last element with its next to last element multiplied by $-1$ and underlined. To get a better view of this consider figure 2.1. If we choose $x_1 = 1$ and $x_2 = 0$ (figure 2.1(a)) and fathom the partial solution $S^{k_i} = \{1, -2\}$ (figure 2.1(b)) we can continue with the other branch of $x_2$ that is $x_2 = 1$ or $S^{k_i+1} = \{1, 2\}$. If this is also fathomed (figure 2.1(c)) we take $S^{k_i+2} = \{-1\}$, that is we choose $x_1 = 0$ since all options where $x_1 = 1$ have already
been implicitly enumerated. Ergo, if two children to a node are fathomed the parent is also fathomed (figure 2.1(d)).

Of course whenever we cannot fathom a partial solution we simply need to augment that solution with an assignment of one of the free variables and repeat this process until all nodes are implicitly enumerated.

2.4.5 Fathoming and node ordering

What remains to be determined is how we fathom partial solutions as well as the order in which we visit the nodes.

Geoffrion suggest just looking at the best (not necessarily unique or feasible) completion $x^\ell$ of $S$ which can trivially be created by assuming $x^\ell_j = 0$ for each free variable when minimizing objective function with all $c_j \geq 0$. If this is not feasible we do nothing further to find the best feasible completion but instead attempt to determine that no feasible completion of $S$ is better than the incumbent. If this is the case, then it is impossible to complete $S$ to make it both feasible and better than the incumbent.

Furthermore Geoffrion presents a method for how to choose the next variable to augment to a partial solution that has not yet been fathomed, but any ordering is sufficient for a complete algorithm. For further details on how this is all done we refer to Geoffrion’s article [8, p. 13].
Chapter 3
Related work

Before moving on with the thesis we have to note that the subject of optimization and specifically Integer Programming is famous and very vast. There are books and entire courses dedicated to integer programming and due to this we simply cannot cover everything. Therefore we refer to the following further readings.

3.1 Basics of Integer Programming

If further interested in integer programming, other applications not presented in this thesis and different solve methods, there are various books and most of them would be sufficient for reading more for this thesis. For example Integer Programming: Theory and Practice by Karlof [9], Integer Programming by Wolsey [10] or Combinatorial Optimization by Papadimitriou [11].

3.2 The Complexity of Integer Programming

In computational complexity theory we divide problems into classes depending on their difficulty. The most relevant for us are:

- NP - Class of computational problems for which a given solution can be verified as a solution in polynomial time.
- NP-hard - Class of problems which are at least as hard as the hardest problems in NP.
- NP-complete - Class of problems which contains the hardest problems in NP.

Integer programming is NP-hard while 0-1 integer programming is NP-complete and is a part of Karp’s 21 NP-complete problems [1]. The obvious conclusion can be drawn: if
we have a solver for integer programming we can solve a 0-1 integer programming problem as well. We can reduce 0-1 integer programming to integer programming.

That being said, we can take alternative approaches to solving our 0-1 integer programs by for example reducing it to another NP-complete problem, the most famous one being boolean satisfiability (SAT) which for example could be solved using constraint programming. Further reading about constraint programming can be found in Constraint Programming by Mayoh [12] among other. More specific implementations that we have considered are presented later in this chapter.

Some of the books in the previous section also contain some information regarding the complexity of integer programming. Although, we prefer Algorithm Design by Kleinberg and Tardos [6]. More information about computational complexity can also be found in Computational Complexity: A Modern Approach by Arora [13] or Computational Complexity by Papadimitriou [14].

3.3 Improvements

The studies presented in this section are related to improving the efficiency when solving 0-1 ILPs.

3.3.1 Evaluating the Impact of AND/OR Search on 0-1 Integer Linear Programming

This very interesting article by Radu Marinescu and Rina Dechter suggests that the binary branch and bound algorithm can be improved by merging unexplored branches together whenever the variables branched on would generate two identical subtrees [15]. This is done, as explained in more detail in section 7.3, by defining a constraint graph in which all variables related via constraints are connected and from it and from this graph determining the “Context” of each node in the search tree. Whenever the contexts are the same the subtrees are identical. The idea to test a variant of the best-first search method was partly inspired by this article as some of their best results came from a variant of it.

This is probably a very good optimization for our implementation and especially for the problems on which our method performs the worst on (difficult ones). In these problems, specifically when we have a sparse $A$ matrix, we get contexts which do not consist of all variables which then would lead to possibilities of merging subtrees. The opposite is when the $A$ matrix is full, i.e. all variables are present in all constraints, in which case this optimization would not give any improvement at all since contexts would always differ on different nodes.

The thesis doesn’t do any tests on implementations of the AND/OR search tree but it is still a very relevant optimization especially for future work.

3.3.2 Infeasibility Pruning

Dr. John Chinneck at Carleton University has written course material about, among other, binary integer programming [7]. We have used this article as inspiration in our thesis al-
3.4 Other takes on solving 0-1 ILPs

Naturally the binary branch and bound algorithm is not the only way to solve a 0-1 ILP as hinted previously in this chapter. There are several other methods which we didn’t consider in this thesis due to the limited time frame and we will present some of them in this section. The following three papers have also been considered.

3.4.1 The Reduce and Optimize Approach

Gerardo Trevino Garza published this approach to solving binary integer programs (BIPs) at Arizona State University in 2009\[16\]. We did give this article a shot but since it requires a reduction of the problem (LP-relaxation is suggested) we decided that it was not what we were looking for.

The main idea is to find a set $S$ of indices of variables to be set to one that contains the entire set of indices of variables that are equal to one at the optimal solution. In addition to this, the method requires $S$ to be small for the algorithm to terminate within reasonable time. The main problem is that finding a set $S$ that is small enough is as hard as the original problem, and so the reduction is used as a heuristic for finding this set $S$.

3.4.2 An algorithm for large scale 0-1 Integer Programming with application to airline crew scheduling

The ideas proposed in this article by Dag Wedelin\[17\] seemed relevant since the problem studied was very similar to ours. However, since this algorithm used approximation schemes, we chose to discard it as a possibility of inspiration since we decided fairly early on to attempt to solve the problem exactly. Additionally, this scheme required the choosing of method parameters which ought to be problem dependent. As was utilised in this article, the notion of solving Lagrangian relaxations were considered prior to focusing on enumeration methods.

3.4.3 Pivot-and-Fix; A Mixed Integer Programming Primal Heuristic

Another interesting heuristic approach was suggested in this paper by Mahdi Namazifar, Robin Lougee-Heimer and John Forrest\[18\]. It is designed to find feasible solutions to MIPs to help solvers run faster and is based on solving an LP relaxation prior to performing a tree search in the vicinity of this approximate solution to fix variables not yet satisfying integrality constraints. Suggesting that this heuristic improves performance of MIP solvers...
in general, this might have been an applicable improvement to look into had we decided on algorithms based on solving LP relaxations.

3.5 Linear Programming in Database

This article concerns a different course of action taken to solve a linear program in a database\(^{[19]}\). Akira Kawaguchi and Andrew Nagel used stored procedures to access data within a relational database to then perform the simplex method on LPs defined within that database. The perceived lack of tools for solving these types of problems was also reiterated which further motivated our own project. It seemed to have the same limitations as in our case regarding the lack of portability, but not due to working inside a computation engine - instead due to different implementations and limitations of relational databases, such as limited column sizes and truncation errors. Still, the test results were of some interest since the project was in many ways similar to our own and it made apparent the ineptitude of relational databases to represent LPs in a straightforward and efficient way.

3.6 Other applications of ILPs

There are many applications for ILP and no less for binary ILPs. It is quite hard to find articles with the same application of binary ILPs as this thesis presents, though. This is most likely due to Qliks unique engine which masks the visualized data in binaries. We will not cite any specific articles here but instead mention scheduling, packing, set cover and set partitioning which are all famous problems that can be solved via binary ILPs. Further concretizations of this is for example the travelling salesman problem which is a scheduling problem or the knapsack problem which is a packing problem. The main concept remains similar to the application in this thesis though, and that is to decide whether the decision variable \(x_i\) should be a part of the solution \((x_i = 1)\) or not \((x_i = 0)\).
Chapter 4
Introducing the basics of Qlik’s internal structure

As noted in the introduction the goal of this thesis is to analyze implementations of integer programming algorithms in Qlik’s products. To understand how this is possible we introduce some basic knowledge regarding Qlik’s internal structures.

4.1 Loading data from a database source in Qlik

To load data into Qlik one uses Qlik’s script engine. The script shares similar syntax to SQL with commands like "select", "from", "where", etc. Loading data via database connections such as "OLE-DB"[20] and "ODBC"[21] is also supported along with regular load from .txt and .xls files. The script engine loads and transforms the data to fit the internal database that it is to be stored in.

4.2 Qlik Internal Database

Qlik Internal Database (QIDB) is where the entire database is stored. The QIDB is stored in RAM at runtime to have faster access when doing selections in the application. Thus having a very large database will cause memory usage of your computer to spike.

The QIDB stores data in quite a special and compact way. For example, two tables consisting of a total of four fields - "Author" consisting of names of authors, "Title" of book titles, "Genre" of book genres and "Score" of double values between 0 and 5, we could have initial data such as in table 4.1. Later in this example, we will use the authors’ last names along with abbreviations of the titles. If we load the fields in order of appearance,
the data would be represented as a table "fields" seen in table 4.2 and as a table "tables" seen in table 4.3.

<table>
<thead>
<tr>
<th>T1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td>Title</td>
<td>Genre</td>
</tr>
<tr>
<td>Norah James</td>
<td>Sleeveless Errand</td>
<td>Thriller</td>
</tr>
<tr>
<td>Douglas Adams</td>
<td>The Hitchhiker’s Guide to the Galaxy</td>
<td>Sci-Fi</td>
</tr>
<tr>
<td>George R.R. Martin</td>
<td>Dying of the Light</td>
<td>Sci-Fi</td>
</tr>
<tr>
<td>George R.R. Martin</td>
<td>A Game of Thrones</td>
<td>Fantasy</td>
</tr>
<tr>
<td>C.S. Lewis</td>
<td>The Lion, the Witch and the Wardrobe</td>
<td>Fantasy</td>
</tr>
<tr>
<td>Stephen King</td>
<td>The Shining</td>
<td>Thriller</td>
</tr>
<tr>
<td>Stephen King</td>
<td>The Mist</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Score</td>
</tr>
<tr>
<td>Sleeveless Errand</td>
<td>3.5</td>
</tr>
<tr>
<td>The Hitchhiker’s Guide to the Galaxy</td>
<td>4.0</td>
</tr>
<tr>
<td>Dying of the Light</td>
<td>3.5</td>
</tr>
<tr>
<td>A Game of Thrones</td>
<td>4.5</td>
</tr>
<tr>
<td>The Lion, the Witch and the Wardrobe</td>
<td>4.0</td>
</tr>
<tr>
<td>The Shining</td>
<td>4.0</td>
</tr>
<tr>
<td>The Mist</td>
<td>4.0</td>
</tr>
</tbody>
</table>

Table 4.1: Example tables to be loaded into QIDB.

<table>
<thead>
<tr>
<th>$F_x$</th>
<th>$V_x$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Author)</td>
<td>James</td>
<td>Adams</td>
<td>Martin</td>
<td>Lewis</td>
<td>King</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(Title)</td>
<td>SE</td>
<td>tHGttG</td>
<td>DotL</td>
<td>AGoT</td>
<td>LtWatW</td>
<td>Shining</td>
<td>Mist</td>
</tr>
<tr>
<td>2</td>
<td>(Genre)</td>
<td>Thriller</td>
<td>Sci-Fi</td>
<td>Fantasy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(Score)</td>
<td>3.5</td>
<td>4.0</td>
<td>4.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Field representation of initial data.

As you can see, duplicate data from the same field is not stored multiple times, but only the first time it occurs. We note that all indices (subscripted with "x") contain integers only and that the extra explanation in the parentheses are just for increased readability.

The creation of table "fields" is straightforward. As a rule of thumb; if the entry does not already exist, append it. If it already exists we can use the previous entered value and thus exclude all duplicates. So we add the values to "fields" in the order: Norah James, Sleevelless Errand, Thriller, Douglas Adams, The Hitchhiker’s guide to the Galaxy, Sci-Fi, George R.R. Martin, etc. for T1 according to Table 4.1. Note that the duplicates of authors and genres are not added to "fields".
We must also keep track of which entry belonged to which table, thus creating the table "tables". After every entry in "fields" we create a mapping in "tables" which makes it possible to reconstruct the actual tables. Each table gets an index $T_x$ and for each field in that table we store the field index $F_x$ and the index $R_x$ representing where in "fields" we can find the value of the field. Thus for table $T_1$, $T_0$, we have three fields $F_1$, $F_0 = F_0$, $F_2$, $F_x = F_1$, and $F_3$, $F_0 = F_2$. For example we can find the first input of $T_1$, Norah James, at index 0 of the Author field and the last input of $T_1$, Thriller, at index 0 of the Genre field as seen in table 4.3.

### 4.3 Inference Machine

Qlik’s softwares have a unique way of presenting the data as well as making it interactive and easier to digest. Users may select values from the data set and the inference machine in Qlik’s engine calculates a new state and which data that is related or unrelated to the current selection, coloring them green or gray accordingly. The inference machine has two primary tasks:

1. Create a doc state (see 4.4) that represents which state each data is in.
2. Create the Hypercube (see 4.5) which holds all relevant data to be displayed.

The inference machine runs every time the user clicks or otherwise modifies the state of the document to generate fresh doc states and Hypercubes in real time.

### 4.4 From QIDB to doc state

When talking about the doc state in this section, we refer to table 4.4 as "doc state fields" and table 4.5 as "doc state tables". This is in contrast to the tables from section 4.2 which we now refer to as "QIDB fields" and "QIDB tables".

When a user selects a variable to focus on, a doc state is created which is a \{0, 1\} bit-mask stating which nodes in the field/table-representation are active, i.e. which values may be related to the selected data. In the section 4.2 example, if a user were to click on Thriller, the doc states created would be as in tables 4.4 and 4.5.

The generation of the doc state is done with help from the inference machine and we can follow the logic of it by hand. In the above example when the user clicks on Thriller
4. Introducing the basics of Qlik’s internal structure

Table 4.4: Doc state-representation of fields.

<table>
<thead>
<tr>
<th>( F_x \cup I_x )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Author)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 (Title)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 (Genre)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 (Score)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Doc state-representation of tables.

| T1  | 1 0 0 0 0 1 1 |
| T2  | 1 0 0 0 0 1 1 |

Table 4.6: Doc state-representation of fields step 1.

the entire row of genres in "doc state fields" becomes inactive (value 0) except Thriller itself which is set to active (value 1), which can be seen in table 4.6.

Table 4.6: Doc state-representation of fields step 1.

<table>
<thead>
<tr>
<th>( F_x \cup I_x )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Author)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 (Title)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 (Genre)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 (Score)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7: Doc state-representation of tables step 1.

Having one row completed we can fill out the rest of the tables by looking in "QIDB tables". To begin with, since \( I_0 = 1 \) in "doc state fields" and \( I_1 = 0 \) and \( I_2 = 0 \) we know that all zeros in the genre row in "QIDB tables" are going to become active in "doc state tables". As seen in "QIDB tables", table 4.3 \( I_0 = 0, I_5 = 0 \) and \( I_6 = 0 \) in the genre row. Thus we can fill out "doc state tables" with ones on indices 0, 5, and 6 and zeros on the rest as seen in table 4.7. This of course means that inputs number 0, 5 and 6 are all related to Thriller, and by looking in table 4.1 we can confirm that fact.

Table 4.7: Doc state-representation of tables step 1.

| T1  | 1 0 0 0 0 1 1 |
| T2  |               |

Now that we completely filled out the T1 row we know that the indices 0, 5 and 6 are all active in all fields in T1, that is Author, Title and Genre (though, we already handled Genre). Thus we go back and look in "QIDB tables" and see that for the Author row we have \( R_0 = 0, R_5 = 4 \) and \( R_6 = 4 \), which then tells us that our "doc state fields" should be
active, have ones, at indices 0, 4 and 4. Doing this for both Author and Title rows gives us table 4.8.

<table>
<thead>
<tr>
<th>$F_i \setminus V_i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (Author)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 (Title)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 (Genre)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 (Score)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.8: Doc state-representation of fields step 2.

Since Title is a field in both T1 and T2 we can now infer activeness from the Title row to the Score row in "doc state fields". Since $I_0$, $I_5$ and $I_6$ are active we can look in "QIDB tables" and see that the active scores are in fact the values at these indices, i.e. 0, 1 and 1. Thus we can fill out the Score row of our "doc state fields" to receive our resulting table 4.4.

Lastly we look at the field that was shared among the tables, here Title, and let that infer which rows of the table are to be active in the T2 row in "doc state tables". As before we see indices 0, 5 and 6 as active and set all rows with values 0, 5 and 6 in "QIDB tables" as active (remember that index 0, 5 and 6 coincides with value 0, 5 and 6 by a coincidence in this example only) giving us the resulting "doc state tables" in table 4.5.

### 4.5 The Hypercube

The hypercube is where all the relevant data tuples are stored. By help of the inference machine and doc state the hypercube gathers all the active posts (represented by ones in the doc state) and fetches the corresponding values from QIDB. The tuples are stored in a big table as dimensions and expressions (similar to the concept of keys and values). We can see the hypercube as an object that collects all relevant data that are spread out in the data model due to the associative data model (see section 4.6). The hypercube is stored in RAM for quick access with the downside of taking up a lot of memory when handling large data. It is constructed after the inference machine has traversed the data model to create dependencies and after the calculation of expressions (computing e.g. the sum of an inferred subfield).

### 4.6 The associative model in Qlik

As per the example in section 4.2 the QIDB is an associative database as opposed to a relational database (see [22] for a deeper explanation of relational and associative models). In short a relational database is record based and works with entities and attributes. Each entity (row) will have a number of attributes (columns) which are stored in a relation (table). See table 4.9 for an example of a relational model. The associative database on the other hand stores all data discretely and independently with an unique identifier and
Introducing the basics of Qlik’s internal structure

<table>
<thead>
<tr>
<th>Author</th>
<th>Title</th>
<th>Genre</th>
<th>Year</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Norah James</td>
<td>Sleeveless Errand</td>
<td>Thriller</td>
<td>1930</td>
<td>3.5</td>
</tr>
<tr>
<td>Douglas Adams</td>
<td>The Hitchhiker’s Guide to the Galaxy</td>
<td>Sci-Fi</td>
<td>1979</td>
<td>4.0</td>
</tr>
<tr>
<td>George R.R. Martin</td>
<td>Dying of the Light</td>
<td>Sci-Fi</td>
<td>1977</td>
<td>3.5</td>
</tr>
<tr>
<td>George R.R. Martin</td>
<td>A Game of Thrones</td>
<td>Fantasy</td>
<td>1996</td>
<td>4.5</td>
</tr>
<tr>
<td>C.S. Lewis</td>
<td>The Lion, the Witch and the Wardrobe</td>
<td>Fantasy</td>
<td>1950</td>
<td>4.0</td>
</tr>
<tr>
<td>Stephen King</td>
<td>The Shining</td>
<td>Thriller</td>
<td>1977</td>
<td>4.0</td>
</tr>
<tr>
<td>Stephen King</td>
<td>The Mist</td>
<td>Thriller</td>
<td>1984</td>
<td>4.0</td>
</tr>
</tbody>
</table>

**Table 4.9:** Relational model.

then stores the relationships as associations also with unique identifiers. The associations then connect the identifiers with each other just like the "tables" table in QIDB. See image 4.1 for an example of an associative model.

![Associative model](image)

**Figure 4.1:** Associative model

The associative model in Qlik will hopefully grant us some edge when designing our optimization algorithm.
In this chapter, we will test two commercial optimizers and measure their efficiency and capability to handle large inputs for different inputs of the warehouse problem defined in section 1.3. Then we will provide Qlik-related tests with our own implementations. Each method will be presented one by one and afterwards they will be compared.

5.1 Problem structure

After solving some smaller examples by hand and running some tests on computer we were convinced that the problem difficulty depended heavily on the data, so for each problem size we had to consider different models to see on what type of problems each tool shines. This is interesting because if we manage to find a group of problems where the associative model puts us leaps and bounds in front of other methods, we can constrain our solver to specialize in these problems and potentially arrive at something both very useful or even ground-breaking. The different problems we considered were all 8 possible combinations of the following:

- Demands on 25% of the products (Low demands) versus demands on every product (High demands).
- Every warehouse contains every product (Low sparsity) versus each warehouse contains 50% of the products (High sparsity).
- Each product present in a warehouse has a minimum quantity of 20% of the maximum possible demand (High products) versus each product may exist in any quantity (Low products).

The first item means that we either have relatively few or many constraints. The second item corresponds to having either a full or a sparse $A$-matrix. The third item means that $a_{ji}$ has minimum value of 20% of max demand or 0 (see section 1.2 on page 14 for notations).
The following 16 problem sizes were considered:

- 10 warehouses, 10 products.
- 50 warehouses, 50 products.
- 100 warehouses, 100 products.
- 200 warehouses, 200 products.
- 500 warehouses, 500 products.
- 1000 warehouses, 1000 products.
- 100 warehouses, \{10, 20, 30, 40, 50\} products.
- \{10, 20, 30, 40, 50\} warehouses, 100 products.

The two final entries in the problem sizes are mainly there to inspect which dimension changes provide the most difference in computational difficulty. To run tests on these problems, we created a data generator. Apart from being able to set the number of warehouses \(n\) and the number of products \(\geq m\), the following properties are important to note:

- The maximum time of a warehouse is constant 20.
- The maximum demand for a product is constant 100.
- Quantities of products depend on the maximum demand.
- Feasibility is guaranteed by inserting the maximum demand for each product in a random warehouse.

Consider a brute force approach consisting of systematically iterating over all possible solutions. For decision problems we might not have to generate all possible since any solution is satisfactory. In our case, on the other hand, we are looking for a best solution and must therefore compare all possible solutions. This is in fact the worst case scenario of the implicit enumeration method from section [2.4] - if the only solution to the problem happens to be the most expensive one. We have therefore not considered running any tests on any brute force method since we already know of a method which is at least as good.

## 5.2 Microsoft Solver Foundation

Our first step in benchmarking was using the tool Microsoft Solver Foundation [23]. We created a solver for the warehouse problem in C++/Microsoft Visual Studio.
5.2.1 Setting up the model

Microsoft Solver Foundation provides a straightforward interface for the user to create a model - one simply creates a model and adds:

- As many decisions as needed, and define their domains. In the warehouse problem case we are working with zero-one integer programming, thus the domains are boolean.
- As many constraints as needed. This is easily done through the overloaded operators which allows decisions to, for example, be multiplied with double values as well as added together. The resulting term of decisions is then compared with our right hand side to completely define our constraints.
- A goal which is our objective function along with a selection of whether we want to maximize or minimize the solution.
- Various options - specifically the user selects which type of algorithm to be used (Simplex- or Constraint Programming-directive). We will mainly be using the Simplex Directive for benchmarking since it seemed to have better performance on larger test cases.

5.2.2 Results

All raw data can be seen in [24]. We will present and highlight the most interesting points. Note that the execution time is in fact only the time between calling solve and when we receive our answer. Since Qlik’s domain is not made out of pre-built matrices on the correct form we must transform the data, giving us a lot of overhead.

A simple view of averages of runtime statistics over test cases of different sizes can be viewed in table 5.1.

While averages over a problem size doesn’t tell us too much directly (since the difficulty of our problems differ greatly within a specific problem size) we can use these values to compare the method’s efficiency with other methods’.

We can see the obvious trend of the number of branches increasing exponentially with increasing number of warehouses and number of products in figure 5.1. We chose to exclude the cases of 200 warehouses and 200 products or larger since these instance had way more branches than the rest and make the graph unreadable.

5.3 Gurobi

One of the cutting edge solvers for optimization is Gurobi[25]. Along with CPLEX[26] these are the optimizers that one would like to compare to in this area of work. Unlike Microsoft Solver Foundation this software is standalone and is optimized to solve optimization problems solely, whilst the Solver Foundation is more of a plugin for Visual Studio. That being said, it would be "unfair" to run Gurobi with all of it’s optimizations against our own implementation and even the Solver Foundation, since there might be possibilities to implement many of the optimizations from Gurobi in our own implementation
5. Method

<table>
<thead>
<tr>
<th>Size (warehouses × products)</th>
<th>Average number of branches</th>
<th>Average execution time (ms)</th>
<th>Unfinished tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 × 10</td>
<td>27.75</td>
<td>24.25</td>
<td>0</td>
</tr>
<tr>
<td>10 × 100</td>
<td>38.125</td>
<td>33.25</td>
<td>0</td>
</tr>
<tr>
<td>20 × 100</td>
<td>259.125</td>
<td>103.125</td>
<td>0</td>
</tr>
<tr>
<td>30 × 100</td>
<td>1060.75</td>
<td>476.875</td>
<td>0</td>
</tr>
<tr>
<td>40 × 100</td>
<td>1851.38</td>
<td>856.25</td>
<td>0</td>
</tr>
<tr>
<td>50 × 50</td>
<td>2109.25</td>
<td>647.75</td>
<td>0</td>
</tr>
<tr>
<td>50 × 100</td>
<td>3568</td>
<td>2018.88</td>
<td>0</td>
</tr>
<tr>
<td>100 × 10</td>
<td>1402.5</td>
<td>204.375</td>
<td>0</td>
</tr>
<tr>
<td>100 × 20</td>
<td>5146.88</td>
<td>1029</td>
<td>0</td>
</tr>
<tr>
<td>100 × 30</td>
<td>4963.38</td>
<td>1350</td>
<td>0</td>
</tr>
<tr>
<td>100 × 40</td>
<td>7292.38</td>
<td>2206.63</td>
<td>0</td>
</tr>
<tr>
<td>100 × 50</td>
<td>8772.88</td>
<td>3418.75</td>
<td>0</td>
</tr>
<tr>
<td>100 × 100</td>
<td>36216.1</td>
<td>32198.5</td>
<td>0</td>
</tr>
<tr>
<td>200 × 200</td>
<td>1483130</td>
<td>4302890</td>
<td>0</td>
</tr>
<tr>
<td>500 × 500</td>
<td>4188139</td>
<td>23632356</td>
<td>5</td>
</tr>
<tr>
<td>1000 × 1000</td>
<td>*</td>
<td>*</td>
<td>8</td>
</tr>
</tbody>
</table>

**Table 5.1:** Averages over different sizes of the warehouse problem solved by Microsoft Solver Foundation.

**Figure 5.1:** Line chart over number of branches per problem size in Microsoft Solver Foundation.

given enough time. Therefore we have turned off some optimizations that Gurobi does before running our tests.
5.3.1 Setting up the model

After installing Gurobi to a computer one simply needs to import the correct header files to use the solver package. Then, one can create a model with constraints and objective function in the same way as with Microsoft Solver Foundation but with different syntax (see section 5.2.1). We also turned off the following optimization options:

- Presolving - Gurobi reduces the the initial matrix by identifying overlapping constraints among other things.

- Cuts - Gurobi uses Branch and Cut to solve IPs and generates extra constraints at every branch to further reduce the search space (see section 2.3.3).

- Heuristics - Gurobi spends a part of the execution time on trying to find a heuristic solution to determine the best path to take while branching. The implicit enumeration algorithm from section 2.4 uses the heuristic "best branch first", but we can motivate turning this option off since solving the linear program as a part of the bounding procedure could also be seen as a heuristic. The heuristics that we turn off are external (Gurobi uses 14 different heuristics[25]).

5.3.2 Results

Despite making Gurobi run suboptimally it beats the Solver Foundation both in execution time and in number of branches. The raw data can be found at [24] and we will only present averages in this section. Again we note that the execution time is time spent solving the system and does not include time spent building the model.

The averages of runtime statistics over test cases of different sizes can be seen in table 5.2. One branch in this table means one full solution of a linear system and each full simplex consists of a number of simplex iterations which is the statistic we can see under the column "Simplex iterations".

The line chart in figure 5.2 tells us that the difficulty indeed increases exponentially with problem size, even for state of the art solvers like Gurobi.

For future reference we also present the setup time for the model in Gurobi in table 5.3. The setup times are only taken as random samples and not as an average over all setup times per problem size due to not saving this measurement from the beginning.

5.4 A first attempt at Implicit Enumeration

The Implicit Enumeration method introduced in section 2.4 seems to resonate very well with Qlik’s underlying data model since the product can provide all the necessary calculations given that we make the right selections. Thus, we can select a candidate solution and then check for feasibility. We decided to implement an Implicit Enumeration scheme coupled with the best-first search heuristic also mentioned in the same section to match against the results we obtained through Gurobi and Microsoft Solver Foundation. To be able to focus on the theory and getting indications on whether our methods seem promising, we
5. Method

<table>
<thead>
<tr>
<th>Size (warehouses × products)</th>
<th>Nodes explored</th>
<th>Simplex iterations</th>
<th>Average execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 × 10</td>
<td>9.125</td>
<td>13</td>
<td>0.00875</td>
</tr>
<tr>
<td>10 × 100</td>
<td>15.5</td>
<td>36.375</td>
<td>0.01</td>
</tr>
<tr>
<td>20 × 100</td>
<td>35</td>
<td>88.25</td>
<td>0.0125</td>
</tr>
<tr>
<td>30 × 100</td>
<td>52.75</td>
<td>136.875</td>
<td>0.0125</td>
</tr>
<tr>
<td>40 × 100</td>
<td>129.5</td>
<td>287.375</td>
<td>0.025</td>
</tr>
<tr>
<td>50 × 50</td>
<td>107.125</td>
<td>235</td>
<td>0.0175</td>
</tr>
<tr>
<td>50 × 100</td>
<td>94.875</td>
<td>262.5</td>
<td>0.025</td>
</tr>
<tr>
<td>100 × 10</td>
<td>79.375</td>
<td>109.375</td>
<td>0.015</td>
</tr>
<tr>
<td>100 × 20</td>
<td>92.25</td>
<td>120.625</td>
<td>0.02</td>
</tr>
<tr>
<td>100 × 30</td>
<td>81.375</td>
<td>128.875</td>
<td>0.01625</td>
</tr>
<tr>
<td>100 × 40</td>
<td>124.625</td>
<td>237.25</td>
<td>0.02375</td>
</tr>
<tr>
<td>100 × 50</td>
<td>125.125</td>
<td>175</td>
<td>0.02625</td>
</tr>
<tr>
<td>100 × 100</td>
<td>234.375</td>
<td>611.125</td>
<td>0.0575</td>
</tr>
<tr>
<td>200 × 200</td>
<td>870.25</td>
<td>3566.5</td>
<td>0.33125</td>
</tr>
<tr>
<td>500 × 500</td>
<td>4232.25</td>
<td>29894.9</td>
<td>5.99875</td>
</tr>
<tr>
<td>1000 × 1000</td>
<td>611176</td>
<td>6850130.625</td>
<td>2004.65</td>
</tr>
</tbody>
</table>

Table 5.2: Averages over different sizes of the warehouse problem solved by Gurobi.

![Figure 5.2: Line chart over number of branches per problem size in Gurobi.](image)

created this implementation outside of QIX engine in a mashup using the software development kit Qlik Sense Analytics for Visual Studio[27]. This plugin gave us the possibility to communicate with the QIX engine and use it as a computational platform.
5.4 A first attempt at Implicit Enumeration

<table>
<thead>
<tr>
<th>Size</th>
<th>Setup time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 × 10</td>
<td>0</td>
</tr>
<tr>
<td>10 × 100</td>
<td>0</td>
</tr>
<tr>
<td>20 × 100</td>
<td>0</td>
</tr>
<tr>
<td>30 × 100</td>
<td>0</td>
</tr>
<tr>
<td>40 × 100</td>
<td>0</td>
</tr>
<tr>
<td>50 × 50</td>
<td>0</td>
</tr>
<tr>
<td>50 × 100</td>
<td>1</td>
</tr>
<tr>
<td>100 × 10</td>
<td>0</td>
</tr>
<tr>
<td>100 × 20</td>
<td>0</td>
</tr>
<tr>
<td>100 × 30</td>
<td>0</td>
</tr>
<tr>
<td>100 × 40</td>
<td>0</td>
</tr>
<tr>
<td>100 × 50</td>
<td>0</td>
</tr>
<tr>
<td>100 × 100</td>
<td>2</td>
</tr>
<tr>
<td>200 × 200</td>
<td>6</td>
</tr>
<tr>
<td>500 × 500</td>
<td>35</td>
</tr>
<tr>
<td>1000 × 1000</td>
<td>168</td>
</tr>
</tbody>
</table>

Table 5.3: Setup times for different sizes of the warehouse problem solved by Gurobi.

5.4.1 Best-first search

We decided to implement best-first search in the following way: assume that the variables are sorted in ascending order after their coefficients in the objective function $z = c^T x$. That is, respect the ordering mentioned in section 2.4. Maintaining a priority queue containing candidate solutions - lists of variable indices denoting which variables would be set to 1 - sorted by the cost of a solution, cheapest first, most expensive last, do the following:

1. Initiate the queue to contain only $x_1$.

2. If the queue is empty, no feasible solution exists. Otherwise, extract the first element from the queue and check for feasibility. If this candidate solution represents a feasible solution, return it and terminate the algorithm.

3. Identify the cheapest hitherto unincluded variable - $x_{k+1}$ given that $x_k$ is the last element in the current candidate solution. If one exists, insert the following two candidates in the queue: one list containing the same variables as the current node but with the addition of $x_{k+1}$, and one containing $x_{k+1}$ instead of $x_k$.

4. Return to step 2

If the problem constraints are known to have only nonnegative coefficients coupled with $\geq$ and $>$ constraints, an infeasibility check could be performed before running the algorithm and the first part of step 2 could be omitted. An example of traversal order for a problem with three variables $x_1, x_2, x_3$ and objective function $z = x_1 + 2x_2 + 3x_3$ is given

43
in table 5.4 where the values in parentheses denote the cost associated with the candidate solution.

<table>
<thead>
<tr>
<th>Step</th>
<th>Current solution candidate</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{x_1}(1)</td>
<td>{x_2}(2), {x_1, x_2}(3)</td>
</tr>
<tr>
<td>2</td>
<td>{x_2}(2)</td>
<td>{{x_1, x_2}(3), {x_3}(3), {x_2, x_3}(5)}</td>
</tr>
<tr>
<td>3</td>
<td>{x_1, x_2}(3)</td>
<td>{{x_3}(3), {x_1, x_3}(4), {x_2, x_3}(5), {x_1, x_2, x_3}(6)}</td>
</tr>
<tr>
<td>4</td>
<td>{x_3}(3)</td>
<td>{{x_1, x_3}(4), {x_2, x_3}(5), {x_1, x_2, x_3}(6)}</td>
</tr>
<tr>
<td>5</td>
<td>{x_1, x_3}(4)</td>
<td>{{x_2, x_3}(5), {x_1, x_2, x_3}(6)}</td>
</tr>
<tr>
<td>6</td>
<td>{x_2, x_3}(5)</td>
<td>{{x_1, x_2, x_3}(6)}</td>
</tr>
<tr>
<td>7</td>
<td>{x_1, x_2, x_3}(6)</td>
<td>{}</td>
</tr>
</tbody>
</table>

Table 5.4: Step-by-step best-first search on a problem with objective function \(z = x_1 + 2x_2 + 3x_3\).

5.4.2 Setting up the model

Instead of creating a model as with Solver Foundation we need to input the data in Qlik Sense in such a way that the associations are created correctly. Thus we load a table with columns "product", "warehouse" and "quantity", one with columns "warehouse" and "time" and a third with columns "product" and "demand". This will result in a data model looking something like figure 5.3. However one loads their data, Qlik makes sure that the appropriate fields are created and separated, as opposed to a relational data model. This means that we can look at a field e.g. "warehouse" individually, and select different warehouses and see what subset of the model this specific warehouse is related to (specifically how much quantity of each product it contains). Along with the field of demands we can via some set analysis see whether our demands are met with the current selections or not (read more about set analysis in Qlik in [28]).

5.4.3 Results

All raw data can be seen in [24]. Once again we will present the relevant parts before comparing. It is important to note that execution time is not included here since the tool used to communicate with the engine is very slow and we only use it to motivate further implementations.

A table of averages of runtime statistics over the different test sizes can be viewed in table 5.5.

The line chart for number of branches per problem size can be seen in figure 5.4. Note the "spike" at 50 warehouses and 100 products requiring a greater amount of branches than 100x50. This indicates that some problem types are harder for this method than for e.g. Microsoft Solver Foundation whilst some are easier. Even so, we see that the difficulty increases with strictly larger problem sizes as expected.
5.5 A first analysis

In this section we intend to compare the results of the different methods presented and motivate our future course of action. To visualize the runtime statistics we generated a
5. Method

Recall that we mentioned that the difficulty for a method to solve a problem relies heavily on input data. We see further proof of this if we compare the rows for the 200 by 200 problems where we can see that Microsoft Solver Foundation (MSF) solved all eight problems whilst Qlik Sense Analytics (QSA) failed to solve one. Then in the rows for the 1000 by 1000 instances we see that MSF failed all while QSA solved 4 of them. This is quite clear since larger sizes makes solving the linear system of equations much harder, making each branch of both MSF and Gurobi (GRB) more expensive. But with the QSA implementation based on Implicit Enumeration we don’t solve any linear system but in return it is more likely to visit more nodes since our bounding function is worse.

We can see another interesting property for the implicit enumeration algorithm from figures 5.5, 5.6 and 5.7 which is the fact that the number of branches increases with the optimal value for QSA. This can be explained by the nature of the a best-first search, meaning we have to discard all combinations of cheaper solutions before we reach our optimum. Thus, the bigger the optimal value the more nodes we need to traverse (statistically speaking - chances are still that the problem structure is such that there are only high costs in the field to be optimized which would lead to the optimal value being higher without increasing the number of branches required to come to the solution). As previously mentioned the worst case of implicit enumeration is that the most expensive solution is the only solution which forces us to traverse all nodes.

As we can see both in figures mentioned in the previous paragraph and in the tables of averages in the previous sections, GRB is much better than both QSA and MSF. We cannot argue the fact that MSF is not living up to the standards of GRB but we still are not sure of the fact that GRB is strictly better than our method implemented in QSA. This is due to us not knowing the actual efficiency of one operation if this were to be implemented in the engine. Our intuition is that one operation is much cheaper than solving one linear relaxation, and even cheaper than one simplex iteration. This will especially be true when
5.5 A first analysis

Figure 5.5: Line chart over average number of branches per optimal value for problem of size 50x50.

Figure 5.6: Line chart over average number of branches per optimal value for problem of size 50x100.
5. Method

Figure 5.7: Line chart over average number of branches per optimal value for problem of size 100x100.

the problem size increases. There are actually even some cases where QSA beats GRB in number of operations, even when comparing of branches instead of number of simplex iterations, visualized in figures 5.8, 5.9 and 5.10. These problem instances belong to the set of easier problems and since the implicit enumeration is a greedy algorithm we can expect great result for easier instances and worse for harder instances. We do not include any tests above 100 times 100 since QSA did not finish running them all. But, as mentioned, for the larger problem sizes we will need to investigate the cost of solving a linear system (as done in GRB) versus examining feasibility (as done in QSA) to know the actual performance. We can see this more clearly in figure 5.11 where, despite QSA not finishing all instances and thus not adding those operations to the average, GRB has about a factor 2.3 less iterations than QSA has operations.

All above considered we feel that a continuation in this direction is motivated and we will therefore next look into implementing an Implicit Enumeration algorithm in the QIX engine.

5.6 Implementing Implicit Enumeration in the QIX engine

We have now settled on attempting a combinatorial approach such as a version of the Implicit Enumeration method since we think it fits well with our starting point. This is because the underlying data model also operates in a binary domain, allowing us to use
5.6 Implementing Implicit Enumeration in the QIX engine

Figure 5.8: Average number of branches for GRB and QSA for problems with high demands, high products, low sparsity

Figure 5.9: Average number of branches for GRB and QSA for problems with low demands, high products, low sparsity

calculation tools previously in place which are already optimized for the data structures used to store data in QIDB. Thus, other approaches are prone to cause a lot of overhead: if we start with a method that requires us to solve linear relaxations, we would have to read and extract values from the app, reformat the data for computation, perform calculations, reformat results into selections, send these back to the app and so on. While this is likely better for some problem instances in terms of operations performed, it is clumsy and does not explore whether we can use the QIX engine for efficient optimization.

5.6.1 Qlik configurations

The section in the engine code where we decided to enter the algorithm was located after the construction of the hypercube (remember 4.5 on page 33). This was because it allowed us easy means of access to all necessary data and allowed us to use the sorting provided
5. Method

**Figure 5.10:** Average number of branches for GRB and QSA for problems with low demands, low products, low sparsity

**Figure 5.11:** Comparison between GRB and QSA showing average number of branches for problem sizes 200x200 and 500x500.

by the Qlik Sense interface, which would simplify variable ordering when running the algorithm. We set up an application in Qlik Sense containing a single table with all the relevant data and decided to not include null values for the field containing demand values since a number of our test cases had demands on relatively few products (i.e. few constraints, see section 1.3 on page 14 for an interpretation of the problem used for testing and benchmarking). We did this rather than implementing a null handler since the feature was already available and it represents data we are not interested in anyway, like the quantity of a product not on our shopping list.
5.6.2 Implementation details

The first search heuristic we wanted to test was the best-first search we used in the Qlik Sense Analytics implementation in section 5.4, since this would provide us with an assumption of what speedup to expect when skipping delays stemming from network traffic.

Initially, we started by implementing a heap-based priority queue in C++ based on an std::vector (see [29] for reference on C++ standard library) together with the functions std::push_heap and std::pop_heap (see [29] under <algorithm>). Problem data was maintained by creating an std::map with warehouse IDs as keys and corresponding rows in the hypercube as values. A similar data structure was used to represent demands with product IDs as keys and the demand for that particular product as values. We then proceeded with the best-first search described in section 5.4.1.

Improvements

For larger problem sizes, this approach was sometimes agonizingly slow and we did not get close to the speedup we were expecting. Profiling showed that a lot of time was spent accessing values in the maps as well as in the push and pop operations on the heap. Attempting to remedy this, we speculated that given a good hash function we could simply use std::vector everywhere to gain fast accesses through the operator[] which runs in constant time. In our examples, warehouse IDs and Product IDs were all integer, and lead times from warehouses were also integer forcing all possible solution candidates to have integral cost. Thus, to see if this approach was motivated we could start with an identity hash and simply contain the information about a warehouse with ID \(i\) in position \(i\) of a vector, and store the demand for the product with ID \(j\) in position \(j\) of another vector. Also, since we were exploring the solution tree in best-first order, a cheaper solution could never be added to the queue after all solutions with a given value have been examined. Therefore, a candidate solution with value \(v\) could be registered in position \(v\) of a vector if a variable was maintained to keep track of the value of the nodes we are currently interested in. This would be incremented only when the container has no more candidates of this value.

After performing these optimizations we observed a more satisfying improvement in terms of running speed (see results below). The exact implementation is however dependent on the integrality of the problem, but we suspect that this could be made more general if one would delve into writing specific hash functions for these data types.

5.6.3 Results

A full presentation of the gathered data can be found in [24]. We will here present a summary of the most interesting points, and a table of averages of runtime statistics similar to the ones in previous sections of this chapter can be viewed in table 5.6.

The results for this implementation is similar to those in section 5.4, most easily perceived if looking at the average number of branches which is exactly equal for many of the problem sizes. This was to be expected since we run the same algorithm as we did in the Qlik Sense Analytics section. The reason for some differentiation in number of branches
5. Method

<table>
<thead>
<tr>
<th>Size (warehouses × products)</th>
<th>Average number of branches</th>
<th>Average total time(s)</th>
<th>Unfinished tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 × 10</td>
<td>18.75</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10 × 100</td>
<td>252.625</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20 × 100</td>
<td>1780</td>
<td>0.75</td>
<td>0</td>
</tr>
<tr>
<td>30 × 100</td>
<td>3365.63</td>
<td>1.375</td>
<td>0</td>
</tr>
<tr>
<td>40 × 100</td>
<td>2528.88</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>50 × 50</td>
<td>2353</td>
<td>0.5</td>
<td>0</td>
</tr>
<tr>
<td>50 × 100</td>
<td>9943.88</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>100 × 10</td>
<td>36.625</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100 × 20</td>
<td>581.5</td>
<td>0.125</td>
<td>0</td>
</tr>
<tr>
<td>100 × 30</td>
<td>1141.25</td>
<td>0.125</td>
<td>0</td>
</tr>
<tr>
<td>100 × 40</td>
<td>1950.38</td>
<td>0.375</td>
<td>0</td>
</tr>
<tr>
<td>100 × 50</td>
<td>686.875</td>
<td>0.125</td>
<td>0</td>
</tr>
<tr>
<td>100 × 100</td>
<td>27221</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>200 × 200</td>
<td>203437</td>
<td>146</td>
<td>0</td>
</tr>
<tr>
<td>500 × 500</td>
<td>1319919</td>
<td>1440.14</td>
<td>1</td>
</tr>
<tr>
<td>1000 × 1000</td>
<td>3778242</td>
<td>3326.8</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.6: Averages over different sizes of the warehouse problem solved by best-first search in the QIX engine.

is that the two different implementations explore candidate solutions with the same value in a different order which isn’t specified by the method itself.

Another noteworthy statistic is the time for setting up the model which can be viewed in table 5.7 If we compare this to the time taken to set up the model for Gurobi we see that the time is usually less in QIX and it doesn’t seem to increase at the same rate as for Gurobi. The times in this table are not averages over all setup times but only the most expensive setup time per problem size. These results are based on the time it took to retrieve all relevant data from the hypercube but does not take into account the time the engine took to build the hypercube itself. For the largest examples we ran (4000 × 400), which took a bit less than two minutes which is significant but still less than Gurobi’s setup time for 1000 × 1000 problems.

5.6.4 Proving a point

Having seen these results we note that some problem instances are much easier to solve than others for the implicit enumeration method. This method even beats Gurobi with little to no optimizations on the instances with at least 20% of the demand of each product in warehouses. Furthermore, the largest difference is attained when considering the instance which is harder to satisfy, that is when we have demands on all products.

We will refer to the Implicit Enumeration implementation in QIX engine as QIX. In figures 5.12 and 5.13 we see the time for running the algorithm on Gurobi and QIX on sizes 500x500 and 1000x1000 respectively. In figures 5.14 and 5.15 we see the time spent
5.6 Implementing Implicit Enumeration in the QIX engine

<table>
<thead>
<tr>
<th>Size</th>
<th>Setup time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 × 10</td>
<td>0</td>
</tr>
<tr>
<td>10 × 100</td>
<td>0</td>
</tr>
<tr>
<td>20 × 100</td>
<td>0</td>
</tr>
<tr>
<td>30 × 100</td>
<td>0</td>
</tr>
<tr>
<td>40 × 100</td>
<td>0</td>
</tr>
<tr>
<td>50 × 50</td>
<td>0</td>
</tr>
<tr>
<td>50 × 100</td>
<td>0</td>
</tr>
<tr>
<td>100 × 10</td>
<td>0</td>
</tr>
<tr>
<td>100 × 20</td>
<td>0</td>
</tr>
<tr>
<td>100 × 30</td>
<td>0</td>
</tr>
<tr>
<td>100 × 40</td>
<td>0</td>
</tr>
<tr>
<td>100 × 50</td>
<td>0</td>
</tr>
<tr>
<td>100 × 100</td>
<td>0</td>
</tr>
<tr>
<td>200 × 200</td>
<td>1</td>
</tr>
<tr>
<td>500 × 500</td>
<td>2</td>
</tr>
<tr>
<td>1000 × 1000</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5.7: Most expensive setup times for different sizes of the warehouse problem in QIX.

setting up the model for sizes 500x500 and 1000x1000 respectively.

![Figure 5.12: Time spent on algorithm for GRB and QIX for the problem instance with high demands, high products and low sparsity. Problem size is 500 products and 500 warehouses.](image-url)
5. Method

Figure 5.13: Time spent on algorithm for GRB and QIX for the problem instance with high demands, high products and low sparsity. Problem size is 1000 products and 1000 warehouses.

Figure 5.14: Time spent on setting up the model for GRB and QIX for the problem instance with high demands, high products and low sparsity. Problem size is 1000 products and 1000 warehouses.

As we can see the total time for both running the algorithm and setting up the model is much lower for QIX. We also see that the times increase vastly with problem size for
5.6 Implementing Implicit Enumeration in the QIX engine

Figure 5.15: Time spent on setting up the model for GRB and QIX for the problem instance with high demands, high products and low sparsity. Problem size is 1000 products and 1000 warehouses.

GRB while it only increases a bit for QIX. These results motivate us to run further tests to make the point even more clear.
5. Method

Larger problems

We decided to produce even larger problems of the same type as in the previous section. In these tests we have 2000x2000 and 4000x4000 products times warehouses with high and low demands. They are produced in the same way as explained in section 5.1. Note that for comparison we take the number of simplex iterations made by GRB rather than the number of nodes for reasons explained earlier. The results of these tests can be viewed in table 5.8.

<table>
<thead>
<tr>
<th>Size</th>
<th>Demands</th>
<th>Type</th>
<th>Branches</th>
<th>Setup Time (s)</th>
<th>Algorithm Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000 × 2000 Low</td>
<td>QIX</td>
<td>6304</td>
<td>9</td>
<td>8</td>
<td>203</td>
</tr>
<tr>
<td></td>
<td>GRB</td>
<td>38693</td>
<td>588</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000 × 2000 High</td>
<td>QIX</td>
<td>5855</td>
<td>35</td>
<td>29</td>
<td>758</td>
</tr>
<tr>
<td></td>
<td>GRB</td>
<td>41467</td>
<td>671</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000 × 4000 Low</td>
<td>QIX</td>
<td>21100</td>
<td>35</td>
<td>54</td>
<td>1922</td>
</tr>
<tr>
<td></td>
<td>GRB</td>
<td>121169</td>
<td>2583</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000 × 4000 High</td>
<td>QIX</td>
<td>17451</td>
<td>138</td>
<td>176</td>
<td>3658</td>
</tr>
<tr>
<td></td>
<td>GRB</td>
<td>84156</td>
<td>2734</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: Runtime statistics for QIX and GRB for large input sizes of the warehouse problem.

The statistics have been visualized for easier comparison in figures 5.16, 5.18, 5.19 and 5.20.

Note that the setup time for Gurobi is much higher for these even larger examples. In the 4000x4000 example, the setup time is as large as the algorithm time.

5.7 Another method for Implicit Enumeration in the QIX engine

Up until now we have blindly selected depth first search as our search heuristic for ease of testing. To put these results in perspective, we decided to compare them to another search method, and chose to look into the search heuristic mentioned in section 2.4.3.

Since this method uses a type of infeasibility pruning as well as a better heuristic for node selection the hypothesis is that it will perform worse on easy problems but better on the harder problems. The reasoning is that the cost for heuristics and pruning are not worth paying when dealing with an easy example while it could lead to narrowing down the optimal solution quicker when the solution is expensive.

5.7.1 Implementation details

Recall that the algorithm itself was presented in section 2.4.3.
Figure 5.16: Time spent on algorithm for GRB and QIX for the problem instance with high demands, high products and low sparsity. Problem size is 2000 products and 2000 warehouses.

Figure 5.17: Problem instance with 4000 products and 4000 warehouses.

Figure 5.18: Time spent on algorithm for GRB and QIX for the problem instance with high demands, high products and low sparsity. Problem size is 4000 products and 4000 warehouses.
5. Method

Figure 5.19: Time spent on setting up the model for GRB and QIX for the problem instance with high demands, high products and low sparsity. Problem size is 4000 products and 4000 warehouses.

Figure 5.20: Time spent on setting up the model for GRB and QIX for the problem instance with high demands, high products and low sparsity. Problem size is 4000 products and 4000 warehouses.
In the same way as for the best-first search we hash the demands and the warehouses rows in vectors. The difference between best-first search and this implementation is that we don’t need the priority queue since the next node to be visited is to be calculated depending on the current state of the variables. We will bear in mind that the simple hash that we did in best-first search will increase it’s performance vastly while the calculation of the next node is probably not fully optimized.

We also changed the way we retrieve data from the hypercube. Instead of fetching data from the hypercube every iteration we store all data in our own matrix before we run the algorithm. It turns out that it is quite slow to retrieve data from the hypercube since it is not meant to be retrieved more than once per hypercube creation.

### 5.7.2 Results

A full presentation of the gathered data can be found at [24]. We will here present a summary of the most interesting points, and a table of averages of runtime statistics similar to the ones in previous sections of this chapter can be viewed in table 5.9.

<table>
<thead>
<tr>
<th>Size (warehouses × products)</th>
<th>Average number of branches</th>
<th>Average total time(s)</th>
<th>Unfinished tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 × 10</td>
<td>21.75</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10 × 100</td>
<td>75</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20 × 100</td>
<td>524.5</td>
<td>0.125</td>
<td>0</td>
</tr>
<tr>
<td>30 × 100</td>
<td>2010.25</td>
<td>0.875</td>
<td>0</td>
</tr>
<tr>
<td>40 × 100</td>
<td>1580</td>
<td>0.625</td>
<td>0</td>
</tr>
<tr>
<td>50 × 50</td>
<td>1372</td>
<td>0.25</td>
<td>0</td>
</tr>
<tr>
<td>50 × 100</td>
<td>7968</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>100 × 10</td>
<td>40.75</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100 × 20</td>
<td>680.25</td>
<td>0.25</td>
<td>0</td>
</tr>
<tr>
<td>100 × 30</td>
<td>584</td>
<td>0.25</td>
<td>0</td>
</tr>
<tr>
<td>100 × 40</td>
<td>1265.75</td>
<td>0.25</td>
<td>0</td>
</tr>
<tr>
<td>100 × 50</td>
<td>1006.5</td>
<td>0.25</td>
<td>0</td>
</tr>
<tr>
<td>100 × 100</td>
<td>17913</td>
<td>6.875</td>
<td>0</td>
</tr>
<tr>
<td>200 × 200</td>
<td>60583.8</td>
<td>42.375</td>
<td>0</td>
</tr>
<tr>
<td>500 × 500</td>
<td>744523</td>
<td>1012.43</td>
<td>1</td>
</tr>
<tr>
<td>1000 × 1000</td>
<td>1788200</td>
<td>2326.75</td>
<td>3</td>
</tr>
</tbody>
</table>

**Table 5.9:** Averages over different sizes of the warehouse problem solved by an alternative search heuristic in the QIX engine (QIE).
5.8 A comparison between the different QIX implementations

The two search heuristics are quite different and we need to compare the test results to be able to analyze the heuristics.

It also turns out that the change we made in the alternative search heuristic where we don’t fetch data directly from the hypercube had quite a significant impact on the running time for the algorithm. The points previously proven when we compare Gurobi with the best-first search are still valid since Gurobi had a lower average runtime than our QIX implementation even though the QIX implementation did not finish the hardest tests. Therefore, to be able to compare the two QIX implementations we need to run the tests for best-first search again. We will settle for presenting the comparison between the largest examples, which are the most relevant ones anyway. We now call the best-first search implementation QIX2 (since it is updated) and the alternative search heuristic QIE (QIX Implicit Enumeration). In tables 5.10, 5.11, and 5.12 we can see a comparison between the different problems in sizes 200x200, 500x500 and 1000x1000.

We can see that the 200x200 size looks promising for the alternative search, but then in the larger examples the best-first search is faster in every test, and especially the hard ones. See specifically the row three and 11 in table 5.12 where the best-first search completes in half the time it takes for the alternative search. The alternative search didn’t even complete more tests than the best-first search. For completeness we include the easy 2000x2000 and 4000x4000 problems from section 5.6.4 in tables 5.13 and 5.14. We can see that QIE performs much worse on these tests but that is also the expectation due to the problems being easy.

Statistics for the number of branches and algorithm time for these methods are visualized in figures 5.21 and 5.22.
## 5.8 A comparison between the different QIX implementations

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
<th>Branches</th>
<th>Setup Time(s)</th>
<th>Algorithm Time (s)</th>
<th>Optimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>QIX2</td>
<td>200x200</td>
<td>Demands</td>
<td>Sparsity</td>
<td>Products</td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>20</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>99</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>4293</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>169891</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>56</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>413</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
<td>High</td>
<td>54999</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>1397726</td>
<td>3</td>
<td>421</td>
</tr>
</tbody>
</table>

| QIE    | 200x200 | Demands | Sparsity | Products |
| Low    | Low  | High     | 16         | 1     | 0       | 2          |
| Low    | Low  | Low      | 98         | 0     | 0       | 3          |
| Low    | High | High     | 2502       | 0     | 1       | 6          |
| Low    | High | Low      | 99874      | 1     | 25      | 10         |
| High   | Low  | High     | 74         | 3     | 0       | 3          |
| High   | Low  | Low      | 364        | 3     | 1       | 4          |
| High   | High | High     | 60552      | 2     | 44      | 9          |
| High   | High | Low      | 321190     | 3     | 255     | 11         |

**Table 5.10:** Runtime statistics comparison between QIX2 and QIE for a problem of size 200x200.

![Average algorithm time over test cases for the different search heuristics for implicit enumeration.](image)

**Figure 5.22:** Average algorithm time over test cases for the different search heuristics for implicit enumeration.
<table>
<thead>
<tr>
<th>Method Size</th>
<th>Demands</th>
<th>Sparsity</th>
<th>Products</th>
<th>Branches</th>
<th>Setup Time(s)</th>
<th>Algorithm Time (s)</th>
<th>Optimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>QIX2 500x500</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>155</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>380</td>
<td>6</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>228993</td>
<td>5</td>
<td>51</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>4180226</td>
<td>5</td>
<td>1022</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>717</td>
<td>21</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>11699</td>
<td>20</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>4817265</td>
<td>21</td>
<td>2535</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>QIE 500x500</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>48</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>570</td>
<td>6</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>283930</td>
<td>5</td>
<td>138</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>1801056</td>
<td>5</td>
<td>1100</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>1272</td>
<td>20</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>11680</td>
<td>20</td>
<td>39</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>3113104</td>
<td>20</td>
<td>5724</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

**Table 5.11:** Runtime statistics comparison between QIX2 and QIE for a problem of size 500x500.
### 5.8 A comparison between the different QIX implementations

<table>
<thead>
<tr>
<th>Method Size</th>
<th>Type</th>
<th>Branches</th>
<th>Setup Time(s)</th>
<th>Algorithm Time (s)</th>
<th>Optimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demands Sparsity Products</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>QIX2 1000x1000</strong></td>
<td>Low Low High</td>
<td>1110</td>
<td>20</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Low Low Low</td>
<td>17502</td>
<td>20</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Low High High</td>
<td>18818268</td>
<td>20</td>
<td>8500</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Low High Low</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>High Low High</td>
<td>1209</td>
<td>77</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>High Low Low</td>
<td>53122</td>
<td>77</td>
<td>78</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>High High High</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>High High Low</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td><strong>QIE 1000x1000</strong></td>
<td>Low Low High</td>
<td>2012</td>
<td>20</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Low Low Low</td>
<td>27138</td>
<td>20</td>
<td>59</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Low High High</td>
<td>14168994</td>
<td>20</td>
<td>17272</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Low High Low</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>High Low High</td>
<td>2172</td>
<td>78</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>High Low Low</td>
<td>99220</td>
<td>79</td>
<td>994</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>High High High</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>High High Low</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 5.12:** Runtime statistics comparison between QIX2 and QIE for a problem of size 1000x1000.

<table>
<thead>
<tr>
<th>Method Size</th>
<th>Type</th>
<th>Branches</th>
<th>Setup Time(s)</th>
<th>Algorithm Time (s)</th>
<th>Optimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demands Sparsity Products</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>QIX2 2000x2000</strong></td>
<td>Low Low High</td>
<td>6304</td>
<td>80</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>High Low High</td>
<td>5855</td>
<td>341</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td><strong>QIE 2000x2000</strong></td>
<td>Low Low High</td>
<td>12012</td>
<td>78</td>
<td>115</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>High Low High</td>
<td>11126</td>
<td>337</td>
<td>465</td>
<td>3</td>
</tr>
</tbody>
</table>

**Table 5.13:** Runtime statistics comparison between QIX2 and QIE for a problem of size 2000x2000.
Regarding the algorithm time QIE having a more complex algorithm means it has more possibilities for optimization. Also, the fathoming and augmenting steps both present the possibility to plug in different methods entirely.

Furthermore, the visualizations in figures 5.23 and 5.24 show that the average algorithm time is reduced by about a factor two, while the setup time is increased by around a factor 10 when implementing the temporary hypercube. Although, the total setup time is so much smaller than the algorithm time that it is a cheap price to pay and we see, in figure 5.25, that the total time is barely affected by the increase in setup time. These visualizations is only showing the larger test cases, i.e. 200x200 and larger since they are the most relevant ones to optimize.

**Figure 5.23:** Comparison between average algorithm time for all test cases larger than 200x200. QIX fetches data from the hypercube while QIX2 loads the data into a temporary hypercube.
5.8 A COMPARISON BETWEEN THE DIFFERENT QIX IMPLEMENTATIONS

**Figure 5.24:** Comparison between average setup time for all test cases larger than 200x200. QIX fetches data from the hypercube while QIX2 loads the data into a temporary hypercube.

**Figure 5.25:** Comparison between average total time (algorithm + setup) for all test cases larger than 200x200. QIX fetches data from the hypercube while QIX2 loads the data into a temporary hypercube.
<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
<th>Branches</th>
<th>Setup Time(s)</th>
<th>Algorithm Time (s)</th>
<th>Optimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>QIX2</td>
<td>Demands</td>
<td>Sparsity</td>
<td>Products</td>
<td>Demands</td>
<td>Sparsity</td>
</tr>
<tr>
<td>4000x4000</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>21100</td>
<td>343</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>17451</td>
<td>1272</td>
</tr>
<tr>
<td>QIE</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>41014</td>
<td>338</td>
</tr>
<tr>
<td>4000x4000</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>33732</td>
<td>1292</td>
</tr>
</tbody>
</table>

Table 5.14: Runtime statistics comparison between QIX2 and QIE for a problem of size 4000x4000.
Chapter 6
Discussion

We have come to the one of the last parts of this thesis, which consists of discussing and analyzing the results as well as the goals we presented ourselves with. Recall that one of our goals was to present possibilities for further development. This is treated in chapter 7, where we look to what the future may have in store.

6.1 Measurement of success

We have completed most of the goals for this thesis but some could still be examined in further detail. If you remember our sub goals you will also see that we have completed most of them:

- We came up with an algorithm that reasoned well with the QIX engines structures and also finds an optimal solution to the 0-1 integer programming problem. Then we tested this algorithm against other optimizers in the QSA tool and motivated that further improvements would not be wasted.

- Then we implemented said algorithm in the QIX engine and optimized it before running new tests against Gurobi, the optimizer which seemed most fit for the case.

- As mentioned above, future work will be presented in the next chapter and is therefore also considered completed.

In addition to this we decided to implement a different search heuristic for Implicit Enumeration which was compared to the first search heuristic. We felt like this was a good idea since it looked like we were just choosing our algorithm out of the dark.

On the other hand, the one goal that we did not succeed as well with is the one where we were supposed to exploit Qlik’s structure to get an edge over the commercial solvers. This is left as future work due to time running out but we have had it in mind all along and everything that the thesis has presented lines up with this idea.
6.2 Examining the results

Our hypothesis regarding the alternative search method for the implicit enumeration being faster for harder examples was contradicted. Consider for example the 1000x1000 problem with low demands, high sparsity and high products which took 16410s with best first search and 17272s for the alternative search heuristic. The difference is not that big relatively speaking but we did expect all sparse problems to go a lot faster than with the best first search. We still think our line of reasoning is valid and perhaps we did not do enough optimizations in the alternative search heuristic to compare to the hashed priority queue in the best first search method. Another source of error could be that the problem type is the same - we only solve the warehouse problem. Maybe this search heuristic would be more suited for a different type of problem with a different structure. Even though the matrix is sparse we can find quite cheap solutions since we guarantee feasibility.

We do get to beat Gurobi on the easier problem instances as proven in section 5.6.4 but Gurobi still finished running, proving that it is a very robust tool.

6.3 The hash used with the best-first search heuristic

We mentioned that we believe that it is possible to create good hash functions even when the values of our warehouses are not as compliant as they were but this is not scientifically proven. The map structure in C++ seemed to have some performance issues which may or may not be related to the hashing. Further testing needs to be done to know for sure, and if it turns out that we cannot hash the values cleverly then the performance for this heuristic is reduced significantly.

6.4 Number of branches as a unit of measurement

If you recall chapter 5 we motivated further implementation in the QIX engine via the QSA tool. There we made the assumption that one simplex iteration was much more expensive than one branch in our implicit enumeration algorithm. In hindsight that is probably not entirely correct. One simplex iteration in GRB seems to be one pivot, while one branch (node) in GRB consists of several pivots. So one simplex iteration in GRB is $O(nm)$ where $n$ is the number of variables and $m$ is the number of constraints. In our implicit enumeration we first thought that each branch would basically be trivial and almost constant time, but in fact we do a feasibility check each iteration which needs to check whether all constraints are fulfilled. This is also $O(nm)$. Of course in reality we probably don’t need to check all constraints every time since it is enough for one constraint to be unsatisfied for the system to be unsatisfied which makes our branches a bit cheaper. But we definitely expected a greater speedup in comparison to GRB due to this incorrect assumption.

Number of branches becomes an even worse unit of measurement when we implemented the alternative search heuristic. There every branch becomes much more expen-
sive due to the infeasibility pruning and node selection heuristic. We still included the measurement for consistency though.

6.5 Fetching data from the hypercube

Since the hypercube in Qlik is not designed for this type of use, fetching data from it is quite expensive. The purpose of the hypercube is really just to display the wanted values which is fetched once every time we build it. In our case we fetch data from the hypercube hundreds of thousands of times (at least once in each iteration).

The first thing to note is that we did not include the time spent building the hypercube for Qlik. This happens before any of our own implementation takes effect and takes a significant amount of time (about 100 seconds for a 4000x4000 example). We justify not including this measure since we would have to build the hypercube regardless of which implementation we would have chosen.

Since the fetching of data from the hypercube was slow we implemented a temporary hypercube which basically is a matrix where we store all the data from the hypercube for faster access. The time that this takes is dependent of the size of the constraint matrix (that is, dependent on the number of demands and warehouses). This makes setting up the model a bit more expensive for larger problems and that cost is unnecessary for the easy problems where we have fewer iterations and thus fetch less data from the hypercube. The results in section 5.8 showed a reduction of the algorithm time by around a factor two. Whether we want to build the temporary hypercube or not is therefore a question of how large and hard the problem is, and finding out if the problem is hard in beforehand might be tricky. But, we might have some plans to remedy this in the future that you can read about in chapter 7.

This hypercube optimization is something we found out about very late and it is thus only showing in the results of the alternative search heuristic implementation. We did rerun the tests with the new optimization when comparing the two search heuristics against each other which seemed like the only reasonable course of action. But, the reader should be wary of the fact that the comparisons between the best-first search and Gurobi are all made with the old implementation which means that algorithm time is slower but the setup time is longer. The total running time on the other hand is vastly decreased for the largest examples and that would have given us a better standpoint against Gurobi but it would most likely not have mattered since Gurobi was still beating our algorithm. Also recall the comparison of setup times in chapter 5. There we came to the conclusion that QIX was better than Gurobi but with the new implementation the numbers would have been more in the favor of Gurobi.

What we should have done was to make two different comparisons with Gurobi, one with the old implementation and one with the new to really see the differences and contrasts. But as noted time was just not enough.
6.6 The different problem difficulties

Even though the different tools and methods we used were vastly different from each other, they agreed unanimously on which problems were harder. The three parameters we tweaked to represent different problems were $A$ matrix sparsity, $b$ vector cardinality and $A$ matrix lower bounds (see definition 1.2 on page 14 for notation specifics). All solvers agreed that every problem with a sparse $A$ matrix were harder than all problems with a full one, no matter the settings of the other parameters. The second most important factor seemed to be the lower bound of values in $A$, but in Gurobi it did not differ much.

6.7 The randomness of the problems

The data used in the problems were randomly generated. This might not actually reflect reality where data is more likely to show trends and cluster. That is a fact to consider when drawing conclusions - would any method have a greater advantage on "real" data? It would probably not affect Gurobi since it uses solves the linear relaxation, but consider data where the cost (time) of the warehouse increases with the number of products contained in the warehouse. That is, a larger warehouse is more expensive to fetch products from. This is quite reasonable - it might be located further away or the product may be difficult to find etc. In this case the best-firsts search implementation would be worse since the cheaper solutions are less likely to satisfy the problem by themselves (or in small subsets), while the alternative search heuristic would pick and consider the warehouses that satisfy the most unsatisfied constraints first. For example if the only feasible solution is the most expensive one best-first would have to explicitly enumerate all combinations while the alternative search heuristic would find the feasible solution quite quickly and would be able to discard a lot of nodes through its infeasibility pruning.

6.8 The smaller testcases

In the result section we have presented runtime statistics for many small examples such as $10 \times 10$ as well as $10, 20, 30, 40$ and $50 \times 100$ and the other way around. This provided us with some useful information: we found a possible difference between enumerating methods and simplex-based solvers when we had a noticeable disparity in the amount of products and warehouses, see figure 5.4 on page 46. Even so, these tests turned out not to give us much relevant information to analyze but we decided to keep them for the completeness of the thesis sake. In the earlier stages of this thesis it seemed like an impossible task to even solve $100 \times 100$ problems and we thus wanted to compare some different examples where the warehouses and products varied between 10 and 100. In hindsight we probably should at least had larger incrementations to be able to see any pattern. In figure 6.1 you can see two line charts over the relations between increasing number of products and warehouses respectively. As you can see the algorithm time seems to be increasing with the number of warehouses (which we have already shown with larger examples) and the algorithm time depending on the number of products seem totally random. Also the absolute difference between times is too small to draw any conclusions.
6.9 Conclusions

(a) Number of products ranging from 10 to 50.

(b) Number of warehouses ranging from 10 to 50.

Figure 6.1: Line chart over algorithm time depending on increasing number of warehouse or products.

On the other hand we could also note that running too many large test cases simply would take too much time.

6.9 Conclusions

While testing and benchmarking, we made use of the tools and methods you are likely familiar with at this point. We will attempt to give an overview of our observations as well as possible explanations of these conclusions.
6.9.1 LP Solvers

Transitioning between the two off-the-shelf LP solvers we used, Gurobi and Microsoft Solver Foundation, was simple and they proved to be very similar in terms of usage.

**Gurobi**

After all our testing, Gurobi solved the most problems of all tools we tried, and was the fastest in general. We did, however, discover some weaknesses after using it so much:

- Setting up the model takes relatively long compared to QIX implementations. This is partly because the data already is present in the database. This is one potentially great source of error since we did not consider that the reading of the input files would take very long time when measuring the setup time for Gurobi.

- It cannot detect irrelevant data since it has no built-in inference. An example is the information on all data present in no constraint - in a Qlik Sense app we could simply hide these values and ignore them for computational purposes, but Gurobi had to load them and explicitly set up its equivalent of simplex tableaux. This may not be relevant in practice since you would be unlikely to feed irrelevant data to your solver, but it is possible in a database environment and easy filtering was present.

- Easily satisfiable problems, which are likely to have a cheap solution, are equally hard to find as any other solution for Gurobi. This becomes relevant when dealing with large problem sizes. I.e. it doesn’t change its solving behaviour depending on the size of the problem even when large problem sizes hurt the running time as much as it does.

**Microsoft Solver Foundation**

Even after disabling many features in Gurobi such as multithreading and heuristic solutions, it was still faster and more clever than Microsoft Solver Foundation. Not only did each individual iteration take more time, but we typically visited more branches for all problem types. The only potential advantage we observed with using this tool over Gurobi was that the output was more easily interpreted - with Gurobi you had to know what you were looking for.

6.9.2 Enumeration

As was mentioned in section 2.1 on page 17, enumeration methods are based on relaxing the problem constraints instead of relaxing constraints on integrality. When performing linear relaxations, fewer branches are typically visited when more variables already take integer values in the optimal solution. We observed a similar trait in constraint relaxation: if the constraints were easily satisfiable, the feasible region is larger and more candidate solutions were feasible leading to fewer branches.
Best-first Search

Best-first search is a greedy algorithms with all its possible benefits and limitations. Implemented in QIX, setting up the model was more or less instant. For smaller problems as well as problems with a large portion of candidate solutions being feasible, it had stellar performance and was able to solve many problems quickly that took quite a while for the LP solvers.

One crucial analysis that we have presented previously and will present once again is that the runtime of the best first search algorithm increases with the optimal value of the solution (or rather the difficulty of the problem). This makes sense since the algorithm is greedy and visits the nodes in an increasingly expensive order, hoping to find one which satisfies the constraints. So, if we have a very cheap feasible solution, the best first search will find it very quickly but the number of nodes visited increases exponentially - an expensive solution will most likely not be reached within a reasonable time.

Alternative search heuristic

While it outperformed best-first search on some problem instances, we observed the same weaknesses in both schemes. We do believe that further testing of this kind of method would be motivated since we suspect that the fathoming part and selection of augmenting variables could be made more clever and efficient.
6. Discussion
Chapter 7
One qlick away from the future

In this final chapter we will present concepts which could be developed in the future due to this thesis. Showing that there is potential for optimization in associative data models opens up for subjects which exploits the associative data model to get an edge over the commercial products when considering specific problem structures. We will present two main concepts which would allow Qlik to get ahead of the competition in the business intelligence area.

7.1 Exploiting data dependencies

The associative data model excels in dependencies between data and there are a lot of features in Qlik that could have been given to us for free but that we have not yet exploited. When dealing with large data it can be quite costly both in time and memory to reconstruct the underlying data. That is for example, to select a sub-set of the data via joins (relational database) or through the inference machine (Qlik’s associative model).

Thus we consider the following model:

\[
\begin{align*}
\text{minimize} & \quad c_1^T x + c_2^T y \\
\text{s.t.} & \quad \begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \geq \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \\
& \quad x \in \{0,1\}^s, \\
& \quad y \in \{0,1\}^t, \\
& \quad b_1 \in \mathbb{R}^u, \\
& \quad b_2 \in \mathbb{R}^v,
\end{align*}
\]

where \(b_1\) are the constraints on \(x\) and \(b_2\) the constraints on \(y\). Also we have \(s + t = n\) variables and \(u + v = m\) constraints with \(s \ll t, u \ll v\). Furthermore we have that \(\exists i, j\) such that \(y_i \in y\) depends on \(x_j \in x\) in such a way that if we choose \(x_j\) we must choose \(y_i\) as well.
This dependency can not be represented in a linear way and we already have an advantage against LP solvers. For the sake of argument we will assume that it is possible to set up these nonlinear constraints and solve the system by for example using a different solver. Since this thesis hasn’t focused on nonlinear programs we will just speculate that solving nonlinear programs is at least as hard as solving ILP and that the difficulty also increases with the size of the problem. Then we end up with a system that is at least as hard to solve (reasonably it should be a lot harder) as the ILP with $n$ variables and $m$ constraints for solvers like Gurobi, and as this thesis has covered the difficulty of these problems increase vastly with the size of the problem and already at $n = m = 1000$ optimization takes a couple of hours. But, since $x \ll t$, $u \ll v$ and $y$ depends on $x$ like previously mentioned we can construct problems that would be much easier to solve for the Implicit Enumeration algorithm integrated in the QIX engine.

As a concrete example take $x$ as warehouses, $y$ as products, $c_1^T x$ as the cost for selecting warehouses and $c_2^T y$ as the profit gained from selecting products. With $c_2^T \leq 0$ we want to minimize the costs (profits being negative costs). Then the $A_1$ matrix will represent constraints on which combinations of warehouses that we may select. For example we might not be able to use both warehouse one and two since they both take up too much space. The $A_2$ matrix represents production constraints on the different products. If we produce product one we might not be able to produce product two as well since resources are already used for product one. To make the model contain dependencies between $x$ and $y$ we say that certain products may only be stored in certain warehouses. The connection to reality might be that one warehouse stores construction material and is therefore not suited to store food as well.

### 7.2 Don’t build the hypercube

As mentioned briefly in section 4.5 the hypercube is what uses memory in Qlik’s applications. Our current implementation in the QIX engine from chapter 5 lets QIX build the hypercube for us before running our algorithm. When handling big data, building the hypercube can be costly in regards to both time and space so avoiding building it when just doing an optimization (we don’t need to see the intermediate results) is preferred. By further investigation in exactly how QIX builds the hypercube we could easily fetch just the relevant data and save both memory and unnecessary operations.

As seen in the analysis in chapter 6 we can save ourselves a lot of trouble by not building the hypercube due to the building of temporary hypercubes.

The last thing to point out is that if we want to handle problems of the type mentioned in the previous section, with data dependencies, we will need to construct several hypercubes - causing a lot of overhead. This could also be avoided when not building the hypercube.

### 7.3 AND/OR Search Tree

The AND/OR search tree is one possible extension to a Implicit Enumeration algorithm, and it was suggested by R.Marinescu and R.Dechter in [15] to have potential for great performance improvements for 0-1 ILPs. The idea is that we can reduce the search space
by grouping subtrees based on the context of the node to be expanded. We will present an overview of the method to create an AND/OR search tree. For a more detailed look we refer to the article.

7.3.1 The idea

**Definition 7.1.** Given an 0-1 ILP instance, the interaction graph $G$ is the graph that has a node for each variable and an edge between each node whose variable appear in the scope of the same constraint.

**Definition 7.2.** Given an undirected graph $G = (V, E)$, a directed rooted tree $T = (T, E')$ defined on all its nodes is called a pseudo-tree if any arc of $G$ which is not included in $E'$ is a back-arc, namely it connects a node to an ancestor in $T$.

Consider the 0-1 ILP in figure 7.1(a) where we have four constraints: $F_1(A, B, C)$, $F_2(B, C, D)$, $F_3(A, B, E)$ and $F_4(A, E, F)$. We can then construct a pseudo-tree from the interaction graph for this ILP which will look like in figure 7.1(b) (dotted lines are back-arcs). Given the 0-1 ILP instance, its interaction graph $G$ and a pseudo-tree of $G$, we can construct the AND/OR search tree seen in figure 7.1(c).

![Figure 7.1: The AND/OR search space.](image)

$$\text{minimize: } z = 7A + 3B - 2C + 5D - 6E + 8F$$

subject to:

- $3A - 12B + C \leq 3$
- $-2B + 5C - 3D \leq -2$
- $2A + B - 4E \leq 2$
- $A - 3E + F \leq 1$
- $A, B, C, D, E, F \in \{0, 1\}$
The root of the AND/OR search tree is an OR node, every child to an OR node is an AND node and vice versa. We label OR nodes with the corresponding variable to be branched on, \( X_i \). We label AND nodes with the corresponding variable branched on along with the value assigned, \((X_i, x_i)\).

**Definition 7.3.** Given a 0-1 ILP instance and the corresponding AND/OR search tree \( S_T \) relative to a pseudo-tree \( T \), the context of any AND node \((X_i, x_i) \in S_T\), denoted by \( \text{context}(X_i) \), is defined as the set of ancestors of \( X_i \) in \( T \), including \( X_i \), that are connected to descendants of \( X_i \).

Looking at the figure in 7.1(b) we see that the contexts are: \( \text{context}(A) = \{A\} \), \( \text{context}(B) = \{B, A\} \), \( \text{context}(C) = \{C, B\} \), \( \text{context}(D) = \{D\} \), \( \text{context}(E) = \{E, A\} \) and \( \text{context}(F) = \{F\} \).

**Definition 7.4.** The context-minimal AND/OR search tree is obtained by merging all AND nodes that have the same context.

We end up with the context-minimal AND/OR search tree in figure 7.1(d). For example we have merged the nodes for \( E = 0 \) and \( E = 1 \) in the subtrees where \( A = 0 \) and \( B = 0 \) or \( B = 1 \) respectively since \( \text{context}(E_0) = \{E, 0\} \). We cannot merge these nodes with any E node in the subtree where \( A = 1 \) since they would have \( \text{context}(E_1) = \{E, 1\} \). They are on the other hand merged with each other. As we can see the context minimal tree has far fewer nodes than the original tree (16 nodes vs. 36 nodes), making traversal faster at the cost of memory.

**7.3.2 The benefits**

At the cost of memory we can increase efficiency vastly. Obviously, since we need to store basically every unique branch with its context there will be additional memory costs. On the other hand we could fathom some subtrees in constant time! However we are not certain on how large of an impact this will have in practice, since it is very dependent on how the constraints look, and we leave that for future work to measure.
Work process

In this section we aim to explain our working process and relevant courses of actions which potentially have modified our goals and objectives.

Preparation

Starting with this thesis we did not have much real background of the problem presented to us by our supervisor José at Qlik. What we did know was that the area optimization and mixing mathematics with computer science felt very intriguing. So our initial motivation was something along the lines of "solving this problem is hard and takes a lot of time for large input instances" along with the goal to create an extension to Qlik’s product which solves the problem within a reasonable time frame, even for large inputs. Thus the goal was to create a new or improve an existing algorithm for integer programming with focus on creating a new algorithm which is superior to existing algorithms.

Theory and background

The first question we encountered when starting this project was: "What is an optimization problem?". While this may seem trivial to a person with some mathematical background it was a useful starting point for our research. In the end, we were concerned with solving this type of problems, so we should know exactly what they are. To answer this seemingly simple question, we needed more concrete information and theory regarding integer programming, linear programming and optimization in general. This is when we encountered and explored Simplex as well as other methods for solving linear programs. We mainly focused on the Simplex method and tried to learn how and why it worked by doing a lot of calculations by hand. Concurrently we continuously had meetings with José regarding the fundamentals of the Qlik software to try and get an overview of the flow of data from the database to a visualization on the screen.

At this point in time we had not yet fully understood what it was that we wanted to do in this thesis or even the real life use case of the thesis. This became clear when trying to
explain the goals of the thesis to our LTH supervisor, Thore. We went back to the drawing board and tried to generate a more concrete example of a problem which a customer of Qlik would find useful solving (remember the warehouse problem). After running this example with large inputs we convinced ourselves that solving the matrix equation was hard for large matrices. José was convinced that if it was possible to iterate over possible solutions it would be much easier to solve in Qlik given its associative data model. We found this hypothesis plausible but we still needed more concrete convincing.

Think very hard

The most crucial step of any project is the part where you think very hard until you come up with a solution. After that we just need to implement said solution.

We had a couple of options already to consider. The most boring scenario is where we just take all the data in the QIDB and shove it into an existing solver. This will guarantee optimality but will generate a lot of overhead and it would probably just be worse than solving the problem externally. It is also not very creative. Another option was to limit the search space of the problem. This of removes the guarantee of optimality, but we might be able to find a decent solution very quickly with this method. This idea was quickly discarded after trying out an algorithm where we iteratively generate larger and larger search spaces and terminate after a reasonable amount of time. Apparently trying all combinations of a maximum of $k = 1 \ldots 4$ selections was slower than solving the problem optimally with a commercial solver. The last option was to find a numerical solution to the problem but we would rather have the analytical solution of course.

At this stage we needed more inspiration and decided to read some more about constraint solving and also network programming to see if there was an alternative approach that could suit well with Qlik’s structures (specifically its associative database). After reading about network flow we found that its attributes matched the flow of associations through an associative database very well but we could not transfer our problem directly to a single flow problem. Then we found information regarding the possibility to relax the constraints rather than the integrality of the problem which inspired us to write our own implicit enumeration algorithm. We ended up with the exact same algorithm in terms of node ordering and cutoffs as Balas Additive Algorithm.

After this major breakthrough we redefined the goals and introduction to instead focus on finding an existing algorithm which works well in conjunction with the associative data structure of Qliks engine.

Benchmarking

After implementing the Balas Additive Algorithm with a best first search heuristic algorithm via the SDK Qlik Sense Analytics we needed to compare the results to further motivate our continuation of the thesis. For benchmarking, we wanted the results to be as objective as possible, a difficult task since we observed strong data dependency in the performance of different solvers. We spent some time and thought deciding which types of problems were deemed relevant and expected or proven to have an impact during runtime.
Insofar it was possible, we wanted the test cases to reflect "all possible configurations", or at least a good representation. Some questions we wanted the data to address were "For what types of problems do some methods or some solvers perform particularly well?" "What changes have the most impact on performance?" "For what types of problems do some methods or solvers perform particularly poorly?" "Can some methods or solvers simply be better than others?". We had already run many different tests out of curiosity which served as a thought framework for what properties to consider, but we wanted to standardize these so all approaches could compete on an even footing. This also gave us the opportunity to mass-produce interesting tests in a straightforward and time-efficient manner, providing us with a solid basis for convincing or at least relevant interpretation of results.

**Method**

With benchmarks set it became clear that the route we were taking was promising and so we had some options for the future. The first obvious step was to implement a working version of the implicit enumeration algorithm in Qliks engine. We also needed to run benchmarks on this implementation to see if we get any significant speed-ups. To do this we had to get a crash course in Qliks engine from José. When we got the basic version of the algorithm to work together with the engine we could choose to either focus on improving the algorithm as much as possible (implementing good heuristics, better cutoffs etc.) or look at special structures of the problem that our algorithm excels in solving.

After some thought we decided to go with an alternative search heuristic other than best first search for our implicit enumeration algorithm. It seemed like this would contribute the most to our thesis and seemed reasonable to do within the small time frame that was left. So we implemented the other search heuristic and ran all tests on it and compared it to our previous implementation. It turned out to not be as good as we had hoped for but at least now we know.
Bibliography


Algoritms för optimering öppnar nya möjligheter

Populärvetenskaplig sammanfattning av Daniel Odenbrand, Nils Fagerberg

Vi har testat och analyserat olika metoder och verktyg för att lösa en särskild typ av optimeringsproblem. Framgången har inte varit total, men för vissa problem har vi lyckats slå marknadssledande program och det finns stora möjligheter för vidareutveckling.

Vad är optimering?

Optimering handlar om att hitta den bästa möjliga lösningen på ett problem, och helst ska det gå fort också. Även om detta kan låta enkelt så finns det i praktiken många exempel på riktigt stora och svåra problem från verkligheten där många av de mest kraftfulla metoder och verktyg vi känner till tar lång tid på sig.

Optimering gör skillnad

Qlik är ett lundabaserat företag som jobbar med business intelligence, det vill säga att visualisera och presentera data på ett förståeligt sätt. Deras kunder är ofta företag som vill analysera data för att hitta samband som kan peka på vad som bör förbättras. Optimering över stora datamängder är alltså attraktivt och företagen kan dra stor nytta av att optimaera små detaljer i en storskalig produktion. Tänk dig att Apple får reda på att en del som ingår i alla Iphones kan produceras lite billigare än tidigare och att de tjänar ett par kronor per såld produkt. Eftersom det säljs ca 200 miljoner Iphones på ett år blir det en vinst på nästan en halv miljard per år!

Ett Qlik från framtiden

I nuläget utnyttjar vi inte fördelarna hos Qliks produkt till fullo på grund av den höga kunskapströskeln man måste passera för att förstå allt som händer i Qliks motor, vilket säger oss att framtiden är full av möjligheter för förbättring. En spännande möjlighet finns när viss data hör ihop, d.v.s. att man inte kan svara "ja" på en fråga utan att tidigare ha svarat "ja" på en annan specifik fråga, kan vi dra stor nytta av Qliks produkt som håller koll på beroendet åt oss. Detta är ammonlunda mot om man skulle vilja använda ett kommersiellt program då man måste sätta upp en mycket mer invecklad matematisk modell.

Vårt bidrag


Ett Qlik från framtiden

I nuläget utnyttjar vi inte fördelarna hos Qliks produkt till fullo på grund av den höga kunskapströskeln man måste passera för att förstå allt som händer i Qliks motor, vilket säger oss att framtiden är full av möjligheter för förbättring. En spännande möjlighet finns när viss data hör ihop, d.v.s. att man inte kan svara "ja" på en fråga utan att tidigare ha svarat "ja" på en annan specif

Varje investering.