Fast Solver of Closely Related Quadratic Programming Problems.

Andreas Halle

June 1, 2013

Abstract

Goodtech and MathConsult have developed a tool called PROMAPS. It calculates the power delivery as a function of demand, and the probability for undelivered energy for each load branch in the system, and in the system as a whole. This problem is formulated as several closely related quadratic programming (QP) problems, whose only difference is in the column bounds. A major part of the bottleneck of PROMAPS, has been found to be calls to a QP solver. The aim of this thesis is to develop a fast solver with the characteristics of these QP problems. The characteristics of the QP problems suggest that methods based upon linear programming (LP) are suited for solving them. In this thesis, we present an iterative QP solver using methods based upon linear programming. We also present a tree structured method that reduces the number of calls to a QP solver by a significant amount.

Contents

Chapter 1

Introduction

As electrical transmission systems increase in complexity, it becomes more difficult to understand how different components in the system interact. We often aim to increase the utilization of such systems, and with that comes the need for good planning and operational tools. The Transmission System Operator (TSO) is faced with increasing requirements regarding the reliability of load delivery, and the cost of not delivering agreed energy can be substantial. The need for tools to assist the TSO in analyses that can help prevent unreliable networks is important. Digernes et al. [1] state:

> It is of utmost importance for the TSO to be able to perform detailed and accurate reliability analyses; for the daily operation as well as for future planning (comparison of reinforcement alternatives etc.). By including power flow considerations in the calculations, the operator is able to plan where to locate the spinning reserves to maximize the load delivery reliability. Reliability analyses are also important for system planning, for analyzing different alternatives for network reinforcement's [sic] etc. [1]

Goodtech and MathConsult have developed a tool called PROMAPS (Probability Methods Applied to Power Systems). It calculates the power delivery as a function of demand, and the probability for undelivered energy for each load branch in the system, and in the system as a whole. According to Svendsen et al. [2], a typical calculation sequence in PROMAPS includes the following main functions:

1. Creation of branch reliability models based on unit Markov models and composition and aggregation of states.

2. Selection of a subset of states containing all grid reliability states with a significant reliability.

3. Calculation of maximum power delivery capacity for each of the significant grid reliability states based on an objective function with constraints.

4. Calculation of expected power shortage and creation of delivery reliability models based on the operations composition and aggregation.

5. Calculation of delivery probability, mean visiting duration and visiting frequency for functioning and failed delivery states.

6. Post calculation of various auxiliary variables including economic data.

Among these six functions, the third main function has been identified as the most time-consuming. A major part of the third main function is a call to a QP solver [2].

The maximum power supply can be calculated by an objective function that represents the power delivery profit. A typical objective function is

$$f(x) = -x^T \Phi D x + (c - g)^T x, \qquad (1.1)$$

where $f(x)$ is the objective value (E/s) as a function of x, x is the branch power vector (W), g is a vector containing specific power generation costs (E/J), Φ is a diagonal matrix representing power loss $(1/W)$, D is a diagonal matrix containing specific power transmission costs (E/J), and c is a vector containing specific power delivery price (E/J). The symbols in parentheses are units of measurements, where E denotes the monetary unit, J denotes joule and W denotes watt [1].

We simplify (1.1) to

$$f(x) = x^T H x + b^T x, \qquad (1.2)$$

where $H = -\Phi D$ is a diagonal matrix that represents quadratic cost terms $(E/(W^2 s))$ and $b = c - g$ is a vector that represents linear cost terms (E/J) [1].

The maximum power delivery capacity can be defined as an optimization problem

$$\min_x f(x) \quad \text{subject to } Ax = 0, \ l \le x \le u \qquad (1.3)$$

where A is the grid configuration matrix or reduced incidence matrix, and l and u are the minimum and maximum branch (lower and upper) capacity, respectively [1].

If a branch with index i fails, we can model it by letting $l_i = u_i = 0$ where l_i and u_i denote the ith element of l and u, respectively. We refer to a branch

6

failing as a *breakdown*. We discuss breakdowns in more detail in sections 3.1 to 3.2 and Chapter 5.

In the next chapter, we present a brief introduction to linear and quadratic programming. We then move on to Chapter 3 where we present some instances of the quadratic programming problem (1.2) that Goodtech have supplied. Afterwards, in Chapter 4, we present an iterative optimization method based on Successive Linear Programming (SLP). Then we move on to Chapter 5, where we present methods for eliminating unnecessary work carried out in the third main function of PROMAPS, and thereby improving the speed in which it is performed. We also present implementations of all the aforementioned methods in Chapter 6, along with experiments and results in Chapter 7.

Chapter 2

Linear and Quadratic Programming

In this chapter, we present a brief introduction to LP and QP problems, before we move on to available methods for solving these problems.

2.1 Formulations of LP and QP problems

In linear programming, we aim to maximize or minimize a linear function of some variables x_j for $j = 1, 2, \ldots, n$. We refer to these variables as *decision variables*. The linear function to be optimized is defined as a linear combination of the decision variables:

$$\zeta = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n,$$

and is called the *objective function*, where c_j are known coefficients. This function is maximized or minimized subject to linear *constraints*. These constraints are either equalities or inequalities in a linear combination of the decision variables:

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \left\{ \begin{array}{c} \leq \\ = \\ \geq \end{array} \right\} b.$$

where a_j are known coefficients and b is some given value. The problem of maximizing or minimizing ζ subject to one or more linear constraints is

called a linear programming problem. We can formulate such problems as:

$$
\begin{array}{lrcrcccrcl}
\text{minimize} & c_1 x_1 & + & c_2 x_2 & + & \cdots & + & c_n x_n & & \\
\text{subject to} & a_{11} x_1 & + & a_{12} x_2 & + & \cdots & + & a_{1n} x_n & \leq & b_1 \\
& a_{21} x_1 & + & a_{22} x_2 & + & \cdots & + & a_{2n} x_n & \leq & b_2 \\
& & & & & \vdots & & & & \\
& a_{m1} x_1 & + & a_{m2} x_2 & + & \cdots & + & a_{mn} x_n & \leq & b_m \\
& & & & & x_1, x_2, \ldots, x_n & & & \geq & 0,
\end{array}
$$

where m is the number of constraints and n is the number of decision variables. This formulation is referred to as *standard form* [3]. We can also formulate LP problems in a more compact form:

$$
\min c^T x, \quad \text{subject to } Ax \leq b, \ x \geq 0, \tag{2.1}
$$

where c and x are vectors in $\mathbb{R}^n$, b is a vector in $\mathbb{R}^m$, and A is an $m \times n$ matrix.

Constraints are not necessarily written as *less-than* inequalities, so we have to convert them to get the problem in standard form. An inequality

$$
a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \geq b
$$

might be converted to a *less-than* inequality by multiplying both sides by -1:

$$
-a_1 x_1 - a_2 x_2 - \cdots - a_n x_n \leq -b
$$

An equality constraint can be converted into two inequality constraints. Given the equality constraint

$$
a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = b,
$$

we can convert it into the two inequalities

$$
\begin{aligned}
a_1 x_1 + a_2 x_2 + \cdots + a_n x_n &\leq b \\
a_1 x_1 + a_2 x_2 + \cdots + a_n x_n &\geq b,
\end{aligned}
$$

or equivalently

$$
\begin{aligned}
a_1 x_1 + a_2 x_2 + \cdots + a_n x_n &\leq b \\
-a_1 x_1 - a_2 x_2 - \cdots - a_n x_n &\leq -b.
\end{aligned}
$$

Any combination of values assigned to the decision variables is called a *solution*. If the values satisfy the constraints of the problem, the solution is said to be *feasible*. In contrast, if the solution does not satisfy all the constraints of the problem, the solution is said to be *infeasible*. The set of

all feasible solutions is called the *feasible region*[1] [4]. A feasible solution that attains the minimum (or maximum if we are maximizing) objective value among all the feasible points is called an *optimal solution*.

A convex polytope may be defined as an intersection of half-spaces, and a system of linear inequalities may be regarded as an intersection of half-spaces [5]. Hence, the feasible region may be regarded as a convex polytope. Note that the feasible region might be empty, i.e. there are no feasible points. In that case, the linear programming problem itself is considered infeasible. There is also the concept of *unbounded* problems, where we can attain an arbitrarily large objective value while still remaining feasible.

Consider the following problem:

$$
\begin{aligned}
\text{minimize} \quad & -2x_1 - x_2 \\
\text{subject to} \quad & x_1 + x_2 \leq 3 \\
& x_1 - x_2 \leq 1 \\
& -x_1 + 3x_2 \leq 4 \\
& x_1, x_2 \geq 0.
\end{aligned}
$$

Since the problem is in only two variables, we can illustrate the feasible region in two dimensions. Figure 2.1 shows an illustration of the feasible region of the LP problem above. It also shows the level sets of the objective function from left to right when the function takes the values 0, -2.5 and -5, respectively. The rightmost gray line represents the objective function when it takes its minimum value. Our optimal solution in this example is $(2, 1)$, and the objective value is -5.
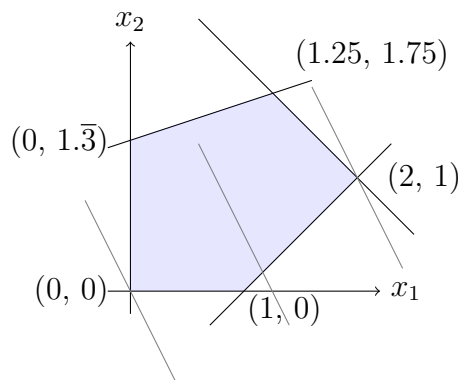

Figure 2.1: Illustration of the feasible region and level sets of the objective function.

For every linear program there exists an associated LP problem called the *dual* problem. The dual of the dual is equal to the original problem,

---

[1]Also referred to as feasible set, search space or solution space.

which we refer to as the *primal* problem. Considering the primal problem as denoted in (2.1), its dual is

$$\max b^T y, \quad \text{subject to } A^T y \geq c, \ y \geq 0.$$

These two problems are related on a number of levels. Note the symmetry between the primal problem and its dual. Also, every feasible solution for a primal problem gives a bound on the optimal objective function value for its dual problem, and vice versa [3, 4, 6].

An optimization problem with a quadratic objective function and linear constraints is called a quadratic program. Since quadratic programs also have linear constraints, everything we have discussed regarding the feasible region of linear programs still holds for quadratic programs. We formulate a quadratic program as:

$$\min \frac{1}{2} x^T G x + c^T x, \quad \text{subject to } Ax \leq b, \ x \geq 0.$$

where G is a symmetrix $n \times n$ matrix, A is a $m \times n$ matrix and c, x and b are vectors in $\mathbb{R}^n$. If the matrix G is positive semidefinite, then the objective function is convex, and the QP problem is considered convex. Convex QP problems are in general easier to solve than nonconvex QP problems because nonconvex QP problems can have suboptimal local minima, whereas convex QP problems does not. In a convex QP problem, all possible local minima are global minima. In this thesis, we only consider convex QP problems.

We consider the linear program from before, in which we replace the linear objective function with a quadratic:

$$(x_1 - 1)^2 + (x_2 - 1)^2.$$

Figure 2.2 depicts the feasible region along with our new quadratic function. The function's minimum value is obviously found at $(1, 1)$, which is depicted with a small dot in the figure, along with the contours of the function.

2.1.1 Solution Methods

Note that in the LP problem illustrated in Figure 2.1, the optimal solution is found at an extreme point of the feasible region. It turns out that in any linear program, there exists an extreme point of the feasible region which attains the optimal objective value. It follows from the maximum principle that the maximum of a convex function on a compact convex set is attained on the boundary [7]. Due to this fact, there are a number of algorithms for solving linear programs that only considers the extreme points of the feasible
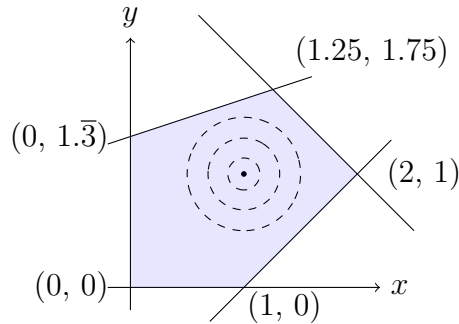
Figure 2.2: Illustration of the feasible region and the contours of the quadratic objective function.

region. They normally move along the boundary of the feasible region from one extreme point to another. Such algorithms are sometimes referred to as *basis-exchange* algorithms, among which the well-known *simplex method* categorize as. In contrast to basis-exchange algorithms, there are *interior point* methods that move in the interior of the feasible region.

In the simplex method, most steps decrease (or increase if we are maximizing) the objective value. While in practice, the simplex algorithm is quite efficient, and is in fact the most widely used of all optimization tools today [4], there are linear programs such as the Klee-Minty cube [8] where the simplex method requires $2^n - 1$ iterations to solve (here $n = m$).

As the simplex method's worst case complexity is proven to be exponential in the size of the problem, the search for algorithms with better theoretical properties arose. Interior point methods have a polynomial worst case time complexity in the size of the problem. This is much more efficient in theory, but in practice, the simplex method tends to be faster than interior point methods on most real-life instances of linear programming. While interior point methods tend to do few, but expensive iterations, the simplex method tends to do many inexpensive iterations.

We mentioned earlier that some QP problems are more difficult to solve than others. For convex QP problems, some methods can solve the problem in polynomial time, such as the *ellipsoid method* [9]. However, solving nonconvex QP problems is shown to be NP-hard [10].

Chapter 3

A QP Formulation

A general formulation of non-linear optimization problems is

$$\min_{x \in \mathbb{R}^n} f(x) \quad \text{subject to} \begin{cases} c_i(x) = 0, & i \in \mathcal{E}, \\ c_i(x) \geq 0, & i \in \mathcal{I}, \end{cases} \tag{3.1}$$

where f and the functions c_i are all smooth, real-valued functions on a subset of $\mathbb{R}^n$, and $\mathcal{I}$ and $\mathcal{E}$ are two finite sets of indices [4].

The problem discussed in this thesis has (from (1.2)) a convex, quadratic, separable objective function

$$f(x) = x^T H x + b^T x$$

and linear constraints. It qualifies as a convex QP problem and is formulated in (1.3) as

$$\min_{x \in \mathbb{R}^n} f(x) \quad \text{subject to } Ax = 0, \; l \leq x \leq u$$

where H is a positive semidefinite diagonal $n \times n$ matrix, A is a $m \times n$ oriented incidence matrix, and b, l, u and x are vectors in $\mathbb{R}^n$.

The function f is called the objective function of the problem. In (1.2) and (1.3), f consists of a quadratic term $(x^T H x)$ and a linear term $(b^T x)$.

In practical applications, a large proportion of the diagonal elements of H and b are zero. For the non-zero elements of H, we typically have $10^{-5} \leq h_i \leq 10^{-1}$. Here h_i denotes the ith diagonal element of H. For the non-zero elements b_i of vector b, we typically have $10 \leq |b_i| \leq 70$. All elements of l and u are non-positive and non-negative, respectively. However, the methods developed in this thesis do not require these values. These values come part from conversations with Goodtech, and part from a couple of real instances that they have supplied. The next section describes the supplied instances in detail.

3.1 Real Instances

There are three instances, and they all:

1. Have very large coefficients in the linear term compared to the quadratic term.

2. Have that all cross-product coefficients are zero in the quadratic term, i.e. in $x^T H x$, H is diagonal.

The instances are called—based on size—*small*, *large* and *vlarge*.

Figure 3.1 shows the value distribution of non-zero elements in H in the three instances. Note that among all the diagonal elements of H, more than 50 percent of them are zero, and among the non-zero elements, most of them are less than 10^{-2}. Note that no non-zero elements in the linear term in any
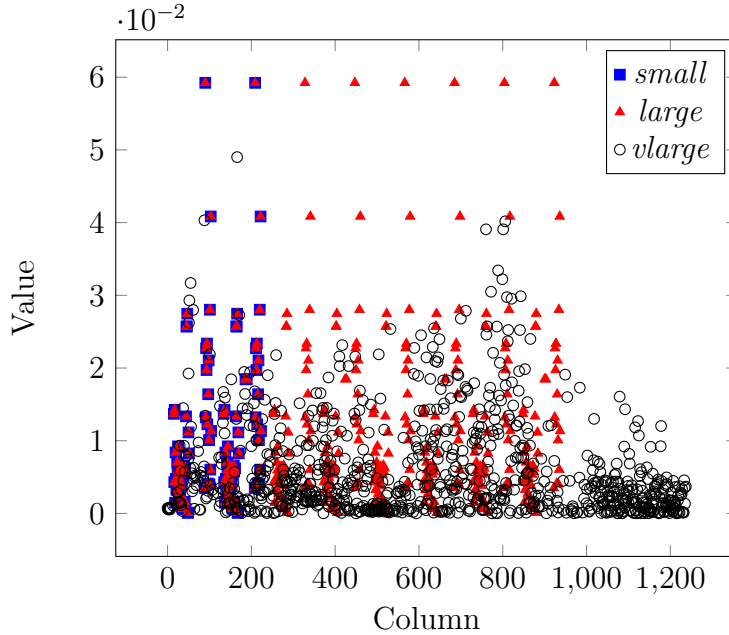

Figure 3.1: Value distribution of non-zero elements in H by column.

of the instances has absolute value less than 20. We see that the values in the linear term are generally of much higher magnitude than the values in the quadratic term.

Let us take a look at some hard statistics of each instance. Table 3.1 shows the problem size of each instance. Table 3.2 shows some statistics about the non-zero elements in the objective function for $i = 1, 2, \ldots, n$

14

Table 3.1: Problem size of each instance

Problem size	_small_	_large_	_vlarge_
Rows	82	328	1127
Columns	238	952	3437
Non-zeroes A	348	1392	4840
Non-zeroes H	108	432	894

Table 3.2: Statistics on non-zero values in the objective function of each instance.

	small and _large_	_vlarge_
$\max(h_{ii})$	2.9614×10^{-2}	4.9011×10^{-2}
$\min(h_{ii})$	4.9290×10^{-5}	1.1026×10^{-5}
$\mathrm{mean}(h_{ii})$	5.2864×10^{-3}	5.8984×10^{-3}
$\max(b_i)$	20	20
$\min(b_i)$	-70	-50

where n is the number of columns. The rows and columns in each problem represent vertices and edges respectively. Avid readers might notice that _small_ and _large_ are very much alike. This is entirely justified as _large_ is four _small_-networks connected, forming one large network.

For each of these instances, Goodtech wants to solve several versions with slight variations in the input. Goodtech wants to simulate that all combinations of edges in the network fail, by forcing their corresponding variables to zero. If one or more variables in an instance is forced to zero, it forms a new instance that we refer to as a _subinstance_ of the original instance. This means that for each of the three instances described in this section, they ideally want to solve a total of $2^n - 1$ subinstances. While this is unrealistic for large n, it is also unrealistic that all edges in the network breaks down. A more realistic approach is to try to solve all subinstances with no more than β breakdowns. That is, for some given β and a problem size n, we solve

$$\sigma(\beta, n) = \sum_{j=0}^{\beta} \binom{n}{j} \qquad (3.2)$$

subinstances with less than or equal to β breakdowns.

3.2 Subinstances

For one or more given variables, their lower and upper bounds are set to zero, i.e. forcing the variables to zero. Let us denote the set of variables that are to be forced to zero by $\mathcal{M}_k$ for some $0 \leq k \leq 2^n - 1$. Forcing these variables to zero in some instance $\mathcal{Q}$ forms a subinstance $\mathcal{Q}_k$. We refer to $\mathcal{M}_k$ as a *modifier* of $\mathcal{Q}$. To simplify notation from here on, we let $\mathcal{Q}_0 = \mathcal{Q}$ and loosen the terminology of *subinstance* so that $\mathcal{Q}_0$ can be called a subinstance of $\mathcal{Q}$ where its modifier $\mathcal{M}_0 = \emptyset$.

To distinguish between optimal solutions over several subinstances, we denote an optimal solution of $\mathcal{Q}_k$ as x_k^*. In any solution of any instance there might be variables that are zero. We denote the set of variables that are zero in an *optimal* solution x_k^* by $\mathcal{Z}_k$. If a modifier $\mathcal{M}_k$ only has variables that are zero in x_0^*, i.e. if $\mathcal{M}_k \subseteq \mathcal{Z}_0$, then $x_k^* = x_0^*$. However, if $\mathcal{M}_k$ has some variables that are non-zero in x_0^*, then $x_k^* \neq x_0^*$.

Generally, for each $\mathcal{Q}_k$ and its modifier $\mathcal{M}_k$ we have that $x_l^* = x_k^*$ for each $l = 0, 1, \ldots, 2^n - 1$ such that $\mathcal{M}_l = S \cup \mathcal{M}_k$ for some $S \subseteq \mathcal{Z}_k$. That is, instances $\mathcal{Q}_k$ and $\mathcal{Q}_l$ have identical optimal solutions if $\mathcal{M}_l$ is obtained by extending $\mathcal{M}_k$ only with variables that are zero in the optimal solution of $\mathcal{Q}_k$. Additionally, for each $\mathcal{Q}_k$ and its modifier $\mathcal{M}_k$ we have that $x_l^* \neq x_k^*$ for all $l = 0, 1, \ldots, 2^n - 1$ if $\mathcal{M}_l = S \cup \mathcal{M}_k$ for some $S \nsubseteq \mathcal{Z}_k$. That is, subinstances $\mathcal{Q}_k$ and $\mathcal{Q}_l$ have different optimal solutions if $\mathcal{M}_l$ is obtained by extending $\mathcal{M}_k$ with any variables that are non-zero in the optimal solution of $\mathcal{Q}_k$.

Chapter 4

An Iterative Method

One interesting property shared by several practical applications is that the coefficients in the linear term of the objective function is of a much higher magnitude than those in the quadratic term. From this, one might think that the quadratic term is close to negligible. By removing the quadratic term from the objective function, leaving only the linear term, the QP problem transforms into an LP problem. Linearizing the objective function using a first-order Taylor series expansion at 0 would give us the same result.

Consider a first-order (linear) Taylor series expansion of a general function f at a point a:

$$T_a(x) = f(a) + \nabla f(a)^T (x - a),$$

where $\nabla f(a)$ is the gradient of f evaluated at a [11]. A linear approximation of function (1.2) becomes

$$T_a(x) = a^T H a + b^T a + (2a^T H + b^T)(x - a).$$

There are a number of terms that cancel each other out. Cancelling them, the approximation becomes

$$T_a(x) = -a^T H a + 2a^T H x + b^T x. \tag{4.1}$$

Now, if $a = 0$ we are left with

$$T_0(x) = b^T x.$$

We introduce a new function $g(x) = T_0(x)$, and we formulate a new LP problem that we call $\mathcal{L}$:

$$\min_{x \in \mathbb{R}^n} g(x) \quad \text{subject to } Ax = 0,\ l \leq x \leq u.$$

We let $\hat{x}$ denote some optimal solution of $\mathcal{L}$, and let x^* denote some optimal solution of $\mathcal{Q}$.

By minimizing $\mathcal{L}$, we can get an idea about the quality of $\mathcal{L}$'s solution with respect to $\mathcal{Q}$ by substituting $\hat{x}$ into (1.2) and checking how much the numerical value deviates from the optimal objective value of $\mathcal{Q}$. For any QP problem and its linearized version, we denote this deviation by Δ. More specifically:

$$\Delta = \left| \frac{f(\hat{x}) - f(x^*)}{f(x^*)} \right|$$

Figure 4.1 shows how Δ changes in respect to the sparsity in the objective function, both in the linear and the quadratic terms, on randomly generated networks with 200 edges and 70 nodes. The percentage of zeroes in the diagonal elements of H is represented by x, while the percentage of zeroes in b is represented by y. Each data point represents the average 100Δ over 100 randomly generated networks.
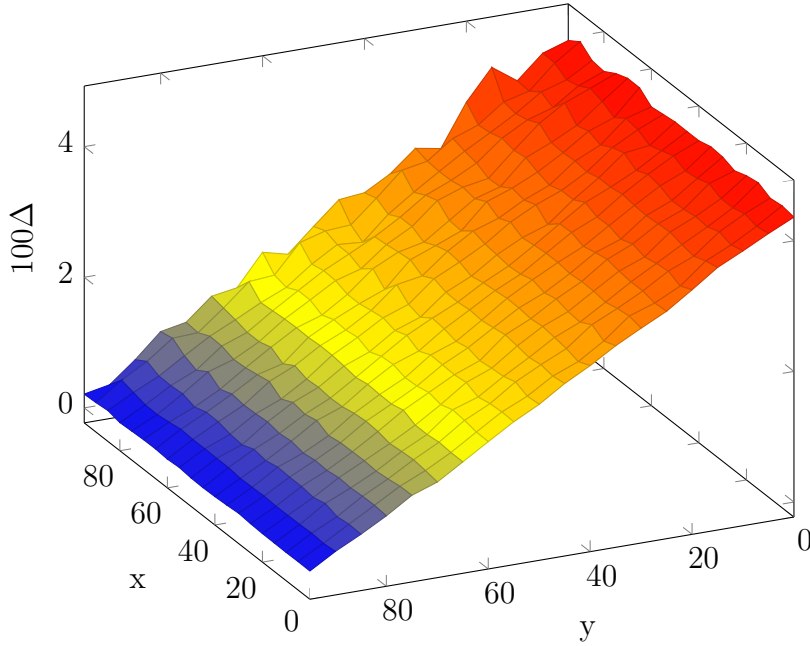

Figure 4.1: Deviation as a function of density in the objective function.

We can see that the density in the diagonal elements of H has close to zero influence on the deviation. While we see that the density of b certainly has more influence, the deviation actually never change more than around 4%.

These results give us incentive to look into solution methods that are based upon linear programming.

4.1 Successive Linear Programming

SLP was first referenced in [12], a paper by Griffith and Stewart in 1961 [13]. They described a procedure they called Mathematical Approximation Programming (MAP) that was in use at Shell Oil Company at the time. The paper is considered a classic in the field of optimization

SLP is a differential technique which utilizes the linear programming algorithm repetitively in such a way that the solution of a linear problem can converge to the solution of the nonlinear problem [12]. The results in the previous section show us that applying an SLP algorithm to a QP problem with the characteristics mentioned in Section 4.2, with an initial guess 0, will give us an approximate solution deviating less than 5% from the optimal objective value, after only one iteration.

4.2 The Method

Consider the quadratic programming problem in (1.2) and (1.3). The constraints are all linear, so by only linearizing the objective function we end up with a linear program. Let T_k denote the Taylor series expansion at point x_k of f. Denote the linear program defined by minimization of T_k subject to these constraints, by $\mathcal{L}_k$. Let $\hat{x}_k$ denote an optimal solution of $\mathcal{L}_k$.

Let $k = 0$ and our initial guess x_0 be given. Applying (4.1) we have that

$$T_k = -x_k^T H x_k + 2x_k^T H x + b^T x.$$

Assume the linear program $\mathcal{L}_k$ has an optimal solution $\hat{x}_k$. We note that $\mathcal{L}_k$ is minimizing in the direction of $x^{\mathcal{U}}$, i.e. the unconstrained minimum of f. Additionally we note that $\hat{x}_k$ lies on the boundary of the feasible region. From this we see that if the unconstrained minimum lies inside the feasible region, i.e. $x^* = x^{\mathcal{U}}$, there exists a point on the line segment between x_k and $\hat{x}_k$ that is closer to x^* than the two end points.

We perform a *line search* by finding the minimum value of f on the straight line between the points x_k and $\hat{x}_k$:

$$\alpha_k \in \arg\min_{\alpha \leq 1} f((1-\alpha)x_k + \alpha\hat{x}_k).$$

Call α_k the *step length*. We calculate x_{k+1} using the step length α_k:

$$x_{k+1} = (1-\alpha_k)x_k + \alpha_k\hat{x}_k.$$

If, however, the unconstrained minimum lies outside the feasible region, i.e. $x^* \neq x^{\mathcal{U}}$, α_k might be greater than 1, and the minimum value of f of all the *feasible* points collinearly with x^* and $\hat{x}_k$ will be $\hat{x}_k$.

We increase k by one and repeat the process from the calculation of T_k until we reach our termination condition. We terminate when the relative difference in objective value of the two last iterations is less than some ϵ. We terminate when

$$\frac{f(x_{k-1}) - f(x_k)}{|f(x_{k-1})|} \leq \epsilon.$$

Algorithm 4.1 describes an algorithm for these steps, given some starting point x_0 and a tolerance ϵ.

Algorithm 4.1: `slp(`x_0`, ` $\epsilon \geq 0$`)`

Set $k \leftarrow 0$
repeat
    $T_k \leftarrow -x_k^T H x_k + 2x_k^T H x + b^T x$
    $\hat{x}_k \leftarrow$ optimal solution of $\mathcal{L}_k$ (Solve)
    $\alpha_k \in \arg\min_{\alpha \leq 1} f((1-\alpha)x_k + \alpha\hat{x}_k)$
    $x_{k+1} \leftarrow (1-\alpha_k)x_k + \alpha_k\hat{x}_k$
    $k \leftarrow k + 1$
until $\dfrac{f(x_{k-1}) - f(x_k)}{|f(x_{k-1})|} \leq \epsilon$

4.3   A Numerical Example

Consider the quadratic program $\mathcal{Q}$:

$$
\begin{array}{llrclcrl}
\text{minimize} & (x-1)^2 & + & (y-1)^2 & - & 2 \\
\text{subject to} & x & + & y & \leq & 3 \\
& x & - & y & \leq & 1 \\
& x & + & 3y & \leq & 4 \\
& & & x, y & \geq & 0,
\end{array}
$$

and let x^* denote one of its optimal points and let f denote its objective function.

We solve $\mathcal{Q}$ using the method described in the previous section. While solving it, we look at a visual representation of the method, along with the linear program in standard form.

Let $x_0 = (0,0)$ such that $T_0 = -2x - 2y$. The linear program $\mathcal{L}_0$ in standard form reads:

$$
\begin{array}{rrrrl}
\text{minimize} & -\ 2x & -\ 2y & & \\
\text{subject to} & x & +\ y & \leq 3 \\
& x & -\ y & \leq 1 \\
& -\ x & +\ 3y & \leq 4 \\
& & x, y & \geq 0.
\end{array}
$$

Figure 4.2 shows a visual representation of $\mathcal{L}_0$ and $\mathcal{Q}$. The figure depicts an initial state of Slp, where you can see the initial guess x_0. It also depicts the unconstrained minimum x^u, which in this case happens to be the same as x^*, the optimal solution to $\mathcal{Q}$. An optimal solution $\hat{x}_0$ to $\mathcal{L}_0$ is also depicted. The striped circles are the contours of f. Note that the feasible region of $\mathcal{L}_k$ and $\mathcal{Q}$ are equal for all k, as the constraints never change during the iterations of the algorithm. Also note that $\mathcal{L}_0$ has multiple optimal solutions, so $\hat{x}_0$ may differ between implementations.
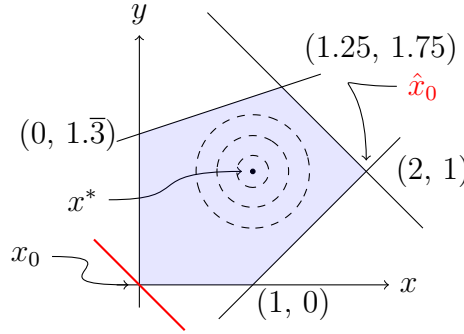

Figure 4.2: A visual representation of $\mathcal{L}_0$ and $\mathcal{Q}$.

We perform a line search between the points x_0 and $\hat{x}_0$. We find a function m of α that range all points collinear with x_0 and $\hat{x}_0$:

$$
f((1 - \alpha)x_0 + \alpha\hat{x}_0) = 5\alpha^2 - 6\alpha = m(\alpha).
$$

To find the minimum of this one-dimensional convex parabola we set $m'(\alpha) = 0$ and solve for α to achieve $\alpha = 0.6$ (see Figure 4.3). Note that $\alpha \in [0, 1]$ because the unconstrained minimum of f is inside the feasible region. Now let

$$
x_1 = 0.4x_0 + 0.6\hat{x}_0 = (1.2, 0.6),
$$

and let us apply (4.1) to calculate a Taylor series expansion at x_1:
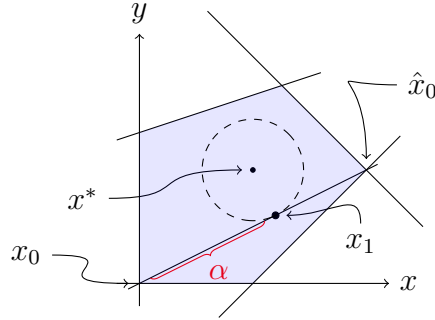
$$
T_1 = 0.4x - 0.8y - 1.8.
$$

21

Figure 4.3: Line search between x_0 and $\hat{x}_0$. The optimal point between x_0 and $\hat{x}_0$ becomes x_1, our starting point for the next iteration.

Now we minimize T_1 subject to the original constraints and call the LP problem for $\mathcal{L}_1$. The linear program of $\mathcal{L}_1$ in standard form reads:

$$
\begin{array}{llrcrcl}
\text{minimize} & & 0.4x & - & 0.8y & & \\
\text{subject to} & & x & + & y & \leq & 3 \\
& & x & - & y & \leq & 1 \\
& - & x & + & 3y & \leq & 4 \\
& & & & x, y & \geq & 0.
\end{array}
$$

We solve $\mathcal{L}_1$ and achieve an optimal solution $\hat{x}_1$ (see Figure 4.4).
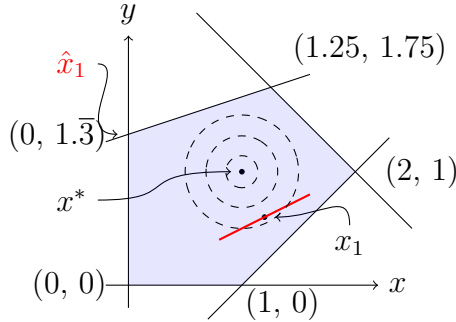

Figure 4.4: A visual representation of $\mathcal{L}_1$ and $\mathcal{Q}$. Note that x_1 now has a red line through it. This is the objective function of $\mathcal{L}_1$.

Again we perform a line search, but this time between the points x_1 and $\hat{x}_1$ and find that $\alpha = 0.27$, $x_2 = (0.88, 0.8)$ and $T_2 = -0.25x - 0.4y - 0.4$. Minimize T_2 subject to the original constraints and call the LP problem for

$\mathcal{L}_2$. The linear program of $\mathcal{L}_2$ in standard form reads:

$$\begin{array}{llrrrl}
\text{minimize} & - & 0.25x & - & 0.4y & \\
\text{subject to} & & x & + & y & \leq 3 \\
& & x & - & y & \leq 1 \\
& - & x & + & 3y & \leq 4 \\
& & & & x, y & \geq 0.
\end{array}$$

We solve $\mathcal{L}_2$ and achieve an optimal solution $\hat{x}_2 = (1.25, 1.75)$ (see Figure 4.5).
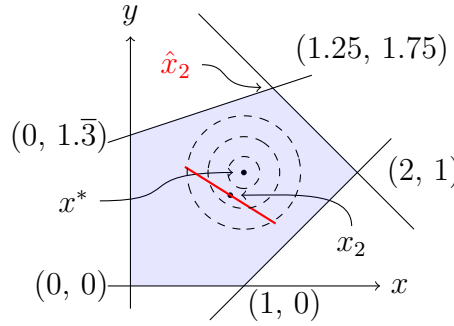

Figure 4.5: A visual representation of $\mathcal{L}_2$ and $\mathcal{Q}$.

Performing line search between x_2 and $\hat{x}_2$ we get that $\alpha = 0.77$ and that $x_3 = (0.96, 1.02)$. Assume $\epsilon = 0.5$, which imples that $\frac{f(x_2) - f(x_3)}{|f(x_2)|} \leq \epsilon$, so we stop and return x_3 as our solution.
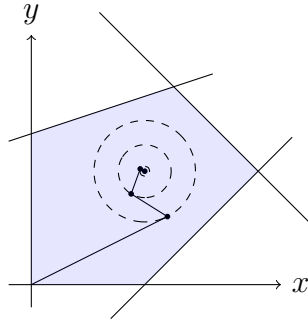

Figure 4.6: Illustration of the path taken by the algorithm.

Figure 4.6 shows the iterates of the algorithm up until the termination condition is met.

4.4 Practical Applications

In practical applications, immediately after a problem is solved, another very similar problem may need to be solved. Often this problem has minor changes in the constraints to the problem before it. If many problems are solved consecutively, it is important with as little downtime as possible between each problem.

We have that

$$x_{\mathcal{B}}^* = B^{-1}b, \tag{4.2}$$

$$z_{\mathcal{N}}^* = (B^{-1}N)^T c_{\mathcal{B}} - c_{\mathcal{N}}, \tag{4.3}$$

$$\zeta^* = c_{\mathcal{B}}^T B^{-1}b, \tag{4.4}$$

where $\mathcal{B}$ denotes the collection of indices corresponding to the basic variables and $\mathcal{N}$ denotes the indices corresponding to the nonbasic variables. Vector x^* is some *primal* optimal solution and z^* is some *dual* optimal solution. The matrix B denotes the matrix corresponding to the basic rows and N denotes the matrix corresponding to the non-basic columns. We let b (not to be confused with b in (1.2)) denote the vector corresponding to the right hand side values of the constraints. The value ζ^* denotes the optimal objective value and c corresponds to the coefficients of the objective function [3].

In (4.3) we see that a change in b does not require us to recompute $z_{\mathcal{N}}^*$. This means that the dictionary is guaranteed to stay dually feasible when b changes. To illustrate this, imagine that the third constraint in $\mathcal{L}_2$ from Section 4.3 changes from $-x + 3y \leq 4$ to $-x + 3y \leq 0$. The new linear program then reads:

$$
\begin{array}{llrcrcl}
\text{minimize} & - & 0.25x & - & 0.4y & & \\
\text{subject to} & & x & + & y & \leq & 3 \\
& & x & - & y & \leq & 1 \\
& - & x & + & 3y & \leq & 0 \\
& & & & x, y & \geq & 0.
\end{array}
$$

Figure 4.7 illustrates the new feasible region. Readers are encouraged to compare Figure 4.5 with Figure 4.7 to note how the feasible region has changed and made the unconstrained minimum infeasible.

Although we do not need to recompute $z_{\mathcal{N}}^*$, we do however, need to recompute $x_{\mathcal{B}}^*$ and ζ^*. From the optimal dictionary of $\mathcal{L}_2$, we have that

$$
B = \begin{bmatrix} 1 & 1 & 0 \\ -1 & 1 & 1 \\ 3 & -1 & 0 \end{bmatrix},
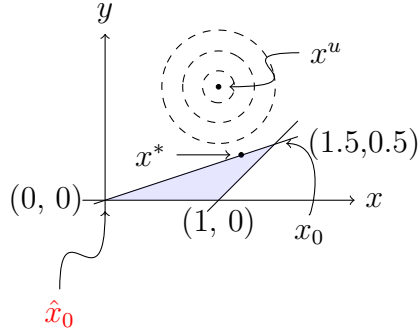$$

Figure 4.7: A visual representation of the new QP problem and its linearized version.

and b has now changed to

$$b = \begin{bmatrix} 3 & 1 & 0 \end{bmatrix}^T.$$

Using (4.2) and (4.4) we can calculate a new basic solution

$$x_{\mathcal{B}}^* = \begin{bmatrix} 0.75 & 2.25 & -0.5 \end{bmatrix},$$

and update our dictionary to become

$$
\begin{array}{rrrrr}
-\zeta & = & 1.01 & - & 0.10w_3 & - & 0.10w_1 \\
\hline
y & = & 0.75 & - & 0.25w_3 & - & 0.25w_1 \\
x & = & 2.25 & + & 0.25w_3 & - & 0.75w_1 \\
w_2 & = & -0.50 & - & 0.50w_3 & + & 0.50w_1
\end{array}
$$

Note that this dictionary is not primally feasible. In the current dictionary, we run the dual simplex method by letting w_1 enter the basis, while letting w_2 leave the basis. This leaves us with the following optimal dictionary:

$$
\begin{array}{rrrrr}
-\zeta & = & 0.58 & - & 0.20w_3 & - & 0.20w_2 \\
\hline
y & = & 0.50 & - & 0.50w_3 & - & 0.50w_2 \\
x & = & 1.50 & - & 0.50w_3 & - & 1.50w_2 \\
w_1 & = & 1.00 & + & 1.00w_3 & + & 2.00w_2
\end{array}
$$

From here we start a new iteration of the algorithm described in Section 4.2 and 4.3 while preserving the old classification of basic and nonbasic variables and using our current primal solution as an initial guess x_0. We get that

$$T_0 = -2.5 + x - y,$$

which becomes our new objective function. Solving that linear program, we end up with an optimal solution $\hat{x}_0 = (0,0)$. Since x_0 and $\hat{x}_0$ are two adjacent vertices of the feasible region, the entire line search takes place on the edge between those points. Since the unconstrained minimum x^u is infeasible, the solution x^* must lie on the boundary of the feasible region. It follows from Figure 4.7 that x^* lies on the edge joining x_0 and $\hat{x}_0$. This means that a line search will find x^* and the algorithm will terminate. We perform the line search and find that $\alpha = 0.2$ and that

$$x_1 = 0.8x_0 + 0.2\hat{x}_0 = (1.2, 0.4) = x^*.$$

Chapter 5

A Tree Representation

In this chapter, we discuss how to represent all subinstances of some arbitrary instance $\mathcal{Q}$. We also discuss how to identify subinstances to which we do not already know the solution. We construct a *tree* of all subinstances with distinct optimal solutions. This tree enables us to efficiently check whether some modifier forms a subinstance with an unknown solution.

If we were to try to solve all 2^n possible subinstances of an instance, we would most likely end up solving a huge amount of subinstances with the same optimal solutions. This is evident in some of the real instances; when we solve the unmodified instance `vlarge`, we already have the solution to 2^{1749} of its possible subinstances, i.e. $|\mathcal{Z}_0| = 1749$. Just trying to unnecessarily solve all these subinstances—let alone store the solutions—would be a huge waste of resources. Instead, we try to identify subinstances with distinct optimal solutions.

Consider a set $\mathbb{U} \subseteq \{0, 1, \ldots, \sigma(\beta, n) - 1\}$ of indices such that if $i, j \in \mathbb{U}$, and $i \neq j$, then $x_i^* \neq x_j^*$. With such a set; given an arbitrary modifier $\mathcal{M}_k$, we can check if there exists an index $l \in \mathbb{U}$ such that $x_k^* = x_l^*$. By checking each index $l \in \mathbb{U}$, we only need to confirm that $\mathcal{M}_k \subseteq \mathcal{Z}_l$ to know that $x_k^* = x_l^*$. However, checking *all* members of $\mathbb{U}$ might not be practical when the set grows large. With a tree representation of $\mathbb{U}$, we can (dis)confirm that $x_k^* = x_l^*$ for a modifier $\mathcal{M}_k$ without checking all indices in the set.

We define a tree as an ordered pair $G = (V, E)$ comprising a set V of vertices together with a set E of edges. Let each vertex $v_k \in V$ represent an index $k \in \mathbb{U}$. For each vertex v_k and its parent v_l, $\mathcal{M}_l$ must be a subset of $\mathcal{M}_k$. Put differently, each vertex and its corresponding modifier must be an extension of the modifier of its parent. The root v_0, which does not have a parent, has a modifier that is equal to the empty set, i.e. $\mathcal{M}_0 = \emptyset$.

Algorithm 5.1 describes an algorithm for finding the index $i \in \mathbb{U}$ for a given modifier $\mathcal{M}_l$ such that $x_l^* = x_i^*$. Note that the return value on the last

line of the algorithm is implementation specific, but it should be a value that somehow tells us that we have not previously found the solution to $\mathcal{Q}_l$.

Algorithm 5.1: $\mathtt{find}(\mathcal{M}_l,\ v_k)$

---

if $\mathcal{M}_l \subseteq \mathcal{Z}_k$ **then**
  $\llcorner$ **return** k
foreach *child vertex v_i of v_k* **do**
  if $\mathcal{M}_i \subseteq \mathcal{M}_l$ **then**
    $j \leftarrow \mathtt{find}(\mathcal{M}_l,\ v_i)$
    if *j is not -1* **then**
      $\llcorner$ **return** j

return -1

Consider a tree G with a vertex set $V = \{0, 2, 3, 7, 8, 10, 12, 15, 16, 28, 29\}$ and an edge set $E = \{\{0, 2\}, \{0, 8\}, \{0, 16\}, \{2, 3\}, \{8, 10\}, \{8, 12\}, \{16, 28\}, \{3, 7\}, \{12, 15\}, \{28, 29\}\}$ and the following sets:

$$\mathcal{M}_0 = \{\} \qquad \mathcal{Z}_0 = \{1, 3\}$$
$$\mathcal{M}_2 = \{2\} \qquad \mathcal{Z}_2 = \{2, 3, 5\} \qquad \mathcal{M}_{12} = \{3, 4\} \qquad \mathcal{Z}_{12} = \{4, 3, 1\}$$
$$\mathcal{M}_3 = \{1, 2\} \qquad \mathcal{Z}_3 = \{1, 2, 4, 5\} \qquad \mathcal{M}_{15} = \{1, 2, 3, 4\} \quad \mathcal{Z}_{15} = \{1, 2, 3, 4\}$$
$$\mathcal{M}_7 = \{1, 2, 3\} \quad \mathcal{Z}_7 = \{1, 2, 3, 5\} \qquad \mathcal{M}_{16} = \{5\} \qquad \mathcal{Z}_{16} = \{1, 3, 5\}$$
$$\mathcal{M}_8 = \{4\} \qquad \mathcal{Z}_8 = \{1, 4, 5\} \qquad \mathcal{M}_{28} = \{3, 4, 5\} \quad \mathcal{Z}_{28} = \{2, 3, 4, 5\}$$
$$\mathcal{M}_{10} = \{2, 4\} \quad \mathcal{Z}_{10} = \{2, 3, 4, 5\} \qquad \mathcal{M}_{29} = \{1, 3, 4, 5\} \quad \mathcal{Z}_{29} = \{1, 2, 3, 4, 5\}$$

The tree G is a tree representation of a set $\mathbb{U}$ of some $\mathcal{Q}$ with 5 variables. Figure 5.1 shows how Algorithm 5.1 would work with input $(\mathcal{M}_{14} = \{2, 3, 4\}, v_0)$. The left side of the figure shows the tree G.

5.1 Representing Sets

Sets of variables, such as the sets $\mathcal{M}_k$ and $\mathcal{Z}_k$ can be stored very compactly on a computer. By using one bit per possible variable, with a total of n possible variables, each set will require exactly n bits of storage. We store these n bits in a bit vector[1] of length n. Each bit represents a variable. More specifically, the rightmost bit represents x_1, the second rightmost bit represents x_2, and so on until the leftmost bit that represents x_n. When a bit is set to 1, it means that its corresponding variable is in the set, while a 0 means that it is *not* in the set.

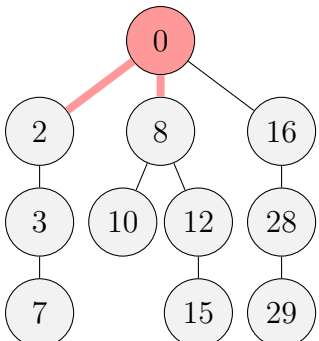
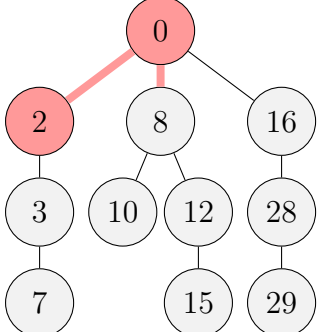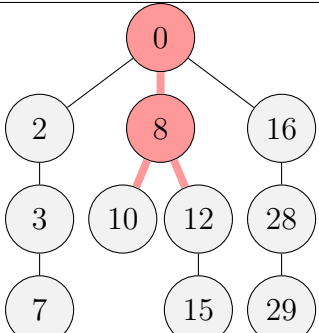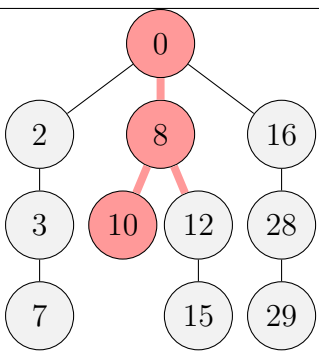[1]Also known as bit array, bitmap, bitset or bit string

Finding $\mathcal{M}_{14} = \{2, 3, 4\}$ **from** v_0

k	$\mathcal{M}_k$	$\mathcal{Z}_k$
0	$\{\}$	$\{1, 3\}$
2	$\{2\}$	$\{2, 3, 5\}$
8	$\{4\}$	$\{1, 4, 5\}$
16	$\{5\}$	$\{1, 3, 5\}$

We see that $\mathcal{M}_{16} \not\subseteq \mathcal{M}_{14}$, so we ignore the subtree of v_{16} completely. However, $\mathcal{M}_2 \cup \mathcal{M}_8 \subseteq \mathcal{M}_{14}$, so we continue searching from the vertices v_2 and v_8.

Finding $\mathcal{M}_{14} = \{2, 3, 4\}$ **from** v_8

k	$\mathcal{M}_k$	$\mathcal{Z}_k$
3	$\{2, 1\}$	$\{2, 1, 4, 5\}$

Since $\mathcal{M}_3 \not\subseteq \mathcal{M}_{14}$, we stop the search and conclude that the subtree of v_2 does not contain the solution to $\mathcal{Q}_{14}$.

Finding $\mathcal{M}_{14} = \{2, 3, 4\}$ **from** v_2.

k	$\mathcal{M}_k$	$\mathcal{Z}_k$
10	$\{2, 4\}$	$\{2, 3, 4, 5\}$
12	$\{3, 4\}$	$\{4, 3, 1\}$

We see that $\mathcal{M}_{10} \subseteq \mathcal{M}_{14}$ and that $\mathcal{M}_{12} \subseteq \mathcal{M}_{14}$, so we continue searching from the vertices v_{10} and v_{12}.

Finding $\mathcal{M}_{14} = \{2, 3, 4\}$ **from** v_4.

k	$\mathcal{M}_k$	$\mathcal{Z}_k$
10	$\{2, 4\}$	$\{2, 3, 4, 5\}$

Since $\mathcal{M}_{10} \subseteq \mathcal{M}_{14} \subseteq \mathcal{Z}_{10}$ we conclude that $x_{14}^* = x_{10}^*$.

Figure 5.1: `find` being executed on G where $\mathcal{M}_{14} = \{2, 3, 4\}$.

By representing sets this way, we can achieve a one-to-one correspondence between a modifier $\mathcal{M}_i$ and its index i by evaluating the bit vector of $\mathcal{M}_i$. For instance the modifier $\{1, 4, 5\}$ can be represented by the bit vector `11001` that evaluates to 25, becoming its index. Another benefit of representing sets as bit vectors is that we can check whether a set is a subset of another set by performing bitwise operations. More specifically, simple `AND` and `NOT` operations will suffice.

Consider two bit vectors A^b and B^b representing two sets A and B, respectively. By performing a bitwise `NOT` operation and a bitwise `AND` operation, A is a subset of B if and only if A^b `AND` `NOT` $B^b = 0$. Similarly, A is a subset of B if and only if $A \backslash B = \emptyset$.

Take for example the three bit vectors, $D^b = 0110$, $E^b = 1110$ and $F^b = 0011$ representing the sets D, E, F, respectively. If we want to check whether D and F are subsets of E, we check if the answer to the following operations are equal to zero:

$$
\begin{array}{rll}
 & 0110 & D^b \\
\text{AND} & 0001 & \text{NOT } E^b \\
\hline
= & 0000 &
\end{array}
\qquad
\begin{array}{rll}
 & 0011 & F^b \\
\text{AND} & 0001 & \text{NOT } E^b \\
\hline
= & 0001 &
\end{array}
$$

From this we can conclude that $D \subseteq E$ and $F \nsubseteq E$.

We can also store sets in a sparse format by only storing the indices of the variables that are actually in the set. However, this format normally requires at least 16 bits[2] per variable in the set. We further discuss the trade-offs of the different methods of storage in Chapter 6.

5.2   Constructing a Tree

Before we can construct a tree similar to the one in Figure 5.1, we need a slighty modified `find` that we call `mfind`. `mfind` is modified to take an output argument that holds a vertex, while the function returns a boolean. The returned boolean tells us whether the output vertex is the solution or not. If the returned boolean is true, then the output argument is the solution vertex, i.e. the vertex of the index that would be returned by `find`. If the boolean is false, then the output argument is the vertex that would be the parent vertex of the modifier we searched for if we were to insert it in the tree. While this can sound confusing, Algorithm 5.2 shows in detail how `mfind` works.

---

[2]By storing each variable index as a `short` integer type. The amount of bits needed is platform dependent and implementation specific, but in `C` it is never less than 16 bits.

Algorithm 5.2: `mfind(`$\mathcal{M}_l$`,` v_k`,` v_*`)`

---

if $\mathcal{M}_l \subseteq \mathcal{Z}_k$ **then**
    └ **return** *true*
foreach *child vertex* v_i *of* v_k **do**
    if $\mathcal{M}_i \subseteq \mathcal{M}_l$ **then**
        $v_* \leftarrow v_i$
        $P \leftarrow$ `mfind(`$\mathcal{M}_l$`,` v_i`,` v_*`)`
        if P **then**
          └ **return** *true*

return *false*

To construct a tree such as the one seen in Figure 5.1, we start with some instance $\mathcal{Q}$, a modifier $\mathcal{M}_0 = \{\}$ and a root v_0. Then, for each possible subinstance $\mathcal{Q}_i$ of $\mathcal{Q}$ such that $|\mathcal{M}_i| \leq \beta$, we run `mfind` and add it to the tree if its solution is distinct, i.e. the returned boolean is false.

Algorithm 5.3: `construct(`$\mathcal{Q}$`,` β`)`

$V \leftarrow \{v_0\}$
$E \leftarrow \emptyset$
foreach *modifier* $\mathcal{M}_i \in \mathcal{P}(\{1, 2, \ldots, n\}), |\mathcal{M}_i| \leq \beta$ **do**
    $P \leftarrow$ `mfind(`$\mathcal{M}_i$`,` v_0`,` v_k`)`
    if $\neg P$ **then**
        $V \leftarrow V \cup \{v_i\}$
        $E \leftarrow E \cup \{(k, i)\}$
$G \leftarrow (V, E)$
return G

Algorithm 5.3 describes an algorithm for constructing a tree G, given some instance $\mathcal{Q}$ and a maximum number of breakdowns β. Here, $\mathcal{P}(S)$ denotes the power set of S. Readers are encouraged to try to construct the tree in Figure 5.1 on their own, following Algorithm 5.3. Note that the tree might change if the order of insertion changes. The combinations of modifiers under the construction of the tree in Figure 5.1 were inserted in lexicographical order.

Chapter 6

Implementation

Recall from Section 3.1 that we discussed limiting the amount of subinstances to solve. We did this by introducing a limit on the number of simultaneous breakdowns in the network by some β. Another approach is to implement a discrete-event simulator (DES). Each event occurs at a particular instant in time that changes the state of the system. In this case, each event would either be a link breaking down, or that a link is repaired.

The difference in these two approaches are prominent, but they share the same core, namely a QP solver. In this chapter we first discuss the implementation of the solver declared in Chapter 4, before we move on to the implementation of the two approaches for solving several subinstances.

6.1   Finding an LP Solver

The method described in Chapter 4 relies on repetitively solving linear programs. Before implementing such a method, it is important to choose an appropriate LP solver. The solver should a) be released under a free license; and b) allow library calls in C/C++

Among a list of about 50 solvers (most of them proprietary), the following three matches the aforementioned criteria:

Clp

   Clp is short for Coin-or linear programming. It is a free linear programming solver released under the Common Public License (CPL). It is primarily meant to be used as a callable library. Its license allows other software to link to the Clp library without requiring that that software is released under the same license (permissive). [14]

GLPK

32

GLPK is short for GNU Linear Programming Kit. It is a free linear (integer) programming software packaged released under the GNU General Public License (GNU GPL). GLPK is organized in the form of a callable library. Its license requires linking software to be released under the GNU GPL (reciprocal). [15]

lp_solve

lp_solve is a free linear (integer) programming solver released under the GNU Lesser General Public License (GNU LGPL). Its license is permissive like the CPL. [16]

Incidentally, these three solvers are the only three open source LP solvers suggested by the NEOS Optimization Guide [17]. In addition to the mentioned criteria, it is important that the solver is fast. Moreover, it needs to be fast on problems that fit the description in Section 4.2.

Testing the three solvers on linearized versions of the instances *small* and *large*—presented in Section 3.1—reveals the running times shown in Table 6.1. The running times are the number of seconds in CPU-time after 1000 runs.

Table 6.1: Running time in CPU-seconds used by each solver to solve each instance 1000 times.

Data Set	Clp	GLPK	lp_solve
small	6.929	26.096	4.734
large	12.832	47.977	23.376

While lp_solve is the fastest solver on a smaller LP problem, it doesn't scale as well as with larger problems as Clp does.

GoodTech need to solve problems with more than 200 variables, and *small* is almost hitting that lower limit, so a good result on *large* is prioritized over the other. This means that Clp outperforms the other two solvers in this test.

Although this is not an extensive test, it is a pretty good indicator that Clp is the fastest solver on similar problems.

6.2 Clp

Clp is written primarily by John J. Forrest, now retired from IBM Research. At the time of writing, Clp is under active development. It is currently managed by John Forrest, Julian Hall, Lou Hafer and Matthew Saltzman. [18]

Matrices in Clp are stored in a compact format using three vectors. The first vector contains all the non-zero elements of the matrix. The second vector contains the indices of the elements in the first vector. The third vector contains an accumulated number of elements in each row/column. The order of the elements depends on whether the matrix is row-ordered or column-ordered. The indices represent the column/row position of the elements, and the accumulated values represent the accumulated number of elements in the row/columns depending on whether the matrix is row-ordered or column-ordered, respectively. To get a better understanding of how they are stored, consider the matrix

$$\begin{bmatrix} 3 & 1 & 0 & -2 & -1 & 0 & 0 & -1 \\ 0 & 2 & 1.1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2.8 & 0 & 0 & -1.2 & 0 \\ 5.6 & 0 & 0 & 0 & 1 & 0 & 0 & 1.9 \end{bmatrix}$$

being stored in row-ordered format, then the element vector would be

$$\begin{bmatrix} 3 & 1 & -2 & -1 & -1 & 2 & 1.11 & 1 & 2.8 & -1.2 & 5.6 & 1 & 1.9 \end{bmatrix}^T,$$

the vector containing the column indices would be

$$\begin{bmatrix} 0 & 1 & 3 & 4 & 7 & 1 & 2 & 2 & 5 & 3 & 6 & 0 & 4 & 7 \end{bmatrix}^T$$

and the vector containing the vector starts (the accumulated indices) would be

$$\begin{bmatrix} 0 & 5 & 7 & 9 & 11 & 14 \end{bmatrix}^T.$$

The `CoinPackedMatrix` class represents such a compact matrix. To keep overhead at a minimum, it is important to implement the new solver (from here on called Slp) using the same compact format.

Clp has a base class that holds data for both linear and quadratic models called `ClpModel`. The class itself knows nothing about the implementation of the algorithms, but it contains all the data about a problem needed to apply an algorithm on that data. This is very handy for passing data throughout Slp without much overhead. It also results in very neat function prototypes as only one parameter is needed for passing lots of information. For instance, the function prototype in Slp for the function that performs the line search described in Chapter 4 looks like this:

```
double lineSearch(const double* p, const double* q,
                  const ClpModel& m);
```
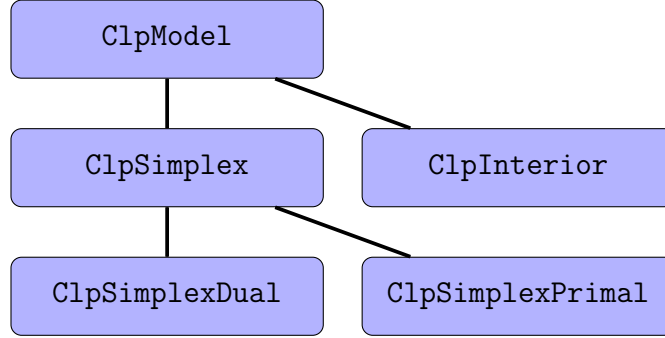
Figure 6.1: Class hierarchy for `ClpModel`

Figure 6.1 shows the class hierarchy for `ClpModel`. The subclasses contain the implementation of the algorithms that their class names describe. For instance, `ClpSimplexDual` contains the implementation of the dual simplex algorithm.

6.3 Slp

While implementing an algorithm, it makes sense to start to implement subfunctions that are independent of other functions.

Implementing the Taylor series expansion is pretty straight forward, as we have a closed-form definition for the specific objective function that we described in Chapter 3. The function prototype for the Taylor series expansion reads:

```
void taylor(double* T, const double* x, const ClpModel& m);
```

where `ClpModel` gives us access to both the linear and quadratic objective term. The result of the Taylor-series expansion, i.e. the coefficients of the new objective function, is put in `T`.

The next step in the Slp algorithm is to solve the new linear program with the new objective function `T`. This is simply done with a call to Clp.

After solving the linear program, the next step is to find the one-dimensional minimizer between our current point and the solution to the linear program we just solved. The step length α_k tells us where this minimizer lies between the two points. To implement this line search, we find a closed-form definition of the step length α_k by letting $m(\alpha) = f((1 - \alpha)x_k + \alpha\hat{x}_k)$ and setting $m'(\alpha) = 0$ and solving for α:

$$\alpha_k = \frac{2x_k^T H x_k - 2\hat{x}_k^T H x_k + b^T x_k - b^T \hat{x}_k}{2\hat{x}_k^T H \hat{x}_k - 4\hat{x}_k^T H x_k + 2x_k^T H x_k}$$

The function prototype for the line search function reads:

```
double lineSearch(const double* p, const double* q,
                  const ClpModel& m);
```

where `ClpModel` gives us access to both the linear and quadratic objective term.

The termination condition of Slp depends on the objective value of the two previous iterations. So, in order to terminate, we need a function that can compute the objective value. The function prototype for this function reads:

```
double objVal(const double* p, const ClpModel& m);
```

where `ClpModel` again gives us access to both the linear and quadratic objective term.

These three functions all have a linear running time in the order of the number of columns in the problem. The running times are dependent on the b vector, which is not stored sparsely.

When it comes to decisions to be made around data types and storage formats, the choices are quite limited. It is pretty much dependant on the linear programming solver that is used. As mentioned earlier, Clp stores matrices in a sparse format, and all the sub functions only operate on non-zero elements in those matrices, so there is not really room for much overhead.

An implementation of Algorithm 4.1 in `C++` looks like this:

```
ClpModel lin = m; // A linear copy of m
do {
  taylor(T, x, m); // Set new linear objective
  lin.chgObjCoefficients(T); // Run the primal simplex method
  lin.primal();

  const double* xhat =  lin.primalColumnSolution();
  double alpha = lineSearch(x, xhat, m);
  if (alpha > 1) alpha = 1;

  double oldval = objVal(x, m);
  for (int i = 0; i < numCols; i++) {
      x[i] = (1-alpha) * xhat[i];
  }
  double val = objVal(x, m);

  stop = (oldval - val);
```

```
    stop /= fabs(oldval);
} while (stop > tolerance);
```

6.4 Solving Subinstances

In this section, we discuss the implementation of the different approaches to solving several subinstances. As mentioned in the beginning of the chapter, the two main approaches are to solve a limited number of subinstances, bounded by $\sigma(\beta, n)$ for some β and a problem size n, or to implement a discrete event simulator that only stores the current state of the network. First we will discuss the implementation of the tree structure, before we move on to the two main approaches.

6.4.1 Tree Structure

All trees consists of vertices, and in our specific case, each vertex contains two sets $\mathcal{M}$ and $\mathcal{Z}$, a solution x^*, and a number of child vertices. To keep vertices as simple as possible, we implement them as simple `structs`. Each vertex `struct` looks like this:

```
struct vertex {
  set<uint16_t> m;
  set<uint16_t> z;
  double* sol;
  vector<struct vertex*> children;
};
```

Readers familiar with `C++` might notice that we use `set` instead of using a bit vector as we discussed in Section 5.1, and we will come to that later. Note that the sets consist of indices of primitive type `uint16_t`, which is short for `unsigned short int`. That means that the sets are limited to $2^{16} = 65536$ elements, and therefore also putting a limit to the size of the QP problems that can be solved. This seems like a reasonable limit as Goodtech wants to solve problems with $200 - 2000$ variables. If they need to solve larger problems in the future, it is possible to change the primitive data type.

   As soon as a vertex is defined, we are ready to implement `mfind`. Remember that `mfind` is a modified version of `find` that tells us which vertex should be the parent of our potentially new vertex in case it has a distinct solution. `mfind` is defined in two parts. The first part is a driver function for starting the algorithm on some vertex. The second part is the actual recursively defined algorithm. These two parts are called `mfind` and `mfindrec`, respectively. We define `mfind` as follows:

```
bool mfind(const set<uint16_t>& m,
struct vertex* v, struct vertex*& ret)
{
  ret = v;
  return mfindrec(m, v, ret);
}
```

The parameters for `mfind` are a modifier, a vertex where the search will begin and an output vertex. In order to search the whole tree, one would use the root vertex as parameter `v`. The function returns a boolean that changes the meaning of the output vertex. The function makes a call to `mfindrec`:

```
bool mfindrec(const std::set<uint16_t>& m,
struct vertex* v, struct vertex*& ret)
{
  if (isSubset(m, v->z)) return true;
  for (struct vertex* vi : v->children) {
    if (isSubset(vi->m, m)) {
      ret = vi;
      bool temp = mfindrec(m, vi, ret);
      if (temp) return true;
    }
  }
  return false;
}
```

It is quite evident from the code that the performance of the `isSubset` function is important. In Section 5.1 we discussed using bit vectors to represent sets. We concluded that simple `AND` and `NOT` operations would tell us whether a set is a subset of another set. We also discussed the possibility of it being more memory efficient than storing sets in a sparse format where only the indices of each variable is stored.

Using a bit vector, each set consumes exactly n bits of storage if our *universe* holds n variables. Using sets of sparsely stored indices, each set consumes a varying amount of bits depending on how many variables are in the set. If each index is of data type `uint16_t`, then each index consumes 16 bits. The trade-off here is quite clear: If a set contains less than one sixteenth of n, then the sparse format uses less memory. This might actually be the case in a lot of sets, especially when solving all possible subinstances where the number of breakdowns are limited by some β. Consider a very small case where $n = 200$ and we solve all subinstances $\sigma(3, 200) = \sum_{j=0}^{3} \binom{200}{j} = S$. With a total number of S sets stored as bit vectors, the memory use

for storing the modifiers is $200S$ bits ≈ 32 megabytes. Whereas, with S sets of sparsely stored indices, the memory use for storing the modifiers is approximately 8 megabytes. However, as β increases, the greater the benefits of storing sets in bit vectors become evident. As soon as β becomes greater than 12, bit vectors will use less memory to store all modifiers. Figure 6.2 shows the ratio of memory used by bit vectors over memory used by sets of sparsely stored indices where $n = 200$ and $5 \leq \beta \leq 30$. Aside from the
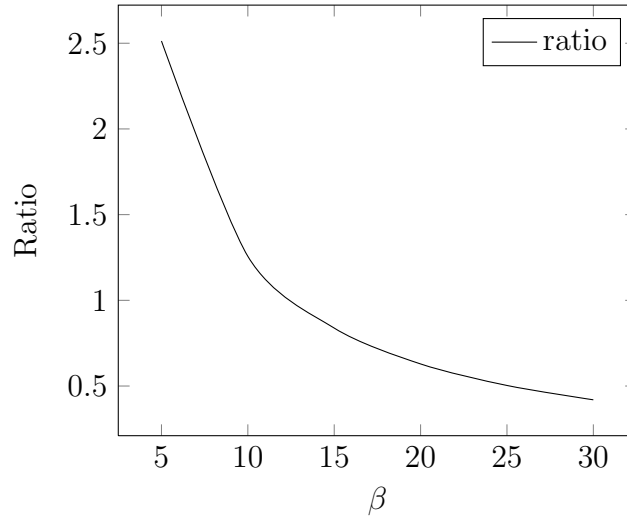

Figure 6.2: The ratio of memory use of `bitset` over `set` as a function of β.

modifier set, we also have to store the set of variables that are equal to zero in the optimal solutions of each subinstance. These sets are more difficult to analyze in terms of memory use, simply because the cardinality is unknown. However, we do know that $|\mathcal{Z}_k| \geq |\mathcal{M}_k|$.

With sets of sparsely stored indices, we only need to check the variables that are actually in the sets in order to determine if a set is a subset of some set. However, with bit vectors, we need to check n bits—where n is the number of variables in the universe—in order to determine the same. This means that the `isSubset` routine will most likely have a different running time with the two implementations.

One way to find out which of the two are most suited for our need is to perform a benchmark of the two. By setting the universe to some size n, and generating sets of random sparsity less than or equal to n, we can test how fast `isSubset` will run. We generate random sets as `bitsets` in C++, storing copies of these sets in the sparse representation, then testing the performance of both implementations of `isSubset`. Figure 6.3 shows the running time of the two implementations in seconds. Both implementations were tested

on 3000^2 `isSubset`-executions, while increasing n. Codes for the benchmark can be found in Appendix B.1.
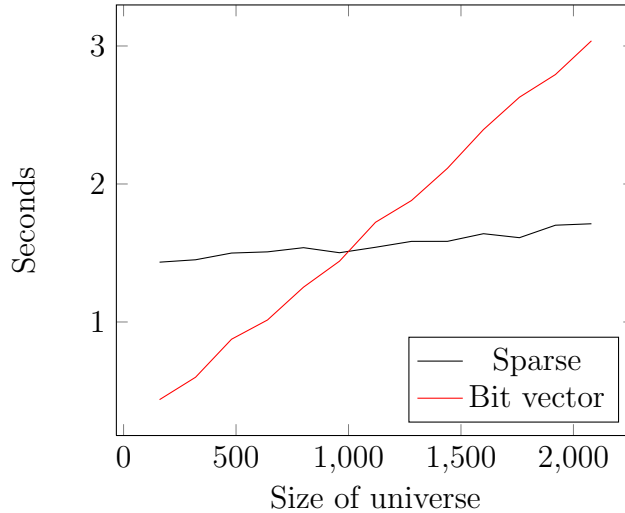

Figure 6.3: CPU-seconds used to execute 3000^2 calls to `isSubset` on randomly generated sets with random sparsity as a function of n.

We see that the bit vector implementation is much faster than the sparse implementation on smaller universes. However, the sparse implementation scales much better than the bit vector implementation when the size of the universe increases. When it comes to memory, the situation is the opposite. The sets of sparsely stored indices obviously uses less memory when β is small, i.e. the modifiers are more sparse. However, they do not scale as nicely as the bit vectors. So the choice relies on what is most important: memory or speed. Since the nature of this thesis is to implement a *fast* solver, we give speed a higher priority than memory efficiency, so we choose to implement the sets of sparsely stored indices. That is, in `C++`, `set`.

6.4.2   Tree Construction

Because we do not need to store any information about edges in the tree, we do not need to implement edges as an object of its own. We defined a vertex `struct` earlier, having a `vector` of child vertices. This means that for each vertex we have, we can reach all child vertices, children's child vertices and so on, from that vertex. So, if we have a pointer to the root of a tree, we can reach all vertices in the tree. Although we earlier defined a tree as an ordered pair (V, E) of a set of vertices V and a set of edges E, we only need a pointer `struct vertex*` to our root to access our tree.

The biggest operations in Algorithm 5.3 is iterating the power set of $\{1, 2, \ldots, n\}$. There are lots of combination generators available, so instead of re-inventing the wheel, we choose a generator that suits our needs. We need one that is easy to use, and is capable of generating combinations of a given cardinality. One implementation, similar in use to `next_permutation` in the standard `C++ STL`, is presented in [19]. Codes from [19] can be found in Appendix B.2.

The remaining parts of the `construct`-routine is just running `mfind` on each generated modifier, and adding it to the tree if necessary. Then at the end, return the pointer to the root vertex.

6.4.3 Discrete Event Simulation

In a discrete event simulation (DES), operations are always triggered by events. In a DES for the reliability analyses discussed in Chapter 1, we only need to consider two types of events; a breakdown in the network, or that a link is repaired. Each link in the network is represented by a variable, so each event can by represented by a single variable.

The current state of our system is easily represented by a quadratic program $\mathcal{Q}$ and a set of variables representing the current broken links in our network. As this set represents broken links, just like our modifiers $\mathcal{M}_k$ represents breakdowns, we choose to denote the current state of the system by $\mathcal{M}$.

Given a variable x to represent a link that breaks down, we need to update the current state of the system. This is done with a simple *union*

$$\mathcal{M} = \mathcal{M} \cup \{x\}.$$

Given a variable x to represent a repaired link, we update the current state with a simple *relative complement*

$$\mathcal{M} = \mathcal{M} \setminus \{x\}.$$

We call these two operations for `break` and `repair`, respectively. After every change of state in the system, we need to solve the subinstance of $\mathcal{Q}$ defined by the modifier $\mathcal{M}$.

As the state of the system changes, we need to solve more and more subinstances. We can choose whether or not we want to store the modifiers and their respective solutions in a tree or not. If we choose to store distinct solutions in a tree, we check the tree for a solution for the subinstance defined by the newly updated modifier, before we potentially solve the problem.

6.5 A Quick Overview

In this section we present a quick overview of the different implementations.

Although Clp is written primarily for solving linear programs, it comes with a QP solver. This QP solver uses an implementation of a primal-dual predictor-corrector interior point method. In addition to Clp's QP solver, we also have the QP solver presented in Chapter 4.

There are four different implementations we have, named `construct_clp`, `construct_slp`, `naive_clp` and `naive_slp`. The implementations with names ending with `_clp` and `_slp` uses Clp's QP solver and Slp, respectively, whenever a QP problem needs to be solved. The `construct` implementations uses the tree structure presented in Chapter 5 for storing subinstances with distinct solutions. The `naive` implementations, however, solves subinstances regardless of whether they have distinct solutions or not.

In the next chapter, we present several experiments to perform on our implementations.

Chapter 7

Computational Experiments

Throughout this chapter we refer to the implementations `construct_clp`, `construct_slp`, `naive_clp` and `naive_slp` as `cClp`, `cSlp`, `nClp` and `nSlp`, respectively.

Now that we are finished with implementations, we are interested in knowing how the methods perform, and how they compare with each other. Performance can be measured in different ways, and in the next section we discuss which experiments we run in order to assess performance.

We test our methods on the specific instances described in Section 3.1, to see how they perform on instances that follow the exact characteristics that the methods were developed for. Testing the performance of our methods on specific instances will tell us how it performs on those specific instances, but not how it performs in general. While the methods developed in this thesis are specified for instances with specific properties, the methods do not *require* these properties. To test the performance of our methods in general, we need to test it on random instances

In several experiments, we will test our methods on randomly generated instances. Unless specified otherwise, all randomly generated instances have the following properties:

- We let $m = \lfloor \frac{7}{20} n \rfloor$ in order to make our random instances roughly as sparse as those presented in Section 3.1.

- For each element b_i in vector b, b_i has a 50 percent probability of being 0.

- For each diagonal element h_{ii} in matrix H, h_{ii} has a 50 percent probability of being 0.

- If the elements in the two last points are *not* 0, then the values lie in

the intervals $10 \leq |b_i| \leq 70$ for b, and $10^{-5} \leq h_i \leq 10^{-1}$ for H, as specified in the beginning of Chapter 3.

Unless specified otherwise, we present all running times in experiments where we use randomly generated instances, as the *average* running time of 10 runs of each test, with a new random instance in each run. The reason for using the average number of seconds, and not the number of seconds used on the *best* run, is because we do not want one generated instance that happens to converge really fast with our algorithm, to represent the running time in the general case.

All implementations are executed with a tolerance of 10^{-7}, i.e. $\epsilon = 10^{-7}$, unless specified otherwise. The tolerance parameter for Slp is used as a stopping criteria for the algorithm, as described in Chapter 4. The tolerance parameter for Clp is used as a stopping criteria for the primal-dual predictor-corrector interior point method as implemented in Clp.

In the next section, we describe our experiments in detail.

7.1   Setup

Experiment 1: We test the different methods on the instance *small*, as introduced in Section 3.1, with varying tolerance. For each specific tolerance, we solve $\sigma(2, 238) = 28442$ subinstances, three times, and record the lowest of the three running times. The purpose is to assess how performance relates to specific tolerances.

Experiment 2: We test our tree structural methods on the instances introduced in Section 3.1. The purpose is to assess how the methods perform in terms of speed on real instances that Goodtech have faced. We run each implementation on each of the instances three times, and record the best out of the three runs.

Experiment 3: We compare `cClp` where we only solve subinstances with distinct solutions, with `nClp` where subinstances are solved regardless whether they are distinct or not. The purpose is to assess how fast our tree structural approach is. Testing these methods on instances of increasing size indicates how well they scale, in addition to indicating their speed. As n reaches high values, anything besides $\beta \leq 1$ is unrealistic, so we let the maximum number of breakdowns be $\beta = 1$.

Experiment 4: We continue to compare the tree structural approach with naively solving all subinstances regardless whether they are distinct or not. We see how the two methods scale as β increases. Since the number of subinstances increases rapidly as β increases, this approach is only feasible

for small values of β. We consider this approach infeasible for any $n > 50$, so we let $n = 50$.

All experiments are performed on a machine running 64 bit Gentoo Linux with version 3.6.11 of the linux kernel. The machine has the following relevant hardware specifications:

- Intel(R) Core(TM) i7 CPU 930 @ 2.80GHz,

- Corsair XMS3 DDR3 1600MHz 12GB CL9,

- Socket LGA 1366.

The CPU has 4 cores and has the following multi-level cache specifications [20]:

- 32KB L1 data cache and 32KB L1 instruction cache per core,

- 256KB L2 data cache per core,

- 8MB L3 data cache shared by all cores.

All codes are compiled with GCC version 4.6.3 with `-O3` optimization flag enabled.

7.2   Results and Observations

In this section, we present results from the experiments introduced in the previous section.

Experiment 1

Table 7.1 shows the running time in CPU-seconds required to solve instance *small*, along with $\sigma(2, 238) = 28442$ of its subinstances, for different tolerances. It is very clear that Slp can not keep up with Clp's QP solver when the tolerance reaches values equal to 10^{-5} and lower. Figure 7.1 shows plotted graphs on the logarithmic scale of the values in Table 7.1.

The reason for Slp being so slow with low-valued tolerances is most likely that Slp converges really slowly as the termination condition becomes too strict. We can measure the number of Slp iterations required to solve a given instance given some tolerance. By solving randomly generated instances where $n = 50$, with Slp, we note the average number of iterations among ten solved instances. Table 7.2 shows the results from this, and we see that the average number of iterations increases rapidly as the tolerance becomes smaller than 10^{-4}. This fits our prediction about why `cSlp` and `nSlp` becomes much slower at lower tolerances.

Table 7.1: Results from experiment 1.

ϵ	cClp	cSlp	nClp	nSlp
10^{-1}	45.51	55.61	72.32	85.51
10^{-2}	46.34	55.89	73.11	85.51
10^{-3}	51.12	59.04	75.60	85.28
10^{-4}	52.46	73.79	77.83	107.39
10^{-5}	54.48	232.53	81.16	355.47
10^{-6}	65.42	1363.46	93.29	2022.25
10^{-7}	70.78	6522.91	100.85	9395.92

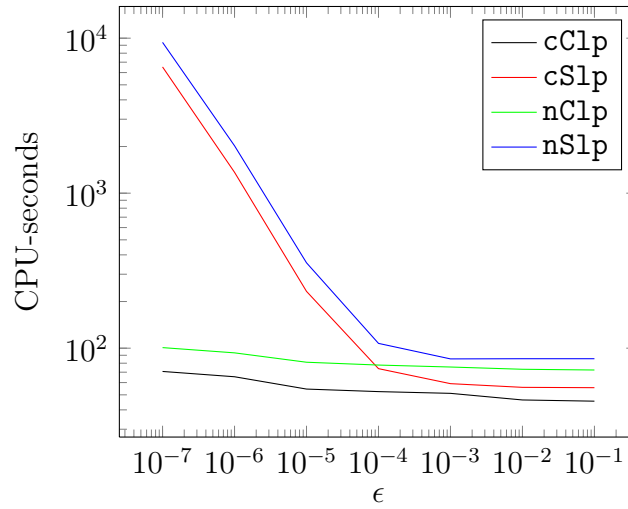Figure 7.1: Running time in CPU-seconds required to solve *small* and $\sigma(2, 238)$ of its subinstances, as a function of ϵ.

Experiment 2

The goal in this experiment is to see how the methods perform on static instances of increasing size. We learned in the previous experiment that Slp can not keep up as the tolerance reaches values lower than 10^{-4}, so we let $\epsilon = 10^{-4}$ to see if Slp may keep up as n increases. Table 7.3 shows the running time in CPU-seconds required to solve the three instances. Unfortunately, we see that even with a high tolerance, Slp can not keep up with Clp as n increases.

Table 7.2: Average number of iterations used by Slp with different tolerances over ten randomly generated instances.

ϵ	Iterations
10^{-1}	1
10^{-2}	1.8
10^{-3}	2.2
10^{-4}	3.5
10^{-5}	8.6
10^{-6}	23.7
10^{-7}	90.1
10^{-8}	223.5
10^{-9}	752.6

Table 7.3: Running times observed in Experiment 2.

Implementation	*small*	*large*	*vlarge*
cClp	0.52	9.55	76.18
cSlp	0.71	32.88	585.60
nClp	0.65	11.68	157.53
nSlp	0.89	39.87	1173.74

Experiment 3

Since the two previous experiments indicate that Clp performs better, we go on to evaluate the tree structural approach based on this solver. Table 7.4 shows an excerpt of the test results. The table shows the average running time in CPU-seconds required to solve random instances of increasing size, along with $\sigma(1, n) = n + 1$ of its subinstances. The rightmost column shows the relative speedup of cClp over nClp. A table with results for all $n = 100, 200, \ldots, 2000$ can found in Appendix A.1. We see that the relative speedup of cClp over nClp increases with the instance size. This is due to the fact that we save more time for each instance we do *not* need to solve, as n increases, simply because larger instances require more time to solve. There are also more subinstances as n grows, making room for more subinstances that we might not have to solve.

Experiment 4

Table 7.5 shows the average running time in CPU-seconds required to solve ten random instances along with $\sigma(\beta, 50) = \sum_{j=0}^{\beta} \binom{50}{j}$ of its subinstances.

Table 7.4: An excerpt of test results from Experiment 3.

n	cClp	nClp	Relative Speedup
500	4.9	5.9	16.9%
1000	42.1	53.0	20.6%
1500	181.5	234.5	22.6%
2000	547.1	710.2	23.0%

The rightmost column represents how many subinstances we actually solved with a call to a QP solver, because their solutions were not found in the tree. We see that the relative speedup of cClp over nClp increases with β. This is due to the fact that as the tree grows in size, the chance that an unsolved subinstance is distinct, decreases. In fact, looking at the rightmost column in Table 7.5, we see that this is the case. This helps us clarify the reason why the relative speedup of cClp over nClp increases.

Table 7.5: Results from Experiment 4.

β	cClp	nClp	Relative Speedup	Distinct Solutions
1	0.03	0.04	25.0%	37.4 (74.3%)
2	0.64	0.94	31.9%	744.3 (58.3%)
3	7.06	15.90	55.6%	9484.7 (45.4%)
4	77.59	188.83	58.9%	82262.5 (32.8%)
5	586.54	1758.23	66.6%	574685.0 (24.2%)

Figure 7.2 shows a plot of the average number of CPU-seconds required to solve one subinstance, as a function of β. We see that as β increases, the number of seconds required to solve one subinstance decreases in the case of cClp, but for nClp, it does not change much. This behavior is very much expected, especially considering our earlier assessment in this experiment.
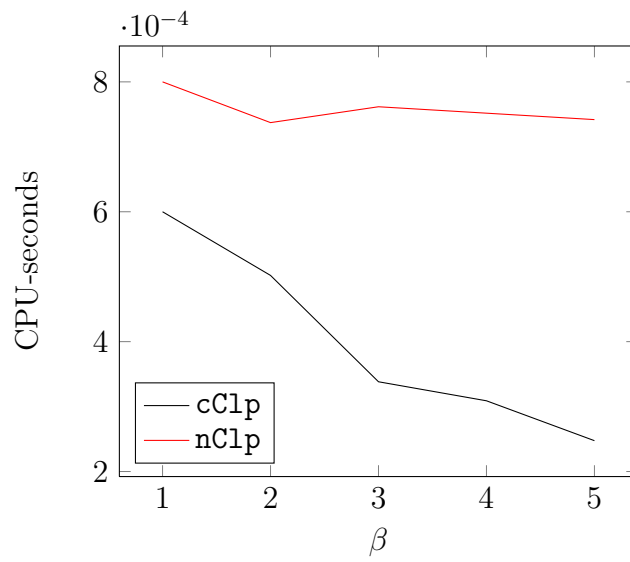
Figure 7.2: Average running time in CPU-Seconds required to solve one subinstance, as a function of β.

Chapter 8

Conclusion

Goodtech and MathConsult have developed a tool called PROMAPS that calculates the power delivery as a function of demand, and the probability for undelivered energy for each load branch in the system, and in the system as a whole. Among several main functions of PROMAPS, its bottleneck was pinpointed to a call to a QP solver. The aim of this thesis was to improve the performance of the methods initiated by this call.

We presented an iterative method based on linear programming, as the presented data suggested that the QP problems were— in lack of a better word—*close* to linear programs. Although this method turned out to be slower than a commercially available alternate solver, due to the fact that the number of iterations increases quite rapidly as the tolerance decreases, it does stand as an alternative method, mainly because it does not require any license fees. It is important to point out that the method is entirely independent of specific LP implementations. The only requirement for implementing the method is an LP solver. There are more available LP solvers than QP solvers in general, but there are also more free LP solvers than free QP solvers [21] [22].

Afterwards, we presented a tree structural approach for reducing the number of calls to the QP solver. This approach reduced the number of calls to the QP solver by a significant amount. Depending on problem size and the number of breakdowns, this approach caused the original method to perform from 16% to around 70% faster, and we have reason to believe that it would increase even further with β. The tree structural approach therefore reduced the average number of CPU-seconds used to solve a subinstance. This method is independent of the choice of QP solver, and since the method relies on reducing the number of calls to the QP solver, and not increasing the speed of the actual solver, the relative speedup will most likely remain solver independent.

Chapter 9

Future Work

In this chapter, we present suggestions to future work regarding the methods described in this thesis.

9.1 Parallelization

Since we are solving a huge amount of QP problems, running several QP solvers in parallel is of interest. We suggest the parallelization of the methods presented in this thesis as a future research topic.

If we are to parallelize the naive method on a distributed computer, we need to analyze the amount of communication that is needed between processors. Given some number of breakdowns β and an instance $\mathcal{Q}$ with m rows and n columns, we can map each subinstance of $\mathcal{Q}$ to some $\mathcal{Q}_i$ for $i = 0, 1, \ldots, \sigma(\beta, n) - 1$. Each processor needs to have a copy of the original instance $\mathcal{Q}$, which is $O(mn)$ in terms of memory size. Given a number of processors p, the main processor needs to transmit this problem to each processor, resulting in $O(pmn)$ in terms of communication before the calculations are started. Now, each processor solves $\frac{\sigma(\beta,n)}{p}$ subinstances, resulting in

$$O(\sigma(\beta, n)n)$$

in terms of communication by sending solutions back to the main processor. Sending few, but huge blocks of data is often faster than sending many small blocks of data. Therefore, an alternative approach is to store all solutions on each processor, and then transmitting them all to the main processor. This would require

$$O\left(\frac{\sigma(\beta, n)n}{p}\right)$$

in terms of memory for each processor.

Further research topics may include parallelization of both the naive approach and the tree structural approach on both a shared memory computer and a distributed computer.

9.2   Coordinate Descent

Consider a minimization problem

$$\min f(x) = x^T H x + b^T x,$$

where H is diagonal a $n \times n$ matrix, and x and b are vectors in $\mathbb{R}^n$. Since H is diagonal, the contours of f are ellipses whose axes are aligned with the coordinate directions [4]. By performing successive one-dimensional minimizations along the coordinate directions $e_1, e_2, \ldots, e_n$, we can find the minimizer of f in n iterations. This is illustrated in Figure 9.1 for $n = 2$.
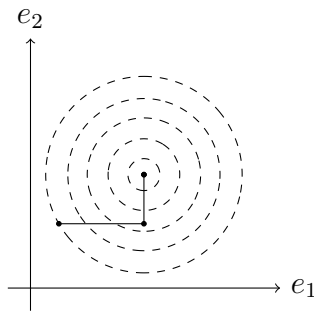

Figure 9.1: Illustration of minimization along the coordinate directions of a quadratic with a diagonal Hessian.

We suggest the development of a direct method based upon coordinate descent as a future research topic.

Bibliography

[1] T. Digernes, A. B. Svendsen, Y. Aabø, C. Hernandez, and M. T. Pálsson, "Analyses of delivery reliability in electrical power systems," *European Safety and Reliability*, 2007.

[2] A. B. Svendsen, J. Eman, T. Tollefsen, Y. Aabø, T. Digernes, S. Løvlund, and J. O. Gjerde, "Online reliability assessment of power system," *The 12th International Conference on Probabilistic Methods Applied to Power Systems*, 2012.

[3] R. Vanderbei, *Linear Programming: Foundations and Extensions*. International Series in Operations Research & Management Science, Springer, 2007.

[4] J. Nocedal and S. Wright, *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering, Springer, 2006.

[5] B. Grünbaum, *Convex Polytopes*. Graduate Texts in Mathematics, Springer, 1967.

[6] S. Boyd and L. Vandenberghe, *Convex Optimization*. Berichte über verteilte messysteme, Cambridge University Press, 2004.

[7] R. Rockafellar, *Convex analysis*. Princeton landmarks in mathematics and physics series, PRINCETON University Press, 1997.

[8] V. Klee and G. J. Minty, "How good is the simplex algorithm?," in *Inequalities* (O. Shisha, ed.), vol. III, pp. 159–175, New York: Academic Press, 1972.

[9] M. K. Kozlov, S. P. Tarasov, and L. G. Khachiyan, "Polynomial solvability of convex quadratic programming," *Doklady Akademii Nauk SSSR*, vol. 248, no. 5, pp. 1049–1051, 1979.

[10] S. Sahni, "Computationally related problems," *SIAM J. Comput.*, vol. 3, no. 4, pp. 262–279, 1974.

[11] T. Apostol, *Calculus: Multi-variable calculus and linear algebra, with applications to differential equations and probability.* Calculus, Wiley, 1969.

[12] R. E. Griffith and R. A. Stewart, "A nonlinear programming technique for the optimization of continuous processing systems," *Management Science*, vol. 7, pp. 379–392, July 1961.

[13] P. Boggs, R. Byrd, R. Schnabel, S. for Industrial, and A. Mathematics, *Numerical Optimization 1984: Proceedings of the Siam Conference on Numerical Optimization, Boulder, Colorado, June 12-14, 1984.* Proceedings in Applied Mathematics Series, Siam, 1985.

[14] J. Forrest, D. de la Nuez, and R. Lougee-Heimer, "Clp user guide." `http://www.coin-or.org/Clp/userguide/`, 2004. [Online; accessed March 2013].

[15] "Glpk (gnu linear programming kit)." `http://www.gnu.org/software/glpk/`, 2012. [Online; accessed March 2013].

[16] "Introduction to lp_solve 5.5.2.0." `http://lpsolve.sourceforge.net/5.5/`, 2012. [Online; accessed March 2013].

[17] J. W. Gregory and R. Fourer, "Neos optimization guide, linear programming faq." `http://www.neos-guide.org/lp-faq`, 2005. [Online; accessed March 2013].

[18] J. Forrest, D. de la Nuez, and R. Lougee-Heimer, "Coin-or linear programming solver." `https://projects.coin-or.org/Clp`, 2004. [Online; accessed March 2013].

[19] W. S. Voon, "Combinations in c++." `http://www.codeproject.com/Articles/4675/Combinations-in-C`, 2009. [Online; accessed April 2013].

[20] "Intel core i7-930 processor (8m cache, 2.80 ghz, 4.80 gt/s intel qpi)." `http://ark.intel.com/products/41447`. [Online; accessed May 2013].

[21] Wikipedia, "Linear programming — Wikipedia, the free encyclopedia," 2013. [Online; accessed May 2013].

[22] Wikipedia, "Quadratic programming — Wikipedia, the free encyclopedia," 2013. [Online; accessed May 2013].

Appendix A

Tables

A.1   Results From Experiment 3

n	cClp	nClp
100	0.1	0.2
200	0.5	0.7
300	1.3	1.7
400	2.8	3.5
500	4.9	5.9
600	7.7	10.9
700	13.2	15.6
800	20.3	26.2
900	29.8	36.7
1000	42.1	53.0
1100	57.2	75.4
1200	76.4	107.4
1300	107.4	137.0
1400	141.7	180.3
1500	181.5	234.5
1600	216.7	300.6
1700	295.8	381.4
1800	366.5	483.6
1900	448.0	611.5
2000	547.1	710.2

Appendix B

Codes

B.1 bitset vs set

```
#define N 65
#define SIZE 32*N

static set<uint16_t>* copy(const bitset<SIZE>* a) {
  set<uint16_t>* res = new set<uint16_t>;
  for (uint16_t i = 0; i < SIZE; i++) {
    if (a->test(i)) res->insert(i);
  }
  return res;
}

int main() {
  srand((unsigned int)time(NULL));

  vector<bitset<SIZE>*> bitsets;

  for (int i = 0; i < 3000; i++) {
    stringstream ss;
    for (int j = 0; j < N; j++) {
      bitset<32> bs(rand());
      ss << bs.to_string();
    }
    bitsets.push_back(new bitset<SIZE>(ss.str()));
  }
```

```
vector<set<uint16_t>*> sets;
for (int i = 0; i < 3000; i++) {
  sets.push_back(copy(bitsets[i]));
}

bool fool;

double t1 = omp_get_wtime();
for (uint16_t i = 0; i < sets.size(); i++) {
  for (uint16_t j = 0; j < sets.size(); j++) {
    fool = isSubset(*sets[i], *sets[j]);
  }
}
double t2 = omp_get_wtime();

for (uint16_t i = 0; i < bitsets.size(); i++) {
  bitset<SIZE> notb = ~(*bitsets[i]);
  for (uint16_t j = 0; j < bitsets.size(); j++) {
    fool = isSubset_bit(notb, *bitsets[j]);
  }
}
double t3 = omp_get_wtime();

cout << "(" << SIZE << ", " << (t2-t1) << ")\n";
cout << "(" << SIZE << ", " << (t3-t2) << ")\n";
}
```

B.2   next_combination

```
template <class BidIt> inline bool next_combination(
BidIt n_begin, BidIt n_end, BidIt r_begin, BidIt r_end)
{
  bool boolmarked=false;
  BidIt r_marked;

  BidIt n_it1=n_end;
  --n_it1;
```

```
BidIt tmp_r_end=r_end;
--tmp_r_end;

for(BidIt r_it1=tmp_r_end;
r_it1!=r_begin || r_it1==r_begin; --r_it1,--n_it1)
{
  if(*r_it1==*n_it1)
  {
    if(r_it1!=r_begin)
    {
      boolmarked=true;
      r_marked=(--r_it1);
      ++r_it1;
      continue;
    }
    else
      return false;
  }
  else
  {
    if(boolmarked==true)
    {
      BidIt n_marked;
      for (BidIt n_it2=n_begin; n_it2!=n_end; ++n_it2)
      {
        if(*r_marked==*n_it2) {
          n_marked=n_it2;break;
        }
      }


      BidIt n_it3=++n_marked;
      for (BidIt r_it2=r_marked;r_it2!=r_end;++r_it2,++n_it3)
      {
        *r_it2=*n_it3;
      }
      return true;
    }
    for(BidIt n_it4=n_begin; n_it4!=n_end; ++n_it4)
      if(*r_it1==*n_it4)
```

```
        {
          *r_it1=*(++n_it4);
          return true;
        }
    }
  }

  return true;//will never reach
}
```

B.3   Generate Random Instance

```
static
CoinPackedMatrix packMatrix(double** m, int rows, int cols)
{
    std::vector<int> row_ind_vec;
    std::vector<int> col_ind_vec;
    std::vector<double> ele_vec;
    CoinBigIndex numels = 0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            double num = m[i][j];
            if (num != 0.0) {
                row_ind_vec.push_back(i);
                col_ind_vec.push_back(j);
                ele_vec.push_back(num);
                numels++;
            }
        }
    }

    const int *rowIndices = &row_ind_vec[0];
    const int *colIndices = &col_ind_vec[0];
    const double *elements = &ele_vec[0];

    CoinPackedMatrix cpm(false, rowIndices, colIndices,
                         elements, numels);
    cpm.setDimensions(rows, cols);
```

```
        return cpm;
}

static double g_randd(double min, double max)
{
        double num = (double) rand() / RAND_MAX;
        return (min + (num * (max - min)));
}

static int g_randi(int min, int max)
{
        return (rand() % (max-min) + min);
}

ClpModel randomInstance(int vertices, int edges, double Hzero,
double bzero)
{
        srand((uint16_t)time(NULL));

        /* Matrix A */
        double** m = (double**) malloc(vertices*sizeof(double*));
        for (int i = 0; i < vertices; i++)
                m[i] = (double*) calloc(edges,sizeof(double));

        int row = 0;
        for (int i = 0; i < edges; i++) {
                m[row][i] = 1;
                int r = row;
                while (r == row) row = rand() % vertices;
                m[row][i] = -1;
        }
        CoinPackedMatrix A = packMatrix(m, vertices, edges);

        for (int i = 0; i < vertices; i++)
                free(m[i]);
        free(m);

        /* Matrix H */
        std::vector<int> row_ind_vec;
        std::vector<int> col_ind_vec;
        std::vector<double> ele_vec;
```

```
int numels = 0;

for (int i = 0; i < edges; i++) {
    /* If not 0 */
    if (((double)rand() / RAND_MAX) >= Hzero) {
        double num = g_randd(0.00001, 0.01);
        row_ind_vec.push_back(i);
        col_ind_vec.push_back(i);
        ele_vec.push_back(num);
        numels++;
    }
}
const int *rowIndices = &row_ind_vec[0];
const int *colIndices = &col_ind_vec[0];
const double *elements = &ele_vec[0];

CoinPackedMatrix H(false, rowIndices, colIndices, elements,
                   numels);
H.setDimensions(edges, edges);

/* Vector b */
double* b = (double*) calloc(edges,sizeof(double));

for (int i = 0; i < edges; i++) {
    /* If not 0 */
    if (((double)rand() / RAND_MAX) >= bzero) {
        double num = g_randi(10, 70);
        if (num >= 30) num = -num;
        b[i] = num;
    }
}

/* Vector lb */
double* lb = (double*) malloc(edges*sizeof(double));
for (int i = 0; i < edges; i++) {
    lb[i] = -g_randd(0.0, 1000.0);
}

/* Vector ub */
double* ub = (double*) malloc(edges*sizeof(double));
for (int i = 0; i < edges; i++) {
```

```
        ub[i] = g_randd(0.0, 1000.0);
    }

    ClpSimplex model;

    model.loadProblem(A, lb, ub, b, 0, 0);
    model.loadQuadraticObjective(H);

    free(b);
    free(lb);
    free(ub);

    return model;
}
```