A cutting plane algorithm for robust scheduling problems in medicine

by

Kjetil Matias Holte

THESIS

for the degree of

Master of Science

(Master i Anvendt matematikk og mekanikk)

Faculty of Mathematics and Natural Sciences
University of Oslo

May 2010

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Abstract

Scheduling problems are central in combinatorial optimization, and there exists a huge literature describing both problems and algorithms. Master surgery scheduling (MSS), where one must assign surgery teams in hospitals to different rooms at different times, is a specialization of the assignment problem, which can be solved using mixed integer programming (MIP).

If not all parameters to an optimization problem are known beforehand, we may use robust optimization to find solutions which are both feasible and good, even if the parameters are not as expected. In particular, we assume such parameters to vary in a given set of possible "realizations". The more realistic assumptions, the better solutions.

In this thesis we model MSS with the aim of minimizing the expected queue lengths in hospitals. The model is made robust by considering the demand to be uncertain, but belonging to a simple polytope. It can be modeled as a bilevel program. The master program looks for feasible schedules minimizing queue lengths, while the slave program looks for feasible demands maximizing queue lenghts.

In order to solve this bilevel program, we use an iterative cutting plane approach. We find a solution with the best possible behaviour for the worst case parameters, by splitting the problem in two. In particular, we implemented the so-called "implementor/adversary" scheme (I-A), recently proposed by Bienstock in the context of portfolio optimization (see [7, 8]). The implementor finds an optimum schedule with respect to a restricted set of feasible parameter vectors. The adversary finds a new vector (not included in the restricted set) which maximizes queues with respect to the current schedule. The new vector is included in the restricted set and the method is iterated. When the adversary is not able to find a new parameter realization which is worsening the value of the current schedule, we are finished. In our experiments on realistic instances we need at most a few hundred of iteration before convergence is reached.

For comparison, we also make a non-robust model of MSS as a reference solution, where we only consider a single demand vector.

Testing shows that I-A is indeed a very effective algorithm, solving large problem instances in reasonable time. Moreover, the solutions were in general found to be considerably better than the non-robust reference solution, even in cases where the demand did not belong to the polytope we assumed. The algorithm also gives us good intermediate solutions with provable bounds on optimality, which can be used even if convergence is not reached.

We believe I-A both offers more flexibility and yields better results compared to the other robust approaches, when applied to the right problems, MSS only being one of these.

Acknowledgements

First of all I would like to express my deepest gratitude to my supervisor, Carlo Mannino for all his help and support. Your open attitude towards a young man who wants to be a researcher is invaluable. Thank you for always being helpful and available, and giving me nudges in the right direction when I was stuck.

I would also like to thank Tomas Norlander and Truls Flatberg for introducing me to the field of surgery scheduling, and Geir Dahl for introducing me to linear programming.

I would like to thank Inés Cendón Rodrígues and Irene Brox Nilsen for help on spelling and grammar and John Christian Ottem for help with LaTeX and for giving me the laugh of the day.

I would like to thank all friends for their love and friendship, and for giving me joy in my everyday. Thank you to all student organisations that have given me a social life besides the studies, the Ares gaming club, IAESTE and Biørneblæs. In general, I would like to thank Studentorchesteret Biørneblæs for generally helping me with my "Master av stål". Without you all I would surely have gone mad.

Most of all I would like to thank all my family for providing me with love and caring always. Your support and inspiration has been invaluable.

Background

Managing large hospitals involves exploiting, assigning, coordinating and scheduling a large number of heterogeneous resources [10, 2]. Not just the doctors and personnel, but also rooms, equipment and many other resources. Optimal use of these resources can save huge amounts of money and also improve the number and quality of treatment of patients. Even small improvements in the planning algorithms can lead to huge improvements, given the large number of hospitals and their sizes. This makes planning in health care in general, and surgery scheduling in particular, an important subject to study.

There are several reasons why finding an optimal surgery schedule is complicated. We need to consider the people directly involved, the surgeons teams and the patients. This is hard enough by itself, with shared resources, requirements and wishes regarding work hours, overtime and holiday planning. But the surgery schedule also affects other departments, like the ward used for recovery [1]. Defining optimality is not easy either. It can be to minimize cost, maximize income, cure as many patients as possible, minimize the number of cancellations, distribute the work load evenly among the doctors, minimize patient queues, ensure efficient use of resources, rooms, wards and personnel or a combination of all these. On top of all this, we must consider uncertainty [17]. The patient demand may vary, duration of surgery and recovery is uncertain, emergencies may disrupt the plan and resources may be unavailable due to illness, repair or cleaning.

To cope with the complex task of surgery scheduling, the problem must be simplified. We may use generous safety margins to ensure that certain constraints are always satisfied, replace objectives with bounds, only requiring that the solution is "good enough", or even ignore parts of the problem, assuming that the missing parts can be adjusted to fit the solution afterwards.

Reduction of the solution space makes the problem easier to solve, but is also likely to remove optimal solutions, reducing the optimal value. And solutions which are optimal with respect to some criterion, may be suboptimal when considering the

whole.

Nevertheless, this is the reality we are facing in hospitals today. Surgery schedules are created manually using simple heuristics and used for years until it is so bad that a new plan is required.

Recently, scientists have come up with algorithms and implementations solving some of the subproblems [1]. These implementations are already used in hospitals to aid in the strategical and tactical planning. However, these tools have their limitations, in that they do not look at the complete picture or make false assumptions to simplify calculations.

The implementor-adversary algorithm (I-A) which will be described in this thesis has a different approach than most other algorithms. Instead of solving a static problem, new cuts are added continuously until we are satisfied with the solution. I-A can handle several objectives and variables, which may also be uncertain. Linearity is not a requirement, neither is convexity. This flexibility makes it possible to consider a bigger picture.

Outline

The theoretical background for the algorithms will be discussed in section 1, starting with linear programming, but also including theory on game theory, bilevel programming, robust optimizations and the dynamic simplex method.

Section 2 describes the background of master surgery scheduling problems in more detail, before it defines the model that will be used with a complete mathematical problem statement. The model is a basis for the algorithm which will be described in section 3, together with possible code optimizations for implementation on a computer.

In section 4 we describe the test cases, before we show the results, both in terms of running time, convergence, and compared to other approaches. Finally, in section 5 we discuss the results and future work, before we draw a conclusion.

Contents

1 Theoretical background

In this section we will first give a short introduction to linear programming. After we have covered the basics, we will look at more advanced topics, like mixed integer programming and bilevel programs. Then we will introduce game theory, before we describe the dynamic simplex algorithm and cutting plane methods. Finally we discuss robustness and how to solve robust programs.

1.1 Introduction to linear programming

A *linear program* is a problem where we seek to maximize or minimize a linear function of some variables which themselves are restricted by a set of linear constraints defining a convex polytope. Linear programming has a huge range of application, from logistics to economics, from scheduling to manufacturing, and is very widely used.

1.1.1 Definitions

For a deeper and more complete introduction to linear programming, the reader can consult books such as Vanderbei [24, chapter 1-6] or Cormen et.al. [12, pp 770-821].

Definition 1.1. A *linear program* in standard form (matrix notation) is a problem on the form:

$$\text{maximize: } c^T x \tag{1}$$
$$\text{subject to: } Ax \leq b \tag{2}$$
$$x \geq 0 \tag{3}$$

The *decision variables* are, as the name implies, the unknowns in the problem, and are usually denoted by x or y. An assignment of values to the decision variables is called a *solution*. If the solution violates any of the *constraints* (2), it is called *infeasible*, else it is called *feasible*. The purpose is to find the *optimal value* to an *objective function* (1). A feasible solution where the objective function attains its optimal value is called an *optimal solution*. A problem where there are no limits to the optimal value is called *unbounded*. We usually also require the variables to be non-negative (3), though this constraint may be circumvented.

1.2 Solving linear programs

The *simplex method* for solving linear programs was invented by G. Dantzig in 1947, which led to a boost in the number of problems formulated as linear programs. Variants of the simplex method are still among the most popular methods for solving linear programs.

Even though Klee and Minty [20] constructed an example where the simplex method runs in exponential time, the algorithm is very fast in practice [23], and usually outperforms the polynomial-time ellipsoid algorithm. Later, Karmakar [19] described polynomial-time interior point methods which now have become increasingly popular.

To solve a linear program consists of:

1. decide whether the problem is feasible (solvable)
2. if it is feasible, decide whether it is unbounded

3. if it is bounded, find any solution x^* where the objective function takes the optimal value.

The simplex method is capable of all three, and also to prove that the result was indeed correct by reordering the indices and using duality.

1.3 Duality

Definition 1.2. Given a problem in standard (matrix) form (1)-(3), the *dual* problem is given by:

$$\begin{aligned} \text{Minimize} \quad & b^T y \\ \text{Subject to:} \quad & A^T y \geq c \\ & y \geq 0 \end{aligned}$$

The dual to a dual problem, is the original or *primal* problem. The primal and dual are closely related, not only in the coefficients, but also in solutions and optimal value. It turns out that every solution to the dual problem gives a bound for the primal problem and vice versa.

We will only state the results, for proofs, the reader may consult Vanderbei [24, chapter 5].

Theorem 1.3 (Weak duality)**.** *Any feasible solution to the primal is less or equal to any feasible solution to the dual. In other words,*

$$c^T x \leq b^T y$$

Thus if we have a feasible solution x to the primal and a feasible solution y to the dual, such that

$$c^T x = b^T y$$

then both x and y are optimal solutions to the primal and dual problem respectively.

Theorem 1.4 (Strong duality)**.** *If a linear program has an optimal solution (i.e. it is feasible and bounded), then the dual also has an optimal solution, and the objective values are equal.*

1.4 Integer variables

In linear programming, all constraints are linear. In particular, all variables are allowed to take a closed interval of values. In practice, this is not always the case. When you are to assign between 1 and 5 persons to a project, you do not want a solution which assigns 3.5 persons.

A problem where all variables only take integer values is called an *integer program* (IP). If we have both continuous and integer variables, we have a *mixed integer program* (MIP).

Example 1.5 (Integer knapsack problem)**.** Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items.

The obvious greedy solution, to pick the items with the highest value/weight ratio does not work, as the following example shows:

Weight	2	2	3
Value	3	3	5

The item with weight 3 is the most valuable, both absolute and by ratio. However, the two smaller items are better if your knapsack has capacity 4.

The integer knapsack problem is NP-hard, though pseudo-polynomial algorithms using dynamic programming exist.

By reduction to the knapsack problem, it can be shown that most IPs and MIPs are in fact NP-hard. For instance, the related problem where you have a set of items and want to minimize the number of knapsack needed to pack them all, is called the *bin packing problem*, and is also NP-hard.

1.4.1 Relaxation

We can not give up on IPs just because they are difficult. In many cases, good, even optimal solutions can be found in reasonable time. A normal approach is to *relax* the constraints, to allow for illegal solutions and hope we get a result that is not too far away from the optimal solution, both in optimal value and in the solution vector.

An example of a relaxation is $x \in \{0, 1, 2, 3\} \Rightarrow x \in [0, 3]$

When we relax a problem, we search for a solution over a larger domain. Thus the optimal value cannot be worse for a relaxed problem; there are no feasible solutions in the original problem that are unfeasible in the relaxed problem.

$$\max \{c^T x | Ax \leq b,\ x \geq 0,\ x \in \mathbb{Z}\} \leq \max \{c^T x | Ax \leq b,\ x \geq 0\}$$

If we solve a relaxed problem, we thus get an upper bound on the solution of the original problem. If we later find a solution to the original problem with optimal value equal to the upper bound, we know that the solution must in fact be optimal.

1.4.2 Branch and bound

We have just seen how to attain an upper bound. We have also seen that any feasible solution must be a lower bound.

Suppose we solve the relaxed problem, and we get an optimal solution x^* which has one or more non-integer values. Then we can add new constraints which avoid this solution. For instance, if we have the relaxed constraint $0 \leq x_1 \leq 4$ and we get a solution with $x_1^* = 2.5$. Then, because x_1 is integer in the non-relaxed problem, we know that it must either be in the interval $[0, 2]$ or $[3, 4]$. In fact, it must be in one of the intervals $[0, 0], [1, 1], [2, 2], [3, 3]$ or $[4, 4]$

These constraints are disjunct, thus we can not add all of them to the problem, as it would obviously make our problem infeasible. But we can split the problem in two subproblems, one with the constraint $x_1 \in [0, 2]$ and one with the constraint $x_1 \in [3, 4]$. One or both subproblems must contain an optimal solution to the unrelaxed problem, as we have only removed fractional infeasible solutions with $x_1 \in (2, 3)$

This branching helps us restrict the problem to exclude infeasible solutions. But now we have 2 problems to solve, instead of one. And we are not guaranteed that these will be easy to solve either. In a problem with many integer variables, we might have to split the problem many times. The set of problems we are considering are called a *search tree*. Just by splitting a problem 20 times, we have more than a million

subproblems (2^{20}). And there might be hundreds of integer variables taking fractional values. We need what is called *cuts*, and for this we will use bounds extensively.

We will work with global lower and upper bounds. A global lower bound is the best of all feasible solutions found so far (the goal is to maximize). A global upper bound is the highest optimal value of all relaxed subproblems. When the global lower and upper bound is equal, we know that we have found the optimal solution.

Similarly, we can find local lower and upper bounds, which only apply to a subproblem. If we find that a local upper bound < global lower bound, then we know that we will never be able to improve the global lower bound with this subproblem, and we can cut it away from the search tree. We say that the node is *pruned*.

There are basically two strategies [21, pp 358-359]. One is *depth-first*, that is, when we split a problem, we only consider one branch. It has the advantage that the branch is very similar in structure to its root, and can often to be reoptimized using a few steps with the dual simplex algorithm. Furthermore, by finding feasible solutions, we attain global lower bounds which may be used for early pruning, reducing the number of nodes to consider. Finally, experience seems to indicate that feasible solutions are more likely to be found deep in the tree than at nodes near the root.

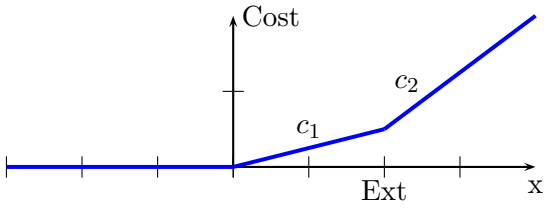
The second strategy is to go *breadth-first*, where we branch all nodes at the current level before we look at nodes deeper in the tree, but this typically requires much memory and results in less pruning.

In practice, we almost always use a depth-first search. However, when we have reached a leaf node (a feasible solution), there are several ways of choosing the next active node. Traditionally depth-first searches use backtracking, we move up in the tree until we see a branch that we have not considered before. But it is also possible to use other criteria, such as choosing the node with the best upper bound.

1.5 Non-linearity

Until now, we have thought of the objective function as a linear function. In many cases, we need more complex functions. The simplest of these are the piece-wise linear functions. In figure (1) is an example of such a function.

Figure 1: Cost function



This function could model the cost of satisfying a certain demand. Negative x means that demand is already satisfied. If we can not satisfy the demand, we have to pay a penalty c_1 for each item that is missing (e.g. to replace the missing items). After a certain threshold Ext, the penalty c_2 is much higher (because we must compensate if all the items can not be replaced anymore).

Note that the function is convex, and that we are minimizing. This is essential, because with a convex function, the slopes are always increasing, and it allows us to be

greedy.

The following linear program is sufficient for solving with the objective function above:

Example 1.6 (minimizing convex function)**.**

$$\begin{aligned} \text{minimize: } & \zeta = c_1 x_1 + c_2 x_2 \\ & x_0 \leq 0 \\ & 0 \leq x_1 \leq 2 \\ & 0 \leq x_2 \\ & x = x_0 + x_1 + x_2 \end{aligned}$$

The objective function ensures that we increase x_0, x_1 and x_2 in that order, because this will be the cheapest. We do not need to disallow solutions like $(x_0, x_1, x_2) = (-1, 1, 3)$, which are feasible but unwanted, as this is inferior to the wanted solution $(0, 2, 1)$ where we have maximized x_0 and x_1 first.

We can even add up many objective functions like this, and it will work just fine. This is because we are minimizing over convex functions, and a sum of convex functions is also convex.

Now suppose we want to *maximize* ζ, maybe because we want to know the worst case that can happen. Then the above construction will not work, because the algorithm will increase x_2 before x_0 and x_1. In fact, the problem is unbounded as you can choose $x_0 \to -\infty$ and $x_2 \to \infty$. We would like to force x_0 to be maximal before we increase x_1, and to force x_1 to be maximal before we increase x_2. For this we need binary variables.

Let

$$y_1 = \begin{cases} 0 & x_1 = x_2 = 0 \\ 1 & x_0 = 0 \end{cases} \tag{4}$$

$$y_2 = \begin{cases} 0 & x_2 = 0 \\ 1 & x_0 = 0, x_1 = 2 \end{cases} \tag{5}$$

The binary decision variables y_1 and y_2 tell us which of the normal variables are not allowed to be used. If $y_1 = 1$, we may use x_1 and x_2, but then we also require x_0 to be at its maximum. Similarly, if $y_2 = 1$, we may use x_2, but then x_0 and x_1 are fixed at their maxima. In order to accomplish this, we need to use a trick.

$$x_1, x_2 \leq y_1 M \tag{6}$$
$$x_0 \geq -(1 - y_1)M \tag{7}$$
$$x_2 \leq y_2 M \tag{8}$$
$$x_1 \geq 2 - (1 - y_2)M \tag{9}$$

In addition come the constraints from example (1.6). Constraint (6) and (7) gives us (4), assuming that M is large enough so that $x_2 < M$ and $x_0 > -M$. Constraint (8) gives us the first line in (5), while (9) gives the second line in (5). In the last case, $x_0 = 0$ is implied, because $x_1 = 2$.

1.6 Introduction to game theory

Here we will do a little sidestep and introduce another field which is closely related to linear programming. *Games* have been played for thousands of years, both simple dice games, Chess, Set, Poker and recently family games like Agricola and Bohnanza. But we are not restricted to games with a set of rules, also situations in real life, such as politics, economics and social relations might be modeled as games. Many of the problems solved as LPs or MIPs can actually be modeled as games. And games may be solved using linear programs.

Because game theory is a wide field, we will restrict the discussion to topics relevant to scheduling problems.

Thus we will only look at *finite two-player zero-sum sequential games*. These are games for two players where the payback for the first player is the loss of the other and vice versa (zero-sum). Furthermore we assume that the number of choices available for both players is finite. In a sequential game (as opposed to *simultaneous* games), the second player knows the action taken by the first player.

The players can choose between different strategies. Let us denote the strategies for the first player by $x \in X$ and for the second player by $y \in Y$. The payback (for the first player) is a result of the different strategies, and hence it can be denoted by a function $f(x, y)$. The payback for the second player is $-f(x, y)$ as the game is zero-sum.

The goal of the first player is to choose a strategy as to maximize his outcome. However, after having seen his choice, the second player will try to maximize her outcome, in other words, minimize his. If both play optimally, we get the following *value* of the game:

$$\zeta = \max_{x \in X} \min_{y \in Y} f(x, y)$$

1.6.1 Dominant strategies

A *dominated strategy* x_d is a strategy that will never be optimal, because it is worse than another *dominating* strategy x_D. In other words, $f(x_d, y) \leq f(x_D, y) \quad \forall y \in Y$. Similarly, we can talk about dominated and dominating strategies y_d and y_D for the second player, where $f(x, y_d) \geq f(x, y_D) \quad \forall x \in X$

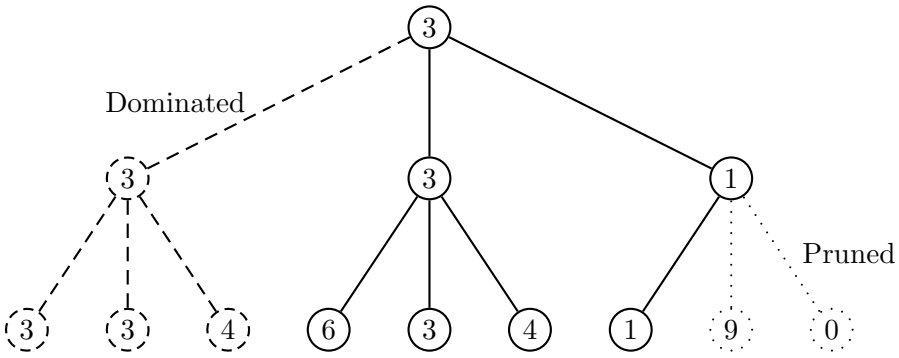
Identifying and removing dominated strategies is very important, because it reduces the problem size, which makes it easier to solve, without affecting the value of the game. The process of removing dominated strategies may be repeated until the game contains no more dominated strategies.

1.6.2 α-β-pruning

In addition to using dominated strategies, we can use a technique called α-β-pruning. In short, we stop checking when we know that we cannot improve our solution. We have an example in figure (2). Remember that the first player wants to maximize and the second wants to minimize. The numbers are the value for the first player.

The left, dashed, branch is dominated by the middle branch, and should be removed from consideration if we know this beforehand. After we have searched through the middle branch, we have a lower bound of 3. When we search the right branch, we quickly encounter 1, which gives us an upper bound of 1 for the right branch. We can prune the rest of the branch, as the first player will never enter it; the middle will always be a better choice. The dotted nodes are never considered and we see that the

Figure 2: Example of dominated strategies and pruning



actual value of the left branch should be 0, not 1. But it does not change the value of the root node anyway, and need not be considered.

Note that after removing the left branch from the tree, we now see that the second player has a dominating strategy which it did not have before. The right branch (with 4 and 0) is always better than the left branch (with 6 and 1).

1.6.3 Upper and lower bounds

If a sequential game is too large or complex to be solved efficiently, we may solve simpler problems to get upper and lower bounds. To get an upper bound, we can either restrict the second player or relax the first player (relaxation: see section 1.4.1). In the first case, we only consider a subset $\hat{Y} \subset Y$.

$$\zeta_{\hat{Y}} = \max_{x \in X} \min_{y \in \hat{Y}} f(x, y)$$

As we minimize over a smaller set, we get an upper bound: $\zeta_{\hat{Y}} \geq \zeta$. We can also relax X so that we maximize over a larger set. This might seem counterintuitive, to increase the problem size. It can, nonetheless, make the problem easier to solve if we can use another algorithm. For instance, if we take the convex hull $\bar{X}$ of the points in X. Then $\bar{X} \supset X$, but we can now use linear programming whereas the other set could require integer programming. Also in this case we get an upper bound:

$$\zeta \leq \zeta_{\bar{X}} = \max_{x \in \bar{X}} \min_{y \in Y} f(x, y)$$

Similarly we get a lower bound if we only consider a subset $\hat{X} \subset X$ or a relaxation $\bar{Y} \supset Y$

$$\zeta \geq \zeta_{\hat{X}} = \max_{x \in \hat{X}} \min_{y \in Y} f(x, y)$$

$$\zeta \geq \zeta_{\bar{Y}} = \max_{x \in X} \min_{y \in \bar{Y}} f(x, y)$$

These upper and lower bounds can be used for α-β-pruning, the I-A algorithm in section 1.8.1, and also to prove correctness of a solution; if we have found an upper and lower bound with the same value, we know that lower bound = optimal value = upper bound. If the gap between the bounds is small, we can estimate the maximum error of a solution and thus prove that a solution is approximately correct.

1.7 Bilevel programming

A natural extension to non-convex constraints (1.5) is bilevel programming (BLP). A bilevel program (BLP) is an optimization problem which itself contains another optimization problem. For a survey of bilevel programming, the reader may consult Dempe [14] or Colson et. al. [11].

In general, a bilevel program can be written

$$\begin{aligned} &\max_x F(x,y) \\ &\qquad \text{such that y solves this problem} \\ &\qquad \max_y f(x,y) \\ &\qquad g(x,y) \leq 0 \\ &x \in X, y \in Y \end{aligned}$$

The inner or lower problem, to find y, is also called the *followers* problem, while the outer or upper BLP is called the *leaders* problem.

Another way to write the inner constraints is

$$y \in \text{argmax}\{f(x,y) : g(x,y) \leq 0, y \in Y\}$$

F and f are the objective function for the leader and follower respectively, while g are constraints. X and Y are the domains from which we can choose x and y, and can also be viewed as constraints. Care should be taken when writing the constraints for the leader, in order not to restrict the follower.

BLP is used in many fields like politics, economics, management and engineering. It may, for instance, involve a decision maker who sets a price, makes a law or similar, and wants to maximize the outcome, assuming that all followers act rationally. This rationality is defined by the followers problem.

1.7.1 Solving BLP by strong duality

If we could formulate a BLP using only *one* LP (or MIP), it might be much easier to solve, as we can use standard techniques and computer programs. This encourages us to represent the inner problem as constraints to the outer problem.

Now suppose the inner problem is a standard LP. Then it is easy to find the dual of the inner problem. Assuming that both the inner problem and its dual are feasible, then we know by strong duality (theorem 1.4) that the problems must have the same optimal value.

The new problem is then:

maximize	outer objective function
subject to	outer constraints
	inner constraints
	inner dual constraints
	inner objective value = inner dual objective value

Even if this formulation looks innocent, it may be very hard to even find feasible solutions. The reason is that the last 3 constraints require you to actually solve an LP program to optimality. Moreover, if the inner problem is not a standard LP program, it might not have any solution at all, as the strong duality theorem is not valid for integer programs for instance. Numerical errors might cause the gap to be strictly positive.

1.8 Dynamic simplex method

The *dynamic simplex method* (DS) [13, section 1.2] is a way of solving linear programs, when the size of the constraint matrix is too large to be written explicity, typically because there are too many constraints. We may then use a *cutting plane* algorithm, by starting with a smaller problem and generate constraints on the fly as they are needed.

Suppose the problem is

$$\begin{aligned} \text{maximize: } & c^T x \\ \text{subject to: } & Ax \leq b \end{aligned}$$

where A and b have extremely many rows. However, we do not know which of the rows that are required when we start solving the problem.

We can rewrite the constraints using set notation: $x \in P$ where $P = \{x \in \mathbb{R}^N : Ax \leq b\}$. We want to solve this problem by starting with a relaxation on P, and add new constraints, or cuts, when needed, until we have found an optimal solution. Such cuts were first introduced by Gomory, and later described by Bender [5].

A *separation oracle* is an algorithm which either concludes that a point $\bar{x} \in \mathbb{R}^N$ satisfies all the constraints $A\bar{x} \leq b$, in other words, that $\bar{x} \in P$. Or if not, returns any constraint that is violated: $\sum_j a_{ij}\bar{x}_j > b_i$.

The separation oracle is independent of the rest of the method, and may use any algorithm to find the answer.

We may now combine the ideas to form an algorithm:

Algorithm 1.7 (dynamic simplex)**.**
Problem: Find optimal solution to $\{\max c^T x : Ax \leq b, x \in \mathbb{R}^N\}$
Output: Optimal solution x^*
Initialization: We start with a some of the rows in A and b defined by an index set I, and define $P_1 = \{x : A_I x \leq b_I, x \in \mathbb{R}^N\}$. Because we use a subset of the constraints, $P_1 \supset P$

1. **Solve:** Find an optimal solution $\bar{x}$ to the linar program $\{\max c^T x : x \in P_i\}$.
2. **Test:** Then use the separation oracle to check if $\bar{x} \in P$. If it is, we are done. Otherwise, the oracle will return a violated constraint $\sum_j a_{ij}\bar{x}_j > b_i$.
3. **Cut:** Define $P_{i+1} = \{x \in P_i : \sum_j a_{ij}\bar{x}_j \leq b\}$. In other words, we add a new row from the constraints in order to remove the unwanted solution $\bar{x}$ and hopefully others. Because we add a new constraint separating $\bar{x}$, we have that $P_i \supset P_{i+1} \supset P$. Now we may solve the problem again with the new set P_{i+1}

1.8.1 Implementor-adversary approach

A special case of the dynamic simplex method is the implementor-adversary approach (I-A), which was successfully applied on portfolio optimization problems by Bienstock [7, 8].

Suppose we have a problem that can be written like a sequential game:

$$\text{find } \zeta^* = \min_{x \in X} \max_{y \in Y} f(x, y)$$

Even though not required, we will suppose f is linear in x and y, as this is sufficient for our purposes. Sequential zero-sum games is a special class of bilevel programs, where

the objective functions for the leader and follower are the same, just negated. It is easy to see that we can also rewrite the problem as follows:

$$\text{find } \zeta^* = \min_{x \in X, y^* \in Y} f(x, y^*) \tag{10}$$

$$f(x, y^*) \geq f(x, y) \qquad \forall y \in Y \tag{11}$$

The interpretation of (11) is that given a solution x for the first player, the best response for the second player is y^*, in other words $f(x, y^*) = \max_{y \in Y} f(x, y)$.

This problem can now be solved using the DS algorithm (1.7): we replace Y with a subset $\tilde{Y}$ and solve this smaller problem. Then we extend $\tilde{Y}$ to cut away unwanted solutions. The difference from standard DS is that instead of adding constraints to remove infeasible solutions, I-A add constraints *and* variables. We have one variable for each $x \in X$ and each $y \in \tilde{Y}$, and $\tilde{Y}$ is extended during the algorithm.

Because I-A has as special structure of being a sequential zero-sum game, and we may exploit this fact to make more efficient algorithms, by using techniques from section 1.6.

Throughout the algorithm we maintain upper and lower bounds on the optimal value ζ^*, which can be used for early termination of the algorithm with approximate solutions. These bounds can be found using theory from section 1.6.3. This is an important difference from the general dynamic simplex method, which only have upper bounds (because we have interchanged minimization and maximization, this corresponds to the lower bound in I-A).

Algorithm 1.8 (Implementor-adversary)**.**
Problem: Find $\zeta^* = \min_{x \in X} \max_{y \in Y} f(x, y)$
Output: optimal solution $x^* \in X$ such that $\zeta^* = max_{y \in Y} f(x^*, y)$
Initialization: $\tilde{Y} = \{y_0\}$, lower bound $L = -\infty$, upper bound $U = \infty$

1. **Implementor problem (Solve):** Solve $\min_{x \in X} \max_{y \in \tilde{Y}} f(x, y)$ with solution x^*. Set lower bound $L = \max_{y \in \tilde{Y}} f(x^*, y)$

2. **Adversarial problem (Test):** Solve $\max_{y \in Y_K} f(x^*, y)$ with solution y^*. Update upper bound by setting $L = min(U, f(x^*, y^*))$

3. **Test (Cut):** If $U - L$ is small enough, less than a chosen ϵ (may be 0), go to the final step. Else we add a new cut so as to define $\tilde{Y} = \tilde{Y} \cup y^*$ and go to step 1.

4. **Final step:** Use any solution x^* where the corresponding adversarial value was at a minimum: $\max_{y \in Y_K} f(x^*, y) = U$.

Worth noting is that, while the optimal solution to the implementor problem is strictly non-decreasing (because $\tilde{Y} \subset \tilde{Y} \cup y^*$), we have no convergence on the adversarial problem, except that it must converge when x^* is globally optimal. This can be demonstrated by an example:

Example 1.9. Let $Z = \{z \in [0, 4]^3 : z \in \mathbb{Z}, \sum z = 6\}$. Suppose you must solve the following problem:

$$\min_{x \in Z} \max_{y \in Z} \zeta = \sum \max(y - x, 0)$$

Instead of considering all 19 elements in Z, we decide that we will use the I-A algorithm. As an initialization step, we set $\tilde{Y} = y_0 = \{0, 0, 0\}$

In the first step in the first round, we could get the solution $x^* = \{2, 2, 2\}$. In fact, any solution would have $\zeta = 0$. We update the lower bound $L = 0$.

The adversary will use the solution x^* found by the implementor to generate an optimal solution $y^* = \{4, 1, 1\}$ with $\zeta = 2$ to the adversarial problem. Note that there exist 9 solutions which are all optimal, and we can choose any. We update $\tilde{Y} = \{\{0, 0, 0\}, \{4, 1, 1\}\}$ and $U = 2$.

In the second round, we get $x^* = \{4, 1, 1\}, \zeta = 0$ with no update to the lower bound. Now there are several solutions to the adversarial with $\zeta = 4$, for instance $y^* = \{0, 2, 4\}$

$\tilde{Y} = \{\{0, 0, 0\}, \{4, 1, 1\}, \{0, 2, 4\}\}$ while the upper bound stays at 2.

In the third round, one of the optimal solutions is $x^* = \{2, 1, 3\}, \zeta = 2$ where we will have to update the lower bound $L = 2$. We will finish as $L = U$. We can use any solution where the following adversarial objective value was optimal, in this example we must choose the first solution $x^* = \{2, 2, 2\}$. Note that the last solution $x^* = \{2, 1, 3\}$ is *not* optimal, as the adversarial solution $y^* = \{3, 3, 0\}$ has $\zeta = 3$.

Another curiousity is that with bad parameters, we may end up with dominated solutions (section 1.6.1). This can be demonstrated by a similar example:

Example 1.10. Let $Z = \{z \in [0, 5]^3 : z \in \mathbb{Z}, \sum z = 5\}$. Suppose you must solve the following problem:

$$\min_{x \in Z} \max_{y \in Z} \zeta = \sum \max(y - x, 0)$$

The optimal value is 4. This is because all the numbers in x cannot be 2 or higher. We may without loss of generality suppose this is the first number: $x = (1, a, b)$ where a and b are arbitrary integers. Then the response $y = (5, 0, 0)$ yields an objective value of 4, which is the best we can do, as $x = (0, a, b)$ is even worse with an objective value of 5.

However, the solution $x = (1, 1, 1)$ also yields an objective value of 4, even though it is dominated by $x = (1, 2, 2)$.

In general we would like to remove dominated strategies, as these can never improve the worst case, and will also worsen the average case. It need to be specified in the model though.

1.8.2 Correctness

We may use section 1.4.1 to argue that the solutions found during iteration of I-A are actually upper bounds. Thus when we find a feasible solution, then this must also be optimal.

By section 1.6.3, the solutions to the implementor and adversarial problem are upper and lower bounds respectively to the real objective value ζ^*. We can never jump past the optimal objective value, as it would contradict the fact that the solution values are lower and upper bounds. If $U = L$, we thus know that $\zeta^* = U = L$. Furthermore, we also have an optimal solution from our implementor step.

A similar argument holds for the dynamic simplex method in section 1.8, where intermediate solutions $\bar{x}$ yield upper bounds and the last, feasible solution, yields a lower bound.

We may accept an approximate solution x with $\zeta = f(x, y) = L \leq \zeta^* \leq U$, in which case we have limited the deviation from the optimal value to $U - L$.

Thus we have proved that if the algorithm stops, it must either have found an optimal solution, or a sufficiently good solution. What remains is to prove that the algorithm eventually stops.

1.8.3 Termination

To prove termination on I-A, we must prove that the algorithm stops after a finite number of steps.

We will assume that Y is finite. Since we always add an element y^* to $\tilde{Y}$, eventually $Y = \tilde{Y}$, and the implementor will solve $\min_{x \in X} \max_{y \in Y} \ f(x, y)$, which by definition has the optimal value ζ^*.

We must still prove that we always add a *new* element to $\tilde{Y}$. Assume that the optimal solution to the implementor problem is x^* and that the optimal solution to the adversarial problem is y^* with $y^* \in \tilde{Y}$. Then:

$$
\begin{aligned}
U &\leq \max_{y \in Y} \ f(x^*, y) \\
&= f(x^*, y^*) && \text{by assumption} \\
&\leq \max_{y \in \tilde{Y}} \ f(x^*, y) && \{y^*\} \subset \tilde{Y} \\
&= \min_{x \in X} \max_{y \in \tilde{Y}} \ f(x, y) && \text{by assumption} \\
&\leq L
\end{aligned}
$$

and we have that x^* must indeed be a global optimal solution.

Thus the algorithm stops after at most $|Y|$ steps and gives a correct result. If we have linearity, then by section 1.8.4, we only need to consider vertices of the convex hull of Y, which can make the algorithm considerably more efficient. This is true, also when Y is a polytope with an infinite number of interior points.

The dynamic simplex method converges if the number of constraints is finite. However, if we have a quadratic objective function and infinitely many constraints, the dynamic simplex method may never terminate, as this example shows:

$$\{\max ||x|| : ||x|| \leq 1\}$$

. It is impossible to define the unit circle as a finite intersection of half-planes, thus you can always find a solution $\bar{x}$ with $||\bar{x}|| > 1$.

1.8.4 Code optimizations

By *algorithm* or *code optimizations* we mean modifications to an algorithm or a computer program to make it work more efficiently or use fewer resources, not solutions to an optimization problem.

Even though we have no guarantee that the iterative cutting planes algorithm converges fast, experiments indicate that the number of iterations needed is low. Still there are several measures we may take to speed up convergence and also running time.

First we note that the adversary only needs *any* solution that proves that the implementor solution is suboptimal. This way we can simplify the adversary and save running time. Note that in the last step, when the implementor has found a global optimal solution, the adversary must find an optimal solution and also prove that it is optimal.

Although we only need any solution from the adversary, we should really consider finding a good solution. Suppose that an element $\bar{y} \in Y$ is a convex combination of

other points in Y:

$$\bar{y} = \sum_{y \in Y} \lambda_y y$$
$$\sum_{y \in Y} \lambda_y = 1$$
$$\lambda_y \geq 0$$

Then, by linearity, we have that

$$f(x, \bar{y}) = f(x, \sum_{y \in Y} \lambda_y y) = \sum_{y \in Y} \lambda_y f(x, y)$$

The above equivalence also shows that we do not actually need to generate all points in Y, but we can restrict ourselves to the vertices of the convex hull of Y.

However, if the bottleneck is in the implementor problem, there are several possible algorithm optimizations. We may reduce the number of iterations if we start with several points in $\tilde{Y}$ rather than one or none. The condition is that the points are chosen carefully using a good heuristic. The drawback is increased complexity in the implementor problem because of the added constraints.

Another code optimization is warm starting. We may exploit the fact that one implementor problem instance is very similar to the next one, as we only add one constraint. The previous solution can be used as a starting point for the current problem, instead of solving it from scratch. This is already available in several commercial solvers.

Similar to warm starting is head starting. This is when we instead of using the initialization set $\tilde{Y} = y_0$, create our own set of adversarial solutions which can be added to $\tilde{Y}$. This set can be created using heuristics, but we can also use a relaxed version of the implementor program for this purpose.

1.9 Robust optimization

So far, the constraints have been written in stone. We have one constraint matrix A with bounds b and a coefficient vector c, all given as input. In real life, however, numbers are not always that certain. Moreover, small changes in input might worsen the objective value drastically, or even make our solution infeasible. It might be possible to make small changes to our solution to make it feasible, but it also might not. Fischetti and Monaci [15] describe many of the methods used to handle uncertainty.

Robust optimization (RO) takes the uncertainty into account when solving the problem by requiring that a solution is feasible for any constraint matrix $A \in \mathcal{A}$ [1]. RO was first described by Soyster [22] who devised a scheme with column-wise uncertainty. Each column A_j in the constraint matrix A was allowed to take any value in a convex set $\mathcal{K}_j$.

$$\min c^T x$$
$$\text{subject to: } \sum_j A_j x_j \leq b, \ \forall A_j \in \mathcal{K}_j$$

[1] uncertainty in coefficients and bounds can be modeled by additional variables and constraints in A

However, as there are no restrictions on uncertainty in rows, all constraints in any can take their worst-case value at the same time. Because a solution is bounded by the worst case constraint in any row, it is bounded by the worst case in *all* rows. [22]

$$\begin{aligned} &\min c^T x \\ &\text{subject to: } \sum_j a_{ij}^* x_j \leq b_i,\ \forall i,\ a_{ij}^* = \sup_{A_j \in \mathcal{K}_j} (A_j)_i \end{aligned}$$

This may easily lead to overconservative solutions with optimal values far from the nominal solutions, if we find feasible solutions at all.

To solve a RO problem, we just assume the worst in all rows and solve it normally using any LP solver.

1.9.1 Ellipsoidal robustness

Robust optimization is indeed robust; any feasible solution remains feasible, even for the worst case scenario. Still we might be interested in relaxing the robustness in order to improve the optimal value. As an alternative, Ben-Tal et al. [3, 4] suggested using ellipsoidal uncertainties. Because the euclidean norm is used, this approach lead to quadratic programs (QP) with constraints on the form

$$a_i^T x + \alpha_i \geq \| B_i x + b_i \|, \quad i = 1, \ldots, M$$

where α_i are fixed reals, a_i and b_i are fixed vectors, and B_i are fixed matrices of proper dimentions. $\| \cdot \|$ is the normal euclidean norm, leading to quadratic terms.

The quadratic terms make the problem more difficult to solve, but state-of-the-art LP solvers are able to solve QP efficiently using interior-point methods.

1.9.2 Bertsimas and Sim

Later, Bertsimas and Sim (BS) [6] suggested another approach which protects against scenarios where at most Γ_i coefficients in row i are allowed to deviate at the same time, while the others stay at their nominal value. The approach can be formulated:

$$\begin{aligned} &\min c^T x + \max_{\{S_0 : |S_0| \leq \Gamma_0\}} \left\{ \sum_{j \in S_0} \hat{c}_j x_j \right\} \\ &\text{subject to: } \sum_j a_{ij} x_j + \max_{\{S_i : |S_i| \leq \Gamma_i\}} \left\{ \sum_{j \in S_i} \hat{a}_{ij} x_j \right\} \leq b_i \quad \forall i \end{aligned}$$

where c and a_{ij} are nominal values, and $c + \hat{c}$ and $a_{ij} + \hat{a}_{ij}$ are the maximum values, and S_i is the set of variables allowed to deviate from the nominal values.

The resulting solutions are always feasible if the assumptions we made above are correct, but even if more than Γ_i variables change in row i, the solution is feasible with a high probability.

The approach works, even if Γ_i is not integer, in that case the last constraint is only a little uncertain corresponding to the fractional part of Γ_i. It also works with MIP, i. e. when some or all the x have integrality constraints.

This problem can be solved as a linear program using strong duality, as described in section (1.7.1) and [6], by adding extra constraints and variables.

1.9.3 Light robustness

As even the BS approach might be too conservative, Fischetti and Monaci [15] suggested another approach. The basic *light robustness approach* is an extension to BS which replaces hard with soft constraints. When a soft constraint is violated, it results in a penalty in the objective function. The new objective is to minimize the sum of these penalties.

$$\min \sum \gamma_i \tag{12}$$

$$\text{subject to: } \sum_j a_{ij}x_j + \max_{\{S_i : |S_i| \leq \Gamma_i\}} \left\{ \sum_{j \in S_i} \hat{a}_{ij}x_j \right\} - \gamma_i \leq b_i \qquad \forall i \tag{13}$$

$$\sum_j a_{ij}x_j \leq b_i \qquad \forall i \tag{14}$$

$$c^T x \leq (1+\delta)z^* \tag{15}$$

The original bounds in the nominal problem are hard constraints (14), while in the robust inequalities (13), the excess can be absorbed by slack variables γ_i at a penalty in (12). To ensure a good optimal value, we require that the new objective value is not much worse than the nominal objective value z^* (15). δ is a constant which influences the degree of robustness.

2 Model

In this section we will describe a mathematical model of a practical problem, namely master surgery scheduling (MSS).

The master surgery schedule (MSS) is a cyclic timetable that defines the number and type of operation rooms available, which hours the rooms will be open, and which surgical groups or surgeons are to be given priority for the operating room time. For instance, orthopedic may have operation room 2 for the block defined as every other Tuesday between 10:00 and 16:00. The schedule is cyclic and repeats itself after a certain time (e.g. every 2 weeks), and is used as a template by the admission planner assigning patients to blocks.

First we will give a background to the different aspects of surgical planning, with notation, then we will describe the problem, both with constraints and objectives, discuss how we can add robustness to the model, before we put it all together into a mathematical problem statement.

2.1 Introduction

At a hospital, you typically have different *surgical groups* like orthopedics or gynecology, and each of them has the ability to treat a set of patients. There are also nurses, anesthesiologists and other professions, which may or may not be considered a part of the group.

The surgeons work on *patients*, which are grouped according to the surgery needed, but may also be grouped according to *difficulty*, i. e. how much time is needed to treat each patient, measured in hours.

In addition to personnel, you also have *resources* like, for instance, operation *rooms* and *equipment*.

The patients are in need of *elective* and *emergency* surgery. The former is planned, while the latter are operations which have to be done within 24 hours, in many cases within fewer hours. Emergency surgery can disrupt the surgery plan, as it is prioritized above elective cases, but on the other hand, we do not want to allocate too many resources to emergency surgery because of inefficiency. Demand for emergency surgery is highly unpredictable and usually not included in any surgery planning, though hospitals might allocate spare resources to be used for emergencies.

To ease short time planning, hospitals try to maintain *queues* in all groups. This way, there are always patients to fill in the schedule, even leading to increased efficiency if the number of is lower than expected. Long queues, however, lead to problems, both for the patients, who risk adverse effects, and to the hospitals, who risk being penalized. Many hospitals operate with *internal* thresholds. Any queue within this limit is considered acceptable. Exceeding the internal threshold will come to the attention of the management, but has few consequences. Violating the *external* threshold, however, will typically imply a penalty on the hospital, and is strongly unwanted. Hospitals can manage queue length by redistributing the available resources where they are needed most, but, in the best of cases, this is done infrequently.

In order to manage queue lengths, we need to consider the *demand*. There might be long term changes in the demand, like less lung cancer due to less smoking, but also seasonal changes, like more broken hips during winter. Seasonal changes may be absorbed in the full year average, but long term changes should be reflected in a new plan.

Not only does the expected demand vary. The actual demand may deviate from the expected demand due to epidemics, accidents, or simply by random fluctuations. This *uncertainty* can make a good plan considerably worse. However, if we give up optimality and aim a little lower, we may absorb these deviations and improve the worst-case situations. Such solutions are called *robust*.

The plan should not only consider the patients at the hospital, but also the personnel. In particular, the use of *overtime* should be minimized, and there might be other objectives too.

2.2 Notation

The set of surgical *groups* is denoted by G. The set of *rooms* is denoted by R. MSS spans over a set of *days* D and *weeks* W. Each day is divided into *blocks*, denoted by B with a given starting and ending hour. We can group the blocks according to the *block lengths* L, to form $B(l), l \in L$. The period from 10:00-12:00 on Tuesday in week 2 is an example of a block of length 1 (each time slot lasts for of 2 hours).

The set of overlapping blocks is denoted by E. Two blocks are overlapping if the day and week are equal, and if the time periods are overlapping at least one point $(b_1, b_2) \in E \Leftrightarrow b_1, b_2 \in B$ and $b_1 \cap b_2 \neq \emptyset$

A choice of $g \in G, r \in R, b \in B$ is called a *pattern*, and the set of patterns is denoted by P. With $P(g)$ we denote the set of patterns with a fixed surgical group g. Similarily $P(r, l)$ is the set of patterns with a given room r and block length l.

The *demand* for each group and length is denoted by y. It can either be given as input to the problem, or it can be a decision variable, depending on the problem.

If the demand is not met, we can talk about *excess demand* or *queues* q. Queues can also be split in *internal* and *external* queues, the external queue being used only if the total queue is exceeding the threshold Ext.

2.3 Master surgery scheduling problems

The MSS problem is to create an optimal schedule. We need to assign blocks and rooms to the different groups in order to fulfill the wishes of the hospital.

But there are several ways of modeling MSS. First you need a way of representing your schedule, then you can consider all the different constraints and objective functions.

2.3.1 Representing schedules

There exist several alternative representations of schedules, a natural choice is, for instance, the *time slot formulation*, where you introduce a binary variable $x_{g,r,t}$ for each group g, room r and time slot t which is 1 if and only if r is assigned to g at time t. This formulation presents a number of difficulties when trying to represent slot contiguity, which is important in case of difficult patients requiring several time slots.

We will instead use the *pattern formulation*, where for each pattern $p = (g, r, b) \in P$, we introduce a binary variable x_p. The difference from the time slot formulation is that we use blocks instead of time slots, which enable us to assign longer, continuous blocks. With this formulation, one pattern corresponds to one patient.

2.3.2 Representing demand

The demand y will be grouped according to surgery g and the length l needed. Thus $y_{g,l}$ denotes the expected number of patients who need l time slots of treatment from surgical group g. In other words, $\sum_l l \cdot y_{g,l}$ is the total number of time slots required by group g during the time horizon.

2.3.3 Constraints

We have three basic constraints: There cannot be more than one group in a room at a given time.

$$\sum_{p_1 \in P(r,b_1)} x_{p_1} + \sum_{p_2 \in P(r,b_2)} x_{p_2} \leq 1 \quad \forall r \in R, (b_1, b_2) \in E$$

Similarly, a group cannot be in more than one place at the same time.

$$\sum_{p_1 \in P(g,b_1)} x_{p_1} + \sum_{p_2 \in P(g,b_2)} x_{p_2} \leq 1 \quad \forall g \in G, (b_1, b_2) \in E$$

And finally, you cannot split rooms and groups, thus x should be a binary variable.

These are the only constraints used in the thesis, but it is easy to add new ones when required. If we want to fix some patterns in the schedule, we just set the corresponding x to 1. Likewise, if we know that some patterns are illegal, for instance, if a team or room is unavailable at a certain time, or that a team and a room are incompatible, this is also easy. We just set the corresponding x to zero.

More complicated constraints are also possible, as long as we can express them as a linear combination of x. The problem does not necessarily become more complicated either. With more constraints, we should expect the problem to be solved faster, as the feasible region is smaller.

2.3.4 Objective function

The objective in *Minimum Queue MSS* (MQ-MSS) is to assign patterns such that the queues are as short as possible. The queue $q_{g,l}$ is defined as the difference between the demand $y_{g,l}$ and the number of assigned blocks $\sum_{p \in P(g,l)} x_p$. In other words: the number of unscheduled patients. In particular the queues should not exceed a certain threshold Ext. For this we use an objective function designed to increasingly penalize longer queues.

Definition 2.1 (objective function in MQ-MSS)**.**

$$f(q) = \sum_{g \in G} \sum_{l \in L} f_{g,l}(q_{g,l})$$

where $f_{g,l}$ is a non-decreasing, convex, piece-wise linear function defined as

$$f_{g,l}(q) = \begin{cases} 0 & q < 0 \\ l \cdot c_0 \cdot q & 0 \leq q < Ext \\ l \cdot c_0 \cdot Ext + l \cdot c_1(q - Ext) & Ext \leq q \end{cases}$$

Thus the marginal cost for the queues is c_0 for the first Ext units, and c_1 for the following queue units. Everything is multiplied with the difficulty, so as not to discriminate the difficult patients.

Further theory on piece-wise linear functions can be found in section 1.5.

In *Minimum overtime MSS*, we fix the maximum queue length and ask ourselves, what is the minimum amount of overtime that we need? More specifically, every pattern now has a non-negative cost c_p, and the objective is to minimize the total cost $c^T x$. The expensive patterns, using blocks with much overtime, would only be assigned if it were really needed.

In addition to the normal constraints, it is also required that the queue is shorter than a given threshold: $q_{g,l} \leq Ext$ $[Ext_{g,l}]$

A hybrid model is also possible, where the objective function penalizes both long queues and the use of overtime. The different objectives must then be given weights relative to their importance.

2.4 Robustness

In standard robust optimization, as described in section 1.9, we add uncertainty sets to all variables in the LP or MIP. In this problem, however, most variables are structural and never uncertain. Either a group is present somewhere, or it is not. Therefore only the demand will be uncertain.

A classical assumption is that each demand $y_{g,l}$ belongs to an interval $[b_{g,l}, \beta_{g,l}]$. The lower bound b is sometimes also denoted as the *nominal value*. However, this assumption is much too pessimistic. Indeed, it is a common experience that, even though the demand of a specific type of operation can actually take any value in the interval, all demands will not assume their upper bounds simultaneously.

We could model the demand as random variables with a given statistical solution, and use the information we have on the distributions to find a solution with an optimal expected value. This is denoted by stochastic programming, and is outside the scope of this thesis. The readers may consult [9].

Instead, we assume that the overall demand of surgery operation hours is limited by some upper bound K. This gives us the following set from which we can draw demand vectors.

$$Y_K = \{y : y_{g,l} \in [b_{g,l}, \beta_{g,l}], y_{g,l} \in \mathbb{Z}, \sum_{g \in G} \sum_{l \in L} l \cdot y_{gl} \leq K]\}$$

The demand should be integral, because there will always be an integer number of patients coming to the hospital.

The number K will also be called a *knapsack constraint*, due to the similarity to the knapsack problem (Example 1.5). The reason for the factor l is that we want to find balanced solutions by limiting the total number of hours. Without the factor, when we only limit the number of patients, the solutions are skewed towards the demanding groups, because these are the worst ones that the adversary can choose.

The knapsack constraint is inspired by ellipsoidal robustness, (section 1.9.1), only here we are using the L_1 manhattan norm instead of the L_2 euclidean norm, to avoid quadratic programs.

It could also be said that it is along the lines of Bersimas and Sim (section 1.9.2), but where we limit the total demand instead of the number of variables which are allowed to change.

We have also used an idea from light robustness (section 1.9.3), that we do not require all demands to be met, rather we operate with soft constraints. Excess demand can be absorbed in queues, but at a cost.

If we, on the other hand, drop the knapsack problem, we are left with the set

$$Y = \{y : y_{g,l} \in [b_{g,l}, \beta_{g,l}] \quad \forall g \in G, l \in L\}$$

Y is a hyperrectangle or an orthotope, where the demand for each group is independent from that of all the other groups, and can take any value in an interval. The idea that uncertainty variables are independent of each other is what characterizes the original robust optimization. As we see in the discussion in section 1.9, robust optimization often leads to overconservative solutions. Indeed, we see that the worst case demand in our problem is simply $y = \beta$, and all other demand vectors are dominated by this worst case.

Still, because of the simple model that follows, robust optimization is widely used, and the set Y will be used in the reference solution which is used for comparing different models (section 3.4).

2.5 Mathematical problem statement

Now that we have defined all parts of the problem, we can write the complete mathematical model.

Definition 2.2 (Minimum queue-MSS)**.**

$$\min_x f(q) \tag{16}$$

$$\text{subject to:} \sum_{p_1 \in P(r,b_1)} x_{p_1} + \sum_{p_2 \in P(r,b_2)} x_{p_2} \leq 1 \qquad \forall r \in R, (b_1, b_2) \in E \tag{17}$$

$$\sum_{p_1 \in P(g,b_1)} x_{p_1} + \sum_{p_2 \in P(g,b_2)} x_{p_2} \leq 1 \qquad \forall g \in G, (b_1, b_2) \in E \tag{18}$$

$$y \in \text{argmax}\{f(q) : y \in Y_K\} \tag{19}$$

$$q_{g,l} = y_{g,l} - \sum_{p \in P(g,l)} x_p \qquad \forall g \in G, l \in L \tag{20}$$

$$x_p \in \{0, 1\} \qquad \forall p \in P \tag{21}$$

The problem is bilevel, as it involves the solution of a "second level" optimization problem (19), which says that the demand should be chosen as bad as possible with respect to the assignment x. Constraint (17) and (18) from section 2.3.3 ensures that the rooms and groups are not assigned twice for 2 overlapping blocks. The queues (20) and objective function (16) are as in section 2.3.4. Constraint (21) ensures that the decision variables are binary.

In case of the reference solution, we replace constraint (19) with $y = \beta$.

3 Algorithm

While the model in section 2 is mathematically correct, it is not designed for being solved by a computer. In this section, we will write an algorithm based on the mathematical model, which will later be implemented and solved using computer programs.

First we will discuss how to implement the different parts of the model, before we put everything together to create a complete algorithm. At the end we will also talk about code optimizations.

3.1 Introduction

The problem statement in section 2.5 is written from a mathematician's point of view. It is possible to solve by enumerating all possible combinations of the variables x and y, and choose the combination which best satisfies all the constraints. But by the time this algorithm had solved a moderately large test set, we would all have been long dead. We could even run out of memory, just by listing all combinations of y. Instead we see that we are dealing with a bilevel program which is described in section 1.7.

Due to a non-standard objective function, we cannot use strong duality as in section 1.7.1. Instead we will use the implementor-adversary approach described in section 1.8.1.

3.2 Transforming the model

The formulation used in the implementor-adversary algorithm (I-A) is

Definition 3.1 (Implementor-adversary formulation)**.**

$$\text{find } \zeta^* = \min_{x \in X} \max_{y \in Y} \; f(x, y)$$

while the model is formulated in definition (2.2).

In order to use I-A, we need to transform the model to the format in definition (3.1)

First we need to change the objective function. As the queues q are functions of the demand y, and the assignments x from (20), we may simply replace $f(q)$ with the equivalent function $f(x, y)$ and drop q from the model, as it is not used elsewhere. Furthermore, constraint (19) may be moved to the objective function, as they contain the same function f.

This gives us the following objective:

$$\min_{x} \max_{y \in Y_K} \; f(x, y)$$

If we write the remaining constraints (17), (18) and (21) as a new constraint $x \in X$, we have exactly the form as in definition (3.1).

3.3 Implementing implementor-adversary

We recall from section 1.8.1 that the I-A algorithm is written

Algorithm 3.2 (implementor-adversary algorithm)**.**
Problem: Find $\zeta^* = \min_{x \in X} \max_{y \in Y_K} f(x, y)$
Output: [near] optimal solution $x^* \in X$ such that $\zeta^* = max_{y \in Y_K} f(x^*, y) - [\epsilon]$
Initialization: $\tilde{Y} = \{b\}$, lower bound $L = -\infty$, upper bound $U = \infty$

1. **Implementor problem:** Solve $\min_{x \in X} \max_{y \in \tilde{Y}} \ f(x, y)$ with solution x^*. Set lower bound $L = \max_{y \in \tilde{Y}} f(x^*, y)$
2. **Adversarial problem:** Solve $\max_{y \in Y_K} \ f(x^*, y)$ with solution y^*. Update upper bound by setting $L = min(U, f(x^*, y^*))$
3. **Test:** If $U - L$ is small enough, go to the final step. Otherwise we add a new cut so as to define $\tilde{Y} = \tilde{Y} \cup y^*$ and go to step 1.
4. **Final step:** Use any solution x^* where the corresponding adversarial value was at a minimum: $\max_{y \in Y_K} \ f(x^*, y) = U$

We will proceed by describing the different parts in detail.

3.3.1 Implementor problem

The implementor problem is to solve $\min_{x \in X} \max_{y \in \tilde{Y}} \ f(x, y)$ From the model in section 2.5, we see that the implementor problem is large, as X is a large set. It is the bottleneck in our algorithm, and code optimizations are essential for improving running time.

In particular, we have a large number of binary variables (21) which slows down the program. The binary constraints are necessary to arrive at a valid solution, but if we relax integrality constraint we still get a valid lower bound. The relaxation should be considerable in order to simplify the problem sufficiently, but it should also be close enough to the original problem, to attain a good lower bound and a good response from the adversary. Also note that the upper bounds found may be wrong, because the adversary responds to relaxed, and thus invalid, solutions x^*. This is also a reason for the relaxation to be close to the original.

With this in mind, we will replace the integrality constraints on x with:

$$0 \leq x_p \leq 1 \qquad \forall p \in P \tag{22}$$

$$\sum_{p \in P(g,l)} x_p = s_{gl} \in \mathbb{Z} \qquad \forall g \in G, l \in L \tag{23}$$

The sum of all assignments for each group and difficulty is used in constraint (20) when calculating the queue, and thus also the objective function. Hence these sums should not be relaxed, that is, they should still be integral (23). The decision variables in (22) have no direct influence on neither the objective function, nor the adversary, and may be relaxed to allow fractional values, though the bounds (0 and 1) should still be the same.

As a consequence of the relaxation, we must also change the equations (17, 18) to avoid solutions like $x = (0.5, 0.5, 0.5)$ for 3 or more mutually incompatible patterns. Instead we redefine E, to form $E(b_i) = \{b_j | b_i \cap b_j \neq \emptyset\}$. If $b_i \in B(1)$ is a block consisting of only one time slot, then $E(b_i)$ is the set of all blocks containing this time slot.

$$\sum_{b \in E(b_1)} \sum_{p \in P(r,b)} x_p \leq 1 \qquad \forall r \in R, b_1 \in B(1) \tag{24}$$

$$\sum_{b \in E(b_1)} \sum_{p \in P(g,b)} x_p \leq 1 \qquad \forall g \in G, b_1 \in B(1) \tag{25}$$

Constraint (24) restricts the rooms, so that for each timeslot, it is used by at most 1 group. It may be used fractionally by several groups, but not adding up to more than 1. This constraint replaces (17).

Similarily, (25) replaces (18) to ensure that a group can not be in more than one place (added up), during any time slot.

This gives us the following mixed integer program:

$$\text{Minimize: } \zeta \tag{26}$$

$$\text{Subject to: } \sum_{b \in E(b_1)} \sum_{p \in P(r,b)} x_p \leq 1 \qquad \forall r \in R, b_1 \in B(1) \tag{27}$$

$$\sum_{b \in E(b_1)} \sum_{p \in P(g,b)} x_p \leq 1 \qquad \forall g \in G, b_1 \in B(1) \tag{28}$$

$$s_{gl} = \sum_{p \in P(g,l)} x_p \qquad \forall g \in G, l \in L \tag{29}$$

$$0 \leq x_p \leq 1 \qquad \forall p \in P \tag{30}$$

$$s_{gl} \in \mathbb{Z} \qquad \forall g \in G, l \in L \tag{31}$$

$$q_{0,y,gl} + q_{1,y,gl} + s_{gl} \geq y_{gl} \qquad \forall y \in \tilde{Y}, g \in G, l \in L \tag{32}$$

$$q_{0,y,gl} \geq 0 \qquad \forall y \in \tilde{Y}, g \in G, l \in L \tag{33}$$

$$q_{0,y,gl} \leq Ext \qquad \forall y \in \tilde{Y}, g \in G, l \in L \tag{34}$$

$$q_{1,y,gl} \geq 0 \qquad \forall y \in \tilde{Y}, g \in G, l \in L \tag{35}$$

$$\zeta \geq \sum_{g \in G} \sum_{l \in L} l \cdot (q_{0,y,gl} \cdot c_0 + q_{1,y,gl} \cdot c_1) \qquad \forall y \in \tilde{Y} \tag{36}$$

Constraints (27) and (28) are the same as (24) and (25), while constraints (29), (30) and (31) are the relaxations described in (22) and (23).

The variables $q_{0,y,gl}$ and $q_{1,y,gl}$ defined in constraint (32)-(35) are the resulting internal and external queues respectively, when the demand is y. Internal queues are between 0 and Ext (33)-(34), while the external queues have no restrictions, except being non-negative (35). These constraints corresponds to the intervals in the objective function in definition 2.1.

We also see that the sum in (36) is equivalent to the definition of $f(q)$ in definition 2.1.

ζ is at least as high as the objective value in (36) from any of the demand vectors in $\tilde{Y}$, and because we want to minimize ζ (26), it must be equal to the maximum objective value. In other words, $\zeta = \min_x \max_{y \in \tilde{Y}} f(x, y)$, as required.

Note that if the set of demand scenarios $\tilde{Y}$ that we consider is large, we get proportionally many variables and constraints in (32)-(36).

3.3.2 Adversarial problem

The purpose of the adversary is to find cutting planes which can be used to remove unwanted solutions on the x variables. Cuts are found by solving $\max_{y \in Y_K} f(x^*, y)$, where x^* is an optimal solution to the implementor problem.

As the set Y_K is relatively small, this is a fairly easy problem. We need to be careful though, as we are maximizing over the non-concave function f described in section 2.3.4.

We will use the same notation as in section 3.3.1. Let $q_{0,gl}$ denote the internal queue, $0 \leq q_{0,gl} \leq Ext$ and let $q_{1,gl}$ denote the external queue, only used if the internal queue is at its maximum.

To maximize over a non-concave function is difficult; we need to introduce binary variables (42) as described in section 1.5.

This gives us the following mixed integer program:

$$\text{maximize: } f(q) = \sum_{g \in G} \sum_{l \in L} (l \cdot c_0 \cdot q_{0,gl} + l \cdot c_1 \cdot q_{1,gl}) \tag{37}$$

$$\text{subject to: } y_{gl} - q_{0,gl} - q_{1,gl} + M z_{0,gl} \geq \sum_{p \in P(g,l)} x_p \qquad \forall g \in G, l \in L \tag{38}$$

$$q_{0,gl} + M z_{0,gl} \leq M \qquad \forall g \in G, l \in L \tag{39}$$

$$q_{1,gl} + M z_{1,gl} \leq M \qquad \forall g \in G, l \in L \tag{40}$$

$$q_{0,gl} + M z_{1,gl} \geq Ext \qquad \forall g \in G, l \in L \tag{41}$$

$$z_{0,gl}, z_{1,gl} \in \{0, 1\} \qquad \forall g \in G, l \in L \tag{42}$$

$$y_{gl} \geq b_{gl} \qquad \forall g \in G, l \in L \tag{43}$$

$$y_{gl} \leq \beta_{gl} \qquad \forall g \in G, l \in L \tag{44}$$

$$\sum_{g \in G} \sum_{l \in L} l \cdot y_{gl} \leq K \tag{45}$$

$$y_{gl} \in \mathbb{Z} \qquad \forall g \in G, l \in L \tag{46}$$

$$\tag{47}$$

The objective function (37) is just the same as in section 2.3.4. M is a sufficiently high constant. Thus constraint (39) is equivalent to $z_{0,gl} = 1 \Rightarrow q_{0,gl} = 0$ and (40) to $z_{1,gl} = 1 \Rightarrow q_{1,gl} = 0$. Constraint (41) is the same as $z_{1,gl} = 0 \Rightarrow q_{0,gl} \geq Ext$.

(40) and (41) ensures that before the external queues are assigned, the internal queues must be full ($= Ext$). Without this constraint, the program would assign the expensive external queues first, because they contribute more to the objective function.

Constraints (39) and (38) allow us to set the queue to zero ($z_{0,gl} = 1$), and thereby ignoring the normal constraint that demand = queues + assigned blocks. Thus we are not forced to choose demand $\geq$ assigned blocks, as we otherwise would have been. This is important, because the implementor should not impose restrictions on the adversary.

Finally, constraints (43)-(46) makes sure that the demand is in the set Y_K as defined in section 2.4.

3.3.3 Test and final step

After solving the implementor and adversarial problems, we have found a solution x^* and the response y^*. If the response proved the optimal value found by the implementor was incorrect, we would like to extend $\tilde{Y}$ by defining $\tilde{Y} = \tilde{Y} \cup y^*$. This way we cut away the unwanted solution x^*, and hopefully many more.

We know that the optimal value to an optimal solution must lie in the interval $[L, U]$, between the lower and upper bound. In particular it cannot be less than L. Thus if the response was not too bad in terms of optimal value (not too far from L), we might be satisfied with our solution x^*.

Or if we improved the lower bound, we might be satisfied with an earlier solution x, which had a corresponding adversarial optimal value not too far from L. We can then return that solution which we now know was not too far from the real optimal value anyway.

3.3.4 Final iterations

When the algorithm (3.2) stops, a [near] optimal solution to the relaxed problem has been found. Most probable this solution is not valid to the non-relaxed problem, and we need to search further. By relaxing, the optimal value to the implementor problem never gets worse. Thus we can keep the lower bound, and we can also keep the set $\tilde{Y}$, which are all valid demand vectors. However, we must discard the upper bound.

We can now add the missing constraint in the implementor problem

$$x_p \in \{0, 1\} \quad \forall p \in P$$

and solve it normally, using the same algorithm (3.2). As we may keep the lower bound L, and also the set $\tilde{Y}$, we expect the problem to be solved fast, because the relaxed problem is not very different from the non-relaxed.

3.3.5 Head start

Instead of starting with $\tilde{Y} = \{b\}$, we may create a larger starting set using a heuristic. This will save iterations in the cutting-plane algorithm, but it will also make the implementor problem larger and thus more complicated.

A starting set should try to cover as many extreme cases as possible, because an adversary solution will typically be extreme. Because of the high cost of long queues in the objective function, an adversary will try to max out demand where he can make long queues, while saving demand where the queues will be short or negative (section 1.8.4).

Using the idea above, we chose the following heuristic:

1. Assign the minimum demand to all groups (required)
2. order the groups
3. assign maximum demand to the first group, then the second group
4. continue until knapsack constraint is reached
5. rest of the groups will have minimum demand

The only part missing is how to order the groups. One way is to use cyclic ordering $[1, 2, \ldots, n], [2, \ldots, n, 1], \ldots, [n, 1, \ldots, n-1]$. Another way is to use a random permutation $[\pi(1), \pi(2), \ldots, \pi(n)]$. Where we have used a head start, we first used cyclic ordering, then random permutations if we wanted more demand vectors in the starting set.

3.4 Reference solution

In order to test the algorithm, we wanted a reference solution. For this we use the cutting plane algorithm as in section 3.3, but with one small difference: $\tilde{Y} = \{\beta\}$. Because β is the worst case demand possible, the adversary will not be able to find any demand vector with a worse objective value, thus the algorithm will stop very fast after only one iteration.

3.4.1 Quadratic objective function

As the reference solution only considers the worst case demand, we could expect that the queues are long. In particular the queues will exceed the external threshold Ext in all or almost all groups. In such a case, the objective function is in practice linear, which opens up for skewed solutions. This is because a balanced solution with $q = (Ext + 10, Ext + 10)$ has the same objective value as an unbalanced solution with $q = (Ext + 20, Ext)$.

In order to fix this, we made a second objective function, which is still piecewise linear, but with so many pieces that it in practice is quadratic.

Note that because we have different objective functions, the objective values can not be compared. We can, however, compare the solution with other solutions found using, for instance, the standard reference solution in section 3.4, or the I-A algorithm in section 3.3.

3.5 Implementation

Even though the simplex algorithm is mathematically simple, it is numerically unstable. In addition we were more interested in the model and the algorithm than the programming, and decided to use an existing solver. Of the solvers we could use, CPLEX [18] is definitely one of the best ones, allowing us to solve MIPs with a reasonable size fast.

There are three ways to run CPLEX that we considered. First you have the graphical OPL studio which is good for writing models fast, with its compact input format. It lacks a good programming interface though, and runs only on windows.

The other extreme is the callable library, where you write a program in for instance C++, and call the CPLEX library methods. This offer total control and should be the fastest. C++ is even my favorite programming language, but we discarded this method because we did not need the low-level control offered.

Perfect for our purpose was the stand-alone command-line CPLEX program. It has pre-built commands for reading a problem in standard LP format, adding additional constraints to a problem, writing the solution to file and of course to solve a problem.

I could either send instructions to CPLEX, wait for the results and quit. Or we could use the interactive mode which was perfect when running the implementor-adversary algorithm.

Python is a high-level programming language which is great for fast development, and offers a relatively easy interface to process management. We ended up building a python script acting as a controller. It had two CPLEX instances as child processes, and handled the communication between them. It also had methods for parsing the output and generating the LP files needed by the CPLEX processes. When it decided that an optimal solution had been found, it would generate a run plot using gnuplot, with statistics on running time and convergence.

This program took some time to write, but made it very easy to test new objective functions and constraints, or new input to the problem. It is very inspiring to have an idea and see the results in a matter of minutes

We also used some PHP scripts so that we could write input to the python scripts through a web browser. With these scripts we could also organize the input sets and also view the results after the program had finished. For some time we also ran CPLEX through the web server, but had to stop because of security issues.

The biggest problem with the program is the "just one more feature"-approach. After a while, the program is quite messy and difficult to understand. The computational

results can be verified though, by looking at the generated .lp files which can be solved by CPLEX manually.

3.5.1 Code optimization

Most of the execution time ($> 90\%$) is used in the CPLEX processes, therefore we did not bother to optimize the python scripts. There are a lot of parameters that can be set in CPLEX which affects the methods used for solving programs. We have not tried many of these settings, but my impression is that CPLEX is very capable of choosing the best settings automatically. And for our purposes, we would probably use more time testing out settings than we would save on running times.

The best optimizations were the ones made to the model and algorithm. And citing Donald Knuth: "premature optimization is the root of all evil". First after we are certain which model and algorithm to use, we should put an effort in optimizing.

3.5.2 Auxiliary constraints

Sometimes even state-of-the-art programs like CPLEX are not as smart as you should expect. Then it is possible to help it along the way.

Even though it should be obvious, that when you add constraints to a problem, the optimal solution cannot improve, it may not be seen by the solver. In that case we may add the constraint that $\zeta \geq$ lower bound. It does not change the problem, and can only improve running time.

It is also possible that there might be two or more solutions, where both have the same optimal value, but one is dominated by the other. We were surprised to see solutions with holes in the schedule and also assignment of blocks to groups who for certain are not going to need them.

These dominated solutions can be removed by adding tie-breaking rules or heuristics, without changing the optimal value. We have removed holes in the schedule, but there are still a few suboptimal schedules which are marked in the text.

4 Results

In this section we will discuss how to test the programs, describe the test cases, and also show the results in terms of running time and optimality for a large number of different inputs and algorithm parameters.

In order to test the programs, both in terms of running time and in optimality, we need test cases. First we need data which can be put into our model and used in the algorithms. But we also want to test the solutions found by the algorithms with realistic demand scenarios.

4.1 Model input

We created the model input with two aims in mind. It should be as realistic as possible, modeling something that could have happened. But we also wanted it to be computationally challenging. In particular it should disfavor greedy algorithms to show the potential of robust optimization.

In the end we came up with two input data sets, a normal and a large. They have much in common, every week consists of 5 days, and every day consists of 6 unit blocks or time slots (denoted by hours, though in practice a unit block would be 2 hours. With $6 \cdot 2$ hours, we include overtime). Both have 6 surgeon groups, and each group have patients in 4 different difficulty classes, each requiring 1-4 unit blocks depending on class. Thus we consider 24 classes in total. We also have 5 rooms available in both cases.

With 5 rooms and 6 surgeon groups, rooms is the main limiting constraint, though surgeon resources may also be limiting, especially if some groups are in a great demand.

With 6 unit blocks, there are a total of 18 blocks each day (6,5,4,3 blocks of length 1,2,3,4).

4.1.1 Normal input data set

This is a schedule running for 1 week, thus it has $1 \cdot 5 \cdot 6 = 30$ unit blocks. With 5 rooms and 6 surgeon groups, we have $30 \cdot 5 = 150$ "room hours" and $30 \cdot 6 = 180$ "surgeon hours".

The demand for each class is given in the following table:

hours	1	2	3	4	Total blocks	Total hours
group 1	5-7	2-5	3-5	0-1	10-18	18-36
group 2	4-5	1-3	1-3	4-7	10-18	25-48
group 3	2-3	5-6	1-3	2-4	10-16	23-40
group 4	2-3	4-6	0-0	2-4	8-13	18-31
group 5	3-5	5-6	2-4	1-2	11-17	23-37
group 6	4-7	3-4	3-5	1-2	11-18	23-38
Sum	20-30	20-30	10-20	10-20	60-100	130-230

The first and second number in each cell is the lower and upper limit, respectively. Total blocks is the sum over all block lengths for each group. Total hours is the number of blocks times the length of the block. Lower limits are summed together, as well as upper limits.

The limit on the total number of hours was set to 150, 155 or 160. The adversary could in other words distribute blocks with a extra demand of 20-30 hours in addition to the lower limit of 130.

The total number of integer decision variables are $18 \cdot 5 \cdot 1 \cdot 6 \cdot 5 = 2700$ (blocks/day, days, weeks, surgeon groups, rooms).

The parameters for the objective function are $c_0 = 1$ and $c_1 = 3$, and $Ext = 1$. In other words, the cost of the first queue unit is 1, while the cost is 3 for every following queue unit.

4.1.2 Large input data set

To really test the algorithm, we made a second test set spanning over 7 weeks. Thus the number of room and surgeon hours is increased 7-fold to 1050 and 1120 respectively.

hours	1	2	3	4	Total blocks	Total hours
group 1	7-10	55-70	14-20	2-3	78-103	167-222
group 2	14-18	62-75	2-4	0-0	78-97	144-180
group 3	31-40	84-95	1-2	0-0	116-137	202-236
group 4	13-15	20-35	1-2	0-0	34-52	56-91
group 5	5-10	73-90	17-25	1-2	96-127	206-273
group 6	16-25	53-70	2-4	0-0	71-99	128-177
Sum	86-118	347-435	37-57	3-5	473-615	903-1179

The quantities are taken from a realistic scenario from a hospital, and we see that reality is not always well-balanced. Group 4 has little to do, while groups 5, 3 and 1 will most likely have to work the maximum 210 hours. We chose a length of 7 weeks to make the number of possibilities as high as possible and to disfavor a simple greedy solution.

The total number of blocks was limited by 950 to 1050, thus we could distribute 47 to 147 extra hours.

The total number of integer decision variables are $18 \cdot 5 \cdot 7 \cdot 6 \cdot 5 = 18900$ (blocks/day, days, weeks, surgeon groups, rooms).

The parameters for the objective function are $c_0 = 1$, $c_1 = 3$ and $Ext = 3$. To reflect the larger input data, the external threshold was increased.

4.2 Running time

The main results on running time are given in table (1).

Name consists of the test set (normal/large), the size of the knapsack constraint (KC) in parenthesis, and the head start (section 3.3.5) given in number of cyclic (up to 24) + random adversary solutions. **Imp.** and **adv.** is the running time in seconds for the implementor and adversary program respectively, and **total** is the sum of these. **Total iter.** is the total number of iterations needed until an optimal solution was found. **Relaxed** and **relaxed iter** is the time and number of iterations needed for the relaxed problem.

The first line is a normal test set with KC 150 and no head start, but with *integer* constraints on the decision variables x as in section 3.3.4, but from the beginning.

Run plots for the normal input set are shown in figures (3)-(15), which show the progress of the algorithm in more detail, both convergence and running time. The optimal value was 42, 52 and 62 for the input with knapsack constraint 150, 155 and 160 respectively.

For the large input sets, the run plots are shown in figures (16)-(18). The optimal values was found to be 94, 181 and 200 for the large input with KC 950, 1000 and 1050 respectively.

Table 1: Running time of cutting planes algorithm with different input

name	imp.	adv.	total	total iter.	relaxed	relaxed iter.
normal(int) - 0	591	10	601	36	-	-
normal(150) - 0	328	11	339	39	333	38
normal(150) - 24	498	10	509	33	507	32
normal(150) - 48	583	9	592	26	581	25
normal(150) - 100	812	8	821	21	819	20
normal(150) - 200	302	3	305	6	263	2
normal(150) - 400	378	2	381	3	378	2
normal(150) - 800	790	7	797	5	739	3
normal(155) - 0	2566	27	2594	87	2473	76
normal(155) - 24	3727	26	3753	74	3703	69
normal(155) - 100	5585	30	5616	67	5534	63
normal(160) - 0	17529	97	17627	230	17625	229
normal(160) - 24	35950	102	36053	223	36051	222
large(950) - 0	1630	14	1645	22	1358	15
large(1000) - 0	852	13	866	20	804	15
large(1050) - 0	10628	57	10686	82	8555	55

Figure 3: Run plot: normal input with no head start.
Total demand was limited to 150 hours. Did not solve relaxed problem first, used integer variables all the time

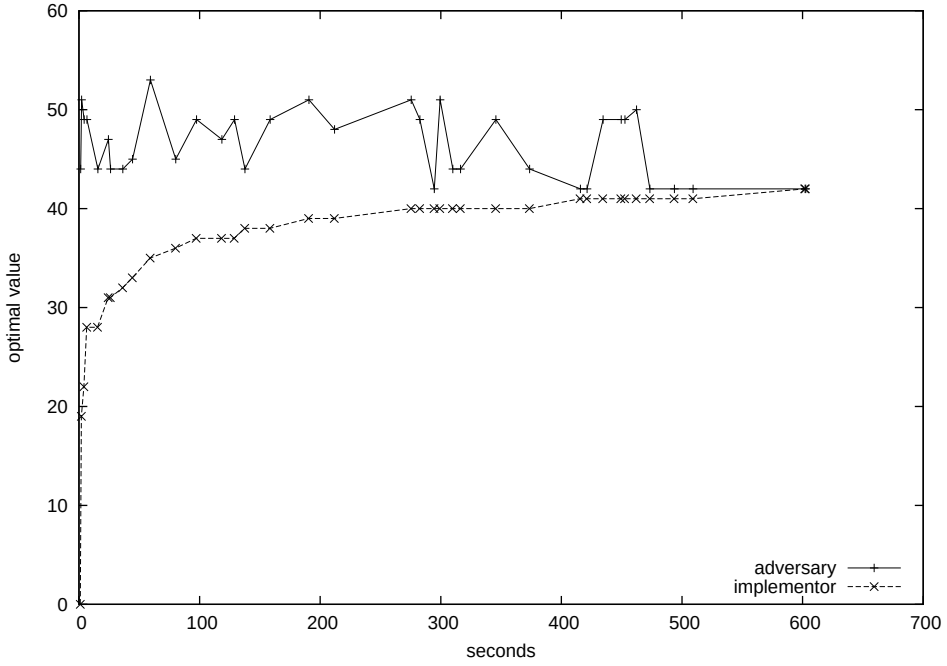


Figure 4: Run plot: normal input with no head start.
Total demand was limited to 150 hours.

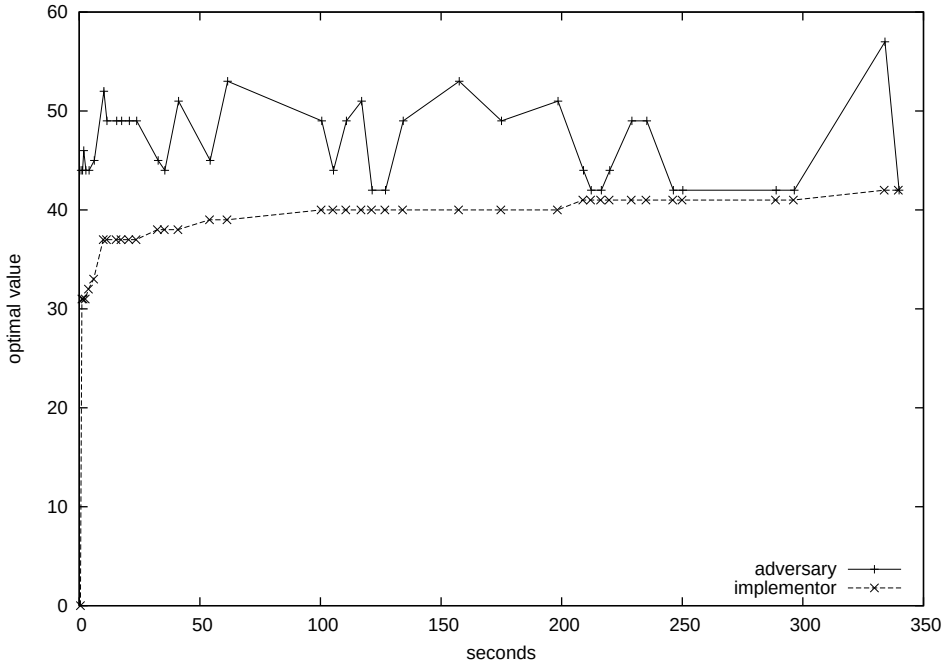


Figure 5: Run plot: normal input with head start 24.
Total demand was limited to 150 hours.

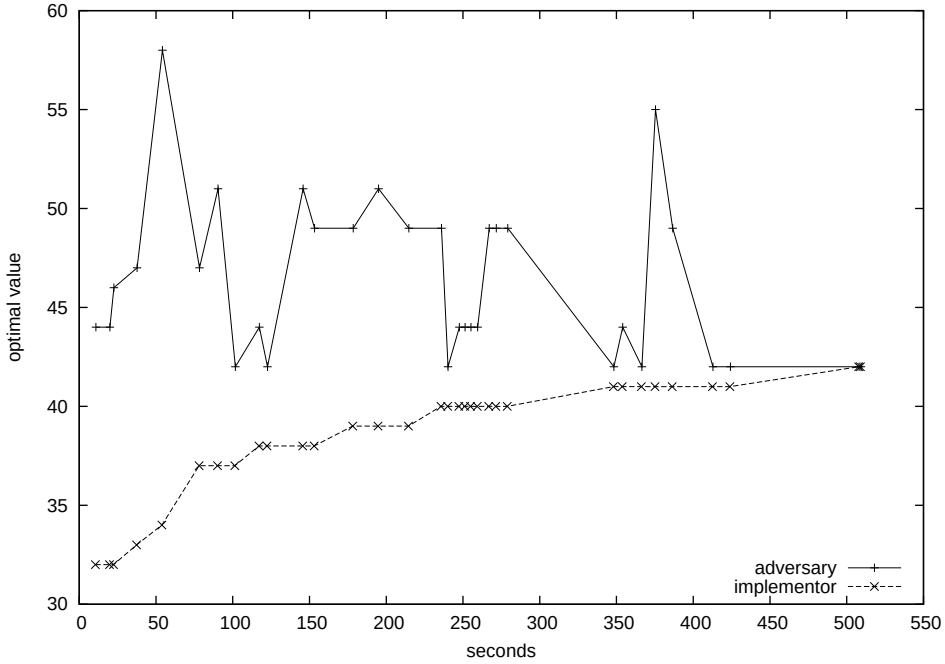


Figure 6: Run plot: normal input with head start 48.
Total demand was limited to 150 hours.

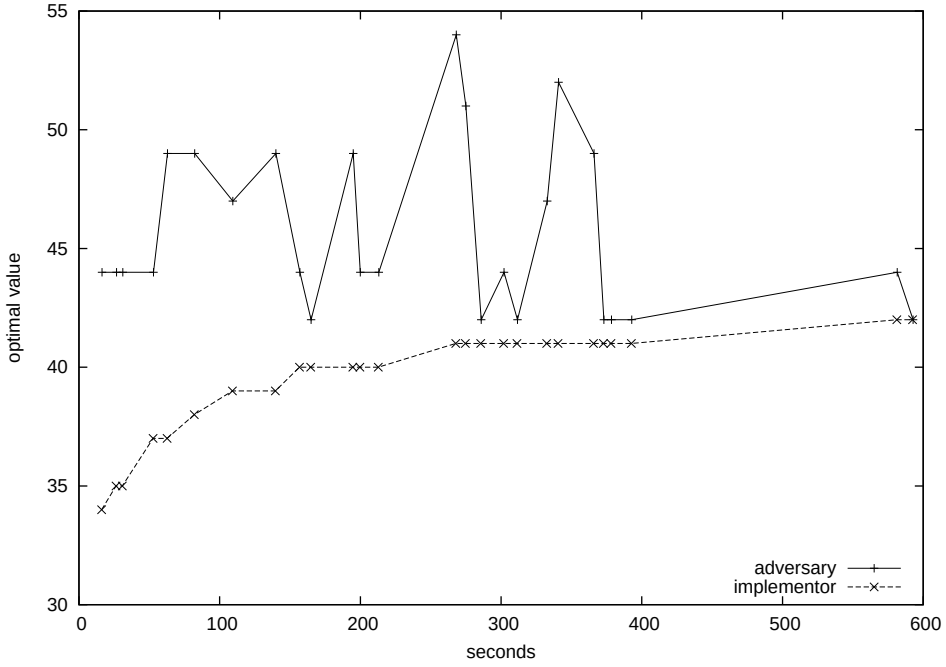


Figure 7: Run plot: normal input with head start 100.
Total demand was limited to 150 hours.

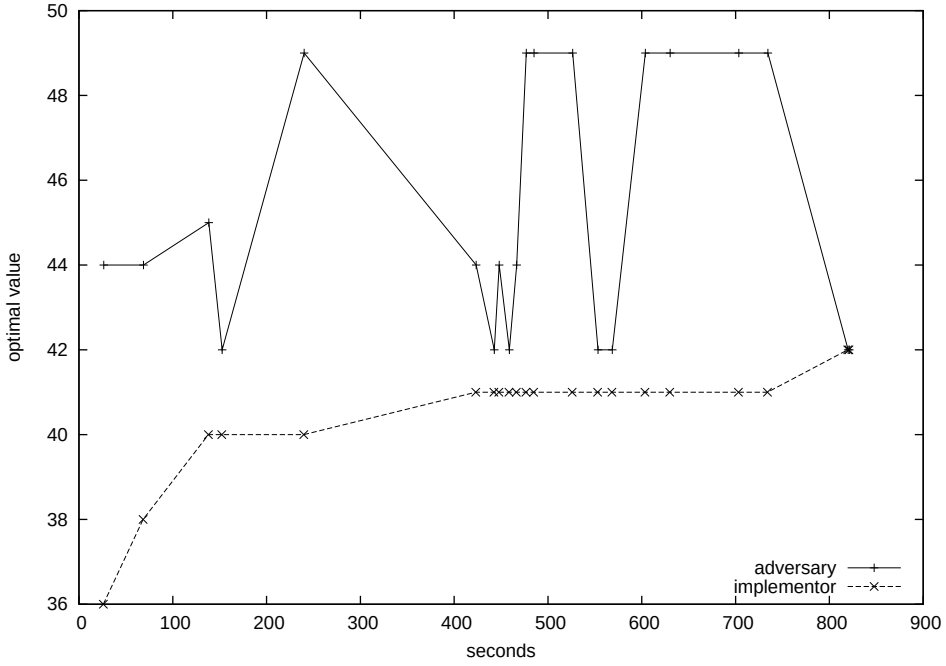


Figure 8: Run plot: normal input with head start 200.
Total demand was limited to 150 hours.

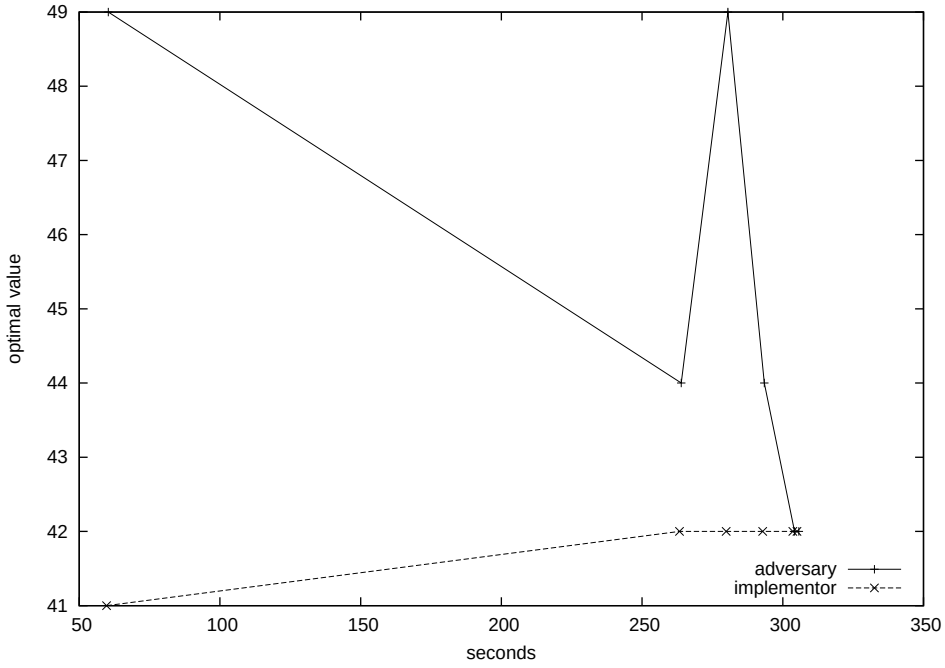


Figure 9: Run plot: normal input with head start 400.
Total demand was limited to 150 hours.

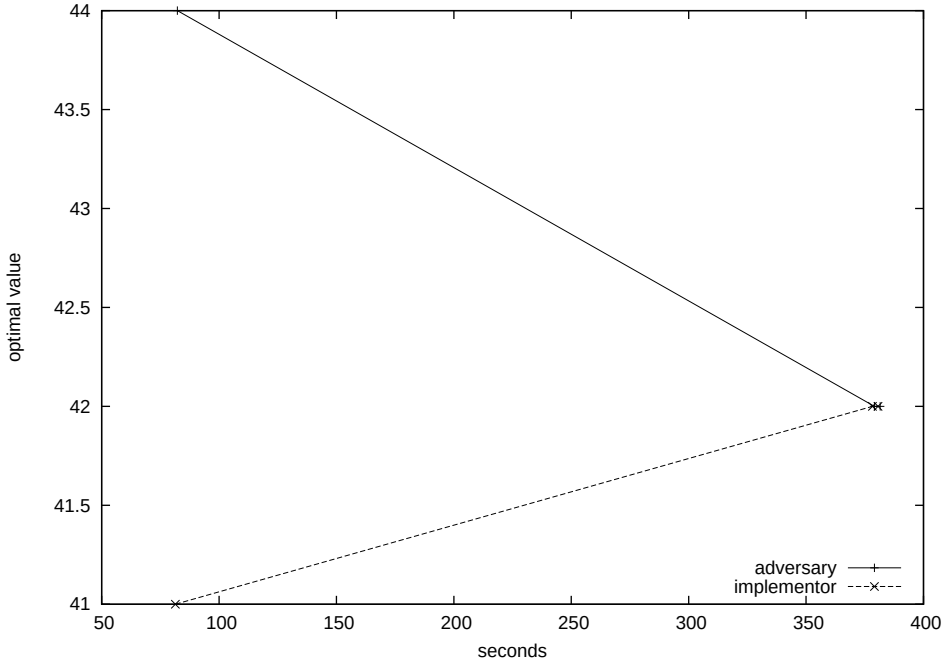


Figure 10: Run plot: normal input with head start 800.
Total demand was limited to 150 hours.

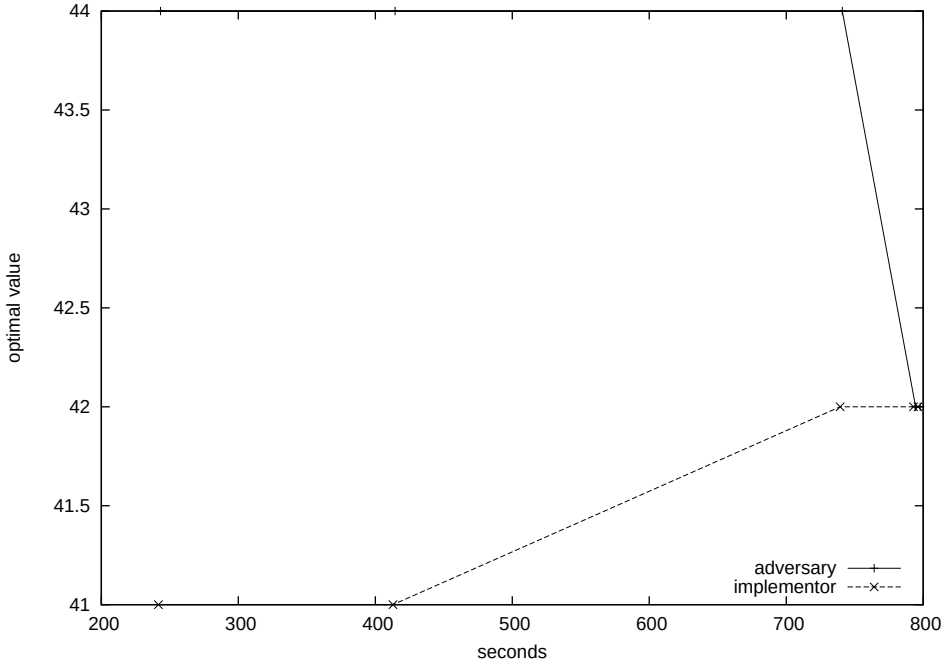


Figure 11: Run plot: normal input with no head start.
Total demand was limited to 155 hours

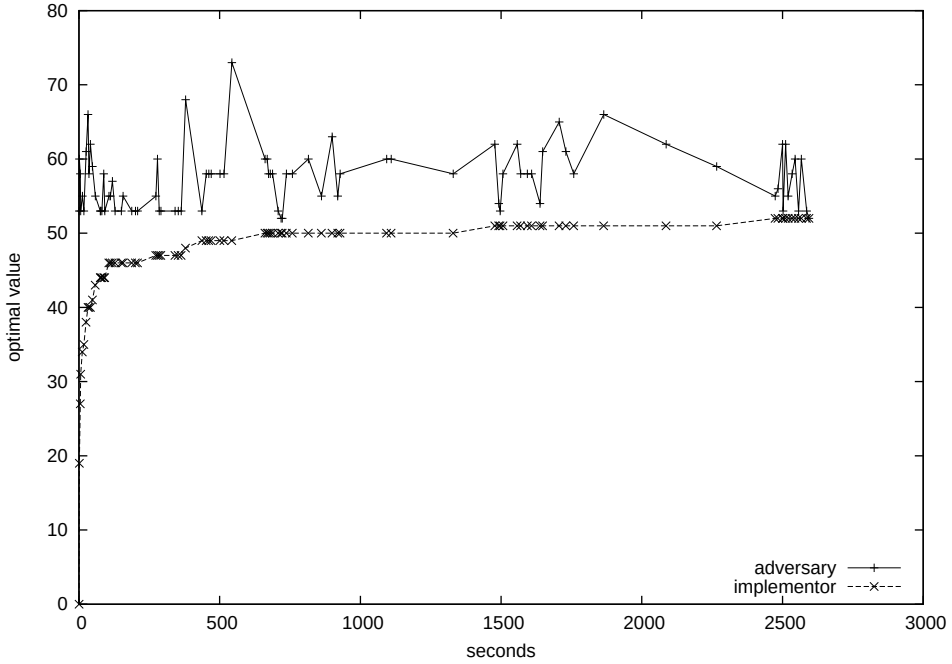


Figure 12: Run plot: normal input with head start 24.
Total demand was limited to 155 hours

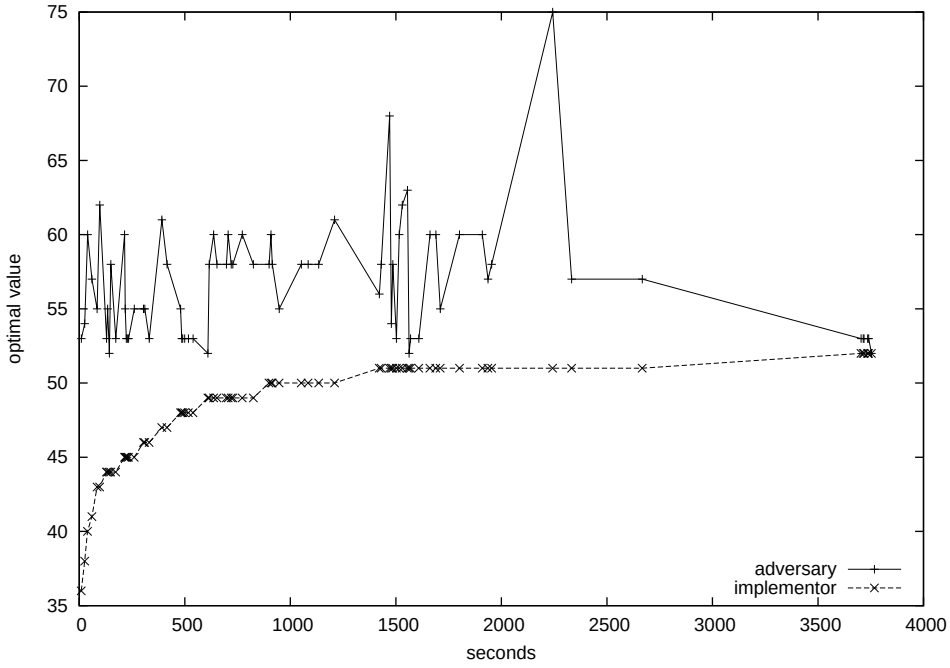


Figure 13: Run plot: normal input with head start 100.
Total demand was limited to 155 hours

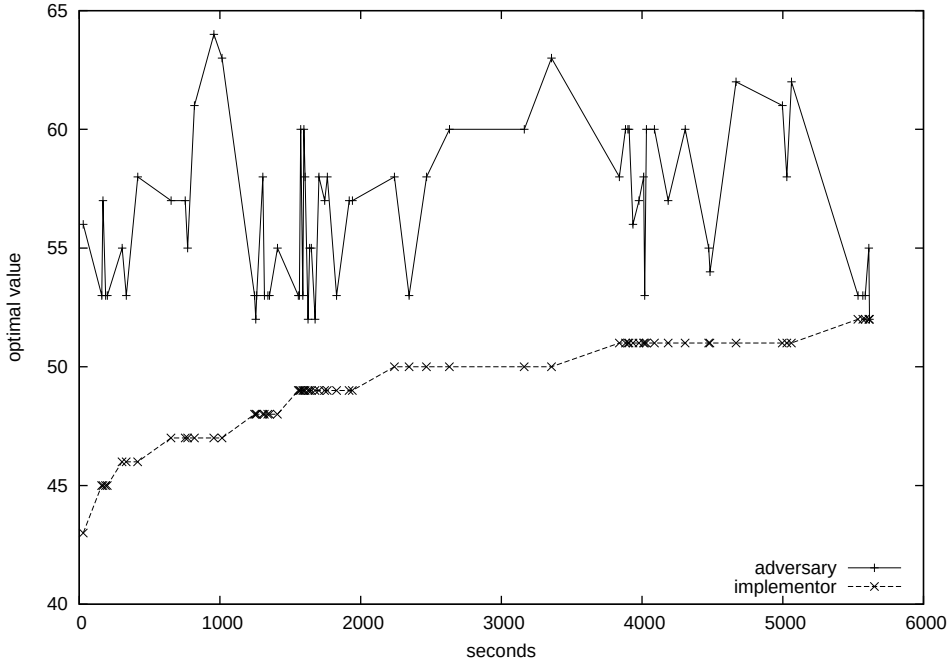


Figure 14: Run plot: normal input with no head start.
Total demand was limited to 160 hours

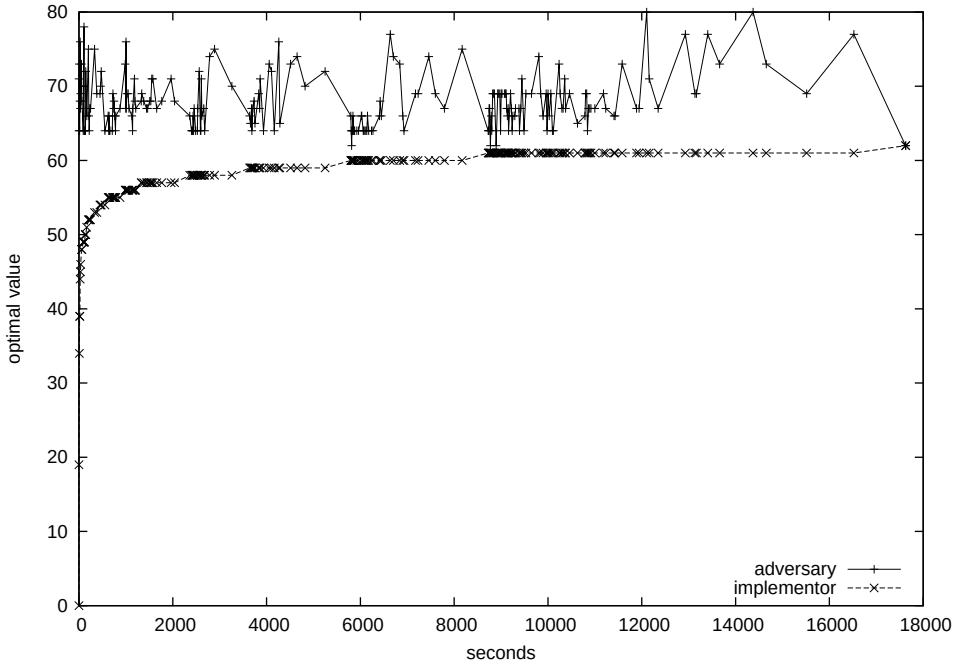


Figure 15: Run plot: normal input with head start 24.
Total demand was limited to 160 hours

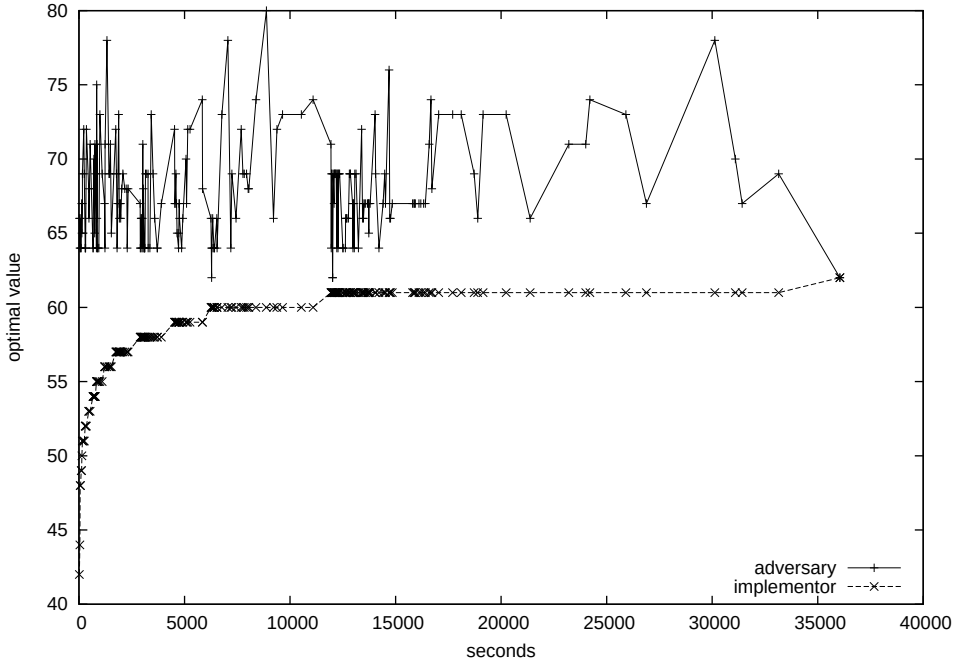


Figure 16: Run plot: large input with no head start.
Total demand was limited to 950 hours

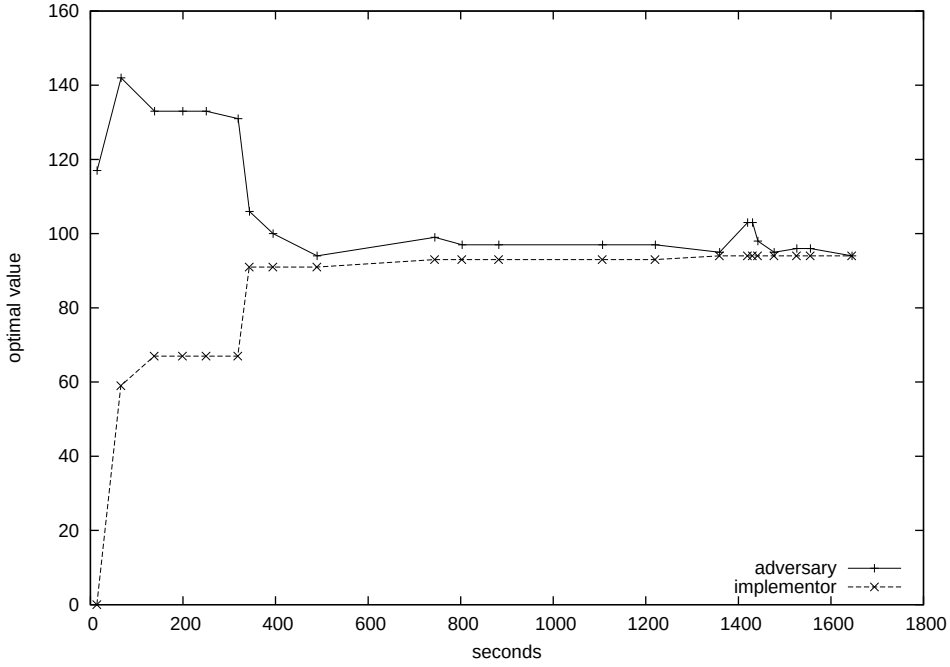


Figure 17: Run plot: large input with no head start.
Total demand was limited to 1000 hours

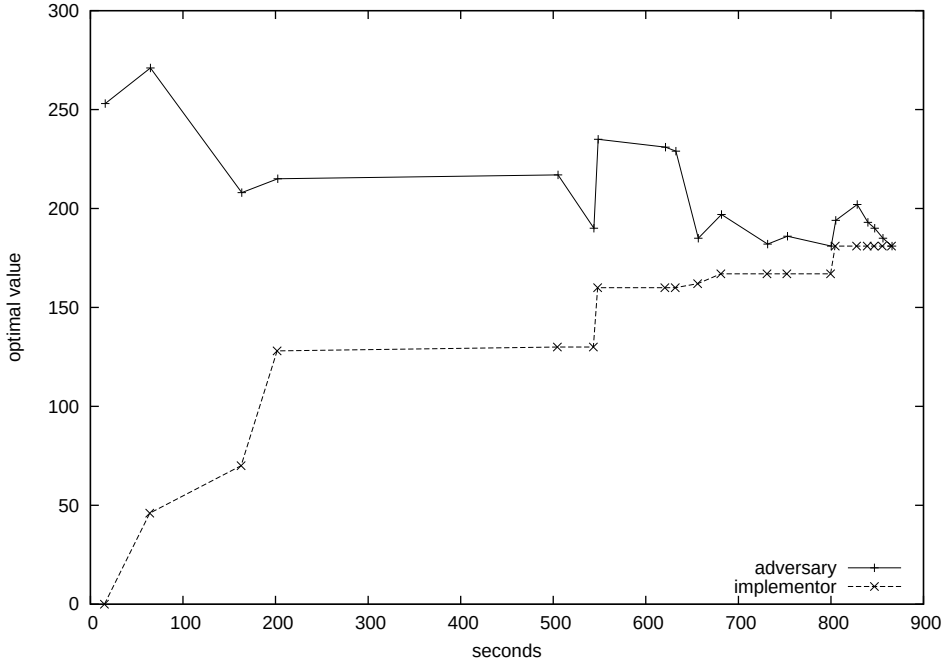


Figure 18: Run plot: large input with no head start.
Total demand was limited to 1050 hours

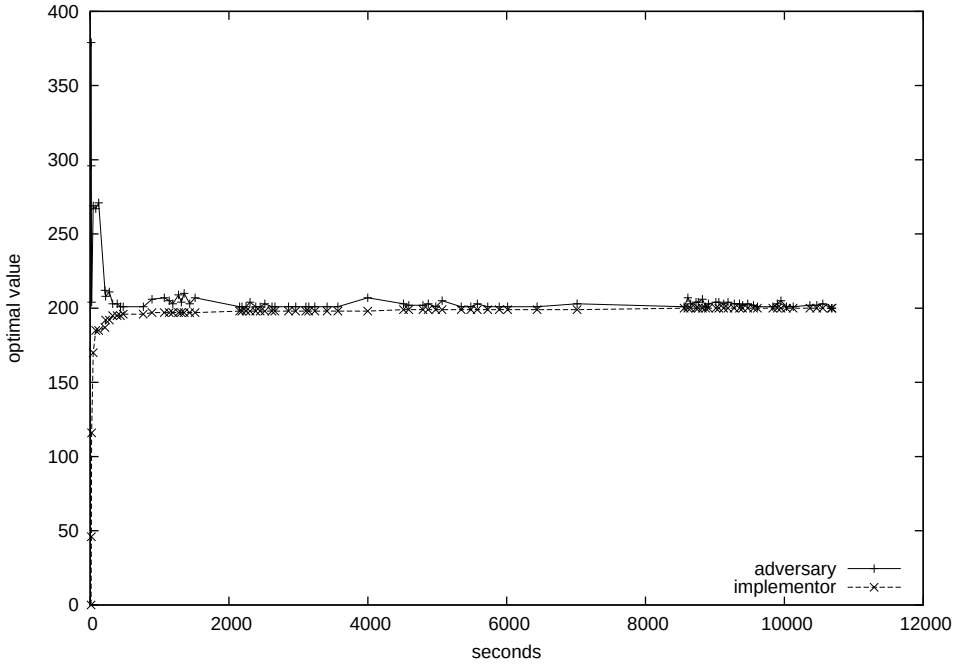


Table 2: Summary of solution with cutting plane algorithm on normal input. KC = 150 hours

hours	1	2	3	4	Total blocks	Total hours
group 1	6	3	3	0	12	21
group 2	5	2	1	4	12	28
group 3	2	6	1	3	12	29
group 4	2	5	0	3	10	24
group 5	4	5	2	1	12	24
group 6	5	3	3	1	12	24
Sum	24	24	10	12	70	150

4.3 Solutions

4.3.1 Cutting plane - normal input

We have included three of the solutions from the normal input, with KC 150, 155 and 160. With the KC150 solution, we have used the non-relaxed problem, and in all cases only the final, and thus feasible solution is included. We used the run with no head start.

The solutions are shown in tables (3), (5) and (7), with summaries in tables (2), (4) and (6).

4.3.2 Reference solution - normal input

We also solved the problem using the reference solution described in section 3.4, both with the same objective function as the cutting plane algorithm, but also with a quadratic objective function (section 3.4.1. The solutions are shown in tables (8)-(11)

Table 3: Solution with cutting plane algorithm on normal input, KC = 150 hours

	Room 1					Room 2					Room 3					Room 4					Room 5				
Day	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
H 1	**2**	**5**	**3**	**6**	**3**	**6**	**4**	**6**	**2**	**2**	**3**	**3**	**2**	**5**	**5**	**1**	**6**	**5**	**3**	**4**	**5**	**1**	**1**	**4**	**6**
H 2	**2**	5	3	6	3	6	4	**4**	2	2	3	3	2	**1**	5	1	6	5	3	4	5	**2**	1	4	6
H 3	**4**	**3**	3	6	**5**	**3**	**4**	**4**	**3**	2	**2**	**5**	**2**	1	**1**	1	6	5	**2**	**4**	5	2	1	4	6
H 4	**1**	3	**3**	6	5	3	4	4	3	**3**	2	**1**	2	1	**1**	**4**	**5**	5	2	4	**6**	2	**6**	4	**6**
H 5	**5**	3	3	**4**	**5**	3	4	**1**	3	**3**	2	1	2	**6**	**2**	4	5	**5**	2	4	6	2	6	**1**	6
H 6	5	3	**6**	4	**1**	3	4	1	3	3	2	**2**	2	**5**	**2**	**6**	5	5	2	4	**1**	**6**	**3**	1	6

Table 4: Summary of solution with cutting plane algorithm on normal input. KC = 155 hours

hours	1	2	3	4	Total blocks	Total hours
group 1	6	3	4	0	13	24
group 2	4	2	2	4	12	30
group 3	2	5	2	2	11	26
group 4	2	5	0	2	9	20
group 5	3	5	2	1	11	23
group 6	5	3	4	1	13	27
Sum	22	23	14	10	69	150

Table 5: Solution with cutting plane algorithm on normal input, KC = 155 hours

	Room 1					Room 2					Room 3					Room 4					Room 5				
Day	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
H 1	**1**	**2**	**4**	**5**	**6**	**6**	**1**	**6**	**6**	**3**	**3**	**4**	**5**	**1**	**1**	**2**	**3**	**1**	**2**	**5**	**4**	**6**	**2**	**3**	**2**
H 2	1	2	4	**4**	6	6	1	**6**	**6**	3	**3**	4	5	1	1	2	**3**	1	2	**4**	**4**	6	2	3	**2**
H 3	1	2	**3**	4	**2**	**5**	1	6	**5**	**3**	3	4	**5**	**2**	1	2	3	1	**6**	4	4	**5**	2	**3**	**5**
H 4	**6**	**2**	3	**6**	2	5	**6**	6	5	3	3	4	**1**	2	**6**	2	3	**5**	**1**	**1**	4	5	2	3	5
H 5	6	2	3	6	2	5	6	6	5	3	**2**	**4**	**2**	2	6	**3**	3	5	**1**	**4**	4	**5**	**1**	**3**	**5**
H 6	6	2	3	6	2	5	6	**2**	**4**	**1**	2	4	**6**	2	6	3	**1**	5	1	4	**1**	5	1	3	5

Table 6: Summary of solution with cutting plane algorithm on normal input. KC = 160 hours

hours	1	2	3	4	Total blocks	Total hours
group 1	6	3	4	0	13	24
group 2	4	2	1	4	11	27
group 3	2	6	2	2	12	28
group 4	2	5	0	2	9	20
group 5	4	5	3	1	13	27
group 6	5	3	3	1	12	24
Sum	23	24	13	10	70	150

Table 7: Solution with cutting plane algorithm on normal input, KC = 160 hours

	Room 1					Room 2					Room 3					Room 4					Room 5				
Day	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
H 1	**1**	**5**	**5**	**3**	**4**	**3**	**3**	**1**	**4**	**3**	**5**	**2**	**6**	**1**	**6**	**4**	**4**	**3**	**5**	**5**	**2**	**1**	**2**	**2**	**2**
H 2	**1**	5	5	3	4	3	3	1	4	3	**5**	2	**6**	1	6	4	4	**2**	5	5	2	**1**	**4**	2	**2**
H 3	**6**	5	**5**	**4**	**3**	**3**	**6**	**2**	**1**	**4**	**4**	2	6	**2**	6	**5**	4	**1**	**3**	5	**2**	**3**	4	**6**	2
H 4	**6**	**6**	**3**	**6**	3	3	**1**	2	**3**	4	**1**	2	**1**	**5**	6	5	4	**5**	**2**	5	2	3	**6**	**1**	2
H 5	6	6	3	6	**6**	3	1	2	3	4	1	**4**	1	5	**1**	**5**	**5**	5	2	**3**	2	3	6	1	2
H 6	**6**	6	3	6	6	3	1	2	3	4	1	4	1	5	1	5	5	5	2	3	2	3	6	1	**5**

Table 8: Summary of solution with reference algorithm on normal input

hours	1	2	3	4	Total blocks	Total hours
group 1	6	4	4	0	14	26
group 2	4	2	1	4	11	27
group 3	2	5	1	2	10	23
group 4	2	5	0	3	10	24
group 5	4	5	3	0	12	23
group 6	5	3	4	1	13	27
Sum	23	24	13	10	70	150

Table 9: Solution with reference algorithm on normal input

	Room 1					Room 2					Room 3					Room 4					Room 5				
Day	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
H 1	**1**	**2**	**3**	**6**	**3**	**3**	**1**	**2**	**1**	**6**	**2**	**4**	**5**	**3**	**4**	**4**	**6**	**4**	**2**	**2**	**5**	**3**	**1**	**5**	**1**
H 2	**4**	**5**	**5**	6	3	**3**	1	**6**	**4**	6	2	4	**2**	3	4	**1**	6	4	2	2	5	3	**1**	5	1
H 3	**4**	5	5	6	**4**	3	**2**	**6**	4	6	**2**	**6**	2	3	**3**	1	**3**	4	2	**2**	**6**	**1**	1	5	1
H 4	4	**1**	**1**	**6**	4	**1**	2	6	**1**	**6**	2	**5**	2	**4**	3	**5**	3	4	2	2	**6**	**6**	**3**	**5**	**5**
H 5	4	1	1	6	4	1	2	6	**3**	6	2	5	2	4	3	5	3	**4**	**5**	2	6	6	3	**1**	5
H 6	4	1	1	6	4	1	2	6	3	**1**	2	5	**5**	**2**	3	**5**	3	4	5	**6**	**6**	6	**2**	1	5

Table 10: Summary of solution with reference algorithm, quadratic objective function, on normal input

hours	1	2	3	4	Total blocks	Total hours
group 1	5	4	4	0	13	25
group 2	4	1	0	5	10	26
group 3	2	4	2	2	10	24
group 4	2	5	0	3	10	24
group 5	4	4	3	1	12	25
group 6	6	2	4	1	13	26
Sum	23	20	13	12	68	150

Table 11: Solution with reference algorithm, quadratic objective function, on normal input

	Room 1					Room 2					Room 3					Room 4					Room 5				
Day	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
H 1	**5**	**3**	**5**	**6**	**5**	**3**	**2**	**1**	**1**	**2**	**4**	**5**	**2**	**4**	**6**	**2**	**1**	**6**	**5**	**3**	**6**	**6**	**4**	**2**	**1**
H 2	**1**	3	5	6	5	3	**6**	1	1	2	4	**4**	2	4	6	2	1	6	5	3	6	**5**	4	**2**	1
H 3	1	**2**	5	6	5	3	**6**	1	**1**	2	**4**	4	**3**	4	6	2	**3**	**2**	5	3	**6**	5	4	2	**1**
H 4	1	2	**6**	**3**	**5**	3	6	**1**	1	2	4	**4**	3	4	**4**	2	3	2	5	**6**	6	5	4	2	1
H 5	**3**	2	**5**	3	5	**5**	6	1	1	**1**	4	**5**	3	**5**	4	1	**4**	2	**6**	6	6	**3**	**4**	2	**3**
H 6	3	2	5	3	**3**	5	6	1	**2**	**5**	4	5	3	**6**	**1**	**1**	**4**	2	**1**	6	**6**	3	4	**4**	**2**

Table 12: Summary of solution with cutting plane algorithm on large input. KC = 950 hours

hours	1	2	3	4	Total blocks	Total hours
group 1	10	59	19	3	91	197
group 2	18	66	2	2	88	164
group 3	33	88	0	0	121	209
group 4	16	28	2	5	51	98
group 5	5	77	17	0	99	210
group 6	24	67	2	2	95	172
Sum	106	385	42	12	545	1050

4.3.3 Cutting plane - large input

We have also solved the large input set with three different values for KC: 950, 1000 and 1050. Because the schedules are so large, we have only included one of them in full, in table (15). The summaries are in tables (12), (13) and (14)

4.3.4 Reference solution - large input

The large input was also solved using the reference solution from section 3.4, and the result are summarized in table (16)

4.4 Optimality testing

Solutions found by a LP or MIP can be proven to be optimal. But they are only optimal in terms of the specific constraints and optimal function. In our case with the cutting

Table 13: Summary of solution with cutting plane algorithm on large input. KC = 1000 hours

hours	1	2	3	4	Total blocks	Total hours
group 1	7	67	18	3	95	207
group 2	17	72	4	0	93	173
group 3	34	88	0	0	122	210
group 4	13	32	2	3	50	95
group 5	5	74	19	0	98	210
group 6	20	63	3	0	86	155
Sum	96	396	46	6	544	1050

Table 14: Summary of solution with cutting plane algorithm on large input. KC = 1050 hours

hours	1	2	3	4	Total blocks	Total hours
group 1	11	67	17	3	98	208
group 2	18	72	3	0	93	171
group 3	36	87	0	0	123	210
group 4	15	32	2	0	49	85
group 5	7	82	13	0	102	210
group 6	22	66	4	0	92	166
Sum	109	406	39	3	557	1050

plane algorithm, optimality is defined by having the minimal worst case bound, when the adversary can choose freely from Y_K.

Two question we can ask ourselves is if the worst case bound is representative for the average performance, and what happens if we draw scenarios with a different knapsack constraint, or none at all?

There are more than 5 billion demand scenarios only on the normal input (for normal input: $|Y| = 5,804,752,896$. Large input: $|Y| = 4,550,926,270,464,000$), and we would like to see how the solutions perform on average too, not just on the worst cases. We are also interested in what happens when the demand is not as expected and the knapsack constraint is violated.

Thus we would like to look at how good the solutions are in practice, by drawing demand scenarios from random distributions and comparing how the demand is met by the different solutions. We want to find the *expected value* of the objective function.

To find the expected value of a function, we can either find the probability of each input, and use this with the function value to calculate the expected value as an integral or a sum. Or we can use the monte-carlo method [16] to draw a random sample from the set of all demand scenarios. The latter is by far the easiest and fastest, and also yields good results.

The function used for comparison is the same as the piece-wise linear objective function used in both the cutting plane algorithm and the reference solution (but not the quadratic reference solution).

4.4.1 Sampling distributions

We still need a statistical distribution from which we draw the random sample. The distribution should match a realistic distribution of the actual demand. For simplicity we chose a *binomial distribution*; first we assign the minimum demand to all groups. Then for all groups, we look at the potential demand for extra patients, and for each patient we flip a coin to see if he should be included or not. Note that the coin is not necessarily fair, we can change the probability to adjust the expected total demand.

In all but 2 cases, the samples were filtered, only using those with a total demand equal to a certain value, like a knapsack constraint. In one case, we used a binomial distribution without filtering, and in the last case a uniform distribution was used, drawing demands for each group/length randomly between the lower and upper bound inclusive.

In all cases, sample size was 100,000, drawn randomly and independent, with replacement. All solutions were tested on the same samples.

Table 15: Solution with cutting plane algorithm, large input. KC = 1000 hours

	Room 1					Room 2					Room 3					Room 4					Room 5				
Day	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Hour 1	5	5	3	4	2	1	3	5	1	6	3	1	1	6	3	2	4	4	5	5	6	2	6	3	1
Hour 2	1	5	3	4	2	5	6	5	1	6	3	1	1	3	3	2	3	4	5	5	6	2	6	2	1
Hour 3	1	3	3	4	5	5	6	5	1	1	2	1	1	3	2	6	5	2	5	3	3	2	6	2	6
Hour 4	1	3	3	4	5	5	1	1	3	1	2	6	5	6	2	6	5	2	5	3	3	2	6	1	6
Hour 5	1	6	3	5	1	5	1	1	3	3	3	5	5	6	6	4	3	6	2	2	6	4	2	1	5
Hour 6	1	6	3	5	1	5	1	1	3	3	3	5	5	6	6	4	3	6	2	2	6	4	2	1	5
Hour 1	5	6	6	2	1	3	1	5	1	2	2	2	2	4	6	1	3	4	3	5	4	5	3	5	3
Hour 2	5	6	1	2	1	6	1	5	1	2	2	4	2	4	6	1	3	4	3	5	3	5	3	5	3
Hour 3	5	6	1	3	1	6	1	5	1	6	2	4	3	5	2	1	3	6	2	5	3	5	2	6	3
Hour 4	2	5	2	3	5	5	1	5	1	6	6	6	3	5	2	1	3	6	2	1	3	2	1	6	3
Hour 5	3	5	2	4	5	5	1	5	3	6	6	6	6	6	3	1	3	3	1	1	2	2	1	5	4
Hour 6	3	5	2	4	5	5	1	5	3	6	4	2	6	6	3	1	3	3	1	1	2	4	1	5	4
Hour 1	1	4	2	6	3	5	5	3	3	2	3	1	1	5	5	6	3	5	2	1	2	2	6	1	4
Hour 2	1	4	2	6	3	5	5	3	3	6	3	1	1	5	5	6	2	5	2	1	2	3	6	1	4
Hour 3	5	4	3	5	3	2	3	4	3	6	3	5	1	6	5	6	2	5	2	1	1	1	6	1	4
Hour 4	5	4	3	5	3	2	3	4	3	6	3	5	1	6	5	6	2	5	2	1	1	1	6	1	4
Hour 5	3	5	6	4	2	2	1	4	6	1	6	3	1	3	3	5	2	5	5	5	1	6	3	1	4
Hour 6	3	5	6	4	2	2	1	4	6	1	4	3	1	3	3	5	2	5	5	5	1	6	3	1	6
Hour 1	3	5	5	3	1	4	3	6	2	3	1	1	2	1	2	5	6	3	4	6	6	2	1	5	5
Hour 2	3	5	5	2	1	4	3	6	3	3	1	1	2	1	2	5	6	3	4	6	6	2	1	5	5
Hour 3	5	5	3	2	2	3	6	5	6	6	2	1	2	3	3	1	2	6	1	5	6	3	1	5	1
Hour 4	5	5	3	2	2	3	6	5	1	6	2	1	2	3	3	1	2	1	4	5	4	3	6	5	1
Hour 5	3	5	3	5	2	2	6	6	1	5	1	1	2	3	6	5	3	1	2	3	4	2	5	6	1
Hour 6	3	5	3	5	2	2	6	6	1	5	1	1	2	3	6	5	3	1	2	3	6	2	5	6	1
Hour 1	2	4	3	5	5	3	2	2	2	2	5	3	1	3	3	1	5	6	4	6	4	1	5	1	1
Hour 2	2	4	3	5	5	3	2	2	2	2	5	3	1	3	3	1	5	6	4	6	4	1	5	1	1
Hour 3	1	2	6	1	1	6	1	2	5	3	5	6	3	3	6	3	3	5	4	2	2	5	1	6	5
Hour 4	1	2	4	1	1	6	1	2	5	3	5	6	3	4	6	3	3	5	3	2	2	5	1	6	5
Hour 5	1	3	4	1	5	3	2	5	3	3	6	4	1	4	1	2	5	2	2	2	5	1	3	5	4
Hour 6	1	3	4	4	5	3	2	5	3	3	6	4	1	6	1	2	5	2	2	2	5	1	3	5	4
Hour 1	1	4	5	4	2	5	3	1	3	5	2	6	2	2	3	4	5	6	1	1	3	2	3	5	6
Hour 2	1	2	5	2	2	5	1	3	6	5	2	6	1	5	1	6	5	2	1	3	3	3	6	3	6
Hour 3	4	2	5	2	2	5	1	3	6	5	1	6	1	5	1	6	5	2	1	3	3	3	6	3	6
Hour 4	4	6	6	2	2	3	2	1	5	5	1	1	3	3	3	5	5	5	1	1	2	3	4	6	6
Hour 5	5	5	2	2	4	1	6	1	5	5	3	1	3	3	3	4	2	5	1	1	2	3	6	6	2
Hour 6	5	5	2	3	4	1	6	1	5	5	3	1	3	2	3	4	2	5	1	1	2	3	6	6	2
Hour 1	1	1	5	3	5	5	4	1	6	3	2	2	3	1	4	3	5	2	5	1	6	3	4	2	2
Hour 2	1	1	5	3	5	5	4	1	6	3	2	2	3	1	4	3	5	2	5	1	6	3	4	2	2
Hour 3	5	6	5	5	3	3	3	2	6	6	2	4	3	1	1	1	1	6	3	5	4	5	1	2	2
Hour 4	5	6	5	5	3	3	3	2	1	6	2	4	3	4	1	1	1	6	3	5	4	5	1	2	2
Hour 5	1	5	5	2	5	6	1	6	1	4	3	2	4	4	6	5	4	1	5	1	4	3	3	3	3
Hour 6	1	5	5	2	5	6	1	6	1	4	3	2	4	4	6	5	4	1	5	1	4	3	3	3	3

Table 16: Summary of reference solution large input.

hours	1	2	3	4	Total blocks	Total hours
1	10	67	17	2	96	203
2	18	72	2	0	92	168
3	36	87	0	0	123	210
4	15	35	1	0	51	88
5	6	69	22	0	97	210
6	25	70	2	0	97	171
Sum	110	400	44	2	556	1050

Table 17: Optimality on cutting plane vs. reference solutions, distributions as in cutting plane algorithm.

	150			155			160		
Name	Max	St.dev	Mean	Max	St.dev	Mean	Max	St.dev	Mean
Normal(int)	42	5.0155	14.6062	47	5.8324	19.2231	58	6.5407	24.2318
N(150)-0	41	5.0512	15.0967	48	5.8815	19.8534	60	6.6106	25.0122
N(150)-24	42	5.0148	14.6053	47	5.8306	19.2214	58	6.5428	24.2266
N(150)-48	42	5.0132	15.7429	49	5.8528	20.5299	60	6.5835	25.6762
N(150)-100	41	5.0513	15.166	47	5.8816	19.9188	58	6.6087	25.0549
N(150)-200	42	4.9939	14.469	48	5.804	19.0035	56	6.4925	23.8889
N(150)-400	42	5.0523	15.1647	47	5.8848	19.9159	58	6.6056	25.0482
N(150)-800	42	4.9766	15.3698	47	5.7965	20.1672	58	6.505	25.3503
N(155)-0	41	4.8664	13.7765	48	5.6714	18.1993	54	6.38	22.9951
N(155)-24	42	4.9574	15.9979	48	5.8358	20.9577	56	6.5573	26.2268
N(155)-100	42	4.8104	15.8434	48	5.6853	20.7643	57	6.4061	25.9982
N(160)-0	41	4.881	14.251	48	5.7055	18.8096	54	6.4139	23.7506
N(160)-24	42	4.8432	14.3189	48	5.6657	18.8754	53	6.3721	23.7787
N(ref.)	48	6.0597	18.2982	58	6.8416	22.729	64	7.4639	27.5197
N(ref.qua)	55	6.4454	23.5485	66	6.9679	27.8029	68	7.3368	32.3257

4.4.2 Comparison of solutions - normal input

The summary of the results is found in tables (17) and (18). The names are as in table (1), but with ref meaning the reference solution and ref.qua meaning the reference solution with quadratic objective (section 3.4).

The tables show the **worst** objective value from the samples, the standard deviation (**st.dev**) and the **mean**. This has been calculated for all samples and all solutions.

4.4.3 Comparison of solutions - large input

The summary of the results are found in table (19), all samples are from the binomial distribution, filtered to extract the samples with total demand 950, 1000, 1050, 1100, 1150 and 1179 hours. Because the maximum number of hours was 1179, this is also what the reference solution was optimized for.

In addition to the tables, we have also included histograms to show more details of the results, not just the mean, worst case and standard deviation. The plots for the test cases 950-1150 (1179 is just a single point) are found in figures (19)-(23).

Table 18: Optimality on cutting plane vs. reference solutions, other distributions.

	190			uniform			binomial 190		
Name	Max	St.dev	Mean	Max	St.dev	Mean	Max	St.dev	Mean
Normal(int)	98	8.3716	62.1015	119	16.9761	54.4077	122	15.3214	62.6615
N(150)-0	101	8.4704	64.0121	123	17.2666	56.0836	124	15.5903	64.5843
N(150)-24	99	8.3702	62.1024	119	16.9729	54.4069	123	15.319	62.6609
N(150)-48	100	8.563	63.3885	120	16.948	55.0981	124	15.3265	63.9162
N(150)-100	99	8.4713	63.5172	121	17.2022	55.5609	124	15.5486	64.1094
N(150)-200	96	8.2853	60.6545	116	16.5463	53.0898	117	14.8872	61.222
N(150)-400	101	8.4773	63.5186	123	17.1919	55.5627	124	15.5445	64.1138
N(150)-800	99	8.3263	64.1199	123	17.2342	56.0675	125	15.5771	64.7093
N(155)-0	95	8.2455	59.2308	113	16.7581	51.8848	115	15.1383	59.7758
N(155)-24	103	8.5509	65.4136	125	17.7563	57.0784	130	16.092	66.0364
N(155)-100	104	8.4518	64.7262	127	18.0158	56.3766	129	16.3359	65.3008
N(160)-0	100	8.3585	61.1695	117	17.3765	53.5395	123	15.7114	61.6831
N(160)-24	96	8.2798	60.6451	116	17.297	53.0629	119	15.6516	61.2161
N(ref.)	101	8.8298	62.8331	115	16.7498	55.248	119	15.2677	63.2885
N(ref.qua)	100	8.0559	65.1561	113	14.523	58.2472	119	13.4363	65.7056

Table 19: Optimality on cutting plane vs. reference, large input

	950			1000			1050		
Name	Max	St.dev	Mean	Max	St.dev	Mean	Max	St.dev	Mean
L(950)	52	4.3074	16.3414	95	9.2535	40.2957	159	13.9631	92.0094
L(1000)	52	4.9809	15.2158	92	10.4993	36.9883	132	12.4063	71.3496
L(1050)	83	9.0418	38.7329	105	11.5113	56.4183	141	12.6039	80.8508
L(ref.)	83	8.8516	41.2878	121	11.2699	67.8522	160	12.2672	99.7554
	1100			1150			1179 (maximum)		
L(950)	229	13.8108	171.6244	311	8.6602	271.7384	336	0	336
L(1000)	183	12.4638	122.5297	237	8.4328	201.2303	257	0	257
L(1050)	173	12.3555	119.6301	212	7.9087	182.1708	230	0	230
L(ref.)	183	11.6021	140.0351	217	7.6496	190.9286	225	0	225

Figure 19: result histogram, large input, KC = 950 hours.

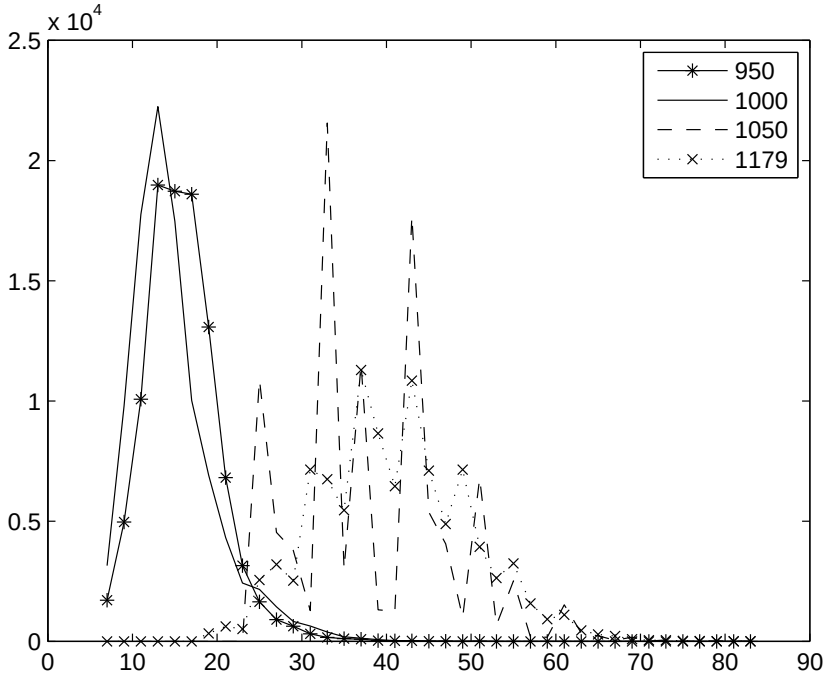


Figure 20: result histogram, large input, KC = 1000 hours.

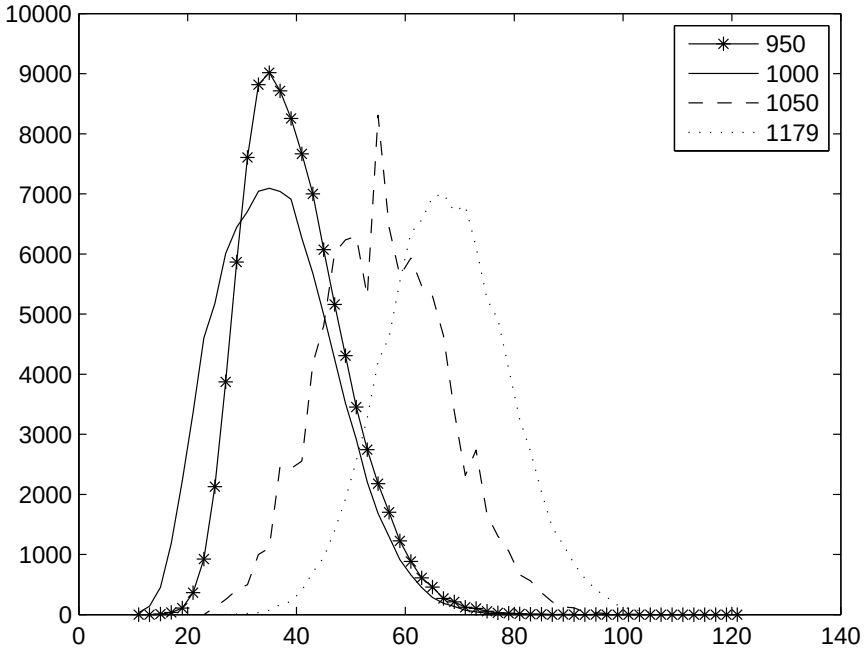


Figure 21: result histogram, large input, KC = 1050 hours.

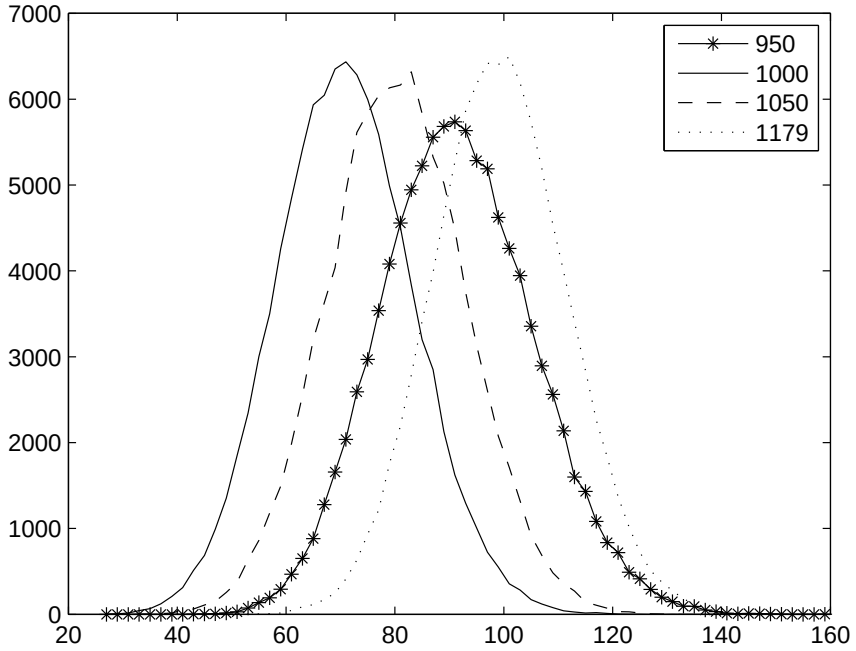


Figure 22: result histogram, large input, KC = 1100 hours.

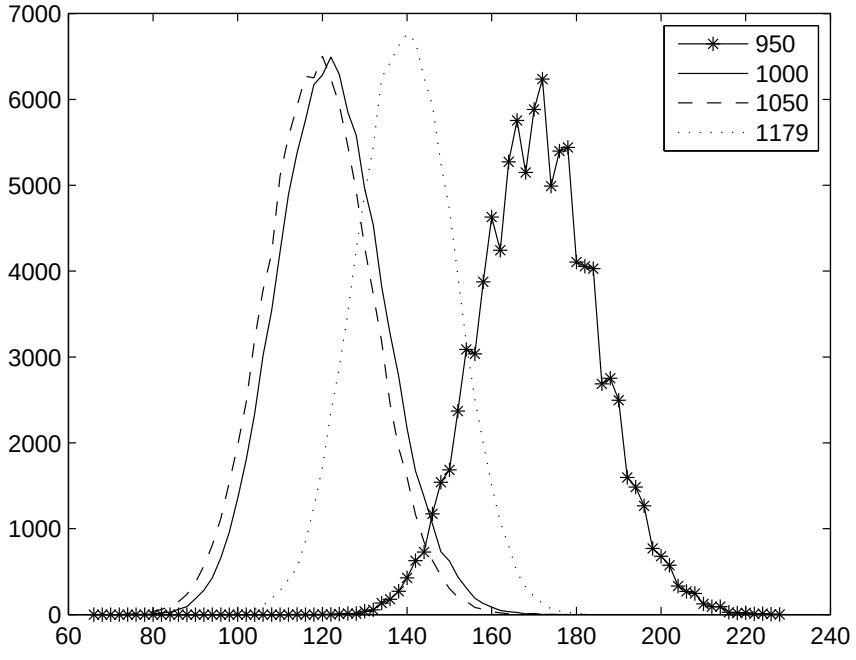
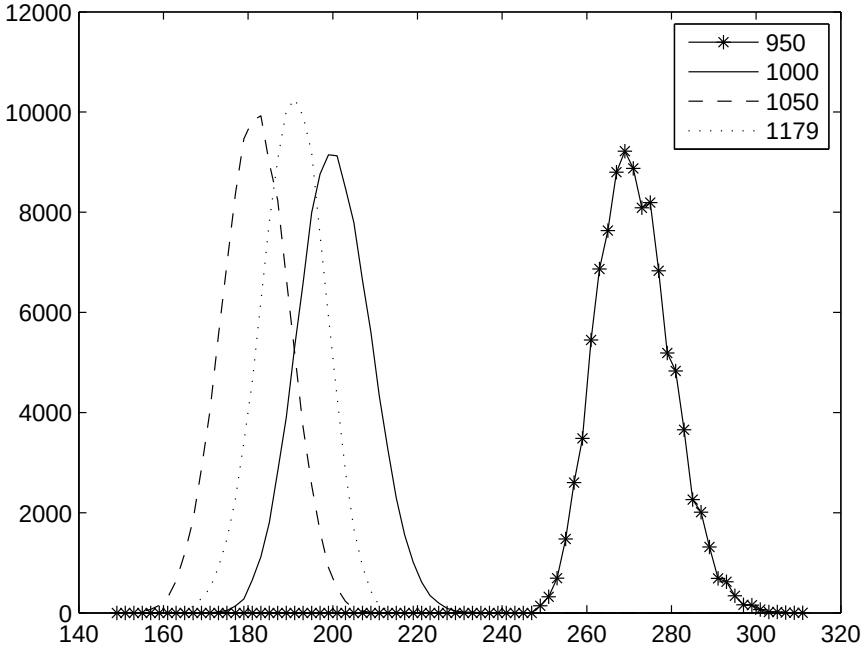


Figure 23: result histogram, large input, KC = 1150 hours.



4.4.4 Optimality of intermediate solutions

The cutting plane algorithm solves MSS each time we run the implementor. Thus we may use one of the intermediate solutions instead of waiting for the algorithm to finish. We tested the 35 intermediate solutions + the final solution from the run on the normal input, with KC = 150 and integrality constraints.

We drew 100,000 samples from the binomial distribution where the total demand was 150 hours, and compared the worst case, mean and standard deviation. The results are shown in table 20. We have also included the **time** until the solution was found and the optimal value of the implementor (**impl.**) and adversary (**adv.**). The worst and mean values are also plotted against the provably worst case found by the adversary in figure (24).

Because we solved the integer problem, all intermediate solutions are feasible. This is not the case with the relaxed problem where we might have solutions with fractional decision variables.

However, because the sum over the groups and lengths $\sum_{p \in P(g,l)} x_p$ is integer, the relaxed solution should not differ too much from an integer solution.

4.4.5 Suboptimal solutions

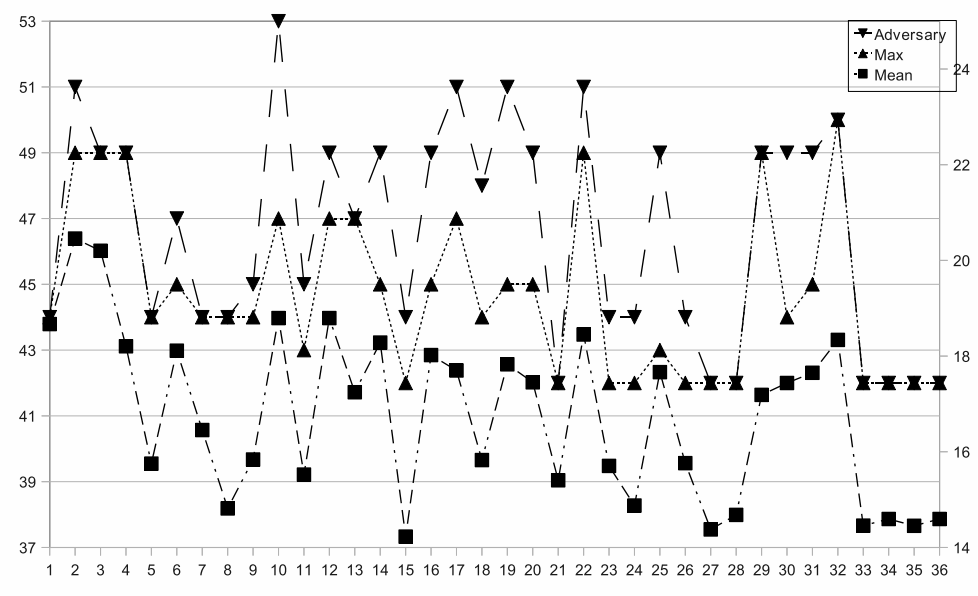
As described in section 3.5.2, there can be dominated solutions with block assignment that exceed the upper bound on the demand. From the normal input with integrality constraints, the intermediate solutions 1,2 and 23 are dominated with 6, 1 and 1 unnecessary block respectively. The normal input with knapsack constraint 150 and head start 800 also has 1 unnecessary block.

On the large input, there were 10, 3 and 1 unnecessary blocks on the problems with knapsack constraint 950, 1000 and 1050 respectively.

Table 20: Statistics on intermediate solutions from cutting plane algorithm

	Time	Impl.	Adv.	Worst	Mean	St.dev.
1	1	0	44	44	18,66865	4,5822
2	2	19	51	49	20,4563	5,864
3	4	22	49	49	20,20062	5,6537
4	6	28	49	49	18,20486	5,7249
5	15	28	44	44	15,75146	4,8323
6	24	31	47	45	18,11227	5,1992
7	25	31	44	44	16,4543	5,0249
8	36	32	44	44	14,81595	4,9915
9	44	33	45	44	15,837	5,1271
10	59	35	53	47	18,79476	5,7315
11	79	36	45	43	15,52263	5,1081
12	97	37	49	47	18,79386	5,8359
13	118	37	47	47	17,24463	5,7035
14	128	37	49	45	18,28234	5,7474
15	137	38	44	42	14,22672	4,9939
16	158	38	49	45	18,02076	5,7066
17	190	39	51	47	17,69959	5,8072
18	211	39	48	44	15,82593	5,1002
19	275	40	51	45	17,82842	5,812
20	282	40	49	45	17,45182	5,7118
21	294	40	42	42	15,40355	4,981
22	299	40	51	49	18,45066	5,7558
23	309	40	44	42	15,70395	5,02
24	316	40	44	42	14,87516	5,0523
25	345	40	49	43	17,66427	5,7218
26	373	40	44	42	15,76221	4,9939
27	415	41	42	42	14,37632	5,0082
28	420	41	42	42	14,67971	4,9892
29	434	41	49	49	17,18925	5,6461
30	449	41	49	44	17,43782	5,7108
31	452	41	49	45	17,65095	5,692
32	462	41	50	50	18,33876	5,7243
33	473	41	42	42	14,45507	5,0024
34	493	41	42	42	14,59415	5,0175
35	508	41	42	42	14,45162	4,9942
36	601	42	42	42	14,59415	5,0175

Figure 24: Optimality analysis. Note that left y-axis is for the adversarial solution and the worst case sample, while the right y-axis is for the average.



5 Discussion

In this section we will comment the results in section 4, in particular, which methods were found to be successful, but also what was found less efficient. Finally we discuss possible future work and draw a conclusion.

5.1 Running time

The results from (4.2) clearly show that the implementor-adversary algorithm is an effective alternative to traditional algorithms, solving large test cases in reasonable time. It was a little surprising to see that the large test input was not much harder to solve than the normal input, even with the 7-fold increase in binary variables. The actual input values had much greater impact on the running time, and we were surprised to see how different knapsack constraints greatly affected running time. After all, the constraint was only used in the adversary, which was extremely fast anyway. The increased number of iterations can explain parts of the increased running time, but not all.

Running time was also influenced by the mathematical model and how it was translated into equations which were sent to CPLEX. An example of this is the room and group constraints, that a room or group can only be used at most once at any time. By using a formulation with pairwise incompatible blocks, the algorithm was very much slower, even though the formulations were mathematically equivalent. Even obvious improvements on the algorithm described in section 3.5.2 had a significant impact on the running time.

5.2 Convergence

Table (1) shows that the number of iterations needed is typically very low. The convergence was even faster on the large test set, suggesting that the number of adversary solutions required are more related to the *dimension* of the adversary solution space, than the actual number of possible solutions. Or more specific, to the number of *vertices in the adversary polytope* (section 1.8.4).

There were some unused group/lenghts in the large problem, thus the dimension was slightly lower than in the normal input. Changing the knapsack constraint greatly affected the number of vertices though, and also had a large effect on convergence.

However, drawing any conclusion is difficult because of the randomness in the algorithm; when facing two or more equivalent solutions, there was no deterministic rule as to which solution should be chosen, neither for the implementor, nor the adversary. And there were few measures taken to avoid dominated strategies (section 1.10). Further testing is needed to determine what really affects convergence and how to improve it.

As expected, adding a head start (section 3.3.5) reduced the number of iterations needed and also increased the objective values of the implementor from the beginning. The latter is useful for obtaining a good lower bound fast. Despite improving convergence, the head start did not boost running times in general. In most cases the algorithm ran considerably longer due to the increased complexity.

While the implementor optimal value clearly converges, there are no indications that the adversarial optimal value does the same. The reason why this is possible is explained in example (1.9). On the other hand, the upper bound (the minimal

adversarial value so far) converged in most cases considerably faster than the lower bound.

5.3 Test analysis

As section 4.4.2 and 4.4.3 show, the cutting plane algorithm yields good results, compared to the reference solutions. Even when demand was not as expected, the algorithm still performed well in many cases. The crucial factor was whether the test scenarios were close *enough* to the adversary solutions used in the algorithm. In particular, section 4.4.3 suggest that it actually can be an advantage to slightly underestimate the adversary. Suboptimality (section 4.4.5) should be taken into account, but the general trend is clear.

Another very interesting result is from table (20) and figure (24). It shows how the mean, worst sample and adversarial optimal value are strongly correlated. The implementor optimal value, on the other hand, had no such relation with respect to the test results.

The strong correlation, together with fast convergence on the upper bound, argues for an early termination of the algorithm, where the best of the intermediate solutions is used.

Furthermore, the final solution was not necessarily optimal, because some intermediate solutions were slightly better.

5.4 Issues

Although the implementor-adversary algorithm was successful, there are issues that should be considered when trying to implement the algorithm. The objective function must reflect the intention of the model. In public healthcare, to limit the worst case is a good objective, but should we also care about the average case, and maybe include it as a secondary objective? The latter requires either a different model, or heuristics on the final solution.

For the algorithm to converge, we need the same objective function in both the adversary and implementor, and care should be taken to ensure this. This is particularly important if the complexity of the objective function increases and we need different implementations (section 1.5).

The adversary solution space is a critical parameter. If there exists one potentially very strong adversary solution, the implementor must use many resources to counter just that one scenario, leaving less resources to other scenarios. For example, if the knapsack constraints are changed as to restrict the number of patients instead of the total number of demand hours, then there exists a very strong scenario with only difficult patients with a potentially large impact on the optimal value, because the function is weighted. To limit the worst case, the implementor must assign many resources to these patients, leading to a skewed solution which on average has a poor objective value.

5.5 Future work

One question we did not have time to investigate deeply is what affects the convergence of the implementor-adversary algorithm. Are there measures we can take to ensure that the number of iterations stay low? Will it always converge fast, also if applied to other models? What characterizes the intermediate solutions, in particular the adversarial

solutions, and can this knowledge be used to improve the head start scenarios? Because convergence is crucial for the feasibility of the algorithm, it is certainly something that should be studied further.

In connection with MSS, it is essential to ask the involved parties at the hospital which objectives should be prioritized. Assumptions that may be obvious for anyone working at a hospital may be unknown to mathematicians implementing the model, and especially to the computer programs solving the model. We have cooperated with hospitals, but need to ensure that the model complies with the intentions of the hospitals. This thesis solves a subproblem of surgery scheduling, namely management of queues, but it could easily be extended to cope with other hospitals wishes.

To limit implementor complexity and improve running time, it might be necessary to replace parts of the strong MIP-formulation with faster heuristics. The tradeoff with heuristics on the implementor are suboptimal solutions and thus false lower bounds. Good heuristics on the adversary may provide the algorithm with a better head start than the current randomly greedy heuristic, with no risk of false bounds.

5.6 Conclusion

The iterative cutting plane method discussed in this paper seems well-capable of handling realistic models from robust surgery scheduling. We expect that the method will prove useful with regard to scheduling problems, either on its own, or in combination with other algorithms. We also expect it to be useful in many other contexts besides scheduling.

6 Bibliography

References

[1] J. Beliën and E. Demeulemeester, *Building cyclic master surgery schedules with leveled resulting bed occupancy*, European Journal of Operational Research, 176 (2007), pp. 1185–1204.

[2] J. Beliën, E. Demeulemeester, and B. Cardoen, *A decision support system for cyclic master surgery scheduling with multiple objectives*, Journal of Scheduling, 12 (2009), pp. 147–161.

[3] A. Ben-tal, A. Goryashko, E. Guslitzer, and A. Nemirovski, *Robust solutions of uncertain linear programs*, Operations Research Letters, 25 (1999), pp. 1–13.

[4] A. Ben-Tal and A. Nemirovski, *Robust convex optimization*, Mathematics of Operations Research, 23 (1998), pp. 769–805.

[5] J. Benders, *Partitioning procedures for solving mixed-variables programming problems*, Computational Management Science, 2 (2005), pp. 3–19.

[6] D. Bertsimas and M. Sim, *Robust discrete optimization and network flows*, Mathematical Programming, 98 (2003), pp. 49–71.

[7] D. Bienstock, *Histogram models for robust portfolio optimization*, Journal of computational finance, 11 (2007), p. 1.

[8] D. Bienstock and N. Özbay, *Computing robust basestock levels*, Discrete Optimization, 5 (2008), pp. 389–414.

[9] J. Birge and F. Louveaux, *Introduction to stochastic programming*, Springer Verlag, 1997.

[10] I. Blöchliger, *Modeling staff scheduling problems. A tutorial*, European Journal of Operational Research, 158 (2004), pp. 533–542.

[11] B. Colson, P. Marcotte, and G. Savard, *Bilevel programming: A survey*, 4OR: A Quarterly Journal of Operations Research, 3 (2005), pp. 87–107.

[12] T. Cormen, *Introduction to algorithms*, The MIT press, 2001.

[13] G. Dahl and C. Mannino, *Notes on combinatorial optimization*, 2009.

[14] S. Dempe, *Bilevel programming-a survey*, Preprint, 11 (2003).

[15] M. Fischetti and M. Monaci, *Light robustness*, Robust and Online Large-Scale Optimization, (2009), pp. 61–84.

[16] J. Halton, *A retrospective and prospective survey of the Monte Carlo method*, Siam review, 12 (1970), pp. 1–63.

[17] E. Hans, G. Wullink, M. van Houdenhoven, and G. Kazemier, *Robust surgery loading*, European Journal of Operational Research, 185 (2008), pp. 1038–1050.

[18] ILOG, *CPLEX version 11.0*.

[19] N. Karmarkar, *A new polynomial-time algorithm for linear programming*, Combinatorica, 4 (1984), pp. 373–395.

[20] V. Klee and G. Minty, *How good is the simplex algorithm*, 1970.

[21] G. Nemhauser and L. Wolsey, *Integer and combinatorial optimization*, Wiley New York, 1999.

[22] A. Soyster, *Convex programming with set-inclusive constraints and applications to inexact linear programming*, Operations Research, 21 (1973), pp. 1154–1157.

[23] D. Spielman and S. Teng, *Why the simplex method usually takes polynomial time*, in Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing, vol. 296, 2001, p. 305.

[24] R. Vanderbei, *Linear programming: foundations and extensions*, Springer Verlag, 2008.