



# Supporting Convenient Application of Linear Programming – Project Report

Bachelor Work  
written by  
**PASTARMOV, Yulian**  
Matr.Nr.:1175918  
Technical University Darmstadt  
Siemens AG - Munich

## Table of Contents

<b>TABLE OF CONTENTS.....</b>	<b>2</b>
<b>1. INTRODUCTION.....</b>	<b>3</b>
1.1 WHAT IS A LINEAR PROGRAM.....	3
1.2 HISTORY OF LPS .....	4
1.3. BASIC PROBLEM SETTING.....	5
1.4 SOLVING LPS.....	6
1.5 SOLVING IPS.....	7
1.5.1 Unimodular Problems.....	8
1.5.2 Branch&Bound .....	8
1.5.3 Non-deterministic approaches.....	9
<b>2 LINEAR PROGRAMMING IN THE WILD.....</b>	<b>11</b>
2.1 LPS AS MATHEMATICAL MODELS OF REAL LIFE.....	11
2.2 SOME EXAMPLES.....	12
<b>3. EXISTING SOLUTIONS FOR SOLVING LPS.....</b>	<b>16</b>
3.1 STRUCTURE OF A SOLVER.....	16
3.1.1 The presolver.....	16
3.1.2 The solver.....	17
3.1.3 The post-solver.....	17
3.2 INPUT AND OUTPUT.....	17
3.2.1 File I/O.....	17
3.2.2 Programming API.....	19
3.3 SHORT REVIEW OF MOST USED SOLVERS.....	19
3.3.1 SoPlex v.1.3.1.....	19
3.3.2 CPLEX v. 10.....	20
<b>4. LP INTERFACE: A GENERIC INTERFACE FOR CONVENIENT APPLICATION OF LP SOLVERS.....</b>	<b>21</b>
4.1 CONCEPT OF A GENERIC INTERFACE.....	21
4.2 IMPLEMENTATION.....	22
4.3 CLASS STRUCTURE OF LP-INTERFACE.....	22
4.4 USER SIDE INTERFACE USAGE.....	23
<b>5. MOPS INTEGRATION.....</b>	<b>24</b>
5.1 INTEGRATION PLAN.....	24
5.2 SPARSE MATRIX STORING TECHNIQUES.....	24
5.2.1 Quadtrees.....	28
5.2.1.1 Inserting Values in a Quadtree.....	30
5.2.1.2 Searching, Retrieving and Deleting of Values in a Quadtree.....	30
5.2.1.3 Implementation details.....	31
5.2.2 Hash lists.....	33
5.3 INTEGRATING THE MATRIX CLASS IN LP-INTERFACE.....	34
5.4 OTHER INTEGRATION ISSUES.....	36
5.5 PERFORMANCE BENCHMARK OF LP-INTERFACE WITH MOPS.....	37
<b>6 CONCLUSION.....</b>	<b>38</b>
<b>REFERNCE.....</b>	<b>39</b>

## 1. Introduction

Main aim of this work is to describe a generic interface to a wide variety of existing commercial and open-source Linear Program Solvers, as well as to discuss in details the work done on integrating one such solver, namely MOPS.

However, before we can even start getting to the details, we should make a short overview of Linear Programming – its definition, application, most common algorithms, some typical problems and their solutions where these exist.

### 1.1 What Is a Linear Program

A Linear Program (LP) (also sometimes called Linear Problem) for the purpose of this work can be defined as a mathematical model of an optimization task, where a minimum or maximum of linear function on real variables is sought, constrained by linear inequalities. These inequalities define some bounds and dependencies between the variables. In other words the general form of one LP is the following:

Minimize or maximize the linear function :  $\mathbf{f}(\mathbf{x}) := \mathbf{c}^T \mathbf{x} : \mathbf{c}, \mathbf{x} \in \mathbb{R}^n$   
 Under the constraints:  $\mathbf{Ax} \leq \mathbf{b} : \mathbf{A} \in \mathbb{R}^{n \times m}; \mathbf{b} \in \mathbb{R}^m$

Where  $\mathbf{x}$  is the vector of variables whose optimum is sought,  $\mathbf{c}$  is the vector of coefficients for the optimization function,  $\mathbf{A}$  is the matrix of coefficients for the set of constraints, and  $\mathbf{b}$  is the right hand side of these constraints. This generic description accepts a lot of improvements and tweaks that make the problem easier to read, or easier for feeding in a computer, but they do not extend its expressive power.

Such enhancements are, for example, constraints of the type  $\mathbf{ax} = \mathbf{b}$  or  $\mathbf{ax} \geq \mathbf{b}$ , or the describing variables bounded in given intervals like  $\mathbf{x}_1 \in [1..10]$  etc. One can easily see that such new constructs are not difficult to write in the standard form of less than or equal inequalities.

One further kind of restriction of LPs is called Integer Programming. These problems have the same form as LPs, but allow only for integer values for optimization variables. Such limitation can be easily overseen as a minor change to the problem setting, however, as we shall later on see, follows to immense increase in the problem difficulty. Nevertheless this category of problems is very important in the practice, because a lot of problems can be described in that way. Often problems of NP-Complexity can be described as IPs. Such problem is for example the question of assigning values to boolean variables such that they fulfill some given equality. As is well known, no polynomial algorithms exist for this set of problems and if such algorithm existed for one of these problems, it will follow that all of them can be solved by polynomial algorithms. This is another indirect proof that solving IPs is also not polynomial problem. Most graph optimization problems are IPs too, like shortest paths or maximal flows, some of these, because of special their properties can be solved as pure LPs as we shall see later on. IP can be the solution of many combinatory problems as well. These normally require heavy brute-force solutions,

but for many of them exist beautiful and fast solutions defined as integer programs. Third category between these also exists – the so called mixed integer programs or MIPs – these have integer restrictions only for a subset of its variables. It is not uncommon to solve pure IPs by relaxing them to MIPs with similar constraints.

## **1.2 History of LPs**

Optimization tasks have been solved from the very beginning of mankind. Even questions of most primitive nature often refer to the search of the best solution, the fastest or cheapest one. The mathematical aspect of their nature has been developed throughout the centuries as mankind has gathered more and more knowledge. Great monuments of the ancient civilizations have certainly involved a lot of mathematical thinking, and precise optimization calculations to yield complex and light constructions capable of withstanding thousands of years under rough meteorological conditions. Therefore one can not give exact date as of when has optimization emerged as a discipline.

The first optimization technique which is known as steepest descent goes back to Gauss. Historically, the first term to be introduced was linear programming, which was invented by George Dantzig in the 1940s. The term “programming” in this context does not refer to computer programming (although computers are nowadays used extensively to solve mathematical programs). Instead, the term comes from the use of program by the United States military to refer to proposed training and logistics schedules, which were the problems that Dantzig was studying at the time. (Additionally, later on, the use of the term "programming" was apparently important for receiving government funding, as it was associated with high-technology research areas that were considered important.)

Linear programming was a mathematical model developed during the second world war to plan expenditures and returns such that it reduces costs to the army and increases losses to the enemy. It was kept secret until 1947. Postwar, many industries found its use in their daily planning.

The founders of the subject are George B. Dantzig, who published the simplex method in 1947, John von Neumann, who developed the theory of the duality in the same year, and Leonid Kantorovich, a Russian mathematician who used similar techniques in economics before Dantzig and won the Nobel prize in 1975 in economics. A second breakthrough came after 1947 when Fiacco and McCormick introduced the Interior Point Method in 1984.

Dantzig's original example of finding the best assignment of 70 people to 70 jobs still explains its success. The computing power required to scan all the permutations to select the best assignment is vast and impossible. He observed that it takes only a moment to find the optimum solution using the simplex method, which is effectively noticing that a solution exists in the corners of the polygon described by the equations formed from the given constraints.

### 1.3. Basic Problem Setting

Let's get back to the mathematical aspect of describing optimization problems. We shall examine one such simple model and look at its exact mathematical description.

Let the target function be  $f(\mathbf{x}_1, \mathbf{x}_2) := 3\mathbf{x}_1 + 5\mathbf{x}_2$ . What we search is its minimum, under the conditions that  $\mathbf{x}_1$  can only be in the interval  $[3, 10]$  and  $\mathbf{x}_2$  has to be not smaller than 2 times  $\mathbf{x}_1$ . This verbal description leads to the following formal description:

$$\begin{aligned} \text{MINIMIZE: } & (3, 5) * \mathbf{x} \\ \text{SUBJECT TO: } & \\ & 3 \leq \mathbf{x}_1 \leq 10 \\ & 2\mathbf{x}_1 \geq \mathbf{x}_2 \end{aligned}$$

Written in standard form:

$$\begin{aligned} \text{MINIMIZE: } & (3, 5) * \mathbf{x} \\ \text{SUBJECT TO: } & \\ & -\mathbf{x}_1 \leq -3 \\ & \mathbf{x}_1 \leq 10 \\ & \mathbf{x}_2 - 2\mathbf{x}_1 \leq 0 \end{aligned}$$

It is often easier to analyze problems when they have some graphical representation. Such representation exists for optimization problems too. However it depends on the number of variables whether our two-dimensional paper would allow us to draw it. For every variable requires new coordinate axis in our drawing. The given problem on the other hand has only two variables so the plane will be quite sufficient to represent it graphically.

In order to study the problem easier we shall draw the constraints as half-planes on the coordinate system defined by their variables (See Fig.1). After that we will see how to interpret the target function as well.

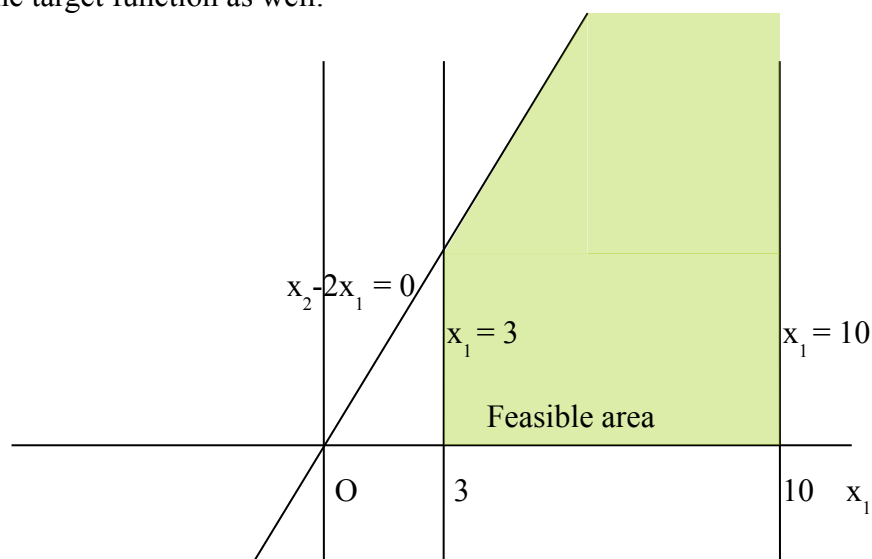


Fig.1 The feasible area of the simple problem given

As we can see constraints define half-planes that cover the allowed areas where the optimum have to be sought. The intersection of these half-planes is called “feasible area” and can be a closed or opened convex polygon, or a single axis or point, or even empty. For proof of this statement see [ScriptOpti]. If the intersection of the half-planes is not empty a solution to the optimization problem can be found or else one says that the problem has no solution. In case there is no boundary in the direction of increment/decrement of the target function one says the problem is unbounded and the solution is  $\pm\text{INF}$ . Therefore solving one LP consists of either pointing out a solution for it, or to determine that such solution does not exist.

The optimization or objective function can be represented as a vector in the plane that shows the direction of increment or decrement of the target function. All points on a line perpendicular to that vector lead to the same value of the target function. Hence, one can imagine sweeping this line in the direction of the target vector and a last point of intersection between the line and the polygon is the point where optimum is achieved. Therefore optimal values are always achieved on the bounds of the problem. Even in case where there is no unique optimum (a bound condition is perpendicular to the target vector, one of its intersection points with other bound constraints is taken to be the solution. Proof of these statements, as well as their further discussion goes well beyond the reach of this work. Further explanations can be found in [ScriptOpti].

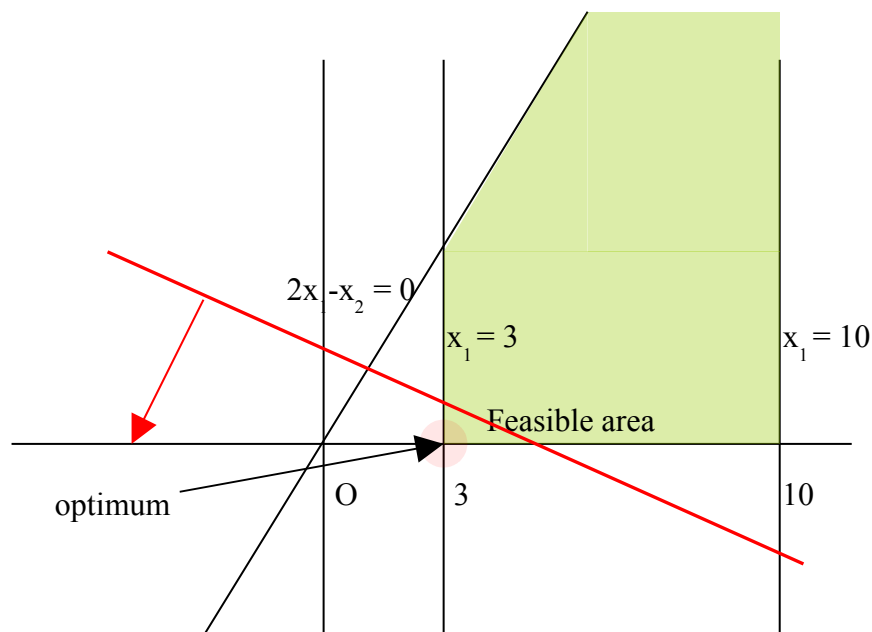


Fig.2 The target function and its increasing direction vector

## 1.4 Solving LPs

Now we have seen that optimal values lay on the boundaries of polyhedra describing the feasible area, next question is how to easily find these boundaries and more precisely the intersection points or corners of these polyhedra. Later on one would like to traverse them in such a way that each step gives a better solution than the

previous one. This will ensure optimality of the solution in case such exists. On the other hand another idea to ensure that optimum has been found is to check all possible solutions and select the best one. Although the traversal of all corners of the feasible area is theoretically possible, such solution is often practically impossible because of the size of the problems solved. Often a solution based on checking all possible values results in programs that can run years on contemporary computers without yielding any solution.

Luckily algorithms have been invented that can relatively fast find optimal solutions by starting from one feasible solution and traversing the set of possible values towards better ones until they reach optimum.

The main algorithm that uses this schema is called the Simplex method. Actually, it is called a method and not algorithm, because it only prescribes general approach of dealing with the problem. Hence, not one but many algorithms exist based on the Simplex method, generally divided in two groups: Primal Simplex Algorithms and Dual Simplex Algorithms. Exact description of these goes beyond the reach of this work. It is however important that the reader is accustomed with the rough structure and differences between these.

For the sake of generality we shall mention the existence of other solving techniques like the Internal Point Method. This one does not traverse possible solutions one by one, but tries to find always smaller ellipses around the optimum, thus eventually yielding optimal solution with the precision needed.

## **1.5 Solving IPs**

Integer Programs are often mistaken as easier than LPs. Mainly because people are used to thinking of working with integers being easier than with real numbers. From mathematical point of view, however, integers present some big difficulties. One of the easiest to comprehend, being the fact that one can not always divide integers and get integer solution. Moreover we mentioned that we can represent constraints of LPs as lines or hyper-planes in the coordinate system of the problem. Lines or planes, however, do not represent discrete objects, they don't disappear between integers and one can easily see that two lines, even though described by integer coefficients, do not always cross in integer point (See fig.3). So the main assumption of all simplex algorithms does not hold anymore and therefore we can not rely on this method for solving IPs. What comes as an idea is to shrink somehow the polyhedron to only include its integer points and then we can use the simplex method on this new polyhedron. Shrinking is not an easy operation, though, and requires a lot of heavy numeric computations. Actually it counts to the group of NP-Complete problems. Very short and rough description of some of the methods to deal with integer problems will be presented in this chapter ahead.

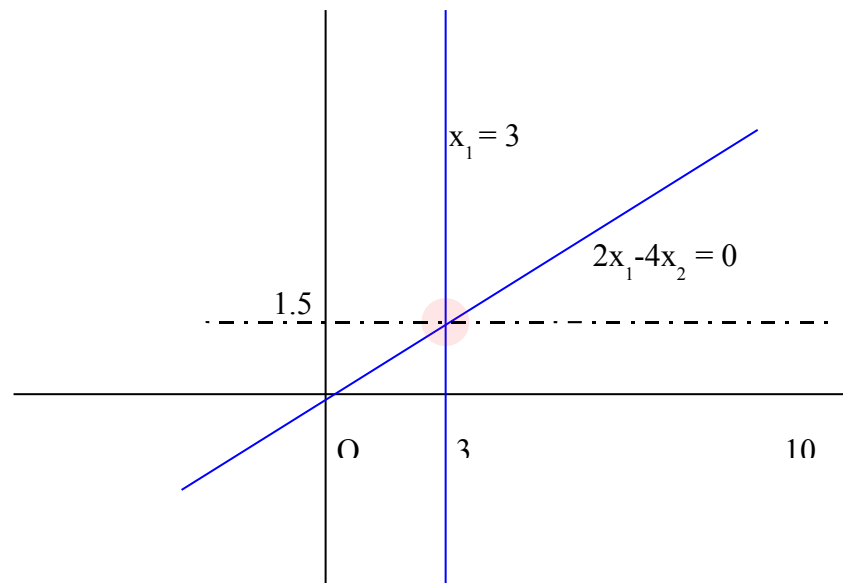


Fig.3 Lines with integer coefficients crossing at non integer point

### 1.5.1 Unimodular Problems

There is an important category of integer optimization problems that can be solved as if they were standard LPs. This is due to the fact that their bound conditions do cross in integer points only. This guarantee exists for the so-called unimodular matrices. A matrix is called unimodular when all its submatrices have determinant  $\pm 1$  or 0. A matrix is called totally unimodular if all its submatrices have determinant  $+1$  or  $-1$ . Such problems that can be defined in terms of totally unimodular matrices can be solved with the simplex method described in the previous chapter.

### 1.5.2 Branch&Bound

Branch and bound algorithm try to separate the feasible area in non-overlapping regions and search consecutively in both. By minimization problems, using some techniques for lower bound approximation, one can cut whole branches of that tree if no better solution can be found than the one already present. One of the most important techniques for lower bounding is called LP-Relaxation. It consists of replacing some integer constraints by new terms in the target function. More about LP-Relaxation can be read in any book on optimization [ScriptOpti2]. Different branch&bound methods differ in the way they do the branching. Some do depth-first-search while others do breath-first-search. Mixed variants are ones that change between both techniques. In the proper balance between them is hidden the key to fast and precise solvers. This is the method used by most IP solvers.

### 1.5.3 Non-deterministic approaches

Sometimes exact solutions are not feasible, if the problem has nonunimodular constraints and consists of large number of variables. In these cases another techniques can be exploited. These can not always guarantee optimality of their solution, but can give good approximation in short time. Such are Greedy algorithms or various Dynamic Programming schemas. These are very well studied and can be found in almost any book on algorithms.

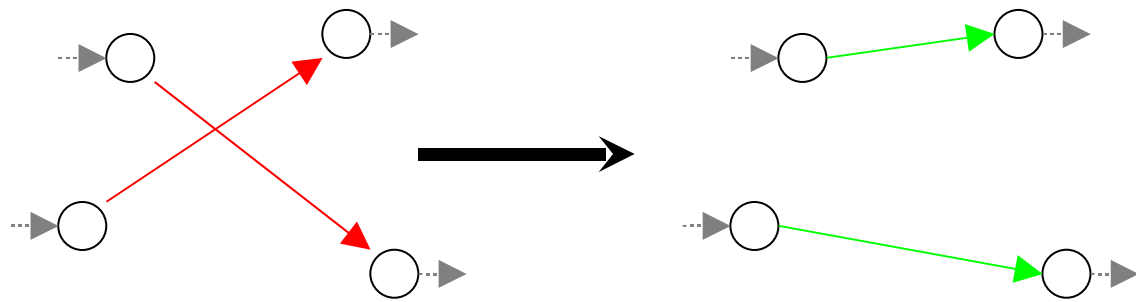
One of the newer approaches is called Genetic Programming. It has got its name because it mimics the way evolution is being done in nature. Strong breeds survive, weak die out. Species mutate over time and exchange genetic code between them, hence changing to better all the time. This approach can be mainly summarized in five steps.

1. Create a set of starting solutions;
2. Create some new solutions by mixing parts from existing ones;
3. Mutate solutions by randomly changing parts of them;
4. Select the best solutions from the newly created (and possibly from the old ones as well);
5. Repeat steps 2 to 4 until acceptable solution has been found.

A bit delicate part is how can one “mix” and “mutate” solutions as to keep them valid. The answer to these questions depends on the problem and can only be defined when very good understanding of the task has been achieved. Even then not every problem accepts such approach. Another tricky part is when to stop. As we said we get no guarantee whatsoever, how good the achieved solution is. What one normally has is a so called fitting function that provides some measure of how good is a given solution. However such functions are difficult to define and in some cases even impossible. Sometimes some fictive best solution boundary can be computed using dual methods, for example

As an example we can give the traveling salesman problem. Given a set of cities, a salesman has to travel between all of them visiting each one exactly once and returning to its own city at the end. How can he achieve this on the shortest possible path?

One starts with a set of generated valid paths. Mixing paths would mean: find two paths, select non crossing parts of them and replace them mutually. An example for mutation (that even results in better results always is) find two edges in the path that cross and replace them by fixing the crossing as represented fig.4. Following this steps one can relatively fast find some good path, but not necessarily the best one!



*Fig.4 Fixing crossing paths in TSP Problem (an example for mutation in genetic programming)*

Another technique in this group is called Simulated Annealing. But we shall only mention its name here and not give precise description of its operation [ScriptOpti2].

## 2 Linear Programming in the Wild

Linear programming is an important field of optimization for several reasons. Many practical problems in operations research can be expressed as linear programming problems. Certain special cases of linear programming, such as network flow problems and multicommodity flow problems are considered important enough to have generated much research on specialized algorithms for their solution. A number of algorithms for other types of optimization problems work by solving LP problems as sub-problems. Historically, ideas from linear programming have inspired many of the central concepts of optimization theory, such as duality, decomposition, and the importance of convexity and its generalizations. Likewise, linear programming is heavily used in microeconomics and business management, either to maximize the income or minimize the costs of a production scheme. Some examples are food blending, inventory management, portfolio and finance management, resource allocation for human and machine resources, planning advertisement campaign etc.

### ***2.1 LPs As Mathematical Models Of Real Life***

Often the process of defining LPs is being called modeling. Modeling is the process of describing parts of the real world in mathematical means. To create a mathematical model one needs detailed knowledge of the exact dependencies and limits that nature or other laws impose on objects. Once armed with this knowledge, one can proceed to express it in the language of mathematics. It might sometimes seem difficult to express some aspects in terms of formulas. In Optimization, modeling means to clearly and unambiguously define the target of the optimization process as a function of the parameters it depends on. First of all find these parameters that play role in defining the target. Analyze the constraints on these parameters and their connections with each other. If the dependencies and target prove to have only linear dependencies or at least allow for such approximation, then the data collected can then be represented in the way of LP. One defines the target as a function of its parameters and defines the constraints and dependencies between these as restrictions for the LP in the form given in chapter 1.1.

Often people create models and solve them only to find flaws in their definitions. These can be: wrong target functions, described by insufficient number of parameters or wrong coefficients for example. Or find that constraints given are too restrictive, even ones that give no solution at all. Sometimes the model is correct, but does not yield the expected results and new considerations about changing the real world's objects or processes have to be done. Often these tests, whether such changes lead to better results, are done on the mathematical model itself and only after satisfactory results are achieved, they are propagated in the real world. This is another reason, why the process is called modeling. It gives a chance for dramatically reducing the cost of many activities by providing a way to test the results of changes and transformations, without having to perform these operations on the reality's counterparts to mathematical models.

## 2.2 Some examples

In this chapter we will give some examples, which shall lead us to the conclusion that our life would not be the same without the large effort, put in designing better approaches to solving optimization problems. These play more and more important role in our life with every day to come. Problems not only get bigger, but they often get fuzzier. This means they can not be easily described with precise formulas, but require testing of many similar definitions. We shall make these problems clear by presenting some examples of optimization problems.

Linear Programming plays vital role in logistics. One of the most common examples is the so called transportation problem:

A company owns a given number of store-houses and shops. It also has a given amount of vehicles to transport goods from store-houses to shops. Given the amounts of goods and the needs of all shops, find the optimal assignments for the drivers in form of routes, that they should travel, in order to supply all shops with everything needed, in the cheapest possible way.

One can see that this problem is too huge for manual solution even when the number of items involved is small. What can we say if we are to optimize manually for big cash-and-carry companies like METRO for example, with thousands of shops and store-houses Europe-wide. Problems of this size can be a challenge even for today's computers.

Even on microscopic scale there exist large optimization problems. Most contemporary computers have out-of-order instructions' execution. This means they can reorder the operations they are fed, such that they can execute them faster. Whatever they do, they should maintain the original meaning of these instructions in their intended order. This is again one optimization task, (partially) solved milliards of times per second by any running computer.

Such examples one can easily read in any book about optimization. What might raise more interest in the field is the following example from the set of problems solved at the Discrete Optimization Department of Siemens AG.

Planning water supply system's operation of big cities can be a very difficult task. A lot of factors play different role in the planning process. People that do this have to consider the needs of water throughout the day and cover usage peaks that have been statistically proven. On the other hand water is being pumped in containers distributed at different points around the city, and these have only bounded capacity. Last but not least energy costs for operating the system should be kept as low as possible. All these conditions have to be considered and balanced throughout the planning process(See fig.5).

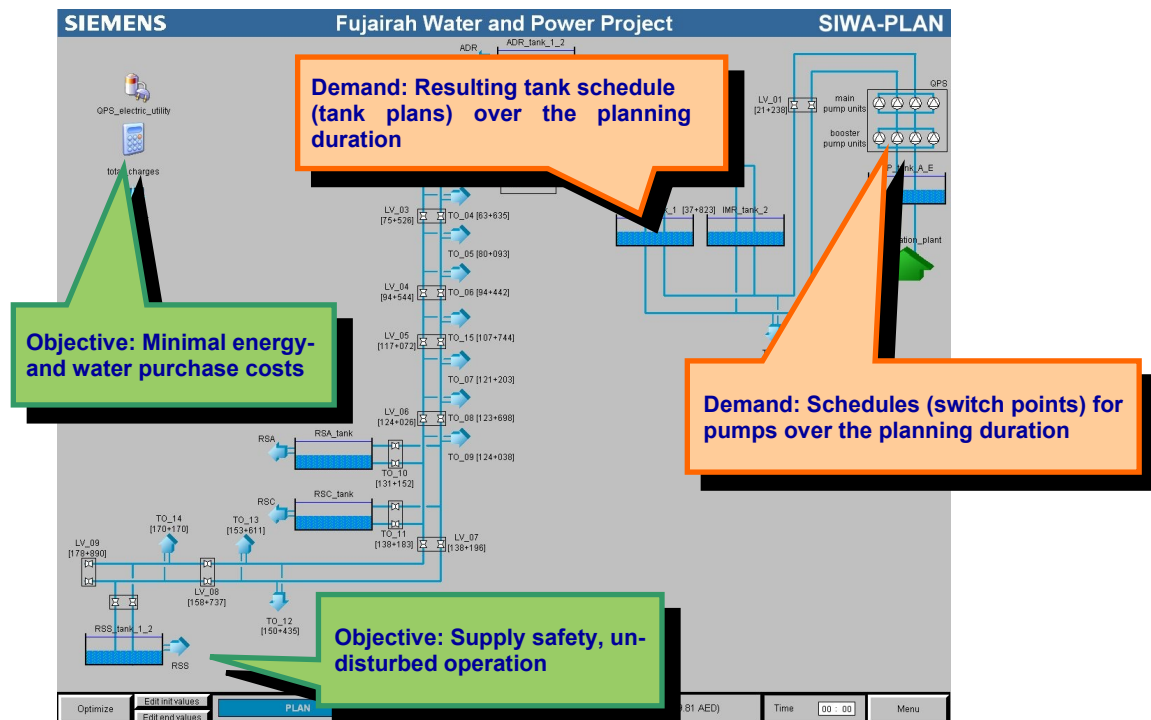


Fig. 5 SIWAPlan parameters and objectives

One possible way of dealing with the problem is to use a simulator, where the real system is defined as a computer model. People can change parameters of different aspects of that system and then test-run it to see the effect of their changes. This step is repeated until acceptable results are achieved. Such approach can be defined as semi-automatic because the system can not find optimal parameters on its own. Nevertheless simulators give the best possible approximation of real-world system. They can present accurate physical model of water flow at any given moment of time which is sometimes a must for technicians and engineers of such facilities.

Using simulator to find optimal parameters can be very lengthy process. Most system water supply systems consist of many pumps and reservoirs. These are being supplied with water and energy from different sources and these can vary in capacity and price too. Therefore it is difficult to give a prognosis what would happen after changes are done. Hence lots of unsuccessful tries have to be done until good results are achieved.

It would be a lot better if the computer provided it has the model of a real system, can compute optimal parameters on its own and even use them automatically to control the pumps. Such system is SIWAPlan's Optim Module implemented at Siemens AG.

First we'll discuss the most important parameters that define the behavior of one water supply system, afterwards a description of how these parameters cooperate will be given and some aspects of how to build these as LP will be presented.

First in the system come water supply points. These are either water basins or just joint points to other water supplier. Their parameters are minimal and maximal water flow per time unit and the cost for one unit. The same parameters apply for energy suppliers as these are equivalent in nature. Next in the system come pump stations. They have one or more pumps that can be operated either together or separate to cover the needs with minimal energy consumption. These have as main parameters the

amount of energy used to operate the pumps and the minimal and maximal water flow per time unit achievable. Moreover water-flow is not linearly proportional to the number of pumps running because physically colliding water streams slow-down each other. Lastly, water containers are defined by their maximal capacity as well as minimal and maximal input and output flows.

What one real system also has is a number of points where water is being consumed. These are being parameterized by the amount of water used at different moments in time. Such parameters however would lead to differential equations and prevent us from solving the problem as LP. Therefore another approach is used. First water consumption is being modeled as a condition over the minimal amount of water in reservoirs throughout time. Then each time slice is modeled as modified flow problem and time slots are then connected with additional constraints that logically bind them, where consequent time slots depend on previous ones (see fig.6).

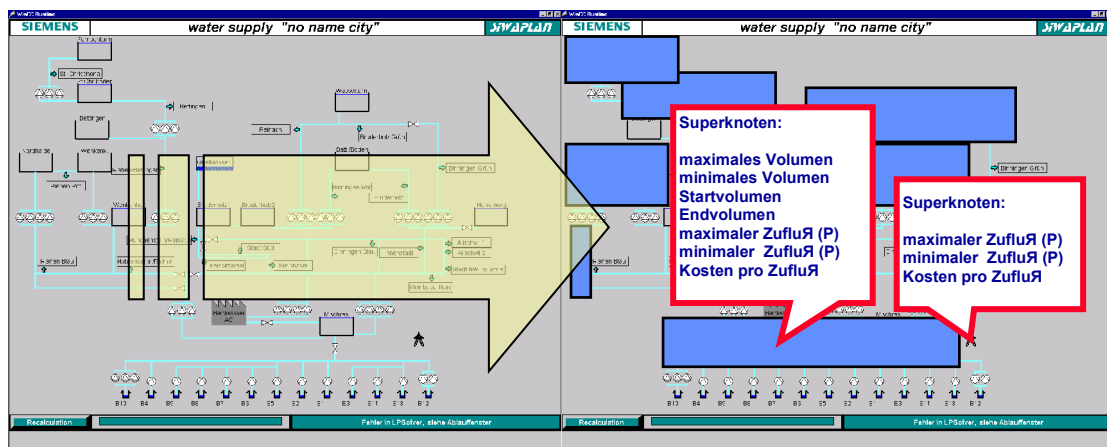


Fig. 6 Simplifying the water supply system to make it LP-able

Let's have a bit closer look how does one LP System for a standard maximal flow problem look like.

Given a graph  $G = (V, A)$ , a capacity function  $c : A \rightarrow Z^+$ , a source node  $s$  and a sink node  $t$ , the maximum flow problem is to find the maximum flow from  $s$  to  $t$  without violating the capacity constraints. This can be formulated as a linear program where the variables are the flow on arc  $(i, j)$ , denoted by  $x_{i,j}$ . We add a circulation arc from  $t$  to  $s$  of infinite capacity, and force conservation (inflow = outflow) at all nodes, including  $s$  and  $t$ . Maximizing the  $s - t$  flow through  $G$  is now equivalent to maximizing the flow from  $t$  to  $s$  on the circulation arc. The LP formulation can now be stated as follows.

**Maximize:**

$$1^T x_{t,s}$$

**subject to:**

$$\sum_{i \in V: (i,j) \in A} x_{i,j} - \sum_{k \in V: (j,k) \in A} x_{j,k} = 0 \quad \forall j \in V \quad (7.1)$$

$$x_{i,j} \leq c(i,j) \quad \forall (i,j) \in V \quad (7.2)$$

$$x_{i,j} \geq 0 \quad \forall (i,j) \in V \quad (7.3)$$

The LP formulation of the maximum flow problem is such that every variable appears at most twice in the flow balance constraints - once with a coefficient of +1 (going "into" the head of the arc) and once with a coefficient of -1 (going "out" of the tail). We do not consider the capacity constraints to be an explicit part of the constraint matrix as these constraints can be handled easily - special techniques exist to solve LPs with lower and upper bound restrictions on the variables without treating them as explicit constraints. Therefore, the constraint matrix (only equations 7.2) is a totally unimodular matrix.

It is not difficult to extend the problem to multi-source multi-sink problem. Standard approach is to define a super-source and super-sink nodes that can provide infinite capacity. These are then connected before all sources and after all sinks respectively. This doesn't change the presented equations. Next to extend the model to support time slicing one have to imagine connecting consequently all nodes of the graph in to their equivalent nodes in the next time slice. It can be imagined as a layered structure with time as the third layer dimension.

Of course the presented structure is slightly simplified version of all the conditions that have been implemented in SIWAPlan in order the software to produce useful results that can be practically used. However the example presents some of the most important aspects of real-world LP programming. Problems often do not consist of only one type of mathematical problem (like graph-flow or pure transportation problem) but often engage different types that mix together and cooperate to model complex systems. On the other hand linear and discrete optimization techniques are powerful enough to describe many problems correctly enough to be used in a lot of fields in science and industry. As we have even seen – problems that might seem to only be describable in terms of differential equations can after preprocessing and modification be described in terms of LPs too.

It is interesting to know that LP-Interface described in this work has been based on classes designed and written for SIWAPlan.

### 3. Existing Solutions for Solving LPs

Now that we know how important optimization tasks are and we revised the most often used approaches of dealing with them, we can review some of the existing software solutions for dealing with LP and IP problems. Typically such programs are called *LP solvers*. A lot of solvers have been developed throughout the years. Some of them are commercial like CPLEX, XPressMP and MOPS, others are open source free solutions like SoPlex and GLPK for example.

#### 3.1 Structure of a solver

These solvers work generally in the same way. First they analyze the problem, try to reduce the constraints or optimize them, as well as reduce the number of variables or their restrictions. This process is known as simplification or pre-solving. Afterwards they perform the actual solving using either simplex based algorithm for LPs or branch-and-bound algorithms for IPs. In the end they perform some finalization steps known as post-solving.

##### 3.1.1 The presolver

The first step is known as presolving. It plays a vital role for the overall performance of the solver. The better the presolver is the fewer data has the solver to deal with. On the other hand it is important that the presolver keeps track of its changes on the model because normally problems get changed in such a way that it is not possible to recognize the original variables afterwards. It is even not uncommon that the presolver deletes all variables for example if it finds a way to compute the optimum itself.

Pre-solvers are important part of solvers which is being actively developed and improved nowadays too. New algorithms are developed that enable pre-solvers to better reduce problems and at the same time keep the structure of the problem suitable for large variety of solving algorithms.

In a paper of 1975, Brearley, Mitra, and Williams [Presolvers] discuss presolving linear programming problems. They recommend recursively:

- folding singleton rows into bounds on the variables
- omitting inequalities that will always be slack
- deducing bounds from constraints that involve several bounded variables
- deducing bounds on dual variables

SoPlex for example has still some problems with its pre-solver. In version 1.2.2 the pre-solver often changes the problem in such a way that no dual values can be computed anymore. So if someone needed these the pre-solver has to be turned off. In version 1.3.1 on the other hand exists some other problem with the pre-solver leading to incorrect optimal values for some problems.

### 3.1.2 The solver

The solver module is the heart of all optimization programs. It can find an optimal solution or report the absence of one. Some solvers can only deal with real-number optimization problems; other can handle both LPs and IPs. Most use simplex primal and dual methods for solving LPs and Branch&Bound techniques for dealing with IPs. The differences in their speed and applicability are normally hidden in the different solutions chosen by these solvers. MOPS's developers claim, that it has one of the fastest branch and bound algorithms among solvers.

### 3.1.3 The post-solver

Post-solvers are used to reconstruct the problem in its original form. As we said the pre-solver often changes considerably the model. Therefore this last step is important if the solver wants to communicate its complete results back to the user in a form that corresponds to original model

What we cover in this chapter are only details concerning the way solvers work to deal with the actual problem. None of the aspects concerning memory management or multithreading are covered mostly because these are not relevant to this report and are field where most solvers differ a lot. Such design decisions are important but not constrained to the field of solvers. Therefore one is encouraged to read more about the theme in books about software design and data processing.

## 3.2 *Input and Output*

Most solvers have both file I/O and programming API. This means they can read problem descriptions and settings from files and save their results in files, but they can as well be called from inside other programs and communicate only through calls to functions defining their interface, exchanging data through the operating memory.

### 3.2.1 File I/O

Different solutions often provide their own file formats as well. These correspond tight to the way data is organized internally in the solver and lead to faster load times or better parsing of the problem. A number of standard formats exist as well. One of these is called MPS format. It is not one of the easiest to read from people, but is well suited for machine parsing. MPS is practically standard now for input format, known to most solvers used today.

We will not present the complete syntax of MPS files, but an example for one such file is presented below.

Let the problem be:

$$\begin{aligned} \text{Minimize: } & 2x + 3y \\ \text{Subject to: } & x + y \leq 6 \\ & x, y \geq 0 \end{aligned}$$

In format MPS this problem looks this way:

```
NAME example
ROWS
  N      OBJECTIV
  L      c1
COLUMNS
  x      OBJECTIV      2      c1      1
  y      OBJECTIV      3      c1      1
RHS
  RHS    c1            6
BOUNDS
  LO BND x            0
  LO BND y            0
ENDDATA
```

Another very popular format is LP. It has more readable form, close to the natural mathematical definition. The same problem looks like this when described in LP file:

```
Minimize
  cost: +2 x +3 y
Subject to
  c1: +1 x +1 y <= 6
End
```

There are tools, similar to compilers that can read some file formats and convert them to another. These allow for defining problems in formats more convenient than MPS and later converting them to this format, known by almost all solvers. One such tool is ZIMPL [ZIMPL]. Again discussion of this particular file format is beyond the reach of this work, and only an example will be provided. One can see that it is a lot easier to read and understand ZIMPL files than MPS files. Moreover as this format is preprocessed it supports some higher level semantics that can be very helpful for describing complex models. Some of these constructions are sets (and named sets), loops and primitive conditional statements.

The same problem written as ZIMPL file:

```
var x;
var y;
minimize cost: 2 * x + 3 * y;
subto c1: x + y <= 6 # lower bounds are 0 per default
```

At last some examples about the advanced semantics of ZIMPL :

```
#define some sets
set I := { 1 .. 10 };
set J := { "a", "b", "c", "x", "y", "z" };
#define a table of parameters
param h[I*J] := | "a", "c", "x", "z" |
  |1| 12, 17, 99, 23 |
  |3| 4, 3, -17, 66*5.5 |
  |5| 2/3, -.4, 3, abs(-4)|
  |9| 1, 2, 0, 3 | default -99;
#define some constraint with a forall cycle and if construct in
subto c1: forall <i> in I do
  if (i mod 2 == 0) then 3 * x[i] >= 4
  else -2 * y[i] <= 3 end;
#define another with a sum construct
subto c2: sum <i> in I :
  if (i mod 2 == 0) then 3 * x[i] else -2 * y[i] end <= 3;
```

### 3.2.2 Programming API

If programs had to communicate using files, this would slow immensely their work. Normally disk drives or any other type of external memory is a lot slower than operating memory. This is the reason, why most solvers have their Application Programming Interface (APIs) that can be used from third programs to expose their functionality without the need to use files to communicate with each other.

These APIs consist of functions for feeding data about variables, constraints and parameters for the solvers, functions to trigger the solving, as well as means for reading back the results. This general description fits roughly all solvers, but the design and implementation details differ a lot. Therefore most users decide for one given solver, and specialize in its particular API. This gives good understanding and tight control over the chosen solver, but it can prove very difficult to change to other solvers, if the one of choice gets discontinued for example.

That is where the idea to have a standard interface and API to many different solvers comes to spare a lot of time and effort, and in the same time offer more flexibility of choice. However before we present one such interface, we shall make a short review of the most used solvers, their advantages and disadvantages in order to gain some better understanding of the state of existing software.

### 3.3 Short Review of Most Used Solvers

A lot of solver programs exist. As already mentioned some are free open source projects, others are commercial solutions. Most solvers are being developed at different universities around the world, some have been taken by companies and are now offered with full customer support but are generally expensive, as these need a lot of highly qualified work to maintain and update. Examples for open source solvers are SoPlex and CLP, whereas XPressMP and CPLEX are commercial solutions.

As we said commercial solutions are sometimes preferred in big companies because support is offered on a well defined basis, however these programs are normally delivered only in binary form and no changes can be made to the way they work. On the other hand open source solvers are given in their original form and it is completely legal to modify them to fit the needs at hand.

We shall review shortly SoPlex as a member of the free solvers group and CPLEX as a commercial solution.

#### 3.3.1 SoPlex v.1.3.1

SoPlex is one of the best equipped free solvers. It supports both LPs and MIPs and possesses fast solving algorithm. What makes it a very good choice is that it is being actively developed and most bug reports and feature requests are considered promptly

and reliably. However SoPlex have still some problems mostly with its pre-solver that are known and work is being done to fix them. The latest released version of SoPlex is 1.3.1, present in two versions one which handles error conditions with asserts at debug time and interrupts from severe errors with abort calls the whole application and one that uses exceptions to handle error conditions.

This Solver comes as a set of classes that have well defined interface to be used as a library from third programs, never the less source code of a pre-built solver is present too. Therefore one can compile and use the solver as standalone application that can read LPs and MIPs from files in MPS and LP format.

The library interface of SoPlex presents both classes for interfacing solver's logic and its data structures in a convenient way. One can modify and solve given model many times without redefining it completely in-between. Therefore the solver fits well in the LP-Interface library that we'll discuss in the next chapter. As a result of a feature-request, exception error handling have been integrated which makes the usage of the library a lot easier and safer from third applications, because the legacy behavior of aborting the application at severe errors can often be unwanted (For example often solvers are only a small part of bigger systems and it is not expected that error in one module bring the whole system down). Exceptions provide structured way to analyze the error condition and provide meaningful feedback to user as well as ensure the overall stability of the system.

One of the most annoying problems still in SoPlex 1.3.1 is that the pre-solver sometimes removes variables and conditions but do not correctly update indexes in target function or other conditions and this leads to incorrect optimization results. A second problem of the pre-solver is that when used no dual values can be computed anymore.

### 3.3.2 CPLEX v. 10

CPLEX is a product of ILOG Company since its 6<sup>th</sup> Version. The most recent version is its 10<sup>th</sup> release.

This solver is considerably faster than SoPlex and has much better data structure for its internal model representation that needs less space in main memory and faster insert and lookup times. However this fact has been proven only by empirical tests made, as no details are present in the documentation to the solver.

CPLEX is being delivered only as dynamic link library with strong mechanisms for preventing unauthorized usage. Header files and lib files for integrating the solver are provided too. Extensive documentation is provided for all version of CPLEX. Not only functions of the API interface are described but a lot of attention is paid to give hints for the effective usage of the library and correctness of the results.

CPLEX v.10 uses exceptions to detect and report error conditions and provides functionality to analyze infeasibility conditions. It is one of the few solvers that pass all tests written to check the compatibility and functionality of different solvers from inside LP-Interface's framework.

## **4. LP Interface: A Generic Interface for convenient application of LP solvers**

As we mentioned in our short review of existing solvers, one main problem in scientific work is migration between solvers. All solvers have their peculiarities and special ways of feeding data and reading their output. This leads to confusion among users of different solvers when they have to work together or change between solvers. Solvers' programmers on the other hand are not very willing to change their interfaces, because they have different ways of seeing the problem and prefer interfaces that most closely resemble internal structures and algorithms used to achieve best performance possible. Therefore a third solution that can bring these two worlds together can prove very helpful for both sides. Users can concentrate on modeling and not on representation details of their models and solvers' programmers can concentrate on improving the algorithms used without having to bother about making their API too cluttered or too different from others. Of course we are not living in a perfect world and such a perfect solution is not possible. However a lot can be achieved in that direction as we shall see.

### ***4.1 Concept of a Generic Interface***

One such effort is the LP-Interface library provided by Siemens Department Discrete Optimization. It is a set of classes. That provides layered and structured access to wide range of solvers. The top layer is the programming API for users. It has a complete set of functions to define, solve and analyze the solution of optimization LP and IP tasks. It also provides easy to understand abstractions of basic structures of LPs and IPs – variables, constraints, restrictions etc. The middle layer is an adapter between solvers' data structures and users' abstractions, doing basic initial preprocessing of the input data to prepare it for feeding in the different solvers, but the actual formatting and transferring to the internal data structures and the communication to solvers' APIs is done by the lowest third layer.

This three-layer model provides enough flexibility for both sides and gives loose-coupling between both interfaces. Striving for generality, the user API can not provide some fancy functions available in some solvers. And on the other hand the underlying lower-level classes should sometimes emulate functionality provided by the user interface that is not directly supported from some solvers. This does not mean that any computations should be performed inside the interface - only functions for simple model transformations, needed for a given solver to understand the model are performed.

We will make this clear in one example. Some solver can not understand ranged variables. These are variables that have their value constrained inside given interval. However this can easily be expressed as restriction in the constraints matrix. Hence the low-level classes of LP-Interface are able to do this transformation. On the other hand if a given solver does not support IPs at all. It is not task of the interface to implement IP solving for that solver. It is only task for it to properly report the

incapability of the solver to the user, so that he can take proper decision for changing the solver, for example.

## **4.2 Implementation**

The idea and implementation of LP-Interface has been proposed and accomplished at the Optimization Department of Siemens AG Corporate Technology. The structure presented so far is the one implemented in this library and has already wide variety of solvers integrated. It works well with many commercial solvers as well as open source ones. It has already been used in different solutions where flexibility and better control over the choice of solvers for different tasks is needed. Hence it has proven the correctness of the ideas standing behind its design.

We shall review some more design decisions that have been taken in the process of defining the interface. LP-Interface strives for keeping as few data about the model in itself as possible. This means data is being channeled to the solver as soon as it is given to the interface. This ensures no misinterpretation can happen from conflicting data coming from the user. As we shall see later on, there are solvers where this particular design decision can not be withheld and some data caching mechanism has to be designed for LP-Interface.

Another important decision made is to keep the interface as much object oriented as possible. This seems as a very natural decision nowadays, but is actually challenge before the programmers of LP-Interface. Main reason for this is the fact that most solvers are written in lower-level languages as FORTRAN, or are projects started back in pure-C days where structures were the most advanced data organization units at hand. Data capsulation can only be achieved in the user's API layer as an abstraction to the POD (plain old data) fed to solvers' interfaces.

Moreover most solvers are only available either as standalone "exe" files or as static libraries to be linked at build time, so they can usually not be exchanged at runtime through newer implementations even when these have exactly the same interface. Newer solvers however often come also as dynamic link libraries. LP-Interface on the other hand has been designed to give choice between many solvers available at the same time, so great care has to be taken to ensure that all solvers can work in the same build of the library.

One future design idea is to implement some plug-in structure that will allow for complete run-time control over the available solvers, and even for integrating new solvers on-the-fly, without the need for restarting the application. In the next chapters some of the challenges that have to be faced in order to integrate this feature in LP-Interface are presented.

## **4.3 Class Structure of LP-Interface**

In the end of this chapter, more detailed description of the internal structure of all layers of LP-Interface will be presented. Some hints about using both endpoints of the application will be given as well as some examples concerning real applications.

The diagram below presents the current structure of LP-Interface.

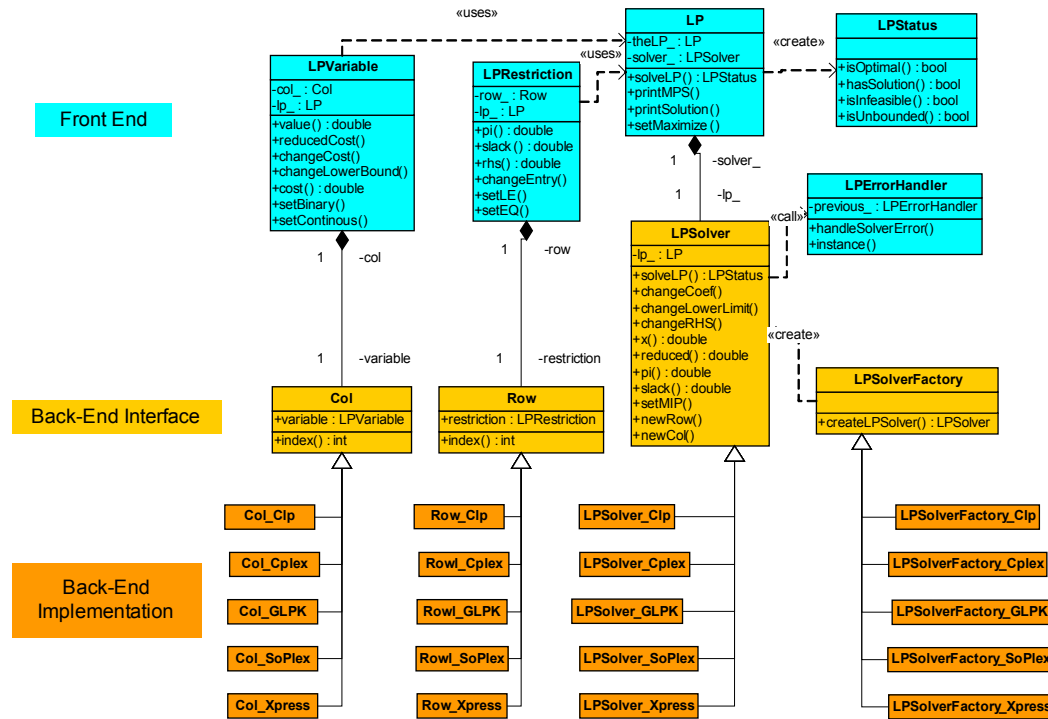


Diagram 1. The Class Structure of LP-Interface

### 4.4 User side interface usage

The whole class structure of LP-Interface is defined in the LPI namespace to prevent naming conflicts. User applications instantiate the LPSolver class and initialize it by choosing the solver to be used. Once the interface is initialized it can be used to define models. First some variables have to be defined by instantiating from the LPVariable class. This class has functions to define variable’s range, type (real, integer, binary) as well as cost for the target function. Next restrictions are defined by creating instances of the LPRestriction class. This class has functions to define constraints as well as a function to set the active variables with their coefficients. Before starting the solver, one can set some parameters. For example it is sometimes useful to disable the simplifier for some solvers, or some users might need increased precision of calculations. When the user is ready to solve the task, the function SolveLP or SolveMIP of the LPSolver instance has to be called. Afterwards solution and its details can be read from the solver, variables and constraints classes. Changes in model can be done at any time and new solving step initiated without the need to completely rebuild the model from scratch. Precise descriptions of the functions and their parameters can be seen in the online documentation provided with LP-Interface.

Details about using the solver-side classes of LP-Interface will follow in chapter 5, describing the integration of MOPS in more details.

## 5. MOPS Integration

As was already stated in the introduction to this work, one main topic of interest is the integration of a new solver to the LP-Interface. This solver is called MOPS and was mainly developed at the Universities of Berlin and the Paderborn. MOPS has very well written branch and bound algorithm and seems to be one of the fastest solvers to date, according to the published benchmarking results.

This, however, comes with its price. One of the most important characteristics of all other solvers is that whatever they internally do with the problem, the structure of the model data stays unchanged. This makes redefinition of parts of the models easy for consequent re-solving. MOPS, on the other hand, works directly on the given structures and changes their representation in a way that prevents any further model modifications, without the need of complete re-import of all the data. Therefore, in order to integrate this solver, we have to give up one of the basic strategies in the design of LP-Interface. Namely find a solution for storing the model parameters inside LP-Interface itself.

### 5.1 Integration Plan

#### 5.2 Sparse Matrix Storing Techniques

The design of the data structures used in many open source solvers is quite simple. They store all matrixes' values row-wise or column-wise as an array of lists of pairs – position, value. Such structures are very easy to implement and use, have very small footprint and correspond well to the way most solvers access their data. Therefore it seems that the design of more advanced data structures stay outside the interests of most solvers' developers. Moreover most of the linear algebra implementations are written by people mostly involved in the research of numerical algorithms and they normally work with regular matrix structures (e.g. diagonal matrices) – therefore designing and using data structures that efficiently store only these special types of matrices.

Such simplified approach is not enough in the case of LP-Interface, because we need data structure that not only allows for fast one-time matrix build and sequent access, but a structure that handles equally well random access for both reading and writing. It should not prefer some special matrix type but work well with all types of sparse matrices.

Another main consideration that arises is to find such data structure that has the best possible balance between performance of operations and in the same time uses as little memory as possible. Undoubtedly the most used operation will be searching for occupied positions in the constraints matrices. These matrices can be extremely large but are normally sparse, rarely exceeding 2-5% of non-zero positions.

A number of different solutions were considered for the design of the matrix structure. The simplest one of these is by a plain 2-Dimensional array. Neither its plain pure

C++ array realization, nor dynamic solutions using the vector class from the C++ STL library are feasible because of the immense memory needed to store large 2-dimensional arrays. Moreover only very few information will be really stored and one can not estimate in advance how big the matrix should really be. Hence such a solution, although virtually the fastest one, outperforming (with  $O(n)=1$  complexity of all its operations) all other solutions, can be easily discredited.

Another solution is to use nested structure of dictionaries in a construction resembling a double hash table. The advantage of this solution is that it requires almost no programming as these classes are present in the STL library too (map class). However in the tests done they have proven to give very poor performance when dealing with large amounts of data. In particular the balanced trees that are in the heart of the MS VC++ implementation of maps, do a lot of rebalancing when fed with ordered data, as often happens when defining optimization models. Nevertheless, the solution can provide square logarithmic access and search times ( $O(n) = \log n * \log n$ ). Another peculiarity that is particularly disappointing about STL's implementation in VC++ are the very long times needed to delete map structures, even rising to 15 minutes to delete a matrix with 4 million elements. Two types of nested constructs were tested here:

```
map< pair< int, int >, double >
map< int, map<int, double > >
```

Quite different in nature they perform almost equal in practice, because the first one leads to a larger tree but requires only one search operation per access. And the second one maintaining a bit larger two-dimensional tree structure keeps its trees smaller and easier to balance.

The third considered solution is a linked list of triplets. A triplet is simply a structure holding horizontal and vertical position of a matrix entry as well as its value. This solution has constant time insertion complexity but linear search and retrieval times which is not considerable for our purpose. Advantage of this solution is its minimal memory space needs. It has the lowest memory footprint of all presented solutions here. A small variation of this structure is a list of lists of values. Each "main list's" entry represents one row or column in the matrix and its "sub-list" the filled positions in the row or column. Thus finding an element on one of the axes will require only a few lookups while searching along the other axis will be as slow as with plain single linked list. As already mentioned in the beginning of the chapter, this is the solution used as internal data structure by most open source solvers.

The next structure we have tried was a hash table. This solution solves some problems with the map based approach – namely, gets rid of the slow rebalancing steps needed by the red-black trees to maintain their performance and has faster destruction procedure because of the lesser amount of pointers involved. Well selected implementation of the hashing function and well chosen size of the pool of hashing values can produce the desired results. A problem immerges if the data to be hashed become much more than the provided hash values' pool. Standard collision resolving algorithm is to store all values that hash to the same value, in a linked list with corresponding key values. This implementation gives linear search time inside colliding values, but these can get up to hundreds of values, hence performance sinks

rapidly with increase of data to be hashed. Large pool is often impossible to keep in memory so it can prove difficult to find suitable threshold for its size. A solution to this problem is to dynamically change hashing pool. This provides a way to keep collision rate from exploding once the hash table gets almost full. Such an implementation exists in the library TURBO [Lauther]. The structure is called a `hash_list`. It doubles its hashing pool each time it gets filled to a given percent. The implementation in the library is very fast and both writing and reading perform very well, even reading being three times faster than writing. The only disadvantage is that the pseudo-random ordering of the hash function does not provide easy mean for enumerating all values for a given row or column in linear time (on the number of values stored for the row/column given). Only searching through all possible indices can yield the filled ones. Another workaround is to define three hash lists. One will be used for row, one for column and the last one for row and column indexing. This solution works fast for all operations needed but uses a lot more memory than a single hash-list.

The last solution tested stems from an idea borrowed from image processing, mainly a structure used for compressing images with large single colored areas. This is analogous to large areas with zero values in matrices. The data structure is called Quadtree. It gives access to elements on 2-Dimensional keys and gives good spatial balance of values, while still keeping the extra service information as little as possible. One good feature of the Quadtrees (QT) is their logarithmic search, retrieval and insertion times. Fast partial traversal of the tree enables for very effective row-/column-wise enumeration and full enumeration is well supported too.

From all presented structures, two seem to fit well to the problem. Hash list with some further facilities to compensate for the missing functions and Quadtrees. Therefore these two structures will be presented in more detail in the next two parts.

As a further note one shall mention that many different approaches exist to deal with the problem. A good way to boost the performance of many implementations is to devise some way of caching queries so that they can be reordered in the waiting queue and executed in a sequence that resembles the internal data organization. Such idea however would require asynchronous sending of queries and some flushing mechanism to collect results. Most software solutions however are built with blocking queries. These only return when the value has been read. Hence to implement such idea would mean to not only redesign LP-Interface but all applications that use it.

As a conclusion of this chapter we present some benchmark results of the two most promising solutions: hash tables and quad trees. The following tests have been done:

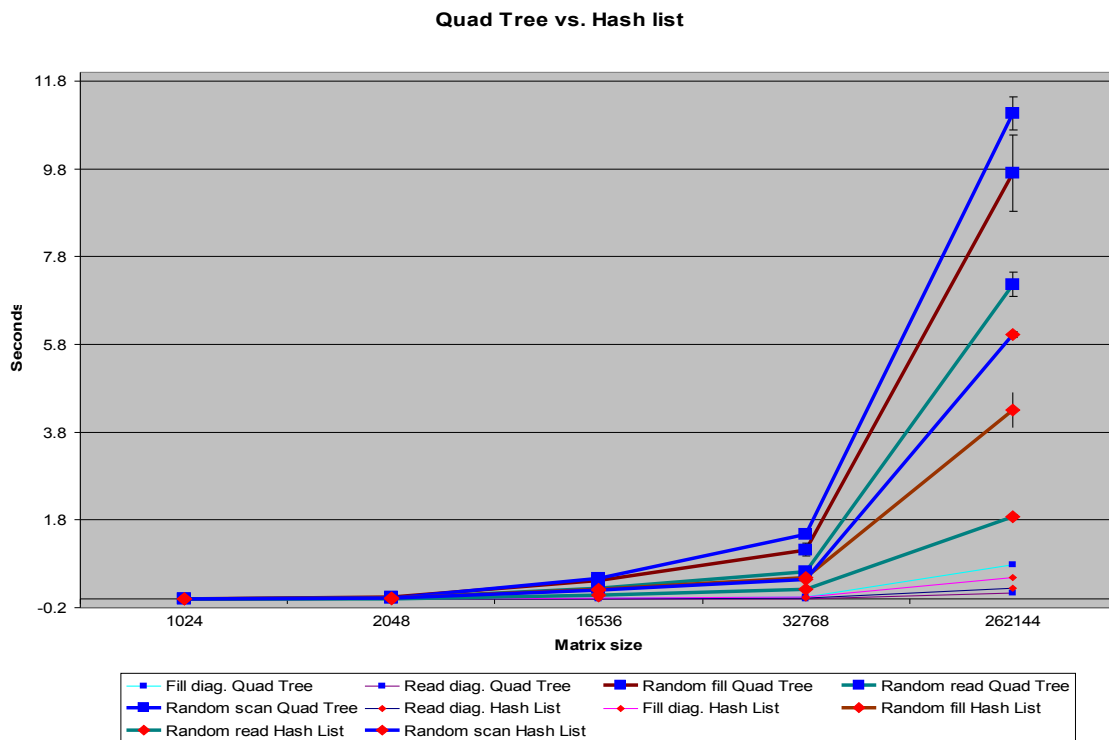
1. Completely fill all entries in a matrix and read them afterwards.
2. Write only the diagonal entries and read them.
3. Fill some random positions in the matrix and read from another random set of positions.
4. Enumerate all entries row-wise.

Tests 3 and 4 have been performed several times to prevent random constellations being better for one data structure. Therefore mean times and standard deviation is reported for these tests.

All tests have been performed on matrices with different sizes from 512x512 to 1048576x1048576 entries. In tests 3 and 4 ten times as much values as the vector length of the matrix have been filled.

Time has been measured with the TURBO Library's Timer class [Lauther] and for all tests no swapping has been assured. STL results have been measured too, but they are not presented here because they fall way behind the other two structures.

The chart presents the time needed to perform different operations on the matrix for both Quadtrees and Hash Lists, so that one can easily compare the results.



*Diagram 2 Hash list vs. Quadtree*

One more interesting comparison is to see how much memory the two best competitors need to store matrices of different sizes. In the following two charts one can read how much kilobytes each representation needs for 4 different test cases with increasing amount of non-zero entries.

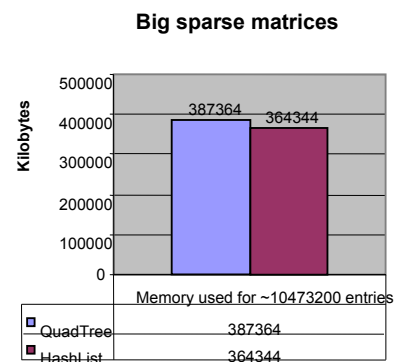
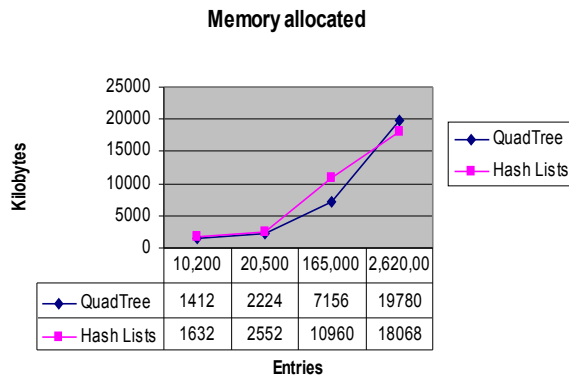


Diagram 5 Memory performance of both structures in head-to-head comparison

### 5.2.1 Quadtrees

A quadtree is a tree data structure in which each internal node has up to four children. Quadtrees are most often used to partition a two dimensional space by recursively subdividing it into four quadrants or regions. The regions may be square or rectangular, or may have arbitrary shapes. This data structure was named a quadtree by Raphael Finkel and J.L. Bentley in 1974. A similar partitioning is also known as a Q-tree. All forms of Quadtrees share some common features:

- They decompose space into adaptable cells
- Each cell (or bucket) has a maximum capacity. When maximum capacity is reached, the bucket splits
- The tree directory follows the spatial decomposition of the Quadtree

For our purpose we shall extend the definition of a Quadtree as follows. A quad tree can be represented as a binary tree with alternating types of nodes – horizontal space dividers and vertical space dividers. Each layer of nesting consists of both types of nodes so that one single layer divides the plane in 4 regions, not-necessarily of the same size (See fig.7). A leaf of the tree contains one or more triplets consisting of values plus their coordinates (in case of Quadtree with complete space decomposition, where position is uniquely defined only by the path in tree, pure values instead of triplets can be stored. It, however, seems that maintaining a partial space

decomposition with leafs holding multiple triplets has better space-speed value). On fig.8 is the graphical representation of a typical Quadtree. Now we can define that the non-zero entries of a matrix correspond to points with integer coordinates in Euclidean space with their values stored at each occupied node. This definition presents no conflict with the presented structure so far, because for each matrix position only one value exists. The given representation of a Quadtree is actually quite similar with a K-d tree although it has the properties and behavior of a Quadtree. Therefore the first picture depicts a 2D K-d tree and only afterwards a Quadtree is depicted.

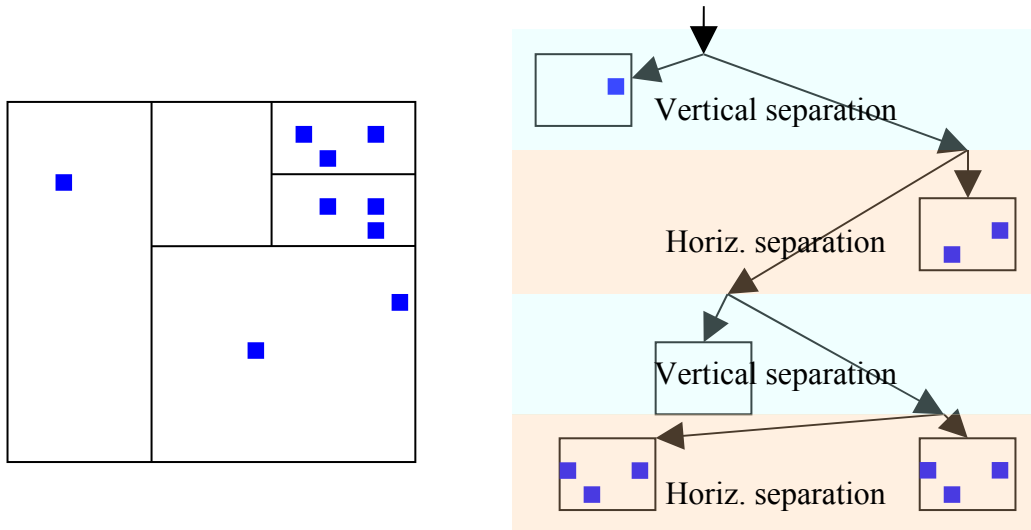


Fig.7 2-Dimensional K-d Trees with buckets used to represent sparse matrix. “With buckets” means that one node can hold more than 1 element (in our case max.3)

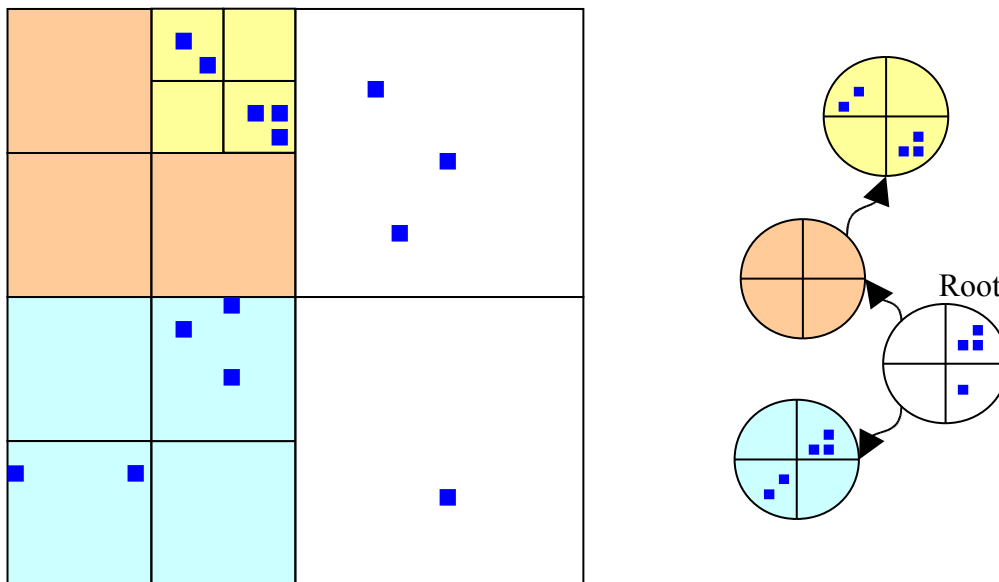


Fig.8 Quadtree with buckets example with buckets' size 3.

The main operations needed are inserting, searching, replacing and deleting of values.

### 5.2.1.1 Inserting Values in a Quadtree

Only insertion will be completely presented, because all other operations are similar in the way they operate on the tree.

Inserting a value in the tree is operation with logarithmic complexity. It consists of traversing down the tree to a leaf node, if the node has more free slots for triplets – store the triplet, if not split the node in 4 new nodes and relocate the values in the new nodes. The overhead to split a node is linear in the number of triplets stored per node. Normally this number is small so the operation has almost constant runtimes.

#### INSERTING A VALUE IN QUADTREE ALGORITHM:

```

p := ROOT
WHILE p <> NULL DO
  IF p IS LEAF_NODE THEN
    FOR i := 1 .. COUNT(p.triplets)
      IF(COORDINATES(p.triplets[i])=COORDINATES(newval) THEN
        VALUE(p.triplets[i]):= VALUE(newval); GOTO 4
      IF COUNT(p.triplets) < MAX_COUNT PUSH_BACK(p.triplets, newval);
      GOTO 4
    ELSE
      SPLITNODE(p); GOTO 4
  ELSE
    IF p IS HOR_SPLITTER THEN
      IF COORDINATES(newval).X < p.MIDVAL THEN p := p.left
      ELSE p := p.right
    IF p IS VER_SPLITTER THEN
      IF COORDINATES(newval).Y < p.MIDVAL THEN p := p.top
      ELSE p := p.bottom
  IF p = NULL THEN PRINT("ERROR IN TREE STRUCTURE")
END

```

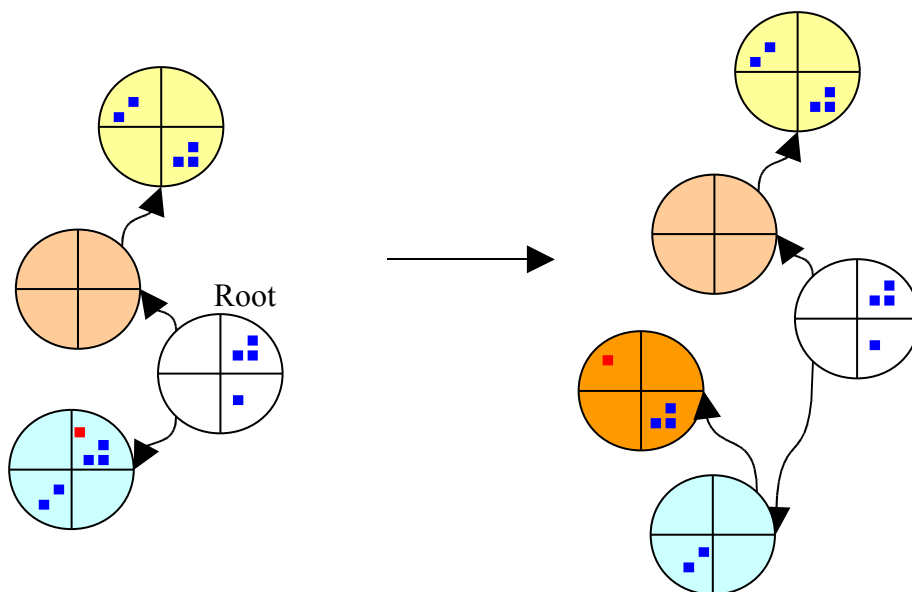


Fig.9 Adding a new value that leads to split of a node in four new nodes

### 5.2.1.2 Searching, Retrieving and Deleting of Values in a Quadtree

Searching and respectively retrieval or changing of values consists of simply traversing the tree and searching the leaf node reached. Deleting of values can easily

be redefined as storing zero value, hence can be implemented as replace operation with only the small price of keeping the memory for the deleted element occupied.

Drawback of the solution is that it needs to have upper boundary of the matrix size upon construction. This problem can be circumvented by setting value larger than any real value that can happen to be used. This will lead to a bit deeper trees and respectively will increase the access times of all operations. However the logarithm function grows too slow for this problem to present a real performance bottleneck.

On figure 10 a typical example of a sparse matrix from the practice is presented with elements grouped in non overlapping sub-matrices and some rows at the bottom representing binding conditions between these sub-matrices. Such form often occurs when model represents more than one smaller sub-problems that have some connection between them and should be optimized together.

This structure can be very efficiently represented by quadtrees because they behave particularly well when such spatial concentration of values is present. Therefore searching and adding values need only few lookups and space is already allocated for new elements in the corresponding buckets.

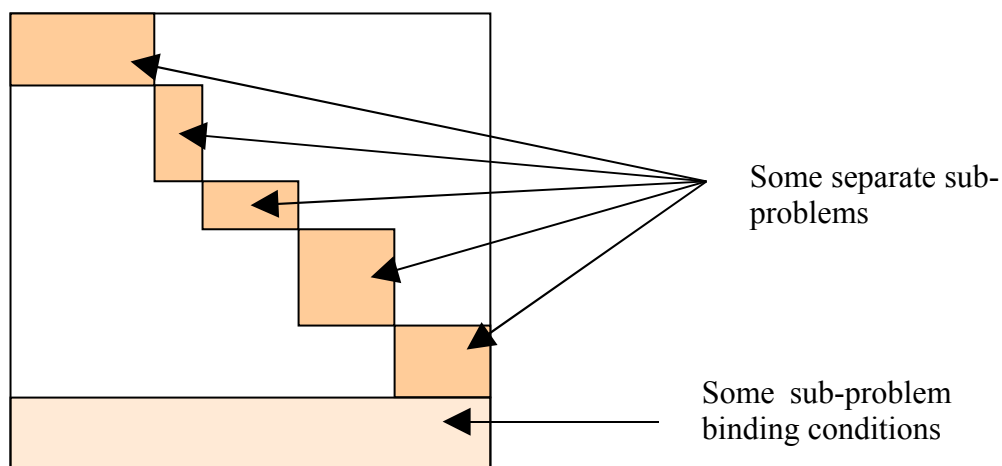


Fig.10 Typical LP matrix structure

### 5.2.1.3 Implementation details

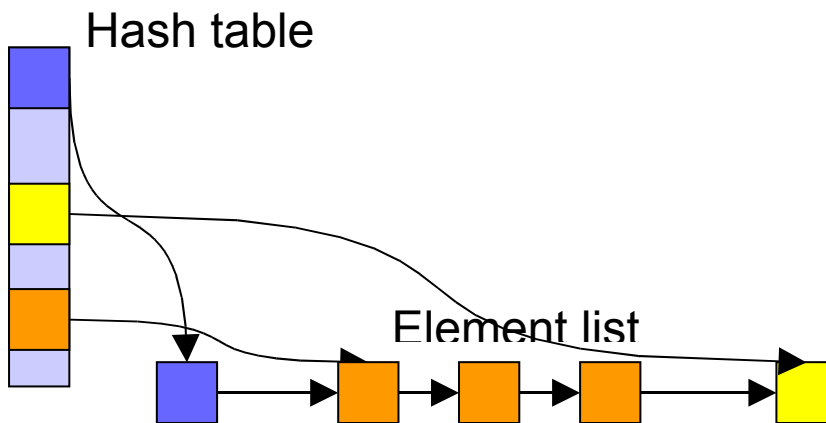
Objected oriented languages like C++ provide easy means of implementing this structure. Our implementation consists of the following classes: QuadTree – keeps the root of the tree as well as the overall matrix size and delegates all operations down the tree. LRNode is a horizontal separator node and also has a member of type `vector<Triplets>`, therefore can represent leaf nodes too. The overhead of keeping an empty list for non-leaf nodes can be neglected compared with the boost in speed given by unified node access throughout the tree. UDNODE is a vertical separator only. It does not need to represent leaf node, because the three has always both separators linked after one another in a single layer. The Triplet class was mentioned before it is just a property class keeping the coordinates of a value in the matrix as well as the value itself. All access functions are in-lined and kept as simple as possible in order to minimize access times for the basic operations, providing up to 500 000 queries per second on a 1,048,576x1,048,576 square matrix with approximately 0.01% of non-zero entries. The size of the matrix may seem strange at first but it is actually a power

of 2 and this is not randomly chosen. Although one can choose any size for the matrix, powers of two give more even distribution of the elements. Therefore the algorithm ensures that the size given in the constructor of a QuadTree is rounded up to the next power of 2 – change that does not increase the time and memory needs of the data structure.

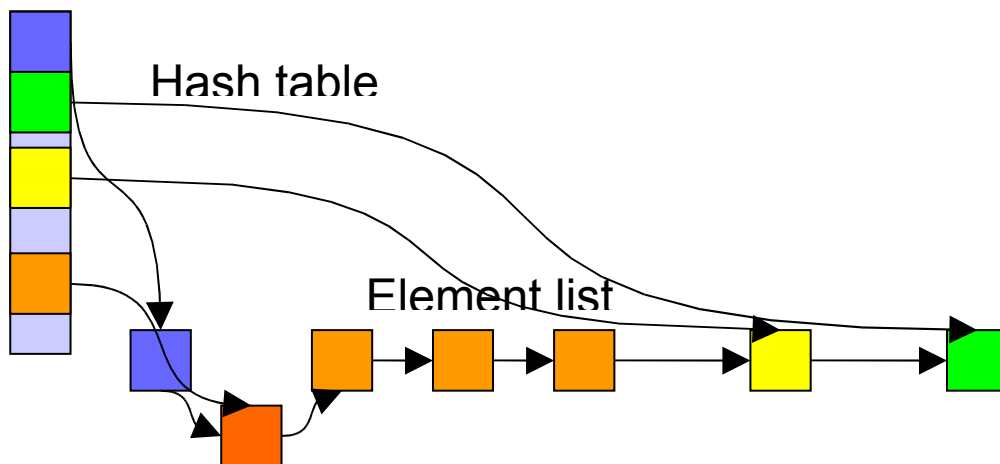
### 5.2.2 Hash lists

Our second structure based on hash tables is called a hash list. A hash table, or a hash map, is a data structure that associates keys with values. The primary operation it supports efficiently is a lookup: given a key (e.g. the coordinates of a value in the matrix), find the corresponding value (e.g. that matrix's value). It works by transforming the key using a hash function into a hash, a number that is used to index into an array to locate the desired location or "bucket" where the values should be. One often used way of storing values that have the same hash value is by keeping a linked list of values that collide with each other. What makes hash lists different is that they use single linked list to store all triplets of values and the hash table links to the first element from that list that has the given hash value. Therefore all colliding values are stored after each other in the list. This spares a lot of space and improves the locality of the whole structure that helps decrease memory cache misses.

The structure can be schematically represented as seen on the figure 11.



*Fig.11 A hash list and its representation in memory*



*Fig.12 The same hash list after adding one element (green) that has new hash key and one element that collides with three other (dark orange)*

The memory overhead per element is a pointer to the next one in the list and a record in the hash table. On the other hand this structure provides  $O(n) = 1$  constant insert and lookup times. Insert only consists of either adding a new element to the end of the list if no collision in the hash table occurs, or inserting the element before the first element with the same hash value and fixing the pointers inside the list and from the hash table. These operations are graphically represented on Fig. 12. Inserts can be slower in some cases as we shall see, but the overall performance still stays constant. What this structure can not provide is a row-wise or column-wise enumeration. If no further memory can be sacrificed these operations have to be implemented by linear search through rows/columns and retrieval of all found elements. Of course such approach is not feasible, because to enumerate all matrix's elements one have to check as much positions as the whole matrix has and this becomes impossible even for moderate matrices. Hence the need for further hash lists that map rows' and columns' indices to lists with filled elements. These structures need further 2 pointers per element and further pointer per row/column for the hashing. Therefore increasing the excess information needed to more than 4 pointers per value. The maintenance of these extra structures adds some time to the insert operation too, but gives virtually constant access time to any part of the matrix.

Another issue to be solved is the size of the hash table used. If it is too small it will lead to many collisions, which would then lead to slower access times. If on the other hand the list is too big, it might use far more space than really needed. The solution to that problem is called a dynamic hash table. Its size grows each time the table gets filled above certain percentage of its size, by doubling itself. All elements in the hash table must be then recalculated to fit the changed hashing function. This operation is not really expensive as it happen exponentially more seldom as the hash tables grows.

Well chosen hashing function leads this approach to very low collision rate and high overall effectiveness of the data structure. For exact implementation description, the reader is encouraged to see the actual implementation in TURBO Library v.1.56 or higher. The description given here is based on exactly that implementation, even the dynamic hash list's resize algorithms has been developed by Prof. Lauther exactly for the problem described in this work.

### **5.3 Integrating the matrix class in LP-Interface**

As we mentioned already two solutions have proven to perform well for the task given. However to choose the one that will best fit LP-Interface library, detailed tests of their performance in real conditions have been done. To ease the testing and to give a chance for future integration of other solutions, a generic matrix interface had to be defined. The interface should not present any specific for a given implementation functions, but at the same time it should not miss any functionality vital for the correct work of LP-Interface.

After careful analysis of the operations performed on the restrictions' matrix inside the interface, we can summarize these as follows:

1. Add/Change coefficients in existing rows and columns;
2. Add new row/column;
3. Remove row/column;
4. Retrieve all non-zero elements for a given row/column;
5. Retrieve the count of all non-zero elements in the matrix;
6. Retrieve the value of an existing element.

Moreover we are somewhat free in the choice of indexing for row and columns inside the matrix as the interface presents the abstract “Row” and “Column” classes that provide tool to assign internal indices not necessarily equal to externally seen rows’ and columns’ numbering.

Using this possibility we can perform some of the listed operations more efficiently by reordering rows and columns in the internal matrix representation.

Operation than can directly profit from this feature is deleting rows and columns. Instead of doing slow copy/move of all following rows or columns, one can simply copy the last row in the deleted one and replace the internal index in the corresponding Row or Column representation class. Eventually mark the last row/column free and zero the elements that have been used there.

Another optimization done is to replace deleting elements with setting them to 0.0. This leads to a kind of “memory leakage” (memory is not completely lost, though, it can be cleared and reclaimed at any time later), but will repay with increased performance the next time this position is assigned again.

Adding and removing rows and columns is an operation that does not require any further reorganization of the matrix, apart from filling values in, if any are given. Both presented structures have only some theoretical upper bound for the maximal matrix size that enables for virtually infinite matrices.

Special attention has already been paid to the row- and column-wise enumeration of the matrix. These operations are very important for MOPS, because before every solving step the whole matrix contents have to be copied in the internal matrix representation of the solver itself [MOPS].

Last operations left to discuss are these working on single elements, especially their setting and reading. These have been already fully covered in the chapters on the different implementations, because they are operations that mostly determine overall matrix performance and are specific for different structures. (See chapters 5.2.1 and 5.2.2 for more details about these).

Using the guidelines given in this chapter, we present the following matrix interface class. All implementations should inherit from this interface in order to assure compliance to LP-Interface.

```
template< class T >
class Triplet_Matrix
{
public:
    virtual ~Triplet_Matrix() = 0 {}
    virtual void Clear(void) = 0;
    virtual void AddValue(const int hind, const int vind, const T val) = 0;
    virtual bool ValueExist(const int hind, const int vind) const = 0;
    virtual T GetValue(const int hind, const int vind) const = 0;
    virtual int getNextValueInCol(const int hind, const int vind, T* val)
                                const = 0;
    virtual int getNextValueInRow(const int hind, const int vind, T* val)
                                const = 0;
};
```

## 5.4 Other Integration Issues

There are few other details around the integration of MOPS that are of interest. Most of the work not already described is of pure technical nature and few implementation decisions had to be taken. However we will cover some of the subtleties that were encountered in this process.

Matrices are not the only structure needed in one such solver. A lot of vector data have to be stored too. This can be represented as a single rowed matrix for example, but this will certainly be pure overkill for such simple and small structure. On the other hand one can not use plain C++ arrays too, because vectors can be sparse but very large as well. Advantage of the variable and constraint classes' in LP-Interface can be used that we already exploited for the delete operation on matrices. Instead of storing the values on their real indexes one can just store them consecutively and use the indirect indexing provided by the classes used to access them. Therefore a simple STL vector arrays can be used for that purpose. These are fast enough and very easy to use.

The MOPS Solver has its own internal memory management. Therefore it has an initialization function that has one parameter - the size of memory to be allocated for the solver. According to the authors there is no easy way to compute apriori correct value for it. They suggest however that values around 800MB should be enough even for very large problems. They also state that because only virtual memory is allocated, almost no physical RAM is used that is not really needed. However windows have it's 2GB virtual memory per process boundary, so one have to be vary careful when using such large values in programs that need huge amounts of memory for other reasons as well.

Another small issue with MOPS is that one can not read parts of the solution but only the whole information about all variables, constraints and duals. Therefore we need some mean to invalidate the solution always when input data has been changed or else old solution can be mistakenly read as actual one.

These two last features of MOPS actually brings us to the situation where the solver class in LP-Interface does most of the processing on input/output and only at the point where SolveLP is called real solver is addressed, and once again to fetch the solution after the problem has been solved. This is a big difference from the initial design and paradigm that stand behind LP-Interface, but is inevitable concerning the solver

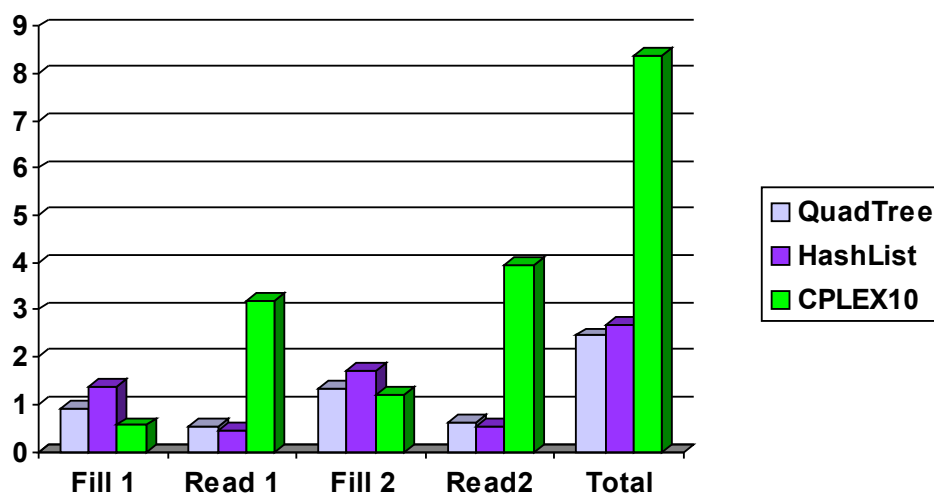
input/output functions. In discussion with the authors of MOPS, we have been assured that this is due to change in future versions of MOPS, so next releases of MOPS might no longer need such workarounds.

Last in this chapter we shall mention that MOPS has a lot of configuration parameters. These have to be made available for the IP-Interface users as well. Some of them like the parameter that switches on or off the simplifier have their own functions dedicated, other are only available as properties of the solver class and the user have to know their names explicitly to change their values. However default values for most of these variables should be sufficient for the normal usage of the solver.

### 5.5 Performance Benchmark of LP-Interface with MOPS

MOPS is a very fast solver. On the other hand the additional overhead to store locally all its information and copy it always from and to the solver can slow down its functionality. Therefore some tests have been conducted to measure performance of the solver from inside LP-Interface. Both speed and memory performance have been measured and compared against some of the fastest solvers already integrated in LP-Interface.

MOPS proves to be really one of the fastest solvers, and even though all its information is stored twice while solver is running it still has average memory footprint. Moreover in the construction phase MOPS is really the fastest solver, because special care has been taken in the development of well suited structure for this very purpose. On the next diagram a comparison against CPLEX has been made on the time needed to construct a problem with 10 000 variables, 10 000 constraints and more than 1 000 000 non-zero entries in the constraints matrix. Both matrices' formats have been tested - Hash lists and Quad Trees.



*Diagram 4 Time for building and reading a complete LP model*

The same test has been done with other solvers as well. Only XPressMP-2003 managed to build the model with filling and reading times of more than 10seconds per

operation. GLPK has been running for 4 minutes without even finishing fill1 phase. SoPlex 1.3.1 has tried to allocate more than 2GB memory, which is the limit of 32bit Widows XP. It is very possible that CPLEX have some caching mechanism while storing its input data. Presumably it does not check for or remove duplicates until the structure is being fed. Therefore it has very small write times and needs considerable amount of time to read back values from the constraints matrix.

## **6 Conclusion**

A short introduction has been made to linear and integer programming. Presented were the mathematical representation and the modeling power of linear programs. The latter was also made clearer on a few examples from praxis. Also most often used computer solvers have been listed with their most important characteristics and differences. Because of the vast differences between solvers and their input and output formats a unified interface was presented that enables for seamless integration of many modern solvers and hides implementation details behind well designed object oriented interface, that is abstraction of paradigms used in modeling language used in defining LPs. In the last chapter a thorough description of the integration of one new solver in the LP-Framework was given, the most important aspect of this being the design and implementation of fast and memory effective matrix structure for temporary storage of input data for the solver.

### **EPILOG**

I would like to thank to the people that helped me in writing this work. First of all I would like to thank to Prof. Dr. Hofmeister and Prof. Dr. Martin for choosing me as one of the few to receive scholarship from the Optimization Group at the Mathematical Department of TU Darmstadt and Discrete Optimization Department at Siemens AG. Next I'd like to thank to Dr. Mueller, Prof. Dr. Lauther and Dr. Moll from Siemens's Discrete Optimization dep., they helped me gather the information presented here. Last but certainly not least I would like to most heartily thank to my parents and my friends for the support and advices throughout the whole time I spent working in Munich.

Yulian Pastarmov  
Munich, February, 28<sup>th</sup> 2007

## REFERENCE

- [Script Opti] Martin, Durr; Script Einführung in die Optimierung; TU Darmstadt; 2005
- [Script Opti2] Martin, Durr; Script Einführung in die Diskrete Optimierung; TU Darmstadt; 2005
- [Lauther] Lauther; TURBO Library v.1.56; Siemens AG CT SE6 München; 2007
- [Zimpl] Koch; ZIMPL User Guide; TU Berlin; 2006
- [MOPS] Zull; MOPS Library v.8.10 Documentation; Paderborn, 2006
- [SIWAPlan] Moll; Presentation of SIWAPlan; Siemens AG CT SE6 München; 2005
- [Presolver] Brearley, Mitra, Williams; Analysis of Mathematical Programming Problems Prior to Applying the Simplex Method; Math. Programming 8; 1975; pp. 54–83
- [Presolver2] Fourer, Gay; Experience with a Primal Presolve Algorithm; AT&T NJ U.S.A; 1993
- [Spatial] Samet; Foundations of Multidimensional and Spatial Data Structures; Elsevier; 2006