Department of Informatics, University of Zürich

Facharbeit

Linear Optimization in Relational Databases

Thomas Preu

Matrikelnummer: 06-738-694

Email: preu@math.uzh.ch

June 15, 2017

supervised by Prof. Dr. M. Boehlen and G. Garmpis

University of Zurich UZH

Department of Informatics

Acknowledgements

I thank the Institut für Mathematik at the University Zürich for using their computing infrastructure. In particular I thank the IT administration, Carsten Rose, Rafael Ostertag and Benjamin Baer, for their continued support.

Abstract/Zusammenfassung

This Facharbeit is concerned with altering the syntax of PostgreSQL 9.4.10 to issue commands to solve linear programming instances stored and encoded as relations and implementing some variants of the simplex algorithm to solve such instances. It provides that for small scale problems. It also surveys several aspects of the simplex algorithm and implements some of these ideas and provides an auxiliary program to import linear programming instances in MPS format.

Diese Facharbeit beschäftigt sich mit der Abänderung der Syntax von PostgreSQL 9.4.10, um Befehle zum Lösen intern als Relationen gespeicherter und codierte linearer Programme absetzen zu können. Dazu implementieren wir mehrere Varianten des Simplexalgorithmuses, welche Probleminstanzen kleiner Grösse lösen können. Wir tragen Hintergrundmaterial zu einigen Aspekten des Simplexalgorithmuses aus der Lehrbuch- und Forschungsliteratur zusammen und stellen ein Hilfsprogramm bereit, das den Import von linearen Programmen im MPS-Format erlaubt.

Contents

1. Background

This Facharbeit is part of a project that wants to implement methods to solve linear programs in an extension of the back end of a PostgreSQL database.

The application lies within feeding nutrients optimally to farm animals. The Swiss Feed Database holds related data and is provided by domain experts. It is assumed that all requirements can be modeled by a (mixed integer) linear program, the modeling itself is not part of the project.

Modern linear programming software such as the proprietary CPLEX, or the open source glpk can solve large linear programs with the number of conditions and variables in the ten thousands in a matter of seconds, however the CPU time required for solution is comparable to the time spent on reading and interface execution (cf. [vP16, fig. 8]). This is a sign that modern solvers are quite adapted to problems from applications but also indicates a different route to further improve running time by implementing a solver directly in the back end of a DBMS and thus reducing the time of data transfer. Another justificationsis that the process of "export$\rightarrow$solve$\rightarrow$reimport" is error prone, not user friendly, not efficient and potentially additional postprocessing to format solutions is needed.

This Facharbeit as a first step is concerned with implementing a solver for (non-integer) linear programs integrated in PostgreSQL. The two main goals are to modify the syntax and parsing in such a way that linear programs can be effectively stored and entered in PostgreSQL and to implement a simplex solver which can process this data.

2. Findings Of The Project

2.1. An Example Of A Linear Program

A linear program consists of three things:

1. A list of real variables, that is usually implicitly given and denoted $x = (x_i)_{i=1}^n$,
2. a list of linear conditions (equations and/or inequalities) in these variable,
3. a linear objective function that is either to be minimized or maximized.

We call an assignment of real numbers to the variables simply a point or vector in our search space. An answer to the problem stated as linear program is a point, such that all conditions are satisfied, and no other point satisfying all conditions has "better" (depending on the direction of optimization) value regarding the objective function. That is, if such a point exists, in which case we call such a point a solution or optimal solution. If no such point exists an answer is simply that statement of non-existence which can be further detailed as either infeasible or unbounded – see below.

In general all linear programs can be reduced to standard form (see [LY16, ch. 2.1] for details), which satisfies additional requirements:

1. for each variable x_i there is exactly a single inequality $x_i \geq 0$, which is summarized in a vector inequality that has to be read componentwise:
$$x \geq 0,$$
2. all other constraints are linear equations which can be summarized in matrix vector form with m rows, where the A is called the system matrix and b the right hand side, and such that $n \geq m$ and $b \geq 0$:
$$Ax = b,$$
3. the objective function is to be minimized and given as a scalar product, where the components of c are called cost coefficients[1]:
$$\min cx.$$

[1]We assume c to be a row vector, thus no transposition is needed in contrast to many text books where $\min c^T x$ is used.

Each variable corresponds to a column in the system matrix A and each equality constraint is a row of A. We will identify columns and rows with variables and equality constraints in the following. The number of rows m and columns n are called the dimensions (in plural) of the problem, the dimension (in singular) means the number of columns. Note: in database terminology one also speaks of rows and columns of a table or relation; the reader is warned not to confuse these notions.

We give an example of a linear program not in standard form:

$$\begin{array}{lrrcl} \max & -1 \cdot x_1 - & 2 \cdot x_2, & & \\ \text{s.t.} & 1 \cdot x_1 + & 2 \cdot x_2 & \leq & 1000, \\ & 2 \cdot x_1 + & 0.5 \cdot x_2 & = & 1250, \\ & & x & \geq & 0 \end{array}$$

One can depict the conditions as halfspaces or hyperplanes and their intersection is the set of points satisfying all conditions, which is called the feasible region (marked by a fat segment in the image). The direction of optimization can be depicted as the (scaled) vector c:

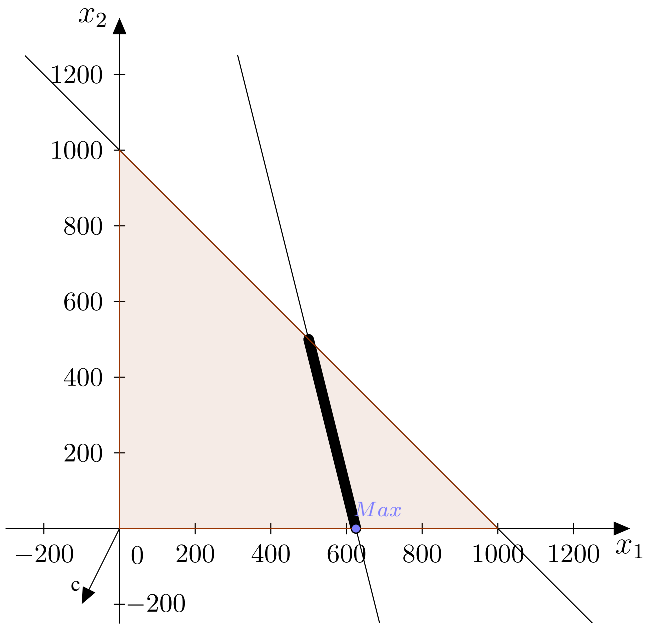

Figure 2.1.: Graphic representation of a linear program.

In the example there is a solution and it is the point $Max(625, 0)$. Obviously this graphic depiction has its limits, in particular if one thinks of a problem in dimension more than 3. Note that the graphic intuition from dimension 2 might be misleading as it cannot capture the full complexity of the problem, such as cycling (cf.2.3).

By changing the sign of the objective function one can get from maximization to minimization. By introducing additional variables and modifying the conditions one finally arrives at an equivalent linear program in standard form:

$$\begin{array}{rllcl} \min & 1\cdot x_1 + & 2\cdot x_2, & & \\ \text{s.t.} & 1\cdot x_1 + & 2\cdot x_2 + \ 1\cdot x_3 & = & 1000, \\ & 2\cdot x_1 + & 0.5\cdot x_2 & = & 1250, \\ & & \qquad\qquad x & \geq & 0 \end{array}$$

Here equivalence means that a solution (s_1, s_2, s_3) of the later program yields a solution (s_1, s_2) of the original problem, and vice versa for any solution (t_1, t_2) of the original program there is a real number t_3 such that (t_1, t_2, t_3) is a solution to the modified program.

If the condition $n \geq m$ for standard form is violated one can use Gaußian elimination to determine superfluous rows and leave them out until the $n \geq m$ is met. The condition $b \geq 0$ can also easily be achieved by sign changes as in the objective function.

A linear program may fail to have a solution in one of two ways. Either there might not be any point satisfying all conditions at once (e.g., $x_3 = -1, x_3 \geq 0$ cannot be satisfied simultaneously for a trivial example). In this case we say that the linear program is not feasible. The other way is that the feasible region is unbounded in the direction of optimization (e.g., $\min -x_1$, s.t. $x_1 \geq 0$ without further conditions). In this situation the linear program is called unbounded. A solution to a linear program may not be unique, we will however not be bother with the question of uniqueness.

The simplex algorithm by G. Dantzig (first presented in [Dan48]) is a method to answer a linear program, i.e., to either find a solution or to show that there is non. See 2.3 for more details.

In practical implementations there are two ways to store the matrices encoding a linear program: dense matrices store all entries including zeros in arrays, sparse matrices can be implemented as linked lists omitting zeros but storing additional row and column information.

2.2. Modifying PostgreSQL

We use PostgreSQL 9.4.10 as starting point for our modifications. More specifically we start from the following revision of the master branch of the git-project for PostgreSQL (see `gitlab.com/ggarmpis/LPinPostgres-ThomasPreu`):

```
* commit e9802122d42aee661113423d290d41b005a9b1b2
| Author:  Tom Lane <tgl@sss.pgh.pa.us>
| Date:  Tue Nov 15 16:17:19 2016 -0500
```

Beginning from a home directory `PGLHome` the unpacked software resides in `PGLHome/postgresql`. As a start to modify the syntax of PostgreSQL we apply a patch provided by the advisor based on a modification to include the possibility to sample from a given table. The details go back to [Con] and are explained there.

The program is maintained and started from a UNIX bash. Some commands are listed in A.2.

A list of essentially modified files is included in A.1 – since the file names of the changed or created files are unique we abbreviate by omitting the full path, which can be retrieved from the appendix. All files from the original version that got modified have comments of the form `/*Simplexchange*/` to mark the beginning of code that got inserted or changed. The only two exceptions are files that do not allow comments without issues in compiling: `Makefile` has an easy to find nodeAlgoscan.c and `errcodes.txt` has a new error code `22P07`.

PostgreSQL processes queries in several stages such as parsing, rewriting, optimization and execution and others which are however not relevant to us (cf. [Con] for general details). In the following we first explain the modification to syntax and parsing, then we discuss the rewrite and optimization stage and finally the executor.

In this paragraph we give details on how the syntax is extended. We introduced a new type of node "algo_expr" that in principle can be used as a frame work to implement new algorithms in PostgreSQL. It is part of a select statement and its syntax is:

```
SELECT * FROM prob_inst ALGO {arguments};
```

The problem has to be specified as a table "prob_inst" according to the encoding format of the algorithm that should be applied. The arguments specify the algorithm to be executed as well as additional parameters. In order to implement new algorithms one has to modify the syntax of PostgreSQL accordingly as well as provide the code for the algorithm in the file `nodeAlgoscan.c`. Currently only the simplex algorithm is supported, indicated by the key word "SIMPLEX". After the next key word "DIM" one must specify the number of rows and columns of the linear program whose system matrix, right hand side and cost function is encoded in the table "asimp_inst".

```
SELECT * FROM asimp_inst ALGO SIMPLEX DIM (dimr,dimc);
```

The grammar is described in `gram.y`. We will call such a statement as above a simplex statement.

In the following two paragraphs we explain the data types and formats of the input and output of a simplex statement. The table "asimp_inst" must have three columns, the first two have to be of type "int" the last of type "float8". An entry (i, j, v) encodes an entry of the system matrix of the linear program, if $i, j > 0$: row i and column j has value v. Unspecified entries are assumed to be 0. If $i = 0$ a cost coefficient v for the j-th variable is specified. If $j = 0$ the right hand side v for row i is specified. This is a presentation of a linear program in standard form which we will explain in section 2.3.

The output is a newly created table with two columns, the first of type "int" the last of type "float8". The entry $(0, s)$ indicates the status of the computation: $s = -1$ indicates that the system is not feasible, $s = -2$ indicates an instance unbounded in the optimal direction of the cost function and $s = -3$ stands for having reached an optimal solution. In the later case an entry (j, v) encodes that the j-th variable in the optimal solution has value v. In the non-feasible case the other entries (j, v) have all $v = 0$ and in the unbounded case the v corresponds to some point on the boundary of the feasible region.

We give some remarks about rewriting and optimization in the next three paragraphs. We store data like dimensions and transitory data needed for the execution of the simplex algorithm in a field "algo_info", which is appended in an rte (relation target entry) of the different nodes involved in the stages from parsing to execution in PostgreSQL. Details of this data structure can be found in `plannodes.h`, `primnodes.h` and `relation.h`. In `execnodes.h` an "AlgoScanState" is introduced that holds this data for execution stage. There are also legacy entries there from the sample scan, that should be removed at some point.

In PostgreSQL target lists specify the data types of return columns of a node. Since we have a different input format compared to the output and both is known up front we need to encode this information at some point. Since target lists are used to store this data we choose to set this up in `parse_target.c`. In discussion with the advisor we were convinced that the proper place to do this should be `parse_clause.c`, since otherwise we might get into trouble when a "ALGO SIMPLEX" modifier is called in a subquery as the return target list of the subquery might not match the entry target list expected in the super query. As a first working implementation we kept the changes in `parse_target.c`.

Most other files that are changed for rewriting and optimization are of minor interest and we simply adapted the changes for a sample scan with the necessary changes. For the estimation of the execution time in `costsize.c` needed for proper optimization we copied the code from sample scan. Thus we only estimate the time for reading and putting out but not for executing the simplex algorithm itself. This is left for future work.

We finally come to the execution stage. For each execution node it is expected by PostgreSQL that there are procedures ExecInit, Exec (or ExecMain) and ExecEnd. In our case ExecInitAlgoScan assigns the data format of input and output table and also the location in physical memory. In general ExecMain is iterated and processes tuple by tuple. ExecAlgoScan in our case reads in all data in the first pass and applies the simplex method. At the end of the first pass it puts out the first row of the output table. Then in iterations it puts out the remaining rows one by one. Finally ExecEndAlgoScan frees memory, e.g., the transitory memory needed to hand over the results from the execution node to the PostgreSQL system. We follow the standard scheme except that we read all the input tuple at once in ExecAlgoScan and only do the output iteratively. This is due to the fact that the simplex algorithm potentially needs all data at once.

We also have an alternative implementation that demonstrates the possibility of leaving the original data on disk and not fetching everything into working memory at the beginning. We do not exploit the algorithmic potential for the simplex method however and only demonstrate a proof of concept. In this implementation we used the functions "heap_markpos" and "heap_restrpos" for heap scans to mark positions on disk memory and later come back to it. In the newest versions of PostgreSQL these functions are no longer provided for sequential scans but still seem to be there for index scans. For this to work it is required that the data is stored ordered first with respect to increasing columns and within that with respect to increasing rows. The code is located in `nodeAlgoscanBackupAOnDisk.c`. By copying it to `nodeAlgoscan.c` and compiling one can activate this implementation.

We also investigated the possibility to use hash maps that are employed within PostgreSQL for index scans but it seemed to much work to access the internal hash tables directly. Thus we relied on the functions accessing the heap in sequential scans.

2.3. Background On Linear Programming

For linear programs there are several different algorithmic methods to solve them. The most popular are the simplex method (cf. [LY16, ch. 3]) and interior point methods such as the ellipsoid method (cf. [LY16, ch. 5]). We did not consider other approaches, such as the very first but outdated Fourier-Motzkin method.

The ellipsoid algorithm has the advantage that it provably runs in polynomial time with respect to the input size of the problem. The simplex method is not polynomial time: for none of the proposed variants polynomial time could be proven and for some of the most common variants there are problem instances known which require exponential time. However for practical instances arising from applications usually only a linear number of iterations of the basic simplex step, say about $3m$, is needed (cf. [LY16, ch. 5.2, p. 118]). There are also some theoretical results pointing in the direction of general well-behavedness of the simplex method, despite some extreme exponential instances ([CSRL01, ch. 29, p. 897], [Sch99, ch. 11.5, p. 143]).

In competitions between interior point algorithms and simplex methods the later usually outperforms the former on instances coming from real world applications. Therefore we will also implement the simplex method.

In the remaining of this section we will first introduce the simplex method, then discuss some variants and finally make additional remarks.

2.3.1. The Iterative Step Of The Simplex Method

The basic idea of the simplex method is to iterate a simplex step starting from a feasible solution until an optimal solution is found, or a verification that there is no optimal solution. For expository reasons we explain the simplex step first and afterwards the additional initialization around this iteration. Here is a pseudo-code description of the main ideas, the details can be found in [LY16, ch. 3].

Algorithm 1 Simplex-Step

```
Ensure: $base$ is a list of length $m$ with integer values $j$ such that $1 \leq j \leq n$ and such that the
    unique feasibility condition is satisfied
Require: $A$ is an $m \times n$-matrix, $b$ is an $m$-vector and $c$ is a $n$-vector
 1: procedure SIMPLEX-STEP($m, n, A, b, c, base$)
 2:     Compute a reduced cost vector $\tilde{c}$ from $c$ and $A$
 3:     if $\tilde{c} \geq 0$ then
 4:         return IsOptimal=true
 5:     Determine column $jIn$ to enter the base from $\tilde{c}$ (avoid cycling: Bland's rule)
 6:     Determine row $iOut$ such that $base(iOut)$ leaves the base from $A, b$ and $jIn$
 7:     if $(jIn, iOut)$ could not be found then
 8:         return IsBounded=false
 9:     Set $base[iOut] = jIn$.
10:     return IsOptimal=false, IsBounded=true
```

One assumes at the beginning of each step that a feasible point is known. We use the structure of a linear program in standard form. The feasible region F is the intersection of the positive orthant[2] given by $x \geq 0$ and an affine-linear[3] subspace of $\mathbb{R}^n$, which is a convex affine polytope. One can show (cf. [LY16, ch. 3]) that there always is a (0-dimensional) vertex of F, provided $F \neq \emptyset$. Furthermore the relative boundary of F is at the boundary of the orthant where some variables have to vanish and one can show that at least $n - m$ variables of a vertex are vanishing. In order to describe a vertex it thus suffices to specify m columns that represent the (potentially) non-vanishing variables; these are called basic variables. This (ordered) assignment of the m rows to m distinct of the n columns is called a base[4].

Actually a base specifies a feasible point uniquely provided certain unique feasibility conditions hold, which are specified now. A base singles out a quadratic $m \times m$-submatrix B of A. Any feasible point needs to satisfy $Ax = b$ and if we set the non-basic variables to 0 then a solution of $B\bar{x} = b$ gives (potentially) non-vanishing coordinates of a solution. $\bar{x}$ is uniquely determined, provided B is invertible, and the associated x is feasible provided $\bar{x} \geq 0$. In this sense does a base specify a feasible point, more precisely a vertex of F, provided the two stated conditions hold.

We will represent these feasible vertices by the base and assume that the unique feasibility conditions hold. The simplex step then produces from a given feasible vertex another one, such that the cost function is at least not increased. In one step the base changes by substituting a leaving column $base(iOut)$ by a new entering one jIn. The criteria to determine jIn and $base(iOut)$ ensure that the unique feasibility conditions are preserved. Geometrically going from one base to another means going from one vertex of F to another that are connected by an edge, which is described by $base, base(iOut)$, and jIn.

We determine $jIn, base(iOut)$ such that the cost is non-increasing – actually we aim at dropping the cost but this cannot be ensured in every step (cf. 2.3.5 for details). This potential drop in cost is determined by so called reduced cost coefficients $\tilde{c}$. Each step is governed by $\tilde{c}, jIn, base(iOut)$ and these data can be computed from A, b, c and $base$.

There are three different behaviors when taking several iterations together. Either the actual cost drops eventually than we go on. Or at some point the cost would drop unboundedly in which case there are feasible points but no optimal ones – geometrically this means that from a feasible vertex we follow a boundary edge that does not stop at a new vertex, but is actually an unbounded ray. The third possibility is that none of the edges at the current vertex have the potential to decrease costs, in this case we have reached a solution.

If we can assure, that each base occurs at most once in any iteration of the simplex step, i.e., we guarantee that we do not cycle through the same bases indefinitely without making progress, this will assure termination of the algorithm as there are only finitely many distinct bases. See below for further remarks on these anti cycling methods.

[2]An orthant is the generalization of quadrants in $\mathbb{R}^2$ and octants in $\mathbb{R}^3$ to arbitrary dimensions.

[3]For the remainder “linear” will always mean “affine-linear”.

[4]The connection to the notion of basis from linear algebra would go beyond the scope of this work and will not be explored.

2.3.2. The Initialization Of The Simplex Method

Now we need to talk about the initialization, i.e., how one gets a feasible vertex resp. a base in the first place. For this one augments the linear program to a new auxiliary program which has an easy feasible point. One introduces m new variables and appends an $m \times m$-identy matrix I as a block to the system matrix $A' = (A, I)$. The right hand side b stays the same. We have $n + m$ variables in the auxiliary linear program and declaring the first n non-basic and the last m (in order) to be basic we get a base. It satisfies the unique feasibility conditions: the identity matrix I is clearly invertible, and the solution of $I\bar{x} = b$ is non-negative, since in standard form $b \geq 0$.

Now we have a feasible point but only for the auxiliary conditions. For this we introduce an auxiliary cost vector $c' = (0, 1)$ with n zeros and m ones. The idea is that the auxiliary variables are expensive, while the original ones are for free. Thus iterating the simplex step on this auxiliary program will try to diminish the auxiliary variables in favor of the original ones.

Algorithm 2 Simplex

```
Require: $A$ is an $m \times n$-matrix, $b$ is an $m$-vector and $c$ is a $n$-vector
 1: procedure SIMPLEX($m, n, A, b, c$)
 2:                 ▷ Initialize an auxiliary program with known feasible point $(0, b)$:
 3:     $A' = (A, I)$ with $I$ an $m \times m$-identity matrix
 4:     $c' = (0, 1)$ with $n$ zeros and $m$ ones
 5:     for $i = 1$ to $m$ do
 6:         $base(i) = n + i$
 7:                 ▷ First phase to find a feasible point via an auxiliary program:
 8:     IsOptimal=false, IsBounded=true
 9:     while IsOptimal=false, IsBounded=true do
10:         SIMPLEX-STEP($m, n + m, A', b, c', base$)
11:     for $i = 1$ to $m$ do
12:         if $base(i) > n$ then
13:             return Infeasible
14:                 ▷ Second phase to find an optimal point via the original program:
15:     IsOptimal=false, IsBounded=true
16:     while IsOptimal=false, IsBounded=true do
17:         SIMPLEX-STEP($m, n, A, b, c, base$)
18:     if IsBounded==false then
19:         return Unbounded
20:     Set $B$ to be the quadratic submatrix $A$ specified by $base$
21:     return Optimal Solution $\bar{x} = B^{-1}b$ padded with 0 for non-basic variables
```

One can show that this indeed works. Either one finds a base where all basic columns are original variables, in which case this base encodes a feasible vertex of the original program that can be used as a starting vertex. Otherwise, i.e., if we find an optimal solution to the auxiliary program that does not have a base exclusively consisting of original variables then one can show that the original program was actually infeasible. Since the auxiliary cost function is bounded from below by 0 the unbounded case cannot occur for the auxiliary program. Pseudo-code for this wrapper algorithm around the simplex step is given above.

In summary the simplex method proceeds as follows. For an auxiliary program we have an easy feasible vertex, encoded in the form of a base satisfying the unique feasibility conditions. Using the simplex step iteratively in a first phase we can either produce from this a feasible vertex of the original program, or show that the original program is actually unfeasible. If we have a feasible point, we can apply the simplex step iteratively in a second phase. The outcomes are either that the program is unbounded or that it has a solution in which case an optimal solution is easily read off from the last base.

2.3.3. The Tableau Simplex Method

Actually the two variants of the the simplex algorithm which we implemented, the tableau simplex and the revised simplex, work a bit different in detail. We will explain the ideas of the differences to the basic version that was described above.

We introduce some notation. For simplicity we assume that we are in one of the phases of the simplex method and do not introduce terminology to distinguish between the two phases. We use k to index the iterations of the simplex step with the start at $k = 0$. We reorder the columns of A such that the basic variables occupy exactly the last m columns, which is actually the case at the beginning of the first phase by default. We can thus subdivide $A_k = (D_k, B_k)$, where D_k is for non-basic columns and B_k for the basic columns and A_k denotes the reordering of A for the base to be in the end after k steps. In particular in the first phase $A_0 = (D_0, B_0) = (A, I)$, where I is the $m \times m$-identity matrix, A is the original system matrix and only the trivial reordering was applied. Similarly we reorder and subdivide the cost vector $c_k = (c_k^D, c_k^B)$. The information is stored in a tableau $T_k = (T_k^b, T_k^D, T_k^B)$ where the first part is a column vector corresponding to the right hand side b and the other two parts correspond to non-basic resp. basic variables. We have $T_0 = (b, D_0, B_0)$.

Algorithm 3 Tableau-Step

```
1:  procedure TABLEAU-STEP($m, n, A_k, b, c_k, base, T_k$)
2:      Compute a reduced cost vector $\tilde{c}_k = (\tilde{c}_k^D, \tilde{c}_k^B) = (c_k^D - c_k^B T_k^D, 0)$
3:      if $\tilde{c}_k \geq 0$ then
4:          return IsOptimal=true
5:      Choose a column $jIn$ with $(\tilde{c}_k)_{jIn} < 0$
6:      Determine $iOut$ s.t. $(T_k)_{iOut,jIn} > 0$ with minimal $(T_k)_{iOut,0}/(T_k)_{iOut,jIn}$
7:      if $(jIn, iOut)$ could not be found then
8:          return IsBounded=false
9:      Set $base[iOut] = jIn$,
10:     Compute the new tableau $T_{k+1}$
11:     Reorder $A_{k+1} = (D_{k+1}, B_{k+1}), c_{k+1} = (c_{k+1}^D, c_{k+1}^B)$
12:     return IsOptimal=false, IsBounded=true
```

The tableau step is just a detailed version of how to actually compute $\tilde{c}_k, jIn$ and $iOut$. See the pseudo-code above. We give a naive complexity analysis assuming that all data is given as dense matrices and vectors, which is actually the case for the tableau method. We indicate by t_i an estimate for the number of floating point operations in line i as given in the pseudo-code. We omit negligible costs and use $m \leq n$ for standard form linear programs:

$$t = t_2 + t_5 + t_{10} = O(mn) + O(mn) + O(m(m+n)) = O(mn)$$

For the assignments to compute the new tableau in line 10 see (3.6) of [LY16, ch. 3.1]. The idea is to replace the column jIn by a standard basis vector[5] e_{iOut}. One achieves this by applying Gaußian elimination to the whole tableau. Then one permutes the columns again to have $T_{k+1} = (B_{k+1}^{-1}b, B_{k+1}^{-1}D_{k+1}, I)$.

2.3.4. The Revised Simplex Method

The Tableau Simplex uses dense representations. However many practical instances are rather sparse. For example the 51 medium sized problems of the collection of linear programs from NetLib have as median only 8.43 entries per row (see 2.6 below) compared to a number of columns of at least 500. The revised simplex method is more suited for implementations with sparse matrices. The tableau method looses sparsity over the course of execution in the $(m \times n)$system matrix A while revised simplex leaves it unaltered and sparsity can only degrade in the smaller $(m \times m)$-matrix B^{-1}.

Algorithm 4 Revised-Step

```
 1: procedure REVISED-STEP($m, n, A_k, b, c_k, base, B_k^{-1}$)
 2:     Compute a reduced cost vector $\tilde{c}_k = (\tilde{c}_k^D, \tilde{c}_k^B) = (c_k^D - (c_k^B B_k^{-1})D_k, 0)$
 3:     if $\tilde{c}_k \geq 0$ then
 4:         return IsOptimal=true
 5:     Choose a column $jIn$ with $(\tilde{c}_k)_{jIn} < 0$
 6:     Form the two tableau columns $(T_k)_0 = B_k^{-1}b$ and $(T_k)_{jIn} = B_k^{-1}((D_k)_{jIn})$
 7:     Determine $iOut$ s.t. $(T_k)_{iOut,jIn} > 0$ with minimal $(T_k)_{iOut,0}/(T_k)_{iOut,jIn}$
 8:     if $(jIn, iOut)$ could not be found then
 9:         return IsBounded=false
10:     Set $base[iOut] = jIn$, reorder $A_{k+1} = (D_{k+1}, B_{k+1})$, $c_{k+1} = (c_{k+1}^D, c_{k+1}^B)$
11:     Compute $B_{k+1}^{-1}$
12:     return IsOptimal=false, IsBounded=true
```

Essentially we do the same operations as in the tableau simplex except that we delay the evaluation of the actual tableau until we determine $iOut$. But since we only need two columns of the tableau for this, this reduces costs for computing the tableau by a factor of n from $O(mn)$ to $O(m)$.

But we need to compute the inverse of a matrix B_{k+1}^{-1} which costs $O(m^3)$ using standard Gaußian elimination. However we only need to replace a single column in B_k to get to B_{k+1}. More conceptually we need to update the inverse of a rank-one-modification by a column

[5]A standard basis vector e_i has zeros everywhere except at the i-th entry which is one.

vector u and a row vector v and this can be done efficiently using a special case of the Sherman–Morrison formula, which itself can be computed by 5 matrix-vector multiplications and 1 matrix-matrix subtraction (cf. slide 109 of [Hala] or [Wik17c]):

$$(A + uv)^{-1} = A^{-1} - \frac{A^{-1}uvA^{-1}}{1 + vA^{-1}u}.$$

We use this updating method in our implementation and the complexity analysis for the dense case compliant with our implementation gives $O(m^2)$.

When computing the reduced costs we could simply use a matrix-vector multiplication with a part of the tableau before. Now we need to do actually two matrix-vector multiplications. So in the end the revised simplex method for dense representations will have the same complexity per step as the tableau simplex:

$$t = O(mn).$$

The advantage is that we can use sparse representations. Denote by τ the number of non-zero elements in the system matrix A. The complexity for sparse matrix-vector multiplication is bounded by 2τ. Updating an inverse of a sparse matrix by a rank-one-modification is more complicated to estimate. The updated inverse usually has non-zeros in the same place as the original one except for some newly added non-zeros called fill-ins. So updating a sparse inverse should require operations proportional to the number of non-zeros plus fill-ins. In any case updating requires at most $O(m^2)$ steps. The complexitiy of the sparse revised simplex is per step at most (cf. slide 109 of [Hala]):

$$t = O(m^2 + \tau).$$

See 2.4 below for further discussion on sparse matrices.

2.3.5. Remarks On Degeneracy

The simplex method proceeds by several steps going from a vertex of the feasible domain to another along edges that promise potential improvement of the objective function. There are several phenomena of degeneracy that may occur in this process and we give some comments about these here.

First of all the feasible region may not posses any vertices at all, despite not being empty. A simple example can be constructed by any non-empty polytop $F \subset \mathbb{R}^n$ and taking $F' = F \times \mathbb{R}^k \subset \mathbb{R}^n \times \mathbb{R}^k$. One can show that up to a linear transformation all examples of this phenomena are of this type (cf. [Gri13, 4.3.17 Korollar]) – see below for a graphic example. In the context of standard form programs this cannot arise, since the condition $x \geq 0$ ensures that no linear subspaces are contained in an associated F. In this sense bringing a linear program in standard form eliminates degeneracy. On the other hand it might also introduce numerical instability (cf. [Gri13, 4.5]).

Another type of degeneracy arises when the rows of a standard form problem are linearly dependent. In this case no square submatrix of the system matrix will be invertible and the unique feasibility condition is not satisfied by any candidate for a base when entering the

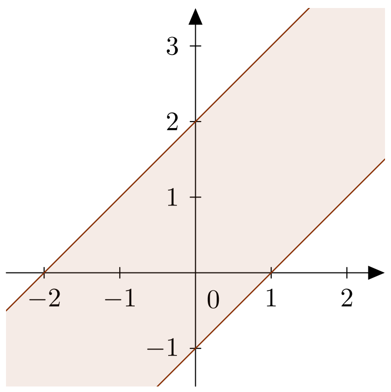

Figure 2.2.: A convex set without vertices.

second phase. Linear dependence can be efficiently detected using Gaußian elimination beforehand. Depending on the right hand side the feasible region is either empty or there are superfluous equality conditions. In the first case no optimal solution exists and we are done, in the second case the superfluous conditions can be detected by Gaußian elimination and discarded. We assume that this kind of degeneracy does not occur in the input to our implementation.

Then a vertex may be overdetermined. Geometrically this means that more than n of the (in-)equality conditions are satisfied with equality. For example at the apex in a square pyramid in 3-space 4 faces are intersecting indicating such a degeneracy. For standard form programs this is equivalent to at least one of the basic variables being 0. In such a situation the same vertex is determined by several different bases and for any two such bases exchanging base columns of one bases one by one by new columns of the second will lead to a sequence of bases all determining the same vertex. One can think of two bases differing only in a single column as two “virtually different” vertices that are connected by a “virtual edge” of length 0.

Although the length of these virtual edges is 0 algorithmically one associates a direction to them and if it is positively aligned with the direction of optimization taking this virtual edge promises potential progress but since we only go length 0 until we hit the next (actually the same) vertex in this direction the value of the objective function is unaltered. Even worse it may occur that after taking several of these virtual edges one ends up at the starting base and one has come full circle. This phenomenon is called cycling and it can occur even if one always takes the virtual edge that promises best alignment with the direction of optimality according to the reduced cost coefficients. See [Möh, slide 4 ff.] for an explicit example in standard form with $(m, n) = (3, 7)$.

Thus one needs “anti-cycling rules” that will assure that each base is visited at most once. There are some numerical methods such as perturbing the system by a small essentially random error once a no-progress step was detected entering potential cycling and thus (hopefully) eliminate the degeneracy – in the example with the square pyramid if one tilts one of the faces slightly the apex will resolve to form a small ridge between two distinct perturbed vertices. One can detect when one has left the original vertex completely and return to the unperturbed system afterwards. This is possible since after each step the status of the algorithm is essentially reconstructable from the discrete information of the list $base$ and no numerical information is needed.

There are also discrete methods such as Bland's rule, which we implemented and sketch in pseudo-code below. It relies on ordering the variables once and for all at the beginning – usually such an order is given implicitly from the start via the the column indices. This rule provably avoids cycling.

Algorithm 5 Bland's rule for revised simplex

```
1: Choose a column $jIn$ with $(\tilde{c}_k)_{jIn} < 0$ s.t. additionally $jIn$ is minimal
2: Form the two tableau columns $(T_k)_0 = B_k^{-1}b$ and $(T_k)_{jIn} = B_k^{-1}((D_k)_{jIn})$
3: Determine $iOut$ s.t. $(T_k)_{iOut,jIn} > 0$ with minimal $(T_k)_{iOut,0}/(T_k)_{iOut,jIn}$ and in case of
     ties take that $iOut$ s.t. additionally $base(iOut)$ is minimal
```

It is common practice that one takes jIn to be such that $(\tilde{c}_k)_{jIn} < 0$ is most negative (see [LY16, ch. 3.4 and ex. 3.18]). One could combine this usual practice with Bland's rule in the following way: one uses the most negative reduced cost coefficient to determine jIn unless the objective value has not made progress in the last step in which case we use Bland's rule. We however only implemented Bland's rule. See [LY16, ex. 3.37] or [Möh, slide 10 ff.] for further details and remarks on anti-cycling.

Finally another case of degeneration would be if the unique feasibility conditions were violated, i.e., either the base submatrix B is not invertible or the solution $B^{-1}b$ has negative components. The auxiliary program with the initial base clearly satisfies the invertibility condition. So we only need to show that this property is inductively preserved. But this is exactly ensured by the choice of $iOut$ such that $(T_k)_{iOut,jIn} > 0$ (cf. [LY16, ch. 3.1, p. 35]). On the other hand, should it be impossible to find such an $iOut$, unboundedness follows. Thus the first part of the conditions is satisfied between iterations.

At the beginning we start with a non-negative solution b padded by some 0 for the auxiliary program, since b is required to be non-negative by the conditions on standard form. The conditions of minimality of $(T_k)_{iOut,0}/(T_k)_{iOut,jIn}$ in choosing $iOut$ is exactly such that the first basic variable that is to become negative following an improving edge is chosen to leave the base and thereby bounded from below by 0, thus non-negativity of all basic and non-basic variables is conserved between steps (cf. [LY16, ch. 3.2]).

We have discussed several degenerate situations. Other issues in the same spirit may arise from numerical inaccuracies. We defer discussion of these to subsection 2.4.4.

2.4. Implementation Of A Simplex Algorithm

In this section we describe several aspects of the implementation of the simplex method in the file `nodeAlgoscan.c`. We first outline the various data structures, procedures and functions that are implemented there. Some of the functions are not used for all of the implemented variants – see 2.4.1.

First of all `matElem` is a data type that stores a real and an integer value together with a pointer to a struct of its own type and is used to implement a linked list storing a column of the system matrix A and similar data in sparse format. The real variable holds the matrix entry, the integer variable holds the row index of the entry. The column of an entry is assumed to be implicitly specified elsewhere.

```
struct matElem
{
        int             row;
        float8          entry;
        struct matElem *cnext;
};
```

The functions ExecInitAlgoScan, AlgoNext, AlgoRecheck, ExecAlgoScan and ExecEndAlgoScan are standard for any scanning execution node and they govern reading and outputting a relation. ExecAlgoScan does not follow the usual setup. Usually the “ExecMain” is iterated and reads a single tuple and then generates an output tuple. In our implementation we read all tuples describing the linear program in the first call and only output one tuple after another in iterated calls. ExecAlgoScan provides the entry point to the simplex algorithm.

simplexinitTab and simplexmainTab are implementations of the algorithms 2 and 1 in the tableau variant. It uses dense matrices which are stored as an array with a single index. To simplify access and emulate an array with two indices we use the helper function matind.

simplexinitRevSparse and simplexmainRevSparse provide an implementation of the same pseudo-code but in the revised variant and using a sparse representation using the data type matElem.

We tried to implement some improvments on inverting matrices. For this we implemented a Hopcroft-Karp bipartite matching algorithm in the functions shortestBFS, hopcroftKarpDFS and hopcroftKarp and an algorithm to find strongly connected components in a directed graph by Tarjan in sccRecursive and sccTarjan. Furthermore we coded functions that perform an LU-decomposition of a block-triangular matrix in upluSingBlock and uplu, as well as some functions uFromRightSolve, uFromLeftSolve, lFromLeftSolve and luFromLeftSolveBlock solving linear systems using the resulting LU-decomposition.

For the signatures of these functions see A.3.

2.4.1. Implemented Simplex Methods

We have four different implementations of the simplex method. In order to switch between these one has to copy the appropriate source file to nodeAlgoscan.c and in one case set a certain switch in the code.

First Implementation: Tableau Simplex

We first implemented a tableau simplex. The file nodeAlgoscanFinDense.c holds the corresponding code and one has to set the variable method to 1 before compiling. All data is read in first and stored as a dense matrix in a single array. Via the helper function matind we access this array a as if it would have two indices. If the dimensions of the system matrix are m rows and n columns the array is effectively an $(m + 2) \times (n + 2)$-array. The array is subdivided as follows

$$\begin{pmatrix} s & c & 0 \\ b_k & A_k & 0 \\ 0 & \tilde{c}_k & 0 \end{pmatrix}.$$

c is the original (auxiliary) cost vector and is unchanged over the iterations of the simplex step. It is a column vector residing in row 0 as indexed in C. b_k and A_k represent the tableau after k-steps and $\tilde{c}_k$ the reduced cost vector. They change in the course of the execution of the algorithm. b_k is in column 0, A_k occupies rows 1 to m and columns 1 to n and $\tilde{c}_k$ is in row $m + 1$. The entry s at $(0, 0)$ is used to indicate the return state ($-1, -2$ or -3 as specified in 2.2) at the end of execution. Just before termination the last row is used to hold the found solution x instead of $\tilde{c}_k$. This is returned to the calling function `ExecAlgoScan`.

Second Implementation: Dense Matrix Revised Simplex

The next implementation is a revised simplex using dense matrix representations. The corresponding code is in the file `nodeAlgoscanFinDense.c` and one has to set the variable `method` to 2 before compiling. We use two arrays. In initializing the simplex we modify A and right hand side b by sign changes in such a way, that $b^+ \geq 0$. The first of the arrays `a` holds these modified A^+ and b^+, and they are not altered between the simplex steps. We omit the index k for readability here and set $p = iOut, q = jIn$. The format is similar as in the tableau variant explained above, except for an added row:

$$\begin{pmatrix} s & c & 0 \\ b^+ & A^+ & 0 \\ 0 & \tilde{c} & 0 \\ 0 & (\hat{a})_p & 0 \end{pmatrix}.$$

The row $(\hat{a})_p = \pi^T A$ is the pivotal row of the simplex tableau in the notation of [Hala, slide 98] (see below for π^T). The inverse matrix to the base submatrix B of A and additional transitory data is stored in an array called `b` (do not confuse with the right hand side equally named b!). It is again a single indexed array that is accessed via two indices through `matind` and therefore effectively an $(m + 2) \times (n + 2)$-array.

$$\begin{pmatrix} 0 & \hat{\pi}^T & \alpha \\ \hat{b} & B^{-1} & \hat{a}_q \\ 0 & y^T & \hat{c}_q \end{pmatrix}.$$

All notation is as in [Hala, slide 98]: the pivot column $\hat{a}_q = B^{-1}A_q$, the pivot factor $\alpha = \hat{b}_p/\hat{a}_{p,q}$, the updated right hand side $\hat{b}_{new} = \hat{b} - \alpha\hat{a}_q$, $\hat{\pi}^T = e_p^T B^{-1}$, $y^T = c^B B^{-1}$ and $\hat{c}_q$ is the q-th entry of the revised cost vector $\tilde{x}$. The inverse matrix is not computed from scratch each time but rather via updates (see 2.3.4). The reduced cost vector is computed from scratch.

Third Implementation: Revised Simplex System Matrix A On Disk

The third implementation is a revised simplex using a dense matrix representation as an array `b` for the inverse matrix B^{-1} as before but accessing the essentially static system matrix A from disk instead of loading it into working memory. The corresponding code is in the file `nodeAlgoscanFinAOnDisk.c`. We use an additional array `redcost` to store the transitory data of `a` from $\tilde{c}_k$. With this implementation we investigate how to modify the data access to make use of disk memory. A requirement is that the relation storing the static tableau data b, c and A is ordered first by column then by row[6]. Thus when iterating through the tuples using standard PostgreSQL-API we access the tuples in this order. We also need to index the start on disk of each column of the tableau before executing simplex. Thus we have to never the less iterate through all given data at the beginning and we will actually have at least as many IO-operations as with the variant before. This implementation is just a proof of concept, in order to efficiently access the different rows one might additionally require that a list of columns is provided along side the tableau data when issuing the `SELECT ... ALGO SIMPLEX ...` command.

Forth Implementation: Revised Simplex Sparse Matrices In Working Memory

Finally we coded an implementation starting from the second implementation above that tries to improve on the inverse matrix. The file `nodeAlgoscanFinSparseWM.c` has the corresponding code. First of all is the representation not a dense inverse matrix, but rather a sparse format representing an LU-decomposition. Second we tried to improve heuristically the number of fill-ins when inverting. We also planed to combine this with an update strategy for this kind of LU-format but time restrictions of this Facharbeit and the complexity of this task which we underestimated did not permit it. We give a detailed account of the code and the ideas behind it in subsection 2.4.3.

2.4.2. Accessing Data

First of all PostgreSQL provides its own API for allocating and freeing memory on the heap. The basic commands are `palloc`, `palloc0` and `pfree`. While one can use standard C system calls it is not recommended. See `https://blog.pgaddict.com/posts/allocation-set-internals` for details.

We started out from the modification described at [Con], which uses blockwise scanning of a relation. We aim at a version where the original system matrix is not stored as a whole in working memory but rather columns are fetched on "as-needed"-basis to minimize time spent for IO, as outlined in 1. Therefore we choose to modify the data management from `nodeSeqscan.c`. The system matrix is stored as a relation in PostgreSQL and the solution has to be written into the data base as another relation. Here we explain some aspects of the IO.

[6]We use the convention that the right hand side b is in column 0, while the cost coefficients c are in row 0

Reading And Outputing Tuples

The code in `ExecInitAlgoScan` and `ExecEndAlgoScan` is a rather straight forward modification from `nodeSeqscan.c`. There is one noteworthy exception though: as discussed in 2.2 we need to set the target lists for the output relation manually in `parse_target.c`. The plan node "node" of type `AlgoScan` defined in `plannodes.h`, that is an argument for `ExecInitAlgoScan`, carries the necessary information for both target lists and at one place we need to switch them:

```
scanstate = makeNode(AlgoScanState);
scanstate->ss.ps.plan = (Plan *) node;
/*further code*/
node->scan.plan.targetlist = node->algo_info->targetlistIn;
ExecAssignScanProjectionInfo(&scanstate->ss);
node->scan.plan.targetlist = node->algo_info->targetlistOut;
```

Starting from `ExecAlgoScan` we can access the relation through its argument equally called "node", which is an execution node of type `AlgoScanState` defined in `execnodes.h`. The following code snippet indicates how to retrieve the row index "indr", column index "indc" and value "val" of an entry of the system matrix:

```
slot = ExecScan((ScanState *) node, (ExecScanAccessMtd) AlgoNext,
    (ExecScanRecheckMtd) AlgoRecheck);
while (!TupIsNull(slot))
{
    indr = DatumGetInt32(slot_getattr(slot,1,&isnull));
    indc = DatumGetInt32(slot_getattr(slot,2,&isnull));
    val = DatumGetFloat8(slot_getattr(slot,3,&isnull));
    /*code to process this entry of the system matrix*/
    slot = ExecScan((ScanState *) node, (ExecScanAccessMtd) AlgoNext,
        (ExecScanRecheckMtd) AlgoRecheck);
}
```

The functions `AlgoNext` and `AlgoRecheck` used as function pointers above are again straight forward modifications of corresponding functions in `nodeSeqscan.c`.

After this we close the relation and do some clean up, so that the relation can potentially be accessed in parallel by another PostgreSQL-client:

```
relation = node->ss.ss_currentRelation;
scanDesc = node->ss.ss_currentScanDesc;
ExecClearTuple(node->ss.ss_ScanTupleSlot);
heap_endscan(scanDesc);
ExecCloseScanRelation(relation);
```

For the output we need to create again in `ExecAlgoScan` tuples for the newly created output relation encoding the found solution. Since in this stage of the program `ExecAlgoScan` is called iteratively to output each tuple separately, we need to have a variable to carry the information of the current column index j to put out the x_j across calls. This is done by the variable "node->param4". The variable "node->a" holds the solution as an array.

```
resTupDesc = ExecTypeFromTL((node->ss.ps.plan)->targetlist , false);
resSlot = MakeSingleTupleTableSlot(resTupDesc);
ExecClearTuple(resSlot);
if (node->param4 <= dimc)
{
    resSlot->tts_values[0] = Int32GetDatum(node->param4);
    resSlot->tts_values[1] = Float8GetDatum((node->a)[node->param4]);
    resSlot->tts_isnull[0] = false;
    resSlot->tts_isnull[1] = false;
    resSlot = ExecStoreVirtualTuple(resSlot);
    node->param4++;
}
return resSlot;
```

In the above it is hardcoded that in the input table the first column holds the row index of an entry of the system matrix as an `int32`, the second the column index as an `int32` and the third its value as a `float8`. Similarly the output types are hardcoded as `int32` for the first column of the output table and `float8` for the second. This is compatible with `parse_target.c`. Should the input relation not follow this data convention the behavior of our code is undefined; one should expect erroneous behavior.

Keeping The System Matrix On Disk

Now we turn our attention to providing support for leaving the system matrix on disk and not storing it in working memory. In `ExecAlgoScan` we initialize two arrays "aStartDiscTid" and "aStartDiscInd" which indicate the start of each column – recall that we assume the relation to be stored sorted by columns as first key and then by rows as second key.

```
aStartDiscTid = palloc0((dimc+2) * sizeof(ItemPointer));
aStartDiscInd = palloc0((dimc+2) * sizeof(int));
relation = node->ss.ss_currentRelation;
scanDesc = node->ss.ss_currentScanDesc;
oldIPD = scanDesc->rs_ctup.t_self;
if (scanDesc->rs_pageatatime)
    oldInd = scanDesc->rs_cindex;
/*store next tuple in slot as above*/
while (!TupIsNull(slot))
{
    /*access indr, indc, val as above*/
    if (aStartDiscTid[indc] == NULL)
    {
        aStartDiscTid[indc] = palloc(sizeof(ItemPointerData));
        *aStartDiscTid[indc] = oldIPD;
        if (scanDesc->rs_pageatatime)
            aStartDiscInd[indc] = oldInd;
    }
    oldIPD = scanDesc->rs_ctup.t_self;
    if (scanDesc->rs_pageatatime)
        oldInd = scanDesc->rs_cindex;
    /*store right hand side and cost coefficients in working memory*/
    /*store next tuple in slot as above*/
}
```

With the above code the two arrays will have the information to point to the tuple just before the start of a column. This is necessary because `ExecScan` iterates to the next tuple before putting it into a slot and making it available for information retrieval.

The next code snippet is for accessing the first tuple of the j-th column. The following tuples in this column can be accessed by iterating `ExecScan` until "indc" of the latest tuple indicates a different column.

```
(node->ss.ss_currentScanDesc)->rs_mctid = *aStartDiscTid[j];
if ((node->ss.ss_currentScanDesc)->rs_pageatatime)
    (node->ss.ss_currentScanDesc)->rs_mindex = aStartDiscInd[j];
heap_restrpos(node->ss.ss_currentScanDesc);
slot = ExecScan((ScanState *) node, (ExecScanAccessMtd) AlgoNext,
    (ExecScanRecheckMtd) AlgoRecheck);
/*access indr, indc, val as above*/
```

2.4.3. Enhancing Sparse Matrix Inverses

As already discussed in 2.3.4 it is not an option to compute the inverse B^{-1} in the revised simplex method as a dense matrix via Gaußian elimination in each simplex step as its time complexity is too bad. There are update strategies for dense B^{-1} that were already discussed above (cf. [Hala, slide 109]). In this subsection we discuss several approaches to make computing and/or updating sparse representations of B^{-1} more efficient and point to several sources in the literature that we found useful or interesting – it is by no means a comprehensive overview of this matter. At the end we discuss what aspects of these improvements we tried to implement and what difficulties we came across.

We studied in particular the following articles and presentations: [Halb] gives an introduction and a practical selection of methods how to generate and update an inverse matrix in the context of the revised simplex method. [DRSL16] is a recent overview article on direct methods for inverting sparse matrices and [AB] provides slides to some lectures on the same topic. Lecture notes on updating inverse matrices for the revised simplex can be found in [Sau] and [HH13] is a recent research article that compares several update methods in this context and provides experimental results.

A fundamental question is how to represent a sparse matrix. There are several different formats that are discussed in [DRSL16] or [AB]. We decided to use an array of linked lists: each list represents a column of the matrix and each node is of type `matElem` and stores the row and the value of the entry as well as a pointer to the next entry in this row. Generally the list is ordered by ascending row index. Later we will discuss an LU-decomposition. The U-factor is stored in the same way, but for the L-factor each linked list represents a row.

Direct Methods For Matrix Inverses

[DRSL16] gives an overview of current direct methods to invert a sparse matrix A. Direct methods are in contrast to iterative solvers which use numerical techniques to approximate solutions – as it is not clear how sparseness can be preserved using iterative methods we do not discuss these any further. We will list several of the results provided there because we base our approach on a selection of these techniques.

The inverse is always represented via some kind of factorization: for symmetric matrices as Cholesky factorization $PAP^T = LL^T$ or for general quadratic matrices the LU-decomposition $PAQ = LU$. In this cases P, Q are permutation matrices and L, U are (lower resp. upper) triangular matrices. [DRSL16] also considers the QR-decomposition $A = QR$ with Q orthogonal (note that permutation matrices are orthogonal in particular) and R upper triangular. We will usually omit the "decomposition" to make terminology smoother from now on.

Except for non-generic cases these decompositions will have non-zeros where the original matrix had non-zeros, but they might have additional ones called fill-ins. Obviously reducing fill-ins is desirable. The freedom in choosing permutations P, Q (called pivoting) can be used to achieve two often contradictory goals: reduction of the number of fill-ins and reduction of pile up of numerical errors.

A nice fact is that the analysis of fill-in generation is related across the different types of decomposition. The sparsity pattern for LU for a square matrix A is bounded above by the one for the QR which is itself related to the one for the Cholesky decomposition for A^TA, at least when the matrix A is strong hall, i.e., its block triangular decomposition (see below) consists of a single block (see [DRSL16, p. 54]).

There are several variants to compute the LU of a square matrix. The up-looking LU (see [DRSL16, p. 41]), the left-looking LU (see [DRSL16, p. 42]) and the right-looking LU (see [DRSL16, p. 45]). Their advantages and disadvantages regarding choosing permutations and implementations are discussed there.

The notion of bandwidth of matrix is introduced in [DRSL16, 8.2]: for a matrix $A = (A_{i,j})$ the lower bandwidth is $\max\{i - j : A_{i,j} \neq 0\}$, the upper bandwidth is $\max\{j - i : A_{i,j} \neq 0\}$ and the bandwidth is the maximum of both. Reduction of bandwidth is connected to fill-in reduction. They also explain the connection from efficient matrix decomposition to related graph problems by viewing a square matrix as adjacency matrix where the non-zeros indicate an edge in a (directed) graph.

At several places (cf. [DRSL16, pp. 12, 66, 67, 71]) results are mentioned that show NP-hardness of problems associated to fill-in reduction. In particular finding permutations P, Q that minimize fill-ins is NP-complete, as well as minimizing the bandwidth. A recent result (cf. [DFU11]) shows that even trying to approximate minimal bandwidth within a factor of 2 is NP-hard via proofing an analog result for a related graph problem. In the face of these negative results it is clear that one resorts to heuristics several of which are discussed in [DRSL16].

One of these strategies is named after Markowitz [DRSL16, pp. 48, 58]. We explain it for the up-looking LU. In the process of computing this decomposition proceeds to give a bigger and bigger square matrix in the upper left corner (after permutation) that is inverted by "adding" a single row and column in each step. We assume that we have computed an LU-decomposition $L_{11}U_{11} = \tilde{A}_{11}$ in the upper left corner:

$$\begin{pmatrix} L_{11} & & \\ l_{21} & 1 & \\ ? & ? & ? \end{pmatrix} \begin{pmatrix} U_{11} & u_{12} & ? \\ & u_{22} & ? \\ & & ? \end{pmatrix} = \begin{pmatrix} \tilde{A}_{11} & \tilde{a}_{12} & \tilde{A}_{13} \\ \tilde{a}_{21} & \tilde{a}_{22} & \tilde{a}_{23} \\ \tilde{A}_{31} & \tilde{a}_{32} & \tilde{A}_{33} \end{pmatrix} = \tilde{A}$$

In this setup $\tilde{A} = PAQ$ is an intermediate permutation of the matrix A. One still has the freedom to change the permutation in the lower right part of $\tilde{A}$ with upper left corner at $\tilde{a}_{22}$ – such a permutation will result in a change of the pivot $\tilde{a}_{22}$. One can estimate the number r of non-zeros in l_{21} and c of non-zeros in u_{12} that will result from different choices of $\tilde{a}_{22}$ and Markowitz rule says to take that, which minimizes the product rc. In this sense it is a local greedy heuristic.

The Markowitz rule is a dynamic method: while performing the LU one computes the permutation alongside. One can also use pre-orderings where permutations are determined and applied before perform the LU. One of these methods is permuting to a block triangular decomposition (abr. BTD). A matrix in (upper) block triangular form by definition looks like this:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ & A_{22} & A_{23} \\ & & A_{33} \end{pmatrix}.$$

A BTD is a permutation in block triangular form such that the diagonal blocks cannot further be refined. If a matrix can only has a trivial BTD as a single block we say that the matrix is strong Hall. One can show that the BTD is essentially unique up to permutations of the blocks and permutations within the blocks, and that the diagonal blocks have the strong Hall property. Since all the lower blocks are zero there cannot arise any fill-ins in this part when performing LU. When permuting to BTD one reduces fill-ins at least in L, while R might still have many fill-ins.

Another advantage of BTD is that one does not actually have invert the upper blocks in order to solve a linear system. Assume we are given

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ & A_{22} & A_{23} \\ & & A_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

We can solve this system in three steps: $x_3 = A_{33}^{-1}b_3$, $x_2 = A_{22}^{-1}(b_2 - A_{23}x_3)$ and $x_1 = A_{11}^{-1}(b_1 - A_{12}x_2 - A_{13}x_3)$. So we are only left with computing LUs for the diagonal blocks.

[DRSL16, 8.7] explains how to find a permutations to put A in BTD. BTD is essentially the same as finding the strongly connected components of the directed graph for which A is the adjacency matrix, provided that the diagonal of A is filled with non-zeros. To achieve this one can regard A as the adjacency matrix of a bipartite graph whose vertices are the rows resp. columns and find a bipartite matching. The invertibility of A is equivalent to the existence of a perfect matching in the associated graph and a perfect matching gives rise to a permutation matrix Q such that the column permutation AQ has a non-zero diagonal. We have implemented the Hopcroft-Karp algorithm to find bipartite matchings and were following [Wik17b]. To identify strongly connected components we implement Tarjan's algorithm as presented in [Wik17d]. It produces a permutation matrix P such that $P(AQ)P^T$ is in BTD.

We briefly sketch the main ideas behind these two algorithms. Hopcroft-Karp greedily matches successively a node u from one part U to a node v from the other part V, i.e., uv is an edge. Then one increases any such matching, as long as possible, to two pairs by finding v' and u' such that $v'u$ and vu' are valid edges: removing the single matching uv and entering two matching $v'u$ and vu' will increase the number of matchings by one. Similarly one proceeds from such a chain with two matches to a chain with three matches and so until one has found a perfect matching.

Tarjan's algorithms starts from any vertex in the graph and performs a depth-first search until all vertices reachable from there are found. While doing this one keeps track of cycles by representing a cycle by the first vertex of this cycle that was entered in depth-first search. The last cycle to be found is cut off and forms a strongly connected component. Then one proceeds to the second last cycle and so on until all reached vertices are discarded. Overall one restarts this process until no further vertices are left.

Permuting A to BTD is not unique, since we can still permute within the blocks and the blocks themselves. One might permute the diagonal blocks further to achieve further fill-in reduction. We observed however that the ordering already produced by Tarjan's algorithms seems to reduce the lower bandwidth within each diagonal block nicely. We looked for theoretical justification of this but could not find anything definitive. There are at least some noteworthy heuristics that reduce the lower bandwidth (cf. [DRSL16, pp. 67f]). We do not know if one of them provably minimizes lower band width. Also take into account that minimizing total band width is NP-complete. So we simply pose this question for further investigation: Does Tarjan's algorithm produce an ordering that provably minimizes lower bandwidth?

We also had another unanswered question during studying the literature for fill-ins. How bad can unavoidable fill-in generation become in the worst case? E.g., can it happen for some matrices A that the fill-in minimizing permutations still have number of full-ins larger than the original number of non-zeros? And how badly is optimal fill-in behaved if we exchange several columns over the course of the iteration of the simplex steps? We speculate that one can construct such badly behaved matrices with the help of expander graphs (see [Wik17a]). This are graphs that have few edges but rather strong connectivity properties. They are rather artificial however and thus might seldomly arise in practical applications.

We conclude this part on fill-in reduction with another remark on the literature. [AB] provides slides for a lecture that also discusses many of the above topics as well.

Update Methods For Matrix Inverses

One needs to distinguish two aspects when discussing updates. The first is whether an actual inverse is updated or some decomposition. The second aspect concerns storage of the update: does the update come as a modification of existing data, such that the modified representation can be used as is, or is it provided as supplemental data, that leads to conceptually added operations.

We will here summarize the findings of [HH13] that compares the Forrest-Tomlin update to three forms of product form updates, the classical one (PF), the alternate product form (APF), and the middle product form (MPF). All of these supplement an LU-decomposition with external update data. Other update methods are the discussed in [Sau] or [Halb].

As already discussed in 2.3.3 does the simplex step modify the base matrix B only in a column which is a rank one modification. Let e_p denote the p-th standard basis vector, that is zero everywhere except for the p-th entry which is 1. If the p-th column of B is to be replaced by the q-th column a_q of the system matrix then the newly created matrix B' can be written as

$$B' = B + (a_q - Be_p)e_p^T = B(I + (\hat{a}_q - e_p)e_p^T) = BE$$

where $\hat{a}_q = B^{-1}a_q$ and is readily available in the course of the matrix step. To represent E we only need to know p and $\hat{a}_q$ which can be efficiently stored. Let σ be the number of non-zeros of $\hat{a}_q$. E is the identity matrix with one column replaced by some column – this is called an η-matrix. These matrices are simple enough such that for any vector v the products $E^{-1}v$ and $v^T E^{-1}$ can be computed in time $O(m + \sigma)$ from p and $\hat{a}_q$ and v.

Assume we have executed the simplex algorithm k steps and have the base matrix B_k. Let $B_k = L_k U_k$ be an LU-decomposition. Let E_{i+1} denote the η-matrix for the i-th step, then we have after k' further steps

$$B_{k+k'} = L_k U_k E_{k+1} E_{k+2} \cdots E_{k+k'}.$$

Storing L_k, U_k and the updates E_i allows for efficient computation of $B_{k+k'}^{-1}v$ for any v. This is the classical PF update. However after k' steps the computational cost was increased by $O((m + \bar{\sigma})k')$, where $\bar{\sigma}$ is the average of the various σ. This and pile up of numerical errors require a restart with a fresh decomposition $B_{k+k'} = L_{k+k'} U_{k+k'}$ every once in a while, say about after $k' \approx O(m)$ steps.

APF and MPF work similarly except that the decompositions look like this:

$$B_{k+k'} = T_{k+k'} \cdots T_{k+2} T_{k+1} L_k U_k, \qquad B_{k+k'} = L_k \tilde{T}_{k+1} \tilde{T}_{k+2} \cdots \tilde{T}_{k+k'} U_k.$$

The T_i and $\tilde{T}_i$ are essentially as easy to obtain and store as the E_i – see [HH13] for the details.

In experimental results in practical instances [HH13] found that the most competitive product form update was MPF comparable to rest-Tomlin update.

Our Approach

Since matrix inverses are only need for the revised simplex method we implemented matrix inverses only for the last three of our four implementations. For the first of these three we implemented B^{-1} as dense matrix and updated according to the formula from [Hala, slide 109]:

$$B'^{-1} = (I - \frac{(\hat{a}_q - e_p)}{\hat{a}_{p,q}} e_p^T) B^{-1}$$

For testing reasons we also implemented a Gaußian elimination algorithm to compare the results of updating inverses with computing inverses from scratch. Our second revised simplex implementation aimed at exploring the possibility of storing the system matrix on disk and we did not change anything for the inverse of the base.

For our implementation in `nodeAlgoscanFinSparseWM.c`, i.e., the last of our four implementations, we tried to combine a BTD with an up-looking LU-decomposition using a Markowitz-like pivoting rule and a MPF update strategy. We explain the different parts and difficulties that arouse. For the various functions involved see 2.4 above.

We explained above that it is necessary to restart the updating from time to time. So assume that we start from the base matrix $B = B_k$ after k steps and we need to find a good LU-decomposition $PBQ = LU$ first. As discussed above permuting B to BTD has many advantages, in particular reducing the lower bandwidth significantly, so this is our first step. The main technical issue is getting the various permutations correct that come from finding a non-zero diagonal first and then applying the strongly connected component algorithm of Tarjan. As also discussed above only the resulting diagonal blocks that are strong Hall need to be inverted.

For each diagonal block we need to compute an LU and we use the up-looking LU as outlined before. We compromise a Markowitz-like fill-in reducing strategy with a numerical stability criterion. Generally we apply column permutations to reduce fill-in and row permutations to ensure numerical stability, except in unstable cases. Assume we have a partial LU-decomposition $L_{11}U_{11}$ already for $\tilde{A}$ a permutation of B and we need to permute a suitable a_{22} from the lower right part:

$$\begin{pmatrix} L_{11} & & \\ l_{21} & 1 & \\ ? & ? & ? \end{pmatrix} \begin{pmatrix} U_{11} & u_{12} & ? \\ & u_{22} & ? \\ & & ? \end{pmatrix} = \begin{pmatrix} \tilde{A}_{11} & \tilde{a}_{12} & \tilde{A}_{13} \\ \tilde{a}_{21} & \tilde{a}_{22} & \tilde{a}_{23} \\ \tilde{A}_{31} & \tilde{a}_{32} & \tilde{A}_{33} \end{pmatrix} = \tilde{A}$$

Denote $\tilde{A}' = \left(\begin{smallmatrix} \tilde{a}_{22} & \tilde{a}_{23} \\ \tilde{a}_{32} & \tilde{A}_{33} \end{smallmatrix}\right)$. If all entries of $\tilde{A}'$ have absolute value below 10^{-2} we permute the maximal entry to $\tilde{a}_{22}$. Otherwise choose among those columns that have entries above 10^{-2} the sparsest column and within this sparest column the row with maximal entry. Should among the eligible columns be several column of minimal sparseness take the column with largest entry.

The wrapper function `uplu` calls the function `upluSingBlock` for each diagonal block. This function computes two arrays of linked lists representing the factors L_i and U_i for the i-th diagonal block. When iteratively computing the various u_{12} and l_{21} we already need to solve triangular systems for the already obtained U_{11} and L_{11} and we call functions `uFromRightSolve`, `lFromLeftSolve` for this. Additionally when we have the decompositions for all the diagonal blocks we need another function to solve equations of the form $Bu = v$. This can be done with `luFromLeftSolveBlock` that calls `lFromLeftSolve` and `uFromLeftSolve` for each block separately and incorporates the information from the unaltered off-diagonal blocks.

For testing we also implemented a standard Gaußian elimination algorithm and compared the resulting explicit dense inverse B^{-1} with the effect `luFromLeftSolveBlock` had on the standard basis which gives the columns of B^{-1}.

After coming this far we discovered that the MPF is actually incompatible with our format of storing LUs only for diagonal blocks and leaving off-diagonal blocks unaltered. MPF requires a full LU-decomposition of the matrix B in order to add updates between these factors. BTD allows to efficiently solve linear systems in a way that requires LU-decomposition of the diagonal blocks only and not of the off-diagonal blocks. But it is impossible to harness the benefits from partial LU-decomposition and fulfill the requirements of a full LU-decomposition in order to apply MPF. The PF and APF should be compatible however as they append the updating factors either in front or at the back and do not interfere with the internal structure of the decomposition. In the end we did not implement any update strategy.

2.4.4. Numerical issues

Finally we make some remarks on numerical issues. When testing the reduced cost coefficient numerical errors can decrease an actual 0 to a small negative value. This leads to wrong decisions when choosing $(iOut, jIn)$ and eventually to wrong results, e.g., can a program be deemed infeasible while it is feasible. We therefore set all small reduced costs below a certain error tolerance to 0:

```
float8  errTol=1.0e-9;
for (j=1; j<=dimc; j++)
{
    /*code to compute reduced cost coefficients*/
    if (fabs(redcost[j]) < errTol)
    {
        redcost[j] = 0;
    }
}
```

When determining $(iOut, jIn)$ in the simplex step we applied the same error tolerance to avoid some $(T_k)_{iOut,jIn} \gtrapprox 0$ which properly would be 0 or below.

We already mentioned the accumulation of numerical errors using updates for the inverse matrix in 2.4.3 as well as numerical stability issues in permuting the pivot to compute an LU. The former can be dealt with using periodical restarts, the later by taking stability into account in the Markowitz-like role. In future code an estimator would be useful that detects degradation of numerical stability during the updates and restarts the process if it gets to bad. Alternatively a simple rule as restarting after $\frac{m}{10}$ steps might also work.

An issue which we briefly touched upon in 2.3.5 is the numerical instability introduced by transforming an arbitrary linear program into standard form (cf. [Gri13, 4.5]). In section 2.5 we discuss an auxiliary program that transforms arbitrary linear programs given in the mps-format to standard form accessible in our PostgreSQL implementation. This might be the cause of some of the trouble we had in testing these instances.

2.5. Auxiliary Programs

In order to do some testing we wrote some auxiliary code. First here is an SQL-statement that generates random linear programs:

```
DROP TABLE asimp_randomTMP;
CREATE TABLE asimp_randomTMP (row int, col int, val float8);
INSERT INTO asimp_randomTMP
  SELECT (random()*20)::int AS row, (random()*50)::int AS col,
  (random()*200)::float8 AS val FROM generate_series (1,300) AS x(n);
DROP TABLE asimp_randomTMP2;
CREATE TABLE asimp_randomTMP2 (row int, col int, val float8);
INSERT INTO asimp_randomTMP2
  SELECT row, col, SUM(val) FROM asimp_randomTMP GROUP BY col, row;
DROP TABLE asimp_randomFIN;
CREATE TABLE asimp_randomFIN (row int, col int, val float8);
INSERT INTO asimp_randomFIN
  SELECT * FROM asimp_randomTMP2 ORDER BY col, row ASC;
```

As the random command may create several entries with the same column and row number we simply sum them up. At the end we sort the result.

As pointed out in [DRSL16, pp. 13f] testing with random matrices might not give accurate results regarding practical problems. Therefore we tested our program on several instances of the NetLib-library (see 2.6). However this library represents most of its problems in a compressed MPS format (see A.4 on uncompressing these files and [Mak16, appendix B] on the MPS format). We wrote an auxiliary program `PGLHome/ProblemsFromMPS/sample` that converts the sample MPS files to .csv-files that can be read in by PostgreSQL.

For this we relied on the the package `GLPK` which is freely available. See [Mak16] for the details. See appendix A.4 for some commands and instructions how to use the auxiliary program. Here we will shortly describe the code `sample.c`.

The program has an absolute path hardcoded in line 40:

```
  strcpy(path, "/home/.../PGLHome/ProblemsFromMPS/");
```

Relative to this it reads MPS-files from the directory `MPSfiles` and puts out .csv-files to process with PostgreSQL in `OutputForPostgresQL`. The linear program from the MPS file is not in standard form so we perform some transformations on the variables and even introduce new rows and columns if needed. `sample` also puts a file in `PostprocessFiles`, which contains code to be executable with the bash calculator `bc`. One needs to copy part of the WARNING-output from PostgreSQL generated by `nodeAlgoscanFinSparseWM.c` at the beginning of this file and after executing it in bc one gets the values of the optimal solution for the original system from the MPS file.

We process the MPS file via calling the API provided by `GLPK`. Here is a part of the code showing the header file, how to import the linear program and how to read out the dimensions:

```
#include <glpk.h>
/* ... other code ...*/
  lp = glp_create_prob();
  glp_read_mps(lp, GLP_MPS_DECK, NULL, pathMPS);
  dimr = glp_get_num_rows(lp);
  dimc = glp_get_num_cols(lp);
```

The main part of the code consists in transforming the linear program in standard form. Inequalities are transformed to equations introducing slack variables. Variables that do not obey the standard form restriction $x_i \geq 0$ need also be transformed. Similar code generates `bc`-statements to compute the cost of the optimal solution for the original problem.

We give a sample of files at different stages of the parsing process. First we present the MPS-file that represents the linear program from 2.1

```
NAME          TEST
ROWS
 N  COST
 L  UP
 E  C1
COLUMNS
    X01       COST              1.
    X01       UP                1.
    X01       C1                2.
    X02       COST              2.
    X02       UP                1.
    X02       C1               0.5
RHS
    B         UP             1000.
    B         C1             1250.
ENDATA
```

The .csv-file generated by this is

```
1, 1, 1.000000
1, 2, 1.000000
1, 3, 1.000000
2, 1, 2.000000
2, 2, 0.500000
0, 1, 1.000000
0, 2, 2.000000
1, 0, 1000.000000
2, 0, 1250.000000
```

Solving this with our last implementation gives the following `BC`-code (stripped of WARNINGS):

```
y0 = -3.000000
y1 = 625.000000
y2 = 0.000000
y3 = 375.000000
```

This has to be inserted in the beginning of the `BC`-file produced by the program `sample`:

```
x1=y1+0.000000;
x2=y2+0.000000;
cost = 0;
cost = cost + (x1 * 1.000000);
cost = cost + (x2 * 2.000000);
```

The program in standard form has different dimension than the original one. At the end of `sample.c` there is a line that puts out these dimensions to standard out as well as to a statistics file – if one applies this transformations to several MPS files this statistics file which is a .csv-file is convenient:

```
printf("standard:␣rows:␣%d,␣cols:␣%d\n", outrow, newcol);
fprintf(fileStat, "%s,␣%d,␣%d\n", argv[1], outrow, newcol);
```

Afterwards it terminates. To sum up `sample` reads an MPS file using `GLPK`, transforms it into standard form and puts this transform out as a .csv-file suitable for our simplex algorithm in PostgreSQL. It also provides a bc-file, that can revert the transformation on a solution of the standard form program.

2.6. Testing

We used several different problems for testing our code and we subdivide them in six categories. We created 5 small instances (category 1) to test whether our simplex implementation recognizes infeasibility and unboundedness correctly in principle. We used 2 exercise problems from a high school book to validate if our code finds correct optima – overall they gave rise to 24 instances (category 2) varying the objective function and the direction of optimization. We created 11 instances (category 3) for testing Hopcroft-Karp and Tarjan's algorithm in `nodeAlgoscanFinSparseWM.c`. Then using the SQL-statement from 2.5 we created 4 random instances (category 4) also primarily for debugging the code to create the BTD and the LU-code. We also created 2 small instances (category 5) to test the parsing of our auxiliary program. In all of these the dimensions were bounded by 100. See appendix A.5 for some code that was used in testing. Note that when testing `nodeAlgoscanFinAOnDisk.c` we need sorted versions of the instances.

Finally we worked with 16 instances (category 6) from the NetLib problem library (see [Gay]). From the 98 problems in NetLib 6 were omitted all together as they were not represented in compressed MPS format. The 16 instances were chosen to represent various incarnations of size, sparsity and square vs. rectangular system matrices. Since the various implementations run into errors for many but the smallest instances we also tested several additional smaller instances. In the end we were not able to remove all the bugs.

The NetLib library provides problems that in standard form have up to about 10000 columns and rows. The biggest one is "fit2d" that has 10525 rows and 21024 columns. Of the 92 problems in compressed MPS format 51 are of medium size, i.e., have at least 500 rows. The median of entries per row of these 51 problems is 8.43 with maximum at 48.19 and minimum at 2.84 – so we see the need for sparse implementations.

For the problems in category 1 all our four implementations come to essentially the same result. For the last of the 5 instances there are several different vertices with the optimal value 12. The tableau simplex ends up at a different vertex than the three implemented revised simplex variants. In principle the pivoting strategy should be the same namely Bland's rule, so it is not clear to us why this different behavior arises.

For the problems in category 2 our implementations also show the same and correct behavior for all instances except for instance "asimp_algebra1_ex17cmax" where the first revised simplex implementations runs into seemingly infinite iterations after entering the second phase of the simplex method. This also occurred when using the sorted instance as input. As we did not plan to make this first revised simplex method productive we did not bother to investigate this error further. Again the instance "asimp_algebra1_ex17cmin" has several optimal vertices and the tableau method pics another vertex than the revised simplex.

Category 3 problems were designed for debugging `nodeAlgoscanFinSparseWM.c`. We did not sort these and so did not test `nodeAlgoscanFinAOnDisk.c` with these instances. The remaining three implementations give the same results up to small numerical deviations smaller than 10^{-12}, except for instance "asimp_testhopcroftkarp6" where the first revised simplex implementation again ran into seemingly infinite iterations.

For the random instances in category 4 the first revised simplex implementation ran into seemingly infinite iterations all the time. The tableau implementation and the third revised simplex implementation had matching results. The second revised simplex had an infeasible as result for "asimp_random5" where the first and last implementation produced an optimal result.

From this point on we only tested the tableau simplex (first implementation) and the revised simplex with enhanced matrix inversion (last implementation), as the other two implementation seemed to have bugs and it seemed not worthy to find them. The idea of `nodeAlgoscanFinAOnDisk.c` was anyway just to demonstrate the possibility of leaving the system matrix on disk, which was successful.

To test and debug the auxiliary program that parsed MPS files we used the two problems from category 5. The results of the first and the last version agreed up to small numerical deviations. They agreed with the expected results of the original programs encoded in MPS.

All tests were conducted on the computing server "jordan0" of the Institute of Mathematics at the University Zürich. This machine has two Intel Xeon 6C E5-2640 2.50 GHz processors with 12 cores and 256 GB RAM memory. As we did not implement any parallelization only a single core should have been used at a time. All instances so far did use less than a second, if they completed, although we did not perform exact time measurements.

name	rows	columns	shape	sparsity	optimal value
AFIRO	27	51	1.8888888889	0.0639070443	-4.6475314286E+02
AGG	488	615	1.2602459016	0.0084666134	-3.5991767287E+07
AGG2	516	758	1.4689922481	0.0115435356	-2.0239252356E+07
BLEND	74	114	1.5405405405	0.0617591275	-3.0812149846E+01
FIT2D	10525	21024	1.9975296912	0.0006237324	-6.8464293294E+04
GROW22	1320	1826	1.3833333333	0.0034509941	-1.6083433648E+08
GROW7	420	581	1.3833333333	0.0107900992	-4.7787811815E+07
KB2	52	77	1.4807692308	0.0726773227	-1.7499001299E+03
SCAGR25	471	671	1.4246284501	0.0064200531	-1.4753433061E+07
SCSD8	397	2750	6.9269521411	0.0103814976	9.0499999993E+02
SHARE2B	96	162	1.6875	0.0469393004	-4.1573224074E+02
SHIP04L	402	2166	5.3880597015	0.0097044785	1.7933245380E+06
SHIP04S	402	1506	3.7462686567	0.0095967705	1.7987147004E+06
SIERRA	3263	4751	1.4560220656	0.0005968065	1.5394362184E+07
STAIR	444	626	1.4099099099	0.0138768961	-2.5126695119E+02
STOCFOR2	2157	3045	1.4116828929	0.0014451745	-3.9024408538E+04

Finally we come to discuss the practical test cases from [Gay]. The table above summarizes characteristics of the 16 considered instances. Rows and columns are the dimensions for the problem in standard form as transformed by the auxiliary `sample`. Shape is the fraction of columns over rows – the closer to 1 the more "square" is a problem and the bigger the more "rectangular" it is. Sparsity is the fraction of the non-zeros of the MPS-system divided by the product of rows and columns of the normal form system. We also state the optimal value as provided by NetLib.

We summarize the behavior of the tableau simplex in yet another table. "correct" means that the optimal values match up to an accuracy of less than 0.001%.

name	behavior of tableau simplex implementation
AFIRO	correct
AGG	wrongly returns infeasible
AGG2	correct
BLEND	result correct to within 1%
FIT2D	too large to fit in memory
GROW22	still in first phase after 50 thousand steps
GROW7	correct
KB2	correct
SCAGR25	correct
SCSD8	wrongly returns infeasible
SHARE2B	correct
SHIP04L	wrongly returns infeasible
SHIP04S	wrongly returns infeasible
SIERRA	wrongly returns infeasible and server connection is terminated abnormally
STAIR	still in second phase after 2.5 mio. steps
STOCFOR2	still in first phase after 0.55 mio. steps

For the last revised simplex implementation all instances terminated the server connection abnormally except for the instance "afiro". Using gdb for the instance "kb2" we were able to determine the cause to be a segmentation fault in the function `uFromRightSolve`. Due to time constraints we did not investigate this error further.

Thus "afiro" is the only instance where two implementations give the correct result in terms of the objective function. We also compared the actual solution vector of both variants with the solution vector returned from `GLPK`. They all differed in some of the variable, but also had some in common. There are 27 rows in "afiro", so one would expect as many non-zeros. The output from the tableau version had 20 non-zeros and 4 of these were in the standard form slack variables. That from sparse revised variant had 23 non-zeros, 3 of which were numerical zeros, and 5 where in the standard form slack variables, including 1 numerical zero. The glpk solution had 13 non-zeros. So ignoring slack variable non-zeros and numerical non-zeros for the tableau there were 16 non-zeros, as well as for the sparse revised simplex, but only 13 non-zeros. These occurred in places where the simplex implementation also had non-zeros. For these essential non-zeros the results of our two implementations did not differ, the glpk solution had in addition to the extra 3 zeros another 3 places where the values differed. Since one expects 27 non-zeros anyway we suspect that the instance "afiro" is somewhat degenerate.

To sum up all of our implementations run successfully in some of the test cases and gave the correct output. The second and third implementation still have severe problems, but as both were only intended as preliminary implementation to study principles of designing such a C program we did not bother to fix these issues. One possible cause may lie in explicitly updating a full inverse matrix that might lead to numerical problems. For the artificial test cases our first and forth implementation give agreeing results. For the real world test cases only one of the 16 was correctly solved by the first and the last implementation, giving the correct value for the objective function. In all the tableau implementation performed best as it correctly solved 6 problems and 1 other within an error of 1%.

2.7. Miscellaneous Useful Remarks

In this section we like to discuss some problems that we came across and took much time to resolve in the hope that other programmers might profit from these remarks. As utilities we used emacs as editor, cscope, git to access an online repository, glpk in a utility program to convert mps-files, the shell calculator bc to postprocess and compare results from our lp solver implementation with glpk output. We did not use a debugger although that might have been handy.

We used several directories `PGLHome/buildxx` for the compiled executables. This way we can compare several versions of the code.

At one point we executed the compile commands not from these directories but by accident within a directory from the source code. It did not spark any obvious problems at first, but when we tried to compile modifications of the source it did not work. The likely cause is that "make" detected some .o-files to be already compiled but these were in different directories than the linker looked for them. The only way to fix it was to reinstall from git.

The command "heap_release_fetch" was called in the implementation of a sample scan presented in [Con] to access the input table which is no longer supported in newer versions of

PostgreSQL. Replacing it with “heap_fetch” created a memory leak that had to be dealt with by additional buffer releases. In the end we settled with an adaption of a sequential scan and it was no longer an issue.

The type of data stored in a table in PostgreSQL is “Datum”. If you know that it is an integer or a double you can convert it using “DatumGetInt32, DatumGetFloat8” and back using “Int32GetDatum, Float8GetDatum.”

In newer versions than PostgreSQL 9.4.10 the function `heap_restrpos` is no longer available for sequential scans. If one wants to implement a “from disk” version one would need to use a different scan, since it seems still available for index scans.

At some point when segmentation faults abounded I got interested in the modifications that PostgreSQL does to the C-allocations to implement a rudimentary garbage collector and found the following very helpful post:
`https://blog.pgaddict.com/posts/allocation-set-internals`.

3. Conclusion And Outlook

The goal of this Facharbeit was to modify the syntax of PostgreSQL such that it is possible to issue a command to solve a linear program stored as a relation and to actually implement the simplex algorithm in the back end to solve such a linear program.

We modified the parser of PostgreSQL along the lines of a model example and implemented several variants of tableau simplex and revised simplex algorithm. We tested these implementations and they were able to solve small test cases, these implementations were however not able to reliably solve medium sized or large scale instances.

We investigated several aspects of the simplex method such as different variants of the algorithm, different strategies to enhance sparse linear system solving which is needed for one of the variants or different approaches to accessing data on disk and working memory. We surveyed some of the state to the art literature on these topics and partially worked out implementations of the algorithms presented there.

For testing our code we implemented an auxiliary program that converts MPS files from NetLib library to .csv-files suitable as input for PostgreSQL and solving with the implemented command.

We sketch some possible further directions of work in this line:

- implementing a reliable simplex method that scales with the problem size. For this one of the update strategies surveyed in this work needed to be implemented. Some unresolved issues arising in our implementation as discussed in the section on testing would needed to be addressed.

- generalize to mixed integer linear programming. As the motivation of this project lies in optimizing feeding strategies, and numbers of living animals do not come as fractions, as do numbers of buyable goods, linear programming with fractions represented as floating point numbers will not suffice.

- comparing different solution strategies. We implemented some of the researched strategies and compared them qualitatively in terms of their applicability to certain artificial and real world test instances. A proper evaluation would need to compare thoroughly tested and scaleable version for which one compares running time.

- finding problem libraries that provide large scale instances and writing suitable parser for these. Obviously before doing this one needs a reliable simplex implementation for at least medium sized problems.

A. Appendices

A.1. List Of Modified Files In PostgreSQL

```
postgresql/src/backend/commands/explain.c
postgresql/src/backend/executor/Makefile
postgresql/src/backend/executor/execProcnode.c
postgresql/src/backend/executor/nodeAlgoscan.c
postgresql/src/backend/nodes/copyfuncs.c
postgresql/src/backend/nodes/equalfuncs.c
postgresql/src/backend/nodes/makefuncs.c
postgresql/src/backend/nodes/outfuncs.c
postgresql/src/backend/nodes/readfuncs.c
postgresql/src/backend/optimizer/path/allpaths.c
postgresql/src/backend/optimizer/path/costsize.c
postgresql/src/backend/optimizer/plan/createplan.c
postgresql/src/backend/optimizer/plan/setrefs.c
postgresql/src/backend/optimizer/plan/subselect.c
postgresql/src/backend/optimizer/util/pathnode.c
postgresql/src/backend/optimizer/util/relnode.c
postgresql/src/backend/parser/analyze.c
postgresql/src/backend/parser/gram.y
postgresql/src/backend/parser/parse_clause.c
postgresql/src/backend/parser/parse_relation.c
postgresql/src/backend/parser/parse_target.c
postgresql/src/backend/utils/adt/ruleutils.c
postgresql/src/backend/utils/errcodes.txt
postgresql/src/include/executor/nodeAlgoscan.h
postgresql/src/include/nodes/execnodes.h
postgresql/src/include/nodes/nodes.h
postgresql/src/include/nodes/parsenodes.h
postgresql/src/include/nodes/plannodes.h
postgresql/src/include/nodes/primnodes.h
postgresql/src/include/nodes/relation.h
postgresql/src/include/optimizer/cost.h
postgresql/src/include/optimizer/pathnode.h
postgresql/src/include/parser/kwlist.h
postgresql/src/include/parser/parse_target.h
```

A.2. Commands Within A Build Directory

We assume the file structure from 2.2 and that we are in a directory `PGLHome/buildxx`.

Shell commands to build executables (a “make check” does only work for the unmodified version); the first line has to be executed only once when creating a new build directory, the second each time the source code has changed:

```
../postgresql/configure --prefix=${PWD} --enable-cassert
  --enable-debug CFLAGS="-ggdb -Og -fno-omit-frame-pointer"
make -j 4 && make install
```

Shell commands to create the database db-test and populate it with some data; has to be executed exactly once when the executables are compiled for the dirst time, after that the database is available also to new build directories:

```
./bin/initdb -D ../../data -U postgres -W
./bin/pg_ctl -D ../../data -l /tmp/postgresql.log -o "-F -p 5401" start
./bin/createdb -U postgres -p 5401 db-test
./bin/psql -p 5401 -d db-test -U postgres -f populate.sql
./bin/pg_ctl -D ../../data -l /tmp/postgresql.log -o "-F -p 5401" stop
```

Shell commands to start and stop the database server and client:

```
./bin/pg_ctl -D ../../data -l /tmp/postgresql.log -o "-F -p 5401" start
./bin/psql -p 5401 -U postgres
./bin/pg_ctl -D ../../data -l /tmp/postgresql.log -o "-F -p 5401" stop
```

A.3. Signatures Of Functions In nodeAlgoScan.c

Signatures common to `nodeAlgoscanFinDense.c`, `nodeAlgoscanFinAOnDisk.c` and `nodeAlgoscanFinSparseWM.c`:

```
AlgoScanState * ExecInitAlgoScan(AlgoScan *node, EState *estate,
    int eflags)
static TupleTableSlot * AlgoNext(AlgoScanState *node)
static bool AlgoRecheck(AlgoScanState *node, TupleTableSlot *slot)
int matind(int i, int j, int numCols)
TupleTableSlot * ExecAlgoScan(AlgoScanState *node)
void ExecEndAlgoScan(AlgoScanState *node)
```

Additional signatures for `nodeAlgoscanFinDense.c`:

```
int simplexinitTab(float8 *a, float8 *b, int *baseRToC, int dimr,
    int dimc)
int simplexmainTab(int mode, float8 *a, float8 *b, int *baseRToC,
    int dimr, int dimc)
int simplexinitRev(float8 *a, float8 *b, int *baseRToC, int dimr,
    int dimc)
int simplexmainRev(int mode, float8 *a, float8 *b, int *baseRToC,
    int dimr, int dimc)
```

Additional signatures for `nodeAlgoscanFinAOnDisk.c`:

```
int simplexinitRevSparse(AlgoScanState *node,
    ItemPointer *aStartDiscTid, int *aStartDiscInd,
    struct matElem **aStart, float8 *rhs, float8 *cost, float8 *b,
    int *baseRToC, int dimr, int dimc)
int simplexmainRevSparse(int mode, AlgoScanState *node,
    ItemPointer *aStartDiscTid, int *aStartDiscInd,
    struct matElem **aStart, float8 *rhs, float8 *cost, float8 *b,
    int *baseRToC, int dimr, int dimc)
```

Additional signatures for `nodeAlgoscanFinSparseWM.c`:

```
int shortestBFS(int *baseRToC, int dimr, struct matElem **aStartc,
    int *matchc, int *matchr, int *distc)
int hopcroftKarpDFS(int *baseRToC, int dimr, struct matElem **aStartc,
    int *matchc, int *matchr, int *distc, int cStart)
int hopcroftKarp(int *baseRToC, int dimr, struct matElem **aStartc,
    int *matchc, int *matchr)
void sccRecursive(int *baseRToC, int dimr, struct matElem **aStartc,
    int *sccList, int i, int *index, int *endStack, int *sccCount,
    int *nodeCount, int *indexList, int *loindList, int *stack,
    int *onStack, int *sccStartTmp)
int sccTarjan(int *baseRToC, int dimr, struct matElem **aStartc,
    int *sccStartTmp, int *sccList)
struct matElem *uFromRightSolve(struct matElem *rhsTriang, int start,
    int end, struct matElem **umat)
struct matElem *uFromLeftSolve(struct matElem *rhsTriang, int start,
    int end, struct matElem **umat)
struct matElem *lFromLeftSolve(struct matElem *rhsTriang, int start,
    int end, struct matElem **lmat, int *rpermInv, float8 *rhsSys,
    int *rpermInv2)
struct matElem *luFromLeftSolveBlock(int *baseRToC, int dimr,
    struct matElem *rhsTriang, struct matElem **aStartc, int sccCount,
    int *sccStart, struct matElem **lmat, struct matElem **umat,
    int *rperminv, int *cperm, float8 *rhsSys)
void upluSingBlock(int *baseRToC, int dimr, struct matElem **aStartc,
    int block, int *sccStart, int *sccListInv, struct matElem **lmat,
    struct matElem **umat, int *rperm, int *rperminv, int *cperm,
    float8 *rhsSys)
void uplu(int *baseRToC, int dimr, struct matElem **aStartc,
    int sccCount, int *sccStart, int *sccListInv,
    struct matElem **lmat, struct matElem **umat, int *rperm,
    int *rperminv, int *cperm, float8 *rhsSys)
int simplexinitRevSparse(float8 *nodea, struct matElem **aStartc,
    float8 *rhs, float8 *cost, float8 *b, int *baseRToC, int dimr,
    int dimc)
int simplexmainRevSparse(int mode, float8 *nodea,
    struct matElem **aStartc, float8 *rhs, float8 *cost, float8 *b,
    int *baseRToC, int dimr, int dimc)
```

A.4. Using GLPK And The Auxiliary Program

We downloaded the problem instances from [Gay] and stored files formated in compressed MPS in the folder `PGLHome/ProblemsFromMPS/CMPSfiles`. We use the tool `emps` that uncompresses these files. The C-source code can also be found at [Gay]. In the bash execute in folder `PGLHome/ProblemsFromMPS`:

```
gcc emps.c -o emps
cd CMPSfiles
for i in *; do ../emps $i > ../MPSfiles/$i; done
cd ..
```

We assume that GLPK was installed separately in a directory `GLPK`. The details on how to install it can be found in the manual. To use the standalone solver coming with GLPK and writing the output to `afiro.glpsol` use

```
GLPK/GLPK_install/bin/glpsol --mps MPSfiles/afiro.mps -o afiro.glpsol
```

Before compiling the sample.c one needs to set the absolute path manually in line 40 of the code, e.g.

```
strcpy(path, "/home/.../PGLHome/ProblemsFromMPS/");
```

We assume that `PGLHome/ProblemsFromMPS` has subfoulders `OutputForPostgresQL` and `PostprocessFiles`. To compile and use our program that produces a .csv-file that can be loaded into PostgreSQL from the folder `OutputForPostgresQL` as in A.5 use

```
gcc -I GLPK/GLPK_install/include/ -c sample.c -o sample.o
gcc -L GLPK/GLPK_install/lib sample.o -lglpk -lm -o sample
cd MPSfiles
for i in *; do ../sample $i; done
cd ..
```

When using a build from `nodeAlgoscanFinSparseWM.c` the output written as warnings from PostgreSQL contains a part like “y0=-3; y1=80.0;. . . ”. When copyed at the beginning of a file like `PostprocessFiles/afiro.post`, and stripped from the “WARNING:” running bc on it performs a reverse of the transformations from MPS format to standard form. The `xi` hold the values of the solution for the original program, the `yi` the output from PostgreSQL. `cost` gives the value of the original cost function.

```
bc PostprocessFiles/afiro.post
y1
x1
y50
x30
cost
```

A.5. Test Instances And Their Dimensions

The folder `PGLHome/ProblemsFromMPS/ProbsCSV` contains several .csv-files that were used in testing our code. Due to some issues with privileges one needs to copy these files to `/tmp` before PostgreSQL can read them. When one is in the directory `PGLHome/buildxx` one has to execute the following bash commands assuming that the server is already up:

```
cp ../ProblemsFromMPS/ProbsCSV/asimp_afiro.csv /tmp/.
./bin/psql -p 5401 -U postgres
```

Inside a running PostgreSQL client one loads this data as follows after connecting to the test database:

```
\c db-test
DROP TABLE IF EXISTS asimp_afiro;
CREATE TABLE asimp_afiro (row int, col int, val float8);
COPY asimp_afiro FROM '/tmp/afiro.csv' WITH CSV HEADER;
```

One can have the output of the algorithm displayed by PostgreSQL or stored in an extra table like “xout” in this example:

```
SELECT * FROM asimp_afiro ALGO SIMPLEX DIM (27,51);
DROP TABLE IF EXISTS xout;
SELECT * INTO xout FROM asimp_afiro ALGO SIMPLEX DIM (27,51);
```

In the following we give a list of commands that execute our simplex implementation on the various test cases. Important are the correct dimensions. We split the instances up according to the contest of testing. The commands work from a syntax point of view regardless which implementation is active.

Basic testing: two feasible, an infeasible, an unbounded instance and an unbounded instance that is bounded in the direction of optimality.

```
SELECT * FROM asimp ALGO SIMPLEX DIM (4,7);
SELECT * FROM asimp_test1 ALGO SIMPLEX DIM (2,3);
SELECT * FROM asimp_testinf ALGO SIMPLEX DIM (4,7);
SELECT * FROM asimp_testunbmax ALGO SIMPLEX DIM (4,10);
SELECT * FROM asimp_testunbmin ALGO SIMPLEX DIM (4,10);
```

From a high school book we took 6 exercises (17 a)-c) and 19 a)-c)) and tested maximizing and minimizing these programs, also for ordered variants. They all have dimensions $(4, 8)$, so we simply show one instance, as the other are straight forward modifications.

```
SELECT * FROM asimp_algebra1_ex17amax ALGO SIMPLEX DIM (4,8);
```

We have several instances to specifically test the implementation of the Hopcroft-Karp and Tarjan's algorithm. For Hopcroft-Karp all 7 instances have dimensions $(26, 26)$, so we give just one instance. Most of these instances give only interesting outputs when the debug flags are set as preprocessor commands in `nodeAlgoscanFinSparseWM.c` like `#define HOPCROFTKARPDEBUG_1 true`

```
SELECT * FROM asimp_testhopcroftkarp7 ALGO SIMPLEX DIM (26,26);
SELECT * FROM asimp_testscctarjan1 ALGO SIMPLEX DIM (8,8);
SELECT * FROM asimp_testscctarjan2 ALGO SIMPLEX DIM (8,9);
SELECT * FROM asimp_testscctarjan3 ALGO SIMPLEX DIM (8,9);
SELECT * FROM asimp_testscctarjan4 ALGO SIMPLEX DIM (8,9);
```

There are a few randomly generated instances:

```
SELECT * FROM asimp_random2 ALGO SIMPLEX DIM (20,50);
SELECT * FROM asimp_random3 ALGO SIMPLEX DIM (20,50);
SELECT * FROM asimp_random4 ALGO SIMPLEX DIM (20,50);
SELECT * FROM asimp_random5 ALGO SIMPLEX DIM (10,100);
```

To test the auxiliary code that converts .mps-files to .csv-files we created two small .mps-files. Once the corresponding .csv-files are read in one calls the simplex as follows:

```
SELECT * FROM asimp_testparser ALGO SIMPLEX DIM (3,5);
SELECT * FROM asimp_test2parser ALGO SIMPLEX DIM (2,3);
```

We have 16 instances from the NetLib-library that we used for testing. Other instances could easily be converted using the auxiliary program (see A.4).

```
SELECT * FROM asimp_afiro ALGO SIMPLEX DIM (27,51);
SELECT * FROM asimp_agg ALGO SIMPLEX DIM (488,615);
SELECT * FROM asimp_agg2 ALGO SIMPLEX DIM (516,758);
SELECT * FROM asimp_blend ALGO SIMPLEX DIM (74,114);
SELECT * FROM asimp_fit2d ALGO SIMPLEX DIM (10525,21024);
SELECT * FROM asimp_grow22 ALGO SIMPLEX DIM (1320,1826);
SELECT * FROM asimp_grow7 ALGO SIMPLEX DIM (420,581);
SELECT * FROM asimp_kb2 ALGO SIMPLEX DIM (52,77);
SELECT * FROM asimp_scagr25 ALGO SIMPLEX DIM (471,671);
SELECT * FROM asimp_scsd8 ALGO SIMPLEX DIM (397,2750);
SELECT * FROM asimp_share2b ALGO SIMPLEX DIM (96,162);
SELECT * FROM asimp_ship04l ALGO SIMPLEX DIM (402,2166);
SELECT * FROM asimp_ship04s ALGO SIMPLEX DIM (402,1506);
SELECT * FROM asimp_sierra ALGO SIMPLEX DIM (3263,4751);
SELECT * FROM asimp_stair ALGO SIMPLEX DIM (444,626);
SELECT * FROM asimp_stocfor2 ALGO SIMPLEX DIM (2157,3045);
```

Bibliography

[AB] P. Amestoy and A. Buttari. Sparse linear algebra: Direct methods. amestoy.perso.enseeiht.fr/COURS/ALC_2012_2013.pdf. Accessed: 2017-02-13.

[Con] N. Conway. Introduction to hacking postgresql. www.neilconway.org/talks/hacking/. Accessed: 2016-12-03.

[CSRL01] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[Dan48] G. Dantzig. Programming in a linear structure. Technical report, U.S. Air Force Comptroller, USAF, Washington, D.C., 1948.

[DFU11] C. Dubey, U. Feige, and W. Unger. Hardness results for approximating the bandwidth. *Journal of Computer and System Sciences*, 77(1):62–90, 2011.

[DRSL16] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. Technical report, Department of Computer Science and Engineering, Texas A&M Univ, faculty.cse.tamu.edu/davis/publications_files/survey_tech_report.pdf, 2016.

[Gay] D. M. Gay. Netlib repository lp. www.netlib.org/lp/data/. Accessed: 2017-04-27.

[Gri13] P. Gritzmann. *Grundlagen der Mathematischen Optimierung*. Springer, 2013.

[Hala] J. Hall. The practical revised simplex method (part 1). www.maths.ed.ac.uk/hall/RealSimplex/25_01_07_talk1.pdf. Accessed: 2017-01-26.

[Halb] J. Hall. The practical revised simplex method (part 2). www.maths.ed.ac.uk/hall/RealSimplex/25_01_07_talk2.pdf. Accessed: 2017-01-26.

[HH13] Q. Huangfu and J. A. J. Hall. Novel update techniques for the revised simplex method. Technical report, School of Mathematics, University of Edinburgh, www.maths.ed.ac.uk/hall/HuHa12/ERGO-13-001.pdf, 2013.

[LY16] D. G. Luenberger and Y. Ye. *Linear and Nonlinear Programming*. Springer, 2016.

[Mak16] A. Makhorin. *GNU Linear Programming Kit*, reference manual for glpk version 4.58 edition, 2016.

[Möh] R. Möhring. Einführung in die lineare und kombinatorische optimierung ws2013, lecture 9, revised simplex method. `https://www.coga.tu-berlin.de/fileadmin/i26/download/AG_DiskAlg/FG_KombOptGraphAlg/teaching/adm1ws13/lectures/lect_col_09.pdf`. Accessed: 2017-02-02.

[Sau] M. Saunders. Large-scale numerical optimization: Notes 8: Basis updates. `web.stanford.edu/class/msande318/notes/notes08-updates.pdf`. Accessed: 2017-02-13.

[Sch99] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley-Interscience, 1999.

[vP16] Laurynas Šikšnys and Torben Bach Pedersen. Solvedb: Integrating optimization problem solvers into sql databases. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, SSDBM '16, pages 14:1–14:12, New York, NY, USA, 2016. ACM.

[Wik17a] Wikipedia. Expander graph — wikipedia, the free encyclopedia, 2017. Accessed: 2017-06-11.

[Wik17b] Wikipedia. Hopcroft–karp algorithm — wikipedia, the free encyclopedia, 2017. Accessed 2017-02-17.

[Wik17c] Wikipedia. Sherman–morrison formula — wikipedia, the free encyclopedia, 2017. Accessed 2017-06-08.

[Wik17d] Wikipedia. Tarjan's strongly connected components algorithm — wikipedia, the free encyclopedia, 2017. Accessed 2017-02-17.