Linear Programming

[V. ch7]: Implementation Considerations

Phillip Keldenich   Ahmad Moradi

Department of Computer Science
Algorithms Department
TU Braunschweig

January 10, 2023

Recap

Implementation and Runtime

Before the Christmas break, we rewrote Simplex in matrix notation.

$$\begin{aligned} \max_{x} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad i = 1, 2, \cdots, m \\ & x_j \geq 0, \quad j = 1, 2, \cdots, n \end{aligned}$$

We introduced slack variables as follows:

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j, \qquad i = 1, \ldots, m$$

w_i is renamed as x_{n+i}.

With these slack variables, we wrote our problem in matrix form:

$$\begin{aligned} \max_x \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

where

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & 1 & & & \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & & 1 & & \\ \vdots & \vdots & \ddots & \vdots & & & \ddots & \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} & & & & 1 \end{pmatrix}, b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}, c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \\ 0 \\ \vdots \\ 0 \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ x_{n+1} \\ \vdots \\ x_{n+m} \end{pmatrix}$$

We reordered the variables (columns of A, components of c, x) depending on the sets $\mathcal{B}, \mathcal{N}$ of basic and non-basic variables such that the basic variables come first. Note that $\mathcal{B}, \mathcal{N}$ change in each Simplex iteration.

We wrote A and x in a partitioned-matrix form as: $A = [B \;\; N]\,, x = \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix}.$

We reordered the variables (columns of A, components of c, x) depending on the sets $\mathcal{B}, \mathcal{N}$ of basic and non-basic variables such that the basic variables come first. Note that $\mathcal{B}, \mathcal{N}$ change in each Simplex iteration.

We wrote A and x in a partitioned-matrix form as: $A = [B \;\; N]\,, x = \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix}$.

We also wrote

$$Ax = [B \;\; N] \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} = Bx_{\mathcal{B}} + Nx_{\mathcal{N}},$$

$$c^T x = \begin{bmatrix} c_{\mathcal{B}} \\ c_{\mathcal{N}} \end{bmatrix}^T \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} = c_{\mathcal{B}}^T x_{\mathcal{B}} + c_{\mathcal{N}}^T x_{\mathcal{N}}.$$

We reordered the variables (columns of A, components of c, x) depending on the sets $\mathcal{B}, \mathcal{N}$ of basic and non-basic variables such that the basic variables come first. Note that $\mathcal{B}, \mathcal{N}$ change in each Simplex iteration.

We wrote A and x in a partitioned-matrix form as: $A = [B \;\; N]\,, x = \begin{bmatrix} x_\mathcal{B} \\ x_\mathcal{N} \end{bmatrix}$.

We also wrote

$$Ax = [B \;\; N] \begin{bmatrix} x_\mathcal{B} \\ x_\mathcal{N} \end{bmatrix} = Bx_\mathcal{B} + Nx_\mathcal{N},$$

$$c^T x = \begin{bmatrix} c_\mathcal{B} \\ c_\mathcal{N} \end{bmatrix}^T \begin{bmatrix} x_\mathcal{B} \\ x_\mathcal{N} \end{bmatrix} = c_\mathcal{B}^T x_\mathcal{B} + c_\mathcal{N}^T x_\mathcal{N}.$$

Previously, we used dictionaries to express the basic values $x_\mathcal{B}$ in terms of non-basic variables. We can do that with matrices as well:

$$Ax = Bx_\mathcal{B} + Nx_\mathcal{N} = b \Leftrightarrow x_\mathcal{B} = B^{-1}b - B^{-1}Nx_\mathcal{N}$$

We reordered the variables (columns of A, components of c, x) depending on the sets $\mathcal{B}, \mathcal{N}$ of basic and non-basic variables such that the basic variables come first. Note that $\mathcal{B}, \mathcal{N}$ change in each Simplex iteration.

We wrote A and x in a partitioned-matrix form as: $A = [B \;\; N]\,, x = \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix}$.

We also wrote

$$Ax = [B \;\; N] \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} = Bx_{\mathcal{B}} + Nx_{\mathcal{N}},$$

$$c^T x = \begin{bmatrix} c_{\mathcal{B}} \\ c_{\mathcal{N}} \end{bmatrix}^T \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} = c_{\mathcal{B}}^T x_{\mathcal{B}} + c_{\mathcal{N}}^T x_{\mathcal{N}}.$$

Previously, we used dictionaries to express the basic values $x_{\mathcal{B}}$ in terms of non-basic variables. We can do that with matrices as well:

$$Ax = Bx_{\mathcal{B}} + Nx_{\mathcal{N}} = b \Leftrightarrow x_{\mathcal{B}} = B^{-1}b - B^{-1}Nx_{\mathcal{N}}$$

The objective function was also expressed in terms of non-basic variables:

$$\begin{aligned} \zeta &= c_{\mathcal{B}}^T x_{\mathcal{B}} + c_{\mathcal{N}}^T x_{\mathcal{N}} \\ &= c_{\mathcal{B}}^T \left(B^{-1}b - B^{-1}Nx_{\mathcal{N}}\right) + c_{\mathcal{N}}^T x_{\mathcal{N}} \\ &= c_{\mathcal{B}}^T B^{-1}b - \left(\left(B^{-1}N\right)^T c_{\mathcal{B}} - c_{\mathcal{N}}\right)^T x_{\mathcal{N}} \end{aligned}$$

We reordered the variables (columns of A, components of c, x) depending on the sets $\mathcal{B}, \mathcal{N}$ of basic and non-basic variables such that the basic variables come first. Note that $\mathcal{B}, \mathcal{N}$ change in each Simplex iteration.

We wrote A and x in a partitioned-matrix form as: $A = [B \;\; N]\,, x = \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix}$.

We also wrote

$$Ax = [B \;\; N] \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} = Bx_{\mathcal{B}} + Nx_{\mathcal{N}},$$

$$c^T x = \begin{bmatrix} c_{\mathcal{B}} \\ c_{\mathcal{N}} \end{bmatrix}^T \begin{bmatrix} x_{\mathcal{B}} \\ x_{\mathcal{N}} \end{bmatrix} = c_{\mathcal{B}}^T x_{\mathcal{B}} + c_{\mathcal{N}}^T x_{\mathcal{N}}.$$

Previously, we used dictionaries to express the basic values $x_{\mathcal{B}}$ in terms of non-basic variables. We can do that with matrices as well:

$$Ax = Bx_{\mathcal{B}} + Nx_{\mathcal{N}} = b \Leftrightarrow x_{\mathcal{B}} = B^{-1}b - B^{-1}Nx_{\mathcal{N}}$$

The objective function was also expressed in terms of non-basic variables:

$$\begin{aligned} \zeta &= c_{\mathcal{B}}^T x_{\mathcal{B}} + c_{\mathcal{N}}^T x_{\mathcal{N}} \\ &= c_{\mathcal{B}}^T \left(B^{-1}b - B^{-1}Nx_{\mathcal{N}}\right) + c_{\mathcal{N}}^T x_{\mathcal{N}} \\ &= c_{\mathcal{B}}^T B^{-1}b - \left(\left(B^{-1}N\right)^T c_{\mathcal{B}} - c_{\mathcal{N}}\right)^T x_{\mathcal{N}} \end{aligned}$$

Question: How do we obtain the dictionary solution (basic solution)?

We also considered the dual Simplex and corresponding dictionaries. The dual slacks are complementary to primal originals and vice versa. We accordingly re-order the dual variables

$$(z_1, \ldots, z_n, y_1, \ldots, y_m) = (z_1, \ldots, z_n, z_{n+1}, \ldots, z_{n+m}).$$

We also considered the dual Simplex and corresponding dictionaries. The dual slacks are complementary to primal originals and vice versa. We accordingly re-order the dual variables

$$(z_1, \ldots, z_n, y_1, \ldots, y_m) = (z_1, \ldots, z_n, z_{n+1}, \ldots, z_{n+m}).$$

We also split the dual into basic and non-basic parts $z_{\mathcal{B}}, z_{\mathcal{N}}$; a dictionary expressed $z_{\mathcal{N}}$ in terms of $z_{\mathcal{B}}$. With matrices:

$$z_{\mathcal{N}} = \left(\left(B^{-1}N\right)^T c_{\mathcal{B}} - c_{\mathcal{N}}\right) + B^{-1}N z_{\mathcal{B}}.$$

We also considered the dual Simplex and corresponding dictionaries. The dual slacks are complementary to primal originals and vice versa. We accordingly re-order the dual variables

$$(z_1, \ldots, z_n, y_1, \ldots, y_m) = (z_1, \ldots, z_n, z_{n+1}, \ldots, z_{n+m}).$$

We also split the dual into basic and non-basic parts $z_\mathcal{B}, z_\mathcal{N}$; a dictionary expressed $z_\mathcal{N}$ in terms of $z_\mathcal{B}$. With matrices:

$$z_\mathcal{N} = \left(\left(B^{-1}N\right)^T c_\mathcal{B} - c_\mathcal{N}\right) + B^{-1}N z_\mathcal{B}.$$

Primal and dual dictionary solutions x^*, z^* are obtained with $x_\mathcal{N} = 0, z_\mathcal{B} = 0$:

$$x_\mathcal{B}^* = B^{-1}b, x_\mathcal{N}^* = 0, z_\mathcal{B}^* = 0, z_\mathcal{N}^* = \left(\left(B^{-1}N\right)^T c_\mathcal{B} - c_\mathcal{N}\right), \zeta^* = c_\mathcal{B}^T x_\mathcal{B}^*.$$

We also considered the dual Simplex and corresponding dictionaries. The dual slacks are complementary to primal originals and vice versa. We accordingly re-order the dual variables

$$(z_1, \dots, z_n, y_1, \dots, y_m) = (z_1, \dots, z_n, z_{n+1}, \dots, z_{n+m}).$$

We also split the dual into basic and non-basic parts $z_{\mathcal{B}}, z_{\mathcal{N}}$; a dictionary expressed $z_{\mathcal{N}}$ in terms of $z_{\mathcal{B}}$. With matrices:

$$z_{\mathcal{N}} = \left(\left(B^{-1}N\right)^T c_{\mathcal{B}} - c_{\mathcal{N}} \right) + B^{-1}N z_{\mathcal{B}}.$$

Primal and dual dictionary solutions x^*, z^* are obtained with $x_{\mathcal{N}} = 0, z_{\mathcal{B}} = 0$:

$$x_{\mathcal{B}}^* = B^{-1}b, x_{\mathcal{N}}^* = 0, z_{\mathcal{B}}^* = 0, z_{\mathcal{N}}^* = \left(\left(B^{-1}N\right)^T c_{\mathcal{B}} - c_{\mathcal{N}} \right), \zeta^* = c_{\mathcal{B}}^T x_{\mathcal{B}}^*.$$

Primal dictionary:

$$\begin{aligned} \zeta\ &= \zeta^* - (z_{\mathcal{N}}^*)^T x_{\mathcal{N}} \\ x_{\mathcal{B}} &= x_{\mathcal{B}}^* - B^{-1}N x_{\mathcal{N}} \end{aligned}$$

We also considered the dual Simplex and corresponding dictionaries. The dual slacks are complementary to primal originals and vice versa. We accordingly re-order the dual variables

$$(z_1, \dots, z_n, y_1, \dots, y_m) = (z_1, \dots, z_n, z_{n+1}, \dots, z_{n+m}).$$

We also split the dual into basic and non-basic parts $z_{\mathcal{B}}, z_{\mathcal{N}}$; a dictionary expressed $z_{\mathcal{N}}$ in terms of $z_{\mathcal{B}}$. With matrices:

$$z_{\mathcal{N}} = \left(\left(B^{-1}N\right)^T c_{\mathcal{B}} - c_{\mathcal{N}} \right) + B^{-1} N z_{\mathcal{B}}.$$

Primal and dual dictionary solutions x^*, z^* are obtained with $x_{\mathcal{N}} = 0, z_{\mathcal{B}} = 0$:

$$x^*_{\mathcal{B}} = B^{-1}b, x^*_{\mathcal{N}} = 0, z^*_{\mathcal{B}} = 0, z^*_{\mathcal{N}} = \left(\left(B^{-1}N\right)^T c_{\mathcal{B}} - c_{\mathcal{N}} \right), \zeta^* = c^T_{\mathcal{B}} x^*_{\mathcal{B}}.$$

Primal dictionary:

$$\begin{aligned} \zeta &= \zeta^* - (z^*_{\mathcal{N}})^T x_{\mathcal{N}} \\ x_{\mathcal{B}} &= x^*_{\mathcal{B}} - B^{-1} N x_{\mathcal{N}} \end{aligned}$$

Dual dictionary:

$$\begin{aligned} -\xi &= -\zeta^* - (x^*_{\mathcal{B}})^T z_{\mathcal{B}} \\ z_{\mathcal{N}} &= z^*_{\mathcal{N}} + B^{-1} N z_{\mathcal{B}} \end{aligned}$$

Recap

Implementation and Runtime

What might be the expensive part of each iteration?

Primal Simplex	Dual Simplex
Suppose $x^*_{\mathcal{B}} \geq 0$	Suppose $z^*_{\mathcal{N}} \geq 0$
while ($z^*_{\mathcal{N}} \ngeq 0$) {	while ($x^*_{\mathcal{B}} \ngeq 0$) {
pick $j \in \{j \in \mathcal{N} : z^*_j < 0\}$	pick $i \in \{i \in \mathcal{B} : x^*_i < 0\}$
$\Delta x_{\mathcal{B}} = B^{-1} N e_j$	$\Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_i$
$t = \left(\max_{i \in \mathcal{B}} \frac{\Delta x_i}{x^*_i} \right)^{-1}$	$s = \left(\max_{j \in \mathcal{N}} \frac{\Delta z_j}{z^*_j} \right)^{-1}$
pick $i \in \text{argmax}_{i \in \mathcal{B}} \frac{\Delta x_i}{x^*_i}$	pick $j \in \text{argmax}_{j \in \mathcal{N}} \frac{\Delta z_j}{z^*_j}$
$\Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_i$	$\Delta x_{\mathcal{B}} = B^{-1} N e_j$
$s = \frac{z^*_j}{\Delta z_j}$	$t = \frac{x^*_i}{\Delta x_i}$
$x^*_j \leftarrow t$	$x^*_j \leftarrow t$
$x^*_{\mathcal{B}} \leftarrow x^*_{\mathcal{B}} - t \Delta x_{\mathcal{B}}$	$x^*_{\mathcal{B}} \leftarrow x^*_{\mathcal{B}} - t \Delta x_{\mathcal{B}}$
$z^*_i \leftarrow s$	$z^*_i \leftarrow s$
$z^*_{\mathcal{N}} \leftarrow z^*_{\mathcal{N}} - s \Delta z_{\mathcal{N}}$	$z^*_{\mathcal{N}} \leftarrow z^*_{\mathcal{N}} - s \Delta z_{\mathcal{N}}$
$\mathcal{B} \leftarrow \mathcal{B} \setminus \{i\} \cup \{j\}$	$\mathcal{B} \leftarrow \mathcal{B} \setminus \{i\} \cup \{j\}$
}	}

Most expensive operations (primal, dual is analogous): $\Delta x_{\mathcal{B}} = B^{-1} N e_j, \Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_i$.

Most expensive operations (primal, dual is analogous): $\Delta x_{\mathcal{B}} = B^{-1}Ne_j, \Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_i$.

A naïve implementation using dictionaries essentially has running time $O(m(m+n))$ per iteration. Essentially, the entire dictionary needs to be rewritten in each iteration (during substitution). With very careful implementation, one can technically get away with $O(1)$ time per updated entry.

Most expensive operations (primal, dual is analogous): $\Delta x_{\mathcal{B}} = B^{-1} N e_j, \Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_i$.

A naïve implementation using dictionaries essentially has running time $O(m(m+n))$ per iteration. Essentially, the entire dictionary needs to be rewritten in each iteration (during substitution). With very careful implementation, one can technically get away with $O(1)$ time per updated entry.

In a naive implementation using matrices, we have to find B^{-1} in each iteration — this takes $O(m^\omega \log^k m)$, where k is some constant and $\omega < 2.3729$ is the *matrix multiplication exponent*; straighforward Gaussian elimination takes $O(m^3)$. Matrix multiplication of $B^{-1}N$ also brings in the dependency on n. In practice, we *never* compute B^{-1}.

Most expensive operations (primal, dual is analogous): $\Delta x_{\mathcal{B}} = B^{-1} N e_j, \Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_i$.

A naïve implementation using dictionaries essentially has running time $O(m(m+n))$ per iteration. Essentially, the entire dictionary needs to be rewritten in each iteration (during substitution). With very careful implementation, one can technically get away with $O(1)$ time per updated entry.

In a naive implementation using matrices, we have to find B^{-1} in each iteration — this takes $O(m^\omega \log^k m)$, where k is some constant and $\omega < 2.3729$ is the *matrix multiplication exponent*; straighforward Gaussian elimination takes $O(m^3)$. Matrix multiplication of $B^{-1}N$ also brings in the dependency on n. In practice, we *never* compute B^{-1}.

$\Delta x_{\mathcal{B}} = B^{-1} N e_j = B^{-1} a_j$ is the solution to $Bx = a_j$.

Most expensive operations (primal, dual is analogous): $\Delta x_{\mathcal{B}} = B^{-1} N e_j, \Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_i$.

A naïve implementation using dictionaries essentially has running time $O(m(m+n))$ per iteration. Essentially, the entire dictionary needs to be rewritten in each iteration (during substitution). With very careful implementation, one can technically get away with $O(1)$ time per updated entry.

In a naive implementation using matrices, we have to find B^{-1} in each iteration — this takes $O(m^\omega \log^k m)$, where k is some constant and $\omega < 2.3729$ is the *matrix multiplication exponent*; straighforward Gaussian elimination takes $O(m^3)$. Matrix multiplication of $B^{-1}N$ also brings in the dependency on n. In practice, we *never* compute B^{-1}.

$\Delta x_{\mathcal{B}} = B^{-1} N e_j = B^{-1} a_j$ is the solution to $Bx = a_j$.
$\Delta z_{\mathcal{N}} = -N^T v$, where v is the solution to $B^T v = e_i$ (proof based on $(B^T)^{-1} = (B^{-1})^T$).

How do we solve these systems?

How do we solve these systems?

If we look at solving the linear equation systems in isolation, we gain almost nothing.

How do we solve these systems?

If we look at solving the linear equation systems in isolation, we gain almost nothing. However, if we find a way to preprocess B such that

- solving the linear equation systems is easy/fast,

How do we solve these systems?

If we look at solving the linear equation systems in isolation, we gain almost nothing. However, if we find a way to preprocess B such that

- solving the linear equation systems is easy/fast,
- updating the preprocessed datastructure to incorporate a base change is easy/fast,

we can speed up the process.

How do we solve these systems?

If we look at solving the linear equation systems in isolation, we gain almost nothing. However, if we find a way to preprocess B such that

- solving the linear equation systems is easy/fast,
- updating the preprocessed datastructure to incorporate a base change is easy/fast,

we can speed up the process.

Idea: LU factorization $B = LU$, where L is a lower and U is an upper triangular matrix.

How do we solve these systems?

If we look at solving the linear equation systems in isolation, we gain almost nothing. However, if we find a way to preprocess B such that

- solving the linear equation systems is easy/fast,
- updating the preprocessed datastructure to incorporate a base change is easy/fast,

we can speed up the process.

Idea: LU factorization $B = LU$, where L is a lower and U is an upper triangular matrix.
Basic approach: Gaussian elimination (there are faster methods).

How do we solve these systems?

If we look at solving the linear equation systems in isolation, we gain almost nothing. However, if we find a way to preprocess B such that

- solving the linear equation systems is easy/fast,
- updating the preprocessed datastructure to incorporate a base change is easy/fast,

we can speed up the process.

Idea: LU factorization $B = LU$, where L is a lower and U is an upper triangular matrix.
Basic approach: Gaussian elimination (there are faster methods).

See example on the board (for later reference, see Chapter 8 in the reference book by Vanderbei).

USING LU-FACTORIZATIONS

Assume we have an LU-factorization $B = LU$ and want to solve $Bx = LUx = y$. How can we do that quickly?

Using LU-Factorizations

Assume we have an LU-factorization $B = LU$ and want to solve $Bx = LUx = y$. How can we do that quickly?

First, we solve $Lz = y$ for z, then $Ux = z$ for x. How to solve the individual systems?

Using LU-Factorizations

Assume we have an LU-factorization $B = LU$ and want to solve $Bx = LUx = y$. How can we do that quickly?

First, we solve $Lz = y$ for z, then $Ux = z$ for x. How to solve the individual systems?

Approach: Forward substitution (for lower triangular matrices) and backward substitution (for upper triangular matrices).
Example: See board (for later reference, see Chapter 8 of the Vanderbei book).

USING LU-FACTORIZATIONS

Assume we have an LU-factorization $B = LU$ and want to solve $Bx = LUx = y$. How can we do that quickly?

First, we solve $Lz = y$ for z, then $Ux = z$ for x. How to solve the individual systems?

Approach: Forward substitution (for lower triangular matrices) and backward substitution (for upper triangular matrices).
Example: See board (for later reference, see Chapter 8 of the Vanderbei book).

What is the runtime of forward/backward substitution?

USING LU-FACTORIZATIONS

Assume we have an LU-factorization $B = LU$ and want to solve $Bx = LUx = y$. How can we do that quickly?

First, we solve $Lz = y$ for z, then $Ux = z$ for x. How to solve the individual systems?

Approach: Forward substitution (for lower triangular matrices) and backward substitution (for upper triangular matrices).
Example: See board (for later reference, see Chapter 8 of the Vanderbei book).

What is the runtime of forward/backward substitution?

Note that $B^T = (LU)^T = U^T L^T$ gives an LU-factorization of B^T, which also allows the second solve we need in Simplex.

Sparsity

One very important notion is *sparsity*.

Sparsity

One very important notion is *sparsity*. How many zeros (compared to all entries) do our matrices and vectors have? A sparse matrix (as opposed to a dense matrix) has only relatively few non-zeros.

Sparsity

One very important notion is *sparsity*. How many zeros (compared to all entries) do our matrices and vectors have? A sparse matrix (as opposed to a dense matrix) has only relatively few non-zeros.

In very many practical applications, only a small fraction of matrix entries is non-zero (also, slack variables). Special datastructures storing only non-zeros and algorithms adapted to them can make use of this to reduce the amount of work if the matrix is sparse. That's also why we left zeros blank in the examples!

SPARSITY

One very important notion is *sparsity*. How many zeros (compared to all entries) do our matrices and vectors have? A sparse matrix (as opposed to a dense matrix) has only relatively few non-zeros.

In very many practical applications, only a small fraction of matrix entries is non-zero (also, slack variables). Special datastructures storing only non-zeros and algorithms adapted to them can make use of this to reduce the amount of work if the matrix is sparse. That's also why we left zeros blank in the examples!

We want to keep the number of non-zeros added by our procedures low (minimize the so-called *fill-in*). Strictly minimizing the fill-in of an LU-factorization is NP-hard (but there are decent heuristics).

Sparsity

One very important notion is *sparsity*. How many zeros (compared to all entries) do our matrices and vectors have? A sparse matrix (as opposed to a dense matrix) has only relatively few non-zeros.

In very many practical applications, only a small fraction of matrix entries is non-zero (also, slack variables). Special datastructures storing only non-zeros and algorithms adapted to them can make use of this to reduce the amount of work if the matrix is sparse. That's also why we left zeros blank in the examples!

We want to keep the number of non-zeros added by our procedures low (minimize the so-called *fill-in*). Strictly minimizing the fill-in of an LU-factorization is NP-hard (but there are decent heuristics).

We can (and sometimes have to) swap rows and columns (this is nothing but relabeling variables and constraints) while computing an LU-factorization. This can be used (heuristically) to reduce fill-in using the so-called minimum degree heuristic:

Sparsity

One very important notion is *sparsity*. How many zeros (compared to all entries) do our matrices and vectors have? A sparse matrix (as opposed to a dense matrix) has only relatively few non-zeros.

In very many practical applications, only a small fraction of matrix entries is non-zero (also, slack variables). Special datastructures storing only non-zeros and algorithms adapted to them can make use of this to reduce the amount of work if the matrix is sparse. That's also why we left zeros blank in the examples!

We want to keep the number of non-zeros added by our procedures low (minimize the so-called *fill-in*). Strictly minimizing the fill-in of an LU-factorization is NP-hard (but there are decent heuristics).

We can (and sometimes have to) swap rows and columns (this is nothing but relabeling variables and constraints) while computing an LU-factorization. This can be used (heuristically) to reduce fill-in using the so-called minimum degree heuristic:

- Before eliminating non-zeros below a pivot in a column, scan for a row with minimum number of uneliminated non-zeros, and swap that row to be the new pivot row.

SPARSITY

One very important notion is *sparsity*. How many zeros (compared to all entries) do our matrices and vectors have? A sparse matrix (as opposed to a dense matrix) has only relatively few non-zeros.

In very many practical applications, only a small fraction of matrix entries is non-zero (also, slack variables). Special datastructures storing only non-zeros and algorithms adapted to them can make use of this to reduce the amount of work if the matrix is sparse. That's also why we left zeros blank in the examples!

We want to keep the number of non-zeros added by our procedures low (minimize the so-called *fill-in*). Strictly minimizing the fill-in of an LU-factorization is NP-hard (but there are decent heuristics).

We can (and sometimes have to) swap rows and columns (this is nothing but relabeling variables and constraints) while computing an LU-factorization. This can be used (heuristically) to reduce fill-in using the so-called minimum degree heuristic:

- Before eliminating non-zeros below a pivot in a column, scan for a row with minimum number of uneliminated non-zeros, and swap that row to be the new pivot row.
- Then scan the uneliminated non-zeros in this row and select the one in whose column there are the fewest possible uneliminated non-zeros. Swap this column to be the new pivot column.

SPARSITY

One very important notion is *sparsity*. How many zeros (compared to all entries) do our matrices and vectors have? A sparse matrix (as opposed to a dense matrix) has only relatively few non-zeros.

In very many practical applications, only a small fraction of matrix entries is non-zero (also, slack variables). Special datastructures storing only non-zeros and algorithms adapted to them can make use of this to reduce the amount of work if the matrix is sparse. That's also why we left zeros blank in the examples!

We want to keep the number of non-zeros added by our procedures low (minimize the so-called *fill-in*). Strictly minimizing the fill-in of an LU-factorization is NP-hard (but there are decent heuristics).

We can (and sometimes have to) swap rows and columns (this is nothing but relabeling variables and constraints) while computing an LU-factorization. This can be used (heuristically) to reduce fill-in using the so-called minimum degree heuristic:

- Before eliminating non-zeros below a pivot in a column, scan for a row with minimum number of uneliminated non-zeros, and swap that row to be the new pivot row.
- Then scan the uneliminated non-zeros in this row and select the one in whose column there are the fewest possible uneliminated non-zeros. Swap this column to be the new pivot column.

One step example: see board. In practice, there are more considerations (numerics, other heuristics).

Updating or Reusing a Factorization

From one iteration to the next, the basis matrix B only changes by one column being replaced with another.

Updating or Reusing a Factorization

From one iteration to the next, the basis matrix B only changes by one column being replaced with another. In other words, $B_{\text{new}} = B + (a_j - a_i)e_i^T$.

Updating or Reusing a Factorization

From one iteration to the next, the basis matrix B only changes by one column being replaced with another. In other words, $B_{\text{new}} = B + (a_j - a_i)e_i^T$.

We can reinterpret this as $B_{\text{new}} = B\underbrace{\left(I + B^{-1}(a_j - a_i)e_i^T\right)}_{=:E}$.

Updating or Reusing a Factorization

From one iteration to the next, the basis matrix B only changes by one column being replaced with another. In other words, $B_{\text{new}} = B + (a_j - a_i)e_i^T$.

We can reinterpret this as $B_{\text{new}} = B\underbrace{\left(I + B^{-1}(a_j - a_i)e_i^T\right)}_{=:E}$.

We have $B^{-1}a_j = \Delta x_{\mathcal{B}}$ (which we computed anyway) and $B^{-1}a_i = e_i$.

Updating or Reusing a Factorization

From one iteration to the next, the basis matrix B only changes by one column being replaced with another. In other words, $B_{\text{new}} = B + (a_j - a_i)e_i^T$.

We can reinterpret this as $B_{\text{new}} = B\underbrace{\left(I + B^{-1}(a_j - a_i)e_i^T\right)}_{=:E}$.

We have $B^{-1}a_j = \Delta x_{\mathcal{B}}$ (which we computed anyway) and $B^{-1}a_i = e_i$.
Therefore, $E = (I + \underbrace{(\Delta x_{\mathcal{B}} - e_i)}_{=:u}\underbrace{e_i^T}_{=:v^T})$.

Updating or Reusing a Factorization

From one iteration to the next, the basis matrix B only changes by one column being replaced with another. In other words, $B_{\text{new}} = B + (a_j - a_i)e_i^T$.

We can reinterpret this as $B_{\text{new}} = B\underbrace{\left(I + B^{-1}(a_j - a_i)e_i^T\right)}_{=:E}$.

We have $B^{-1}a_j = \Delta x_{\mathcal{B}}$ (which we computed anyway) and $B^{-1}a_i = e_i$.
Therefore, $E = (I + \underbrace{(\Delta x_{\mathcal{B}} - e_i)}_{=:u}\underbrace{e_i^T}_{=:v^T})$.

We have $E^{-1} = I - \dfrac{1}{1 + v^T u}uv^T$. Let's prove that (board).

Updating or Reusing a Factorization

From one iteration to the next, the basis matrix B only changes by one column being replaced with another. In other words, $B_{\text{new}} = B + (a_j - a_i)e_i^T$.

We can reinterpret this as $B_{\text{new}} = B\underbrace{\left(I + B^{-1}(a_j - a_i)e_i^T\right)}_{=:E}$.

We have $B^{-1}a_j = \Delta x_{\mathcal{B}}$ (which we computed anyway) and $B^{-1}a_i = e_i$.
Therefore, $E = (I + \underbrace{(\Delta x_{\mathcal{B}} - e_i)}_{=:u}\underbrace{e_i^T}_{=:v^T})$.

We have $E^{-1} = I - \dfrac{1}{1 + v^T u} uv^T$. Let's prove that (board).

Because we can easily invert E (the inverse even has a nice form) we can still solve systems w.r.t. B_{new} instead of B. This works across multiple iterations (i.e., solving $BE_0E_1E_2\cdots E_kx = y$, but becomes less efficient the more Es we add.

Updating or Reusing a Factorization

From one iteration to the next, the basis matrix B only changes by one column being replaced with another. In other words, $B_{\text{new}} = B + (a_j - a_i)e_i^T$.

We can reinterpret this as $B_{\text{new}} = B\underbrace{(I + B^{-1}(a_j - a_i)e_i^T)}_{=:E}$.

We have $B^{-1}a_j = \Delta x_{\mathcal{B}}$ (which we computed anyway) and $B^{-1}a_i = e_i$.
Therefore, $E = (I + \underbrace{(\Delta x_{\mathcal{B}} - e_i)}_{=:u} \underbrace{e_i^T}_{=:v^T})$.

We have $E^{-1} = I - \dfrac{1}{1 + v^T u} uv^T$. Let's prove that (board).

Because we can easily invert E (the inverse even has a nice form) we can still solve systems w.r.t. B_{new} instead of B. This works across multiple iterations (i.e., solving $BE_0E_1E_2 \cdots E_k x = y$, but becomes less efficient the more Es we add.

We can solve this by periodically re-factorizing the new B; in this way, we need much less of the expensive LU-factorizations.

Updating or Reusing a Factorization

From one iteration to the next, the basis matrix B only changes by one column being replaced with another. In other words, $B_{\text{new}} = B + (a_j - a_i)e_i^T$.

We can reinterpret this as $B_{\text{new}} = B \underbrace{(I + B^{-1}(a_j - a_i)e_i^T)}_{=:E}$.

We have $B^{-1}a_j = \Delta x_{\mathcal{B}}$ (which we computed anyway) and $B^{-1}a_i = e_i$.
Therefore, $E = (I + \underbrace{(\Delta x_{\mathcal{B}} - e_i)}_{=:u} \underbrace{e_i^T}_{=:v^T})$.

We have $E^{-1} = I - \frac{1}{1 + v^T u} uv^T$. Let's prove that (board).

Because we can easily invert E (the inverse even has a nice form) we can still solve systems w.r.t. B_{new} instead of B. This works across multiple iterations (i.e., solving $BE_0E_1E_2 \cdots E_k x = y$, but becomes less efficient the more Es we add.

We can solve this by periodically re-factorizing the new B; in this way, we need much less of the expensive LU-factorizations.

The method presented here is only one possibility; it's also possible to actually update the factorization of B. This often leads to suboptimal fill-in and may also run into numerical issues, which means that it also requires re-factorization. For more, see the reference book.

Methods to solve these systems in the most efficient and numerically stable way, in particular those that make use of and maintain sparsity, are actively researched. Many things have to be balanced (numerical stability vs. theoretical efficiency vs. practical efficiency); in practice, it is not always the best theoretical algorithm (w.r.t. O-notation) that is the most useful.

Methods to solve these systems in the most efficient and numerically stable way, in particular those that make use of and maintain sparsity, are actively researched. Many things have to be balanced (numerical stability vs. theoretical efficiency vs. practical efficiency); in practice, it is not always the best theoretical algorithm (w.r.t. O-notation) that is the most useful.

To the best of our knowledge, the best current approach needs $O(d_c^{0.7}m^{1.9} + m^{2+o(1)} + d_c n)$ time for a simplex iteration in which a new LU-factorization is computed, where d_c is the maximum number of non-zeros in any column; this beats the Gaussian elimination (at least in theory even for dense matrices). On the theoretical side, the time needed per iteration is quite difficult to analyze in an amortized fashion, considering multiple Simplex iterations.