Towards Hardware Implementation of the Simplex Algorithm for LP Decoding

Florian Gensheimer  
Optimization Research Group  
University of Kaiserslautern  
67653 Kaiserslautern  
gensheim@mathematik.uni-kl.de

Michael Helmling  
Mathematical Institute  
University of Koblenz-Landau  
56070 Koblenz  
helmling@uni-koblenz.de

Stefan Scholl  
Microelectronic Systems Design Group  
University of Kaiserslautern  
67653 Kaiserslautern  
scholl@eit.uni-kl.de

Abstract—Combining methods of mathematical optimization theory and applications from communications engineering recently led to new approaches for error correction in data transmission systems. In this new discipline, also called LP decoding, researchers so far focussed on theoretical aspects. In this paper, we study how these new algorithms from mathematical theory can be accelerated using dedicated hardware implementations. We address complexity issues of the widely used simplex algorithms and show how new interesting mathematical problems arise if aspects from hardware design are considered.

Index Terms—channel coding, hardware implementation, simplex algorithm, linear programming.

I. INTRODUCTION

Linear programming (LP) is one of the main topics of mathematical optimization. LP problems arise in many economical and industrial areas like production planning, scheduling or logistics. They also play an important role as subproblems in branch-and-bound and cutting-plane algorithms for integer programs, which have a large field of application. The most important algorithm for solving linear programs is the simplex algorithm by G. B. Dantzig (see e.g. [1]). Although the simplex algorithm has an exponential worst-case complexity, it turned out to be very efficient in practice. A rather new application of linear programming is forward error correction or channel coding (see [2], [3] for an introduction).

Channel coding, an engineering discipline, is an essential technique used for correcting errors in digital communication systems that occur during the transmission of data. These transmission errors occur due to bad reception quality, interference between different devices (e.g. mobile phones) or because of physical damage of the devices (storage). Channel coding is used in nearly every communication device today, including smartphones, TV and radio broadcasting, DSL internet access or satellite communications and also in storage devices such as DVDs or USB memory sticks.

A channel coding system consists of two stages: encoder and decoder. The encoder is placed at the transmitter and adds redundant bits before the data is transmitted. The decoder is placed at the receiver and uses the previously added redundancy in order to correct potential transmission errors without the need of retransmission. The decoder is the heart of every channel coding system, since it performs the actual data correction with sophisticated algorithms.

One new approach for error correction decoding is the use of algorithms from the field of mathematical programming: as shown by Feldman et al. [4], it is possible to formulate the decoding problem as a linear program which in turn can be solved and analyzed by methods of mathematical optimization. This symbiosis of linear programming from mathematics and channel coding from engineering is called LP decoding. LP decoding recently led to new interesting mathematical problems and better algorithms for error correction systems. Mathematical optimization algorithms are commonly implemented in software and executed on a platform based on a general purpose processor, such as high performance servers, PCs or laptops, that require large space and power. However, many popular communication devices, like smartphones, have to be small and portable devices with low power consumption, while processing data with high speed.

A general purpose device like a PC processor is designed for great flexibility which allows them to run many different applications. However, if a hardware chip is designed dedicated for just one single algorithm, the chip can be highly optimized for exactly this algorithm. In this case data operations and memory requirements are well known and the chip architecture can be tailored to the specific application’s requirements.

Implementing algorithms as a dedicated hardware circuit shows several advantages over a general purpose hardware:

- speed: speed-ups of several orders of magnitude can be achieved over a PC,
- portability and area: chip area often only several mm²,
- power: power consumption often in the order of milliwatts (PC/laptop: several 10 to several 100 watts).

In the remainder of this paper we consider these dedicated electrical circuits implemented on a chip and call it a hardware implementation or simply hardware. As we will see later, it is a challenging task to implement an algorithm as hardware, because different aspects of the hardware and the algorithm itself have to be considered.

Hardware implementations for the simplex algorithm and LP decoding have not been investigated deeply so far. In [5], a hardware implementation of the simplex algorithm has been presented. However, this implementation uses the standard simplex method without modifications and does not consider LP decoding. For an efficient hardware implementation it
is advantageous to exploit problem specific properties and modifications already at the algorithm level.

As alternatives to the simplex method, interior-point algorithms [6] and a quadratic programming approach [7] for LP decoding were studied in literature. However, both papers only consider software implementations, and the proposed algorithms are substantially more complex to implement in hardware than the simplex method.

While LP decoding so far has been studied mainly from a theoretical point of view, the aim of the present paper is to investigate how LP decoding algorithms can actually be implemented to work in a communication device. We analyze different variants of the simplex algorithm for LP decoding and reveal large differences in hardware complexity. Additionally, we propose a chip architecture of an LP decoder that can be implemented efficiently.

II. THE SIMPLEX ALGORITHM

We start by introducing the fundamentals of linear programming and the simplex method, before reviewing advanced methods based on duality.

A linear program consists of a linear objective function, the value of which is to be optimized, and a set of linear constraints, i.e. (in)equalities, that limit the values of the variables in the program. Any LP can be written in the standard form

\[
\begin{align*}
\text{min} \quad & z = c^T x \\
\text{s.t.} \quad & Ax = b \\
& x \geq 0
\end{align*}
\]  

where \( x \in \mathbb{R}^n \) are the decision variables, \( A \in \mathbb{R}^{m \times n} \) is a matrix with \( m \leq n \), and \( b \in \mathbb{R}^m \), \( c \in \mathbb{R}^n \) are vectors. \( A \) and \( b \) describe the so-called functional constraints (2), while (3) defines the nonnegativity constraints. The vector \( c \in \mathbb{R}^n \) represents the objective function.

Each row of (2) and (3) defines a hyperplane or a halfspace of \( \mathbb{R}^n \), respectively. Thus, the feasible region \( \mathcal{P} = \{ x \in \mathbb{R}^n : Ax = b, x \geq 0 \} \) of the LP is a polyhedron, i.e., the intersection of a finite number of hyperplanes and halfspaces. For a fixed objective value \( z \), (1) is a hyperplane. Geometrically, the minimization can be interpreted as pushing that hyperplane in direction \( -c \) as far as possible without leaving the polyhedron.

The (primal) simplex algorithm is based on the fact that an optimal solution is obtained in a vertex of the polyhedron. The idea of the algorithm is to move iteratively from one vertex of the polyhedron to another in such a way that the sequence of objective function values (1) of the vertices is nonincreasing. Its key observation is that each vertex corresponds to a basis, i.e., a linearly independent size-\( m \) subset of \( A \)'s columns. In order to move to an adjacent vertex, exactly one nonbasic column of \( A \) is exchanged with a basic column. For each nonbasic column a reduced cost value can be computed that tells if adding this column potentially improves the objective value.

In practice, the algorithm operates on the simplex tableau \( T \), a two-dimensional array that contains information about the LP as well as the current basis and reduced costs. Every iteration of the algorithm consists of three main steps:

1) computation of the reduced costs: they determine the column that enters the basis,
2) min-ratio rule: determines the column that leaves the basis,
3) basis exchange: is carried out by a pivot operation on \( T \), i.e., Gaussian elimination step of the form:

\[
\text{for } i \in \{1, \ldots, m\} \setminus \{i^*\} \text{ do }
\]

\[
\text{if } T_{i,j} \neq 0 \text{ then }
T_{i,*} \leftarrow T_{i,*} - \frac{T_{i,j}}{T_{j,j}} \cdot T_{j,*}
\]

The pivot operation constitutes the main computational effort of the simplex algorithm. It is therefore crucial to implement this operation efficiently. This operation is repeated in every iteration until the reduced cost vector is nonnegative, which indicates that the vertex of the current basis is optimal.

A second major approach for solving linear programs stems from duality theory. The basis of this theory is the so-called dual program, a special linear program that corresponds to the original LP. For the linear program in standard form (1)–(3), the dual program (DLP) has the form

\[
\begin{align*}
\text{max} \quad & b^T \pi \\
\text{s.t.} \quad & A^T \pi \leq c
\end{align*}
\]  

where \( \pi \in \mathbb{R}^m \) are the dual variables and \( A^T \pi \leq c \) are the dual constraints. Both linear programs (LP) and (DLP) have the same optimal objective value. Hence, one can equivalently solve the dual problem instead of the program (LP).

The dual simplex algorithm works on the usual simplex tableau \( T \) of (LP). The main difference to the primal simplex algorithm is the fact that the working solution \( x \) is infeasible during execution, while the optimality condition is always fulfilled. This is contrary to the primal simplex which always stays feasible, but does not terminate before optimality is established.

In many applications, like e.g. the LP decoding problem, all variables \( x_i \) have an upper bound of 1. Both the primal and dual simplex algorithm would have to introduce an artificial variable and an additional side constraint for each such variable because the problem has to be transformed into “standard form” (see e.g. [1]). This leads to a larger simplex tableau and a less efficient simplex algorithm. Special versions of the simplex algorithm (both primal and dual) avoid those extra variables and constraints by implicitly handling upper bounds (see [8], [9]).

Linear programs for practical problems often suffer degenerate pivot operations, which means that the step length when moving from one vertex to another reduces to zero, hence the objective function value does not improve. This can dramatically increase the running time and, if occurring frequently, leads to numerical instabilities in the algorithm. Especially for LPs that are highly degenerate—unfortunately, the LP decoding problem was shown to be among those—special care must be taken to avoid degeneracy. We have adapted the
so-called ad-hoc procedure by P. Wolfe [10] to our case which can be implemented with negligible computational overhead and effectively eliminates the influence of degeneracy for the LP decoding problem.

### III. Designing Hardware

In this section we present basics and challenges of hardware design. We show how hardware implementations that are highly optimized for one application can outperform general purpose platforms.

As already mentioned in Section I, dedicated hardware implementations are often faster, smaller and consume less power than software implementations running on a general purpose PC. In general, three design goals of hardware implementation can be identified:

- high calculation speed, i.e., high clock frequency,
- small area (leads to reduced production cost and small chip size), and
- small power consumption (to avoid power supply and heat dissipation problems).

To accelerate an algorithm using hardware, it has to be implemented in form of an electrical digital circuit by the hardware designer. Since software (algorithm) and hardware (microchip) development are fundamentally different, this task is not straightforward. In the following we will present some aspects that become important when implementing algorithms in hardware.

#### A. Memories for Data Storage

Usually a modern PC provides a large amount of memory (up to several gigabytes) for storing data during the calculations, in order to meet the requirements of a wide range of applications. In dedicated hardware implementations the memory sizes can be tailored exactly to the application’s requirement. This allows for large memory reductions (often in the order of kilobytes). Additionally, the algorithm itself can be modified to reduce the amount of required storage, finally resulting in a smaller hardware implementation.

#### B. Number Representation

Another important aspect is the representation of numerical values that are processed by the algorithm. In a digital system, numbers are represented by vectors of bits. Two fundamental ways to interpret bit vectors as numbers are the floating-point and the fixed-point representations [11]. On a PC the values are usually represented by double-precision floating-point numbers with a large number of bits (commonly 64). Double-precision floating-point values provide a very good resolution and a wide range. However, floating-point numbers require complex arithmetic hardware, and are therefore avoided in hardware design whenever possible. Instead, it is desired to use fixed-point numbers which lead to a much lower hardware complexity.

A second question arises consequently: How much bits are sufficient to represent the numbers? Usually a low number of bits allows for faster and smaller hardware units and is therefore beneficial (see Fig. 1). Furthermore, smaller memories can be used for storing data if the number of bits is small. Choosing a low number of bits however leads to a bad resolution of the values and very limited accuracy of the calculations, which may or may not affect the outcome of the algorithm. There is a clear trade-off between calculation accuracy and hardware complexity (i.e., calculation speed, area and power consumption). This has to be considered when implementing algorithms in hardware. The exact number of required bits in the hardware implementation is mostly determined by extensive simulations.

#### C. Arithmetic Units

For arithmetic operations that are carried out in an algorithm, arithmetic hardware units such as adders, multipliers, and so forth have to be implemented. Arithmetic operations have to be used carefully. Assuming fixed-point numbers, additions and subtractions are often cheap to implement. However, multiplications and divisions consume large amounts of hardware area and power and lead to high hardware complexity (see Fig. 1). Also complex arithmetic operations like logarithms, roots, trigonometric functions etc. exhibit high hardware complexity. Sometimes these functions can be simplified by using approximations such as linearization or simple iterative heuristics. Note that multiplications or divisions by powers of two can be implemented very easily by binary shift operations.

The aspects mentioned so far, although this listing is by no means complete, show important differences between software and hardware design. Therefore, the term “complexity” in software engineering is different from “complexity” in hardware design. When dealing with hardware implementations, additional and more sophisticated measures of complexity have to be considered.

### IV. Simplex Performance Study

In this section, we evaluate the variants of the simplex algorithm mentioned in Section II:

- the revised primal simplex,
2 bit representation

<table>
<thead>
<tr>
<th>10</th>
<th>11</th>
<th>00</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

4 different levels, resolution = 1

4 bit representation (with decimal point)

<table>
<thead>
<tr>
<th>100.0</th>
<th>100.1</th>
<th>...</th>
<th>000.0</th>
<th>000.1</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4.0</td>
<td>-3.5</td>
<td>...</td>
<td>0</td>
<td>0.5</td>
<td>...</td>
</tr>
</tbody>
</table>

16 different levels, resolution = 0.5

Fig. 2. example for fixed-point numbers using 2 and 4 bits (2s complement), note the difference in range and resolution

<table>
<thead>
<tr>
<th>Primal</th>
<th>Primal (Bounded)</th>
<th>Dual (Bounded)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(7, 4) Hamming code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>variables</td>
<td>38</td>
<td>31</td>
</tr>
<tr>
<td>constraints</td>
<td>31</td>
<td>24</td>
</tr>
<tr>
<td>tableau size</td>
<td>$32 \times 32$</td>
<td>$25 \times 25$</td>
</tr>
</tbody>
</table>

| (20, 10) LDPC code |
| variables | 120 | 100 | 100 |
| constraints | 100 | 80 | 80 |
| tableau size | $101 \times 101$ | $81 \times 81$ | $81 \times 101$ |

TABLE I

COMPARISON OF LP SIZE PARAMETERS

- the revised primal simplex for bounded variables, and
- the dual simplex for bounded variables.

We analyze their running-time performance and their behavior under limited numerical precision with fixed-point arithmetic. The numerical comparisons are done with the (7, 4) Hamming code and a random (20, 10) LDPC code (see [2] for details).

A. Running-Time Performance

The number of pivot operations, the number of variables and constraints in the LP, and the size of the tableau matrix have major influence on the running-time complexity of the LP solver. The sizes of the LPs for our example decoding problems are shown in Table I. As one can see, the standard primal has the largest size, because it does not handle upper bounds efficiently.

As mentioned in Section II, the simplex iteratively performs pivot steps. The running time is proportional to the number of pivots which may vary for different objective functions. In the channel decoding application, this function is related to the channel noise. Hence, to estimate the running time under realistic conditions, we generate random noise for different signal-to-noise (SNR) ratios; a higher SNR value corresponds to less noise on the channel. In Fig. 3, the relative frequencies of the iteration counts are shown for the (7, 4) Hamming code. The continuous lines show the values for SNR 0. The performance of the primal LP simplex variants proved to be independent of the SNR value. Only the dual simplex benefits substantially from a lower noise level, which is indicated by the dotted curve.

As one can see, the dual simplex method dramatically outperforms the primal variants in terms of iteration counts. An explanation for this effect can be found in the initialization procedure of the simplex algorithms: in the primal simplex, always the same, fixed starting basis is used. On average, the corresponding vertex of the polyhedron will be rather far away from the optimal vertex, such that many pivots are necessary. In contrast, the dual simplex for bounded variables [9] chooses an initial solution that would be optimal in the absence of channel noise. Intuitively, this basis is likely to be “closer” to the optimum even if noise is present. Because this starting solution of the dual simplex is not primal feasible, it cannot be used to speed up the primal algorithms.

Fig. 4 shows for the larger code how the average number of iterations per instance depends on the noise level. Again, the dual simplex is much faster, finishing after 1.66 iterations on average for the highest SNR value.
B. Comparison of Fixed- and Floating-Point Implementation

As outlined in Section III, using fixed-point numbers with low precision is preferable from the hardware point of view. On the other hand, if the numerical precision is too low, round-off errors can break the algorithm in numerous ways, leading to wrong solutions or even infinite loops. Hence a compromise needs to be found between simplicity and correctness. For the LP decoding application, the latter directly translates into the frame error rate, i.e., the average probability of a decoding error. Depending on the usage scenario, a slightly increased error rate might be tolerable, if this in return allows for low-complexity hardware.

For the (7, 4) Hamming code, we compare fixed-point arithmetic with resolutions 16.8, 10.4, and 6.2 with usual double-precision floating-points. Here, the notation \( x.y \) means that each number is represented by \( x \) bits, \( y \) of which constitute the fractional part. This implies that numbers from \( -2^{x-y-1} \) to \( 2^{x-y-1} - 2^{-y} \) can be represented, with a resolution of \( 2^{-y} \).

In our experiments, the dual simplex proves extremely stable, showing practically the same error-correcting performance even with the very poor 6.2 resolution. The primal variants achieve this performance with 16.8; with 10.4 bits the performance drops but is still reasonable, while both primal algorithms are practically useless with the lowest resolution. The resulting error-performance curves for the primal variants are shown in Fig. 5.

For the larger code, we compare the floating-point implementation with fixed-point precisions 16.8 and 8.3. Results are shown in Fig. 6. Again, the dual algorithm is most stable with low precision and practically achieves floating-point performance with 8.3 fixed-point bits. The primal algorithms significantly loose performance with this resolution, but are comparable to the floating-point performance with 16.8.

V. Conclusion

For both codes, the two primal simplex variants largely show the same behavior, whereas the dual simplex for bounded variables performs clearly superior, in terms of both the number of iterations (and thus overall running time) and the robustness against low-precision numerical resolution.

C. Proposed Hardware Architecture

In Fig. 7 we propose a hardware architecture that implements the dual simplex for bounded variables.

It shows the five main parts required by the simplex. The current tableau is stored in the tableau memory. A pivot unit accomplishes pivoting of a column on the fly when it is read. It is followed by an additional unit that performs simple bit flip operations to take care of the variable bounds. The entering variable is determined according to some pricing rule and checks for optimality, i.e., when the simplex has to stop. The leaving variable block is needed to compute the leaving variable using e.g. the min-ratio rule.

Additionally a controller is shown. The controller takes care of the initialization of the other hardware units, keeps track of the basis variable positions, and extracts the values of the variables when the result is declared optimal.

In this paper, we have studied the LP decoding problem with three variants of the simplex algorithm. In terms of running time, the dual simplex for bounded variables has shown to outperform the competing primal variants, finding the optimal solution substantially faster (up to a factor of 50). With respect to hardware implementation complexity, we have also analyzed the behavior of these algorithms under low-precision fixed-point arithmetic. Again, the dual simplex has shown to be most stable, achieving a similar error-correction level as the 64-bit floating-point implementation with 6 and 8 fixed-point bits per number for the Hamming and LDPC code, respectively.
Both the number of iterations and the low size of the number representation are important steps towards a low-power, high-performance hardware LP decoder implementation.

VI. OUTLOOK AND FUTURE RESEARCH

Other improvements of the simplex algorithm, specialized for hardware requirements, might further decrease the complexity of hardware LP decoding. Some possible directions are:

A. Simplex in the Log-Domain

The simplex algorithm usually involves a lot of multiplications and division. However these two operations are costly to implement in hardware, as we have shown in Section III.

One option to transform multiplication and division into a much simpler operation is to perform calculations in the logarithm domain. If the logarithm of all operands is used, a multiplication transforms into an addition and a division into a simple subtraction, thus avoiding costly operations. However this avoidance comes at additional cost for taking logarithm and the exponential function. Further investigations have to show how far the hardware complexity can be reduced by this approach.

B. Adaptive LP Decoding

For larger codes, the complete LP decoding formulation contains a large number of constraints, most of which are not actually needed to describe the optimal solution. Taghavi and Siegel [12] have proposed an adaptive method that starts with an empty LP and inserts inequalities on demand, greatly reducing the size of the number of constraints. Two aspects make this approach particularly appealing: Firstly, as shown in [12] it is possible to upper bound the number of necessary constraints to a very small number, compared to the complete LP. Secondly, inserting additional constraints is possible without any extra operations if one uses the dual simplex algorithm. Since the dual simplex algorithm turned out as the most efficient one by our numerical study in Section IV, using it for adaptive LP decoding is extremely promising.

C. Polyhedral Theory for Fixed-Point Numbers

Mathematically, fixed-point numbers constitute a uniform, discrete grid in the Euclidean space. Ordinary polyhedral theory always assumes a continuous space, thus real numbers with infinite precision. A theoretical study of “discrete” polyhedral theory could, besides being an interesting field of research on its own, lead to more efficient algorithms exploiting the grid structure. Additionally, it might be possible to find theoretical results about numerical stability for given fixed-point precision.

ACKNOWLEDGEMENT

We gratefully acknowledge the Center of Mathematical and Computational Modelling of the University of Kaiserslautern and the German Research Council (DFG) for financial support.

REFERENCES