The Symbolic Interior Point Method

Martin Mladenov
TU Dortmund University, Germany
martin.mladenov@cs.tu-dortmund.de

Vaishak Belle
University of Edinburgh, UK
vaishak@ed.ac.uk

Kristian Kersting
TU Dortmund University, Germany
kristian.kersting@cs.tu-dortmund.de

Abstract

Numerical optimization is arguably the most prominent computational framework in machine learning and AI. It can be seen as an assembly language for hard combinatorial problems ranging from classification and regression in learning, to computing optimal policies and equilibria in decision theory, to entropy minimization in information sciences. Unfortunately, specifying such problems in complex domains involving relations, objects and other logical dependencies is cumbersome at best, requiring considerable expert knowledge, and solvers require models to be painstakingly reduced to standard forms. To overcome this, we introduce a rich modeling framework for optimization problems that allows convenient codification of symbolic structure. Rather than reducing this symbolic structure to a sparse or dense matrix, we represent and exploit it directly using algebraic decision diagrams (ADDs). Combining efficient ADD-based matrix-vector algebra with a matrix-free interior-point method, we develop an engine that can fully leverage the structure of symbolic representations to solve convex linear and quadratic optimization problems. We demonstrate the flexibility of the resulting symbolic-numeric optimizer on decision making and compressed sensing tasks with millions of non-zero entries.

Introduction

A convex quadratic program (QP) is an optimization problem in which a convex quadratic function is minimized over the solution set of a system of linear inequalities. In this paper, we assume that a QP takes on the following standard form

$$\begin{array}{ll} \text{minimize} & c^T x + 1/2 x^T Q x \\ \text{subject to} & Ax = b, x \geq 0, \end{array}$$

where, Q is a symmetric positive semi-definite matrix. Whenever $Q = 0$, we speak of linear programs (LPs). Convex QPs are commonly solved by numerical solvers (Mattingley and Boyd 2012; Grant and Boyd 2008).

In the broad discussion of quadratic programs, normal forms as the one above are the representation of choice, as they abstract away problem specifics. They are widely used to provide insight into the geometric properties of these problems and design solution techniques. Most solvers are designed to take normal forms as input.

Concrete problems arising in applications, however, are easier to express in terms of an algebraic modeling language of parametrized sums, multiplications and set operations, which we refer to as *symbolic*. For example, to specify that the flow in a directed graph G must be conserved, we use the constraint:

$$\forall v \in \mathit{Vertex}(G): \; \sum\nolimits_{u \in Nb^+(v)} x_{uv} - \sum\nolimits_{w \in Nb^-(v)} x_{vw} = 0.$$

This can be read as follows: for every vertex v, if we add the flow x_{uv} from each incoming neighbor $u \in Nb^+(v)$ of v and subtract the flow x_{vw} to each outgoing neighbor $w \in Nb^-(v)$ of v, we get the number 0. On the one hand, such symbolic forms provide a compact representation of structured problems; of course, any problem can also be expressed as a standard form in matrix-vector algebra. However, describing a problem in terms of, say, the coefficients of its constraint matrix is unintuitive and a tedious exercise. On the other hand, symbolic forms are the starting point for designing specialized algorithms. Through ingenuity, the AI, machine learning and operations researchers can make inferences about the structure of a given constraint and leverage them to design a more efficient algorithm (e.g., optimizers for flow problems (Chardaire and Lisser 2002)).

This notational convenience of symbolic forms has been widely recognized within the literature. For example, (Fourer, Gay, and Kernighan 1993; Wallace and Ziemba 2005) are interested in intuitive modeling languages for optimization, and allow sets of objects to index LP variables. Disciplined programming (Grant and Boyd 2008) provides an object-oriented environment in a high-level programming language to enable a structured interface between the model and the solver. However, these and other works address the notational convenience of symbolic forms only: the language is used to simplify the problem specification, but it is eventually converted to a matrix-vector normal form for solving. This is unlike the situation in statistical relational learning (Getoor and Taskar 2007; De Raedt et al. 2016) where symbolic structure is being used extensively.

In this paper, triggered by relational mathematical programming (Kersting, Mladenov, and Tokmakov 2015; Cussens 2015), we take the view that symbolic forms can and should inform the solver without any expert intervention. The main goal of this paper is to demonstrate that structure in linear and quadratic programs can be efficiently detected and exploited automatically by making use of the symbolic representation of the problem within solvers, the job that has so far been solely at the hands of the expert.

To realize this objective, we will proceed as follows. We will first introduce a high-level algebraic-logical language, which sets the frame for our discussion. Its significance is that the standard forms that result from QPs expressed in the language can be viewed as pseudo-Boolean functions. Thus, instead of storing a standard form as a dense or sparse matrix, as is usually the case in QP solvers, we will store it as an Algebraic Decision Diagram (ADD), making use of the problem structure to produce a highly compressed representation. The reason for doing this lies in the fact that ADDs can carry out certain matrix manipulations efficiently.

Nonetheless, even if a mathematical program encoded in a high-level language has been reduced to an efficiently manipulable data structure such as an ADD, it is far from obvious how a generic solver can be engineered for it. Matrix operations with ADDs are efficient only under certain conditions, such as:

- they have to be done recursively, in a specific descent order;
- they have to involve the entire matrix (batch mode): access to arbitrary submatrices is not efficient.

This places rather specific constraints on the kind of method that could benefit from an ADD representation, ruling out approaches like random coordinate descent. In this paper, we aim to engineer and construct a solver for such matrices directly *in circuit representation*. We employ ideas from the matrix-free interior point method (Gondzio 2012), which appeals to an iterative linear equation solver together with the log-barrier method, to achieve a regime where the constraint matrix is only accessed through matrix-vector multiplications. Specifically, we show a ADD-based realization of the approach leverages the desirable properties of these representations (e.g. caching of submatrices), leading to a robust and fast engine. We demonstrate the flexibility of this symbolic-numeric optimizer on decision making and compressed sensing tasks with millions of non-zero entries.

The remainder of this paper is organized as follows. We start off by a short recap of decision diagrams and their basic properties. Next, we formulate a symbolic language for defining QPs and relate it to ADDs. We then proceed with the discussion of a solver for QPs expressed in that language. Before concluding we provide numerical illustrations of the method along with a discussion of related work.

Algebraic Decision Diagrams

A BDD (Bryant 1986) is a compact and efficiently manipulable data structure for a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$. Its roots are obtained by the Shannon expansion of the *cofactors* of the function: if f_x and $f_{\overline{x}}$ denote the partial evaluation of $f(x,\ldots)$ by setting the variable x to 1 and 0 respectively, then

$$f = x \cdot f_x + \overline{x} \cdot f_{\overline{x}}.$$

When the Shannon expansion is carried out recursively, we obtain a full binary tree whose non-terminal nodes, labeled by variables $\{\ldots, x_i, \ldots\}$, represent a function: its left child is f's cofactor w.r.t. x_i for some i and its right child is f's cofactor w.r.t. $\overline{x_i}$. The terminal node, then, is labeled 0 or 1 and corresponds to a total evaluation of f. By further ordering the variables, a graph that we call the (ordered) BDD of f can be constructed such that at the kth level of the tree, the cofactors wrt the kth variable are taken. Given an ordering, BDD representations are *canonical*: for any two functions $f, g\colon \{0,1\}^n \to \{0,1\}$, $f \equiv g \Leftrightarrow f_x \equiv g_x$ and $f_{\overline{x}} \equiv g_{\overline{x}}$; and *compact* for Binary operators $\circ \in \{+, \times, \ldots\}$: $|f \circ g| \leq |f||g|$.

ADDs generalize BDDs in representing functions of the form $\{0,1\}^n \to \mathbb{R}$, i.e. pseudo-Boolean functions, and so inherit the same structural properties as BDDs except, of course, that terminal nodes are labeled with real numbers (Fujita, McGeer, and Yang 1997; Clarke, Fujita, and Zhao 1996). Consider any real-valued vector of length m: the vector is indexed by $\lg m$ bits, and so a function of the form $\{0,1\}^{\lg m} \to \mathbb{R}$ maps the vector's indices to its range. Thus, an ADD can represent the vector. By extension, any real-valued $2^m \times 2^n$ matrix A with row index bits $\{x_1, \ldots, x_m\}$ and column index bits $\{y_1, \ldots, y_n\}$ can be represented as a function $f(x_1, y_1, x_2, \ldots)$ such that its cofactors are the entries of the matrix. The intuition is to treat x_1 as the most significant bit, and x_m as the least. Then A represented by a function f as an ADD is: $(f_{\overline{x_1 y_1}} \quad f_{\overline{x_1} y_1})$ as the first row and $(f_{x_1 \overline{y_1}} \quad f_{x_1 y_1})$ as the second, where each submatrix is similarly defined wrt the next significant bit. Analogously, when multiplying two m-length vectors represented as Boolean functions $f, g\colon \{0,1\}^{\lg m} \to \mathbb{R}$, we can write $[f_{\overline{x_1}} \quad f_{x_1}][g_{\overline{x_1}} \quad g_{x_1}]^T$, taking, as usual, x_1 as the most significant bit.

As can be gathered by the cofactor formulations, the ADD representation admits two significant properties. First, when identical submatrices occur in various parts of the matrix, e.g. in block and sparse matrices, different cofactors map to identical functions leading to compact ADDs. Second, matrix algorithms involving recursive-descent procedures on submatrices (e.g. multiplication) and term operations (addition, maximization) are efficiently implementable by manipulating the ADD representation. In sum, standard matrix-vector algebra can be implemented over ADDs efficiently, and in particular, the caching of submatrices in recursively defined operations (while implicitly respecting the variable ordering) will best exploit the superiority of the representation.

First-Order Logical Quadratic Programs

Given a finite-domain first-order logical language $\mathcal{L}$, the syntax for first-order (FO) logical quadratic programs is:

$$\begin{aligned} \underset{v}{\text{minimize}} \quad & \sum_{\{x,x' : \theta(x,x')\}} q(x,x')v(x)v(x') + \sum_{\{x:\zeta(x)\}} c(x)v(x) \\ \text{subject to} \quad & \{y : \psi(y)\} : \sum_{\{x:\phi(x,y)\}} a(x,y)v(x) \geq b(y) \end{aligned}$$

where x, y and z are *tuples* of ground atoms, and we write $\delta(z)$ to mean that the $\mathcal{L}$-formula δ mentions the atoms z, and read $\{z : \delta(z)\}$ as the set of all assignments to z satisfying $\delta(z)$.[1]

[1] Since ground atoms always evaluate to true or false, we often refer to $x, y, z, \ldots$ as Boolean variables.

$$\begin{array}{ll} \text{minimize}_{v(x),v(\bar{x})} & v(x) + v(\bar{x}) \\ \text{subject to} & v(x) \geq 1, v(x) + v(\bar{x}) \geq 1, \end{array} \qquad \underbrace{\begin{bmatrix} 0\ (\overline{xy}) & 1\ (x\bar{y}) \\ 1\ (\bar{x}y) & 1\ (xy) \end{bmatrix}}_{A}, \underbrace{\begin{bmatrix} 1\ (\bar{y}) \\ 1\ (y) \end{bmatrix}}_{b}, \underbrace{\begin{bmatrix} 1\ (\bar{x}) \\ 1\ (x) \end{bmatrix}}_{c}.$$

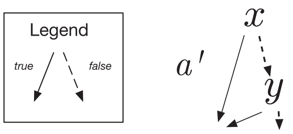


Figure 1: A first-order mathematical program, its normal form, and the ADD representation of its constraint matrix.

The constraints are to be read procedurally as follows: for every assignment to y satisfying $\psi(y)$, consider the constraint $\sum_C a(x, y)v(x) \geq b(y)$, where the set C are those assignments to x satisfying $\phi(x, y)$, and a and b are pseudo-Boolean functions. For example, consider:

$$\text{minimize}_v \sum_{\{x:\text{TRUE}\}} v(x) \text{ s.t. } \{y : \text{TRUE}\} : \sum_{\{x:\ x\vee y\}} v(x) \geq 1$$

This program is equivalent to the ground LP shown in Fig 1(left), where $v(x)$ and $v(\bar{x})$ are the two real-valued decision variables. Moreover, the LP's normal form is given in Fig 1(center). For the sake of simplicity, we use an LP for illustrating the language, but everything carries over to the QP case.

To get a sense of how constraints in this language are reduced to ADDs, let I and J be finite sets, $a : I \times J \to \mathbb{R}$, $b : I \to \mathbb{R}$ be real-valued functions, and $\{x(j) \mid j \in J\}$ be a set of variables. Then, the system of linear inequalities $\forall i \in I : \sum_{j\in J} a(i, j)x(j) \geq b(i)$ is identical to the matrix-vector inequality $Ax \geq b$, where $A \in \mathbb{R}^{|I|\times|J|}$ is a matrix such that $A_{ij} = a(i, j)$, b is the vector $b_i = b(i)$ and $x \in \mathbb{R}^{|J|}$ is an unknown vector representing the variables. In our language, the constraint $\{y : \psi(y)\} : \sum_{\{x:\phi(x,y)\}} a(x, y)v(x) \geq b(y)$ can be rewritten as

$$\{y : \text{TRUE}\} : \sum_{\{x:True\}} a(x, y)1_\psi(y)1_\phi(x, y)v(x) \geq b(y)1_\psi(y),$$

where 1 is an indicator function, e.g., if $1_\phi(x, y)$ is the function that returns 1 whenever $\phi(x, y)$ holds and 0 otherwise. Thus, the pseudo-Boolean function $a'(x, y) := a(x, y)1_\psi(y)1_\phi(x, y)$ identifies with the matrix $A \in \mathbb{R}^{2^{|x|}\times 2^{|y|}}$ in the canonical form representation of this constraint. The other elements of the LP can be represented as pseudo-Boolean functions analogously. Assuming that a, b, c and q have symbolic representations that can be built recursively, the corresponding a', b', c' and q' can be obtained by ADD multiplication. Formally:

Proposition 1: *There is an algorithm for building an ADD for terms from a first-order logical mathematical program without converting the program to the canonical (ground) form. The ADD obtained from the algorithm is identical to the one obtained from the ground form.*

Solution Strategies for FO Logical QPs

Given the representation language, we now turn towards solving logical QPs. To prepare the discussion, let us establish an elementary notion of algorithmic correctness for ADD implementations. Given a first-order logical mathematical program, we assume that we have in hand the ADDs for A, b and c. Then, it can be shown:

Theorem 2: *Suppose A, b and c are as above, and e is any arithmetic expression over them involving standard matrix binary operators. Then there is a sequence of operations over their ADDs, yielding a function h, such that $h = e$.*

The kinds of expressions we have in mind are $e = Ax - b$ (which corresponds to the residual in the corresponding system of linear equations). The proof is as follows:

Proof: For any $f, g : \{0, 1\}^n \to \mathbb{R}$, and (standard) binary matrix operator $\circ$, observe that $h = f \circ g$ iff $h_x = f_x \circ g_x$ and $h_{\bar{x}} = f_{\bar{x}} \circ g_{\bar{x}}$. By canonicity, the ADD for h is precisely the same as the one for f and g composed over $\circ$. In other words, for any arithmetic expression e over $\{f, g, \circ\}$ represented as matrices, the ADD realization h is the same function. ▮

To guide the subsequent construction of the engine, we will first briefly go over the operations previously established as efficient with ADDs (Clarke, Fujita, and Zhao 1996), and some implications thereof for a solver strategy.

Theorem 3: *(Fujita, McGeer, and Yang 1997; Clarke, Fujita, and Zhao 1996) Suppose A, A' are real-valued matrices. Then the following can be efficiently implemented in ADDs using recursive-descent procedures:*

- *accessing and setting a submatrix A^* of A;*
- *termwise operations, i.e. $(A \circ A')_{ij} = A_{ij} \circ A'_{ij}$ for any termwise operator $\circ$;*
- *vector and matrix multiplications.*

The proof of these claims and the ones in the corollary below always proceed by leveraging the Shannon expansion for the corresponding Boolean functions. We refer interested readers to (Fujita, McGeer, and Yang 1997; Clarke, Fujita, and Zhao 1996) for the complexity-theoretic properties of these operations. It is worth noting, for example, that multiplication procedures that perform both (naive) block computations and ones based on Strassen products can be recursively defined. For our purposes, we get:

Corollary 4: *Suppose $d = [d_1 \ \cdots \ d_m]$ and e are m-length real-valued vectors, and k is a scalar quantity. Then the following can be efficiently implemented in ADDs using recursive-descent procedures:*

- *scalar multiplication, e.g. $k \cdot d$;*
- *vector arithmetic, e.g. $d + e$;*
- *element sum, e.g. $\sum_i d_i$;*
- *element function application, e.g. if $w\colon \mathbb{R} \to \mathbb{R}$, then computing $w(d) = [w(d_1) \ \cdots \ w(d_m)]$;*
- *norms, e.g. $\|d\|$ and $\|d\|^2$.*

A Naive Ground-and-Solve Method

The most straightforward approach for solving a first-order logical QP (FOLQP) is to reduce the problem to the normal form and use a standard solver for QPs, such as an interior point method, an augmented Lagrangian method, or some form of active set method, e.g. generalized simplex. The correctness of this solution strategy is guaranteed by the semantics of the logical constraints: in general, every FOLQP can be brought to the normal form. While this method would work, it exhibits a significant drawback in that the optimization engine cannot leverage knowledge about the symbolic structure of the problem. That is, even if the problem compiles to a very small ADD, the running time of the optimizer will depend (linearly at best) on the number of nonzeros in the ground matrix. Clearly, large dense problems will be completely intractable. However, as we will see later, some dense problems can still be attacked with the help of structure.

The Symbolic Interior Point Method

While the ground-and-solve method is indeed correct, one can do significantly better, as we will show now. Specifically, we will construct a solver that automatically exploits the symbolic structure of the FOLQP, in essence, by appealing to the strengths of the ADD representation. The reader should note that much of this discussion is predicated on problems having considerable logical structure, as is often the case in real-word problems involving relations and properties.

Recall from the previous discussion that in the presence of cache, (compact) ADDs translate to very fast matrix-vector multiplications. Moreover, from Corollary 4, vector operations are efficiently implementable, which implies that given an ADD for A, x and b, the computation of the residual $y = Ax - b$ is more efficient than its matrix counterpart (Fujita, McGeer, and Yang 1997). Analogously, a descent along a direction ($x_k = x_{k-1} + \alpha\Delta x$) given ADDs for x_{k-1} and Δx has the same run-time complexity as when performed on dense vectors. However, we have to remark that for ADD-based procedures to be efficient, we need to respect the variable ordering implicit in the ADD. Thus, the solver strategy rests on the following constraints:

(*) *the engine must manipulate the matrix only through recursive-descent arithmetical operations, such as matrix-vector multiplications (matvecs, for short);*

(**) *operations must manipulate either the entire matrix or those submatrices corresponding to cofactors (i.e. arbitrary submatrices are non-trivial to access).*

We will now devise a method that satisfies these requirements. We proceed in two steps. In step 1, we will demonstrate that solving a QP can be reduced to solving a sequence of linear equations over A. Next, in step 2, we will make use of an iterative solver that computes numerical solutions by a sequence of residuals and vector algebraic operations. As a result, we will obtain a method that fully utilizes the strengths of ADDs. Due to space constraints, we will not be able to discuss the construction in full detail, and so we sketch the main ideas that convey how ADDs are exploited.

```
Algorithm 1: Primal-Dual Barrier Method
Input: $(x^0, y^0, s^0)$ with $(x^0, s^0) \geq 0$
$k \leftarrow 0$;
while stopping criterion not fulfilled do
    Solve (2) with $(x, y, s) = (x^k, y^k, s^k)$ to obtain a
      direction $(\Delta x^k, \Delta y^k, \Delta s^k)$;
    Choose step length $\alpha_k \in (0, 1]$;
    Update $(x^{k+1}, y^{k+1}, s^{k+1}) \leftarrow \alpha_k(\Delta x^k, \Delta y^k, \Delta s^k)$
      $k \leftarrow k + 1$
return $\mathbf{x}^k$;
```

```
Algorithm 2: Conjugate Gradient Method
t Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^n$
$k \leftarrow 0,\ r_0 \leftarrow b - Ax_0$;
while stopping criterion not fulfilled do
    $k \leftarrow k + 1$;
    if $k = 1$ then
        $p_0 \leftarrow r_0$;
    else
        $\tau_{k-1} = (r_{k-1}^T r_{k-1})/(r_{k-2}^T r_{k-2}^T)$;
        $p_k = r_{k-1} + \tau_{k-1} p_{k-1}$;
    $\mu_k = (r_{k-1}^T r_{k-1})/(p_k^T A p_k)$;
    $x^k = x_{k-1} + \mu_k p_k$ ;
    $r^k = r_{k-1} - \mu_k A p_k$ ;
return $\mathbf{x}^k$;
```

Step 1: From quadratic programs to linear equations. A prominent solver for QPs in standard form is the primal-dual barrier method, see e.g. (Potra and Wright 2000), sketched in Alg. 1. This method solves a perturbed version of the first-order necessary conditions (KKT conditions) for QP:

$$Ax = b, \quad -Qx + A^T y + s = c, \quad XSe = \mu e, \quad (x, s) \geq 0,$$

where $X = \mathrm{diag}(x_1, \ldots, x_n)$, $S = \mathrm{diag}(s_1, \ldots, s_n)$, and $\mu \geq 0$. The underlying idea is as follows (we refer to (Potra and Wright 2000; Gondzio 2012) for more details): by applying a perturbed Newton method to the equalities in the above system, the algorithm progresses the current solution along a direction obtained by solving the following linear system:

$$\begin{bmatrix} A & 0 & 0 \\ -Q & A^T & I \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} b - Ax \\ c - A^T y - s \\ \mu e - Xs \end{bmatrix}, \tag{1}$$

where e is a vector of ones. Observe that besides A and Q, which we already have decision diagrams for, the only new information that needs to be computed is in the form of residuals in the right-hand side of the equation. By performing two pivots, Δx and Δs can be eliminated from the system, reducing it to the so-called *normal equation*:

$$A(Q + \Theta^{-1})^{-1} A^T \Delta y = f, \tag{2}$$

where Θ is the diagonal matrix $\Theta_{ii} = \frac{x_i}{s_i}$. Once Δy is determined, Δx and Δs are recovered from Δy as $\Delta x =$

$(Q + \Theta)^{-1}(A^T \Delta y - g)$ and $\Delta s = h - A^T \Delta y$, where h and g are obtained from the residuals via vector arithmetic. The reader will note that constructing the left-hand side involves the matrix inverse $(Q + \Theta^{-1})^{-1}$. The efficiency of computing this inverse cannot be guaranteed with ADDs, unfortunately. Consequently, we assume that either the problem is separable (Q is diagonal), in which case computing this inverse reduces to computing the reciprocals of the diagonal elements (an efficient operation with ADDs), or that we only have box constraints, meaning that A is a diagonal matrix, in which case solving the equation reduces to solving $(Q + \Theta)\Delta y' = A^{-1} f$ and re-scaling. The general case of going beyond these assumptions is omitted here for space reasons (Gondzio 2012).

The benefit of reformulating (1) into (2) can be appreciated from the observation that (2) becomes positive-semidefinite, which is crucial for solving this system by residuals. To reiterate: the primal-dual barrier method solves a quadratic program iteratively by solving one linear system (2) in each iteration. Constructing the right-hand side of this linear system only requires the calculation of residuals and vector arithmetic, which can be done efficiently with ADDs. Moreover, this system does not require taking arbitrary submatrices of A or Q. Hence, the primal-dual barrier method meets our requirements. Now, let us investigate how (2) can be solved via a sequence of residuals.

Step 2: From linear equations to residuals. To solve (2), we employ the conjugate gradient method (Golub and Van Loan 1996), sketched in Alg. 2. Here, the algorithm uses three algebraic operations: (1) matrix-vector products; (2) norm computation ($r^T r$) and (3) scalar updates. From Corollary 4, all of these operations can be implemented efficiently in ADDs. There is, however, one challenge that remains to be addressed. As the barrier method approaches the solution of the QP, the iterates s^k and x^k approach complementary slackness ($x_i s_i = 0$). This means that the diagonal entries of the matrix Θ in (2) tend to either 0 or $+\infty$, making the condition number of (2) unbounded. This is a severe problem for any iterative solver, as the number of iterations required to reach a specified tolerance becomes unbounded. To remedy this situation, Gondzio (2012) proposes the following approach: first, the system can be regularized to achieve a condition number bounded by the largest singular value of A. Second, due to the IPM's remarkable robustness to inexact search directions, it is not necessary to solve the system completely. In practice, decreasing the residual by a factor of 0.01 to 0.0001 has been found sufficient. Finally, a partial pivoted Cholesky factorization can be used to speed-up the convergence. That is, perform a small number k (say 50) Cholesky pivots, and use the resulting trapezoidal matrix as a preconditioner. More details on this can be found in (Gondzio 2012). Gondzio also demonstrated that this approach does lead to a practical algorithm. Unfortunately, in our setting, constructing this preconditioner requires that we query k rows of N and perform pivots with them. This forces us to partially back down on our requirement (**), since we need random access to k rows. However, by keeping k small, we can guarantee the ADD in this unfavorable regime will be kept to a minimum.

Thus, we have a method for solving QPs, implemented solely with ADD operations, and much of this work takes full advantage of the ADD representation (thereby, inheriting its superiority). Intuitively, one can expect significant speed-ups over matrix-based methods when the ADDs are compact, e.g. , arising from structured (in a logical sense) problems.

Empirical Illustration

Our aim here is to investigate the empirical performance of our ADD-based interior point solver. There are three main questions we wish to investigate, namely: **(Q1)** in the presence of symbolic structure, does our ADD-based solver perform better than its matrix-based counterpart? **(Q2)** On structured sparse problems, does solving with ADDs have advantages over solving with sparse matrices? And, **(Q3)**, can the ADD-based method handle dense problems as easily as sparse problems?

To evaluate the performance of the approach, we implemented the entire pipeline described here, that is, a symbolic environment to specify QPs, a compiler to ADDs, based on the popular CUDD package, and the symbolic interior-point method described in the previous section.

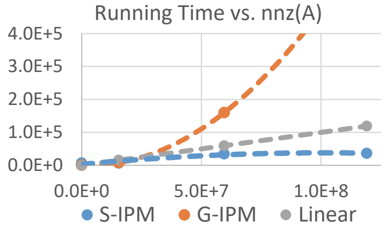
To address **(Q1)** and **(Q2)**, we applied the symbolic IPM on the problem of computing the value function of a family of Markov decision processes used in (Hoey et al. 1999). These MDPs concern a factory agent whose task is to paint two objects and connect them. A number of operations (actions) need to be performed on these objects before painting, each of which requires the use of special tools, which may or may not be available currently. Painting and connecting can be done in different ways, yielding results of various quality, and each requiring different tools. The final product is rewarded according whether the required level of quality is achieved. Since these MDPs admit compact symbolic representations, we consider them good candidates to illustrate the potential advantages of symbolic optimization. The computation of an MDP value function corresponds to the following first-order logical LP:

$$\text{min.} \sum\nolimits_{s:\texttt{state}(s)} v(s) \text{ , s.t. } \{s : \texttt{state}(s), a : \texttt{act}(a)\} :$$
$$v(s) \geq \texttt{rew}(s) + \gamma \sum\nolimits_{s':\texttt{state}(s')} \texttt{tprob}(s, a, s')v(s')$$

where s, s', a are vectors of Boolean variables, `state` is a Boolean formula whose models are the possible states of the MDP, `act` is a formula that models the possible actions, and `rew` and `tprob` are pseudo-Boolean functions that model the reward and the transition probability from s to s' under the action a. We compared our approach to a matrix implementation of the primal-dual barrier method, both algorithms terminate at the same relative residual, 10^{-5}. The results are summarized in Fig. 2(top).

The symbolic IPM outperforms the matrix-based IPM on the larger instances. The most striking observation is that the running time depends mostly on the size of the ADD, which practically translates to scaling sublinearly in the number of nonzeroes (nnz(A)). This is illustrated on the bottom-left of Fig. 2, where these running times are plotted versus nnz(A) and compared with a hypothetical linear running time with a slope of 0.001. To the best of our knowledge, no generic method, sparse or dense, can achieve such scaling behavior. This answers **(Q1)** and **(Q2)** affirmatively.

Problem Statistics				Symbolic IPM		Ground IPM
name	#vars	#constr	$nnz(A)$	|ADD|	time[s]	time[s]
factory	131.072	688.128	4.000.000	1819	6899	**516**
factory0	524.288	2.752.510	15.510.000	1895	**6544**	7920
factory1	2.097.150	11.000.000	59.549.700	2406	**34749**	159730
factory2	4.194.300	22.020.100	119.099.000	2504	**36248**	≥ 48hrs.



Problem Statistics		Symbolic IPM	FWHT IPM
n	m	time[s]	time[s]
2^{12}	2^{10}	**8.3**	18
2^{13}	2^{11}	**20.2**	28.2
2^{14}	2^{12}	46.5	**43.9**
2^{15}	2^{13}	99.2	**65.7**

Figure 2: Evaluations on MDP (top), with a graphical comparison (bottom-left) and compressed sensing (bottom-right).

Our second illustration concerns the problem of compressed sensing (CS). That is, we are interested in recovering the sparsest solution to the problem $||Ax - b||^2$, where $A \in \mathbb{R}^{m \times n}$ and $n >> m$ where n is the signal length and m the number of measurments. While this is a hard combinatorial problem, if the matrix A admits the so-called restricted isometry property, the following convex problem (Basis Pursuit Denoising, or BPDN): $\min_{x \in \mathbb{R}^n} \tau ||x||_1 + ||Ax - b||_2^2$ recovers the exact solution. The matrices typically used in CS tend to be dense, yet highly structured, often admitting an $O(n \log n)$ FFT-style recur-and-cache matrix-vector multiplication scheme, making BPDN solvers efficient. A remarkable insight is that these fast recursive transforms are very similar to the ADD matrix-vector product. In fact, if we take A to be the Walsh matrix $W_{\log(n)}$ which admits an ADD of size $O(\log n)$, the ADD matrix-vector product resembles the Walsh-Hadamard transform (WHT). Moreover, this compact ADD is extracted automatically from the symbolic specification of the Walsh matrix. We illustrate this in the following experiment. We apply the Symbolic IPM to the BPDN reformulation of (Fountoulakis, Gondzio, and Zhlobich 2014), with the Walsh matrix specified symbolically, recovering random sparse vectors. We compare to the reference MATLAB code provided by Fountoulakis *et al.* using MATLAB's implementation of WHT. The task is to recover a sparse random vector with $k = 50$ normally distributed nonzero entries. Results are reported in Fig. 2(bottom).

While the method does not scale as well as the hand-tailored solution, we demonstrate that the symbolic approach can handle dense matrices reasonably well, supporting an affirmative answer of **(Q3)**.

Related Work

There has been a long-standing interest in modeling languages for optimization, e.g. (Fourer, Gay, and Kernighan 1993; Wallace and Ziemba 2005). While they provide a syntax for sets of objects to index LP variables, they do not provide a high-level logical language for the constraints. Disciplined programming (Grant and Boyd 2008) enables an object-oriented approach to constructing optimization problems, and provides a structured interface between the model and the solver by means of which geometric properties such as the curvature of the objective can be inferred. In contrast to our goals, the language is also not logical, and the solver not generic in the way we suggest. (That is, in our setting, high-level logical constraints are all that are expected from the user.) In more recent work, (Diamond and Boyd 2015) also advocate matrix-free methods for optimization problems expressed in human-readable forms. However, they assume that the user has already analyzed her model and identified that it could be broken down into efficient operations. They provide the environment to implement these operations in a way in which the solver can exploit them. In that sense, their approach also differs from ours, where the user provides her logical constraints, and the solver takes it from there.

First-order logical representations for mathematical programs are also considered in (Gordon, Hong, and Dudík 2009; Kersting, Mladenov, and Tokmakov 2015; Cussens 2015). However, the rely on matrix-vector normal forms for solving them, whether lifted or not, and do not employ ADDs.

The efficiency of ADDs for matrix-vector algebra was established in (Clarke, Fujita, and Zhao 1996). In particular, the use of ADDs for compactly specifying (and solving) Markov decision processes (i.e. representing transitions and rewards as Boolean functions) was popularized in (Hoey et al. 1999); see (Zamani et al. 2013; Cui and Khardon 2016) for recent offerings. We differ fundamentally from these strands of work in that we are advocating the realization of solution methods for generic optimization problems using ADDs, which (surprisingly) has never been studied in great detail to the best of our knowledge. Therefore, we call our line of research as *symbolic numerical optimization.*

Conclusions

A long-standing goal in machine learning and AI, which is also reflected in the philosophy on the *democratization of data,* is to make the specification and solving of real-world problems simple and natural, possibly even for non-experts.

To this aim, we considered first-order logical mathematical programs that support individuals, relations and connectives, and developed a new line of research of symbolically solving these programs in a generic way. In our case, a matrix-free interior point method was argued for. Our empirical results demonstrate the flexibility of this research direction. The most interesting avenue for future work is to explore richer modeling languages paired with more powerful circuit representations.

Acknowledgements The authors would like to thank the anonymous reviewers for their suggestions, and are grateful to Scott Sanner for his valuable feedback. Vaishak was supported in part by the Research Foundation-Flanders (FWO-Vlaanderen).

References

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on* 100(8):677–691.

Chardaire, P., and Lisser, A. 2002. Simplex and interior point specialized algorithms for solving nonoriented multicommodity flow problems. *Operations Research* 50(2):260–276.

Clarke, E. M.; Fujita, M.; and Zhao, X. 1996. Multi-terminal binary decision diagrams and hybrid decision diagrams. In Sasao, T., and Fujita, M., eds., *Representations of Discrete Functions*. Boston, MA: Springer US. 93–108.

Cui, H., and Khardon, R. 2016. Online symbolic gradient-based optimization for factored action mdps. In *IJCAI*.

Cussens, J. 2015. First-order integer programming for MAP problems. *CoRR* abs/1507.02912.

De Raedt, L.; Kersting, K.; Natarajan, S.; and Poole, D. 2016. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Morgan & Claypool Publishers.

Diamond, S., and Boyd, S. 2015. Matrix-free convex optimization modeling. *arXiv preprint arXiv:1506.00760*.

Fountoulakis, K.; Gondzio, J.; and Zhlobich, P. 2014. Matrix-free interior point method for compressed sensing problems. *Mathematical Programming Computation* 6(1):1–31.

Fourer, R.; Gay, D. M.; and Kernighan, B. W. 1993. *AMPL: A Mathematical Programming Language*. The Scientific Press, San Francisco, CA.

Fujita, M.; McGeer, P.; and Yang, J.-Y. 1997. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design* 10(2):149–169.

Getoor, L., and Taskar, B., eds. 2007. *Introduction to Statistical Relational Learning*. MIT Press.

Golub, G. H., and Van Loan, C. F. 1996. *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press.

Gondzio, J. 2012. Matrix-free interior point method. *Comp. Opt. and Appl.* 51(2):457–480.

Gordon, G.; Hong, S.; and Dudík, M. 2009. First-order mixed integer linear programming. In *UAI*, 213–222.

Grant, M., and Boyd, S. 2008. Graph implementations for nonsmooth convex programs. In *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences. Springer. 95–110.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. Spudd: Stochastic planning using decision diagrams. In *UAI*, 279–288.

Kersting, K.; Mladenov, M.; and Tokmakov, P. 2015. Relational linear programming. *Artificial Intelligence Journal (AIJ)* OnlineFirst.

Mattingley, J., and Boyd, S. 2012. CVXGEN: A code generator for embedded convex optimization. *Optimization and Engineering* 12(1):1–27.

Potra, F., and Wright, S. J. 2000. Interior-point methods. *Journal of Computational and Applied Mathematics* 124:281–302.

Wallace, S., and Ziemba, W., eds. 2005. *Applications of Stochastic Programming*. SIAM, Philadelphia.

Zamani, Z.; Sanner, S.; Delgado, K. V.; and de Barros, L. N. 2013. Robust optimization for hybrid mdps with state-dependent noise. In *IJCAI*.