Solving unconstrained binary polynomial programs with limited reach

Jens Vinther Clausen[a], Yves Crama[b], Richard Lusby[a], Elisabeth Rodríguez-Heck[c], Stefan Ropke[a,*]

[a] *Department of Technology Management and Economics, Technical University of Denmark, Anker Engelunds Vej 1 Bygning 101A, 2800 Kgs. Lyngby, Denmark*
[b] *HEC Liège - Management School, University of Liège, Rue Louvrex 14 (N1), 4000 Liège, Belgium*
[c] *RWTH Aachen University, The Chair of Operations Research, Kackertstraße 7, 52072 Aachen, Germany*

Abstract

Unconstrained Binary Polynomial Programs (UBPs) are a class of optimization problems relevant in a broad array of fields. In this paper, we examine an example from communication engineering, namely low autocorrelation binary sequences and propose a new dynamic programming approach that is particularly effective on UBP instances that have a limited so-called reach, which is the maximum difference between any two variable indices in the problem. Based on the reach, the dynamic programming approach decomposes the problem into a number of overlapping stages that can be solved in parallel. On a set of publicly available low autocorrelation binary sequence problems, we demonstrate the superiority of the approach by showing that the method solves to optimality for the first time several previously unsolved instances. In particular, we provide a direct comparison between the proposed method and a modern version of a previously proposed dynamic program for UBPs. We give a detailed analysis of the connection between the two different algorithms and demonstrate that the advantage of the proposed dynamic program is in its ability to implicitly identify the multilinear polynomials that are required in the recursive steps of the two dynamic programs. For perspective, a comparison to several other methods is also provided.

Keywords: unconstrained binary polynomial program, dynamic programming, low autocorrelated binary sequences

1. Introduction

Consider a set of variables $x_j \in \{0,1\}$ for $j \in J = \{1,\dots,n\}$ and a set of *monomials* $M_i \subseteq J$ for $i \in I$. The unconstrained binary polynomial program (UBP) is defined as follows:

$$\min\{\sum_{i\in I} c_i \prod_{j\in M_i} x_j, x \in \{0,1\}^n\} \tag{1}$$

where c_i for $i \in I$ are the coefficients of the monomials. Problem (1) is NP-hard in general, as it encompasses in particular unconstrained quadratic binary programming. The objective function of (1) is multilinear, that is, it does not contain squares or higher powers of single variables because for binary variables $x_j = x_j^2$ is satisfied. Note that the term monomial usually refers to a product of variables $\prod_{j\in M_i} x_j$. For the sake of simplicity, we abuse the terminology slightly and refer to the subsets M_i of *variable indices* as monomials.

*Corresponding author

Email addresses: jensvintherclausen@gmail.com (Jens Vinther Clausen), yves.crama@uliege.be (Yves Crama), rmlu@dtu.dk (Richard Lusby), erodriguezheck@gmail.com (Elisabeth Rodríguez-Heck), ropke@dtu.dk (Stefan Ropke)

Problem (1) is very general, and finds its relevance for example in classical operations research applications such as uncapacitated facility location (Hammer, 1968; Dearing et al., 1992; Goldengorin et al., 2003) or the p-median problem (Goldengorin and Krushinsky, 2011), but also in applications coming from different fields such as computer vision (Fix et al., 2015; Freedman and Drineas, 2005; Ishikawa, 2011; Kolmogorov and Zabih, 2004) or statistical mechanics (Bernasconi, 1987; Liers et al., 2010). When the objective function of (1) is quadratic, the problem is equivalent to a 0–1 linear programming problem whose set of feasible solutions is defined by the Boolean Quadric Polytope, which is isomorphic to the Cut Polytope (Padberg, 1989; De Simone, 1990). The quadratic case has attracted much interest on its own and can be used to model classical combinatorial optimization problems such as maximum cut, maximum stable set or minimum vertex cover, among others. However, in this paper we limit ourselves to the higher-degree case. Moreover, every pseudo-Boolean function, that is, every mapping $f : \{0,1\}^n \rightarrow \mathbb{R}$ admits a unique multilinear representation (Hammer et al., 1963b; Hammer and Rudeanu, 1968). This is easy to see since a pseudo-Boolean function with n variables will have exactly 2^n possible inputs and it can thereby be represented by a table with an entry for each possible input. Once the table representation is established it is easy to construct the corresponding multilinear polynomial. Consider for example the two variable pseudo-polynomial function given by the table

(x_1, x_2)	(0,0)	(0,1)	(1,0)	(1,1)
$f(x_1, x_2)$	7	9	13	42

We can then write the corresponding multilinear polynomial as

$$\begin{aligned} f(x_1, x_2) &= 7(1 - x_1)(1 - x_2) + 9(1 - x_1)x_2 + 13x_1(1 - x_2) + 42x_1x_2 \\ &= 7(1 - x_1 - x_2 + x_1x_2) + 9(x_2 - x_1x_2) + 13(x_1 - x_1x_2) + 42x_1x_2 \\ &= 7 - 7x_1 - 7x_2 + 7x_1x_2 + 9x_2 - 9x_1x_2 + 13x_1 - 13x_1x_2 + 42x_1x_2 \\ &= 7 + 6x_1 + 2x_2 + 27x_1x_2. \end{aligned}$$

However, finding a multilinear expression of a given mapping f using a tabulation approach may be extremely time consuming since the table size grows exponentially with n. Using the pseudo-Boolean formalism, (1) finds relevance in even more applications, such as maximum satisfiability (for a detailed list of applications see for example Boros and Hammer (2002); Crama and Hammer (2011). The relation between pseudo-Boolean functions and UBPs also implies that every UBP can be in principle expressed as an unconstrained non-linear binary optimization problem.

In this paper, we develop a new dynamic programming algorithm that is designed to address instances of (1) that have limited *reach*. The reach of (1) is a metric that states the maximum difference between any two variable indices across all monomials in the problem. Formally, we define the reach w of (1) to be $w = \max_{i \in I}\{w(M_i)\}$, where $w(M_i) = \max\{M_i\} - \min\{M_i\} + 1$. As an example, the monomial $M_i = \{x_1, x_2, x_4, x_9\}$ has a reach of 9. Obviously, the reach of a monomial M_i, and of an instance of (1), depends on the indexation order of the variables; however, we do not consider the problem of reordering the variable indices in this paper. (In view of the relation between reach and bandwidth outlined hereunder, determining a reordering of the variables that minimizes the reach is NP-Complete; see e.g., Papadimitriou (1976).) The proposed dynamic programming algorithm exploits the limited reach of an instance to decompose the problem into a sequence of stages that can be solved, potentially in parallel.

One particular class of UBPs where the notion of reach is inherently relevant in the structure of the instances is the Low (off-peak) Autocorrelation Binary Sequences (LABS) problem. Applications of this problem can be found in communication engineering (Bernasconi, 1987; Mertens, 1996)

or statistical mechanics (Liers et al., 2010) among others. On a set of publicly available LABS instances (MINLPLib, 2020; POLIP, 2020), our dynamic programming algorithm outperforms existing approaches, solving several previously unsolved instances. The algorithm does bear some similarity to a previously proposed dynamic program of Hammer et al. (1963a,b) known as the *Basic Algorithm*. We therefore discuss the connection between these two dynamic programming approaches in detail and explain the improved computational performance of the new approach.

Similar to Crama et al. (1990) we can associate an undirected graph $G = (V, E)$ with (1). Here, $V = \{1, \ldots, n\}$ and the set E contains an edge (j, k) if there exists $i \in I$ such that $\{j, k\} \subseteq M_i$ (in other words, $(j, k) \in E$ if x_k and x_j occur together in at least one of the monomials of the UBP). This graph is called the *co-occurrence graph* of the UBP. The reach of a UBP is related to well-known graph theoretical metrics such as *treewidth*, *pathwidth* and *bandwidth* of the co-occurrence graph (see Bodlaender (1998) for definitions). In particular, we have that

$$treewidth(G) \leq pathwidth(G) \leq bandwith(G) = reach(G) - 1,$$

which implies that a UBP with limited reach has an associated graph with limited treewidth. The first and second inequality follow from Lemma 3 and Theorem 44 of Bodlaender (1998), respectively, while the third inequality follows from the definitions of reach and bandwidth. The relation between reach and treewidth implies that UBP instances with limited reach can be solved in polynomial time by the Basic Algorithm, following results in Crama et al. (1990). The proposed dynamic programming algorithm also solves instances with limited reach in polynomial time.

Note also that for the specific case of the LABS problem, Liers et al. (2010) make the following claims (in their paper, R denotes the reach): "For finite R the model can be solved in a time of order $O(N2^{2(R-3)})$ by transfer matrix methods". (See Conway (2017) for the relation between dynamic programming and transfer matrix methods.) The authors do not provide further details and have apparently not attempted an implementation.

The contributions of this paper are hence threefold: 1) We develop a new dynamic programming algorithm that is effective on UBPs with limited reach and demonstrate through computational experiments the superior performance of this approach on a publicly available set of LABS instances. The proposed approach is able to optimally solve several instances for which no optimal solution has been previously reported. The asymptotic running time of the dynamic programming algorithm is linear in the number of variables when solving instances with a fixed reach. 2) We provide a detailed description of the connection between the proposed dynamic programming algorithm and the Basic Algorithm as well as a modern implementation of the Basic Algorithm. In doing so, we highlight the similarities and differences between the two respective algorithms and identify what the improved computational performance can be attributed to. 3) We provide a comparison of several different solution methods on a publicly available set of LABS instances.

The rest of this paper is structured as follows. We provide a review of relevant literature in Section 2, and we present the proposed dynamic programming algorithm in Section 3. In Section 4, we review the Basic Algorithm of Hammer et al. (1963a) and analyze its connection to the proposed dynamic program. Numerical results are discussed in Section 5 and conclusions are drawn in Section 6.

2. Literature Review

In this paper, we directly compare the performance of the proposed dynamic programming algorithm with the performance of a modern implementation of the so-called Basic Algorithm. The Basic Algorithm was first introduced in (Hammer et al., 1963a,b). A detailed description

of the algorithm can be found in (Crama et al., 1990; Boros and Hammer, 2002). The general idea of this algorithm is to eliminate the variables of the multilinear objective function of (1) one by one, leveraging dynamic programming principles and local optimality conditions. Each iteration of the Basic Algorithm defines a multilinear polynomial with one less variable than the polynomial of the previous iteration, such that the optimal values of both polynomials are equal. In this way, one can optimize the polynomial corresponding to any of the iterations without losing optimality. The optimization problem of the last iteration can be trivially solved (as it contains a single variable) and the optimal values of all variables can then be reconstructed using backtracking. The key difficulty of the Basic Algorithm as originally introduced in Hammer et al. (1963a,b) lies in computing the multilinear polynomial of each intermediate iteration in an efficient way. Crama et al. (1990) observed that this step can be carried out in constant time on graphs of bounded tree-width. For the general case, they proposed an efficient branch-and-bound algorithm to perform the computation. To carry out the numerical experiments of the present paper, we implemented the algorithm presented in (Crama et al., 1990), as the original implementation was no longer available. We refer the reader to Section 4 for a detailed explanation of the Basic Algorithm and its connection with our dynamic programming algorithm.

An algorithm based on a dynamic programming-type of recursion for UBP was recently presented in Del Pia and Di Gregorio (2022). This algorithm guarantees to optimally solve a UBP only if the hypergraph associated with its objective function satisfies a certain property, which we discuss in the following. A hypergraph is a generalization of the notion of a graph, where the edges (called *hyperedges*) can contain more than two vertices (see for example Berge (1976)). In other words, a hypergraph $H = (V, E)$ can be viewed as a set of vertices V together with a set E of subsets of V. The polynomial objective function of (1) can be naturally associated with a hypergraph, where the vertices correspond to the variables of the polynomial ($V = J$) and the hyperedges correspond to the monomials ($E = \{M_i : i \in I\}$). The notion of cycles in graphs can be generalized in different ways when considering hypergraphs, the most common definitions being Berge-cycles, α-cycles, β-cycles and γ-cycles (see Anstee (1983); Beeri et al. (1983); Fagin (1983)). The property required by the algorithm in Del Pia and Di Gregorio (2022) to optimally solve (1) is that the hypergraph associated with the multilinear objective function must be β-acyclic. A β-cycle of length $\ell \geq 3$ is a sequence of vertices and hyperedges $(v_1, e_1, v_2, e_2, \ldots, v_\ell, e_\ell, v_{\ell+1} = v_1)$ such that all vertices v_i, $i \in \{1, \ldots, \ell\}$ are distinct, all hyperedges e_i, $i \in \{1, \ldots, \ell\}$ are distinct, and $v_i \in e_{i-1}, e_i$ and is not contained in any other edge from the sequence. The LABS instances considered in this paper do not satisfy β-acyclicity, as these instances can contain many β-cycles. The algorithm presented in Del Pia and Di Gregorio (2022) can also be applied to instances containing β-cycles, but in this case it does not solve the problem to optimality; it only reduces the size of the problem by fixing the values of a subset of variables and provides a rule to extend a given optimal solution of the reduced instance to an optimal solution of the general instance. However, obtaining an optimal solution of the reduced instance can be a complex task in general, as the reduced problem can still be very difficult to solve. Moreover, it is shown in Del Pia and Di Gregorio (2022) that, as the ratio of monomials per variables increases in the set of instances considered, the number of variables fixed by the algorithm decreases. This is consistent with the observation made in the current and previous papers (Buchheim and Rinaldi (2007); Crama and Rodríguez-Heck (2017); Elloumi et al. (2021)), that the most difficult UBP instances are the *denser* instances, i.e., those with higher ratios of monomials per variables. Moreover, the dynamic programming algorithm presented in this paper is especially good at solving dense instances, when compared to other approaches in the literature.

Many other algorithms have been proposed in the literature to solve UBPs. Linearization-based approaches were first introduced in (Fortet, 1959; Watters, 1967; Zangwill, 1965; Glover and Woolsey, 1973, 1974) and have attracted much interest in the last few years with recent publications

presenting polyhedral results (Buchheim et al., 2019; Del Pia and Khajavirad, 2016, 2018; Fischer et al., 2018; Buchheim and Rinaldi, 2007), families of valid inequalities (Crama and Rodríguez-Heck, 2017; Del Pia and Khajavirad, 2018; Del Pia et al., 2020; Khajavirad, 2023) and decomposability results (Del Pia and Khajavirad, 2018). Approaches based on quadratic reformulations were first introduced in (Rosenberg, 1975), and have gained recent attention (Anthony et al., 2016, 2017; Verma and Lewis, 2020; Boros et al., 2020; Elloumi et al., 2021) due to their theoretical interest, but also because of their applicability in fields like computer vision (Ishikawa, 2011; Freedman and Drineas, 2005) or quantum computing (Glover et al., 2019). Dantzig-Wolfe decomposition-based approaches have been recently considered for linearizations of (1) by Clausen (2021). The author considers applications in image restoration and instances of the LABS problem. The structure of the LABS instances, in particular, is such that the application of Dantzig-Wolfe decomposition leads to a reformulation of the linearized (1) with particularly appealing computational properties. Clausen (2021) also proposes an enumeration approach to solve the resulting subproblems and shows that this can be advantageous when the size of the subproblems is small. The structure of the LABS instances and the enumeration strategy inspired the development of the proposed dynamic programming algorithm. Finally, general purpose algorithms to solve mixed-integer nonlinear programming (MINLP) problems such as the well-known α-branch-and-bound algorithm (Adjiman et al., 1998) can also be applied in this case.

3. Dynamic programming

In this section we present a dynamic programming algorithm (see Cormen et al. (2009) for an introduction to dynamic programming) for solving UBP instances with limited reach w. At each stage the algorithm works on a subset, of cardinality w, of all variables. Stage 1 *includes* variables $x_1, \ldots, x_w$, while stage s includes variables $x_s, \ldots x_{w+s-1}$. The final stage is $\bar{s} = n - w + 1$ and includes the variables $x_{n-w+1}, \ldots x_n$. A partial solution for stage s is an assignment of 0/1 values to the variables included in stage s. Each monomial $M_i, i \in I$, is assigned to one of the stages that includes all the variables of the monomial. Note that there could be more than one such stage. We assign a monomial to a stage based on the lowest variable index in the monomial. If $\min\{M_i\} \leq \bar{s}$ then we assign M_i to stage $\min\{M_i\}$, otherwise M_i is assigned to stage $\bar{s}$. For example, if $\bar{s} = 6$ then the monomial $\{3, 4, 6\}$ would be assigned to stage 3 while the monomial $\{8, 9\}$ would be assigned to stage $\bar{s} = 6$. We let I_s be the set of monomials that are assigned to stage s.

For each $s = 1, \ldots, \bar{s}$, let us define the (`simplecost`) function sc^s obtained by retaining only the monomials of I_s in the objective function of Problem (1), that is,

$$sc^s(x_s, \ldots, x_{s+w-1}) \triangleq \sum_{M_i \in I_s} c_i \prod_{j \in M_i} x_j$$

(we use $\triangleq$ to denote a definition). Let us next define the (`partialcost`) function pc^t obtained by retaining only the monomials of $I_1, \ldots, I_t$:

$$pc^t(x_1, \ldots, x_{t+w-1}) \triangleq \sum_{s=1}^{t} sc^s(x_s, \ldots, x_{s+w-1}) = \sum_{s=1}^{t} \sum_{M_i \in I_s} c_i \prod_{j \in M_i} x_j. \tag{2}$$

We remark that the definition implies that $pc^0(x_1, \ldots, x_{w-1}) = 0$. Note that trivially, in view of (2),

$$pc^t(x_1, \ldots, x_{t+w-1}) = pc^{t-1}(x_1, \ldots, x_{t+w-2}) + sc^t(x_t, \ldots, x_{t+w-1}). \tag{3}$$

Moreover, $pc^{\overline{s}} = \sum_{i \in I} c_i \prod_{j \in M_i} x_j$ is the original objective function of (1). In the following, we focus on minimizing $pc^{\overline{s}}$ by dynamic programming. We define accordingly the auxiliary (`cost`) function

$$C^t(x_t, \ldots, x_{t+w-1}) \triangleq \min\{pc^t(\tilde{x}_1, \ldots, \tilde{x}_{t-1}, x_t, \ldots, x_{t+w-1}) : (\tilde{x}_1, \ldots, \tilde{x}_{t-1}) \in \{0,1\}^{t-1}\}. \quad (4)$$

In words, $C^t(x_t, \ldots, x_{t+w-1})$ is the minimum value of $pc^t(\tilde{X})$ among all vectors $\tilde{X}$ that are compatible with $(x_t, \ldots, x_{t+w-1})$. In particular, $C^1(x_1, \ldots, x_w) = pc^1(x_1, \ldots, x_w)$ and

$$\min\{C^t(x_t, \ldots, x_{t+w-1}) : (x_t, \ldots, x_{t+w-1}) \in \{0,1\}^w\} = \min\{pc^t(X) : X \in \{0,1\}^{t+w-1}\},$$

so that we can solve Problem (1) by computing the value of $C^{\overline{s}}(x_{\overline{s}}, \ldots, x_{\overline{s}+w-1})$ for all 2^w assignments of its variables and retaining the minimum value achieved.

Now, we claim that C^t can be computed by the following recursion:

Proposition. *For $t > 1$,*

$$\begin{aligned} C^t(x_t, \ldots, x_{t+w-1}) &= \min\{C^{t-1}(0, x_t, \ldots, x_{t+w-2}), C^{t-1}(1, x_t, \ldots, x_{t+w-2})\} \\ &\quad + sc^t(x_t, \ldots, x_{t+w-1}). \end{aligned}$$

Proof. In view of Eq. (4),

$$\begin{aligned} C^t(x_t, \ldots, x_{t+w-1}) &= \min\{pc^t(\tilde{x}_1, \ldots, \tilde{x}_{t-2}, 0, x_t, \ldots, x_{t+w-1}), \\ &\qquad pc^t(\tilde{x}_1, \ldots, \tilde{x}_{t-2}, 1, x_t, \ldots, x_{t+w-1}) : \\ &\qquad\qquad \tilde{x}_1, \ldots, \tilde{x}_{t-2} \in \{0,1\}\} \end{aligned}$$

and hence, by Eq. (3),

$$\begin{aligned} C^t(x_t, \ldots, x_{t+w-1}) &= \min\{pc^{t-1}(\tilde{x}_1, \ldots, \tilde{x}_{t-2}, 0, x_t, \ldots, x_{t+w-2}) + sc^t(x_t, \ldots, x_{t+w-1}), \\ &\qquad pc^{t-1}(\tilde{x}_1, \ldots, \tilde{x}_{t-2}, 1, x_t, \ldots, x_{t+w-2}) + sc^t(x_t, \ldots, x_{t+w-1}) : \\ &\qquad\qquad \tilde{x}_1, \ldots, \tilde{x}_{t-2} \in \{0,1\}\}. \end{aligned}$$

Using again Eq. (4), we find

$$\min\{pc^{t-1}(\tilde{x}_1, \ldots, \tilde{x}_{t-2}, 0, x_t, \ldots, x_{t+w-2}) : \tilde{x}_1, \ldots, \tilde{x}_{t-2} \in \{0,1\}\} = C^{t-1}(0, x_t, \ldots, x_{t+w-2})$$

and

$$\min\{pc^{t-1}(\tilde{x}_1, \ldots, \tilde{x}_{t-2}, 1, x_t, \ldots, x_{t+w-2}) : \tilde{x}_1, \ldots, \tilde{x}_{t-2} \in \{0,1\}\} = C^{t-1}(1, x_t, \ldots, x_{t+w-2}).$$

Proposition 3 follows immediately. □

The dynamic programming algorithm for computing $C^{\overline{s}}$ follows from Proposition 3 and is shown in Algorithm 1.

Lines 1 to 3 of Algorithm 1 can be computed in time $O(w2^w|I_1|)$, and results can be stored in an array of size 2^w: indeed, we have to compute C^1 for each of the 2^w partial solutions and computing C^1 for a single partial solution takes $O(w|I_1|)$. Similarly, lines 5 to 7 can be computed in time $O(w2^w|I_t|)$ and stored in an array of size 2^w. The minimization in line 9 can be done while computing $C^t(x_t, \ldots, x_{t+w-1})$ in the last iteration of lines 5 to 7 and does therefore not contribute to the running time. Overall the computational complexity is $O(\sum_1^{\overline{s}} w2^w|I_t|) = O(w2^w \sum_1^{\overline{s}} |I_t|) = O(w2^w|I|)$ or simply $O(|I|)$ for a fixed reach w and therefore polynomial in the input size for fixed w. We note

Algorithm 1 Dynamic programming algorithm

1: **for** $(x_1, \ldots, x_w) \in \{0,1\}^w$ **do**
2: Compute and store $C^1(x_1, \ldots, x_w)$
3: **end for**
4: **for** $t = 2 : \bar{s}$ **do**
5: **for** $(x_t, \ldots, x_{t+w-1}) \in \{0,1\}^w$ **do**
6: Compute $C^t(x_t, \ldots, x_{t+w-1})$ using Proposition 3 and store results.
7: **end for**
8: **end for**
9: **return** $\min_{(x_{\bar{s}}, \ldots, x_{\bar{s}+w-1}) \in \{0,1\}^w} C^{\bar{s}}(x_{\bar{s}}, \ldots, x_{\bar{s}+w-1})$

that $|I_t| \leq 2^w - 1$ for all t, assuming that I contains no duplicate monomials. This means that the computational complexity can also be expressed as $O(w2^w \sum_1^{\bar{s}} 2^w) = O(w2^w \bar{s} 2^w) = O(nw4^w)$ or $O(n)$ for a fixed w. We also note that in the instances considered in this paper, $|I_t|$ is much smaller than the upper bound $2^w - 1$.

The algorithm can be modified to return the solution $(x_1, \ldots, x_n)$ that minimizes the objective function by backtracking through the tables with stored results. This does not change the algorithm's computational complexity.

3.1. Dynamic programming algorithm: example

We demonstrate the dynamic program on the following UBP:

$$\begin{aligned} \min\ & 64x_1x_2x_3 + 64x_2x_3x_4 + 64x_3x_4x_5 - 32x_1x_2 - 32x_1x_3 - 64x_2x_3 \\ & - 32x_2x_4 - 32x_3x_5 - 64x_4x_5 + 16x_1 + 32x_2 + 48x_4 \\ & x_i \in \{0,1\} \quad \forall i \in \{1,2,3,4,5\} \end{aligned}$$

The reach of this instance is $w = 3$, and the dynamic programming algorithm has $\bar{s} = 5 - 3 + 1 = 3$ stages. The first (resp. second and third) stage includes variables x_1, x_2, x_3, (resp. x_2, x_3, x_4 and x_3, x_4, x_5) and we have

$I_1 = \{M_1 = \{1,2,3\}, M_2 = \{1,2\}, M_3 = \{1,3\}, M_4 = \{1\}\}$,
$I_2 = \{M_5 = \{2,3,4\}, M_6 = \{2,3\}, M_7 = \{2,4\}, M_8 = \{2\}\}$,
$I_3 = \{M_9 = \{3,4,5\}, M_{10} = \{3,5\}, M_{11} = \{4,5\}, M_{12} = \{4\}\}$.

Figures 1a, 1b, and 1c display the information that is calculated in each of the three stages. Looking at Figure 1a, the first column gives the partial solution index. The second column shows that partial solution (i.e., the assignment of values to variables x_1, x_2, and x_2). The last column shows the value of $C^1(x_1, x_2, x_3)$.

As an example $C^1(1,1,0) = -32 + 16 = -16$ since M_2 and M_4 are the only active monomials from I_1 when $(x_1, x_2, x_3) = (1,1,0)$. One may notice that M_8 also is active for this partial solution, but this monomial is assigned to I_2, not I_1, and is therefore not considered at this point.

The calculations for the second stage are shown in Figure 1b. The first two columns are interpreted as in Figure 1a. The output of the stage, $C^2(x_2, x_3, x_4)$ is shown in column 5. To compute $C^2(x_2, x_3, x_4)$ using Proposition 3 one needs to compute $sc^2(x_2, x_3, x_4)$ and $\min\{C^1(0, x_2, x_3), C^{t-1}(1, x_2, x_3)\}$. The value of $sc^2(x_2, x_3, x_4)$ is shown in column 3, while column 4 points to the rows of Figure 1a that contain $C^1(0, x_2, x_3)$ and $C^{t-1}(1, x_2, x_3)$, where the underlined value indicates the minimum of the two.

a) Stage 1

	X^1	C^1
#	x_1, x_2, x_3	
1	0,0,0	0
2	0,0,1	0
3	0,1,0	0
4	0,1,1	0
5	1,0,0	16
6	1,0,1	-16
7	1,1,0	-16
8	1,1,1	16

b) Stage 2

	X^2	sc^2	comp.	C^2
#	x_2, x_3, x_4			
1	0,0,0	0	1,5	0
2	0,0,1	0	1,5	0
3	0,1,0	0	2,6	-16
4	0,1,1	0	2,6	-16
5	1,0,0	32	3,7	16
6	1,0,1	0	3,7	-16
7	1,1,0	-32	4,8	-32
8	1,1,1	0	4,8	0

c) Stage 3

	X^3	sc^3	comp.	C^3
#	x_3, x_4, x_5			
1	0,0,0	0	1,5	0
2	0,0,1	0	1,5	0
3	0,1,0	48	2,6	32
4	0,1,1	-16	2,6	-32
5	1,0,0	0	3,7	-32
6	1,0,1	-32	3,7	-64
7	1,1,0	48	4,8	32
8	1,1,1	16	4,8	0

Figure 1: Dynamic programming example

Consider, for example, partial solution 7 from stage 2. To compute $C^2(x_2, x_3, x_4)$ for this choice of x_2, x_3, x_4 we need to compute

$$C^2(1,1,0) \quad = \quad \min\{C^1(0,1,1), C^1(1,1,1\} + sc^2(1,1,0).$$

$sc^2(1,1,0)$ evaluates to -32 since monomials $M_6 = \{2,3\}$ and $M_8 = \{2\}$, with objective coefficients -64 and 32, are active for this choice of x_2, x_3, and x_4. The value of $C^1(0,1,1)$ and $C^1(1,1,1\}$ can be found in rows 4 and 8 of Figure 1a and $C^1(0,1,1) = 0$ is the smaller of the two, as indicated in row 7, column 4 of Figure 1b. We then get $C^2(1,1,0) = 0 - 32 = -32$, as indicated in column 5.

Figure 1c is for stage 3 and is similar to Figure 1b. From column 5 of Figure 1c, we see that the optimal solution has value -64 (partial solution 6) and backtracking from partial solution 6 in stage 3 we see that the optimal solution is $(x_1, x_2, x_3, x_4, x_5) = (0,1,1,0,1)$.

Notice that the computation done for a particular row in any of the tables in Figure 1 is independent of the other rows in the table and only uses the information from the previous stage. This means that the rows can be computed in parallel and this parallelization has been used in the implementation of the dynamic programming algorithm.

4. Basic Algorithm

We recall here the Basic Algorithm from Hammer and Rudeanu (1968), using the notation and presentation of Crama et al. (1990). Defining $f(x_1, x_2, \ldots, x_n) = \sum_{i \in I} c_i \prod_{j \in M_i} x_j$, we write a generic UBP as

$$\min_{(x_1,x_2,\ldots,x_n) \in \{0,1\}^n} f(x_1, x_2, \ldots, x_n). \tag{5}$$

We can rewrite $f(x_1, x_2, \ldots, x_n)$ as

$$f(x_1, x_2, \ldots, x_n) = x_1 g_1(x_2, \ldots, x_n) + h_1(x_2, \ldots, x_n),$$

where the functions g_1 and h_1 are readily derived from (5). Based on g_1 we construct a new pseudo-Boolean function

$$\psi_1(x_2, \ldots, x_n) = \begin{cases} g_1(x_2, \ldots, x_n) & \text{if } g_1(x_2, \ldots, x_n) < 0 \\ 0 & \text{otherwise} \end{cases}$$

and we define $f_2(x_2, \ldots, x_n) = \psi_1(x_2, \ldots, x_n) + h_1(x_2, \ldots, x_n)$. Clearly,

$$f_2(x_2, \ldots, x_n) = \min_{x_1} f(x_1, \ldots, x_n),$$

since there is an optimal solution $(x_1^*, x_2^*, \ldots, x_n^*)$ to (5) where $x_1^* = 1$ if $g_1(x_2^*, \ldots, x_n^*) < 0$ and $x_1^* = 0$ otherwise. Consequently,

$$\min_{(x_1,x_2,\ldots,x_n)\in\{0,1\}^n} f(x_1, x_2, \ldots, x_n) = \min_{(x_2,\ldots,x_n)\in\{0,1\}^n} f_2(x_2, \ldots, x_n).$$

The problem has been simplified by eliminating one variable and the procedure can be recursively applied on f_2 until we reach a function with just one variable that can be solved by inspection. The general procedure is shown in Algorithm 2. The time consuming step in Algorithm 2 is constructing $\psi_1(x_2, \ldots, x_n)$ based on $g_1(x_2, \ldots, x_n)$ in line 7. Crama et al. (1990) propose a branch-and-bound algorithm for this step. This algorithm is outlined in Appendix A.

Algorithm 2 Basic Algorithm

1: **function** BASICALG($f(x_1, x_2, \ldots, x_n)$)
2: **if** $n = 1$ **then**
3: **return** $\arg\min_{x_1\in\{0,1\}} f(x_1)$
4: **else**
5: // Eliminate variable x_1
6: construct g_1 and h_1 such that

$$f(x_1, x_2, \ldots, x_n) = x_1 g_1(x_2, \ldots, x_n) + h_1(x_2, \ldots, x_n)$$

7: construct pseudo-Boolean functions

$$\psi_1(x_2, \ldots, x_n) = \begin{cases} g_1(x_2, \ldots, x_n) & \text{if } g_1(x_2, \ldots, x_n) < 0 \\ 0 & \text{otherwise} \end{cases} \text{ and then}$$

$$f_2(x_2, \ldots, x_n) = \psi_1(x_2, \ldots, x_n) + h_1(x_2, \ldots, x_n)$$

8: $(x_2^*, \ldots, x_n^*) =$ BASICALG($f_2(x_2, \ldots, x_n)$)
9: **if** $g_1(x_2^*, \ldots, x_n^*) < 0$ **then**
10: **return** $(1, x_2^*, \ldots, x_n^*)$
11: **else**
12: **return** $(0, x_2^*, \ldots, x_n^*)$
13: **end if**
14: **end if**
15: **end function**

4.1. Basic Algorithm: example

To further illustrate the workings of the Basic Algorithm, we demonstrate the first iteration and show how x_1 can be eliminated from the pseudo-Boolean function given in the example in Section 3.1. The function to be minimized is:

$$\begin{aligned} f_1(x_1, x_2, x_3, x_4, x_5) = {} & 64x_1x_2x_3 - 32x_1x_2 - 32x_1x_3 + 16x_1 \\ & + 64x_2x_3x_4 + 64x_3x_4x_5 - 64x_2x_3 - 32x_2x_4 - 32x_3x_5 - 64x_4x_5 + 32x_2 + 48x_4. \end{aligned}$$

This can be restated as:

$$f_1(x_1, x_2, x_3, x_4, x_5) = x_1 \cdot g_1(x_2, x_3) + h(x_2, x_3, x_4, x_5),$$

where

$$\begin{aligned} g_1(x_2, x_3) &= 64x_2x_3 - 32x_2 - 32x_3 + 16, \text{ and} \\ h_1(x_2, x_3, x_4, x_5) &= 64x_2x_3x_4 + 64x_3x_4x_5 - 64x_2x_3 - 32x_2x_4 - 32x_3x_5 - 64x_4x_5 + 32x_2 + 48x_4. \end{aligned}$$

The highest indexed variable that is part of any monomial that involves x_1 is x_3. Given $g_1(x_2, x_3)$, we can construct the multilinear polynomial $\psi_1(x_2, x_3) = -16x_2 - 16x_3 + 32x_2x_3$ (see Appendix A for more details), and use it to obtain the following pseudo-Boolean function:

$$f_2(x_2, x_3, x_4, x_5) = \psi_1(x_2, x_3) + h_1(x_2, x_3, x_4, x_5).$$

Minimizing $f_1(x_1, x_2, x_3, x_4, x_5)$ is equivalent to minimizing $f_2(x_2, x_3, x_4, x_5)$, where the optimal value of x_1 is encoded in the function $\psi_1(x_2, x_3)$. One would then repeat the process, eliminating x_2 from $f_2(x_2, x_3, x_4, x_5)$, x_3 from $f_3(x_3, x_4, x_5)$, and so on until the value of x_5 can be determined by inspection.

4.2. Connection to the proposed dynamic programming algorithm

The Basic Algorithm and the proposed dynamic program are similar but inherently different. At each stage of each algorithm a single variable is eliminated. The Basic Algorithm minimizes a pseudo-Boolean function $f_1(x_1, x_2, \ldots, x_n)$ by recursively identifying a pseudo-Boolean function ψ_1, expressed as a multilinear polynomial, which can be used to obtain a new pseudo-Boolean function that has one less variable. The proposed dynamic program minimizes a pseudo-Boolean function by partitioning its terms into subsets of monomials based on the reach, w. Each subset corresponds to a stage and has an associated pseudo-Boolean function of w variables. Consecutive stages have $w - 1$ variables in common. The method iteratively enumerates the solutions to each stage k and, for all stages $k > 1$, recursively identifies the best setting of variable x_{k-1}. In what follows, we show, by way of the example in Section 3.1, that the proposed dynamic program implicitly identifies the subfunctions that are used in the recursive step of the Basic Algorithm, without explicitly generating their algebraic, multilinear polynomial representation. Recall that the example is:

$$\begin{aligned} \min f_1(x_1, x_2, x_3, x_4, x_5) = {} & 64x_1x_2x_3 + 64x_2x_3x_4 + 64x_3x_4x_5 - 32x_1x_2 - 32x_1x_3 - 64x_2x_3 \\ & - 32x_2x_4 - 32x_3x_5 - 64x_4x_5 + 16x_1 + 32x_2 + 48x_4 \\ x_i \in \{0, 1\} \quad & \forall i \in \{1, 2, 3, 4, 5\}. \end{aligned}$$

For reference, we also restate the sets of monomials I_k for $k = 1, 2, 3$:

$$\begin{aligned} I_1 &= \{\{1, 2, 3\}, \{1, 2\}, \{1, 3\}, \{1\}\} \\ I_2 &= \{\{2, 3, 4\}, \{2, 3\}, \{2, 4\}, \{2\}\} \\ I_3 &= \{\{3, 4, 5\}, \{3, 5\}, \{4, 5\}, \{4\}\}. \end{aligned}$$

In the first stage of the proposed dynamic program, only monomials in I_1 are considered. The corresponding pseudo-Boolean function to minimize is therefore:

$$C^1(x_1, x_2, x_3) = 64x_1x_2x_3 - 32x_1x_2 - 32x_1x_3 + 16x_1.$$

The proposed dynamic program begins by enumerating all of the possible solutions to $C^1(x_1, x_2, x_3)$. By retaining the information regarding the possible choices for x_1, the variable can be eliminated from explicit consideration in the second stage. This is equivalent to applying the first iteration of the Basic Algorithm to $C^1(x_1, x_2, x_3)$ and eliminating x_1, with the key difference being that the algebraic form of the function $\psi_1(x_2, x_3)$ is *not* identified. Instead, its values are obtained through a comparison of tabulated entries. Let us consider applying the first iteration of the Basic Algorithm to $C^1(x_1, x_2, x_3)$. We know that $C^1(x_1, x_2, x_3)$ can be rewritten as

$$C^1(x_1, x_2, x_3) = x_1 \cdot g_1(x_2, x_3),$$

where $g_1(x_2, x_3) = 64x_2x_3 - 32x_2 - 32x_3 + 16$. Note that in the first stage, all monomials depend on x_1. As such, there is no $h_1(x_2, x_3)$ function. Based on $g_1(x_2, x_3)$, we can construct the following multilinear polynomial:

$$\psi_1(x_2, x_3) = -16x_2 - 16x_3 + 32x_2x_3.$$

Since the optimal value of x_1 is encoded in the function $\psi_1(x_2, x_3)$, this could be used instead of the alternative tabulated form employed by the proposed dynamic program. The advantage of the tabulated form, however, is that one can avoid identifying the algebraic form of $\psi(x_2, x_3)$, which is computationally demanding in general. By eliminating variable x_1, we can formally define the pseudo-Boolean function that is considered at the second stage of the proposed dynamic program as:

$$C^2(x_2, x_3, x_4) = \underbrace{\psi_1(x_2, x_3)}_{comp} + \underbrace{64x_2x_3x_4 - 64x_2x_3 - 32x_2x_4 + 32x_2}_{sc^2} \tag{6}$$

For ease of reference, we underline components of the equation that correspond to columns of the tables given in the example in Section 3. This function contains a component that arises from the elimination of x_1 and a component that corresponds to the monomials in I_2, since the latter appear when proceeding to the second stage. The proposed dynamic algorithm does not identify the algebraic form of $C^2(x_2, x_3, x_4)$; its values are computed using the tabulated form of $\psi_1(x_2, x_3)$ and the pseudo-Boolean function specifically associated with the monomials of set I_2.

The second stage of the proposed dynamic program proceeds similarly. With an enumerated set of values for $C^2(x_2, x_3, x_4)$, variables x_1 and x_2 need not be explicitly considered in the third stage. This enumeration procedure is equivalent to applying an iteration of the Basic Algorithm on $C^2(x_2, x_3, x_4)$ and eliminating x_2. We note that $C^2(x_2, x_3, x_4)$ can be rewritten as

$$C^2(x_2, x_3, x_4) = x_2 \cdot g_2(x_3, x_4) + h_2(x_3, x_4),$$

where $g_2(x_3, x_4) = -32x_3 + 64x_3x_4 - 32x_4 + 16$ and $h_2(x_3, x_4) = -16x_3$. We can then construct the pseudo-Boolean function

$$\psi_2(x_3, x_4) = -16x_3 - 16x_4 + 32x_3x_4$$

and use it to define the pseudo-Boolean function that is implicitly considered at the third stage as:

$$C^3(x_3, x_4, x_5) = \underbrace{\psi_2(x_3, x_4) + h_2(x_3, x_4)}_{comp} + \underbrace{64x_3x_4x_5 - 32x_3x_5 - 64x_4x_5 + 48x_4}_{sc^3}$$

Like (6), this contains a part that encodes the optimal setting of the eliminated variables x_1 and x_2, given x_3 and x_4, and a part that contains all monomials that appear when moving to the third stage. Note that the algebraic forms of the functions $\psi_2(x_3, x_4)$ and $h_2(x_3, x_4)$ are not identified. The

proposed dynamic program provides an enumerated set of values for $\psi_2(x_3, x_4) + h_2(x_3, x_4)$ through a comparison of tabulated entries for the eliminated variable x_2. Since $C^3(x_3, x_4, x_5)$ considers the same set of monomials and either explicitly or implicitly considers all variables, minimizing $C^3(x_3, x_4, x_5)$ is equivalent to minimizing $f_1(x_1, x_2, x_3, x_4, x_5)$. The implementation of the proposed dynamic algorithm is given below. We have included the $\psi(.)$ and $h(.)$ functions for comparison.

Stage 1

x_1	x_2	x_3	C^1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	16
1	0	1	-16
1	1	0	-16
1	1	1	16

Stage 2

x_2	x_3	x_4	*comp.*	$\psi_1(x_2, x_3)$	$h_1(x_2, x_3)$	sc^2	C^2
0	0	0	$\min\{0, 16\}$	0	0	0	0
0	0	1	$\min\{0, 16\}$	0	0	0	0
0	1	0	$\min\{0, -16\}$	-16	0	0	-16
0	1	1	$\min\{0, -16\}$	-16	0	0	-16
1	0	0	$\min\{0, -16\}$	-16	0	32	16
1	0	1	$\min\{0, -16\}$	-16	0	0	-16
1	1	0	$\min\{0, 16\}$	0	0	-32	-32
1	1	1	$\min\{0, 16\}$	0	0	0	0

Stage 3

x_3	x_4	x_5	*comp.*	$\psi_2(x_3, x_4)$	$h_2(x_3, x_4)$	sc^3	C^3
0	0	0	$\min\{0, 16\}$	0	0	0	0
0	0	1	$\min\{0, 16\}$	0	0	0	0
0	1	0	$\min\{0, -16\}$	-16	0	48	32
0	1	1	$\min\{0, -16\}$	-16	0	-16	-32
1	0	0	$\min\{-16, -32\}$	-16	-16	0	-32
1	0	1	$\min\{-16, -32\}$	-16	-16	-32	-64
1	1	0	$\min\{-16, 0\}$	0	-16	48	32
1	1	1	$\min\{-16, 0\}$	0	-16	16	0

In summary, the proposed dynamic program is similar to the Basic Algorithm in that it iteratively eliminates a single variable. However, unlike the Basic Algorithm, it does not identify the algebraic forms of the multilinear polynomials that are needed in the recursive step. Instead, an enumeration of the values of these functions is available in tabulated form. In general, the comparison step at stage

k of the proposed dynamic program provides an enumeration of the values of $\psi_{k-1}(x_k, \ldots, x_{k+w-2}) + h_{k-1}(x_{k+1}, \ldots, x_{k+w-2})$. Not needing to identify the algebraic forms of the multilinear polynomials may alleviate some of the computational burden of the Basic Algorithm. We also note, however, that for general pseudo-Boolean functions (i.e., not necessarily with limited reach) an enumeration in tabulated form can be very expensive. A limited reach controls the level of enumeration needed, and this is what the dynamic programming algorithm exploits.

5. Computational results

We test the proposed dynamic programming approach on a publicly available set of benchmark LABS instances and compare its performance with that of several other approaches. In particular, we compare it with solutions provided by a modern implementation of the Basic Algorithm, by the commercial solver CPLEX on the standard linearization, by CPLEX on the standard linearization with 2-link inequalities added as a pool of cuts (Crama and Rodríguez-Heck, 2017), and by CPLEX given a simple quadratization. As is the case for linearizations, there exist many possibilities to define a quadratization (see for example Anthony et al. (2017); Boros et al. (2020); Crama et al. (2022); Fix et al. (2015); Freedman and Drineas (2005); Ishikawa (2011); Verma and Lewis (2020)). In this paper, we use the so-called to `Lex` quadratization, since it has been shown to work well in the experiments performed by Elloumi et al. (2021). Furthermore, we also include the results for the Partial Quadratic Convex Reformulation (PQCR) approach reported in Elloumi et al. (2021). We refer readers to the above-mentioned references for additional details.

We provide a brief description of the LABS problem and of the available instances in Section 5.1, and then report on the performance of the different approaches in Section 5.2

5.1. LABS instances

As defined in Liers et al. (2010), the 1-dimensional Low Autocorrelation Binary Sequence problem with tunable interaction range w for n Ising spins (or LABS problem for short) is the following optimization problem:

$$E_w(s) = \min_{s \in \{-1,1\}^n} \sum_{i=1}^{n-w+1} \sum_{d=1}^{w-1} \left(\sum_{j=i}^{i+w-1-d} s_j s_{j+d} \right)^2 . \tag{7}$$

Although $s \in \{-1, 1\}^n$, a conversion to the domain $x \in \{0, 1\}^n$ can be made with the usual transformation $x_i = \frac{s_i+1}{2}$ for $i = 1, \ldots, n$, hence the LABS problem is a particular case of the Unconstrained Binary Polynomial Program (1) with degree at most 4 and reach w. (The constant term that occurs after the transformation of (7) from $(-1, 1)$ to $(0, 1)$-variables is usually discarded from the resulting UBP instances, and their optimal value is therefore negative.)

In our experiments, we consider instances of LABS for different values of $n \in \{20, \ldots, 60\}$ and of $w \in \{3, \ldots, n\}$. This instance set is publicly available at (MINLPLib, 2020; POLIP, 2020). The structure of the instances is described by the scheme "$n.w$", e.g., 20.3, or 40.5, where n is the number of original variables in the instance and w is the reach. The LABS instances are not complete, in the sense that they do not necessarily include all monomials of reach w, they are simply limited to monomials of reach w or less. As an illustration, Figure 2 shows the structure of the co-occurrence graph of instance 20.5. All but two of our instances have a degree of four. Exceptions to this are 20.3 and 25.3, which each have a degree of two. Despite the rather small number of variables, the largest LABS instances are remarkably difficult to solve by classical integer programming methods.

Remark. For unexplained reasons, we are not able to recreate the instances from the POLIP library using formula (7) when $w > \frac{n}{2}$. We find a small number of discrepancies between the coefficients of the POLIP instances and those computed by (7). The results mentioned in Section 5.2 refer to the solution of the instances downloaded from POLIP.

5.2. Performance comparison

Table 1 compares the performance of the different methods when applied to the LABS instances. All tests, except the PQCR experiments, were performed on a computer with a 2.9GHz Intel Xeon Gold 6226R dual-processor (32 cores) with 756 GB memory, while all algorithms were limited to using one thread/core. A time limit of three hours was enforced and CPLEX 12.10 was used. The results for the PQCR experiments was obtained in Elloumi et al. (2021) where a server with 64GB of RAM and two 2.5GHz Intel CPUs (24 cores in total) is used. The time limit for the PQCR experiments is three hours as well and the algorithms are, to the best of our understanding, allowed to use all cores/threads.

The first column of Table 1 shows the name of the instance, the second column shows the optimal objective value when known (up to a positive constant due to the change of variables from s_i to x_i). The objective values for 10 instances that have been solved to optimality for the first time are displayed in bold. Objective values that are underlined indicate 3 additional instances that have been optimally solved for the first time; however, they have been solved with the parallelized version of the dynamic programming algorithm and they will be discussed separately. Columns three to five show results from CPLEX. These columns either show the time to solve the instance to optimality or the optimality gap if the problem isn't solved to optimality within the time limit. Column three shows results from the standard linearization of the UBP, column four shows results from the standard linearization plus the 2-link inequalities suggested in Crama and Rodríguez-Heck (2017). Column five shows results from letting CPLEX solve a quadratic binary problem resulting from the `Lex` quadratization (see Elloumi et al. (2021)). Column six shows results from the PQCR method as described in Elloumi et al. (2021). (We note that instance 20.20 is missing from the results presented in (Elloumi et al., 2021) and is, at the time of writing, not displayed on the web pages of POLIP and MINLPLib. It is, however, available when downloading the complete set of instances from (POLIP, 2020).) Dashes indicate that PQCR didn't compute a lower bound within the time limit. Columns seven and eight show the computation time for the dynamic programming algorithm and the Basic Algorithm, respectively. Dashes indicate that the instance was not solved within the time limit or that the method ran out of memory. Results in columns seven and eight were produced using one thread. For all methods the running time is rounded to one decimal. The last row shows the number of instances solved to optimality for each method. We note that the PQCR method is able to solve three more instances to optimality (22 in total) if allowed a running time of 10 hours, and that the instances solved by the parallelized version of the dynamic programming algorithm are not counted in the last row.

To the best of our knowledge, the PQCR method is the algorithm that has been able to solve the largest number of LABS instances to optimality before the present paper. Table 1, however, clearly shows that both the dynamic programming algorithm and the Basic Algorithm solve more instances to optimality compared to PQCR, and that the methods also outperform a generic solver like CPLEX. Even in its simplest, non parallelized version, the dynamic programming algorithm computes 10 previously unknown optimal values, when compared to the best results presented by Elloumi et al. (2021). The difference in running time between the previous state-of-the-art and the two algorithms studied in this paper can be tremendous. Instance 40.10 is a good example of this. PQCR solves the instance in 10550 seconds. The dynamic programming algorithm needs less than 0.05 seconds and the Basic Algorithm uses 0.3 seconds. Comparing the dynamic programming to

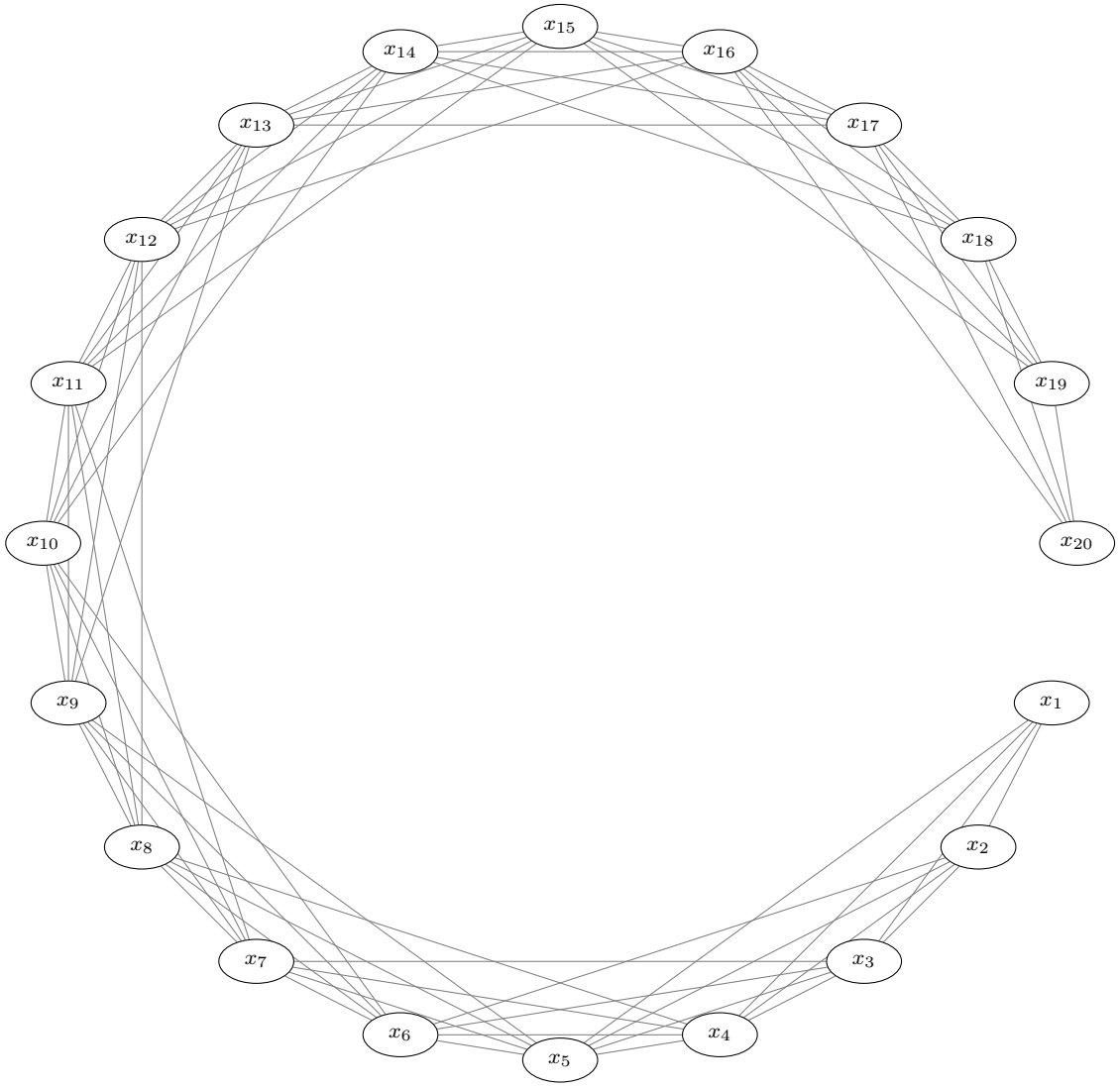

Figure 2: Shows the structure of auto-correlated sequence instance 20.5, nodes (variables) are connected with edges if they appear in a monomial together.

the Basic Algorithm we see that the dynamic programming algorithm clearly does better, being able to solve 5 more instances to optimality and often being more than a magnitude faster on the instances both algorithms can solve.

Of course, as expected, the performance of the dynamic programming algorithm deteriorates when the reach increases. When $w = n$, in particular, the algorithm only involves one stage which boils down to assigning all (2^n) possible binary values to the variables. More generally, when w is close to n, dynamic programming has no clear advantage over complete enumeration. In practice, with the computational resources used in our experiments, values of the reach above $w = 30$ cannot be efficiently handled by the algorithm.

Table 2 provides detailed results for the Basic Algorithm and gives some insight into why the dynamic programming algorithm performs better on the LABS instances. Columns two and three of the table show the number of variables and monomials in each instance, respectively, while column four indicates time spent. Columns five and six show statistics from the branch-and-bound method described in Appendix A. The branch-and-bound method is called $n - 1$ times. Column five states the maximum number of branch-and-bound nodes in any of these $n - 1$ invocations, while column six shows the total number of branch-and-bound nodes over all $n - 1$ calls. Looking at instance 20.20 as an example, we see that almost 200,000 branch-and-bound nodes are explored in the most difficult call, which most likely is the first since g_1 contains most variables in this case (note that the reach is n in this particular instance). Since g_1 depends at most on $n - 1$ variables, we know that the full branch-and-bound tree for the computation of ψ_1 in the first call contains at most $2^n - 1$ nodes (of which 2^{n-1} are leaves). So in the worst case the branch-and-bound tree might contain $2^{20} - 1 = 1,048,575$ nodes, whereas it only contains around 200,000 nodes in the actual computation. We conclude that the lower and upper bounds have pruned some parts of the tree; however, this reduction is too insignificant to make the method competitive with the dynamic programming algorithm on these instances. The dynamic programming method will, on the 20.20 instance, enumerate the objective function corresponding to all $2^{20} = 1,048,576$ possible assignments of $0/1$ values to the 20 variables in the instance. The computation needed for testing one variable assignment in the dynamic programming algorithm is in general much simpler than the work needed to evaluate one branch-and-bound node that entails lower and upper bound computations as well as combining two UBPs in each of the internal nodes. Column 3 of Table 2 shows that $|I|$ is much lower than the maximum possible for a UBP instance with the same reach and confirms that the LABS instances are *sparse*, which is beneficial for the dynamic programming algorithm (see discussion on computational complexity in Section 3).

Table 3 shows the effect of the simple parallelization of the dynamic programming algorithm, suggested in Section 3. The table displays results from allowing the algorithm to use 1,2,4,8,16 or 32 threads. The one thread results are identical to those reported in Table 1. Times reported are "wall-clock" time. We report the speedup relative to the one-thread result for the results obtained using more than one thread. If t_x is the wall clock time for solving a specific instance using x threads, then the speedup using x threads is calculated as

$$\frac{t_1}{t_x}.$$

No speedup is reported if the one-thread version did not solve a particular instance. We see that impressive speedups are possible on the most challenging instances. Instance 30.30 is an example of this, where a factor 29.2 speedup is obtained using 32 threads. We observe that the speedup often is less than 1.0 for easy instances meaning that the parallel version spends a longer time compared to the sequential version. This is to be expected due to the overhead involved in spawning multiple

name	opt objective	CPLEX time (s)	CPLEX 2links time (s)	CPLEX Lex time (s)	PQCR time (s)	Dynamic programming time (s)	Basic algorithm (s)
20.03	-72	0.0	0.0	0.0	0	0.0	0.0
20.05	-416	9.7	8.5	7.5	0	0.0	0.0
20.10	-2936	154.7	111.5	62.6	34	0.0	0.1
20.15	-5960	416.5	542.0	189.4	3435	0.1	2.7
20.20	-4648	1301.6	1189.2	323.9	missing	1.8	35.6
25.03	-92	0.0	0.0	0.0	0	0.0	0.0
25.06	-960	311.4	119.2	62.7	6	0.0	0.0
25.13	-8148	3201.2	1961.1	1232.6	3665	0.0	1.0
25.19	-14644	6.47%	1.07%	3403.6	8788	1.8	70.1
25.25	-10664	75.05%	48.5%	7353.0	4832	101.4	1930.9
30.04	-324	10.3	5.0	6.8	1	0.0	0.0
30.08	-2952	33.93%	5067.1	5358.1	857	0.0	0.0
30.15	-15744	47.71%	30.26%	14.72%	7456	0.1	6.6
30.23	-30460	>100%	95.49%	72.42%	8118	43.5	1970.2
30.30	-22888	>100%	>100%	>100%	9838	5303.9	-
35.04	-384	192.5	13.8	13.4	2	0.0	0.0
35.09	-5108	>100%	53.4%	48.61%	7833	0.0	0.2
35.18	-31168	>100%	>100%	75.96%	0.3%	1.7	76.4
35.26	**-55288**	>100%	>100%	>100%	2.8%	501.9	-
35.35	-	>100%	>100%	>100%	5.2%	-	-
40.05	-936	9.69%	2535.5	1761.8	34	0.0	0.0
40.10	-8248	>100%	>100%	>100%	10550	0.0	0.3
40.20	**-50576**	>100%	>100%	>100%	3.9%	8.8	448.5
40.30	-94992	>100%	>100%	>100%	-	-	-
40.40	-	>100%	>100%	>100%	10.6%	-	-
45.05	-1068	16.43%	8969.7	4025.1	67	0.0	0.0
45.11	-12748	>100%	>100%	>100%	1.3%	0.0	0.7
45.23	**-85504**	>100%	>100%	>100%	4.9%	93.2	-
45.34	-152784	>100%	>100%	>100%	-	-	-
45.45	-	>100%	>100%	>100%	-	-	-
50.06	**-2160**	>100%	45.3%	30.91%	2.8%	0.0	0.0
50.13	-23792	>100%	>100%	>100%	1.7%	0.1	3.6
50.25	**-125104**	>100%	>100%	>100%	-	468.2	-
50.38	-	>100%	>100%	>100%	-	-	-
50.50	-	>100%	>100%	>100%	-	-	-
55.06	**-2400**	>100%	59.55%	45.77%	3.1%	0.0	0.1
55.14	**-33272**	>100%	>100%	>100%	3.7%	0.2	8.2
55.28	**-191032**	>100%	>100%	>100%	-	4780.0	-
55.41	-	>100%	>100%	>100%	-	-	-
55.55	-	>100%	>100%	>100%	-	-	-
60.08	**-6792**	>100%	>100%	>100%	3.20%	0.0	0.2
60.15	**-45232**	>100%	>100%	>100%	-	0.4	20.7
60.30	-261368	>100%	>100%	>100%	-	-	-
60.45	-	>100%	>100%	>100%	-	-	-
60.60	-	>100%	>100%	>100%	-	-	-
#opt		10	13	15	19	33	28

Table 1: Performance comparison on the LABS instances.

threads. We performed the computational tests on a high-performance computing setup where resources are shared with other users. This means that timings are associated with a degree of uncertainty since programs from other users may have used threads not occupied by our experiment, and such programs could have been competing with our program, e.g., for memory access.

The parallelization enables us to solve three more instances to optimality (highlighted in bold in Table 3). The most difficult of these is instance 45.34. Solving this instance needs around 448 GB of memory to store tables. There are ways of reducing the memory footprint, but memory consumption will be a limiting factor when attempting to solve more difficult instances using dynamic programming.

6. Conclusion

In this paper, we have proposed a simple dynamic programming algorithm for solving UBP instances having limited *reach*. The dynamic programming algorithm allows us to solve many instances from the LABS set of instances to optimality for the first time. The LABS instances are part of the (MINLPLib, 2020) and (POLIP, 2020) instance library and provide very difficult UBP instances. We have shown that the dynamic programming algorithm is closely related to the Basic Algorithm due to Hammer et al. (1963a,b), and that the Basic Algorithm also performs very well on the LABS instances.

For future work, it could be interesting to use either the dynamic programming algorithm or the Basic Algorithm to solve sub-problems in a decomposition algorithm for the UBP, based on either Lagrangian decomposition or Dantzig-Wolfe decomposition with linking variables.

Acknowledgements

The authors gratefully acknowledge the financial support of the Danish Council for Independent Research (Grant ID: DFF - 7017-00341). We would also like to thank Marco Lübbecke and the Chair of Operations Research, RWTH Aachen University for hosting the research stay of Jens Vinther Clausen, without which this paper would have never happened.

References

Adjiman, C., Dallwig, S., Floudas, C., Neumaier, A., 1998. A global optimization method, αbb, for general twice-differentiable constrained NLPs-I. Theoretical advances. Computers and Chemical Engineering 22, 1137–1158.

Anstee, R., 1983. Hypergraphs with no special cycles. Combinatorica 3, 141–146.

Anthony, M., Boros, E., Crama, Y., Gruber, A., 2016. Quadratization of symmetric pseudo-Boolean functions. Discrete Applied Mathematics 203, 1 – 12.

Anthony, M., Boros, E., Crama, Y., Gruber, A., 2017. Quadratic reformulations of nonlinear binary optimization problems. Mathematical Programming 162, 115–144.

Beeri, C., Fagin, R., Maier, D., Yannakakis, M., 1983. On the desirability of acyclic database schemes. Journal of the ACM (JACM) 30, 479–513.

Berge, C., 1976. Graphs and Hypergraphs. North-Holland Publishing Company, Amsterdam.

name	n	$\|I\|$	time (s)	max BB nodes	sum BB nodes
20.03	20	38	0.0	2	36
20.05	20	207	0.0	8	133
20.10	20	833	0.1	244	2,541
20.15	20	1494	2.7	7,317	39,109
20.20	20	1859	35.6	197,794	375,213
25.03	25	48	0.0	2	46
25.06	25	407	0.0	19	391
25.13	25	1782	1.0	1,703	19,332
25.19	25	3040	70.1	101,233	599,760
25.25	25	3677	1930.9	4,974,796	9,887,083
30.04	30	223	0.0	8	216
30.08	30	926	0.0	83	1,374
30.15	30	2944	6.6	7,317	83,440
30.23	30	5376	1970.2	1,435,147	9,054,042
30.30	30	6412	-	-	-
35.04	35	263	0.0	8	256
35.09	35	1381	0.2	138	3,420
35.18	35	5002	76.4	47,593	622,590
35.26	35	8347	-	-	-
35.35	35	9391	-	-	-
40.05	40	447	0.0	8	293
40.10	40	2053	0.3	244	6,614
40.20	40	7243	448.5	197,789	2,657,782
40.30	40	12690	-	-	-
40.40	40	15384	-	-	-
45.05	45	507	0.0	8	338
45.11	45	2813	0.7	523	14,607
45.23	45	10776	-	-	-
45.34	45	18348	-	-	-
45.45	45	21993	-	-	-
50.06	50	882	0.0	19	866
50.13	50	4457	3.6	1,703	49,635
50.25	50	14412	-	-	-
50.38	50	25446	-	-	-
50.50	50	30271	-	-	-
55.06	55	977	0.1	19	961
55.14	55	5790	8.2	3,441	98,347
55.28	55	19897	-	-	-
55.41	55	33318	-	-	-
55.55	55	40402	-	-	-
60.08	60	2036	0.2	83	2,964
60.15	60	7294	20.7	7317	210,231
60.30	60	25230	-	-	-
60.45	60	43689	-	-	-
60.60	60	52575	-	-	-

Table 2: Basic Algorithm detailed results .

	1 thread	2 threads		4 threads		8 threads		16 threads		32 threads	
Name	time (s)	time (s)	speedup	time (s)	speedup	time (s)	speedup	time (s)	speedup	time (s)	speedup
20.03	0.0	0.0	0.7	0.0	0.6	0.0	0.2	0.0	0.1	0.0	0.1
20.05	0.0	0.0	0.6	0.0	0.6	0.0	0.4	0.0	0.2	0.0	0.1
20.10	0.0	0.0	1.0	0.0	1.0	0.0	1.5	0.0	1.7	0.0	1.2
20.15	0.1	0.0	1.4	0.0	2.2	0.0	3.5	0.0	5.3	0.0	7.7
20.20	1.8	0.9	1.9	0.5	3.7	0.3	6.9	0.1	12.5	0.1	21.1
25.03	0.0	0.0	0.5	0.0	0.6	0.0	0.2	0.0	0.1	0.0	0.1
25.06	0.0	0.0	0.7	0.0	0.7	0.0	0.4	0.0	0.3	0.0	0.1
25.13	0.0	0.0	1.0	0.0	1.5	0.0	2.6	0.0	4.1	0.0	5.0
25.19	1.8	1.1	1.6	0.6	3.0	0.3	5.8	0.2	10.9	0.1	19.4
25.25	101.4	51.5	2.0	26.2	3.9	13.4	7.6	6.9	14.7	3.7	27.7
30.04	0.0	0.0	0.7	0.0	0.6	0.0	0.3	0.0	0.1	0.0	0.1
30.08	0.0	0.0	0.7	0.0	1.0	0.0	1.1	0.0	0.5	0.0	0.2
30.15	0.1	0.1	1.3	0.1	2.3	0.0	4.0	0.0	3.8	0.0	7.4
30.23	43.5	26.4	1.6	13.6	3.2	7.0	6.2	4.6	9.4	1.9	22.4
30.30	5303.9	2705.4	2.0	1355.5	3.9	686.9	7.7	457.8	11.6	181.9	29.2
35.04	0.0	0.0	0.7	0.0	0.5	0.0	0.4	0.0	0.1	0.0	0.1
35.09	0.0	0.0	0.6	0.0	0.9	0.0	1.2	0.0	0.7	0.0	0.5
35.18	1.7	1.1	1.5	0.6	2.8	0.3	5.3	0.2	7.4	0.1	16.9
35.26	501.9	305.8	1.6	153.9	3.3	78.1	6.4	43.7	11.5	20.8	24.1
35.35	-	-	-	-	-	-	-	-	-	-	-
40.05	0.0	0.0	0.5	0.0	0.7	0.0	0.3	0.0	0.1	0.0	0.1
40.10	0.0	0.0	0.8	0.0	1.0	0.0	1.4	0.0	1.3	0.0	1.0
40.20	8.8	5.9	1.5	3.1	2.9	1.6	5.6	1.0	8.6	0.6	15.3
40.30	-	6611.8	-	3354.9	-	1708.2	-	949.1	-	447.7	-
40.40	-	-	-	-	-	-	-	-	-	-	-
45.05	0.0	0.0	0.8	0.0	0.6	0.0	0.3	0.0	0.2	0.0	0.1
45.11	0.0	0.0	0.8	0.0	1.1	0.0	1.6	0.0	1.3	0.0	1.8
45.23	93.2	60.8	1.5	31.3	3.0	16.0	5.8	9.3	10.0	4.6	20.1
45.34	-	-	-		-		-	-	-	9604.4	-
45.45	-	-	-	-	-	-	-	-	-	-	-
50.06	0.0	0.0	0.7	0.0	0.8	0.0	0.7	0.0	0.3	0.0	0.1
50.13	0.1	0.1	1.2	0.0	1.9	0.0	3.0	0.0	3.9	0.0	4.3
50.25	468.2	305.4	1.5	154.4	3.0	78.5	6.0	42.1	11.1	20.7	22.6
50.38	-	-	-	-	-	-	-	-	-	-	-
50.50	-	-	-	-	-	-	-	-	-	-	-
55.06	0.0	0.0	0.5	0.0	0.8	0.0	0.7	0.0	0.2	0.0	0.1
55.14	0.2	0.1	1.3	0.1	2.2	0.0	3.8	0.0	5.6	0.0	5.7
55.28	4780.0	3001.5	1.6	1528.2	3.1	781.1	6.1	471.4	10.1	203.9	23.4
55.41	-	-	-	-	-	-	-	-	-	-	-
55.55	-	-	-	-	-	-	-	-	-	-	-
60.08	0.0	0.0	1.3	0.0	1.3	0.0	1.3	0.0	0.5	0.0	0.3
60.15	0.4	0.3	1.4	0.2	2.5	0.1	4.4	0.1	7.2	0.1	7.3
60.30	-	-	-	7369.7	-	3744.4	-	2126.9	-	979.2	-
60.45	-	-	-	-	-	-	-	-	-	-	-
60.60	-	-	-	-	-	-	-	-	-	-	-
#solved	33	34		35		35		35		36	

Table 3: Dynamic programming, impact of parallelization

Bernasconi, J., 1987. Low autocorrelation binary sequences: statistical mechanics and configuration space analysis. Journal de Physique 48, 559–567.

Bodlaender, H.L., 1998. A partial k-arboretum of graphs with bounded treewidth. Theoretical Computer Science 209, 1–45.

Boros, E., Crama, Y., Rodríguez-Heck, E., 2020. Compact quadratizations for pseudo-Boolean functions. Journal of Combinatorial Optimization 39, 1–21.

Boros, E., Hammer, P.L., 2002. Pseudo-Boolean optimization. Discrete Applied Mathematics 123, 155–225.

Buchheim, C., Crama, Y., Rodríguez-Heck, E., 2019. Berge-acyclic multilinear 0–1 optimization problems. European Journal of Operational Research 273, 102–107.

Buchheim, C., Rinaldi, G., 2007. Efficient reduction of polynomial zero-one optimization to the quadratic case. SIAM Journal on Optimization 18, 1398–1413.

Clausen, J.V., 2021. Automatic Dantzig-Wolfe Reformulation of Mixed Integer Linear Programs. Ph.D. thesis. The Technical University of Denmark.

Conway, A.R., 2017. The design of efficient dynamic programming and transfer matrix enumeration algorithms. Journal of Physics A: Mathematical and Theoretical 50, 353001.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2009. Introduction to Algorithms. 3rd ed., MIT Press, Cambridge.

Crama, Y., Elloumi, S., Lambert, A., Rodriguez-Heck, E., 2022. Quadratization and convexification in polynomial binary optimization. Technical Report. HEC Liège - Management School, University of Liège, Belgium. URL: `https://hdl.handle.net/2268/295296`.

Crama, Y., Hammer, P.L., 2011. Boolean Functions: Theory, Algorithms, and Applications. Cambridge University Press, New York, N.Y.

Crama, Y., Hansen, P., Jaumard, B., 1990. The basic algorithm for pseudo-Boolean programming revisited. Discrete Applied Mathematics 29, 171–185.

Crama, Y., Rodríguez-Heck, E., 2017. A class of valid inequalities for multilinear 0-1 optimization problems. Discrete Optimization 25, 28–47.

De Simone, C., 1990. The cut polytope and the Boolean quadric polytope. Discrete Mathematics 79, 71–75.

Dearing, P.M., Hammer, P.L., Simeone, B., 1992. Boolean and graph theoretic formulations of the Simple Plant Location Problem. Transportation Science 26, 138–148.

Del Pia, A., Di Gregorio, S., 2022. On the complexity of binary polynomial optimization over acyclic hypergraphs, in: Proceedings of SODA 2022.

Del Pia, A., Khajavirad, A., 2016. A polyhedral study of binary polynomial programs. Mathematics of Operations Research 42, 389–410.

Del Pia, A., Khajavirad, A., 2018. The multilinear polytope for acyclic hypergraphs. SIAM Journal on Optimization 28, 1049–1076.

Del Pia, A., Khajavirad, A., 2018. On decomposability of multilinear sets. Mathematical Programming 170, 387–415.

Del Pia, A., Khajavirad, A., Sahinidis, N.V., 2020. On the impact of running intersection inequalities for globally solving polynomial optimization problems. Mathematical Programming Computation 12, 165–191.

Elloumi, S., Lambert, A., Lazare, A., 2021. Solving unconstrained 0-1 polynomial programs through quadratic convex reformulation. Journal of Global Optimization 80, 231–248.

Fagin, R., 1983. Degrees of acyclicity for hypergraphs and relational database schemes. Journal of the ACM (JACM) 30, 514–550.

Fischer, A., Fischer, F., McCormick, S.T., 2018. Matroid optimisation problems with nested non-linear monomials in the objective function. Mathematical Programming 169, 417–446.

Fix, A., Gruber, A., Boros, E., Zabih, R., 2015. A hypergraph-based reduction for higher-order binary Markov random fields. IEEE Transactions on Pattern Analysis and Machine Intelligence 37, 1387–1395.

Fortet, R., 1959. L'algèbre de Boole et ses applications en recherche opérationnelle. Cahiers du Centre d'Études de Recherche Opérationnelle 4, 5–36.

Freedman, D., Drineas, P., 2005. Energy minimization via graph cuts: settling what is possible, in: IEEE Conference on Computer Vision and Pattern Recognition, pp. 939–946.

Glover, F., Kochenberger, G., Du, Y., 2019. Quantum bridge analytics I: a tutorial on formulating and using QUBO models. 4OR 17, 335–371.

Glover, F., Woolsey, E., 1973. Further reduction of zero-one polynomial programming problems to zero-one linear programming problems. Operations Research 21, 156–161.

Glover, F., Woolsey, E., 1974. Technical note: converting the 0-1 polynomial programming problem to a 0-1 linear program. Operations Research 22, 180–182.

Goldengorin, B., Ghosh, D., Sierksma, G., 2003. Branch and peg algorithms for the simple plant location problem. Computers & Operations Research 30, 967–981.

Goldengorin, B., Krushinsky, D., 2011. Complexity evaluation of benchmark instances for the p-median problem. Mathematical and Computer Modelling 53, 1719–1736.

Hammer, P.L., 1968. Plant location - A pseudo-Boolean approach. Israel Journal of Technology 6, 330–332.

Hammer, P.L., Rosenberg, I., Rudeanu, S., 1963a. Application of discrete linear programming to the minimization of boolean functions. Revue de Mathématiques Pures et Appliquées 8, 459–475. In Russian.

Hammer, P.L., Rosenberg, I., Rudeanu, S., 1963b. On the determination of the minima of pseudo-Boolean functions. Studii si Cercetari Matematice 14, 359–364. In Romanian.

Hammer, P.L., Rudeanu, S., 1968. Boolean Methods in Operations Research and Related Areas. Springer-Verlag, Berlin.

Ishikawa, H., 2011. Transformation of general binary MRF minimization to the first-order case. IEEE Transactions on Pattern Analysis and Machine Intelligence 33, 1234–1249.

Khajavirad, A., 2023. On the strength of recursive McCormick relaxations for binary polynomial optimization. Operations Research Letters 51, 146–152.

Kolmogorov, V., Zabih, R., 2004. What energy functions can be minimized via graph cuts? IEEE Transactions on Pattern Analysis and Machine Intelligence 26, 147–159.

Liers, F., Marinari, E., Pagacz, U., Ricci-Tersenghi, F., Schmitz, V., 2010. A non-disordered glassy model with a tunable interaction range. Journal of Statistical Mechanics: Theory and Experiment 2010, L05003.

Mertens, S., 1996. Exhaustive search for low-autocorrelation binary sequences. Journal of Physics A: Mathematical and General 29, L473.

MINLPLib, 2020. A library of mixed-integer and continuous nonlinear programming instances. `http://www.minlplib.org/`.

Padberg, M., 1989. The Boolean quadric polytope: some characteristics, facets and relatives. Mathematical Programming 45, 139–172.

Papadimitriou, C.H., 1976. The NP-completeness of the bandwidth minimization problem. Computing 16, 263–270.

POLIP, 2020. Library for polynomially constrained mixed-integer programming. `http://polip.zib.de/polip.php`.

Rosenberg, I.G., 1975. Reduction of bivalent maximization to the quadratic case. Cahiers du Centre d'Études de Recherche Opérationnelle 17, 71–74.

Verma, A., Lewis, M., 2020. Optimal quadratic reformulations of fourth degree pseudo-Boolean functions. Optimization Letters 14, 1557–1569.

Watters, L.J., 1967. Reduction of integer polynomial programming problems to zero-one linear programming problems. Operations Research 15, 1171–1174.

Zangwill, W.I., 1965. Media selection by decision programming. Journal of Advertising Research 5, 30–36.

Appendix A. Computing $\psi_1(x_2, \ldots, x_n)$

The time consuming step in Algorithm 2 is constructing $\psi_1(x_2, \ldots, x_n)$ based on $g_1(x_2, \ldots, x_n)$ in line 7. One way of doing this is to tabulate all the possible input values and their corresponding output values as shown in Section 1. The corresponding pseudo-Boolean function can then be computed from this table. Hammer et al. (1963a,b), Hammer and Rudeanu (1968) describe an algebraic, but rather cumbersome method to obtain $\psi_1(x_2, \ldots, x_n)$ while Crama et al. (1990) propose a branch-and-bound algorithm for this step. We illustrate the branch-and-bound algorithm on the example $g_1(x_2, x_3) = 64x_2x_3 - 32x_2 - 32x_3 + 16$ from Section 4.1 and we refer to Crama et al. (1990) for more details. An important ingredient in the branch-and-bound algorithm is the ability to compute lower and upper bounds for a multilinear function. Crama et al. (1990) propose two simple methods. The first method is:

- **Lower bound:** add the constant term and all negative coefficients in the function. For g_1 we get $-32 - 32 + 16 = -48$.
- **Upper bound:** add the constant term and all positive coefficients in the function. For g_1 we get $64 + 16 = 80$.

The second method iterates through each variable in g_1 and computes the lower bound and upper bound found using the first method after fixing the current variable to zero or one. As an example, if we fix x_2 to zero g_1 reduces to $-32x_3 + 16$ and the lower bound increases to -16 while the upper bound decreases to 16. If we fix x_2 to one then g_1 reduces to $64x_3 - 32x_3 - 16 = 32x_3 - 16$ and the lower bound increases to -16 while the upper bound decreases to 16. Based on this analysis our lower bound for g_1 increases to -16 and the upper bound reduces to 16 since these bounds hold no matter what value x_2 takes. We can do a similar analysis for x_3, and the results are summarized in Tables A.4 and A.5. (Due to the symmetry between x_2 and x_3 in g_1 we get similar results for both variables). Taking the maximum (resp. minimum) of the values in the bottom row of Table A.4 (resp. Table A.5) gives the lower (resp. upper) bound of method 2.

LB after variable fixing		
	x_2	x_3
fix to 0	-16	-16
fix to 1	-16	-16
min	-16	-16

Table A.4: Improved lower bound

UB after variable fixing		
	x_2	x_3
fix to 0	16	16
fix to 1	16	16
max	16	16

Table A.5: Improved upper bound

If the upper bound is less than 0 then $\psi_1(x_2, \ldots, x_n) = g_1(x_2, \ldots, x_n)$, and if the lower bound is greater than or equal to 0 we have $\psi_1(x_2, \ldots, x_n) = 0$. In any other case we have to branch which is illustrated in Figure A.3 for the g_1 function of the example. We choose to branch on x_2 first. Fixing x_2 to 0 and 1 yields simpler pseudo-Boolean functions as shown in the figure. We can again compute lower and upper bounds using the two methods described above. Since the lower (resp. upper) bound is less than (resp. greater than) zero in both child nodes, it is necessary to branch again in both nodes. After branching on x_3 the resulting pseudo-Boolean function is a constant and each node can be fathomed. To construct the resulting function $\psi(x_2, x_3)$ we backtrack through the tree and construct the expression shown in Figure A.4. We get

$$\psi(x_2, x_3) = x_2\,(0x_3 + -16(1 - x_3)) + (1 - x_2)\,(-16x_3 + 0(1 - x_3))$$

which we simplify to

$$\psi(x_2, x_3) = -16x_2 - 16x_3 + 32x_2x_3$$

In our implementation, we used lower/upper methods 1 and 2 and a weaker version of method 2. The weaker version does not aggregate identical monomials after fixing a variable: in the example above, we saw that g_1 reduces to $64x_3 - 32x_3 - 16$ after fixing x_2 to one and from this expression we can compute the weaker lower bound -48. If we gather the x_3 terms the lower bound improves to -16 as we saw above. It turned out that using the weaker bounds resulted in the lowest overall computation time in the branch-and-bound algorithm (due to faster bound computations) despite exploring more branch-and-bound nodes. The presented computational results are, therefore, based on the weaker bound.

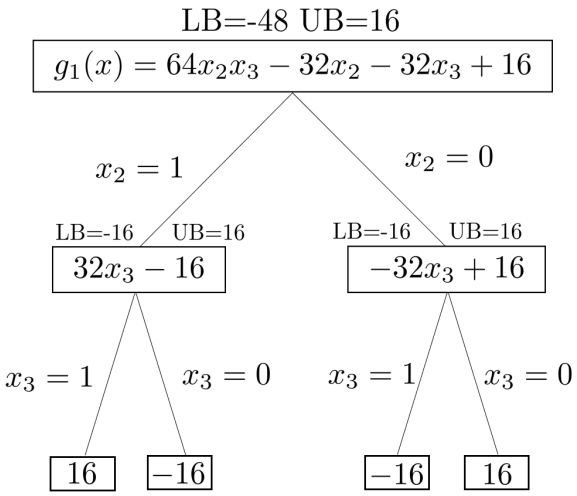

Figure A.3: Branch-and-bound tree

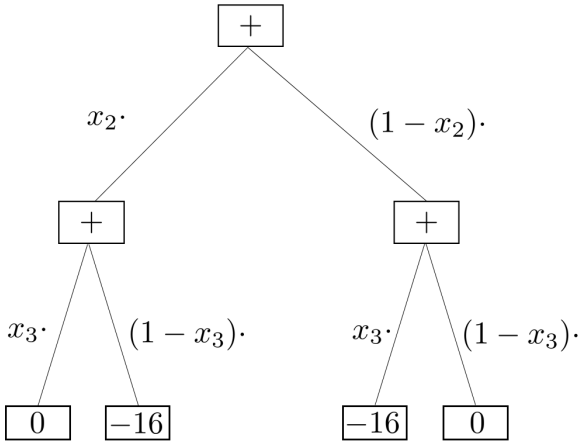

Figure A.4: Tree for constructing ψ