Encoding PB Constraints into SAT via Binary Adders and BDDs – Revisited

Neng-Fa Zhou[1] and Håkan Kjellerstrand[2]

[1] CUNY Brooklyn College & Graduate Center
[2] hakank.org

Abstract. Pseudo-Boolean constraints constitute an important class of constraints. Despite extensive studies of SAT encodings for PB constraints, there are no generally accepted SAT encodings for PB constraints. In this paper we revisit encoding PB constraints into SAT via binary adders and BDDs. For the binary adder encoding, we present an optimizing compiler that incorporates preprocessing, decomposition, constant propagation, and common subexpression elimination techniques tailored to PB constraints. For encoding via BDDs, we compare three methods for converting BDDs into SAT, namely, path encoding, 6-clause node encoding, and 2-clause node encoding. We experimentally compare these encodings on three sets of benchmarks. Our experiments revealed surprisingly good and consistent performance of the optimized adder encoder in comparison with other encoders.

1 Introduction

A Pseudo-Boolean (PB) constraint is a linear integer constraint where all the variables are Boolean (0 or 1). PB constraints constitute an important class of constraints. Many constraint models for combinatorial problems, such as model checking and planning, contain PB constraints. PB constraints also serve as an intermediate language for compiling higher-level constraints, such as global constraints [6]. PB constraints have been well studied in the SAT community. Several SAT encodings have been proposed, including BDDs [1, 5, 13], sorting networks [13], totalizers [4], log-encoded adders [21], and order-encoded adders [19]. There are also extensions of SAT solvers for natively supporting PB constraints [2, 9]. Specialized encoding algorithms have been proposed for encoding cardinality PB constraints [3, 10, 18]

Despite the extensity of the studies of SAT encodings for PB constraints, there are no generally accepted SAT encodings for PB constraints. Theoretical studies may not be able to provide a correct indication on the performance, and many empirical studies only used implementations that lack optimizations.

In this paper, we revisit the binary-adder and BDD encodings for PB constraints. We present a compiler that incorporates several optimizations in the translation of PB constraints into binary adders, including *preprocessing*, *decomposition*, *constant propagation*, and *common subexpression elimination*. We also empirically compare several encodings of BDDs into SAT for PB constraints, including *path encoding*, *6-clause node encoding*, and *2-clause node encoding*. All these encodings apply the Tseitin transformation [13] on BDDs, and guarantee the same order of code size as the BDDs.

We have implemented the adder encoder and the BDD encoders in the PicatSAT compiler [22], and have compared these encoders on three sets of benchmarks. While theoretical studies have ruled out the adder encoding as viable due to its incapability of maintaining GAC (Generalized Arc Consistency) on PB constraints, and past empirical studies have unanimously confirmed its poor performance [1, 13, 16], our experiments revealed surprisingly good and consistent performance of the optimized adder encoder in comparison with the BDD and other encoders.

2 PB Constraints and GAC

A PB constraint is a linear integer constraint that takes the form of $\Sigma_1^n(a_i \times X_i)\ \gamma\ b$, where a_i's and b are integers, X_i's are 0/1 integer domain variables, and γ is a relational operator in $\{=, \neq, >, \geq, <, \leq\}$. The constraint becomes a *cardinality* constraint when all the a_i's are the same.

A SAT encoder converts constraints into CNF clauses. An important question to ask about an encoder is whether unit propagation enforces GAC [12] on the generated code.

Definition 1. *GAC*
For an n-ary constraint $p(X_1, \ldots, X_n)$, a value v_i in X_i's domain is gac-supported if for each $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$ there exists a value v_j in X_j's domain such that $p(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n)$ is true. The constraint is said to be GAC if every value in every variable's domain is gac-supported. This condition can be given more formally as:

$$\forall_{i \in \{1..n\}} \forall_{v_i \in X_i} \exists_{v_1 \in X_1, \ldots, v_{i-1} \in X_{i-1}, v_{i+1} \in X_{i+1}, \ldots, v_n \in X_n} p(v_1, v_2, \ldots, v_n)$$

where variables are used to denote their domains.

In order to maintain GAC of a constraint, constraint propagation excludes unsupported values from the domains of the variables. A system of constraints becomes *inconsistent* if any variable's domain becomes empty.

Since it is expensive to maintain GAC for large constraints, many CP systems only maintain a weaker consistency, called *bounds-consistency*, which ensures that every bound value is supported. For a PB constraint, GAC and bounds-consistency are equivalent because all the variables are Boolean and all the values are bounds.

The GAC property is important because unit propagation forces values on variables and reduces the number of backtracks during search. Nevertheless, it can be expensive or may require prohibitively large code to enforce GAC. Modern SAT solvers are quite complicated, and an encoder should generate code that properly balances the code size and the propagation strength.

Consider, for example, the PB constraint:

$$P + 2 \times Q + 2 \times R + 3 \times S + 3 \times T \;=\; 5$$

A GAC encoder achieves the following, amongst others:

- $P = 1$ entails $Q = 1$, $R = 1$, $S = 0$, and $T = 0$.
- $S = 1$ and $T = 1$ entails inconsistency.

For this constraint, the following CNF code is returned by the logic optimizer Espresso [8]:

(1) $\neg Q \vee \neg R \vee \neg S$	(2) $\neg Q \vee \neg R \vee \neg T$	(3) $P \vee S \vee T$
(4) $\neg P \vee R$	(5) $\neg P \vee Q$	(6) $Q \vee R$
(7) $\neg S \vee \neg T$		

It is not difficult to check that, with this code, unit propagation maintains GAC on the constraint.

3 The Adder Encoding and its Optimizations

This section presents the adder encoding for PB constraints and its optimizations. PB constraints are special arithmetic constraints, and the techniques can be viewed as specializations of the techniques used for general arithmetic constraints.

3.1 Log Encoding

Our adder encoder adopts the sign-and-magnitude log encoding for domain variables. For a domain variable X, let m be the maximum absolute value in X's domain. A vector of Boolean variables, called *bits*, $<X_{n-1}X_{n-2}\ldots X_1X_0>$ is utilized to represent X's magnitude, where $n = \lceil log_2(m) \rceil$. If the domain contains both negative and positive values, then another Boolean variable is employed to represent the sign. Each combination of values of the bits represents a valuation for the variable: $X_{n-1} \times 2^{n-1} + X_{n-2} \times 2^{n-2} + \ldots + X_1 \times 2 + X_0$. For PB constraints, our adder encoder never introduces negative-domain auxiliary variables.

For an integer-domain variable, some of the bits in its log encoding may be inferred from the values in the domain. For example, consider the constraint $Y = 10 \times X$, where X is Boolean and Y has the domain $\{0, 10\}$. Y's log encoding is $<Y_3Y_2Y_1Y_0>$. Our adder encoder infers $Y_2 = 0$ and $Y_0 = 0$ from the two values in the domain, and only uses two Boolean variables to encode Y, and uses the following two clauses to encode the constraint: $\neg Y_1 \vee Y_3$ and $Y_1 \vee \neg Y_3$.

3.2 Special PB Constraints

Our adder encoder treats a PB constraint as a special one if it is either a *small* PB constraint that contains no more than 6 variables or a *cardinality* constraint of the form $\Sigma_1^n X_i \ \gamma \ B$, where $n > 6$, and `abs(`B`)` is 0, 1, or 2. For small PB constraints, our adder encoder uses Espresso to find optimal or near optimal code. For cardinality constraints, our adder encoder employs specialized encoders, such as the two-product algorithm [10] and the sequential counter algorithm [18], depending on the cardinalities.

3.3 Primitive Arithmetic Constraints

Our adder encoder breaks down a non-special PB constraint into the following types of primitive constraints: $X + Y = Z$, and $X \times Y = Z$, and $X \ \gamma \ Y$, where X, Y, Z are integer variables or integers. These primitive constraints are further converted to binary adders and comparators, using *ripple carry adders* for $X + Y = Z$, the *shift-and-add* algorithm for $X \times Y = Z$, and recursive algorithms for $X \gamma Y$ [22]. Our adder encoder makes use of Espresso to find codes for basic adders and comparators. For a full adder, our encoder uses 10 clauses; for a half adder, it uses 7 clauses.

Constants in primitive constraints are exploited, through *constant propagation* [22], to infer the values of bits and the equivalence relationships between some bit pairs. For example, for the constraint $X + 2 = Y$, our encoder infers $Y_0 = X_0$ and $Y_1 = \neg X_1$, and for the constraint $2 \times X = Y$, it infers $Y_0 = 0$ and $Y_{i+1} = X_i$ for $i > 0$.

3.4 Breaking Large PB Constraints

There are many different ways to break a PB constraint into primitive constraints, and the decision on which algorithm to use has great impact on the quality of the generated code. Our adder encoder follows the following steps to break PB constraints:

1. **Combine power-of-2 terms**: For each subexpression of the form:

   $$2^{k-1} \times X_{k-1} + \ldots + 2^0 \times X_0$$

   our adder encoder replaces it with an auxiliary variable X, which has the domain $0..2^k - 1$. This transformation introduces no new Boolean variables, because X's log encoding only reuses existing Boolean variables.
2. **Factor out terms with common coefficients**: For each group of terms that have the same non-unit coefficient $\{a \times Y_1, \ldots, a \times Y_k\}$ ($a \neq 1$ and $a \neq -1$), our adder encoder introduces an auxiliary variable V for the sum of the variables $V = Y_1 + \ldots + Y_k$, and introduces another auxiliary variable U for $U = a \times V$. The domains of U and V are computed based on the coefficient a, the variables Y_i's, as well as the original constraint such that the resulting constraints are all bounds-consistent.
3. **Break the constraint**: After the above two steps, all the terms only have unit coefficients. In this step, our adder encoder follows the algorithm in Figure 1, which is similar to the Huffman coding algorithm [11], to break the constraint until it becomes primitive.

```
decompose(a_1 × X_1 + a_2 × X_2 + ... + a_n × X_n γ b):
    add all the terms a_i × X_i into a priority queue Q
    while the constraint is not primitive:
        remove two terms a_i × X_i and a_j × X_j from Q
            where a_i = a_j, and X_i and X_j have the smallest domains
        post T = X_i + X_j
        add the term a_i × T into Q
    post the primitive constraint
```

Fig. 1. Breaking unit-coefficient PB constraints

When posting a primitive constraint in Step 2 and Step 3, our adder encoder looks up the constraint store to see if an identical constraint has been posted. If so, it reuses the auxiliary variable, rather than introducing a new one. This technique eliminates *common subexpressions* in constraints, and can significantly reduce the code size. Our adder encoder also ensures that all the constraints posted to the constraint store are bounds-consistent. This preprocessing narrows the domains of variables before the constraints are converted to CNF.

Consider, for example, the PB constraint:

$$P + 2 \times Q + 2 \times R + 3 \times S + 3 \times T \;=\; 5$$

Assume that small PB constraints are not treated by a logic optimizer, then our adder encoder breaks the above constraint into the following triplets:

$$\begin{array}{lll} U \;=\; P + 2 \times Q & & U \;\in\; 0..3 \\ V \;=\; 2 \times R & & V \;\in\; 0..2 \\ W \;=\; S + T & & W \;\in\; 0..1 \\ X \;=\; 3 \times W & & X \;\in\; 0..3 \\ Y \;=\; U + V & & Y \;\in\; 0..5 \\ X + Y \;=\; 5 & & \end{array}$$

Variable U combines the two terms with power-of-2 coefficients: P and $2 \times Q$. Variable U is encoded as $<QP>$, which requires no new Boolean variables. Note that all the constraints are made bounds-consistent, and unsupported bound values are removed from the domains. For example, W's domain is narrowed from 0..2 to 0..1 after value 2 is found to be unsupported. Also note that not all the constraints are GAC after preprocessing. For example, value 2 in U's domain and value 2 in X's domain are not gac-supported, but they remain because they don't violate bounds consistency.

Constant propagation enables reuse of bits in the encodings of the variables. Let V's encoding be $<V_1V_0>$, X's encoding be $<X_1X_0>$, and Y's encoding be $<Y_2Y_1Y0>$. Our adder encoder infers the following:

$$\begin{array}{ll} V_0 = 0, V_1 = R & \text{from } V \;=\; 2 \times R \\ X_0 = W, X_1 = W & \text{from } X \;=\; 3 \times W \\ Y_0 = \neg X_0 & \text{from } X + Y \;=\; 5 \end{array}$$

In total, only three auxiliary Boolean variables are introduced for encoding the new variables.

With the code generated by our adder encoder for PB constraints, unit propagation generally is not able to guarantee GAC on the constraints. For example, for the above PB constraint, when $P = 1$, unit propagation is not able to force $Q = 1$, $R = 1$, $S = 0$, and $T = 0$. However, it is able to detect inconsistency when $S = 1$ and $T = 1$.

4 Encoding BDDs into SAT

BDDs (Binary Decision Diagrams) have been used as an intermediate form for translating PB constraints into SAT [1, 13, 16]. A node N in a BDD represents a constraint. Each node has a chosen variable, denoted as $N.v$, and has two children: the left child (called *0-child*, denoted as $N.0$) represents the resulting constraint after $N.v$ is assigned 0, and the right child (called *1-child*, denoted as $N.1$) represents the resulting constraint after $N.v$ is assigned 1. A true constraint is represented as a terminal $\top$, and a false constraint is represented as a terminal $\bot$. We assume that BDDs are always *ordered*, meaning that all the nodes on the same layer have the same chosen variable, and *reduced*, meaning that all the nodes that represent the same constraint are merged into one node. We also assume that the constraint represented by every node is GAC.

A BDD-based encoder takes two steps to translate a PB constraint into SAT: it builds a BDD from the constraint, and then traverses the BDD to generate CNF clauses. There are different ways to build a BDD from a constraint, based on orderings of variables. A reasonable choice, which is adopted in our BDD encoders, is to order terms on coefficients, from the largest absolute coefficient to the smallest absolute coefficient [13]. Figure 2 shows the first two layers of the BDD for the PB constraint:

$$3 \times S + 3 \times T + 2 \times Q + 2 \times R + P \;=\; 5$$

The right child of node 3 is $\bot$, because the path to the child ($S = 1$ and $T = 1$) causes inconsistency once the constraint is made GAC. Node 4 represents the constraint:

$$2 \times Q + 2 \times R + P \;=\; 5$$

which entails $Q = 1$, $R = 1$, and $P = 1$ once the constraint is made GAC. Under node 4, there is a one-way path to a $\top$ terminal. Node 5 is shared by two paths from the root to it.

There are also different ways to encode a BDD into CNF clauses. The Tseitin transformation given in [13] introduces an auxiliary Boolean variable for each node, which is true if and only if there is a path from the node to a $\top$ terminal. A unit clause is generated for the root that forces the root's auxiliary variable to be true.

Let the auxiliary variable for a node be r, the chosen variable of the node be x, the auxiliary variable for the 0-child be c_0, and the auxiliary variable for the

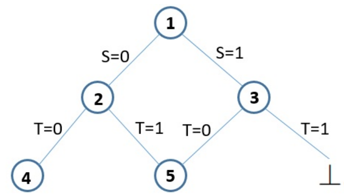

Fig. 2. The first two layers of the BDD for $3 \times S + 3 \times T + 2 \times Q + 2 \times R + P \;=\; 5$

1-child be c_1. The Tseitin transformation in [13] uses the following six clauses to connect the node r and its children:

(1) $x \wedge c_1 \rightarrow r$	(2) $\neg x \wedge c_0 \rightarrow r$	(3) $x \wedge \neg c_1 \rightarrow \neg r$
(4) $\neg x \wedge \neg c_0 \rightarrow \neg r$	(5) $c_0 \wedge c_1 \rightarrow r$	(6) $\neg c_0 \wedge \neg c_1 \rightarrow \neg r$

Clauses (5) and (6) are redundant, and are added to increase the propagation strength. This encoding, called *6-clause node encoding* in this paper, enforces GAC with unit propagation [13].

The *2-clause node encoding* only uses clauses (1) and (2) above. It still maintains GAC if the constraint of every node is GAC. It is shown in [1] that, for *monotonic* constraints, clause (2) can be simplified to $c_0 \rightarrow r$.

Another encoding, called *path encoding*, is to generate clauses to ban *no-good* paths that lead to $\perp$ terminals. This is similar to *direct encoding* [7, 20] of constraints. In order to guarantee the same size order of the generated code as the BDD, this encoding introduces an auxiliary variable, denoted as $N.a$, for each node N that is shared by two or more paths from the root. The following gives an algorithm for generating clauses from a BDD:

```
gen(N, S):
    if N = ⊥:
        emit S
    if N = ⊤:
        return
    if N is a shared node:
        emit S ∪ {N.a}
        S = {¬N.a}
    gen(N.0, {N.v} ∪ S)
    gen(N.1, {¬N.v} ∪ S)
```

The algorithm `gen` takes a BDD node N, and a set of literals S. In the beginning, N is the root of a BDD, and S is empty. If $N \;=\; \perp$, then the algorithm emits the literals in S as a clause, which bans the path. If $N \;=\; \top$, the algorithm does nothing. If N is a shared node, the algorithm emits a clause $S \cup \{N.a\}$, which means that the path to the node entails $N.a$, and resets S to $\{\neg N.a\}$. The algorithm recurses on the children as follows: when going down to $N.0$, it adds $N.v$ to S; when going down to $N.1$, it adds $\neg N.v$ to S.

Table 1. A comparison on code size (PB'16 benchmarks, unit: 1000)

Benchmark	Adder		BDD_p		BDD_{n2}		BDD_{n6}		PBSugar		PBLib	
	vars	cls	vars	cls	vars	cls	vars	cls	vars	cls	vars	cls
sha1-128-21-4	**8**	42	15	**37**	23	46	23	108	51	223	34	93
sh-80-21-1	**8**	43	15	**38**	23	46	23	109	51	223	34	93
sh-80-21-2	**8**	43	15	**38**	23	46	23	109	51	223	34	93
sh-80-21-4	**9**	43	15	**38**	23	46	23	109	51	223	34	93
sh-80-21-5	**8**	43	15	**38**	23	46	23	108	51	223	34	93
sh-80-21-6	**8**	43	15	**38**	23	46	23	109	51	223	34	93
sh-80-21-7	**8**	43	15	**38**	23	46	23	109	51	223	34	93
sh-80-21-9	**8**	43	15	**38**	23	46	23	108	51	223	34	93
sh-96-21-6	**8**	43	15	**38**	23	46	23	109	51	223	34	93
sh-96-21-7	**8**	43	15	**37**	23	46	23	108	51	223	34	93
su3hP128	**230**	852	263	**525**	443	705	443	1983	836	3572	443	705
su3pyP0125	**111**	411	126	**252**	213	339	213	953	402	1717	214	340
su4hP064	**58**	318	127	**254**	189	316	189	905	353	1563	189	316
su4hP128	**231**	1266	508	**1016**	754	1262	754	3621	1411	6247	755	1263
su4pyP0064	**30**	162	65	**129**	96	160	96	460	180	795	96	161
su4pyP0125	**112**	610	245	**489**	363	607	363	1741	679	3005	364	608
su5hP032	**32**	146	52	**105**	72	124	72	359	134	604	72	124
su5hP064	**127**	586	209	**418**	287	496	287	1438	533	2415	288	497
su5pyP0032	**17**	76	27	**54**	37	64	37	186	69	313	37	64
su5pyP0064	**65**	299	106	**213**	146	252	146	731	272	1228	147	253

For example, for the partial BDD shown in Figure 2, the algorithm generates $\neg S \vee \neg T$ for the path from the root to the right-most terminal $\perp$, and two clauses for the paths to node 5: $S \vee \neg T \vee A_5$ and $\neg S \vee T \vee A_5$, assuming the auxiliary variable introduced for node 5 is A_5.

The path encoding is *correct* in the sense that the generated code is satisfiable if and only if there is a path from the root to a $\top$ terminal. The auxiliary variables introduced for shared nodes do not affect the correctness because $\alpha \rightarrow \beta$ is equivalent to $\alpha \rightarrow N.a \wedge N.a \rightarrow \beta$ for any formulas α and β. The generated code also maintains GAC via unit propagation if the constraint of every node in the BDD is GAC.

5 Experimental Results

We implemented the binary adder encoder and the three BDD encoders for PB constraints in the PicatSAT compiler,[3] and empirically evaluated them using three sets of benchmarks: a selection of instances from PB competition 2016[4], a set of Integer Programming (IP) benchmarks taken from [15], and a set of cumulative scheduling benchmarks used in the MiniZinc Challenge[5]. The benchmarks are available at http://picat-lang.org/download/pb_bench.tar.gz. We also included PBSugar (version 1.1.1) [19] and PBLib [16],[6] two cutting-edge PB encoders, in the comparison on the PB'16 benchmarks, and Chuffed[7], a cutting-

[3] http://picat-lang.org/

[4] http://www.cril.univ-artois.fr/PB16/

[5] http://www.minizinc.org/challenge.html

[6] The default settings of PBSugar and PBLIb were used in the comparison.

[7] https://github.com/chuffed/chuffed

Table 2. A comparison on solving time (PB'16 benchmarks, seconds)

Benchmark	Adder		BDD_p		BDD_{n2}		BDD_{n6}		PBSugar		PBLib	
	lgl	glu	lgl	glu	lgl	glu	lgl	glu	lgl	glu	lgl	glu
sh-128-21-4	55.77	28.46	104.35	>1200	67.04	40.27	54.69	>1200	**7.97**	>1200	32.45	>1200
sh-80-21-1	17.57	5.30	8.65	5.40	9.81	5.99	10.71	13.42	5.91	**2.80**	15.03	16.88
sh-80-21-2	27.26	84.67	31.74	32.38	59.10	19.49	21.62	618.99	**4.65**	650.81	17.25	990.43
sh-80-21-4	21.50	4.81	4.57	7.15	13.60	4.76	18.83	40.51	**3.50**	66.56	10.75	21.78
sh-80-21-5	87.14	62.25	33.64	750.32	32.33	32.21	12.85	64.61	**6.04**	33.60	36.52	110.15
sh-80-21-6	21.10	9.04	10.56	5.95	9.69	5.09	18.12	15.58	**3.85**	5.64	20.56	21.36
sh-80-21-7	17.57	7.76	17.85	10.17	9.57	9.76	19.14	199.12	**3.63**	51.11	9.70	99.45
sh-80-21-9	28.21	12.35	21.85	487.37	22.01	52.84	**10.32**	65.10	15.51	253.62	13.85	447.79
sh-96-21-6	30.27	16.27	**8.69**	512.42	21.07	51.98	15.85	56.43	12.36	155.75	18.86	54.66
sh-96-21-7	17.62	8.88	20.61	574.24	23.11	7.53	16.97	273.27	**5.13**	689.63	18.83	130.35
su3hP128	55.17	26.19	35.69	**1.23**	9.80	6.92	16.00	9.49	20.86	18.40	7.89	1.59
su3pyP0125	20.12	5.23	3.59	**0.57**	4.20	0.85	8.25	3.58	13.46	4.67	3.70	0.71
su4hP064	3.46	1.52	5.40	2.08	5.68	3.18	9.58	5.88	29.20	5.55	4.35	**0.97**
su4hP128	14.82	10.03	396.68	146.55	63.64	7.40	72.68	91.22	254.33	74.41	21.80	**4.15**
su4pyP0064	1.84	**0.26**	1.93	0.40	2.52	1.47	4.33	1.82	4.00	2.26	1.96	0.45
su4pyP0125	6.08	**1.65**	18.26	6.51	7.22	15.99	60.46	19.95	69.13	10.40	6.50	1.95
su5hP032	10.06	3.08	3.25	1.71	5.01	2.55	5.47	2.38	11.66	2.19	3.33	**0.73**
su5hP064	205.66	49.01	59.94	55.98	67.85	53.23	66.43	8.01	87.66	12.39	23.59	**5.61**
su5pyP0032	2.38	0.82	1.47	0.64	2.23	0.61	3.43	0.45	4.20	0.85	1.45	**0.32**
su5pyP0064	25.23	15.30	9.56	8.35	13.30	8.61	11.23	2.96	30.38	3.56	16.05	**1.95**

edge solver that integrates SAT and CP solving techniques, in the comparison on cumulative scheduling. We did the experiment on Linux Ubuntu with an Intel i7 3.30GHz CPU and 32GB RAM, and used the SAT solvers Glucose (version 4.1)[8] and Lingeling (version 587f)[9] in the experiments. The time limit was 20 minutes per instance.

5.1 PB'16 Benchmarks

Most of the instances used in the DEC-SMALLINT-LIN category in PB'16 only involve small PB constraints that have no more than 6 variables or cardinality constraints. For small PB constraints, all of our encoders, including BDD encoders, use Espresso to find optimal or near optimal codes, and for cardinality constraints, our encoders use specialized algorithms. Only two benchmarks, namely sha and sumineq, contain non-special PB constraints. We selected 10 instances from each of these two benchmarks.

Table 1 gives the number of variables (vars) and the number of clauses (cls), both in thousands, of the CNF code generated by each of the encoders. The column Adder is for the adder encoder, BDD_p for path encoding, BDD_{n2} for 2-clause node encoding, BDD_{n6} is for 6-clause node encoding. Adder has the fewest variables, while BDD_p has the fewest clauses. PBSugar generates the largest code.

Table 2 gives the solving time, in seconds, of the CNF code solved using Lingeling (lgl) and Glucose (glu). Adder is competitive with other encoders; it had no timeouts, and only had one instance (su5hP064) that took more than 100s. Among the BDD encoders, BDD_{n2} performed the best; it had no timeouts,

[8] http://www.labri.fr/perso/lsimon/glucose/
[9] http://fmv.jku.at/lingeling/

Table 3. A comparison on code size (IP benchmarks, unit:1000)

Benchmark	Adder		BDD_p		BDD_{n2}		BDD_{n6}	
	vars	cls	vars	cls	vars	cls	vars	cls
maxclosed_ineq_10_100_10	**3**	**21**	77	165	104	192	104	571
maxclosed_ineq_10_100_100	**27**	**181**	691	1482	934	1725	934	5116
maxclosed_ineq_10_200_10	**3**	**18**	74	159	99	184	99	543
maxclosed_ineq_20_100_1000	**476**	**3404**	26808	55365	31752	60308	31752	180622
maxclosed_ineq_30_200_1000	**753**	**5495**	77087	157276	86182	166371	86182	498899

Table 4. A comparison on compile time (IP benchmarks, seconds)

Benchmark	Adder	BDD_p	BDD_{n2}	BDD_{n6}
maxclosed_ineq_10_100_10	**0.04**	0.23	0.23	0.46
maxclosed_ineq_10_100_100	**0.31**	4.38	4.44	6.39
maxclosed_ineq_10_200_10	**0.03**	0.19	0.20	0.42
maxclosed_ineq_20_100_1000	**7.23**	391.29	389.56	456.52
maxclosed_ineq_30_200_1000	**9.64**	1811.72	1805.27	1991.46

and had no instance that took more than 100s. In terms of number of wins, PBSugar did the best on sh with Lingeling, and PBLib did the best on sumineq with Glucose.

For each of the PB'16 instances, the compile time is negligible in comparison with the solving time.

5.2 IP Benchmarks

An integer-domain variable can be Booleanized using the log-encoding, and an IP constraint can easily be converted to a PB constraint. Let X be an integer domain variable, and $<X_{n-1}X_{n-2}\dots X_1X_0>$ be X's log encoding. We can replace X by X's log-encoded value $X_{n-1}2^{n-1} + X_{n-2}2^{n-2} + \dots + X_1 2 + X_0$. In this way, only Boolean variables remain, and linear constraints become PB constraints.

Tables 3, 4, and 5 give the results on a set IP benchmarks taken from [15]. PBSugar and PBLib failed to compile all of the instances, probably due to the large sizes of the instances. Adder generates the most compact code. The code size also reflects the compile time. For example, it took Adder 10s to compile maxclosed_ineq_30_200_1000, while it took BDD_{n6} 1991s to compile the instance. Adder is also the fastest in terms of solving time.

Table 5. A comparison on solving time (IP benchmarks, seconds)

Benchmark	Adder		BDD_p		BDD_{n2}		BDD_{n6}	
	lgl	glu	lgl	glu	lgl	glu	lgl	glu
maxclosed_ineq_10_100_10	0.32	**0.08**	1.18	0.50	1.07	0.47	2.83	0.59
maxclosed_ineq_10_100_100	1.26	**0.28**	7.08	19.81	7.66	14.56	16.80	2.28
maxclosed_ineq_10_200_10	0.28	**0.03**	1.08	0.47	1.15	0.49	2.62	0.68
maxclosed_ineq_20_100_1000	29.53	**15.08**	617.31	35.84	498.26	30.42	>1200	100.57
maxclosed_ineq_30_200_1000	61.42	**21.40**	>1200	125.06	>1200	94.86	>1200	>1200

Table 6. A comparison on cumulative scheduling benchmarks

Benchmark	Adder		BDD_p		BDD_{n2}		BDD_{n6}		Chuffed	
	solved	psolved	solved	psolved	solved	psolved	solved	psolved	solved	psolved
cargo_challenge(5)	3	3	2	3	2	3	2	3	0	5
carpet-cutting(5)	1	5	0	5	0	5	0	5	2	5
cyclic-rcpsp(5)	3	5	4	5	4	5	4	5	3	5
mspsp(6)	6	6	6	6	6	6	6	6	6	6
rcpsp-wet(5)	3	4	3	4	3	4	3	4	4	5
rcpsp(5)	4	5	4	5	4	5	4	5	2	5
smelt(5)	4	5	4	5	4	5	4	5	4	5
total (36)	**24**	33	23	33	23	33	23	33	21	**36**

5.3 Cumulative Scheduling Benchmarks

The `cumulative` constraint is one of the most important global constraints [6]. It is well used in resource-constrained real-world scheduling problems. Given a set of tasks, each of which has a feasible starting time, a duration, and an amount of resources needed for its running, the `cumulative` constraint ensures that the total resource consumption at any time is within a given limit. The `cumulative` constraint can be decomposed into *occupation* and *resource* constraints. An occupation constraint tells if a task *occupies* a time point P, meaning that it starts at or before P, and ends after P. A resource constraint, which is a PB constraint, for a time point P ensures that the total amount of resources consumed by the running tasks at P does not exceed the limit. In this experiment, we used *task decomposition* [17], which enforces the resource constraint at the starting time of each task.

Table 6 gives the results on a set of benchmarks used in the MiniZinc Challenge. The SAT codes were solved using Lingeling. All the benchmarks are optimization problems. For each benchmark, the number in the parentheses indicates the total number of instances. The column, solved, indicates the number of completely solved instances. An instance is considered solved if an optimal solution was given and its optimality was proven. The column, psolved, indicates the number of partially solved instances, i.e., instances for which a non-optimal solution was displayed or an optimal solution was given but its optimality was not proven.

Once again, the experiment showed the competitiveness of Adder, which had 24 of the 36 instances solved, and 33 psolved. The BDD encoders had the same performance, in terms of solved (23) and psolved (33). Chuffed, which outperformed all the official winners in the MiniZinc Challenge, returned a partial solution for each of the instances, but only solved 21 instances completely.

6 Related Work

Our binary adder encoder mimics how basic arithmetic operations are performed on binary numbers by the computer. The way our adder encoder breaks large PB constraints into primitive ones is similar to the way language compilers break large expressions into triplets. Constant propagation is proposed in [22] to reduce code sizes of primitive arithmetic constraints that involve constants. This paper introduces new techniques for encoding PB constraints. The technique that combines terms with power-of-two coefficients is especially effective for the IP benchmarks. The Huffman coding algorithm for breaking large PB constraints is effective for avoiding creating auxiliary variables with large domains.

Our adder encoder differs from the adder encoding proposed in [13], which adds bits bucket by bucket. For a PB constraint $\Sigma_1^n(a_i \times X_i)\ \gamma\ b$, the bucket adder encoder first distributes each variable X_i into buckets based on the binary representation of a_i. For example, for the term $5 \times X$, it puts X into the position-0 bucket and the position-2 bucket. It then sums up the buckets from the lowest position to the highest one, and ensures that the total satisfies the constraint. The bucket adder encoder performs no consistency checking, constant propagation, or subexpression elimination. The bucket adder encoding has been evaluated in multiple experiments [1, 13, 16]; all of them confirmed compactness but poor performance of the encoding.

PBSugar [19] decomposes PB constraints into primitive constraints of the form $S_{i+1} = S_i + a_i \times X_i$, and uses order-encoded adders for them. An improvement implemented in PBSugar uses a counter matrix, which facilitates both inter-constraint and intra-constraint sharing of common primitive constraints. This improvement is closely related to the BDD encoding proposed in [4]. It also generalizes the counter encoding for cardinality constraints [18]. Like sorting-network encoding [13], which uses unary adders, and BDD encoding, order encoding also suffers from code explosion.

There are different ways to convert a PB constraint into a BDD, and there are also different ways to encode a BDD into SAT. The ordering that favors terms with the largest coefficients is considered reasonable [13]. The 6-clause node encoding is used in [13], which includes two redundant clauses for increasing propagation strength. The path encoding is not well studied. It introduces fewer variables but generates longer clauses than node encodings. Modern SAT solvers all incorporate a technique called, *watched literals* [14], which make lengths of clauses a less important factor. All the BDD encodings achieve GAC. For monotonic constraints, GAC can be achieved by an encoding that only requires a binary clause and a ternary clause for each BDD node [1].

7 Conclusion

In this paper we have reviewed the adder and BDD encodings for PB constraints. For the adder encoding, we presented several optimizations, and for the BDD encoding, we compared three methods to encode BDDs into SAT. Our experimental results show that the optimized adder encoder not only generates compact code but is also generally competitive in runtime.

Past theoretical and empirical studies have unanimously confirmed the poor performance of encoding PB constraints via adder networks. Our adder encoder is different from the bucket-adder encoder studied in the past. Our adder encoder uses optimized adders, and incorporates several optimizations in compilation, including preprocessing constraints to achieve bounds consistency, propagating constants in constraints to infer bit values and their equivalence relationships, using specialized encoders for small and special PB constraints, decomposing large PB constraints using the Huffman coding algorithm in order to avoid creating large-domain variables, and eliminating common subexpressions to avoid duplicating primitive constraints. Our adder encoder is arguably more difficult to implement than the bucket-adder encoder. However, the payoff is great.

Future work includes investigating more optimizations for the adder encoder, and tailoring SAT solvers to the adder encoder.

Acknowledgement

This project was supported in part by the NSF under grant number CCF1618046.

References

1. Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at BDDs for pseudo-Boolean constraints. *J. Artif. Intell. Res.*, 45:443–480, 2012.
2. Fadi A. Aloul, Arathi Ramani, Igot L. Markov, and Karem A. Sakallah. Generic ILP versus specialized 0-1 ILP: An update. In *International Conference on Computer-Aided Design*, pages 450–457, 2002.
3. Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming*, pages 108–122, 2003.
4. Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-Boolean constraints into CNF. In *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 181–194, 2009.
5. Constantinos Bartzis and Tevfik Bultan. Efficient BDDs for bounded arithmetic constraints. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(1):26–36, 2006.
6. Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog. Technical report. http://sofdem.github.io/gccat/gccat/.
7. Hachemi Bennaceur. The satisfiability problem regarded as a constraint satisfaction problem. In *ECAI*, pages 155–159, 1996.
8. Robert King Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.
9. Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(3):305–317, 2005.
10. Jingchao Chen. A new SAT encoding of the at-most-one constraint. In *Proc. of the 9th Int. Workshop of Constraint Modeling and Reformulation*, 2010.
11. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.
12. Rina Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, 2003.
13. Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

14. Matthew W. Moskewicz, Conor F. Madigan, Zhao Zhao, Lintao Zhang, and Sharad Malik. Engineering a (super?) efficient SAT solver. In *Proceedings of the 2001 Design Automation Conference (DAC-01)*, pages 530–535. ACM Press, 2001.
15. Justyna Petke. *Bridging Constraint Satisfaction and Boolean Satisfiability*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2015.
16. Tobias Philipp and Peter Steinke. PBLib - A library for encoding pseudo-Boolean constraints into CNF. In *SAT*, pages 9–16, 2015.
17. Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, 2011.
18. Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *CP*, pages 827–831, 2005.
19. Naoyuki Tamura, Mutsunori Banbara, and Takehide Soh. Compiling pseudo-Boolean constraints to SAT with order encoding. In *2013 IEEE ICTAI*, pages 1020–1027, 2013.
20. Toby Walsh. SAT v CSP. In *CP*, pages 441–456, 2000.
21. Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Inf. Process. Lett.*, 68(2):63–69, 1998.
22. Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *CP*, pages 671–686, 2017.