Dynamic Programming and Tabled Logic Programming for Encoding Single-Constant Multiplication into SAT (Declarative Pearls)

Neng-Fa Zhou[1][0000-0003-2507-7031], Chufeng Jiang[1][0009-0000-2810-5013], Hendrik Bierlee[3][0000-0001-6766-5435], and Peter Stuckey[2][0000-0003-2186-0459]

[1] City University of New York, {nzhou@brooklyn, cjiang@gradcenter}.cuny.edu
[2] Monash University, peter.stuckey@monash.edu
[3] Department of Computer Science, KU Leuven, Belgium
henk.bierlee@kuleuven.be

Abstract. Encoding single-constant multiplication (SCM) constraints into SAT is essential for solving a wide range of combinatorial problems involving integer arithmetic. Although optimal SAT encodings for SCM are known, they are often prohibitively expensive to generate for large constants. This paper introduces a dynamic programming (DP) approach that offers a practical balance between encoding quality and encoding time. The proposed method recursively decomposes the binary representation of the target constant into chunks and eliminates common subexpressions across the chunks and their one's complements. The objective is either to minimize the number of additions (the *min-k* objective) or to minimize the number of half/full adders (the *min-a* objective). The DP algorithm is naturally expressed and efficiently implemented using mode-directed tabling in Picat. Although the approach does not guarantee optimality, experimental results show that it produces high-quality encodings while substantially reducing encoding time. This makes SAT-based techniques considerably more practical for applications involving SCM constraints. The full implementation is publicly available at: https://github.com/nfzhou/Picat/blob/master/scm.pi.

Keywords: SCM · SAT · Dynamic programming · Declarative programming · Tabled Logic Programming

1 Introduction

The performance of Boolean Satisfiability (SAT) solvers has improved dramatically over the past two decades, making it feasible to tackle many combinatorial optimization problems by encoding them into SAT instances and solving them using SAT solvers. Nevertheless, for many integer programming problems, SAT-based solvers still struggle due to the lack of optimizations in the encodings for integer-domain constraints.

One fundamental integer-domain constraint is single-constant multiplication (SCM), of the form $c \cdot x = y$, where c is a fixed constant, and x and y are integer-domain variables. In SAT, SCM constraints are typically encoded using multiple additions, subtractions, and bit-shifts (i.e. multiplications by powers of two). Since subtractions can always be represented as additions, and bit-shifts and additions with zero bits are free in SAT, the encoding quality is primarily determined by minimizing the number of additions (the *min-k* objective) and the number of half/full adders (the *min-a* objective). The former objective only depends on c, while the latter also depends on the bit-width of x.

The SCM problem is well studied in the context of hardware design [4]. It is NP-hard to minimize the number of additions/subtractions for SCM [3]. Previous approaches to SCM are either based on graph methods [8], Common Subexpression Elimination (CSE) [14], Integer Linear Programming (ILP) [10], SAT-based solvers [11, 13], as well as various heuristic methods [12, 9]. The *min-a* objective was first covered by [5]. For the Multiple Constant Multiplication (MCM) problem, the generalization of SCM, where a set of constants should be covered by the circuit, both optimal and heuristic methods have been developed [1].

Unlike in hardware design, where the time required to find an SCM solution does not affect the performance of the resulting circuit, in SAT solving, the encoding time directly contributes to the total solving time. Although precomputed databases of optimal SCM encodings for relevant coefficients and bit-widths can reduce this cost [2], such databases cannot feasibly cover the full range of coefficients and bit-widths required in practice. As a result, for any coefficient and bit-width combination outside of the database, a naive fallback is used to encode the multiplication.

In this paper, we propose a dynamic programming (DP) approach to SCM encoding that offers a practical balance between encoding quality and generation time. The proposed method recursively decomposes the binary representation of the target constant into chunks and eliminates common subexpressions across the chunks and their one's complements. The method fits within the general framework of decomposing a given integer c using one of three operator types, namely, $(c_l \ll s) + c_r$ (SPLUS), $(c_l \ll s) - c_r$ (SMINUS), or $c_l - (sc_r \ll s)$ (MINUSS) [2], where $\ll s$ denotes left shift of s bits or equivalently multiplying by 2^s. Because the decomposition is guided by the binary representation of c, the search space for c_l, c_r, and s is considerably reduced.

We implemented the proposed DP algorithm using tabling in Picat [18, 20]. The implementation models SCM as a planning problem, where a solution is represented as a set of operations. For each decomposition step, it takes the union of the sub-plans generated for the chunks, thereby eliminating duplicate operations. The use of mode-directed tabling enables concise modeling of both the *min-k* and *min-a* objectives. The entire program is under 100 lines and serves as an excellent example of declarative programming and illustrating its expressiveness for arithmetic encoding problems.

Although the program does not guarantee optimality, it is designed to strike a favorable trade-off between encoding quality and encoding time, and the ex-

perimental results demonstrate strong practical performance. For the *min-k* objective, the *DPmin-k* encoder solves all 1023 instances in the SCM database [2] in under one second and produces optimal encodings for over 93% of them. For the *min-a* objective, the *DPmin-a* encoder similarly solves all 16,368 database instances within one second, achieving optimal encodings for more than 57% of the cases. The approach also scales to very large constants efficiently. For example, it returns in 8.75 seconds a 7-operation plan for 171398451, only 1 operation away from the known optimal solution [11], while the *min-k* and *min-a* models given in [2] fail to solve it within 24 hours.

The remainder of this paper is organized as follows. Section 2 describes the proposed DP approach in detail. Section 3 presents the Picat implementation using tabling. Section 4 reports experimental results and analysis. Finally, Section 5 concludes the paper and outlines future research directions.

2 A Dynamic Programming Approach to SCM Encoding

Dynamic programming is not a natural approach to SCM encoding because a fundamental assumption of dynamic programming, optimal substructure, does not hold. Given all optimal solutions to SCM for coefficients c_l and c_r it may not be possible to create an optimal solution to SCM for $c = (c_l \ll s) + c_r$. This is because a non-optimal recipe for c_l that shares many intermediate results with a non-optimal recipe for c_r may overall generate a better solution than combining any two optimal solutions. In practice, however, this behavior is rare allowing us to take advantage of the efficiency of DP without too much loss of optimality.

Example 1. Consider computing the constant integer 55415. We can decompose it as $55415 = (6143 \ll 3) + 6271$. The optimal solution for $c_l = 6143$ requires two additions and $c_r = 6271$ requires three additions, respectively. Combining these optimal solutions yields six additions in total:

$$
\begin{aligned}
55415 \cdot x &= (6143 \cdot x) \ll 3 + 6271 \cdot x \\
6143 \cdot x &= x \ll 13 - 2049 \cdot x \\
2049 \cdot x &= x \ll 11 + x \\
6271 \cdot x &= (49 \cdot x) \ll 7 - x \\
49 \cdot x &= x \ll 6 + 15 \cdot x \\
15 \cdot x &= x \ll 4 - x
\end{aligned}
$$

However, by using non-optimal recipes that share intermediate results (i.e. 6143), we can achieve a better overall solution only using 4 additions in total:

$$
\begin{aligned}
55415 \cdot x &= (6143 \cdot x) \ll 12 + 6271 \cdot x \\
6143 \cdot x &= x \ll 12 + 2047 \cdot x \\
6271 \cdot x &= x \ll 7 + 6143 \cdot x \\
2047 \cdot x &= x \ll 11 + x
\end{aligned}
$$

As the SCM problem violates the optimal substructure, DP does not guarantee optimality. To mitigate the issue, the proposed DP approach uses a union operation when constructing the solution recursively. Instead of independently merging sub-plans, the approach takes the union of the sub-plans computed for different chunks and dynamically eliminates redundant intermediate operations. Although this strategy does not restore the optimal substructure, it enables the identification and sharing of common intermediate values across chunks, leading to high-quality encodings in practice.

2.1 Problem Definition and Notation

The single-constant multiplication (SCM) constraint has the form $c \cdot x = y$, where c is a fixed constant, and x and y are integer-domain variables. When $c = 0$ or $c = 1$, the constraint can be trivially encoded as an equality constraint. When c is even, the constraint $c \cdot x = y$ can always be rewritten as $c' \cdot x \ll s = y$, where c' is an odd integer and $\ll$ denotes the left-shift operator. In the remainder of this paper, we assume that c is odd, meaning that its binary representation always ends with a 1.

The target constant c can be decomposed using one of the following operator types when $c > 1$:

$c = (c_l \ll s) + c_r$ (SPLUS)
$c = (c_l \ll s) - c_r$ (SMINUS)
$c = c_l - (c_r \ll s)$ (MINUSS)

The *min-k* objective aims to minimize the number of operations. Since additions with zero bits are free in SAT, the SPLUS operation could be free depending on the bit-width of x. For example, the addition

$$(5 \times x) \ll 4 + x$$

is free if the bit width of x is less than 5. The *min-a* objective aims to minimize the number of one-bit adders required in the final encoding by counting out free additions.

The DP approach decomposes the binary representation of c into chunks using one of the operator types when c is not a base case, and it recursively applies the method to each chunk or its complement. When multiple decompositions are possible, the DP approach explores all of them.

In the following, the constant c is always given in binary format, and the following notations are used to express pattens and operations on binary strings:

- $.^n$: indicates a binary string of length n. In particular, 0^n indicates a string of 0s, and 1^n indicates a string of 1s.
- $|\alpha|$: indicates the length of binary string α.
- $\bar{\alpha}$: indicates the 1's complement of α.

2.2 Details of the Dynamic Programming Formulation

The Base Case: When $c = 1$, the empty set is returned as the recipe.

The SPLUS Rule: The SPLUS rule decomposes c's binary representation into α and β, where both α and β are an odd integers. Let $c_l = \alpha$, $c_r = \beta$, and s be $|\beta|$. The rule recursively applies the DP approach on c_l and c_r, respectively. This rule returns a recipe that takes the union of the recipes from c_l and c_r with the following SPLUS operation added:

$$c = (c_l \ll s) + c_r$$

Example 2. The binary string `110101` (53) has the following three SPLUS decompositions:

$\alpha = 1101$	$\beta = 01$	$c_l = 1101$	$c_r = 1$	$s = 2$
$\alpha = 11$	$\beta = 0101$	$c_l = 11$	$c_r = 101$	$s = 4$
$\alpha = 1$	$\beta = 101001$	$c_l = 1$	$c_r = 10101$	$s = 5$

The SMINUS Rule: The SMINUS rule decomposes c's binary representation into α and β, where α's lowest bit is 0. Let $c_l = \alpha + 1$, $c_r = \bar{\beta} + 1$, and $s = |\beta|$. The rule recursively applies the DP approach on c_l and c_r, respectively. This rule returns a recipe that takes the union of the recipes from c_l and c_r with the following SMINUS operation added:

$$c = (c_l \ll s) - c_r$$

It is obvious that the SMINUS equation holds since $(1 \ll |\beta|) - \bar{\beta} - 1 = \beta$.

Example 3. The binary string `110101` has the following three SMINUS decompositions:

$\alpha = 11010$	$\beta = 1$	$c_l = 11011$	$c_r = 1$	$s = 1$
$\alpha = 110$	$\beta = 101$	$c_l = 111$	$c_r = 11$	$s = 3$
$\alpha = 0$	$\beta = 110101$	$c_l = 1$	$c_r = 1011$	$s = 6$

The MINUSS Rule: The MINUSS rule decomposes c's binary representation into α and β, where α's lowest bit is 0 and $\alpha \neq 0$. Let $c_l = (1 \ll |\alpha\beta|) - (1 \ll |\beta|) + \beta$, $c_r = \bar{\alpha}$, and $s = |\beta|$. The rule recursively applies the DP approach on c_l and c_r, respecitvely. This rule returns a recipe that takes the union of the recipes from c_l and c_r with the following MINUSS operation added:

$$c = c_l - (c_r \ll s)$$

It is not difficult to see that the MINUSS equation holds since

$$(1 \ll |\alpha\beta|) - (1 \ll |\beta|) - (\bar{\alpha} \ll |\beta| = (\alpha \ll |\beta|)$$

Example 4. The binary string `110101` has the following two MINUSS decompositions:

$\alpha = 11010$	$\beta = 1$	$c_l = 11111$	$c_r = 101$	$s = 1$
$\alpha = 110$	$\beta = 101$	$c_l = 11101$	$c_r = 1$	$s = 3$

Discussion The DP approach is incomplete: some encodings, possibly optimal in terms of the number of additions, cannot be generated. For example, for the SCM $1965 \cdot x$, an optimal encoding (shown below) only requires three additions [2]:

$$1965 \cdot x = (131 \cdot x) \ll 4 - 131 \cdot x$$
$$131 \cdot x = x \ll 7 + 3 \cdot x$$
$$3 \cdot x = x \ll 1 + x$$

However, the best encoding obtained via DP requires four additions. The limitation stems from the fact that DP's decompositions are driven by the target integer's binary representation, and some integers, such as 131 in the example, never appear as a chunk or as the complement of a chunk in that representation.

The major advantage of this restriction is that there at most $O(\log c)$ ways of splitting the coefficient c along its binary encoding, rather than $O(c)$ if we considered each possible split for SPLUS.[4]

3 Implementation Using Tabling

Tabling, also known as memoization in logic programming, is a program-evaluation strategy in which the system tables the results of subgoal calls and reuses them to avoid redundant computation and infinite loops [16, 17, 21]. Mode-directed tabling, which was introduced in [6, 7] and later incorporated in other systems (e.g., [15, 19]), lets the programmer declare modes on tabled predicate arguments to control which arguments are used for variant checking and how answers are tabled based on the modes. Mode-directed tabling is particularly useful to express dynamic programming algorithms. This section gives an implementation of the DP algorithm presented in the previous section using tabling in Picat [20].

In Picat, in order to have all of the calls and answers of a predicate or function tabled, users just need to add the keyword `table` before the first rule. Picat also supports mode-directed tabling. For example, the predicate `sp(X,Y,Path,W)`, defined below, searches for a path with the minimum cost from `X` to `Y` in a graph specified by the `edge` predicate.

```
table(+, +, -, min)
sp(X, Y, Path, Cost) ?=>
    Path = [(X, Y)],
    edge(X, Y, Cost).
sp(X, Y, Path, Cost) =>
    Path = [(X, Z)|Path1],
    edge(X, Z, W),
    sp(Z, Y, Path1, Cost1),
    Cost = W + Cost1.
```

[4] Even more for MINUSS since $c_l > c$.

The mode `+` indicates an input argument, the mode `-` indicates an output argument, and the mode `min` instructs Picat to only table one optimal answer for the same input arguments.

Recently, Picat has been extended to support the table modes `mmin` and `mmax`, which instruct the system to table all optimal answers. If `min` were replaced with `mmin` in the example above, the predicate could be used to enumerate all lowest-cost paths between a given pair of nodes.

The following Picat program implements the DP approach:

```
import ordset.

table (+, +, mmin, -)
scm(1, XBits, Cost, Plan) =>
    Cost = 0, Plan = [].
scm(C, XBits, Cost, Plan) =>
    (split_pp(C, C1, C2, S),           % SPLUS: C = (C1 << S) + C2
     OP = $splus(C, C1, C2, S)
    ;
     split_pn(C, C1, C2, S),           % SMINUS: C = (C1 << S) - C2
     OP = $sminus(C, C1, C2, S)
    ;
     split_np(C, C1, C2, S),           % MINUSS: C = C1 - (C2 << S)
     OP = $minuss(C, C1, C2, S)
    )
    comp_op_cost(XBits, OP, ThisCost),
    scm(C1, XBits, _Cost1, Plan1),
    scm(C2, XBits, _Cost2, Plan2),
    Plan12 = union(Plan1, Plan2),
    Plan = Plan12.insert((OP, ThisCost)),
    Cost = sum([T : (_, T) in Plan]).
```

As a plan is represented as an ordered set, the program imports the `ordset` module in the beginning.

The predicate `scm(C, XBits, Cost, Plan)` returns a plan (`Plan`) with cost (`Cost`) for the multiplication `C*X`, where `XBits` is the bit-width of `X`.

The first rule handles the base case when `C = 1`.

The second rule encodes the DP recurrences using a disjunction of three branches. After decomposing the binary representation of `C` into `C1` and `C2`, the rule recursively obtains a plan `Plan1` for `C1` and a plan `Plan2` for `C2`. It then forms a plan for `C` by taking the union of `Plan1` and `Plan2`, adding the newly derived operation together with its associated cost. The total cost is the sum of the costs of all operations in the resulting plan. Note that the disjunction introduces nondeterministic search: if one branch fails, the next branch is automatically tried.

The `comp_op_cost(XBits, OP, Cost)` predicate binds `Cost` to the cost of operation `OP`. For the *min-k* objective, the cost is always 1. For the *min-a* objective, the cost is the number of adders needed for the operation, which is dependent on `XBits`.

Note that the table mode `mmin` is used, which instructs Picat to table all minimum-cost solutions. If the `min` table mode were used instead, the final solution might not be as optimal.

4 Experimental Results

To evaluate the performance of the DP-based encoders (*DPmin-k* and *DPmin-a*), we conducted experiments across a range of problem scales—from small instances with known optimal solutions to medium and very large integers that challenge existing optimal solvers. The DP encoders were compared against a baseline shift-and-add model implemented in Picat (*Baseline*) and two models implemented in MiniZinc [2], *min-k*, which minimizes the number of addition operations, and *min-a*, which minimizes the number of one-bit half/full adders, both of which can guarantee optimality. All experiments were conducted on a system running Ubuntu 22.04 under WSL2, equipped with an Intel Core i7-12700H processor (2.30 GHz), and 32 GB of RAM.

4.1 Evaluation on Small-Sized Integers

DP-based encoders were first evaluated on small-sized integers with bit widths ranging from 1 to 16 bits, for which the optimal solutions obtained from MiniZinc *min-k* and *min-a* models are available [2]. The *min-k* recipe includes optimal encodings for 1023 odd integers up to 2047. *DPmin-k* found encodings for all the 1023 integers in 0.137 second. Among the encodings, 952 (93%) of them are optimal, and 71 are near optimal with each requiring only one more addition than the optimal one.

A similar evaluation was conducted for *DPmin-a*. The *min-a* recipe includes 16368 encodings for 1023 odd integers up to 2047 and bit widths from 1 to 16 bits. *DPmin-a* found encodings for the 16368 cases in 0.903 seconds. Among the encodings, 9424 (57.58%) are optimal in terms of the number of half/full adders, and an additional 5298(32.37%) are near-optimal, requiring no more than five extra half/full adders. A further 1471 (8.99%) exhibit a moderate overhead of six to ten extra half/full adders, and the remaining 175 encodings (1.07%) incur a high overhead, requiring between eleven and sixteen more.

4.2 Evaluation on Medium-Sized Integers

Table 1 compares *DPmin-k* and *Baseline* on time and encoding quality using ten randomly generated medium-sized integers with bit widths ranging from 17 to 25. Although *DPmin-k* incurs higher encoding time, it generates much higher-quality encodings in a compact way. For example, for the integer 30261031,

the *Baseline* encoding requires 17 additions, while the *DPmin-k* encoding only requires 5 additions. The MiniZinc *min-k* model failed to produce an encoding within 24 hours.

Table 1. Comparison of encoding time (in seconds) and encoding quality (number of additions) between *Baseline* and *DPmin-k* on medium-sized integers.

c	***Baseline***		***DPmin-k***	
	time	#adds	time	#adds
464571	0.001	12	0.153	6
7137249	0.001	14	0.201	5
8176693	0.001	12	0.467	6
9330001	0.001	12	1.265	6
11895821	0.001	10	0.494	6
14049001	0.001	15	0.915	6
15055315	0.001	15	0.798	6
16532753	0.001	11	0.599	5
27587603	0.001	12	0.868	6
30261031	0.001	17	0.757	5

Table 2 presents the corresponding comparison between *DPmin-a* and *Baseline* under an 8-bit variable-width assumption on the same set of integers. Although *DPmin-a* is slower than *Baseline*, it achieves notable improvements in encoding quality. Specifically, for the integer 30261031, the number of required half/full adders is reduced from 128 in the *Baseline* encoding to 60 in the *DPmin-a* encoding. As with *min-k*, the MiniZinc-based *min-a* encoder is not able to handle any of these tested instances within 24 hours.

Table 2. A comparison on encoding time (seconds) and quality (the number of half/full adders) between *Baseline* and *DPmin-a* under an 8-bit variable-width assumption on medium-sized integers.

c	***Baseline***		***DPmin-a***	
	time	#adders	time	#adders
464571	0.001	88	0.01	43
7137249	0.001	104	0.02	44
8176693	0.001	88	0.02	57
9330001	0.001	88	0.026	60
11895821	0.001	72	0.028	58
14049001	0.001	112	0.025	51
15055315	0.001	112	0.024	51
16532753	0.001	80	0.018	56
27587603	0.001	88	0.034	62
30261031	0.001	128	0.021	60

4.3 Evaluation on Large Integers

The DP-based encoders are capable of handling large integers of 26 bits or more. For example, for $c = 171398451$, *DPmin-k* returns the following encoding in 8.755 seconds:

```
5 = 1 * (1 << 2) + 1
13 = 1 * (1 << 3) + 5
653 = 5 * (1 << 7) + 13
10461 = 653 * (1 << 4) + 13
167381 = 10461 * (1 << 4) + 5
307 = 5 * (1 << 6) - 13
171398451 = 167381 * (1 << 10) + 307
```

which only uses one more addition than the known optimal encoding [11].

DPmin-a returns the following encoding for $c = 171398451$ and $|x| = 8$ in 0.061 seconds

```
5 = 1 * (1 << 2) + 1
13 = 1 * (1 << 3) + 5
77 = 1 * (1 << 6) + 13
205 = 1 * (1 << 7) + 77
268435251 = 1 * (1 << 28) - 205
268434739 = 268435251 - 1 * (1 << 9)
268432691 = 268434739 - 1 * (1 << 11)
268391731 = 268432691 - 5 * (1 << 13)
267867443 = 268391731 - 1 * (1 << 19)
263673139 = 267867443 - 1 * (1 << 22)
255284531 = 263673139 - 1 * (1 << 23)
171398451 = 255284531 - 5 * (1 << 24)
```

which uses 68 half/full adders. For this instance, no optimal encoding is known.

The difference in encoding times between *DPmin-k* and *DPmin-a* for large integers results from the fact that *DPmin-a* tables fewer answers than DPmin-k. For example, for $c = 3626291$, *DPmin-k* returns 14 encodings, while *DPmin-a* only returns 6 encodings for $|x| = 8$.

Overall, the experimental results provide strong quantitative evidence that the proposed DP-based encoders achieve a favorable trade-off between encoding quality and computational efficiency across a wide spectrum of problem scales. On small-sized instances with known optimal solutions, *DPmin-k* and *DPmin-a* closely match optimal encodings, validating their effectiveness. On medium-sized integers, the DP-based encoders consistently outperform the baseline approach in terms of encoding quality, albeit at the cost of increased encoding time. Finally, the proposed DP-based encoders scale naturally to very large integers, producing high-quality encodings even for instances far beyond the limits of existing optimal solvers.

5 Conclusion

In this paper, we have introduced a DP based method to solve SCM problems by utilizing the binary representation of the target constant at acceptable encoding times. Our DP-based encoders yield optimal solutions over a variety of small

instances at the cost of theoretical optimality, and scales efficiently to large constants where optimal encoders break down. Our approach is succinctly and declaratively implemented in Picat, due to its mode-directed tabling. The current implementation is ready for deployment in SAT encoders, and can also be used to compute tight bounds to speed up optimal encoders.

Our work could be extended in different ways. An important direction would be to adapt the approach to MCM problems, as multiple constants often share common subexpressions, and there are opportunities to reuse intermediate results. The proposed DP framework is well-fitted to this setting by using `union` operation to represent such common intermediate values across different constants.

Second, machine learning can make the search decisions and pruning more knowledgeable at least for *DPmin-k*. In contrast, *DPmin-a* adopts a cost model to steer its way through pruning, while *DPmin-k* simply counts the number of operations per each decomposition path even when there are many unnecessary paths explored. Learned strategies can help direct the search toward more promising decompositions.

Third, it may be possible to improve optimality and speed by looking at other number representations or hybrid strategies. These advancements offer opportunities to close the gap between generating efficient encodings and solving arithmetic constraints efficiently with high-quality SAT-based solvers.

References

1. Aksoy, L., Flores, P.F., Monteiro, J.: Exact and Approximate Algorithms For The Filter Design Optimization Problem. IEEE Trans. Signal Process. **63**(1), 142–154 (2015). https://doi.org/10.1109/TSP.2014.2366713, `https://doi.org/10.1109/TSP.2014.2366713`
2. Bierlee, H., Dekker, J.J., Lagoon, V., Stuckey, P.J., Tack, G.: Single constant multiplication for SAT. In: Dilkina, B. (ed.) Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 21st International Conference, CPAIOR 2024, Uppsala, Sweden, May 28-31, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14742, pp. 84–98. Springer (2024). https://doi.org/10.1007/978-3-031-60597-0_6, `https://doi.org/10.1007/978-3-031-60597-0\_6`
3. Cappello, P., Steiglitz, K.: Some Complexity Issues in Digital Signal Processing. IEEE Transactions on Acoustics, Speech, and Signal Processing **32**(5), 1037–1041 (1984)
4. de Dinechin, F., Kumm, M.: Multiplication by Constants. In: Application-Specific Arithmetic, pp. 365–426. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-42808-1_12
5. Garcia, R., Volkova, A.: Toward the multiple constant multiplication at minimal hardware cost. IEEE Trans. Circuits Syst. I Regul. Pap. **70**(5), 1976–1988 (2023). https://doi.org/10.1109/TCSI.2023.3241859
6. Guo, H., Gupta, G.: Simplifying dynamic programming via tabling. In: Jayaraman, B. (ed.) Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3057, pp. 163–177. Springer (2004)

7. Guo, H.F., Gupta, G.: Simplifying dynamic programming via mode-directed tabling. Softw., Pract. Exper. **38**(1), 75–94 (2008)
8. Gustafsson, O.: Towards Optimal Multiple Constant Multiplication: A Hypergraph Approach. In: 42nd Asilomar Conference on Signals, Systems and Computers, ACSSC 2008, Pacific Grove, CA, USA, October 26-29, 2008. pp. 1805–1809. IEEE (2008). https://doi.org/10.1109/ACSSC.2008.5074738, https://doi.org/10.1109/ACSSC.2008.5074738
9. Gustafsson, O., Dempster, A., Johansson, K., Macleod, M., Wanhammar, L.: Simplified design of constant coefficient multipliers. Circuits Systems and Signal Processing **25**, 225–251 (01 2006). https://doi.org/10.1007/s00034-005-2505-5
10. Kumm, M.: Optimal Constant Multiplication Using Integer Linear Programming. IEEE Trans. Circuits Syst. II Express Briefs **65-II**(5), 567–571 (2018). https://doi.org/10.1109/TCSII.2018.2823780, https://doi.org/10.1109/TCSII.2018.2823780
11. Lagoon, V., Metodi, A.: Deriving optimal multiplication-by-constant circuits with a sat-based constraint engine. In: Proceedings of the MODREF 2020 workshop (2020)
12. Lefèvre, V.: Multiplication by an integer constant. Technical Report N4192, INRIA (2001)
13. Ma, S., Ampadu, P.: Optimal SAT-based Minimum Adder Synthesis of Linear Transformations. In: Lee, H., Geiger, R.L. (eds.) 62nd IEEE International Midwest Symposium on Circuits and Systems, MWSCAS 2019, Dallas, TX, USA, August 4-7, 2019. pp. 335–338. IEEE (2019). https://doi.org/10.1109/MWSCAS.2019.8885033, https://doi.org/10.1109/MWSCAS.2019.8885033
14. Macleod, M.D., Dempster, A.G.: Common subexpression elimination algorithm for low-cost multiplierless implementation of matrix multipliers. Electronics Letters **40**(11), 651–652 (2004)
15. Santos, J., Rocha, R.: Mode-directed tabling and applications in the yaptab system. In: Simões, A., Queirós, R., da Cruz, D.C. (eds.) 1st Symposium on Languages, Applications and Technologies, SLATE 2012, Braga, Portugal, June 21-22, 2012. OASIcs, vol. 21, pp. 25–40. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012). https://doi.org/10.4230/OASICS.SLATE.2012.25
16. Tamaki, H., Sato, T.: OLD resolution with tabulation. In: ICLP. pp. 84–98 (1986)
17. Warren, D.S.: Memoing for logic programs. Comm. of the ACM, Special Section on Logic Programming **35**, 93–111 (1992)
18. Zhou, N.F., Fruhman, J.: A user's guide to Picat. https://picat-lang.org (2013–2026), [Online; accessed May 2, 2026]
19. Zhou, N.F., Kameya, Y., Sato, T.: Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In: ICTAI. pp. 213–218 (2010)
20. Zhou, N., Kjellerstrand, H., Fruhman, J.: Constraint Solving and Planning with Picat. Springer Briefs in Intelligent Systems, Springer (2015). https://doi.org/10.1007/978-3-319-25883-6
21. Zhou, N.F., Sato, T., Shen, Y.D.: Linear tabling strategies and optimizations. Theory and Practice of Logic Programming **8**(1), 81–109 (2008)