PW-CT: Extending Compact-Table to Enforce Pairwise Consistency on Table Constraints

Anthony Schneider(✉) and Berthe Y. Choueiry(✉)

Constraint Systems Laboratory, University of Nebraska-Lincoln, Lincoln, USA
{aschneid,choueiry}@cse.unl.edu

Abstract. The Compact-Table (CT) algorithm is the current state-of-the-art algorithm for enforcing Generalized Arc Consistency (GAC) on table constraints during search. Recently, algorithms for enforcing Pairwise Consistency (PWC), which is strictly stronger than GAC, were shown to be advantageous for solving difficult problems. However, PWC algorithms can be costly in terms of CPU time and memory consumption. As a result, their overhead may offset the savings of search-space reduction. In this paper, we introduce PW-CT, an algorithm that modifies CT to enforce full PWC. We show that PW-CT avoids the high memory requirements of prior PWC algorithms and significantly reduces the time required to enforce PWC.

1 Introduction

Consistency properties and algorithms for enforcing them on a Constraint Satisfaction Problem (CSP) are one of the most intensively studied topics in Constraint Programming (CP). Consistency algorithms are used for inference and effectively reduce the search space of solving a CSP. In particular, Generalized Arc Consistency (GAC) has recently been the focus of extensive research for a good reason: it lends itself towards simple yet highly effective algorithms. Indeed, the low cost and effectiveness of GAC algorithms when paired with an ordering heuristic like dom/wdeg have made them the de facto baseline for research. The current state-of-the-art in GAC algorithms are Compact-Table (CT) [8] and STRBit [27], both of which use bitsets to quickly check for supports and perform tabular reduction – the process of removing invalid tuples from constraints.

Algorithms that enforce pairwise-consistency (PWC) have received a relatively modest amount of attention [23]. Recent algorithms have shown promise on some benchmarks [15–17,20], at times considerably reducing the size of the search space. However, enforcing GAC with either CT or STRBit outperforms

The original version of this chapter was revised: The title has been corrected. The correction to this chapter is available at https://doi.org/10.1007/978-3-319-98334-9_48

Supported by NSF Grant No. RI-1619344. Work completed utilizing the Holland Computing Center of the University of Nebraska, which receives support from the Nebraska Research Initiative. We thank the reviewers for constructive feedback.

J. Hooker (Ed.): CP 2018, LNCS 11008, pp. 345–361, 2018.
https://doi.org/10.1007/978-3-319-98334-9_23

these PWC algorithms due to the latter's initialization overhead, memory usage, and computational cost.

In this paper, we introduce PW-CT, an algorithm for enforcing full PWC, as an extension of CT [8]. PW-CT requires few modifications to the original CT structures, exploits mechanisms from existing PWC algorithms, and integrates additional improvements discussed in this paper. More specifically, PW-CT uses CT (i.e., GAC) as much as possible to avoid costly PWC checks in two ways: by ensuring the problem is GAC before resorting to any PWC checks and by identifying situations where GAC guarantees PWC. Finally, it exploits properties of the dual CSP to speed-up processing and reduce memory consumption. We compare the performance of PW-CT to that of state-of-the-art GAC and full PWC algorithms. We show that PW-CT dominates the latter by a large margin and outperforms both STRBit and CT on several benchmarks.

This paper is structured as follows. Section 2 provides background information. Section 3 reviews the state of the art. Section 4 identifies directions to improve PWC algorithms. Section 5 discusses PW-CT. Section 6 discusses our experiments. Finally, Sect. 7 concludes this paper.

2 Background

A Constraint Satisfaction Problem (CSP) is defined as $\mathcal{P}= (\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ the set of their domain values, and $\mathcal{C}$ is a set of constraints $c_i = \langle R_i, scope(c_i)\rangle$, where R_i is a relation defined on the variables in the scope of the constraint, $scope(c_i) \subseteq \mathcal{X}$, restricting the combination of values that the variables can take at the same time. The arity of a constraint is the cardinality of its scope. Solving a CSP requires assigning to each variable a value from its domain such that all the constraints are satisfied. In this paper, we consider table constraints, where relations are given by their allowed tuples. We use the relational projection operator π to restrict a tuple to a set of variables.

In the *hypergraph* representation of a CSP, the vertices represent CSP variables. Hyperedges represent constraints and connect the variables in the scope of the constraints. In the *dual graph* representation, the vertices represent the CSP constraints and edges connect vertices that share variables. Figure 1 shows the dual graph of a CSP with four nonbinary constraints. The edges are equality constraints forcing the shared variables to agree on the assigned values.

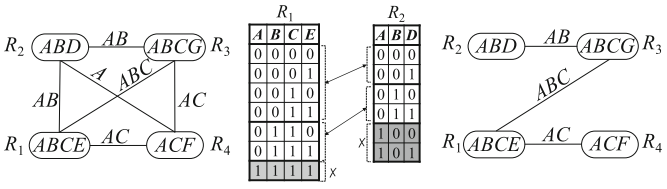

Fig. 1. Dual graph (left), subscopes and blocks (center), a minimal dual graph (right)

We designate by *subscope* the set of CSP variables shared by two constraints, $subscope(c_i, c_j) = scope(c_i) \cap scope(c_j)$. Multiple edges in the dual graph may be

labeled by the same subscope. In Fig. 1, each of the subscopes AB and AC label two edges and each of the subscopes A and ABC label one edge. The equality constraints of the dual graph are binary and *piecewise functional* [10,23]. A binary constraint is said to be piecewise functional if the domains of the variables in its scope can be partitioned such that a set from one variable is supported by at most one set in the other and vice versa. Because of the piecewise functionality of the constraints of the dual graph, each subscope partitions the tuples of a constraint into sets of equivalent tuples, which we call *blocks*. In Fig. 1, the subscope AB partitions each of the two relations R_1 and R_2 into three blocks. We define the *signature* of a block as the set of variable-value pairs of the inducing subscope (e.g., $\{\langle A,0\rangle,\langle B,0\rangle\}$). Thus, a signature is uniquely determined by a combination of a constraint, subscope, and tuple. Janssen et al. [11] and Dechter [7] observed that, in the dual graph, an edge between two vertices is *redundant* if there exists an alternate path between the two vertices such that the shared variables appear in every vertex in the path. Redundant edges can be removed without affecting the set of solutions. Janssen et al. [11] introduced an efficient algorithm for computing the *minimal dual graph* by removing redundant edges. Many minimal graphs may exist, but all are guaranteed to have the same number of edges. Figure 1 shows an example of a minimal dual graph.

In this paper, we exploit the following two consistency properties:

Definition 1. Generalized Arc Consistency (GAC) [18,26]: *A constraint network $\mathcal{P} = (\mathcal{X},\mathcal{D},\mathcal{C})$ is GAC iff, for every constraint $c_i \in \mathcal{C}$, and $\forall x_j \in scope(c_i)$, every value $v \in D(x_j)$ is consistent with c_i (i.e., appears in some support of c_i).*

Definition 2. Pairwise Consistency (PWC) [9]: *A constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D},\mathcal{C})$ is PWC iff, for every tuple t_i in every constraint c_i there is a tuple t_j in every constraint c_j such that $\pi_{subscope(c_i,c_j)}(t_i) = \pi_{subscope(c_i,c_j)}(t_j)$, t_j is called a PW-support of t_i in c_j. A CSP that is both PWC and GAC is said to be full PWC* (fPWC).

3 Related Work

The CT algorithm [8] is the current state-of-the-art algorithm for enforcing GAC.[1] It makes heavy use of a data structure called an RSparseBitSet, which is similar to the sparse set structure [5,6]. In this paper, we extend the definition of the RSparseBitSet to facilitate PWC operations.

STRBit [27] is a GAC algorithm similar to CT in that it operates on bitsets. STRBit differs from CT in its data structures and how it propagates changes. To our knowledge, a comparison of the performance of these two algorithms has not yet appeared in the literature.

Samaras and Stergiou introduced PW-AC [23], an algorithm for enforcing PWC. PW-AC operates on the dual graph of the CSP, taking advantage of the piecewise functionality of the equality constraints. It propagates deletions

[1] In this paper, we consider that a GAC algorithm applies tabular reduction.

of blocks of tuples, rather than individual tuples, and maintains counts of the living tuples in each block. Notably, it iterates over *every pair* of constraints with a non-empty subscope.

The parametrized algorithms PerTuple [12,13] and PerFB [24] enforce m-wise consistency, which generalizes pairwise consistency to every set of m constraints. Both algorithms use data structures that group the equivalent tuples in a constraint based on the piecewise-functionality property.

Algorithm eSTR [15], an extension of the STR family of algorithms for GAC [14,25], enforces fPWC. It maintains PWC by tracking the counts of PW-supports of each tuple in a constraint relative to all other constraints in the problem and verifying a valid PW-support for each tuple as it is processed by STR. Algorithm eSTR^w is a modification of eSTR and enforces a weakened version of fPWC by not re-queuing a constraint after a PW-support is lost.

Algorithms HOSTR and MaxRPWC+r [20] enforce consistency properties that are weaker than PWC and incomparable to each other. fHOSTR, a variant of HOSTR, enforces full PWC, but was found by its authors to be too expensive relative to its weakened version. Paparrizou and Stergiou show that HOSTR and MaxRPWC+r outperform STR2 [14] on certain benchmarks.

Some approaches for enforcing higher order consistencies apply GAC after reformulating the CSP with new constraints or variables. Algorithm DkWC [19] enforces k-wise consistency by adding new hybrid constraints to the problem. The Factor Encoding (FE) enforces fPWC by adding new variables to the problem, thereby increasing the arity of constraints [16]. A decomposition of the FE lessens the imposed arity increases from FE while still enforcing fPWC [17].

4 Improving PWC Algorithms

Below, we describe four distinct techniques to improve the performance of PWC algorithms. These methods can be combined or exploited in isolation.

4.1 Piecewise Functionality

As mentioned in Sect. 2, Samaras and Stergiou [23] exploit the piecewise-functional property of the equality constraints of the dual graph to infer the blocks of equivalent tuples of two constraints with shared variables. If a tuple τ in a constraint c_i does not have a PW-support in another constraint c_j, all tuples in the block induced by $\pi_{subscope(c_i,c_j)}(\tau)$ on c_i can be immediately removed. Further, all other *blocks* of tuples that are PW-supported by τ in all other neighboring constraints (i.e., have the same signature) must also be deleted. This operation is in stark contrast with most GAC-based algorithms that search for supports one tuple at a time (except, of course, AC-5 [10]).

4.2 Pairwise Vs Subscope Reasoning

Algorithms for enforcing pairwise consistency usually operate on *every pair* of constraints with overlapping subscopes (e.g., PW-AC partitions relations pairwise, eSTR counts supports pairwise, etc.). Karakashian et al. [13] and Schneider

et al. [24] exploit the fact that, for a given subscope, any number of relations induce on a relation R_i the same unique partition. For example, in Fig. 1, the blocks induced by subscope $\{A, B\}$ on relation R_1 are the same for *any* relation R_j such that $subscope(R_1, R_j) = \{A, B\}$.

Consequently, identifying and storing a relation's partitions based on unique subscopes rather than by the degree of a vertex in the dual graph can significantly reduce the memory requirements of algorithms that exploit the pairwise functionality of the equality constraints of the dual graph.

4.3 Minimal Dual Graph

As stated in Sect. 2, we can remove redundant edges in the dual graph of a CSP without affecting the set of solutions. In fact, Janssen et al. [11] show that enforcing PWC on a dual graph is equivalent to enforcing PWC on any of its minimal dual graphs. Importantly, removing redundant edges can reduce not only the degree of the graph (thus reducing the number of pairs of constraints over which a PWC algorithm must iterate) but also the number of unique subscopes that a PWC algorithm must take into consideration. For instance, in the example shown in Fig. 1, removing redundant edges eliminates: (1) The need to compute and store the partitions of R_1 for the subscope $\{A, B\}$ and the partitions of R_3 for the subscope $\{A, C\}$ and (2) The subscope $\{A\}$ and the partition it induces on each of R_2 and R_4. Consequently, a minimal dual graph can reduce the number of neighbors of a constraint in the problem, the number of unique subscopes incident to a constraint, and may entirely eliminate some subscopes from the problem. We conclude that a PWC algorithm that operates on a minimal dual graph may reduce its memory requirements and increase its propagation speed because of the reduced number of subscopes to consider per constraint and the total number of unique subscopes.

4.4 Determining When GAC Is Enough to Enforce PWC

In some situations, GAC is enough to enforce PWC between constraints. The algorithm eSTR, for example, only checks for PW-supports over "non-trivial" subscopes, which are subscopes with a cardinality strictly greater than one [15]. In fact, the particularity of constraints intersecting on at most one variable is discussed by Bessiere et al. [3] but PWC is unexplainably excluded from the corresponding theorem. Below, we restate this property and give a proof:

Proposition 1. *GAC is sufficient to enforce PWC over trivial subscopes.*

Proof. Consider the CSP $\mathcal{P}= (\mathcal{X}, \mathcal{D}, \mathcal{C})$. If a subscope is trivial (e.g., $subscope = \{x\} \subset \mathcal{X}$) the signature of each block induced by this subscope is one variable-value pair (e.g., $\langle x, a\rangle$). Thus, the block loses all PW-supports only if $\langle x, a\rangle$ is removed from the problem. If $\langle x, a\rangle$ is deleted, a tabular-reduction algorithm necessarily removes all tuples with $\langle x, a\rangle$ from the problem. On the other hand, if $\langle x, a\rangle$ is alive after enforcing GAC, then, by definition, $\forall c_i \in \mathcal{C}$ such that

$x \in \text{scope}(c_i)$, there is at least one living tuple τ in the relation of c_i such that $\pi_x(\tau) = a$. □

We elucidate a particular situation, which arises during search, in which the above property holds even for non-trivial subscopes as long as GAC is enforced on a constraint prior to running a PWC algorithm:

Proposition 2. *GAC is sufficient to enforce PWC on a block induced by a non-trivial subscope whose signature includes a deleted variable-value pair.*

Proof. This proposition follows from Proposition 1. Consider a block b_i induced by a non-trivial subscope σ_i on the constraint c_i. If a dead variable-value pair $\langle x, a\rangle$ is in the block's signature, a tabular-reduction GAC algorithm removes all tuples with $\langle x, a\rangle$ from the problem, and as a result, it removes all the PW-supports of b_i from the relations of neighboring constraints because they necessarily also contain $\langle x, a\rangle$ in their signatures. □

Algorithm eSTR [15] implicitly applies this principle by ensuring that all the variable-value pairs of a tuple are alive before checking whether or not the tuple has PW-supports in neighboring constraints. We exploit Proposition 2 in PWC algorithms in a slightly more efficient manner. Assume a CSP is already PWC, after a variable is instantiated, we run an STR-based GAC, which may delete tuples from constraints. We now need to process these deleted tuples because some of them may be the sole tuples of some blocks that were the PW-support of other blocks in other constraints. In the case that a variable-value pair deleted by GAC appears in the signature of a block in which one of these deleted tuples appears, we can safely skip the processing we intended to do because its result is ensured by GAC. This operation is implemented in function EnforcePWC (Algorithm 4) in Sect. 5.2. In summary, Algorithm eSTR exploits the property by checking first whether the tuple is GAC and we exploit the property by avoiding checking PWC on blocks that we know are dead.

5 PW-CT

We now introduce PW-CT, an fPWC algorithm that exploits the mechanisms presented in Sect. 4. First, we describe how we modify the RSparseBitSet class of the CT algorithm [8] and the additional data structures required for enforcing PWC. Then, we provide the pseudocode of PW-CT.

5.1 Data Structures

PW-CT exploits the functions and data structures CT [8]. Below, we review the CT data structures and the additions required for PW-CT.

Support Structures. Both CT and PW-CT represent the living tuples in a constraint as an RSparseBitSet. The RSparseBitSet stores four members: an array of reversible 64-bit integers called words,[2] a reversible integer called limit that represents the number of non-zero integers in words, an array called index that stores the position of all non-zero integers in words in locations less-than or equal-to limit, and an array called mask used to modify the set. Demeulenaere et al. [8] introduce member functions of the RSparseBitSet used by PW-CT which we briefly review: function `addToMask` takes an array and alters mask to be the bitwise OR of the array and the current mask, function `intersectWithMask` alters words to be the bitwise AND of the current words and mask, and function `clearMask` sets the integers in mask to 0.

The RSparseBitSet for a constraint c_i is denoted as `living`(c_i). The data structure supports$[c_i, x, a]$ is a static array of bits corresponding to the tuples of a constraint c_i that have the value a for variable x.[3] To improve performance of various functions in PW-CT, we introduce a structure indices$[c_i, x, a]$, which is an RSparseBitSet that stores the positions in supports$[c_i, x, a]$ that are non-zero.

PW-CT uses two maps. The first, incidentCons$[\sigma]$, gives the list of constraints incident to a non-trivial subscope σ. The second, incidentSubscopes$[c_i]$, gives the list of non-trivial subscopes incident to a constraint c_i. We can optionally use the minimal dual graph to reduce the number of generated subscopes in each map without affecting the level of consistency enforced (see Sect. 4). Importantly, all these support structures are created at initialization.

Blocks. We represent a block as a simple structure with a member sets, which is a vector of pointers to supports$[c_i, x, a]$ representing the signature of the block, and a member commonIndices, which is an RSparseBitSet of the indices shared by all of the supports in sets. Performing an intersection of the sets in a block computes the set of tuples with the signature corresponding to sets. In PW-CT, blocks are never stored but always computed dynamically during search.

The function CreateBlock (Algorithm 1) takes as input a constraint, tuple, and subscope and returns a block structure, which can be used to dynamically compute the partition of tuples of the constraint with the corresponding signature. The RSparseBitSet commonIndices improves performance of some operations of the methods listed in Algorithm 2. Note that the method `initIntersection` called in Line 7 is defined in Algorithm 2 and makes use of the call `swap` in Line 5.

Additional Methods for the RSparseBitSet Class. Algorithm 2 introduces additional methods for the RSparseBitSet class for use in PW-CT. The method `initIntersection` is used in CreateBlock to initialize the RSparseBitSet with the indices common to a collection of RSparseBitSets.

[2] 64-bit on most current architectures.

[3] Note that we have added the additional parameter c_i to supports[] to uniquely determine the constraint's supports we are referring to in the pseudocode.

Algorithm 1. CreateBlock(c_i, τ, σ)

```
   Input: A constraint c_i, a tuple τ, and a subscope σ
   Output: A block b
1  j ← 0
2  foreach variable x ∈ σ do
3  |  b.sets[j] ← supports[c_i][x][τ[x]]    // τ[x] is the value for x in tuple τ
4  |  ind[j] ← indices[c_i][x][τ[x]]
5  |  if ind[j].limit < ind[0].limit then swap(ind[j], ind[0])
6  |  j ← j + 1
7  b.commonIndices.initIntersection(ind)
8  return b
```

We overload the original RSparesBitSet method `intersectIndex` to operate on blocks. It is similar in behavior to the original `intersectIndex`, differing in that it determines if a *block* of tuples has a support in the set, rather than a single variable-value pair. The method `removeBlock` computes the set-difference between the RSparseBitSet and a block of tuples.

Now, we list functions omitted from Algorithm 2 for brevity. The methods `save` and `restore` respectively save and restore the state of the reversible elements in the RSparseBitSet. We maintain the number of living bits when altering the set and `numSet` returns this value.[4] PW-CT relies on the ability to discover the tuples removed between two points in time (the delta of the set). To this end, method `computeDelta` returns an RSparseBitSet containing the bits removed between the current state of the RSparseBitSet and the last stored state. Method `clearDelta` readies the set to track the next set of removed tuples, but does not alter the currently set bits. These were implemented using the method `save` and comparing the reversible primitives of the current state of the set and its previously saved state. Method `addBlockToMask` behaves like the original `addToMask`, but adds to the mask only those bits common to all bit-sets in the block. Its implementation follows from `addToMask` and `intersectIndex`. We also assume that the bits in the RSparseBitSets are iterable and treat the bits and the tuples they represent interchangeably in our pseudocode for simplicity.

5.2 Enforcing PW-CT

Roughly speaking, PW-CT has two main phases: a GAC phase, in which CT is executed until quiescence, and a PWC phase that performs a *single* pass over the tuples deleted by CT to uncover new non-PWC blocks. PW-CT maintains two queues: CTQueue tracks constraints that must be 'checked for GAC' and PWCQueue tracks constraints that have lost tuples thus threatening the PW-consistency of blocks in other constraints. Both queues are sets.

Function Lookahead (Algorithm 3) is the entry point for PW-CT. Lines 4 to 7 run CT until quiescence and enqueues constraints modified by GAC into

[4] This can be done efficiently in C++ with Clang/GCC's `__builtin_popcountll`.

Algorithm 2. Additional algorithms required for RSparseBitSet

```
1  Method initIntersection(sets: A vector of RSparseBitSets):
2      limit ← −1
3      index ← ∅
4      Expand words and index to size of sets[0].words
5      foreach i ← 0 to sets[0].limit do
6          offset ← sets[0].index[i]
7          bits ← sets[0].words[offset]
8          for set ∈ sets and bits ≠ 0 do
9              bits ← bits & set.words[offset]                    // Bitwise AND
10         if bits ≠ 0 then
11             words[offset] ← bits
12             limit ← limit + 1
13             index[limit] ← offset
14 Method intersectIndex(block: A Block created by createBlock):
       // If limit < block.commonIndices.numSet(), iterate from 0 to limit
15     for offset ∈ block.commonIndices do
16         intersection ← words[offset]
17         for set ∈ block.sets and intersection ≠ 0 do
18             intersection ← intersection & set.words[offset]    // Bitwise AND
19         if intersection ≠ 0 then return offset
20     return -1
21 Method removeBlock(block: A Block created by createBlock):
22     for i ← limit to 0 do
23         offset ← index[i]
24         if offset ∈ block.commonIndices then
25             b ← 64-bit Integer with all bits set
26             for set ∈ block.sets and b ≠ 0 do
27                 b ← b & set.words[offset]                      // Bitwise AND
28             words[offset] ← words[offset] & ∼b                  // Bitwise NOT
29             if words[offset] = 0 then
30                 index[i] ← index[limit]
31                 index[limit] ← offset
32                 limit ← limit −1
```

PWCQueue. CT enqueues constraints with modified variables into CTQueue, thus, when execution hits Line 9, the problem is GAC but not necessarily PWC. Lines 9 to 13 call function EnforcePWC (Algorithm 4) on modified constraints to determine if the removal of tuples in each constraint c_i in the queue causes the loss of a PW-support in another constraint.

EnforcePWC iterates over all subscopes incident to a constraint and the constraint's most recently removed tuples, checking whether the block induced

Algorithm 3. LOOKAHEAD($\mathcal{P}$) Enforces PWC on a CSP $\mathcal{P}$

```
   Input: A CSP P = (X, D, C)
   Output: Whether the current problem is consistent
1  consistent ← true
2  if P has not been preprocessed then
3      consistent ← PreProcess(C)
4  while consistent and not empty(CTQueue) do
5      c_i ← pop(CTQueue)
6      consistent ← CompactTable(c_i)
7      if c_i was modified then push(c_i,PWCQueue)
8      if consistent and empty(CTQueue) then
9          mCons ← PWCQueue
10         for c_i ∈ mCons and consistent do
11             consistent ← EnforcePWC(c_i)
12             living(c_i).clearDelta()
13             PWCQueue ← PWCQueue \{c_i}
14 return consistent
```

by the combination of each subscope and tuple is empty. As discussed in Sect. 4.4, any blocks whose signatures have variable-value pairs removed by GAC necessarily have had all of their supporting blocks in neighboring constraints removed as well. The loop beginning at Line 4 in EnforcePWC (Algorithm 4) takes advantage of this insight by discarding blocks of tuples from consideration for PW-support checks, skipping unnecessary calls to ReviseBlock (Algorithm 5). It uses a mechanism similar to the incremental and reset-based updates [22], where Δ_x is the set of values of variable x removed by the previous call to CT.

Lines 16 to 19 check the block induced by each removed tuple for the current subscope for validity by calling function ReviseBlock (Algorithm 5). If no other tuples in the induced block are alive in the constraint, ReviseBlock removes the piecewise-functional blocks from all other constraints incident to the current subscope, and enqueues the constraints modified during this process. Multiple tuples in the set of removed tuples may belong to the same block for a given subscope, so, Line 19 removes all other tuples from that block from the set of tuples to check (as successive calls for the same block would be redundant).

It is advantageous to interleave CT and EnforcePWC calls because tuples removed by EnforcePWC may enable value deletions that can be propagated quickly by CT. To prevent running EnforcePWC until quiescence on the first pass, a copy of the queue is created in Line 9 of Lookahead (Algorithm 3). As a result, each modified constraint is processed at most once at each PWC pass.

Proposition 3. *If the CSP is initially PWC,* Lookahead *guarantees fPWC.*

Proof. Consider a constraint c_i altered by CT. Because the problem was PWC prior to running CT, the only 'endangered' blocks in c_i have tuples deleted by CT. To enforce PWC, we need to check if any block b_i whose signature is a

Algorithm 4. EnforcePWC(c_i) Propagates invalid blocks of c_i

```
   Input: Constraint c_i that has been modified by CT
   Output: Whether the current problem is consistent
 1 tupsToCheck ← living(c_i).computeDelta()
 2 for σ ∈ incidentSubscopes[c_i] do
 3     tupsToCheck.save()
 4     for variable x ∈ σ s.t. x was modified on previous call to CT do
 5         if |D(x)| < |Δ_x| then
 6             tupsToCheck.clearMask()
 7             for value a ∈ D(x) do
 8                 tupsToCheck.addToMask(supports[c_i][x][a])
 9             tupsToCheck.intersectWithMask()
10         else
11             for value a ∈ Δ_x do
12                 b ← an empty block
13                 b.sets ← supports[c_i][x][a]
14                 b.indices ← indices[c_i][x][a]
15                 tupsToCheck.removeBlock(b)
16     for τ ∈ tupsToCheck do
17         consistent ← ReviseBlock(c_i, σ, τ)
18         if not consistent then return false
19         tupsToCheck.removeBlock(CreateBlock(c_i, σ, τ))
20     tupsToCheck.restore()
21 return true
```

Algorithm 5. ReviseBlock(c_i, σ, τ) Removes supports of empty block

```
   Input: A constraint c_i, a subscope σ, and a tuple τ
   Output: Whether the current problem is consistent
 1 if living(c_i).intersectIndex(CreateBlock(c_i, σ, τ)) = -1 then
 2     for c_j ∈ incidentCons[σ] s.t. c_i ≠ c_j do
 3         living(c_j).removeBlock(CreateBlock(c_j, σ, τ))
 4         if living(c_j) was modified then
 5             if living(c_j).numSet() = 0 then return false
 6             push(c_j,PWCQueue)
 7             push(c_j,CTQueue)
 8 return true
```

combination of a deleted tuple τ of c_i, a subscope σ_i incident to c_i and c_i is empty as a result of CT. If we find a block b_i to be empty, we can remove the blocks that are PW-supports of b_i from all constraints c_j incident to σ_i. Because each c_j modified in ReviseBlock is added to the PWCQueue (Line 6), the removal of any tuple in c_j by ReviseBlock that emptied a block induced on any subscope

σ_j is necessarily detected by the next call to EnforcePWC(c_j). Running CT in between calls to EnforcePWC on any modified constraint ensures that the domains of the variables in the scope of the constraint are 'synced' with the constraint's relation, thus, ensuring fPWC. □

Proposition 4. *The time complexity of calling* EnforcePWC *on a constraint is* $O((|\mathcal{C}| \cdot t) \cdot (\lceil \frac{t}{64} \rceil \cdot |\mathcal{C}| + |\sigma|))$*, where* t *is the number of tuples in the largest constraint and* σ *the largest subscope.*

Proof. ReviseBlock iterates over the constraints incident to a subscope, which in the worst case is $|\mathcal{C}|-1$. Each constraint may need to call `removeBlock`, which requires iterating over $\lceil t/64 \rceil$ elements. Creating the block requires iterating over σ. The only tuples evaluated by ReviseBlock are those that have been removed from a constraint, and at most t tuples can be removed. A removed tuple can be revised for each of its constraint's incident subscopes. In the worst case, a constraint has $|\mathcal{C}| - 1$ neighbors in the dual graph, and each neighbor induces a unique subscope. Therefore, each tuple in the problem may cause ReviseBlock to be called $O((|\mathcal{C}| \cdot t) \cdot (\lceil \frac{t}{64} \rceil \cdot |\mathcal{C}| + |\sigma|))$ times. □

Algorithm 6. PreProcess(C) — Runs CT and removes non-PWC tuples

```
   Input: A set of constraints C
   Output: Whether the current problem is consistent
1  Run CT until quiescence
2  if consistent then
3      consistent ← InitPWC(C)
4      if consistent then
5          forall c_i ∈ C do living(c_i).clearDelta()
6          consistent ← InitPWC(C)
7  return consistent
```

PW-CT requires an additional initialization step to guarantee that preprocessing enforces fPWC. Consider the tuple $\tau = \{\langle A, 1\rangle, \langle B, 1\rangle, \langle C, 1\rangle, \langle E, 1\rangle\}$ in R_1 in Fig. 1. Each variable-value pair in τ has a GAC support in R_2, but no PW-support. EnforcePWC operates on deleted tuples in order to propagate PW-support removals, but because τ's variable-value pairs are GAC, τ is not deleted by CT. Therefore, EnforcePWC is not called. To remedy this situation, all blocks that initially lack PW-supports need first to be removed from the problem. Once this removal is done, EnforcePWC can then evaluate the deleted tuples and propagate any other blocks that are emptied by their removal. Function PreProcess (Algorithm 6) accomplishes this operation by first enforcing GAC with CT and then calling function InitPWC (Algorithm 7).

Function InitPWC considers each subscope σ in the problem. It begins by finding the constraint c_s with the smallest number of living tuples incident to σ.

Algorithm 7. InitPWC(C) Partially enforces PWC on the constraints

```
   Input: A set of constraints C
   Output: Whether the problem is consistent at preprocessing
 1 for σ ∈ Subscopes do
 2     c_s ← constraint with fewest living tuples ∈ incidentCons[σ]
 3     foreach c_i ∈ incidentCons[σ] do living(c_i).clearMask()
 4     toCheck ← living(c_s)                                  // Makes a copy
 5     for τ ∈ toCheck do
 6         tuplePWC ← true
 7         foreach c_i ∈ incidentCons[σ] and tuplePWC do
 8             if living(c_i).intersectIndex(CreateBlock(c_i, τ, σ)) = −1 then
 9                 tuplePWC ← false
10         if tuplePWC then
11             foreach c_i ∈ incidentCons[σ] do
12                 living(c_i).addBlockToMask(CreateBlock(c_i, τ, σ));
13         toCheck.removeBlock(CreateBlock(c_s, τ, σ))
14     foreach c_i ∈ incidentCons[σ] do
15         living(c_i).intersectWithMask()
16         if living(c_i) was modified then
17             push(c_i,PWCQueue)
18             push(c_i,CTQueue)
19         if living(c_i).numSet = 0 then return false
20
21 return true
```

The algorithm checks if the blocks induced by the subscope σ for each tuple τ in $\texttt{living}(c_s)$ has a PW-support in all constraints incident to σ (Lines 6 to 9). If the block is supported, then, for each constraint c_j incident to σ, we add the block of tuples induced by τ to the mask of $\texttt{living}(c_j)$ (Line 12). After all blocks of c_s are processed, the masks of each RSparseBitSet $\texttt{living}(c_j)$ contain only PW-supported tuples. Line 15 removes non-PWC tuples by calling the `intersectWithMask` method for each $\texttt{living}(c_j)$. In practice, we found that in some problems the number of initially non-PWC tuples can be extremely large causing significant slowdown in the first call to EnforcePWC after PreProcess. To alleviate some of this burden, Function PreProcess runs twice InitPWC. The second call to InitPWC tends to remove fewer tuples than the first and guarantees the removal of any block that lost PW-supports due to the first call. Note that only the first call is strictly necessary for correctness.

6 Experiments

Our experiments run on 72 benchmarks of non-binary CSPs.[5] Instances with intension and global constraints are converted to positive table constraints, resulting in a total of 2,210 instances. We evaluate and compare a total of 11 algorithms (Table 1) starting with the following: STR2 [14], STRBit [27], and CT [8] enforce GAC by table reduction; eSTR2 [15] enforces fPWC; eSTR2^w [15] enforces a weakened version of fPWC; STRBit+PW-AC is an unpublished hybrid algorithm that runs STRBit until quiescence before running PW-AC (greatly enhancing PW-AC's effectiveness); and PW-CT enforces fPWC. Additionally, we evaluate eSTR2^m, eSTR2^{wm}, and PW-CT^m as variants of eSTR2, eSTR2^w, and PW-CT respectively, obtained on a minimal dual graph. Finally, Algorithm STRBit+PW-AC^{ms} is a variant of STRBit+PW-AC that operates on a minimal dual graph and propagates via subscopes. We test ordering heuristics dom/ddeg [2] and dom/wdeg [4],[6] finding the first consistent solution with our custom solver. We limit each run to 7,200 s and 8GB of memory. When an algorithm times out, we add 7,200 s to the CPU time and indicate with a $>$ sign that the time reported is a lower bound.

We perform a paired t-test between all algorithms with a significance level $\alpha = .05$. Under dom/ddeg, CT is the best algorithm with statistical significance. Most notably, we find that both PW-CT and PW-CT^m outperform all other eight algorithms, *including both* STR2 and STRBit. To our knowledge, PW-CT is the first fPWC algorithm to dominate a recent GAC algorithm. Exploiting a minimal dual graph improves, with statistical significance, *every* PWC algorithm tested. One exception occurs in the benchmark bddLarge, which times out while computing the minimal dual graph. Indeed, the number of unique, non-trivial subscopes in the problem is extremely large, averaging 1,123,484 over its instances.[7] Surprisingly, PW-CT completes every instance in the benchmark, while all other PWC algorithms fail to complete even one instance. Not including bddLarge, we are able to load 1,707 instances of which 854 have non-trivial subscopes. The average number of unique subscopes on the full graph is 399.5, and 264.3 on the minimal. The remainder of the section discusses only the variants of the PWC algorithms exploiting a minimal dual graph given their superiority.

Table 1 compares the performance of the 11 algorithms. It provides the number of instances completed (#Cmpltd), the total CPU time (ΣCPU) over instances completed by at least one algorithm, the number of memouts (#MO), and the average number of node visits for instances completed by all algorithms (#NV). We exclude instances not solved by any algorithm.

PW-CT has almost the same number of memouts as GAC algorithms, testifying to its low memory consumption relative to all other PWC algorithms (34

[5] http://www.cril.univ-artois.fr/CPAI08/.

[6] Although dom/wdeg is generally more effective than dom/ddeg, the decisions made by dom/wdeg are considered too unstable to objectively allow comparing algorithms' performance. Researchers studying the performance of HLC during search typically use dom/ddeg in their experiments [1,20,21].

[7] Because bddLarge is an extreme outlier, we omit it from the results.

Table 1. Summary statistics of all tested instances

		dom/ddeg				dom/wdeg			
		#Cmpltd	ΣCPU (sec)	#MO	#NV	#Cmpltd	ΣCPU (sec)	#MO	#NV
71 Benchmarks tested with 2,210 total instances total									
GAC	CT	1,411	>449,421	65	2.90M	1,474	>338,246	65	1.65M
	STR2	1,284	>1,327,706	64	2.90M	1,355	>1,164,208	64	1.65M
	STRBit	1,370	>765,923	64	2.90M	1,445	>600,089	65	1.65M
fPWC	PW-CT	1,403	>579,112	65	0.99M	1,428	>715,885	65	0.82M
	PW-CT^m	1,403	>567,500	65	0.99M	1,431	>696,622	65	0.82M
	STRBit+PW-AC	1,213	>1,738,628	137	0.99M	1,263	>1,752,986	139	0.84M
	STRBit+PW-AC^{ms}	1,247	>1,472,804	113	0.99M	1,290	>1,527,538	113	0.84M
	eSTR2	1,231	>1,750,990	102	0.99M	1,282	>1,769,454	102	0.83M
	eSTR2^m	1,248	>1,588,847	99	0.99M	1,295	>1,627,281	99	0.83M
wPWC	eSTR2^w	1,227	>1,784,866	102	1.01M	1,280	>1,769,529	102	1.1M
	eSTR2^{wm}	1,243	>1,629,846	99	1.01M	1,294	>1,622,247	99	1.1M

to 72 fewer memouts). PWC algorithms that exploit a minimal dual graph incur fewer memouts than their original versions, testifying to the importance of using a minimal dual graph for PWC algorithms. When using dom/wdeg, the results of the pairwise t-tests are identical to dom/ddeg with the exception of STRBit and PW-CT. STRBit improves dramatically, beating both variants of PW-CT. Even so, it is clear from Table 1 that the performance of PW-CT is substantially closer to state-of-the-art GAC than to other algorithms used to enforce fPWC.

Figure 2 shows cumulative graphs of the number of instances completed for a given time using dom/ddeg: it compares the performance of PW-CT^m with the other GAC algorithms (left) and PWC algorithms (right). While CT wins, PW-CT^m is a close second, dominating the other two GAC algorithms. PW-CT^m clearly dominates all PWC algorithms by a large margin.

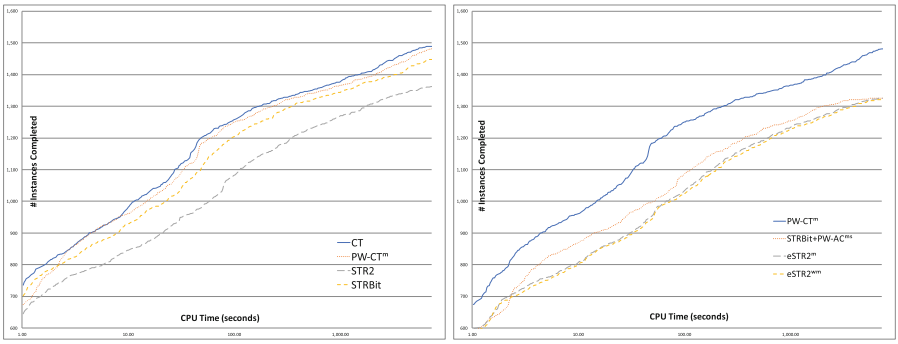

Fig. 2. PW-CT^m vs. GAC-based (left) and PWC algorithms (right) with dom/ddeg

Table 2 shows select benchmarks where PW-CT outperforms CT. These benchmarks were previously shown to benefit from enforcing PWC [20]. It is

Table 2. Select benchmarks using dom/ddeg (#I is the number of instances completed by at least one algorithm, and σ is the average number of non-trivial subscopes)

Benchmarks	#I	$\#\sigma$	ΣCPU (sec)						
			CT	STR2	STRBit	PW-CT^m	STRBit+PW-AC^{ms}	eSTR2^m	eSTR2^{wm}
aim-100	24	146.9	>46,305	>54,997	>51,105	10,766	>19,078	>24,030	>37,687
aim-200	12	218.5	>34,208	>40,218	>35,594	69	121	253	>16,773
dubois	10	2.0	>29,801	>32,711	>31,363	10,798	>17,833	>20,061	>22,621
modRenault	50	61.6	2,553	>12,520	6,036	440	1,517	2,037	2,446
radar-8-24-3-2	1	113	2,621	3,450	3,452	2,612	>7,200	3,475	3,475

thus not surprising that those results hold for PW-CT. Notably, with the exception of the radar benchmark, all PWC algorithms outperform CT, emphasizing the usefulness of enforcing fPWC on difficult problems.

7 Conclusion

In this paper, we show that all PWC algorithms benefit from using a minimal dual graph to improve time and space cost and that the performance of PW-CT, our new algorithm for fPWC, is second only to CT, the best GAC algorithm.

References

1. Balafrej, A., Bessière, C., Paparrizou, A.: Multi-armed bandits for adaptive constraint propagation. In: Proceedings of IJCAI 2015, pp. 290–296 (2015)
2. Bessière, C., Régin, J.-C.: MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118, pp. 61–75. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61551-2_66
3. Bessière, C., Stergiou, K., Walsh, T.: Domain filtering consistencies for non-binary constraints. Artif. Intell. **172**, 800–822 (2008)
4. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of ECAI 2004, pp. 146–150 (2004)
5. Briggs, P., Torczon, L.: An efficient representation for sparse sets. ACM Lett. Program. Lang. Syst. **2**(1–4), 59–69 (1993)
6. le Clément, V., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: Proceedings of the CP Workshop on TRICS 2013 (2013)
7. Dechter, R.: Constraint Processing. Morgan Kaufmann, Burlington (2003)
8. Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.-C., Schaus, P.: Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 207–223. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_14
9. Gyssens, M.: On the complexity of join dependencies. ACM Trans. Database Syst. **11**(1), 81–108 (1986)
10. Hentenryck, P.V., Deville, Y., Teng, C.M.: A generic arc consistency algorithm and its specializations. Artif. Intell. **57**, 291–321 (1992)
11. Janssen, P., Jégou, P., Nougier, B., Vilarem, M.: A filtering process for general constraint-satisfaction problems: achieving pairwise-consistency using an associated binary representation. In: IEEE Workshop on Tools for AI, pp. 420–427 (1989)

12. Karakashian, S., Woodward, R., Choueiry, B.Y.: Improving the performance of consistency algorithms by localizing and bolstering propagation in a tree decomposition. In: Proceedings of AAAI 2013, pp. 466–473 (2013)
13. Karakashian, S., Woodward, R., Reeson, C., Choueiry, B.Y., Bessiere, C.: A first practical algorithm for high levels of relational consistency. In: Proceedings of AAAI 2010, pp. 101–107 (2010)
14. Lecoutre, C.: STR2: optimized simple tabular reduction for table constraints. Constraints **16**(4), 341–371 (2011)
15. Lecoutre, C., Paparrizou, A., Stergiou, K.: Extending STR to a higher-order consistency. In: Proceedings of AAAI 2013, pp. 576–582 (2013)
16. Likitvivatanavong, C., Xia, W., Yap, R.H.C.: Higher-order consistencies through GAC on factor variables. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 497–513. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_37
17. Likitvivatanavong, C., Xia, W., Yap, R.: Decomposition of the factor encoding for CSPs. In: Proceedings of IJCAI 2015, pp. 353–359 (2015)
18. Mackworth, A.K.: Consistency in networks of relations. Artif. Intell. **8**, 99–118 (1977)
19. Mairy, J.-B., Deville, Y., Lecoutre, C.: Domain k-wise consistency made as simple as generalized arc consistency. In: Simonis, H. (ed.) CPAIOR 2014. LNCS, vol. 8451, pp. 235–250. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07046-9_17
20. Paparrizou, A., Stergiou, K.: Strong local consistency algorithms for table constraints. Constraints **21**(2), 163–197 (2016)
21. Paparrizou, A., Stergiou, K.: On neighborhood singleton consistencies. In: Proceedings of IJCAI 2017, pp. 736–742 (2017)
22. Perez, G., Régin, J.-C.: Improving GAC-4 for table and MDD constraints. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 606–621. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_44
23. Samaras, N., Stergiou, K.: Binary encodings of non-binary constraint satisfaction problems: algorithms and experimental results. In: JAIR vol. 24, pp. 641–684 (2005)
24. Schneider, A., Woodward, R.J., Choueiry, B.Y., Bessiere, C.: Improving relational consistency algorithms using dynamic relation partitioning. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 688–704. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_50
25. Ullmann, J.R.: Partition search for non-binary constraint satisfaction. Inf. Sci. **177**(18), 3639–3678 (2007)
26. Waltz, D.: Understanding line drawings of scenes with shadows. In: Winston, P. (ed.) The Psychology of Computer Vision, pp. 19–91. McGraw-Hill Inc., New York City (1975)
27. Wang, R., Xia, W., Yap, R.H.C., Li, Z.: Optimizing simple tabular reduction with a bitwise representation. In: Proceedings of IJCAI 2016, pp. 787–793 (2016)