Implementation and Evaluation of a Compact Table Propagator in Gecode

Linnea Ingmar

28th April 2017

Contents

1 Introduction

In Constraint Programming (CP), every constraint is associated with a propagator algorithm. The propagator algorithm filters out impossible values for the variables related to the constraint. For the TABLE constraint, several propagator algorithms are known. In 2016, a new propagator algorithm for the TABLE constraint was published [2], called Compact Table (CT). Preliminary results indicate that CT outperforms the previously known algorithms. There has been no attempt to implement CT in the constraint solver Gecode [3], and consequently its performance in Gecode is unknown.

1.1 Goal

The goal of this thesis is to implement a CT propagator algorithm for the TABLE constraint in Gecode, and to evaluate its performance with respect to the existing propagators.

1.2 Contributions

State the contributions, perhaps as a bulleted list, referring to the different parts of the paper, as opposed to giving a traditional outline. (As suggested by Olle Gallmo.)

This thesis contributes with the following:

- The relevant preliminaries have been covered in Section 2.
- The algorithms presented in [2] have been modified to suit the target constraint solver Gecode, and are presented and explained in Section 3.
- The CT algorithm has been implemented in Gecode, see Section 4.
- The performance of the CT algorithm has been evaluated, see Section 5.
- ...

2 Background

This section provides a background that is relevant for the following sections. It is divided into five parts: Section 2.1 introduces Constraint Programming. Section 2.3 gives an overview of Gecode, a constraint solver. Section 2.4 introduces the TABLE constraint. Section 2.5 describes the main concepts of the Compact Table (CT) algorithm. Finally, Section 2.6 describes the main idea of Reversible? Sparse Bit-Sets, a data structure that is used in the CT algorithm.

2.1 Constraint Programming

This section introduces the concept of Constraint Programming (CP).

CP is a programming paradigm that is used for solving combinatorial problems. A problem is modelled as a set of *constraints* and a set of *variables* with possible values. The possible values of a variable is called the *domain* of the variable. All the variables are to be assigned a value from their domains, so that all the constraints of the problem are satisfied. Sometimes the solution should not only satisfy the set of constraints for the problem, but should also maximise or minimise some given function.

A constraint solver (CP solver) is a software that solves constraint problems. The solving of a problem consists of generating a search tree by branching on possible values for the variables. At each node in the search tree, the solver removes impossible values from the domains of the

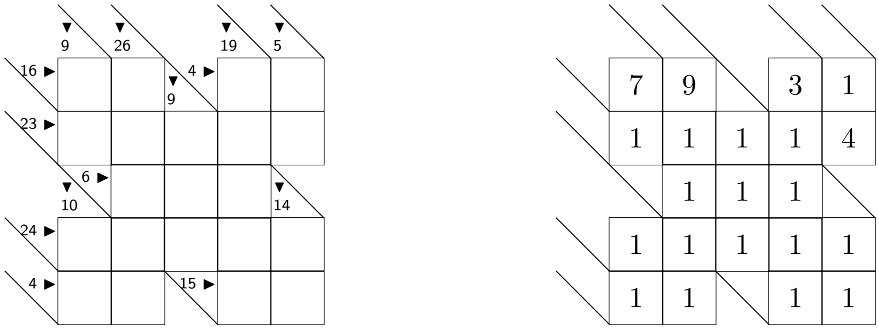

Figure 1: A Kakuro puzzle [1](left) and its solution (right).

variables. This filtering process is called *propagation*. Each constraint is associated with at least one propagator algorithm, whose task is to detect values that would violate the constraint if the variables were to be assigned any of those values, and remove those values from the domain of the variables.

To build intuition and understand the ideas of CP, the concepts can successfully be demonstrated with logical puzzles. One such puzzle is Kakuro, a kind of mathematical crossword where the "words" consist of numbers instead of letters, see Figure **??**. The game board consists of blank cells forming rows and columns, called *entries*. Each entry has a *clue*, a prefilled number indicating the sum of that entry. The size of the board can vary. The objective is to fill in digits from 1 to 9 inclusive into each cell such that for each entry, the sum of all the digits in the entry is equal to the clue of that entry, and such that each digit appears at most once in each entry.

A Kakuro puzzle can be modelled as a constraint satisfaction problem with one variable for each cell, and the domain of each variable being the set $\{1, .., 9\}$. The constraints of the problem is that the sum of the variables that belong to a given entry must be equal to the clue for that entry, and the values of the variables for each entry must be distinct.

An alternative way of phrasing the constraints of Kakuro, is to for each entry explicitly list all the possible combinations of values that the variables in that entry can take. For example, consider an entry of size 2 with clue 4. The only possible combinations of values are $\langle 1, 3\rangle$ and $\langle 3, 1\rangle$, since these are the only tuples of 2 distinct digits whose sums are equal to 4. This way of listing the possible combinations of values for the variables is in essence the Table constraint – the constraint that is addressed in this thesis.

Really solve the Kakuro in Figure **??**. Kvllspyssel!

After gaining some intuition of CP, now follows some formal definitions. The definitions are based on [5], [1], and [6].

Definition 1. ***Constraint.*** *Consider a finite sequence of* n *variables* $X = x_1, \dots, x_n$*, and a corresponding sequence of* domains $D = D_1, \dots, D_n$*, that are possible values for the respective variable. For a variable* $x_i \in X$*, its domain* D_i *is denoted by* $dom(x_i)$*.*

- *A* constraint c *on* X *is a relation, denoted by* $rel(c)$*. The associated variables* X *are denoted* $vars(c)$*, and we call* $|vars(c)|$ *the* arity *of* c*. The relation* $rel(c)$ *contains the set of* n*-tuples that are allowed for* X*, we call those* n*-tuples* solutions *to the constraint* c*.*

- *For an* n*-tuple* $\tau = \langle a_1, \dots, a_n\rangle$ *associated with* X*, we denote the ith value of* τ *by* $\tau[i]$ *or* $\tau[x_i]$*. The tuple* τ *is* valid *for* X *if and only if each value of* τ *is in the domain of the*

[1]From *200 Crazy Clever Kakuro Puzzles - Volume 2*, LeCompte, Dave, 2010.

corresponding variable: $\forall i \in 1 \ldots n, \tau[i] \in dom(x_i)$*, or equivalently,* $\tau \in D_1 \times \ldots \times D_n$*.*

- *An n-tuple* τ *is a* support *on a n-ary constraint* c *if and only if* τ *is valid for* $vars(c)$ *and* τ *is a solution to* c*, that is,* τ *is contained in* $rel(c)$*.*

- *For an n-ary constraint* c*, involving a variable* x *such that the value* $a \in dom(x)$*, an n-tuple* τ *is a* support for (x, a) *on* c *if and only if* τ *is a support on* c*, and* $\tau[x] = a$*.*

Definition 2. ***CSP.*** *A Constraint Satisfaction Problem (CSP) is a triple* $\langle V, D, C\rangle$*, where:* $V = v_1, \ldots, v_n$ *is a finite sequence of variables,* $D = D_1, \ldots, D_n$ *is a finite sequence of domains for the respective variable, and* $C = \{c_1, \ldots, c_m\}$ *is a set of constraints, each on a subsequence of* V*.*

Definition 3. ***Stores.*** *A* store s *is a function, mapping a finite set of variables* $V = v_1, \ldots, v_n$ *to a finite set of domains. We denote the domain of a variable* v_i *under* s *by* $s(v_i)$ *or* $dom(v_i)$*.*

- *A store* s *is* failed *if and only if* $s(v_i) = \varnothing$ *for some* $v_i \in V$*.*

- *A variable* $v_i \in V$ *is* fixed, *or* assigned, *by a store* s *if and only if* $|s(v_i)| = 1$*.*

- *A store* s *is an* assignment *store if all variables are fixed under* s*.*

- *Let* c *be an n-ary constraint on* V*. A store* s *is a* solution store *to* c *if and only if* s *is an assignment store and the corresponding n-tuple is a solution to c:* $\forall i \in \{1, \ldots, n\}, s(v_i) = \{a_i\}$*, and* $\langle a_1, \ldots, a_n\rangle$ *is a solution to* c*.*

- *A store* s_1 *is* stronger *than a store* s_2*, written* $s_1 \preceq s_2$ *if and only if* $s_1(v) \subseteq s_2(v)$ *for all* $v \in V$*.*

2.2 Propagation and propagators

Constraint propagation is the process of removing values from the domains of the variables in a CSP that can never appear in a solution store to the CSP. In a CP solver, each constraint that the solver implements is associated with one or more propagation algorithms, called propagators, whose task is to remove values that are in conflict with its respective constraint.

To have a well-defined behaviour of propagators, there are some properties that they must fulfill. Now follows a definition of a propagator and the obligations that they must meet, taken from [5] and [6].

Definition 4. ***Propagators.*** *A* propagator p *is a function mapping stores to stores:*

$$p : store \mapsto store$$

In a CP-solver, a propagator is implemented as a procedure that also returns a status message*. A propagator must fulfill the following properties:*

- *A propagator* p *is a decreasing function:* $p(s) \preceq s$ *for any store* s*. This property guarantees that constraint propagation only removes values.*

- *A propagator* p *is a monotonic function:* $s_1 \preceq s_2 \Rightarrow p(s_1) \preceq p(s_2)$ *for any stores* s_1 *and* s_2*. This property guarantees that constraint propagation preserves the strength-ordering of stores.*

- *A propagator is correct for the constraint it implements. A propagator* p *is correct for a constraint* c *if and only if it does not remove values that are part of supports for* c*. This property guarantees that a propagator does not miss any potential solution store.*

- *A propagator is* checking*: for a given assignment store s, the propagator must decide whether s is a solution store or not for the constraint it implements. If s is a solution store, it must signal subsumption, otherwise it must signal failure.*
- *A propagator must be* honest*: it must be* fixpoint honest *and* subsumption honest*. A propagator p is fixpoint honest if and only if it does not signal fixpoint if it does not return a fixpoint, and it is subsumption honest if and only if it does not signal subsumption if it is not subsumed by an input store s.*

  A propagator p is at fixpoint *on a store s if and only if applying p to to s gives no further propagation: $p(s) = s$ for a store s. If a propagator p always returns a fixpoint, that is, if $p(s) = p(p(s))$, p is* idempotent.

  A propagator is subsumed *by a store s if and only if all stronger stores are fixpoints: $\forall s' \preceq s, p(s') = s$.*

Note that the honest property of a propagator does not mean that a propagator is obliged to signal fixpoint or subsumption if it has computed a fixpoint or is subsumed, only that it must not claim that it is at fixpoint or is subsumed if it is not. Thus, it is always safe (though in many cases not so efficient for the CP-solver) for a propagator to signal 'not fixpoint', except for a solution store when it must signal either fail or subsumption. In fact, a propagator must not even prune values. An extreme case is the identity propagator i, with $i(s) = s$ for all input stores s, which would be correct for all constraints, as long at is it checking and honest.

To give a measure of how strong the constraint propagation of a propagator is, it is common to declare a *consistency level* of a propagator. There are three commonly used consistency levels, **value consistency, bounds consistency**, and **domain consistency**.

Definition 5. ***Domain consistency.*** *A constraint c is domain consistent on a store s if and only if for all variable-value pair (x, a) such that $x \in vars(c)$ and $a \in dom(x)$, there exists at least one support for (x, a) on c. A propagator p is domain consistent, iff c is domain consistent on $p(s)$ for all stores s such that $p(s)$ is not a failed store.*

Bounds consistency, Value consistency.

2.3 Gecode

Gecode [3] is a popular constraint programming solver written in C++.

Define the parts of the Gecode API that is used later (propagate(), status messages...)

2.4 The Table Constraint

The TABLE constraint, also called EXTENSIONAL , explicitly expresses the possible combinations of values for the variables as a sequence of n-tuples.

Definition 6. ***Table constraints.*** *A (positive[2])* table constraint c *is a constraint such that $rel(c)$ is defined explicitly by listing all the tuples that are solutions to c.*

[2]There are also negative table constraints, that list the forbidden tuples instead of the allowed tuples.

2.5 Compact-Table Propagator

2.6 Sparse Bit-Set

3 Algorithms

This chapter presents the algorithms that are used in the implementation of the CT propagator in Section 4. In what follows, when we refer to an array a, $a[0]$ denotes the first element (indexing starts from 0), a.length() the number of its cells and $a[i : j]$ all its cells in the closed interval $[i, j]$, where $0 \leq i \leq j \leq a.\text{length}() - 1$.

3.1 Sparse Bit-Set

This section describes Class `SparseBitSet`, which is the main data-structure in the CT algorithm for maintaining the supports. Algorithm 1 shows the pseudo code for Class `SparseBitSet`. The rest of this section describes its fields and methods in detail.

```
Class SparseBitSet

words: array of long                                              // words.length() = p
index: array of int                                               // index.length() = p
limit: int
mask: array of long                                               // mask.length() = p

Method initSparseBitSet(nbits: int)
  p ← ⌈nbits/64⌉
  words ← array of long of length p, first nbits set to 1
  mask ← array of long of lengthp, all bits set to 0
  index ← [0, . . . , p − 1]
  limit ← p − 1

Method isEmpty() : Boolean
  return limit = −1

Method clearMask()
  for i ← 0 to limit do
    offset ← index[i]
    mask[offset] ← 0^64

Method addToMask(m: array of long)
  for i ← 0 to limit do
    offset ← index[i]
    mask[offset] ← mask[offset] | m[offset]                       // bitwise OR

Method intersectWithMask()
  for i ← limit downto 0 do
    offset ← index[i]
    w ← words[offset] & mask[offset]                              // bitwise AND
    if w ≠ words[offset] then
      words[offset] ← w
      if w = 0^64 then
        index[i] ← index[limit]
        index[limit] ← offset
        limit ← limit − 1

Method intersectIndex(m: array of long) : int
  for i ← 0 to limit do
    offset ← index[i]
    if words[offset] & m[offset] ≠ 0^64 then
      return offset
  return −1
```

Algorithm 1: Pseudo code for Class SparseBitSet.

3.1.1 Fields

Todo: Add examples.

Lines 2-5 of Algorithm 1 shows the fields of Class `SparseBitSet` and their types. Now follows a more detailed description of them.

- `words` is an array of p 64-bit words which defines the current value of the bit-set: the ith bit of the jth word is 1 if and only if the $(j-1) \cdot 64 + i$th element of the set is present. Initially, all words in this array have all their bits set to 1, except the last word that may have a suffix of bits set to 0. Example.

- `index` is an array that manages the indices of the words in `words`, making it possible to performing operations to non-zero words only. In `index`, the indices of all the non-zero words are at positions less than or equal to the value of the field `limit`, and the indices of the zero-words are at indices strictly greater than `limit`.

- `limit` is the index of `index` corresponding to the last non-zero word in `words`. Thus it is one smaller than the number of non-zero words in `words`.

- `mask` is a local temporary array that is used to modify the bits in `words`.

The class invariant describing the state of the class is as follows:

$$\begin{aligned} &\texttt{index} \text{ is a permutation of } [0, \ldots, p-1], \text{ and} \\ &\forall i \in \{0, \ldots, p-1\} : i \leq \texttt{limit} \Leftrightarrow \texttt{words}[\texttt{index}[i]] \neq 0^{64} \end{aligned} \tag{3.1}$$

3.1.2 Methods

We now describe the methods in Class `SparseBitSet` in Algorithm 1.

- initSparseBitSet() in lines 6-11 initialises a sparse bit-set-object. It takes the number of bits as an argument and initialises the fields described in 3.1.1 in a straightforward way.

- isEmpty() in lines 12-13 checks if the number of non-zero words is different from zero. If the limit is set to -1, that means that all words are zero-words and the bit-set is empty.

- clearMask() in lines 14-17 clears the temporary mask. This means setting to 0 all words of `mask` corresponding to non-zero words of `words`.

- addToMask() in lines 18-21 collects elements to the temporary mask by applying a word-by-word logical bit-wise *or* operation with a given bit-set (array of long). Once again, this operation is only applied to indices corresponding to non-zero words in `words`.

- intersectWithMask() in lines 22-31 considers each non-zero word of `words` in turn and replaces it by its intersection with the corresponding word of `mask`. In case the resulting new word is 0, it (its index) is swapped with (the index of) the last non-zero word, and `limit` is decreased by one.

  In Section 4 we will see that the implementation actually can skip line 30 because it is unneccesary to save the index of a zero-word in a copy-based solver such as Gecode. We keep this line here though, because otherwise the invariant in (3.1) would not hold.

- intersectIndex() in lines 32-37 checks whether the intersection of `words` and a given bit-set (array of long) is empty or not. For all non-zero words in `words`, we perform a logical bit-wise *and* operation in line 35 and return the index of the word if the intersection is non-empty. If the intersection is empty for all words, -1 is returned.

3.2 Compact-Table (CT) Algorithm

This section describes the CT algorithm, a domain consistent propagation algorithm for any Table constraint c. Algorithm 2 shows the interface for Class CT-Propagator, which implements the CT algorithm. The rest of this section will describe its fields and methods in detail.

1: **Class** CT-Propagator

2: `vars`: array of variables
3: `validTuples`: SparseBitSet // Current valid tuples
4: `supports`: array of bit-sets // `supports`$[x, a]$ is the bit-set of supports for (x, a)
5: `residues`: array of int // `residues`$[x, a]$ is the last found index for a support for (x, a)

6: **Method** initCT(*variables*: array of variables, *tuples*: list of tuples)
7: Algorithm 3

8: **Method** updateTable(x: variable)
9: Algorithm 4

10: **Method** filterDomains()
11: Algorithm 5

12: **Method** propagate()
13: Algorithm 6

Algorithm 2: Interface for CT propagator class.

3.2.1 Fields

Add examples with figures for describing the fields.

Lines 2-5 of Algorithm 2 shows the fields of Class `CT-Propagator` and their types. Now follows a more detailed description of them. In what follows, we let the *initial domain* of a variable $x \in$ `vars`(c), denoted $\underline{\mathrm{dom}}(x)$, be the domain that x has before CT has performed any propagation, in contrast to $\mathrm{dom}(x)$ which is the current domain of x. The *initial table* for a table constraint c is the list of tuples $T_0 = \langle \tau_0, \tau_1, \ldots, \tau_{p_0-1} \rangle$ of length p_o that are given as input to initCT(), and the *initial valid table* for c is the subset $T \subseteq T_0$ of size $p \leq p_0$ such that $\forall i \in \{1, \ldots, p_0\} : \tau_i \in T$ iff τ_i is a support on c for the initial domains of the variables.

- `vars` represents $vars(c)$, the variables associated with c.

- `validTuples` represents the current table, that is, the current valid supports for c. If the initial valid table for c is $\langle \tau_0, \tau_1, \ldots, \tau_{p-1} \rangle$, then `validTuples` is a `SparseBitSet` object of initial size p, such that value i is contained (is set to 1) if and only if the ith tuple is valid:

$$i \in \texttt{validTuples} \Leftrightarrow \forall x \in vars(c) : \tau_i[x] \in \mathrm{dom}(x) \tag{3.2}$$

- `supports` represents the supports for each variable-value pair (x, a), where $x \in vars(c) \wedge a \in \underline{\mathrm{dom}}(x)$. It is a static array of words `supports`$[x, a]$, seen as bit-sets. The bit-set `supports`$[x, a]$ is such that the bit at position i is set to 1 if and only if the tuple τ_i in the initial valid table of c is initially a support for (x, a):

$$\forall x \in vars(c) : \ \forall a \in \underline{\mathrm{dom}}(x) : \\ \texttt{supports}[x,a][i] = 1 \quad \Leftrightarrow \\ (\tau_i[x] = a \quad \wedge \quad \forall y \in vars(c) : \tau_i[y] \in \underline{\mathrm{dom}}(y))$$

`supports` is computed once during the initialisation of CT and then remains unchanged.

- `residues` is an array such that for each variable-value pair (x, a), $\texttt{residues}[x, a]$ denotes the index of the word in `validTuples` where a support was found for (x, a) the last time it was sought for.

3.2.2 Methods

We now describe the methods of Class `CT-Propagator`.

Initialisation. The initialisation of the fields of Class `CT-Propagator` is described in Algorithm 3. The method initialiseCT() takes two parameters: *variables*, that are the variables associated to the constraint c, and *tuples*, that is a list of tuples that define the initial table for c.

```
Method initCT(variables: array of variables, tuples: list of tuples)
  foreach x ∈ variables do
    dom(x) ← dom(x) \ {a ∈ dom(x) : a > tuples.max() or a < tuples.min()}
    if dom(x) = ∅ then
      return Failed
  npairs ← sum {|dom(x)| : x ∈ variables}          // Number of variable-value pairs
  ntuples ← tuples.size()                           // Number of tuples
  nsupports ← 0                                     // Number of found supports
  vars ← variables
  residues ← array of length npairs
  supports ← array of length npairs with bit-sets of size ntuples
  foreach t ∈ tuples do
    supported ← true
    foreach x ∈ vars do
      if t[x] ∉ dom(x) then
        supported ← false
        break                                       // Exit loop
    if supported then
      foreach x ∈ vars do
        supports[x, t[x]][nsupports] ← 1
        residues[x, t[x]] ← ⌊nsupports / 64⌋        // Index for the support in validTuples
      nsupports ← nsupports + 1
  foreach x ∈ vars do
    dom(x) ← dom(x) \ {a ∈ dom(x) : supports[x, a] is empty}
    if dom(x) = ∅ then
      return Failed
  validTuples ← SparseBitSet with nsupports bits
  return Fixpoint
```

Algorithm 3: Pseudo code for initialising the CT-propagator.

Lines 2-5 performs simple bounds propagation to limit the domain sizes of the variables, which limit the sizes of the incremental data structures. It removes from the domain of each variable x all values that are either greater than the largest element or smaller than the smallest element in the initial table. If a variable has a domain wipe-out, *Failed* is returned.

Lines 6-8 initialise local variables that will be used later.

Lines 9-11 initialise the fields `vars`, `residues` and `supports`. The field `supports` is initialised as an array of bit-sets, with one bit-set for each variable-value pair, and the size of each bit-set being the number of tuples in *tuples*. Each bit-set is assumed to be initially filled with zeros.

Lines 12-22 set the correct bits to 1 in `supports`. For each tuple t, we check if t is a valid support for c. Recall that t is a valid support for c if and only if $t[x] \in \mathrm{dom}(x)$ for all $x \in scp(c)$. We keep a counter, *nsupports*, for the number of valid supports for c. This is used for indexing the tuples in `supports` (we only index the tuples that are valid supports). If t is a valid support, all elements in `supports` corresponding to t are set to 1 in line 20. We also take the opportunity to store the word index of the found support in `residues`$[x, t[x]]$ in line 21.

Lines 23-24 removes values that are not supported by any tuple in the initial valid table. In case a variable has a wipe-out of its domain, *Failed* is returned.

Line 27 initialises `validTuples` as a `SparseBitSet` object with *nsupports* bits, initially with all bits set to 1 since *nsupports* number of tuples are initially valid supports for c. $nsupports > 0$, otherwise we would have returned *Failed* as no variable-value pair would be supported.

Performing propagation. When the propagator is invoked for propagation, the method propagate() in Algorithm 2 is called. Before defining this function, we need to define the help functions updateTable() and filterDomains(). Performing propagation consists of two steps: updating the current table and filtering out inconsistent values from the domains of the variables. We now describe these processes in more detail.

1. *Updating the current table.*

```
1: Method updateTable(x: variable)
2:   validTuples.clearMask()
3:   foreach a ∈ dom(x) do
4:     validTuples.addToMask(supports[x, a])
5:   validTuples.intersectWithMask()
6:   if validTuples.isEmpty() then
7:     return Failed                                  // No valid tuples left
```

Algorithm 4: Method updateTable() in Class CT-Propagator. The infrastructure is such that this method is called for each variable whose domain is modified since the previous call to propagate().

The method updateTable() in Algorithm 4 filters out (indices of) tuples that have ceased to be supports for the input variable x. Lines 3-4 stores the union of the set of valid tuples for each value $a \in \mathrm{dom}(x)$ in the temporary mask and Line 5 intersects `validTuples` with the mask, so that the indices that correspond to tuples that are no longer valid are set to 0 in the bit-set. Line 6 checks whether the current table is empty, in which case we return *Failed* in line 7 because there are no valid tuples left.

The algorithm is assumed to be run on an infrastructure that runs updateTable() before every call to propagate(), for each variable $x \in vars(c)$ whose domain has changed since the previous call to propagate().

2. *Filtering of domains.* After the current table has been updated, inconsistent values must be removed from the domains of the variables. It follows from the definition of the bitsets `validTuples` and $\texttt{supports}[x, a]$ that (x, a) has a valid support if and only if

$$(\texttt{validTuples} \cap \texttt{supports}[x, a]) \neq \emptyset \tag{3.3}$$

Therefore, we must check this condition for every variable-value pair (x, a) and remove a from the domain of x if the condition is not satisfied any more. This is implemented in the method filterDomains() in Algorithm 5.

```
1:  Method filterDomains()
2:    count_unassigned ← 0
3:    foreach x ∈ vars such that |dom(x)| > 1 do
4:      foreach a ∈ dom(x) do
5:        index ← residues[x, a]
6:        if validTuples[index] & supports[x, a][index] = 0 then
7:          index ← validTuples.intersectIndex(supports[x, a])
8:          if index ≠ −1 then
9:            residues[x, a] ← index
10:         else
11:           dom(x) ← dom(x) \ {a}
12:     if |dom(x)| > 1 then
13:       count_unassigned ← count_unassigned + 1
14:   if count_unassigned ≤ 1 then
15:     return Subsumed
16:   else
17:     return Fixpoint
```

Algorithm 5: Method filterDomains() in Class CT-Propagator. The infrastructure is such that this method is called after updateTable() in Algorithm 4 has been called for each variable whose domain is modified, and only if at least one variable x

Line 2 initialises a counter for the number of unassigned variables.

Lines 3-13 performs the actual filtering of the domains. We note that it is only necessary to consider a variable $x \in \texttt{vars}$ whose domain size is larger than 1, because we will never filter out values from the domain of an assigned variable. To see this, assume we removed the last value for a variable x, causing a wipe-out for x. Then by the definition in equation (3.2) `validTuples` must be empty, which it will not be upon invocation of filterDomains(). Hence, we need only consider $x \in \texttt{vars}$ such that $|\text{dom}(x) > 1|$.

In Lines 5-6 we see if the cached word index still has a support for (x, a). It it has not, we we search for an index in line 7 in `validTuples` where a valid support for the variable-value pair (x, a) is found, thereby checking the condition in (3.3). If such an index exists, we cache it in $\texttt{residues}[x, a]$, and if it does not, we remove a from $\text{dom}(x)$ if (x, a) in line 11 since there is no support left for (x, a).

Lines 12-13 increments the counter of unassigned variables if $|\text{dom}(x)| > 1$.

Lines 14-15 return the correct propagator status message. If the number of unassigned variables is at most one, the propagator is subsumed. Otherwise, the propagator is at fixpoint.

After defining updateTable() and filterDomains(), we are now ready to define the method propagate() in Class CT-Propagator, shown in Algorithm 6.

It should be unneccessary to check if validTuples is empty as that is done in updateTable already. However, when I try to remove the check in the c++ code it crashes, maybe because of synchronisation issues between advise() and propagate().

```
Method propagate()
  if validTuples.isEmpty() then
    return Failed
  return filterDomains()
```

Algorithm 6: Method propagate() in Class CT-Propagator. updateTable() (Algorithm 4) is called, and if the current table is empty, we are in a failed node. Otherwise, filterDomains() (Algorithm 5) is called, and the return value of that method is returned.

Optimisations. If x is the only variable that has been modified since the last invocation of CT, it is not necessary to attempt to filter out values from x, because every value of of x will have a support in `validTuples`. Hence, in Algorithm 5, we only execute Lines 4-11 for a variable x if x was not the only modified variable.

3.2.3 Proof of properties for CT

This section proves that the CT Propagator is indeed a well-defined propagator implementing the Table constraint. We formulate the following theorem, which we will prove by a number of lemmas.

Theorem 3.1. *CT is an idempotent, domain consistent propagator implementing the* Table *constraint, fulfilling the properties in Definition 4.*

To prove Theorem 3.1, we formulate and prove the following lemmas. In what follows, we denote $CT(s)$ the resulting store of running either `initCT()` or `propagate()` on an input store s, depending on if it is the first time or not that the propagator is called.

Lemma 3.2. *CT is a decreasing function.*

Proof of Lemma 3.2. Since CT only removes values from the domains of the variables, we have $CT(s) \preceq s$ for any store s. Thus, CT is a decreasing function. □

Lemma 3.3. *CT is idempotent.*

Proof of Lemma 3.3. To prove that CT is idempotent, we shall show that CT always reaches fixpoint for any input store s, that is, $CT(CT(s)) = CT(s)$ for any store s.

Suppose $CT(CT(s)) \neq CT(s)$ for a store s. Since CT is monotonic and decreasing, we must have $CT(CT(s)) \prec CT(s)$, that is, CT must prune at least one value a from a variable x from the store $CT(s)$.

By (3.3), there must exists at least one tuple τ_i that is a support for (x, a) under the store $CT(s)$: $\exists i : i \in$ `validTuples` $\wedge\ \tau_i[x] = a$. After `updateTable()` is perfomed on $CT(s)$, we still have $i \in$ `validTuples`, because τ_i is still valid in $CT(s)$. Since `filterDomains()` only removes values that have no supports, it is impossible that a is pruned from x, since τ_i is a support for (x, a). Hence, we must have $CT(CT(s)) = CT(s)$. □

Lemma 3.4. *CT is correct for the* Table *constraint.*

Proof of Lemma 3.4. CT does not remove values that participate in tuples that are supports on a TABLE constratint c, since `filterDomains()` and `initCT()` only removes values that have no supports on c. Thus, CT is correct for TABLE . □

Lemma 3.5. *CT is checking.*

Proof of Lemma 3.5. For an input store s that is an assignment store, we shall show that CT signals failure if s is not a solution store, and signals subsumption if s is a solution store.

First, assume that s is not a solution store. That means that the tuple $\tau = \langle s(x_1), \ldots, s(s_n) \rangle \notin c$.

There are two cases, either it is the first time CT is applied or it has been applied before. If it is the first time, then `initCT()` is called. Since τ is not a solution to c, there is at least one variable-value pair $(x_i, s(x_i))$ which is not supported, so $s(x_i)$ will be pruned from x in `initCT()`, which will report failure in line Line **??** in Algorithm 3.

If it is not the first time that CT is called, `propagate()` is called. Since there are no valid tuples left, `validTuples` will be empty after the call to `updateTable()` and CT reports failure.

Now assume that s is a solution store. CT signals failure in `filterDomains()` because all variables are assigned. Initialisation? Maybe change so that initCT detects subsumption. □

Lemma 3.6. *CT is honest.*

Proof of Lemma 3.6. Since CT is idempotent, CT is fixpoint honest. It remains to show that CT is subsumption honest. CT signals subsumption on input store s if there is at most one unassigned variable x in `filterDomains()`. After this point, no values will ever be pruned from x by CT, because there will always be a support for (x, a) for each value $a \in dom(x)$. Hence, CT is indeed subsumed by s when it signals subsumption.

□

Lemma 3.7. *CT is domain consistent.*

Proof of Lemma 3.7. There are two cases; either it is the first time CT is called, or it is not. Both after a call to `initCT()` and `filterDomains()`, for each variable-value pair (x, a) there exists at least one support, because we filter out those values that have no support. □

Lemma 3.8. *CT is a monotonic function.*

Proof of Lemma 3.8. Consider two stores s_1 and s_2 such that $s_1 \preceq s_2$. Since CT is domain consistent, each variable-value pair (x, a) that is part of $CT(s_1)$, must also be part of $CT(s_2)$, so $CT(s_1) \preceq CT(s_2)$. □

After proving Lemmas 3.2-3.8, proving Theorem 3.1 is trivial.

Proof of Theorem 3.1. The result follows by Lemmas 3.2- 3.8. □

4 Implementation

Comparison with implementation in OR-tools Copy-function.

This section describes an implementation of the CT propagator using the algorithms presented in Section 3. The implementation was made in the C++ programming language in the Gecode library.

The bit-set matrix `supports` is static and could be shared between all solution spaces.

The bit-set `validTuples` changes dynamically during propagation and must therefore be copied for every new space. Can save memory by only copying the non-zero words.

No need to save the `tuples` as a field in the propagator class as all the necessary information is encoded in `validTuples` and `supports`.

How is the index mapping done in supports and residues?

5 Evaluation

This chapter presents the evaluation of the implementation of the CT propagator presented in Section 4. In Section 5.1, the evaluation setup is described. In Section 5.2 presents the results of the evaluation. The results are discussed in Section 5.3.

5.1 Evaluation Setup

The correctness of the CT propagator was validated with the existing unit tests in Gecode for the table constrataint.

A large number of benchmarks were run, comparing the performance of the CT propagator with the three other existing propagators in Gecode for the table constraint. The instances used in the benchmarks were written in MiniZinc [4], a solver-independent constraint modeling language. The experiments were run under Gecode 5.0 on a 16-core machine with Linux Ubuntu 14.04.5 (64 bit), Intel Xeon Core of 2.27 GHz, with 25 GB RAM and 8 MB L3 cache. The machines were accessed via a shared server.

The following section presents the results of the experiments.

5.2 Results

5.2.1 Comparing CT and CT(fix)

The main purpose of this benchmark is to compare the performance of CT and CT(fix). Figure 5.2.1 shows that the performance of CT and CT(fix) are the same. CT and CT(fix) outperform all other methods. Among the other methods, the performance is essentially the same.

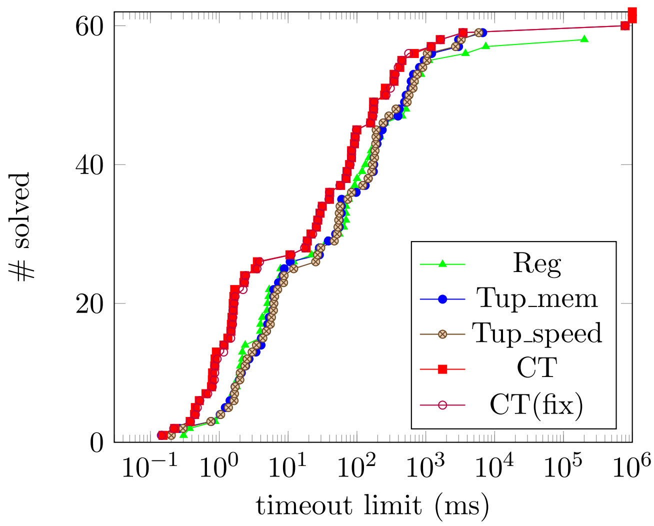

Figure 2: **Crosswords.** Number of solved instances as a function of time for 5 methods. The benchmark consists of 72 instances, using a timeout of 1000 seconds. Within the timeout, CT and CT(fix) could solve 60 instances, Tup_mem , and Tup_speed could solve 59 instances, and Reg could solve 58 instances.

5.2.2 Benchmarks from [2]

This set of benchmarks contains 1,621 CSP instances that where used in the experiments in [2], written in XCSP2.1. This corresponds to a large variety of instances, taken from 37 series. 117 instances were skipped due to parse problems. Due to the high number of instances, the runtime for each instance was only measured once. A timeout of 1000 seconds was used.

Add more graphs and captions.

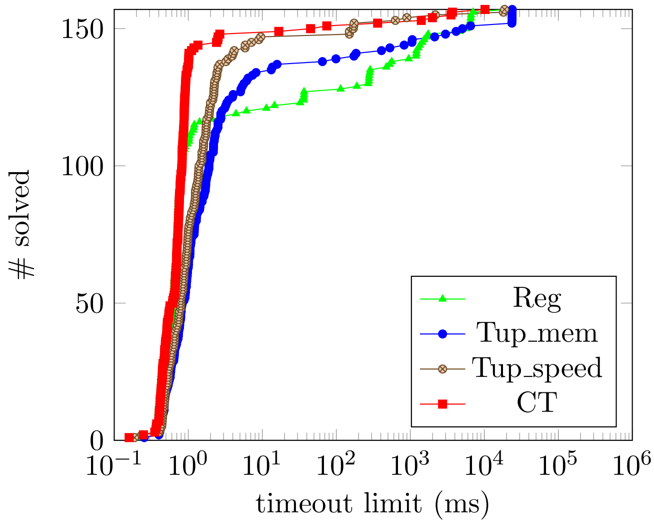

Figure 3: **Kakuro easy**.

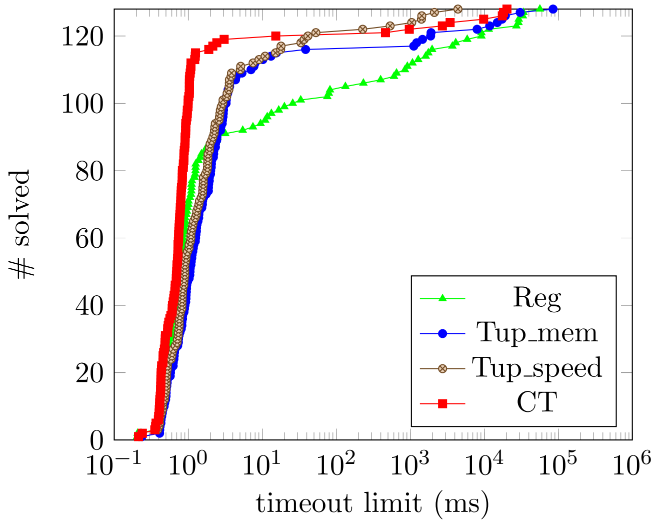

Figure 4: **Kakuro medium**.

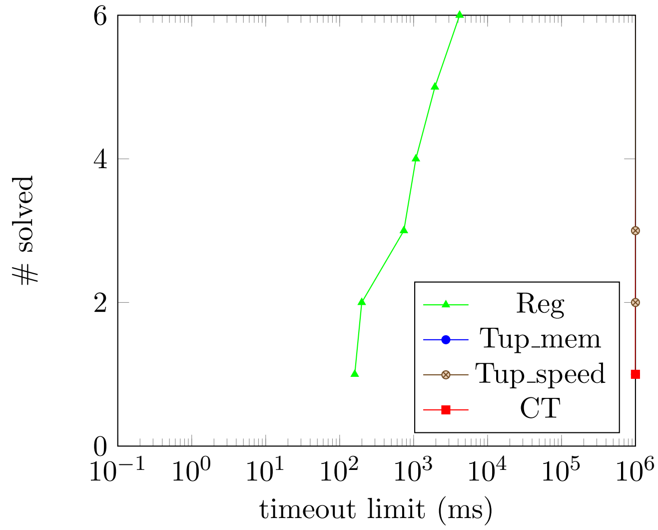

Figure 5: **MDD09**.

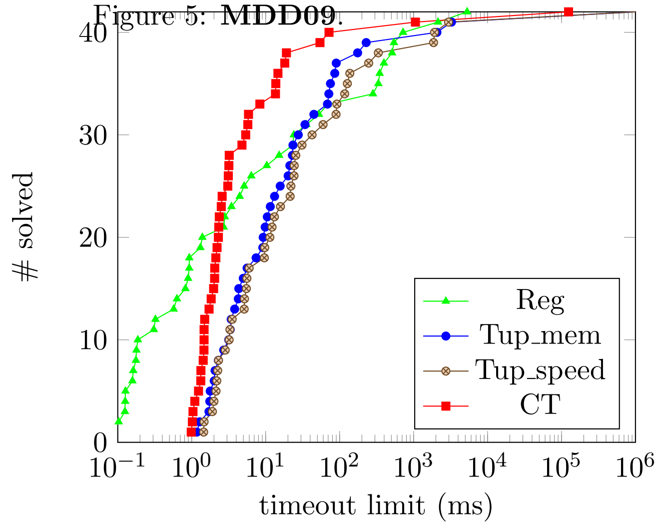

Figure 6: **Nonograms**.

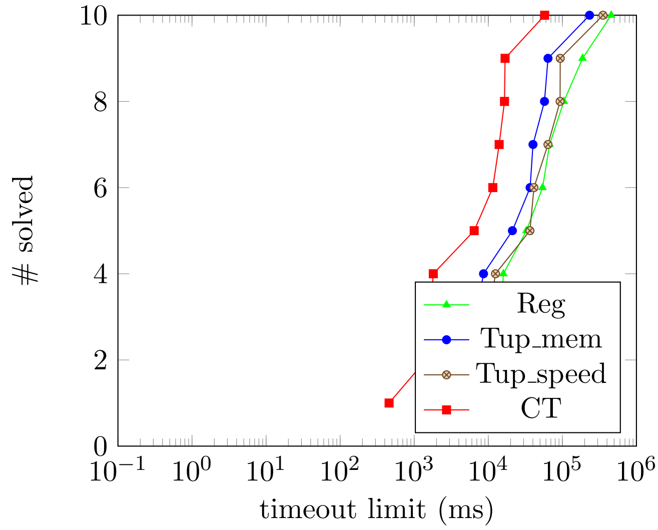

Figure 7: **RandJC2500**.

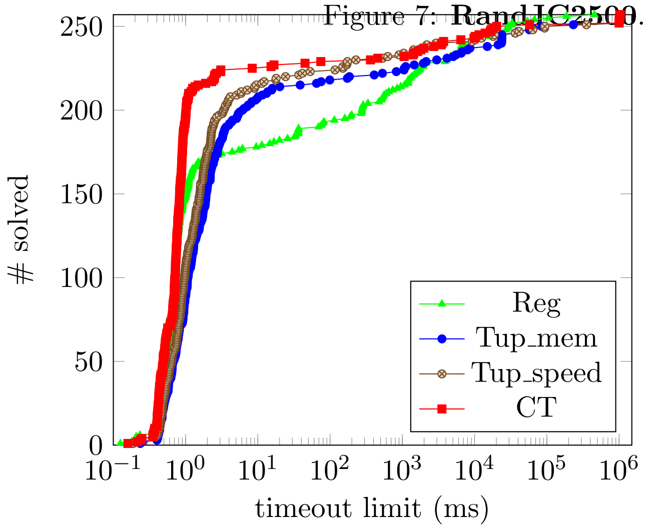

Figure 8: All instances.

5.3 Discussion

6 Conclusions and Future Work

References

[1] K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[2] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J. Régin, and P. Schaus. Compact-Table: Efficiently filtering table constraints with reversible sparse bit-sets. In M. Rueher, editor, *CP*, volume 9892 of *Lecture Notes in Computer Science*, pages 207–223. Springer, 2016.

[3] Gecode Team. Gecode: A generic constraint development environment, 2016.

[4] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In C. Bessière, editor, *CP 2007*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007. the MiniZinc toolchain is available at `http://www.minizinc.org`.

[5] C. Schulte and M. Carlsson. Finite domain constraint programming systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 14, pages 495–526. Elsevier, 2006.

[6] C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling and programming with gecode, 2017. Available at `http://www.gecode.org/doc-latest/MPG.pdf`.

A Source Code

This appendix presents the source code for the implementation described in Section 4.