Implementation and Evaluation of a Compact Table Propagator in Gecode

Linnea Ingmar

17th March 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 [1], 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 [2], and consequently its performance in Gecode is unknown.

1.1 Goal

The goal of this thesis is to implement a CT progatator 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 [1] 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 2.2 gives an overview of Gecode, a constraint solver. Section 2.3 introduces the Table constraint. Section 2.4 describes the main concepts of the Compact Table (CT) algorithm. Finally, Section 2.5 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.

Perhaps add an example somewhere here.

A constraint 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 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.

After this intuitive description we are ready to define some concepts.

Todo: Define concepts

- CSP
- Valid tuple for a variable
- Tuples that are supports for a constraint
- Value-, bounds-, and domain-consistency
- Constraint store

Propagation A propagator p is a function from constraint stores to constraint stores that performs constraint propagation.

Define what a propagator must fulfill. (Contracting, monotonic...)

2.2 Gecode

Gecode [2] is a popular constraint programming solver written in C++.

2.3 The Table Constraint

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

2.4 Compact-Table Propagator

2.5 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 $0 \leq i \leq j \leq a.\text{length}() - 1$. When we refer to a two-dimensional array m, $m[i][*]$ denotes row i and $m[*][j]$ column j, seing m as a matrix.

3.1 Sparse Bit-Set

This section describes Class `SparseBitSet`, which is the main datastructure 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.

```
1:  Class SparseBitSet

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

6:  Method isEmpty() : Boolean
7:    return limit = −1

8:  Method clearMask()
9:    for i ← 0 to limit do
10:     offset ← index[i]
11:     mask[offset] ← 0^64

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

16: Method intersectWithMask()
17:   for i ← limit downto 0 do
18:     offset ← index[i]
19:     w ← words[offset] & mask[offset]                   // bitwise AND
20:     if w ≠ words[offset] then
21:       words[offset] ← w
22:       if w = 0^64 then
23:         index[i] ← index[limit]
24:         limit ← limit − 1

25: Method intersectIndex(m: array of long) : int
26:   for i ← 0 to limit do
27:     offset ← index[i]
28:     if words[offset] & m[offset] ≠ 0^64 then
29:       return offset
30:   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`. The indices of all the

non-zero words are in `index` at positions less than or equal to the value of the field `limit`. The indices of the zero-words will be overwritten by indices of non-zero words because only non-zero words are relevant. In the paper, they switch place of the indices of the zero-word and the non-zero word. But since we implement this in Gecode, which is copy-based, we can overwrite it instead. Explain this difference?

- `limit` is 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:

$$\texttt{index}[0:\texttt{limit}] \text{ is a permutation of a subset of } [0,\ldots,p-1], \text{ and}$$
$$\forall i \in \{0,\ldots,\texttt{limit}\} : \texttt{words}[\texttt{index}[i]] \neq 0^{64}$$

In the paper, the invariant looks like this:

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

but for us it is different because we overwrite the indices of the zero-words.

3.1.2 Methods

Add method initialiseSparseBitSet() to show the initialisation of the object?

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

- isEmpty() in lines 6-7 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.
- clearMask() in lines 8-11 clears the temporary mask. This means setting to 0 all words of `mask` corresponding to non-zero words of `words`.
- addToMask() in lines 12-15 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 16-24 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 replaced by (the index of) the last non-zero word, and `limit` is decreased by one.
- intersectIndex() in lines 25-30. checks whether the instersection 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 28 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. Is this a good idea or is the pseudo code too spread out?

```
1: Class CT-Propagator

2: scp: array of variables
3: currTable: SparseBitSet                                   // Current supported tuples
4: supports: array of bit-sets          // supports[x, a] is the bit-set of supports for (x, a)
5: lastSize: array of int                // lastSize[x] is the last size of the domain of x.c
6: residues: array of int   // residues[x, a] is the last found support for (x, a). No residues
   yet!

7: Method initialiseCT()
8:    Algorithm 3

9: Method updateTable()
10:   Algorithm 5

11: Method filterDomains()
12:   Algorithm 6

13: Method propagate()
14:   Algorithm 7
```

Algorithm 2: Interface for CT propagator class.

3.2.1 Fields

Add examples with figures for describing the fields.

Lines 2-6 of Algorithm 2 shows the fields of Class `CT-Propagator` and their types. Now follows a more detailed description of them.

- `scp` represents the scope of the constraint c.
- `currTable` represents the current table, that is, the valid supports for c. If the initial table of c is $\langle \tau_0, \tau_1, ..., \tau_{p-1} \rangle$, then `currTable` 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:

$$i \in \texttt{currTable} \Leftrightarrow \forall x \in scp(c) : \tau_i[x] \in \text{dom}(x) \tag{3.1}$$

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

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

  Seing `supports` as a matrix, we have that the column $\texttt{supports}[*][i]$ encodes the ith support for c. `supports` is computed once during the initialisation of CT and then remains unchanged.

- `lastSize` is an array that contains the domain size of each variable x right before the previous invocation of CT on c.

- `residues` is an array such that for each variable-value pari (x, a), `residues`$[x, a]$ denotes the index of the word in `currTable` 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.

```
1:  Method initialiseCT(variables, tuples)
2:    npairs ← sum {|dom(x)| : x ∈ variables}          // Number of variable-value pairs
3:    ntuples ← tuples.size()                                    // Number of tuples
4:    nsupports ← 0                                    // Number of found supports
5:    scp ← variables
6:    lastSize ← array of length scp.length filled with -1                // Dummy value
7:    residues ← array of length npairs
8:    supports ← array of length npairs with bit-sets of size ntuples
9:    foreach t ∈ tuples do
10:     supported ← true
11:     foreach x ∈ scp do
12:       if t[x] ∉ dom(x) then
13:         supported ← false
14:         break                                                    // Exit loop
15:     if supported then
16:       foreach x ∈ scp do
17:         supports[x, t[x]][nsupports] ← 1
18:         residues[x, t[x]] ← ⌊nsupports/64⌋          // Index for the support in currTable
19:         nsupports ← nsupports + 1
20:   foreach x ∈ scp do
21:     dom(x) ← dom(x) \ {a ∈ dom(x) : supports[x, a] = 0}
22:   currTable ← SparseBitSet with nsupports bits
```

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

Lines 2-4 initialise local variables that will be used later.

Lines 5-8 initialise the fields `scp` , `lastSize`, `residues` and `supports`. The array `lastSize` is filled with the dummy value -1 because we want to have $|\text{dom}(x)| \neq$ `lastSize`$[x]$ the first time that the propagation algorithm runs. The field `residues` will be filled with values later, in line 18. The field `supports` is initalised 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 9-19 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 $t[x] \in \text{dom}(x)$ 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 17. We also take the opportunity

to store the index of the found support in `residues`$[x, t[x]]$ in line 18. Perhaps break out lines 9-19 to an own algorithm-environment? I haven't implemented residues yet (to keep the first version as simple as possible) but this is how I would do it.

Line **??** initialises `currTable` 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. We assume that `SparseBitSet` has a constructor that takes nr of bits as arguments.

Lines 20-21 removes values that are not supported by any tuple in the initial table. Optimisation, not neccassary as these values will be removed upon the first propagation otherwise.

To limit the size of `supports`, one could perform simple bounds propagation in initialiseCT(). Algorithm 4 is inserted in the beginning of initialiseCT(), in line 2 in Algorithm 3. 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. This is not neccassary, only an optimisation. Might change later to take the intersection of the values in the tupleset with the domain of the variables, but this is cheaper since it is very cheap to find min and max for tuplesets.

```
1: foreach x ∈ variables do
2:     dom(x) ← dom(x) \ {a ∈ dom(x) : a > tuples.max() or a < tuples.min()}
```

Algorithm 4: Simple initial propagation for keeping down the size of `supports`, to be inserted in the beginning of initialiseCT() in Algorithm 3.

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()
2:   foreach x ∈ scp such that |dom(x)| ≠ lastSize[x] do
3:     lastSize[x] ← |dom(x)|
4:     currTable.clearMask()
5:     foreach a ∈ dom(x) do
6:       currTable.addToMask(supports[x, a])
7:     currTable.intersectWithMask()
8:     if currTable.isEmpty() then
9:       break                                   // No valid tuples left
```

Algorithm 5: Method updateTable() in Class CT-Propagator.

The method updateTable() in Lines 1-9 of Algorithm 2 filters out (indices of) tuples that have ceased to be supports since the last invocation of the propagator. The ovarall idea is that for each variable $x \in$ `scp`, we keep only those values in `currTable` that correspond to valid tuples for x, and the complement of those values are set to 0. We note that it is only necessary to consider a variable x if its domain has changed since the last invocation of the propagator, because only then may the set of valid tuples have changed. Line 2 chooses exactly those variables.

Line 3 records the domain size for x in `lastSize`$[x]$ and line 4 clears the temporary mask (sets all bits to 0).

Lines 5-6 stores the union of the set of valid tuples for each value $a \in \text{dom}(x)$ in the temporary mask and Line **??** intersects `currTable` with the mask. Line 8 checks whether the current table is empty, in which case we break in line **??** because there are no valid tuples left.

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 bit-sets `currTable` and `supports`$[x, a]$ that (x, a) has a valid support if and only if

$$(\texttt{currTable} \cap \texttt{supports}[x, a]) \neq \emptyset \tag{3.2}$$

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 6.

```
1:  Method filterDomains()
2:    count_unassigned ← 0
3:    foreach x ∈ scp such that |dom(x)| > 1 do
4:      foreach a ∈ dom(x) do
5:        index ← currTable.intersectIndex(supports[x, a])
6:        if index ≠ −1 then
7:                                                    // No residues yet!
8:        else
9:          dom(x) ← dom(x) \ {a}
10:         if |dom(x)| > 1 then
11:           count_unassigned ← count_unassigned + 1
12:   if count_unassigned ≤ 1 then
13:     return Subsumed
14:   else
15:     return Fixpoint
```

Algorithm 6: Method filterDomains() in Class CT-Propagator.

This method would be slightly more sofisticated if residues were implemented. For now, disregard the empty line in the if-statement.

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

Lines 3-11 performs the actual filtering of the domains. We note that it is only neccessary to consider a variable $x \in$ `scp` whose domain size is larger than 1, because we will never filter out values from the domain of an assigned varible. 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 (3.1) `currTable` must be empty, which it will not be upon invocation of filterDomains(). Hence, we need only consider $x \in$ `scp` such that $|\text{dom}(x) > 1|$.

In line 5, we search for an index in `currTable` where a valid support for the variable-value pair (x, a) is found, thereby checking the condition in (3.2). In line 9 we remove a from $\text{dom}(x)$ if (x, a) is not supported any more.

Lines 10-11 increments the counter of unassigned variables if $|\text{dom}(x)| > 1$.

Lines 12-13 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 7.

```
1: Method propagate()
2:   updateTable()
3:   if currTable.isEmpty() then
4:     return Failed
5:   return filterDomains()
```

Algorithm 7: Method propagate() in Class CT-Propagator. updateTable() (Algorithm 5) is called, and if the current table is empty, we are in a failed node. Otherwise, filterDomains() (Algorithm 6) is called, and the return value of that method is returned.

3.2.3 Propagator Status Messages

Not really sure where to put this text or whether it fits here. I will need to modify it and I know that I need to define all the concepts properly in the background. I will just let it "lie around" here for now.

A propagtor signals a status message after propagation. For CT there are three possible status messages; *Fail*, *Subsumed* and *Fixpoint*.

Fail. A propagator must correctly signal failure if it has decided that the constraint is unsatisfiable for the input store S. At the latest a propagator must be able to decide whether or not a S is a solution store when all variables have been assigned. CT has two ways of detecting a failure: either when all bits in `currTable` are set to zero – meaning that none of the tuples are valid, or when the size of the domain of a variable is zero.

Subsumption. A propagator is not allowed to signal subsumption if it could propagate further at a later point. At the latest, a propagator must signal subsumption if all the variables are assigned in S and a S is a solution store. CT signals subsumption when at most one variable is not assigned, since this is the pointe where no more propagation can be made.

Fixpoint. A propagator is not allowed to claim that it has computed a fixpoint if it could still propagate. CT always computes a fixpoint if it is not subsumed or wipes out the domain of a variable. To understand this, consider two consequtive calls to propagate(). Let T be the set of valid tuples and x be the set of variables. The first time propagate() is executed, a (possibly empty) subset T_r of T is invalidated (that is, the corresponding bits in `currTable` are set to 0) in updateTable(). Assuming `currTable` is not empty after the return of updateTable(), which would cause a backtrack in the search, filterDomains() is called. This method removes a (possibly empty) set of values V_i from the domain of each variable x_i. Each value $v_i \in V_i$ has the property that a subset T_{v_i} of the tuples in T_r are the last supports for (x_i, v_i). In other words, $T \setminus T_{v_i}$ does not contain a support for (x_i, v_i). So in the second call to updateTable(), no tuples are invalidated, because none of the tuples in $T \setminus T_r$ is a support for any variable-value pair $(x_i, v_i) \in \{x_i\} \times V_i, i \in \{1...|x|\}$. Hence, the second call to propagate() does not give any further propagation.

4 Implementation

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 `currTable` 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 `currTable` and `supports`.

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

5.2 Results

5.3 Discussion

6 Conclusions and Future Work

References

[1] 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.

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

A Source Code

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