Implementation and Evaluation of a Compact Table Propagator in Gecode

Linnea Ingmar

15th 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

Todo: 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 chapter provides a background that is relevant for the following chapters. It is divided into three parts: Section 2.1 introduces Constraint Programming. Section 2.2 gives an overview of Gecode, a constraint solver. Finally, Section 2.3 introduces the Table constraint.

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. One small example is the set of variables $\{x, y\}$ with domain $\{1, 2, 3\}$, together with the constraint $x < y$. This problem has two solutions, namely $(x, y) = (1, 2), (x, y) = (1, 3)$ and $(x, y) = (2, 3)$. If, on the other hand, the domains of x and y would have been $\{2, 3\}$ and $\{1, 2\}$, respectively, then there would exist no solution since none of the possible assignments of x and y would have satisfied the constraint $x < y$.

Sometimes the solution should not only satisfy the set of constraints for the problem, but should also maximise or minimise some given function. Take the same small example as above,

with the domains of x, y being $\{1, 2, 3\}$, and suppose that the solution should maximise the sum of x and y. Then the solution is $(x, y) = (2, 3)$.

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 variable.

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

Propagation

Definition 1. *A domain D is a complete mapping from a fixed (countable) set of variables V to finite sets of integers. A domain D is failed, if $D(x) = \emptyset$ for some $x \in V$. A variable $x \in V$ is fixed by a domain D, if $|D(x)| = 1$. The intersection of domains $D1$ and $D2$, denoted $D1 \sqcap D2$, is defined by the domain $D(x) = D1(x) \cup D2(x)$ for all $x \in V$. A domain $D1$ is stronger than a domain $D2$, written $D1 \sqsubseteq D2$, if $D1(x) \subset D2(x)$ for all $x \in V$.*

A propagator p is a function that maps domains to domains.

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.

3.1 Sparse Bit-Set

This section describes the class `SparseBitSet` which is the main datastructure in the CT algorithm for maintaining the supports.

3.2 Compact-Table (CT) Algorithm

This section describes the CT algorithm.

Compact-Tablce (CT) is a domain consistent propagation algorithm for a table constraint c. It dynamically maintains a set of valid supports regarding the current domain of each variable. This description is mainly taken from [**?**].

3.2.1 Fields

Todo: Add examples with figures for describing the fields.

The fields of the class `CT-Propagator` are as follows:

- `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:

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

- `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 main idea is that 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):

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

  One can optimise this definition slighty since it is unnecessary to keep track of invalid supports. Instead, only the tuples that are valid supports for c need to be indexed in $\texttt{supports}[x, a]$, potentially making the bit-set shorter if only a subset of the tuples in the initial table are valid supports. If the initial table is $T = \langle \tau_0, \tau_1, ..., \tau_{p-1} \rangle$ and it turns out that $\tau_0, ..., \tau_{i-1}$ are valid supports and τ_i is invalid, then τ_{i+1} will correspond to index i instead of $i + 1$ in $\texttt{supports}[x, a]$. So a better definition reads:

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

  where $j = |\{\tau_k \mid \tau_k \in T \ \wedge \ \tau_k \text{ is valid } \wedge \ k < i\}|$. Seing `supports` as a matrix, we have that the column $\texttt{supports}[*, *][i]$ encodes the ith valid 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), $\texttt{residues}[x, a]$ denotes the index of the word in `currTable` where a support was found for (x, a) the last time it was sought for.

The initialisation of the fields of Class `CT-Propagator` is described in Algorithm **??**. 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.

Lines 2-3 initialise `scp` and `lastSize`. The array `lastSize` is filled with the dummy value -1 because we want to have $|\text{dom}(x)| \neq \texttt{lastSize}[x]$ the first time that the propagation algorithm runs.

Lines 4-6 initialise local variables that will be used later.

Line 7 initialises `supports` 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 the initial table. Each bit-set is assumed to be initially filled with zeros.

Lines 8-16 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 $x \in scp(c)$. If t is a valid support, all elements in `supports` corresponding to t are to be set to 1. We keep a counter, *no_supports*, for the number of valid supports for c. This is used for indexing the tuples in `supports` in the way discussed above. The method setElemsInColumn(i, t) is assumed to set all values corresponding to the values in t in the ith column in `supports` to 1. Todo: pseudocode for setElemsInColumn() Todo: Perhaps break out lines 8-16 to an own algorithm-environment.

Todo: trimToWidth().

Line 18 initialises `currTable` as a `SparseBitSet` object with *no_supports* bits, initially with all bits set to 1 since i tuples are initially valid supports for c.

Line 19 initialises `residues` as an array of ints filled with zeros. Recall that `residues`$[x, a]$ is the index of the word where the last found support for (x, a) resides in `currTable`. Since we have not sought for any supports yet, we let 0 be our start guess. Todo: Actually, we could initialise `residues`$[x, a]$ in the if-statement in line 14.

To keep down the size of `supports`, one could perform simple bounds propagation in initialiseCT(). Algorithm **??** is inserted after line 3 in initialiseCT(). 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.

3.2.2 Propagator Status Messages

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 chapter 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 can 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.

```
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 the class SparseBitSet.

```
1: Method initialiseCT(variables, tuples)
2:    scp ← variables
3:    lastSize ← array of length scp.length filled with -1                    // dummy value
4:    size ← sum {|dom(x)| : x ∈ scp}
5:    ntuples ← tuples.size()                                                  // Number of tuples
6:    no_supports ← 0
7:    supports ← BitSets(size, ntuples) // bit-set matrix with size rows and ntuples columns
8:    foreach t ∈ tuples do
9:       supported ← true
10:      foreach x ∈ scp do
11:         if t[x] ∉ dom(x) then
12:            supported ← false
13:            break                                                          // Exit loop
14:      if supported then
15:         no_supports ← no_supports + 1
16:         supports.setElemsInColumn(no_supports, t)
17:   supports.trimToWidth(no_supports)     // Keep only the first no_supports bits for each
      row
18:   currTable ← SparseBitSet(no_supports)       // SparseBitSet with no_supports bits
19:   residues ← array of size size filled with 0
```

Algorithm 2: Pseudo code for initialising the CT-propagator.

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

Algorithm 3: Simple initial propagation for keeping down the size of `supports`.

```
1: Class CT-Propagator

2: scp: array of variables
3: currTable: SparseBitSet                                   // Current supported tuples
4: supports: array of BitSet          // 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 updateTable()
8:   foreach x ∈ scp such that |dom(x)| ≠ lastSize[x] do
9:     lastSize[x] ← |dom(x)|
10:    currTable.clearMask()
11:    foreach a ∈ dom(x) do
12:      currTable.addToMask(supports[x, a])
13:      currTable.intersectWithMask()
14:      if currTable.isEmpty() then
15:        break                                                // No valid tuples left

16: Method filterDomains()
17:   count_unassigned ← 0
18:   foreach x ∈ scp such that |dom(x)| > 1 do
19:     foreach a ∈ dom(x) do
20:       index ← currTable.intersectIndex(supports[x, a])
21:       if index ≠ −1 then
22:                                                            // No residues yet!
23:       else
24:         dom(x) ← dom(x) \ {a}
25:         if |dom(x)| > 1 then
26:           count_unassigned ← count_unassigned + 1
27:   if count_unassigned ≤ 1 then
28:     return Subsumed
29:   else
30:     return Fixpoint

31: method propagate()
32: updateTable()
33: if currTable.isEmpty() then
34:   return Failed
35: return filterDomains()
```

Algorithm 4: Pseudo code for CT propagator class.