Implementation and Evaluation of a Compact Table Propagator in Gecode

Linnea Ingmar

20th February 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. There need not exist a solution to a given problem, and a solution need not be unique; zero or more assignments of values to the variables may satisfy the constraints. One small example is the set of variables $\{x, y\}$ that both have the domain $\{1, 2, 3\}$, together with the constraint $x < y$. This problem has several 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 be optimal in some way; it should either 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, that is, values that the solver detects would violate at least one of the constraints if the variables were to be assigned to any of those values. This filtering process is called *propagation*. Each constraint that the solver supports is associated with at least one propagator algorithm, whose task is to detect impossible values for the constraint. A solution is found when all the variables have been assigned a value that

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.

3.2.1 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.

```
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

Function isEmpty() : Boolean
  return limit = $-1$

Function clearMask()
  for $i \leftarrow 0$ to limit do
    offset $\leftarrow$ index[$i$]
    mask[offset] $\leftarrow 0^{64}$

Function addToMask(m: array of long)
  for $i \leftarrow 0$ to limit do
    offset $\leftarrow$ index[$i$]
    mask[offset] $\leftarrow$ mask[offset] | m[offset]          // bitwise OR

Function intersectWithMask()
  for $i \leftarrow$ limit downto 0 do
    offset $\leftarrow$ index[$i$]
    $w \leftarrow$ words[offset] & mask[offset]                 // bitwise AND
    if $w \neq$ words[offset] then
      words[offset] $\leftarrow w$
      if $w = 0^{64}$ then
        index[$i$] $\leftarrow$ index[limit]
        limit $\leftarrow$ limit $- 1$

Function intersectIndex(m: array of long) : int
  for $i \leftarrow 0$ to limit do
    offset $\leftarrow$ index[$i$]
    if words[offset] & $m$[offset] $\neq 0^{64}$ then
      return offset
  return $-1$
```

Algorithm 1: Pseudo code for the class SparseBitSet.

```
Class CT-Propagator

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

Function initialise(variables, tuples)
  scp ← variables
  foreach $x \in$ scp do
    $\mathrm{dom}(x) \leftarrow \mathrm{dom}(x) \setminus \{a \in \mathrm{dom}(x) : a >$ tuples.$max()\}$
    $\mathrm{dom}(x) \leftarrow \mathrm{dom}(x) \setminus \{a \in \mathrm{dom}(x) : a <$ tuples.$min()\}$
  size ← $sum\,\{|\mathrm{dom}(x)| : x \in$ scp$\}$
  ntuples ← tuples.$size()$                                  // Number of tuples
  supports ← BitSets(size, ntuples)   // bit-set matrix with size rows and ntuples columns
  no_supports ← 0
  foreach $t \in$ tuples do
    supported ← true
    foreach $x \in$ scp do
      if $t[x] \notin \mathrm{dom}(x)$ then
        supported ← false
        break                                                   // Exit loop
    if supported then
      no_supports ← no_supports + 1
      supports.setElemsInColumn(no_supports, $t$)
  supports.trimToWidth(no_supports)  // Keep only the first no_supports bits for each row
  currTable ← SparseBitSet(no_supports)     // SparseBitSet with no_supports bits

Function updateTable()
  foreach $x \in$ scp such that $|\mathrm{dom}(x)| \neq$ lastSize[$x$] do
    lastSize[$x$] ← $|\mathrm{dom}(x)|$
    currTable.clearMask()
    foreach $a \in \mathrm{dom}(x)$ do
      currTable.addToMask(supports[$x, a$])
      currTable.intersectWithMask()
      if currTable.isEmpty() then
        break

method filterDomains()
no_assigned ← 0
foreach $x \in$ scp do
[2] foreach $a \in \mathrm{dom}(x)$ do
[3] index ← currTable.intersectIndex(supports[$x, a$])
      if index $\neq -1$ then
[4]                                                          // No residues yet!
      else
[4] $\mathrm{dom}(x) \leftarrow \mathrm{dom}(x) \setminus \{a\}$
          if $|\mathrm{dom}(x)| = 0$ then
            return $Failed$
          else if $|\mathrm{dom}(x)| > 1$ then
[5] no_assigned ← $no\_assigned + 1$
 if no_assinged $\leq 1$ then
   return $Subsumed$
 else
```