Implementation and Evaluation of a Compact Table Propagator in Gecode

Linnea Ingmar

15th May 2017

Contents

1 Introduction

Constraint Programming (CP) is a programming paradigm that is used for solving combinatorial problems. Within the paradigm, a problem is modelled as a set of *constraints* on a set of *variables* that each can take on a number of possible values. The possible values of a variable is called the *domain* of the variable. A solution to a constraint problem must assign all variables a value from their domains, so that all the constraints of the problem are satisfied. Additionally, in some cases the solution should not only satisfy the set of constraints for the problem, but also maximise or minimise some given function.

A solution to a constraint problem is found by generating a search tree, branching on possible values for the variables. At each node in the search tree, conflicting values are filtered out from the domains of the variables in a process called *propagation*, effectively reducing the size of the search tree. Each constraint is associated with a *propagation algorithm*, called a *propagator*, that implements the propagation for that constraint by removing values from the variables that are in conflict with the constraint.

The Table constraint expresses the possible combinations of values that the associated variables can take as a sequence of tuples. Assuming finite domains, the Table constraint can theoretically encode any kind of constraint and is thus very powerful. The design of propagation algorithms for Table is an active research field, and several algorithms are known. In 2016, a new propagation algorithm for the Table constraint was published [**?**], called Compact Table (CT). Their results indicate that CT outperforms all previously known algorithms.

A constraint programming solver (CP solver) is a software that solves constraint problems. *Gecode* [**?**] is a popular CP solver written in C++ which combines state-of-the-art performance with modularity and extensibility. Presently, Gecode has three existing propagation algorithms for Table, but there have been no attempts to implement CT in Gecode befor this project. Consequently, its performance in Gecode was unknown. The purpose of this thesis is therefore to implement CT in Gecode and to evaluate and compare its performance with the existing propagators for the Table constraint. The results of the evaluation indicate that CT outperforms the existing propagation algorithms in Gecode for Table, which suggests that CT should be included in the solver.

1.1 Goal

The goal of this bachelor's thesis is the design, documentation and implementation of a CT propagator algorithm for the Table constraint in Gecode, and the evaluation of its performance compared to the existing propagators.

1.2 Contributions

The following section lists the contributions made by this bachelor thesis, while simultaneously serving as a description of the outline of the written dissertation.

- The preliminaries that are relevant for the rest of the dissertation are covered in Section 2.
- The algorithms presented in the paper that is the starting point of this project [**?**] have been modified to suit the target constraint solver Gecode, and are presented and explained in Section 3.
- Several versions of the CT algorithm has been implemented in Gecode, and the implementation is discussed in Section 4.

- The performance of the CT algorithm has been evaluated, and the results are presented and discussed in Section 5.
- The conclusion of the project is that the results indicate that CT outperforms the existing propagation algorithms, which suggests that CT should be included in Gecode, this is discussed in Section 6.
- Several possible improvements and flaws have been detected in the current implementation that need to be fixed for the code to reach production quality, these are listed in Section 6.

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 TABLEconstraint. 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).

Constraint Programming (CP) is a programming paradigm that is used for solving combinatorial problems. Within the paradigm, a problem is modelled as a set of *constraints* on a set of *variables* that each can take on a number of possible values. The possible values of a variable is called the *domain* of the variable. A solution to a constraint problem must assign all variables a value from their domains, so that all the constraints of the problem are satisfied. Additionally, in some cases the solution should not only satisfy the set of constraints for the problem, but also maximise or minimise some given function.

A constraint solver (CP solver) is a software that solves constraint problems. The process of solving 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 purpose is to detect and remove values from the domains of the variables that can never participate in a solution because assigning them to the variables would violate the constraint, effectively shrinking the domain sizes and thus pruning the search tree. When sufficient[1] propagation has been performed and a solution is still not found, the solver must *branch* the search tree, following some heuristic, which typically involves selecting a variable and "guessing" its value, moving the search to a new node in the tree where propagation starts over again.

Propagation interleaved with branching continues along a path in the search tree, until the search reaches a leaf node, which can be either a *solution node* or a *failed node*. In a solution node a solution to the problem is found: all variables are assigned a value from their domains, and all the constraints are satisfied. In a *failed node*, the domain of a variable has become empty, which means that a solution could not be found along that path. From a failed node, search must backtrack and continue from a node where all branches have not been tried yet.

[1]Here "sufficient" might either mean that no more propagation can be made, or that more propagation is possible, but the solver has decided that it is more efficient to branch to a new node instead of performing more propagation at the current node.

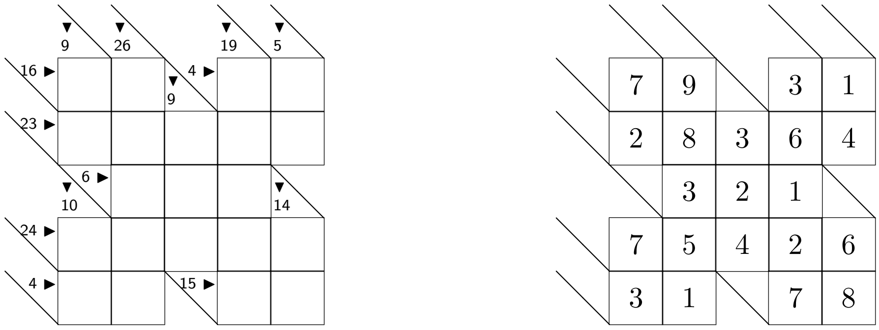

Figure 1: A Kakuro puzzle [3](left) and its solution (right). Todo: solve the Kakuro.

If all leaves of the tree consist of failed nodes, then the problem is unsatisfiable, else there is a solution that will be found eventually[2].

To build intuition and understanding of the ideas of CP, the concepts can be illustrated with logical puzzles. One such puzzle is Kakuro, somewhat similar to the popular puzzle Sudoku, a kind of mathematical crossword where the “words” consist of numbers instead of letters, see Figure 1. 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 objective is to put 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 $\{1, \ldots, 9\}$. 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 coist 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.

After gaining some intuition of CP, here follows some formal definitions, based on [**?**], [**?**], and [**?**].

We start by defining *constraints*, that essentially are relations among variables.

Definition 1. ***Constraint.*** *Consider a finite sequence of n variables $X = x_1, \ldots, x_n$, and a corresponding sequence of finite* domains *$D = D_1, \ldots, D_n$ ranging over integers, that are possible values for the respective variable. For a variable $x_i \in X$, its domain D_i is denoted by $dom(x_i)$. Its* domain size *is $|dom(x_i)|$ and its* domain width *is $(max(dom(x_i)) - min(dom(x_i)) + 1)$.*

- *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, and we call those n-tuples* solutions *to the constraint c.*

[2]Here “eventually” means “if search is allowed to go on forever”.

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

- *For an n-tuple* $\tau = \langle a_1, \ldots, a_n \rangle$ *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 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* $\tau[x] = a$.

Note that Definition 1 restricts the domains to range over finite sets of integers. Constraints can be defined on other sets of values, but in this thesis only finite integer domains are considered.

After defining constraints, we define *constraint satisfaction problems*:

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 finite set of constraints, each on a subsequence of* V.

During the search for a solution to a CSP, the domains of the variables will vary: along a path in the search tree, the domains shrink until they are assigned a value (a solution node) or until the domain of a variable becomes empty (a failed node). When encountering a failure, the search backtracks to a node in the search tree where all branches are not yet exhausted, and the domains of the variables are restored to the domains that the variables had in that node, so that the search continues from an equivalent state. A current mapping of domains to variables is called a *store*:

Definition 3. ***Stores.*** *A* constraint 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* $s(v_i) = \emptyset$ *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* $s_1(v_i) \subseteq s_2(v_i)$ *for all* $v_i \in V$.

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 participate in a solution store to the problem. In a CP solver, each constraint that the solver implements is associated with one or more propagation algorithms (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 have. The following is a definition of propagators and the obligations that they must meet, taken from [**?**] and [**?**].

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*.*

The possible status messages are Fail*,* Subsumed*,* Fixpoint*, and* Not fixpoint*. 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 to itself, that is, if $p(s) = p(p(s))$ for all stores s, p is* idempotent*. A propagator is* subsumed *by a store s if and only if all stronger stores are fixpoints: $\forall s' \preceq s,\ p(s') = s$.*

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 is not a strict obligation, though it is desirable as it 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 exclude 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 Subsumed, otherwise it must signal Fail.*
- *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 Subsumed if it is not subsumed by an input store s.*

This is in fact a rather weak definition; a propagator is not even obliged to prune values from the domains of the variables, as long as it can decide whether a given assignment store is a solution store or not. An extreme case is the identity propagator i, with $i(s) = s$ for all input stores s. As long as i is checking and honest, it could implement any constraint c, because it fulfills all the other obligations: it is a decreasing and monotonic function (because $i(s) = s \preceq s$) and it is correct for c (because it never removes values).

Also, note that the honest property does *not* mean that a propagator is *obliged* to signal Fixpoint or Subsumed if it has computed a fixpoint or is subsumed, only that it must not claim fixpoint or subsumption if that is not the case. Thus, it is always safe for a propagator to signal Not fixpoint, except for an assignment store when it must signal either Fail or Subsumed as required by the honest property.

So why not stay on the safe side and always signal Not fixpoint? The reason is that the CP solver can benefit from the information in the status message: if a propagator p is at fixpoint, there is no point to execute p again until the domains of the variables change. If p is subsumed by a store s, there is no point to execute p ever again along the current path in the search tree, because all the following stores will be stronger than s. Thus, detecting fixpoints and subsumption can save many unnecessary operations.

The concept *consistency level* gives a measure of how strong the propagation of a propagator is. There are three commonly used consistency levels, **value consistency, bounds consistency**, and **domain consistency**.

Definition 5. ***Bounds consistency.*** *A constraint c is bounds consistent on a store s if and only if there exists at least one support for the lower and for the upper bound of each variable associated with c:* $\forall x \in vars(c)$, $(x, min(dom(x)))$ *and* $(x, max(dom(x)))$ *have a support on c.*

Definition 6. ***Domain consistency.*** *A constraint c is domain consistent on a store s if and only if there exists at least one support for all values of each variable associated with c:* $\forall x \in vars(c)$, *and* $\forall a \in dom(x)$, (x, a) *has a support on c.*

Todo: Value consistency.

Value consistency is weaker than bounds consistency, and bounds consistency is weaker than domain consistency.

A propagator p is said to have a certain consistency level if after applying p to any input store s, the resulting store $p(s)$ always has that consistency level. Enforcing a stronger consistency level might remove more values from the domains of the variables, but might be more costly.

The propagator that is concerned in this project is domain consistent.

2.3 Gecode

Gecode [?] (Generic Constraint Development Environment) is a popular constraint programming solver written in C++ and distributed under the MIT license. It has state-of-the-art performance while being modular and extensible. It supports the modular development of the components that make up the CP solver, including specifically the implementation of new propagators. Furthermore, Gecode is well documented and comes with a complete tutorial.

Developing a propagator for Gecode means implementing a C++ object inheriting from the base class Propagator, that complies to a given interface. The propagator can store any data structures as instance members, for saving state information between executions.

One such data structure is *advisors*, which can inform propagators about variable modifications. The purpose of an advisor is to, as its name suggests, advise the propagator of whether it needs to be executed or not. Whenever the domains of a variable changes, the advisor is executed. Once running, it can signal fixpoint, subsumption or failure if it detects such a state.

Advisors enable *incrementality*: they can ensure that the propagator do not need to scan all the variables to see which ones have been modified since its last invocation. Propagators that use data structures to avoid scanning all variables and/or all domains of the variables in each execution is said to be *incremental*.

Search in Gecode is copy-based. Before making a decision in the search tree, the current node is copied, so that the search can restart from a previous state in case the decision fails, or in case more solutions are sought for. This implies some concerns regarding the memory usage for the stored data structures of a propagator, since allocating memory and copying large data structures is time-consuming, and large memory usage is undesirable.

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 7. ***Table constraints.*** *A (positive*[4]*)* table constraint c *is a constraint such that* $rel(c)$ *is defined explicitly by listing all the tuples that are solutions to c.*

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

Theoretically, any constraint could be expressed using the TABLE constraint, simply by listing all the allowed assignments for its variables, making the TABLE constraint very powerful. However, it is typically too memory consuming to represent a constraint in this way (exponential space in the number of variables). Furthermore, common constraints typically have a certain structure that is difficult to take advantage of if the constraint is represented extensionally [?].

Nevertheless, the TABLE constraint is an important constraint. Todo: Typical use cases? Add an example.

In Gecode, the TABLE constraint is called EXTENSIONAL. Gecode provides three propagators for EXTENSIONAL, one where the possible solutions are represented as a DFA, based on [?], and two where the solutions are represented as a tuple set. Of the last two, one is based on [?] and is memory efficient. The other is more efficient in terms of execution time, and is more incremental.

2.5 Compact-Table Propagator

The compact table (CT) algorithm is a domain consistent propagation algorithm that implements the TABLE constraint. It was first implemented in OR-tools, a constraint solver, where it outperforms all previously known algorithms and was first described in [?]. Before this project, no attempts to implement CT in Gecode were made, and consequently its performance in the framework is unknown.

Compact table relies on bit-wise operations using a new data-structure called *reversible sparse bit-set* (see Section 2.6). The propagator maintains a reversible sparse bit set object, `currTable`, which stores the indices of the current valid tuples in a bit-set. Also, for each variable-value pair, a bit-set mask is computed, that stores the indices of the tuples that are supports for that variable-value pair. These bit-set masks are stored in an array, `supports`.

Propagation consists of two steps:

1. Updating `currTable` so that it only contains indices of valid tuples.

2. Filtering out inconsistent values from the domains of each variable, that is, values that no longer have a tuple that supports it.

Both steps rely heavily on bit-wise operations on `currTable` and `supports`. CT is discussed more deeply in Section 3.

2.6 Reversible Sparse Bit-Sets

Reversible sparse bit-sets [?] is a data structure for storing a set of values. It avoids performing operations on words of only zeros, which makes it efficient to perform bit-wise operations with other bit-sets (such as intersecting and unioning).

A reversible sparse bit-set has an array of ints, `words`, that are the actual stored bits, an array `index` that keeps track of the indices of the non-zero words, and an int `limit` that is the index of the last non-zero word in `index`. Also, it has a temporary mask (array of ints) that is used to modify `words`.

Some CP-solvers use a mechanism ca called *trailing* to perform backtracking (as previously discussed, Gecode uses copying instead), where the main idea is to store a stack of operations that can be undone upon backtrack. These CP-solvers typically expose some “reversible” objects to the users using this mechanism, among them the reversible version of the primitive type int. The first word of the name of the data-structure comes from the assumption that `words` consists of reversible ints.

In the following section, a data structure that is similar to reversible sparse bit-sets except that it consists of ordinary ints and not reversible ints will be called a sparse bit-set. Sparse bit-sets are discussed in Section 3.

3 Algorithms

This chapter presents the algorithms that are used in the implementation of the CT propagator in Section 4. In the following section we use the notation that for an array a, $a[0]$ denotes its first element (indexing starts from 0), a.length() its number of 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()
   Method i ← 0 to limit do
      offset ← index[i]
      mask[offset] ← 0^64

Method reverseMask()
   Method i ← 0 to limit do
      offset ← index[i]
      mask[offset] ← ∼ mask[offset]                    // bitwise NOT

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

Method intersectWithMask()
   Method 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
   Method 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. Here 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:

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

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 18-17 clears the temporary mask. This means setting to 0 all words of `mask` corresponding to non-zero words of `words`.
- reverseMask() in lines 14-21 reverses the bits in the temporary mask.
- addToMask() in lines 22-25 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 26-35 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 34 because it is unnecessary to save the index of a zero-word in a copy-based solver such as Gecode. We keep this line here though, as the invariant in (3.1.1) would not hold otherwise.

- intersectIndex() in lines 36-41 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 39 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

The CT algorithm is a domain consistent propagation algorithm for any TABLE constraint c. Section 3.2.1 presents pseudo code for the CT algorithm and a few variants and Section 3.2.2 proves that CT fulfills the propagator obligations.

3.2.1 Pseudo code

When posting the propagator, the input is an initial table, that is a list of tuples $T_0 = \langle \tau_0, \tau_1, \ldots, \tau_{p_0-1} \rangle$ of length p_o. In what follows, we call the *initial valid table* for c the subset $T \subseteq T_0$ of size $p \leq p_0$ where all tuples are a support on c for the initial domains of $vars(c)$. For a variable x, we distinguish between the *initial domain* $\underline{\mathrm{dom}}(x)$ and the *current domain* $\mathrm{dom}(x)$ or $s(x)$. In an abuse of notation, we denote $x \in s$ for a variable x that is part of store s. We denote $s[x \mapsto A]$ the store that is like s except that the variable x is mapped to the set A.

The propagator state has the following fields.

- `validTuples`, a `SparseBitSet` object representing 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`, a static array of bit-sets representing the supports for each variable-value pair (x, a). 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 valid table of c is initially a support for (x, a):

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

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

- `residues`, an array of ints 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.

Algorithm 2 shows the CT algorithm. Lines 1-4 initialises the propagator if it is being posted (initialised). CT reports failure in case a variable domain was wiped out in INITIALISECTor if `validTuples` is empty, meaning no tuples are valid. If the propagator is not being posted, lines 6-9 call UPDATETABLE() for all variables whose domains have changed since last time. UPDATETABLE() will remove from `validTuples` the tuples that are no longer supported, and CT reports failure if all tuples were removed. If at least one variable was pruned, FILTERDOMAINS() is called, which will filter out values from the domains of the variables that no longer have supports, enforcing domain consistency. CT is subsumed if there is at most one unassigned variable left, otherwise CT is at fixpoint. The condition for fixpoint is correct because CT is idempotent, which is shown in the proof of Lemma 3.5. Why the condition for subsumption is correct is shown in the proof of Lemma 3.8.

```
PROCEDURE CompactTable(s : store) : ⟨StatusMessage, store⟩
if the propagator is being posted then                      // executed in a constructor
  s ← InitialiseCT(s, T0)
  if s = ∅ then
    return ⟨FAIL, ∅⟩
else                                                         // executed in an advisor
  foreach variable x ∈ s whose domain has changed since last time do
    UpdateTable(s, x)
    if validTuples.isEmpty() then
      return ⟨FAIL, ∅⟩
  if validTuples has changed since last time then // executed in the propagate function
    s ← FilterDomains(s)
if there is at most one unassigned variable left then
  return ⟨SUBSUMED, s⟩
else
  return ⟨FIX, s⟩
```

Algorithm 2: Compact Table Propagator.

The initialisation of the fields is described in Algorithm 3. InitialiseCT() takes the initial table T_0 as argument.

```
PROCEDURE InitialiseCT(s: store, T_0: list of tuples) : store
 1: foreach x ∈ s do
 2:     R ← {a ∈ s(x) : a > T_0.max() ∨ a < T_0.min()}
 3:     s ← s[x ↦ s(x) \ R]
 4:     if s(x) = ∅ then
 5:         return ∅
 6: npairs ← sum {|s(x)| : x ∈ s}                        // Number of variable-value pairs
 7: ntuples ← T_0.size()                                               // Number of tuples
 8: nsupports ← 0                                              // Number of found supports
 9: residues ← array of length npairs
10: supports ← array of length npairs with bit-sets of size ntuples
11: foreach t ∈ T_0 do
12:     supported ← true
13:     foreach x ∈ s do
14:         if t[x] ∉ s(x) then
15:             supported ← false
16:             break                                                           // Exit loop
17:     if supported then
18:         foreach x ∈ s do
19:             supports[x, t[x]][nsupports] ← 1
20:             residues[x, t[x]] ← ⌊nsupports / 64⌋     // Index for the support in validTuples
21:             nsupports ← nsupports + 1
22: foreach x ∈ s do
23:     R ← {a ∈ s(x) : supports[x, a] = ∅}
24:     s ← s[x ↦ s(x) \ R]
25:     if s(x) = ∅ then
26:         return ∅
27: validTuples ← SparseBitSet with nsupports bits
28: return s
```

Algorithm 3: Initialising the CT-propagator.

Lines 1-5 perform simple bounds propagation to limit the domain sizes of the variables, which in turn will limit the sizes of the 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 (its domain becomes empty), $Failed$ is returned.

Lines 6-8 initialise local variables for later use.

Lines 9-10 initialise the fields `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 11-21 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 19. We also take the opportunity to store the word index of the found support in `residues`$[x, t[x]]$ in line 20.

Lines 22-24 remove values that are not supported by any tuple in the initial valid table. The procedure returns in case a variable has a domain wipe out.

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. At this point $nsupports > 0$, otherwise we would have returned at line 26.

PROCEDURE UPDATETABLE(s: store, x: variable)
1: `validTuples`.clearMask()
2: **foreach** $a \in s(x)$ **do**
3:    `validTuples`.addToMask(`supports`$[x, a]$)
4: `validTuples`.intersectWithMask()

Algorithm 4: Updating the current table. The infrastructure is such that this procedure is called for each variable whose domain is modified since last time.

The procedure UPDATETABLE() in Algorithm 4 filters out (indices of) tuples that have ceased to be supports for the input variable x. Lines 2-3 stores the union of the set of valid tuples for each value $a \in \text{dom}(x)$ in the temporary mask and Line 4 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.

The algorithm is assumed to be run on an infrastructure that runs UPDATETABLE() for each variable $x \in vars(c)$ whose domain has changed since last time.

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 `validTuples` and `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 FILTERDOMAINS() in Algorithm 5.

PROCEDURE FILTERDOMAINS(s) : store
1: **foreach** $x \in s$ such that $|s(x)| > 1$ **do**
2:    **foreach** $a \in s(x)$ **do**
3:       $index \leftarrow$ `residues`$[x, a]$
4:       **if** `validTuples`$[index]$ & `supports`$[x, a][index] = 0$ **then**
5:          $index \leftarrow$ `validTuples`.intersectIndex(`supports`$[x, a]$)
6:          **if** $index \neq -1$ **then**
7:             `residues`$[x, a] \leftarrow index$
8:          **else**
9:             $s \leftarrow s[x \mapsto s(x) \setminus \{a\}]$
10: **return** s

Algorithm 5: Filtering variable domains, enforcing domain consistency.

We note that it is only necessary to consider a variable $x \in s$ such that $s(x) > 1$, 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 (3.2.1) `validTuples` must be empty, which it will not be upon invocation of FILTERDOMAINS, because then COMPACTTABLE() would have reported failure.

In Lines 3-4 we check if the cached word index still has a support for (x, a). It it has not, we search for an index in line 5 in `validTuples` where a valid support for the variable-value pair (x, a) is found, thereby checking the condition in (3.2.1). If such an index exists, we cache it in `residues`$[x, a]$, and if it does not, we remove a from $\text{dom}(x)$ if (x, a) in line 9 since there is no support left for (x, a).

Optimisations. If x is the only variable that has been modified since the last invocation of CompactTable(), 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 2-9 for $s \setminus \{x\}$.

Variants. The following lists some variants of the CT algorithm.

CT(Δ) – *Using delta information in* UpdateTable*().* A variable x's delta, Δ_x, is the set of values that were removed from x since last time. If the infrastructure provides information about Δ_x, that information can be used in UpdateTable(). Algorithm 6 shows a variant of UpdateTable() that uses delta information. If Δ_x is smaller than $\text{dom}(x)$, we accumulate to the temporary mask the set of invalidated tuples, and then reverse the temporary mask before intersecting it with `validTuples`.

PROCEDURE UpdateTable(s: store, x: variable)
`validTuples`.clearMask()
if Δ_x is available $\wedge\ |\Delta_x| < |s(x)|$ **then**
  foreach $a \in \Delta_x$ **do**
    `validTuples`.addToMask(`supports`$[x, a]$)
  `validTuples`.reverseMask()
else
  foreach $a \in s(x)$ **do**
    `validTuples`.addToMask(`supports`$[x, a]$)
`validTuples`.intersectWithMask()

Algorithm 6: Updating the current table using delta information.

CT(T) – *Fixing the domains when only one valid tuple left.* This variant is the only addition made to the algorithm presented in [**?**]. If only one valid tuple is left after all calls to UpdateTable() are finished, the domains of the variables can be fixed to the values for that tuple. Algorithm 7 shows an alternative to lines 10-11 in Algorithm 2. This assumes that the propagator maintains an extra field T – a list of tuples representing the initial valid table for c.

if `validTuples` has changed since last time **then** // executed in the propagate function
  if $(\textit{index} \leftarrow$ `validTuples`.indexOfFixed()$) \neq -1$ **then**
    return $\langle$**SUBSUMED**$, [x \mapsto T[\textit{index}][x]]_{x \in s}\rangle$
  else
    $s \leftarrow$ FilterDomains(s)

Algorithm 7: Alternative to lines 10-11 in Algorithm 2, assuming the initial valid table T is stored as a field.

For a word `w`, there is exactly one bit set if and only if

$$\texttt{w} \neq 0 \quad \wedge \quad (\texttt{w} \,\&\, (\texttt{w} - 1)) \;=\; 0,$$

a condition that can be checked in constant time. This is implemented in Algorithm 8, which returns the bit index of the set bit if there is exactly one bit set, else −1. The method IndexOfFixed() is added to Class `SparseBitSet`and assumes access to builtin MSB which returns the index of the most significant bit of a given int.

PROCEDURE IndexOfFixed(): int
1: $index\_of\_fixed = -1$
2: **if** $\texttt{limit} = 0$ **then**
3: $\quad offset \leftarrow \texttt{index}[0]$
4: $\quad w \leftarrow \texttt{words}[offset]$
5: $\quad$ **if** $(w \,\&\, (w-1)) = 0$ **then** // Exactly one bit set
6: $\quad\quad index\_of\_fixed = offset \cdot 64 + \text{MSB}(\texttt{words}[offset])$
7: **return** $index\_of\_fixed$

Algorithm 8: Checking if exactly one bit is set in `SparseBitSet`.

3.2.2 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 executing CompactTable(s) on an input store s.

Lemma 3.2. *CT is domain consistent.*

Proof of Lemma 3.2. There are two cases; either it is the first time CT is called, or it is not. In the first case, InitialiseCT() is called, which removes all values from the domains of the variables that have no support. In the second case, UpdateTable() is called for each variable whose domain has changed, and in case `validTuples` is modified, FilterDomains() removes all values from the domains that are no longer supported. If `validTuples` is not modified, all values still have a support because all tuples that were valid the previous time still are valid.

So in both cases, every variable-value pair (x, a) has a support on c, which shows that CT is domain consistent. □

Lemma 3.3. *CT is a decreasing function.*

Proof of Lemma 3.3. 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 3.4. *CT is a monotonic function.*

Proof of Lemma 3.4. 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 $CT(s_1) \preceq CT(s_2)$. □

Lemma 3.5. *CT is idempotent.*

Proof of Lemma 3.5. 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 (3.2.1), there must exists at least one tuple τ_i that is a support for (x, a) under the store $CT(s)$: $\exists i : i \in \texttt{validTuples} \ \wedge \ \tau_i[x] = a$. After UPDATETABLE() is performed on $CT(s)$, we still have $i \in \texttt{validTuples}$, 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 3.6. *CT is correct for the* TABLE *constraint.*

Proof of Lemma 3.6. CT does not remove values that participate in tuples that are supports on a TABLE constraint c, since FILTERDOMAINS() and INITIALISECT() only removes values that have no supports on c. Thus, CT is correct for TABLE. □

Lemma 3.7. *CT is checking.*

Proof of Lemma 3.7. 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 INITIALISECT() 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 INITIALISECT(), which will return a failed store, which results in failure in line 4 in Algorithm 2.

If it is not the first time that CT is called, `validTuples`will be empty after all calls to UPDATETABLE() have finished, because there are no valid tuples left, which results in failure in line 9 in Algorithm 2.

Now assume that s is a solution store. CT signals subsumption in line 13 in Algorithm 2 because all variables are assigned and `validTuples` is not empty.

□

Lemma 3.8. *CT is honest.*

Proof of Lemma 3.8. 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.

□

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

Todo: Structure up this section.

This section describes the implementation of the CT algorithm presented in Section 3. It reveals some important implementation details that the pseudo code conceals, and documents the design decisions made during the implementation.

The implementation was done in C++ in the context of the latest version of Gecode, at the time of writing Gecode 5.0, and following the coding conventions of the solver. No C++ standard library data structures were used, as there is little control over how they allocate and use memory. The implementation follows the pseudo code in in Section 3.2.1 very closely. The correctness of the CT propagator was checked with the existing unit tests in Gecode for the TABLE constraint.

CT reuses the existing tuple set data structure for representing the initial table that is used in the existing propagators for TABLE in Gecode, and thus the function signature for the CT the propagator is the same as to the signature of the previously existing propagators. The tuple set is only used upon initialisation of the fields, except for the variant CT(T) where the tuple set is maintained as a field.

The implementation uses C++ templates to support both integer and boolean domains.

Indexing `residues` and `supports`. For a given variable-value pair (x, a), its corresponding entry `supports`$[x, a]$ and `residues`$[x, a]$ must be found, which requires a mapping $\langle variable, value\rangle \mapsto$ `int` for indexing `supports`and `residues`. Two indexing strategies are used; *sparse arrays* and *hash tables*. For variables with compact domains (range or close to range), indexing is made by allocating space that depend on domain width of `supports` and `residues`, and by storing the initial minimum value for the variable, so that `supports`$[x, a]$ and `residues`$[x, a]$ is stored at index $a - min$ in the respective array. If the domain is sparse, the sizes of `supports` and `residues` is the size of the domain, and the index mapping is kept in a hash table. The indexing strategy is decided per variable. Let $R = \frac{\text{domain width}}{\text{domain size}}$. The current implementation uses a sparse array if $R \leq 3$, and a hash table otherwise. The threshold value was chosen by reasoning about the memory usage and speed of the different strategies. Let a memory unit be the size of an int, and assume that a pointer is twice the size of an int. The sparse array strategy consumes $S = (\text{width} + 2 \cdot \text{width})$ memory units, because `residues` is an array of ints and `supports` is an array of pointers (we neglect the "+1" from the int that saves the initial minimum value). The hash table strategy consumes $H = (2 \cdot \text{size} + \text{size} + 2 \cdot \text{size})$ memory units as best, because at best, the size of the hash table is $2 \cdot \text{size}$. The quantities S and H are equal when $R = \frac{4}{3} \approx 1.33$. Because the hash table might have collisions, this strategy is not always constant time. Therefore the value 3 was chosen, as a trade-off between speed and memory. The optimal threshold value should be found by further experiments.

Copying only the non-zero words in `validTuples`. Some effort was spent on redesigning the sparse bit-set to only keep the non-zero words to save memory and minimise copying by placing all the zero-words at the end of the array. However, the swapping of the words introduces problems with `residues`. If the current index of a word is not the same as the original index of the word, the information saved in `residues` is not correct. No simple solution to this problem could be found, and the idea was abandoned.

Advisors. The implementation uses advisors that decide whether the propagator needs to be executed or not. The advisors execute UPDATETABLE(x) whenever the domain of x changes, schedule the propagator for execution in case `validTuples` is modified, and reports failure in case `validTuples` is empty. There are several benefits to using advisors. Firstly, without advisors, the propagator would need to scan all the variables to determine which ones them have been modified since the last incocation of the propagator, and execute UPDATETABLE() on those, which would be time consuming. Secondly, the advisors can store the data structures that belong to its variable. Meaning that when that variable is assigned, the memory used for storing information about that variable can be freed.

Or-tools. The implementation of CT in OR-tools was studied. They use two versions of CT, one for small tables (≤ 64 tuples) that only use one word for `validTuples` instead of an array. Though this is a promising idea, this variant was not implemented due to time constraints. Also, during propagation the authors of the OR-tools first reason on the bounds of the domains of the variables, enforcing bounds consistency, before enforcing domain consistency. The reason to this is that iterating over the domains is expensive. This optimisation was implemented, and the variant is denoted CT(B) in the evaluation of different versions of CT (Section 5).

Memory usage. Since `supports` consists of static data (only computed once), this array is allocated in a memory area that is shared among nodes in the search tree, which means that it does not need to be copied when branching, in constrast to the rest of the data structures, which are allocated in a memory space that is specific to the current node.

Profiling. Profiling tools were used to locate the parts of the program where most of the time is spent. Some optimisations could be performed based on this information. Specifically, a speed-up could be achieved by decreasing the number of memory accesses in some of the methods in `SparseBitSet`. The profilation shows that the bottleneck in the implementation are the bit-wise operations in `SparseBitSet`, and also that a significant amount of time is spent in FilterDomains(). Include figures.

Using delta information. In the version CT(Δ) that uses the set of values D_x that has been removed since last time, the current implementation uses the incremental update if $|\Delta_x| < |s(x)|$. It is possible that the optimal would be to generalise this condition to $|\Delta_x| < k \cdot |s(x)|$, where $k \in \mathbb{R}$ is some constant, something that remains to be investigated.

5 Evaluation

This chapter presents the evaluation of the implementation of the CT propagator presented in Section 4.

The benchmarks consist of 30 series with ith a total of 1507 CSP instances that were used in the experiments in [?]. The instances contain Table constraints only.

Table 1: Benchmarks series and their characteristics.

name	number of instances	arity	table size	variable domains
A5	50	5	12442	0..11, a few 0..1
A10	50	10	51200	some 0..1, some 0..19, a few singleton
AIM-50	23	3, a few 2	$3-7$	0..1
AIM-100	23	3, a few 2	$3-7$	0..1
AIM-200	22	3, a few 2	$3-7$	0..1
BDD Large	35	15	approx. 7000	0..1
Crosswords WordsVG	65	$2-20$	$3-7360$	0..25
Crosswords LexVG	63	$5-20$	$49-7360$	0..25
Crosswords Wordspuzzle	22	$2-13$	$1-4042$	0..25
Dubois	12	3	4	0..1
Geom	100	2	approx. 300	1..20
K5	10	5	approx. 19000	0..9
Kakuro Easy	172	$2-9$	$2-362880$	1..9
Kakuro Medium	192	$2-9$	$2-362880$	1..9
Kakuro Hard	187	$2-9$	$2-362880$	0..9
Langford 2	20	2	$1-1722$	Vary from 0..5 to 0..41
Langford 3	16	2	$3-2550$	Vary from 0..5 to 0..50
Langford 4	16	2	$5-2652$	Vary from 0..7 to 0..51
MDD 05	25	7	approx. 29000– approx. 57000	0..4
MDD 07	9	7	approx. 40000	0..4
MDD 09	10	7	approx. 40000	0..4
Mod Renault	50	$2-10$	$3-48721$	0..41 or smaller
Nonograms	180	2	$1-1562275$	1..15 to 1..980
Pigeons Plus	40	2, some higher	$10-390626$	0..9 or smaller
Rands JC2500	10	7	2500	0..7
Rands JC5000	10	7	5000	0..7
Rands JC7500	10	7	7500	0..7
Rands JC10000	10	7	10000	0..7
TSP 25	15	2, a few 3	$25-23653$, a few 1	Vary from singleton to 0..1000
TSP Quat 20	15	2, a few 3	$380-23436$, a few 1	Vary from singleton to 0..1000

All instances were written in MiniZinc [?], the instances used in [?] were originally written in XCSP2.1, but compiled into MiniZinc. Of the 1621 instances that were used in [?], only 1507 could be used due to parse errors. The benchmarks series and their characteristics are presented in Table 1. The experiments were run under Gecode 5.0 on 16-core machines 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 shared servers.

The performance of different versions of CT were compared, and the winning version was compared against the existing propagators for the TABLE constraint in Gecode.

The following section presents the results of the experiments.

First the results of comparing different versions of CT are presented, and then the results of comparing the seemingly best version of CT with the existing propagators in Gecode for the TABLE constraint.

5.1 Comparing different versions of CT

5.1.1 Evaluation Setup

Four different versions of CT were compared on a subset of the benchmarks series listed in Table 1. A timeout of 1000 seconds was used and each instance was run once for each version. Todo: run them several times and compute the average. The versions and their denotations are:

CT Basic version.

CT(Δ) CT using Δ_x, the set of values that has been removed from $\text{dom}(x)$ since last execution, as described in Algorithm 6.

CT(T) CT that explicitly stores the initial valid table T as a field and fixes the domains of the variables to the last valid tuple, as described in Algorithm 7.

CT(B) CT that during propagation reasons about the bounds of the domains before enforcing domain consistency, an implementation detail discussed in Section 4.

5.1.2 Results

The plots from the experiments are presented in Appendix A.

5.1.3 Discussion

From the results we see that CT(Δ) outperforms the other variants. The performance of CT and CT(T) is the same, and CT(B) is overall slower than the other variants. On *AIM-50*, which only contain instances with 0/1 variables, the performance of CT, CT(Δ), and CT(B) is the same, which is expected because they collapse to the same variant for domains of size 2.

5.2 Comparing CT against existing propagators

Gecode provides an EXTENSIONAL constraint, which comes with three different propagators: one where the extension is given as a DFA, one non-incremental memory-efficient one where the extension is given as a tuple set, and one incremental time-efficient one where the extension is also given as a tuple set.

DFA This is based on [?].

B – Basic positive tuple set propagator This is based on [?].

Add pseudocode for B.

I – Incremental positive tuple set propagator This is based on explicit support maintenance. The propagator state has the following fields, where a *literal* is a $\langle x, n \rangle$ pair.

- array of variables: X
- tuple set: T
- $L[\langle x, n \rangle]$ is the latest seen tuple where position x has value n. Initialized to the first such tuple, and set to $\perp$ after the last such tuple has been processed.
- $S[\langle x, n \rangle]$ is a set of encountered supports (tuples) for $\langle x, n \rangle$. Initialized to $\emptyset$.
- W_S is a stack of literals, whose support data needs restoring. Initially empty.
- W_R is a stack of literals no longer supported, and whose domain therefore needs updating and whose support data need clearing. Initially empty.

Algorithm 9 shows the algorithm for the incremental tuple set propagator. When the propagator is being posted, FindSupport($\langle x, n \rangle$) is called for every literal $\langle x, n \rangle$. Lines 6 – 8 are executed in an advisor, and they call RemoveSupport($\langle x, n \rangle$) for every literal $\langle x, n \rangle$ that has been removed since last time. The rest of the algorithm removes all the literals in W_R and calls FindSupport($\langle x, n \rangle$) for all literals $\langle x, n \rangle$ in W_S whose support data needs restoring.

PROCEDURE Extensional() : *bool*

1: **if** the propagator is being posted **then** // executed in a constructor
2:     **foreach** $x \in X$ **do**
3:         **foreach** $n \in D(x)$ **do**
4:             FindSupport($\langle x, n \rangle$)
5: **else** // executed in an advisor
6:     **foreach** $\langle x, n \rangle$ that has been removed since last time **do**
7:         **foreach** $t \in S[\langle x, n \rangle]$ **do**
8:             RemoveSupport($t, \langle x, n \rangle$)
9: **while** $W_R \neq \emptyset \vee W_S \neq \emptyset$ // executed in the propagator proper
10:     **foreach** $\langle x, n \rangle \in W_R$ **do**
11:         $D(x) \leftarrow D(x) \setminus \{n\}$
12:         **if** $D(x)$ was wiped out **then**
13:             **return false**
14:     $W_R \leftarrow \emptyset$
15:     **foreach** $\langle x, n \rangle \in W_S$ where $n \in D(x)$ **do**
16:         FindSupport($\langle x, n \rangle$)
17:     $W_S \leftarrow \emptyset$
18: **return true**

Algorithm 9: Incremental positive tuple set propagator.

Algorithm 10 finds a tuple that supports a given literal $\langle x, n \rangle$. If no such exists, then the literal is added to W_R, else the tuple is added to the set of encountered valid tuples for the literals associated with the tuple.

```
PROCEDURE FindSupport(⟨x, n⟩)
ℓ ← L[⟨x, n⟩]
while ℓ ≠ ⊥ ∧ ∃y ∈ X : ℓ[y] ∉ D(y)
    ℓ ← L[⟨x, n⟩] ← next tuple for ⟨x, n⟩
if ℓ = ⊥ then
    W_R ← W_R ∪ {⟨x, n⟩}
else
    foreach y ∈ X do
        S[⟨y, ℓ[y]⟩] ← S[⟨y, ℓ[y]⟩] ∪ {ℓ}
```

Algorithm 10: Recheck support for literal $\langle x, n\rangle$.

Algorithm 11 clears the support data for a tuple l that has become invalid, by removing l from the set of valid tuples for each variable. The associated literals are also added to W_S, because support data for them need to be restored.

```
PROCEDURE RemoveSupport(ℓ, ⟨x, n⟩)
foreach y ∈ X do
    S[⟨y, ℓ[y]⟩] ← S[⟨y, ℓ[y]⟩] \ {ℓ}
    if y ≠ x ∧ S[⟨y, ℓ[y]⟩] = ∅ then
        W_S ← W_S ∪ {⟨y, ℓ[y]⟩}
```

Algorithm 11: Clear support data for unsupported literal $\langle x, n\rangle$. Note: n is actually not used here.

5.2.1 Evaluation Setup

The winning variant from the experiments in Section 5.1, CT(Δ), was compared against the two existing propagators in Gecode for the TABLE constraint, as well with the propagator for the REGULAR constraint on the benchmarks series listed in Table 1. The propagators are denoted:

CT The Compact Table Propagator, version CT(Δ).

DFA Layered graph (DFA) propagator, based on [**?**].

B Basic positive tuple set propagator, based on [**?**].

I Incremental positive tuple set propagator.

5.2.2 Results

The plots from the experiments are presented in Appendix B.

5.2.3 Discussion

Overall performance. Comparing CT against B and I over all series, CT performs either as well as, or better than both B and I. B or I does not outperform CT on any of the series, except possibly on *AIM-100* and *AIM-200*, where B is marginally faster than CT. On some of the series CT is up to a factor 10 faster.

Comparing CT against DFA, DFA outperforms CT on *MDD 07/09*, *AIM-**, *Pigeons plus* and *TSP 25*. Also, DFA performs well on *Mod Renault*, where a constant time seems to be spent in the initialisation of DFA, while the solvetime is very close to 0 for all instances. The reason might be that DFA is a more suitable algorithm for the problem types in these series, for example if DFA can compress the input table into a space and/or time-efficient DFA in these cases. On the other series however, DFA is outperformed by the other algorithms.

The impact of table size on performance. Most of the series have varying table sizes, and on those series it is hard to analyze the impact of table size on performance. However, looking at the series that do not vary so much in table size, there seems to be a correlation between table size and performance. The increase of performance for CT compared to B and I is larger on the series that contain instances with large table sizes only (see *A5*, *A10*, *K5*, *MDD 05*, and *Rands JC**), than on the series that contain only small tables (see *AIM-**, *Dubois*, and *Geom*).

The property shows particularly well on the four *Rands JC** series, where arity and domain size are constant while the table size increase from 2500 to 10000 in steps of 2500. On these series, the performance gain seems to increase with an increasing table size.

The impact of arity on performance. Again, many of the series have varying arities, so to analyze the impact of arity we are limited to the series that do not vary so much in arity. Many series where CT shows little or none performance gain have constraints with low arities (see *AIM-**, *Dubois*, *Geom*, *Langford **), though there are exceptions to this (see *Pigeons Plus*, *TSP 25/Quat*). However, the series with low arities also have small tables, while the series with larger arities tend to have larger tables, which makes it hard to tell whether the it is the arity or the table size that impact the performance gain.

The impact of domain size on performance. It is hard to draw any conclusions of whether the domain size affect the performance gain of CT. Among the series with small domain sizes, some have little or no performance gains (see *AIM-**, *Dubois*) and some have a large performance gains (see *MDD 05*, *BDD Large*). The same is true for the series with larger domain sizes; some have modest performance gains (see *Nonograms*, *Kakuro **), while some have larger performance gain (see *Rands JC**, *Crosswords **).

Runtime vs. Solvetime. Both the runtime and the solvetime was measured. The runtime is the solvetime plus the parsing of the FlatZinc file as well as the posting of the propagators. The discrepancy between the runtime and solvetime is different for the various algorithms. It is largest for DFA, which shows that the initialisation of DFA takes longer time compared to the other algorithms. CT has a larger discrepancy than B and I, so initialising CT takes longer time than initialising B and I. The reason could be that CT performs more initial propagation than B and I, or that the initialisation of the data structures is more time-consuming for CT. Todo: Check if CT performs more initial propagation.

6 Conclusions and Future Work

In this bachelor thesis project, a new propagator algorithm for the Table constraint, called Compact Table (CT), was implemented in the constraint solver Gecode, and its performance was evaluated compared to the existing propagators for Table in Gecode, as well as the propagator for the Regular constraint. The results of the evaluation is that CT outperforms the existing propagators in Gecode for Table, which suggests that CT should be included in the solver. The performance gains from CT seems to be largest for constraints with large tables, and more modest for constraints with low arities.

For the implementation to reach production quality, there are a few things that need to be revised. The following lists some known improvements and flaws.

- There is an unfound error that causes a crash due to corrupt data in very few cases. The most likely cause of this error is that some allocated memory area is too small, and that the data is modified outside of this area.
- Some memory allocations in the initialisation of the propagator depend on the domain widths rather than the domain sizes of the variables. This is unsustainable for pathological domains such as $\{1, 10^9\}$. In the current implementation, a memory block of size 10^9 is allocated for this domain, but ideally it should not be necessary to allocate more than 2 elements. Though the problem seems trivial, it requires some work, because of indexing problems.
- The threshold value for when to use a hash table versus an array for indexing the supports should be calibrated with experiments.
- In the variant using delta information, the current implementation uses the incremental update if $|\Delta_x| < |s(x)|$. It is possible that this condition can be generalised to $|\Delta_x| < k \cdot |s(x)|$, where $k \in \mathbb{R}$, something that remains to be investigated.
- Implement the generalisations of the CT algorithm described in [**?**].

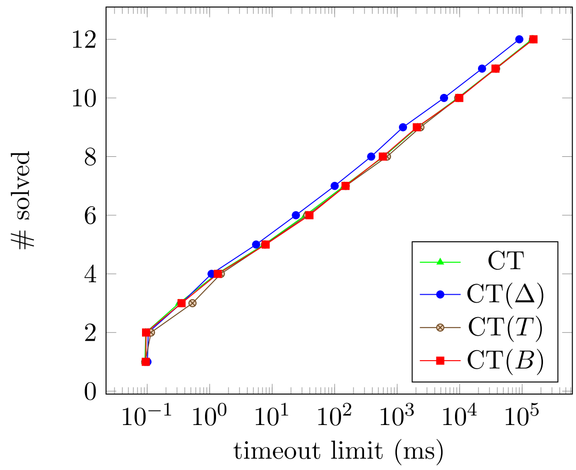

Figure 4: **Langford 4**.

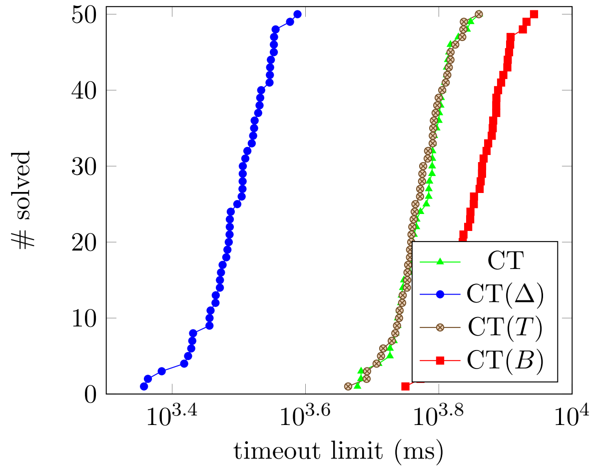

Figure 5: **A5**.

A
Plots from comparison of different versions of CT

Each plot shows the number of instances solved as a function of timeout limit in milliseconds. The measured time is the total runtime, including parsing of the FlatZinc file and the posting of the propagators.

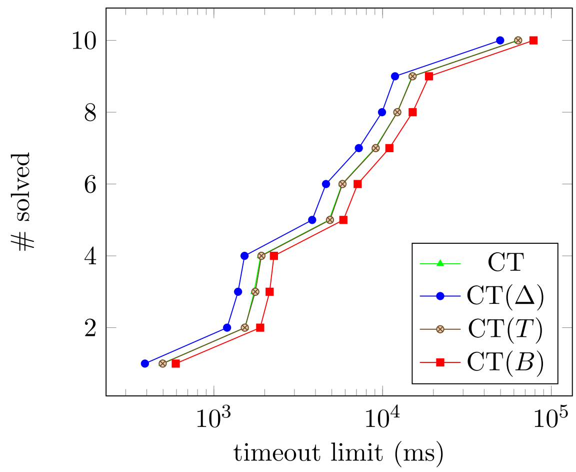

Figure 2: **Rands JC2500.**

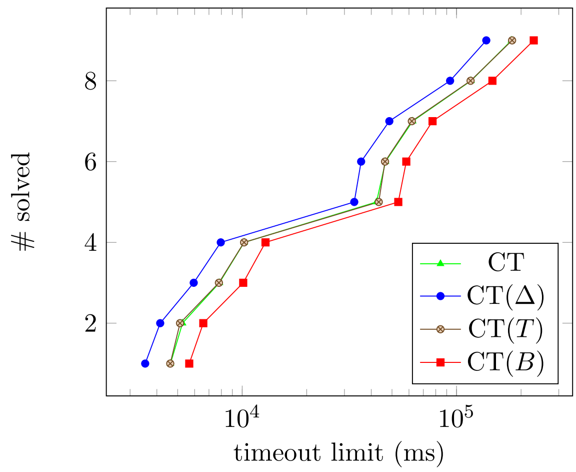

Figure 3: **Rands JC5000**.

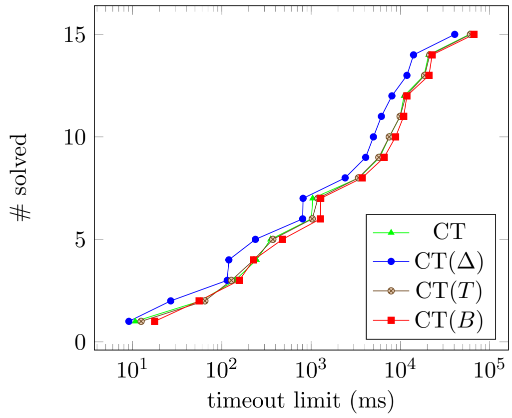

Figure 6: **TSP Quat 20**.

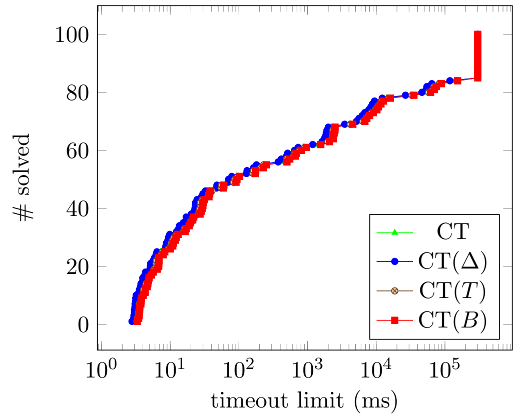

Figure 7: **Geom**.

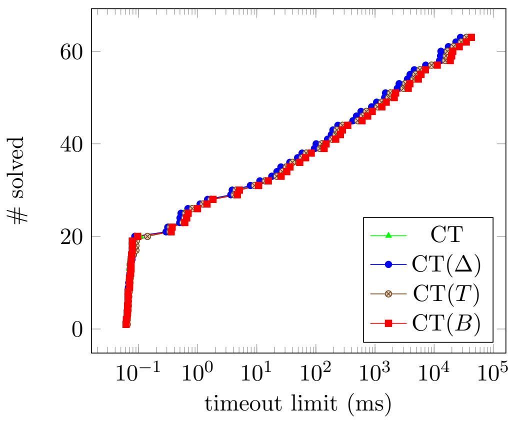

Figure 8: **Crosswords LexVG**.

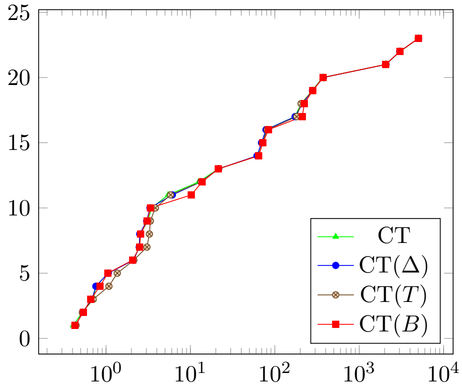

Figure 9: **AIM 50**.

B
Plots from comparison of CT against existing propagators

Each plot shows the number of instances solved as a function of timeout limit in milliseconds. The leftmost column shows runtime, the rightmost column shows solvetime.

Todo: One figure per serie, referring to left/right does not work.

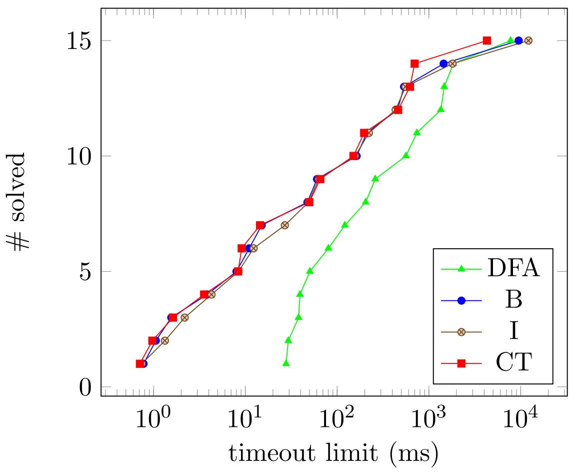

Figure 10: **Langford 2**.

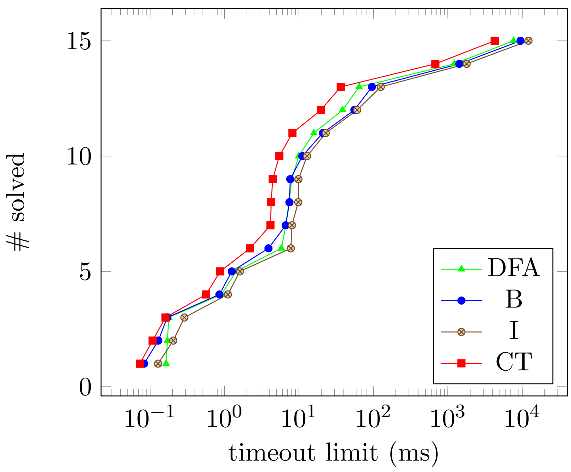

Figure 11: **Langford 2**.

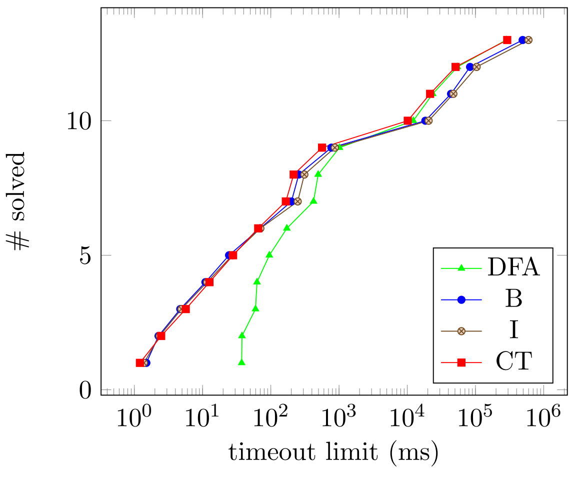

Figure 12: **Langford 3**.

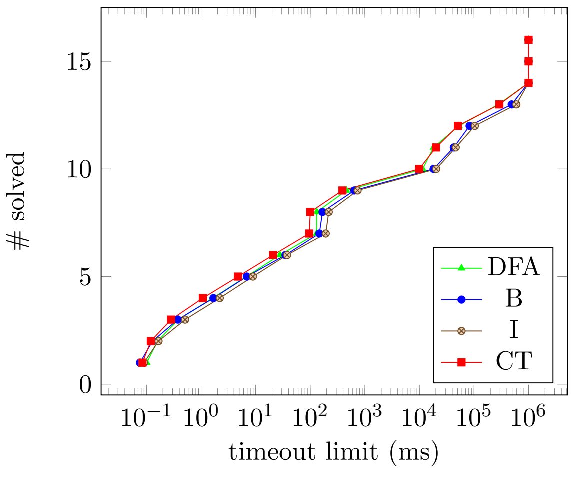

Figure 13: **Langford 3**.

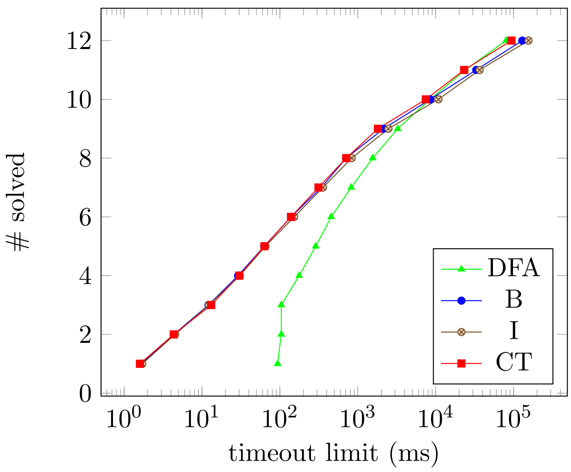

Figure 14: **Langford 4**.

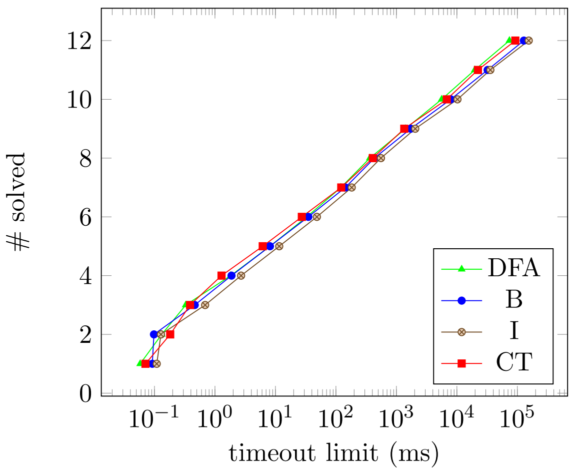

Figure 15: **Langford 4**.

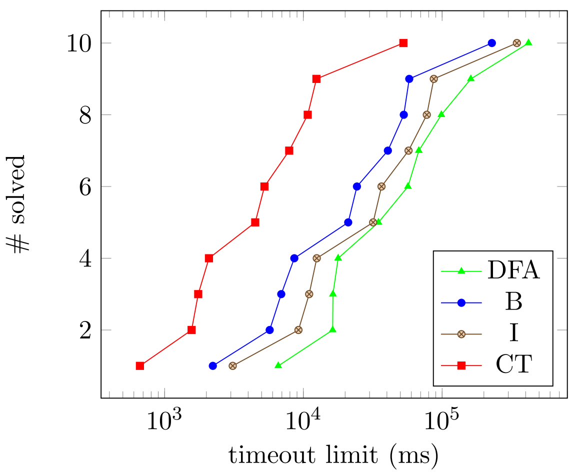

Figure 16: **Rands JC2500.**

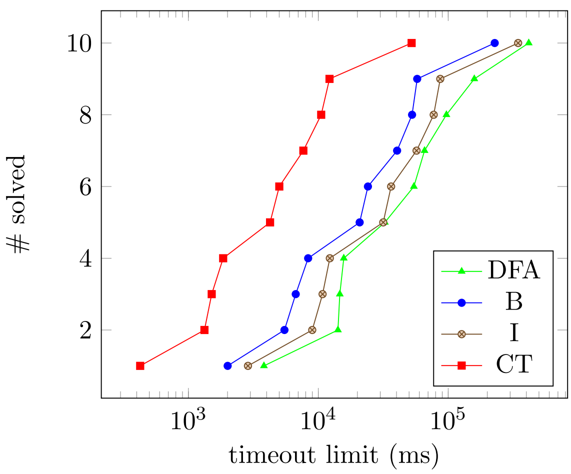

Figure 17: **Rands JC2500.**

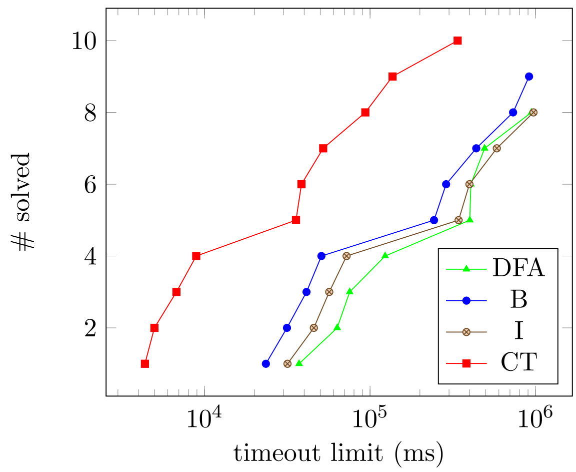

Figure 18: **Rands JC5000.**

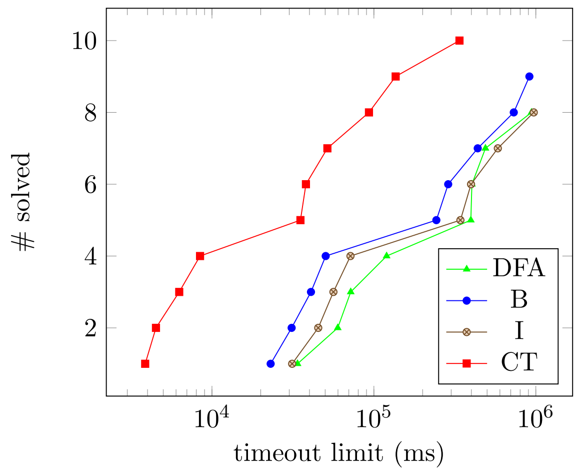

Figure 19: **Rands JC5000.**

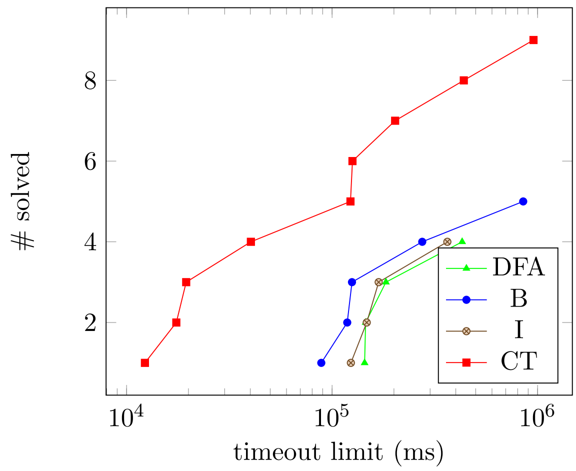

Figure 20: **Rands JC7500**.

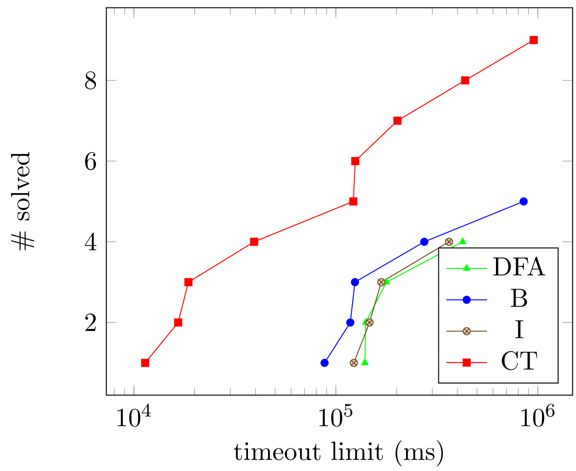

Figure 21: **Rands JC7500**.

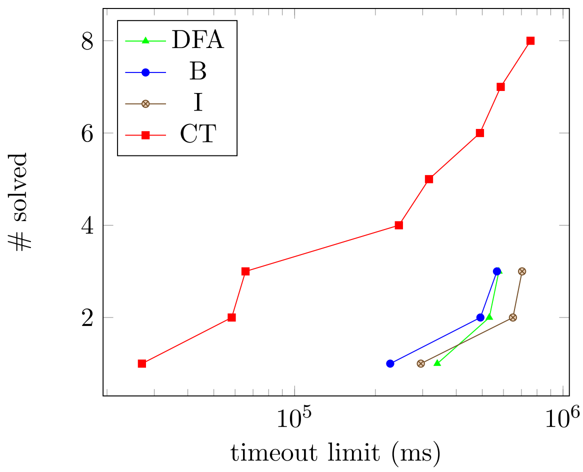

Figure 22: **Rands JC10000**.

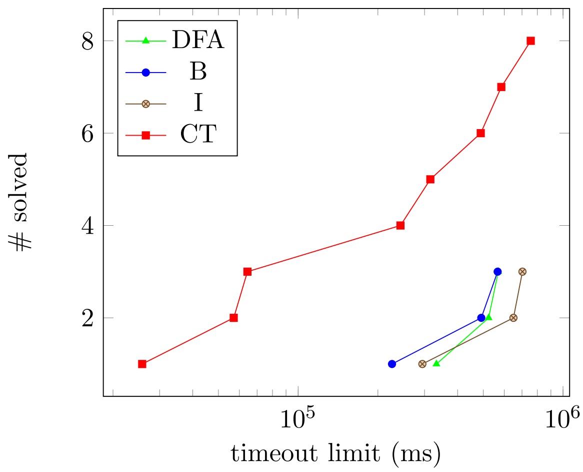

Figure 23: **Rands JC10000**.

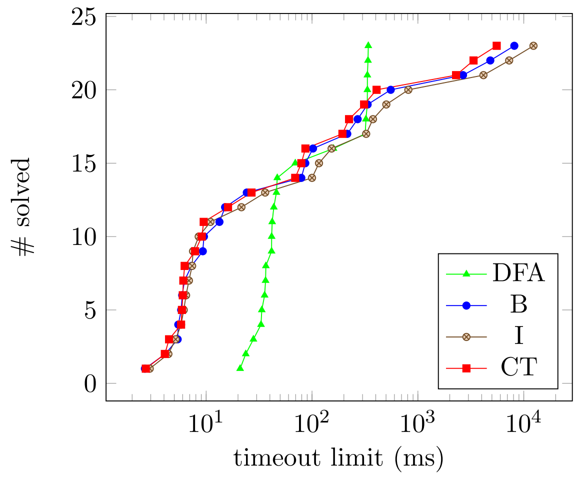

Figure 24: **AIM-50**.

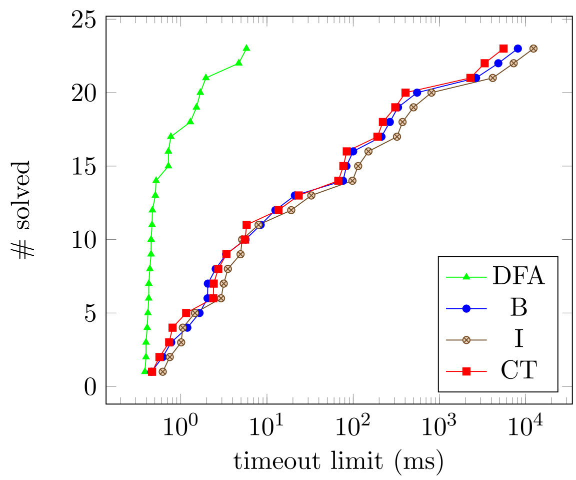

Figure 25: **AIM-50**.

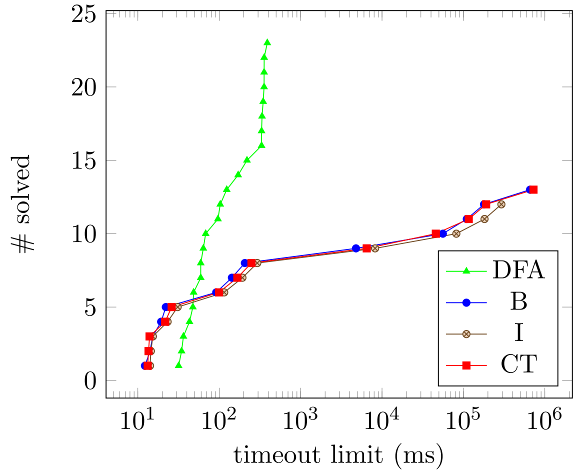

Figure 26: **AIM-100**.

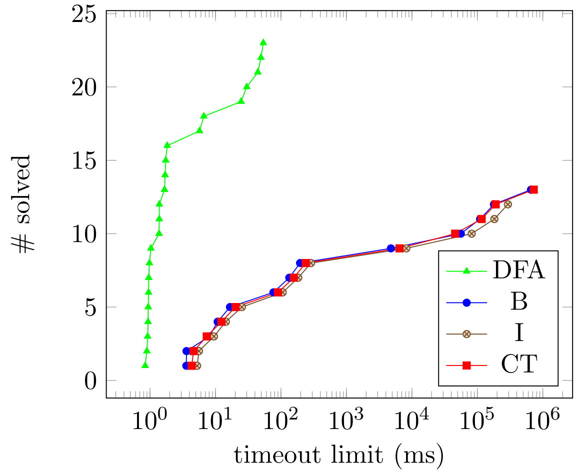

Figure 27: **AIM-100**.

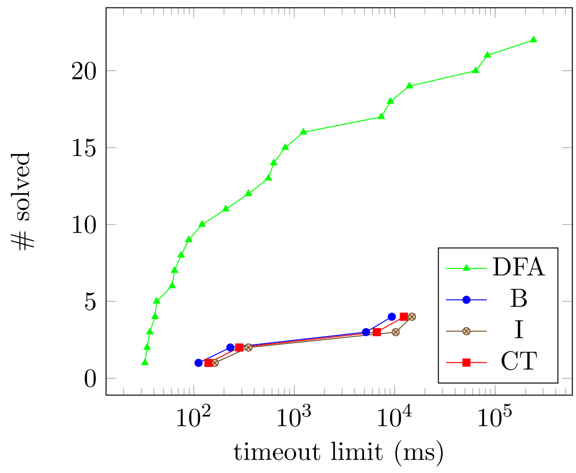

Figure 28: **AIM-200**.

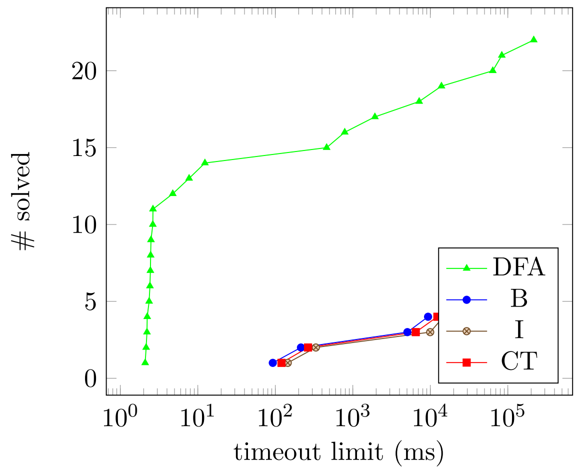

Figure 29: **AIM-200**.

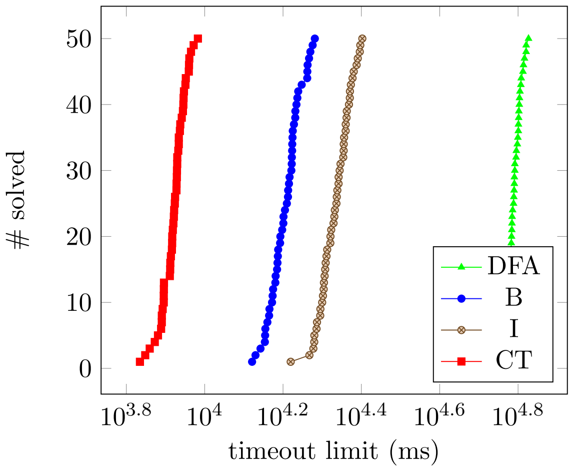

Figure 30: **A5**.

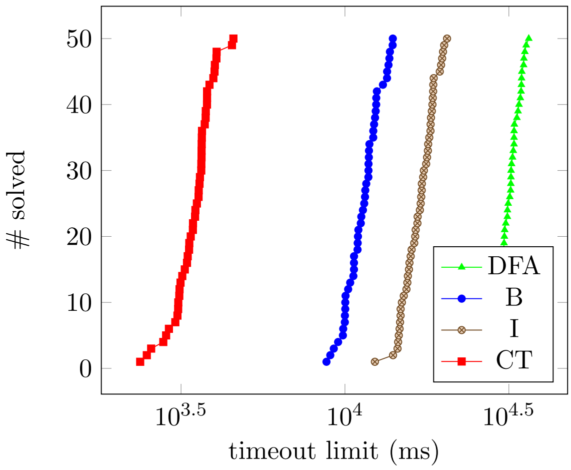

Figure 31: **A5**.

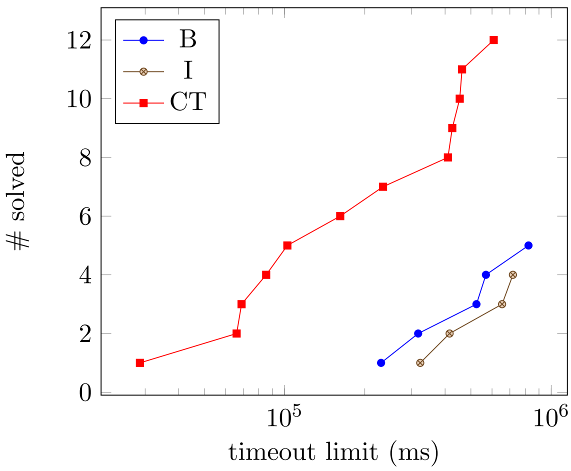

Figure 32: **A10**.

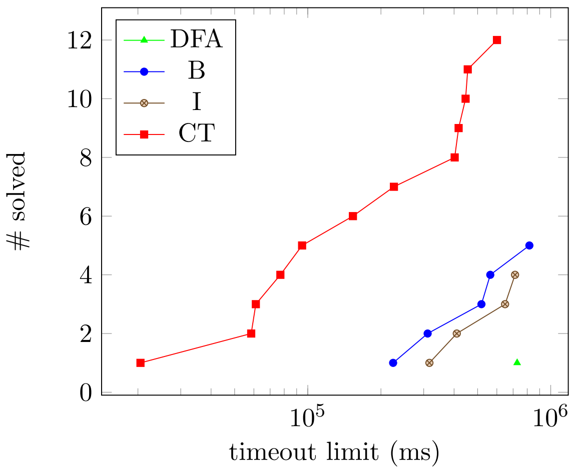

Figure 33: **A10**.

Ωb

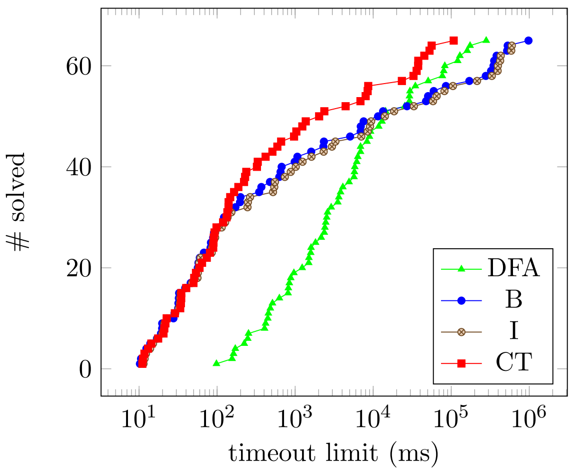

Figure 34: **Crosswords WorldVG**.

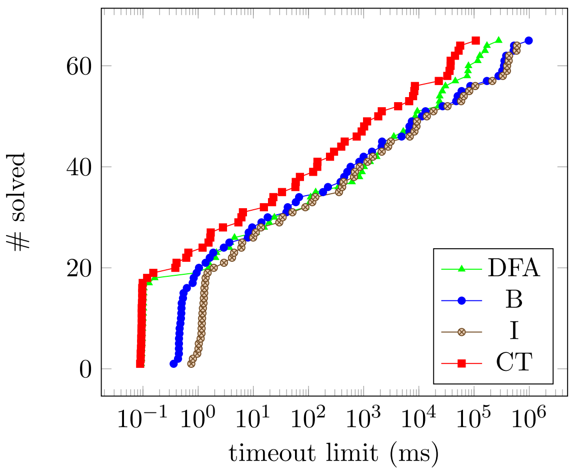

Figure 35: **Crosswords WorldVG**.

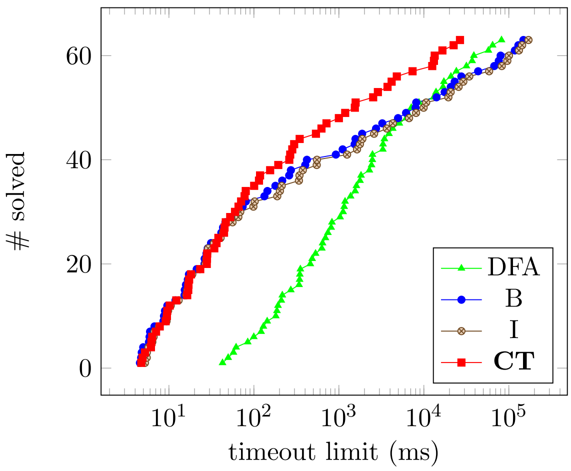

Figure 36: **Crosswords LexVG**.

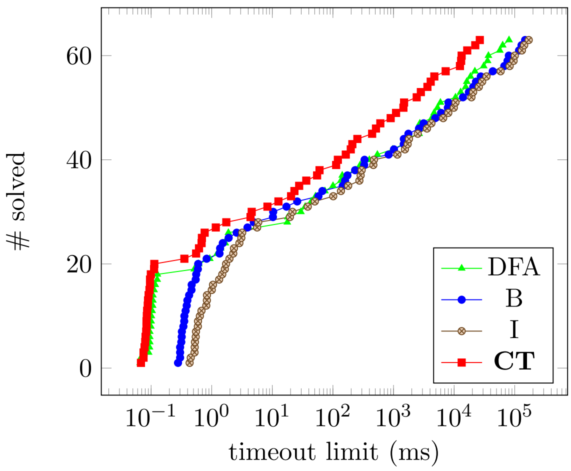

Figure 37: **Crosswords LexVG**.

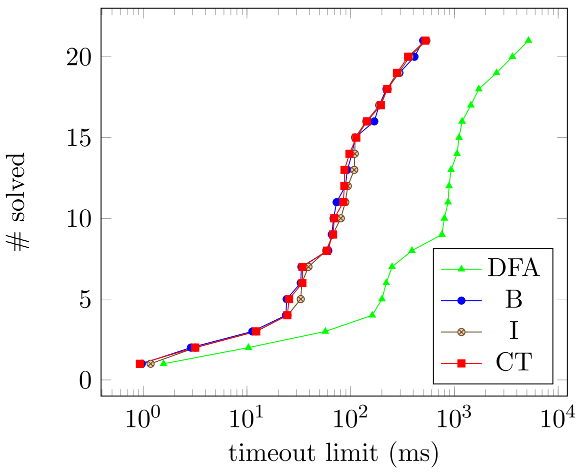

Figure 38: **Crosswords Wordspuzzle**.

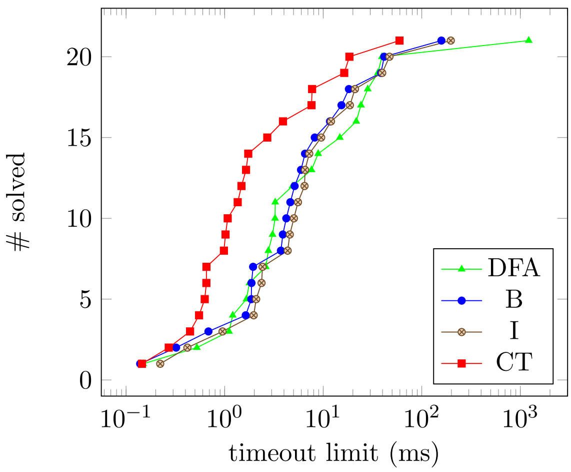

Figure 39: **Crosswords Wordspuzzle**.

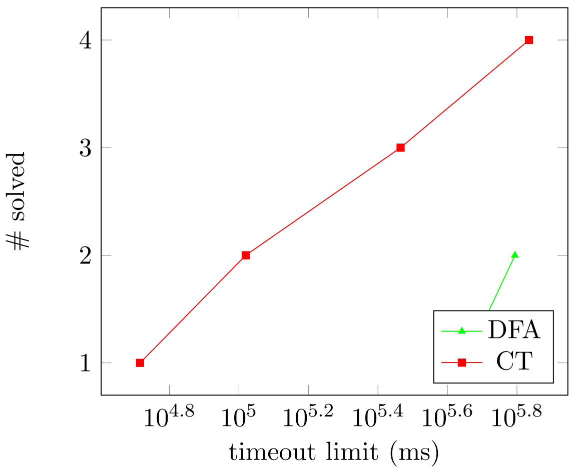

Figure 40: **MDD 05**.

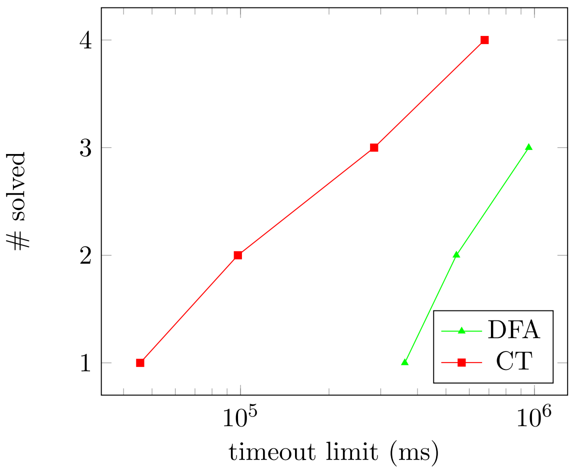

Figure 41: **MDD 05**.

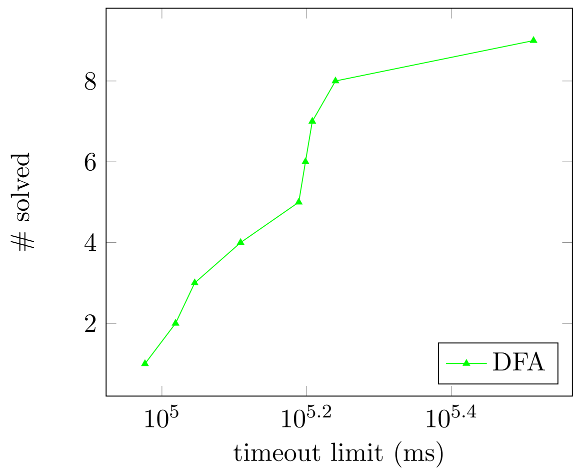

Figure 42: **MDD 07**.

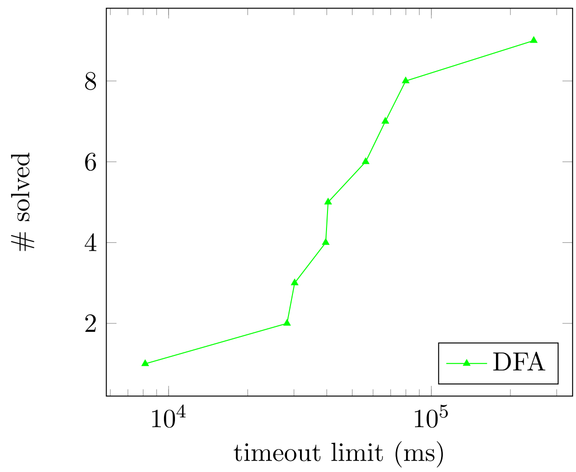

Figure 43: **MDD 07**.

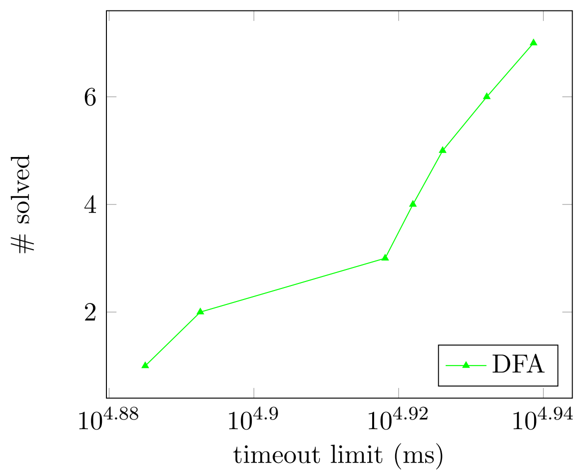

Figure 44: **MDD 09**.

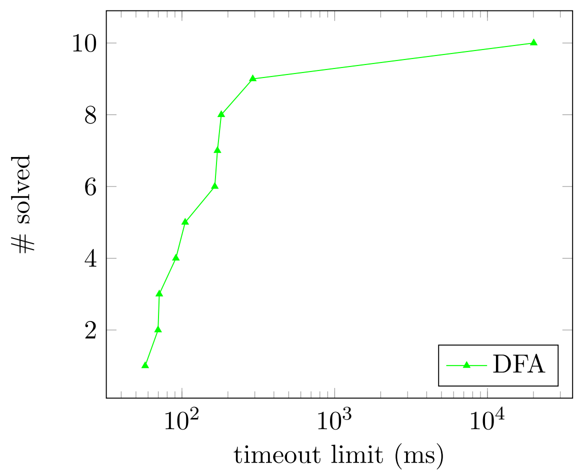

Figure 45: **MDD 09**.

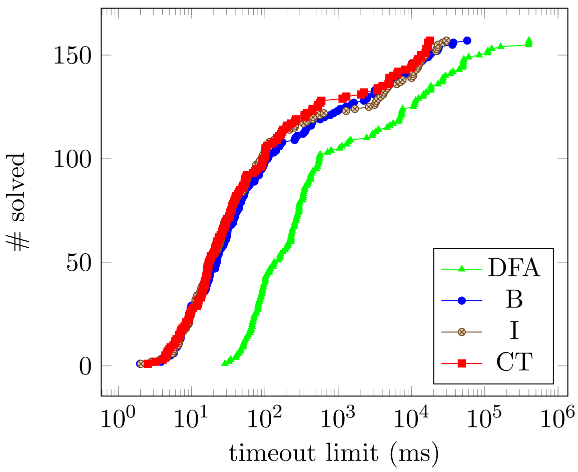

Figure 46: **Kakuro easy**.

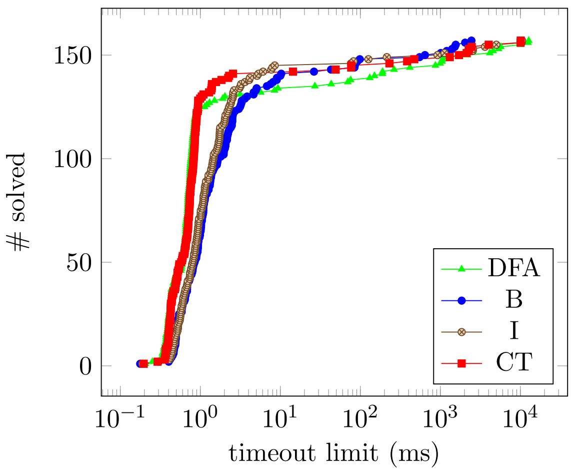

Figure 47: **Kakuro easy**.

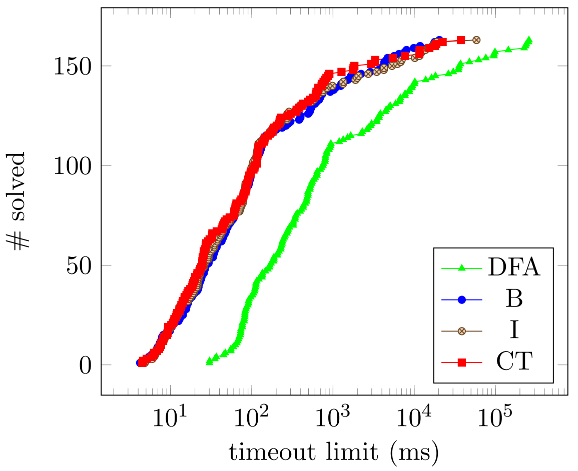

Figure 48: **Kakuro Medium**.

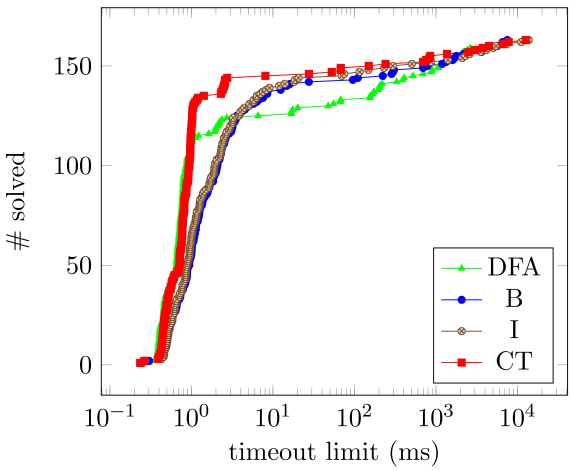

Figure 49: **Kakuro Medium**.

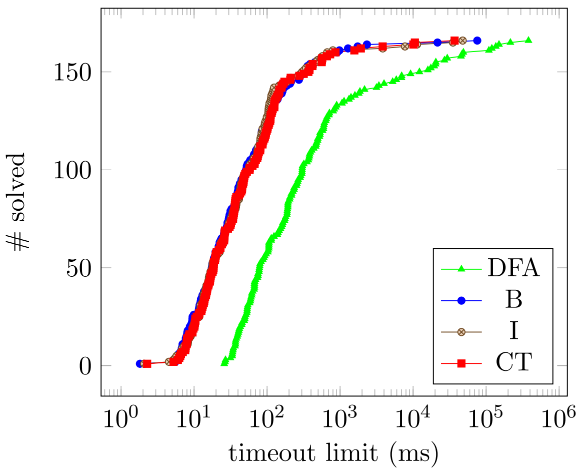

Figure 50: **Kakuro Hard**.

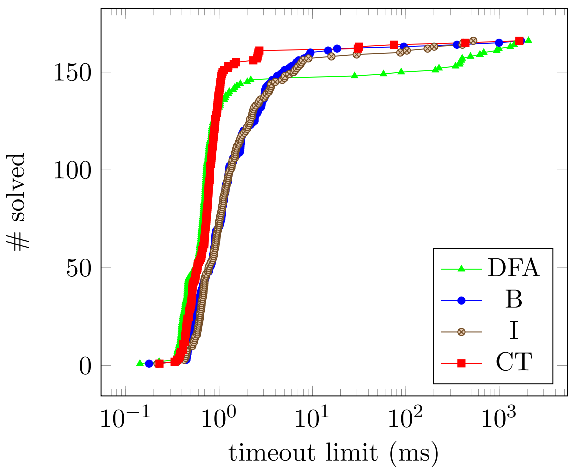

Figure 51: **Kakuro Hard**.

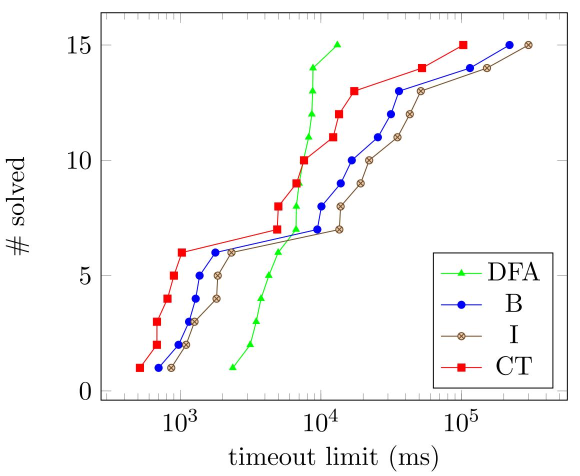

Figure 52: **TSP 25**.

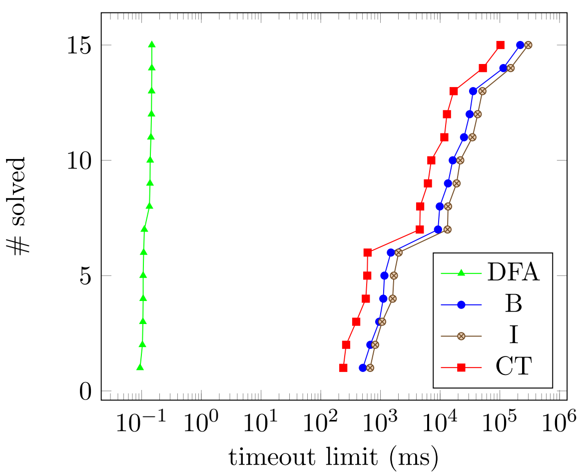

Figure 53: **TSP 25**.

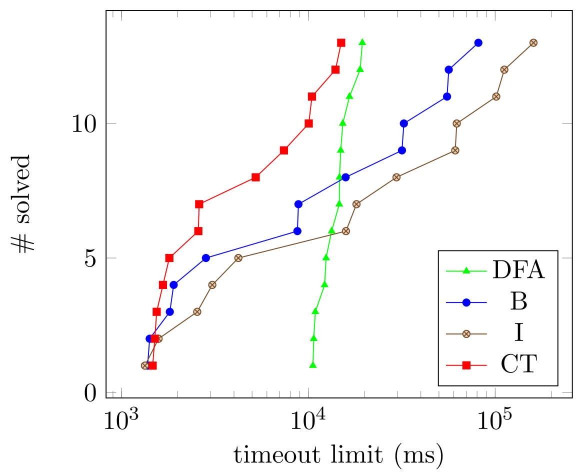

Figure 54: **TSP Quat 20**.

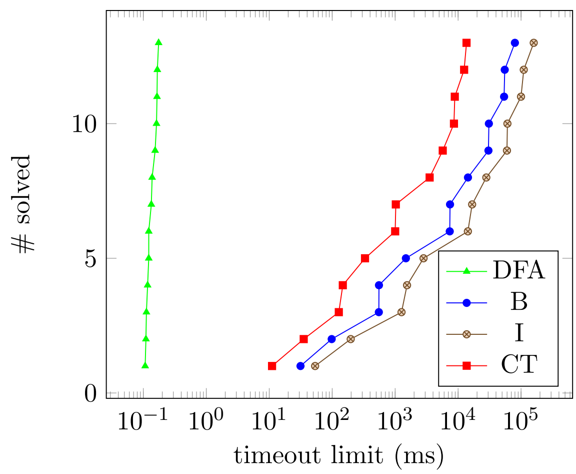

Figure 55: **TSP Quat 20**.

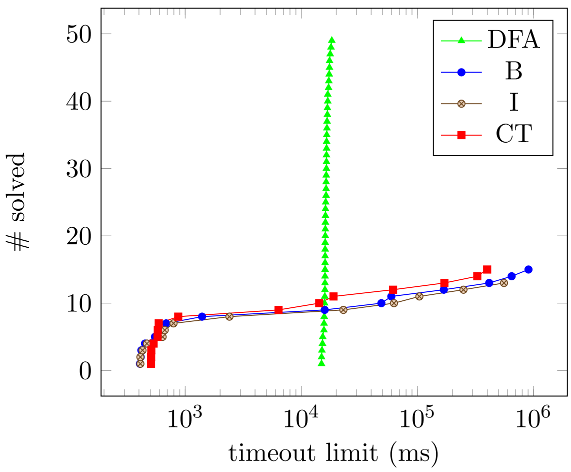

Figure 56: **Mod Renault**.

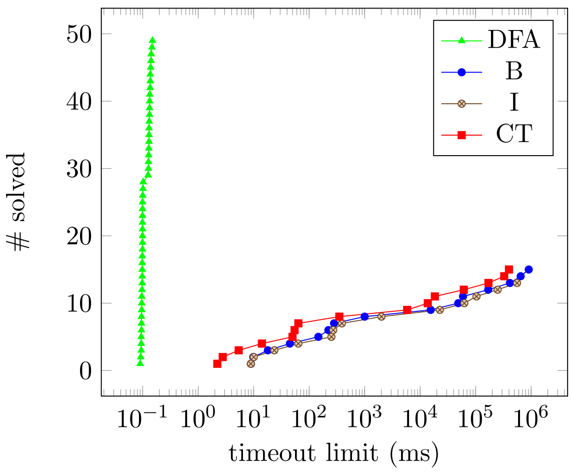

Figure 57: **Mod Renault**.

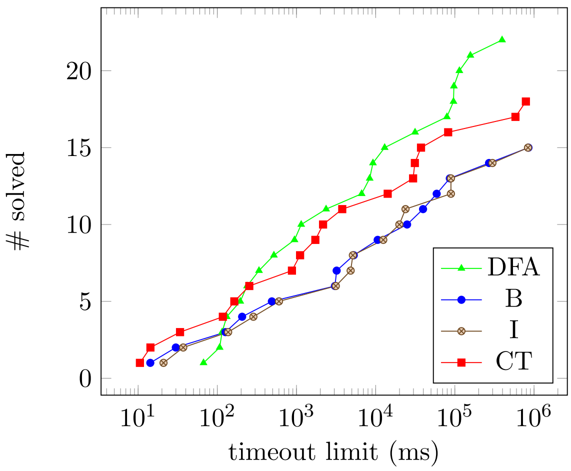

Figure 58: **Pigeons Plus**.

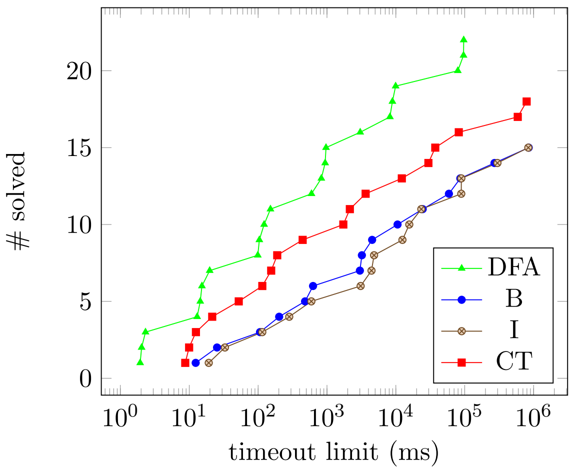

Figure 59: **Pigeons Plus**.

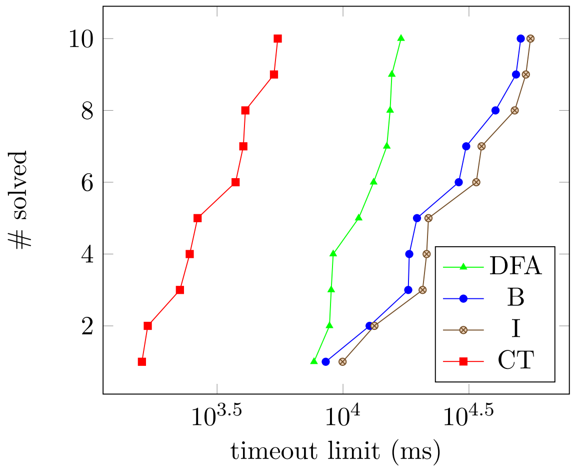

Figure 60: **K5**.

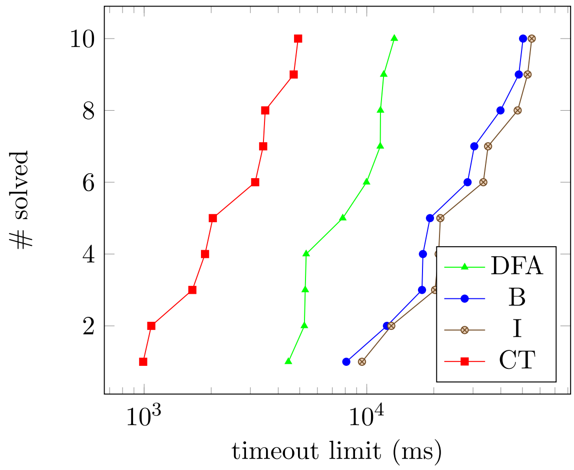

Figure 61: **K5**.

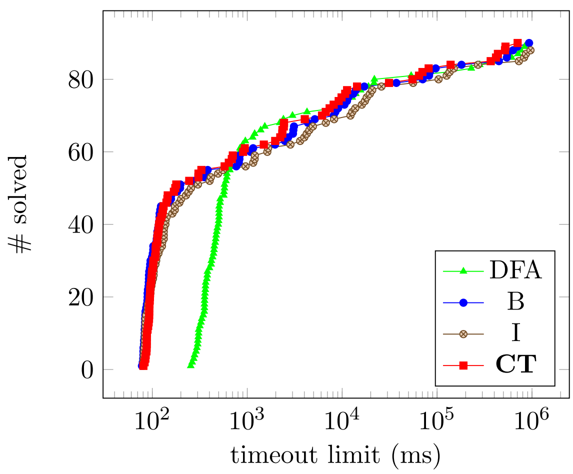

Figure 62: **Geom**.

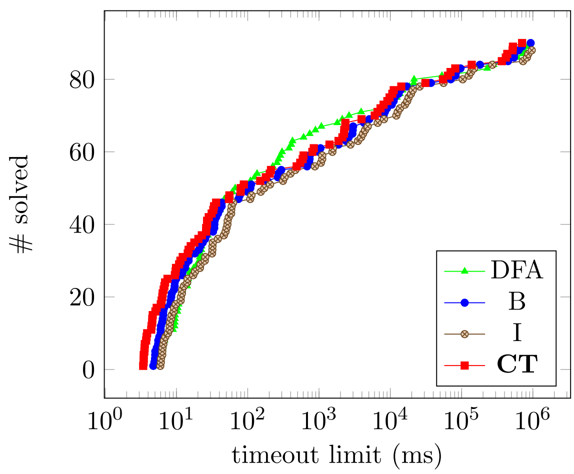

Figure 63: **Geom**.

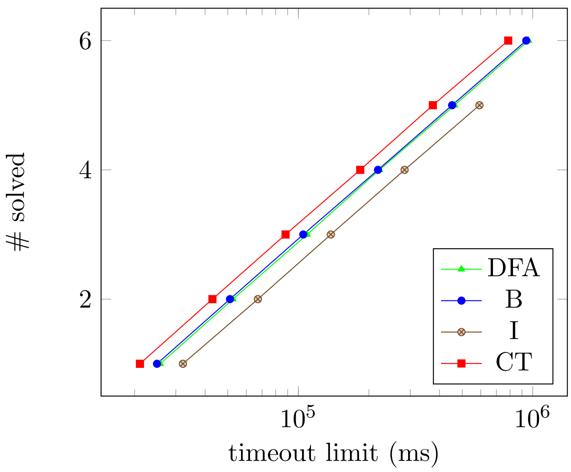

Figure 64: **Dubois**.

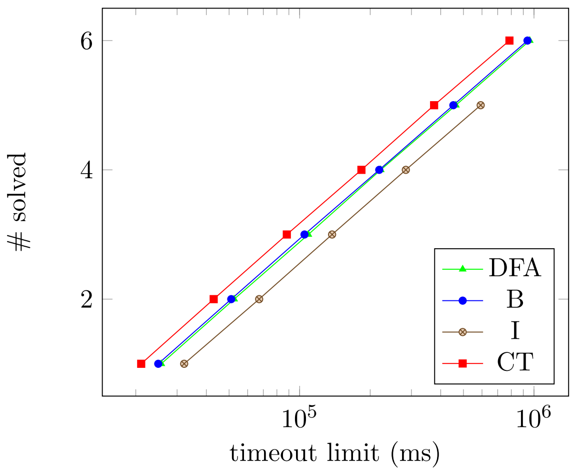

Figure 65: **Dubois**.

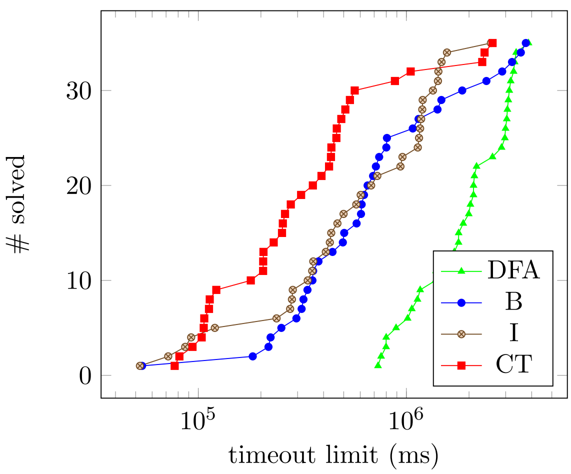

Figure 66: **BDD Large**.

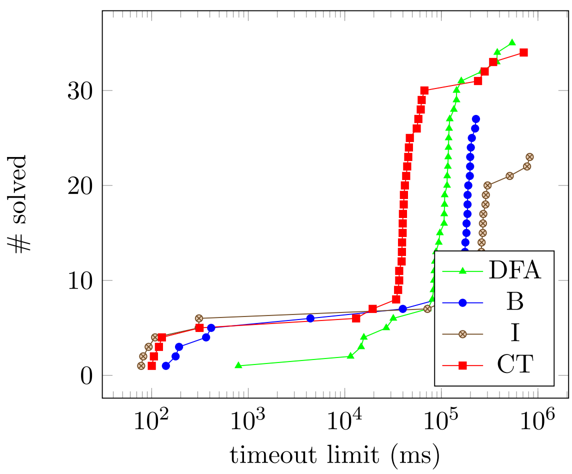

Figure 67: **BDD Large**.

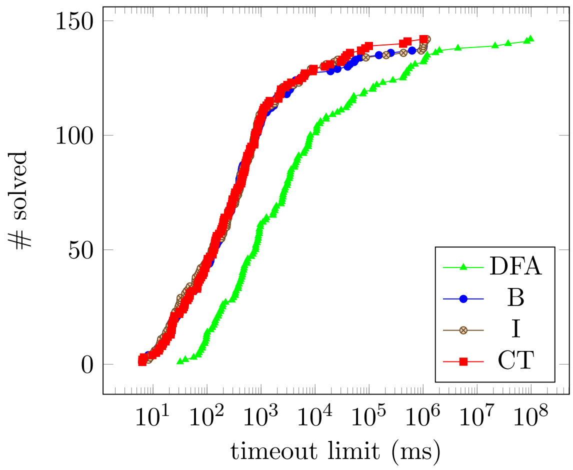

Figure 68: **Nonograms**.

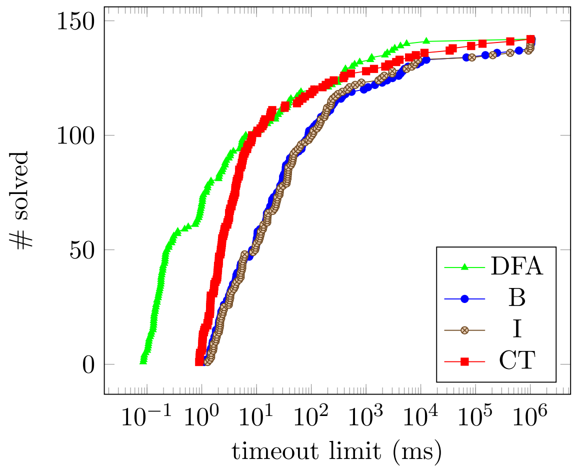

Figure 69: **Nonograms**.