Implementation and Evaluation of a Compact Table Propagator in Gecode

Linnea Ingmar

10th May 2017

Contents

1 Introduction

Constraint Programmint (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. Each variable is to be assigned a value from its domain, 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, impossible 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 Tableconstraint was published [**?**], called Compact Table (CT). Preliminary results indicate that CT outperforms the 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++, and combines state-of-the-art performance with modularity and extensibility. Gecode has three existing propagation algorithms for Table. Before this project there had been no attempt to implement CT in Gecode and consequently its performance in Gecode was unknown. The purpose of this thesis is to implement CT in Gecode and to evaluate and compare its performance with previously 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 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

Now follows the contributions made by this bachelor thesis, which also works 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 **??**.
- 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 **??**.
- Several possible improvements and flaws have been detected in the current implementation that prevent the code to meet commercial standards, these are listed in Section **??**.

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 *constrains* 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. Each variable is to be assigned a value from its domain, 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 solving of a problem consists of generating a search tree by branching on possible values for the variables. At each node in the search tree, the solver removes impossible values from the domains of the variables. This filtering process is called *propagation*. Each constraint is associated with at least one propagator algorithm, whose task is to detect 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 pruning the size of the search tree.

To build intuition and understand the ideas of CP, the concepts can be demonstrated with logical puzzles. One such puzzle is Kakuro, somewhat similar to the popular puzzle Soduko, 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 size of the board can vary. The objective is to fill in digits from 1 to 9 inclusive into each cell such that for each entry, the sum of all the digits in the entry is equal to the clue of that entry, and such that each digit appears at most once in each entry.

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

An alternative way of phrasing the constraints of Kakuro, is for each entry explicitly list all the possible combinations of values that the variables in that entry can take. For example,

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

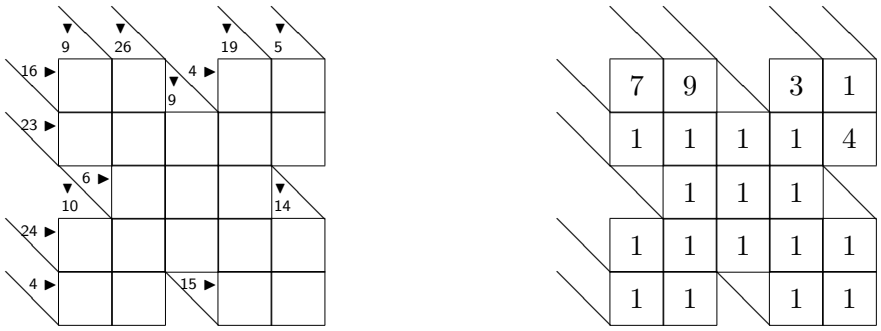

Figure 1: A Kakuro puzzle [1](left) and its solution (right).

consider an entry of size 2 with clue 4. The only possible combinations of values are $\langle 1, 3\rangle$ and $\langle 3, 1\rangle$, since these are the only tuples of 2 distinct digits whose sums are equal to 4. This way of listing the possible combinations of values for the variables is in essence the TABLE constraint – the constraint that is addressed in this thesis.

Really solve the Kakuro in Figure 1. Kvllspyssel!

After gaining some intuition of CP, now follows some formal definitions. The definitions are 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 $dom(x_i)$.*

- *A* constraint *c on X is a relation, denoted by $rel(c)$. The associated variables X are denoted $vars(c)$, and we call $|vars(c)|$ the* arity *of c. The relation $rel(c)$ contains the set of n-tuples that are allowed for X, we call those n-tuples* solutions *to the constraint c.*
- *For an n-tuple $\tau = \langle a_1, \ldots, a_n\rangle$ 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 a *constraint satisfaction problems*, which is a central concept.

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

During the search of 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 was found) or until the domain of a variable becomes empty (a failure). Upon a failure, search backtracks to a node in the search tree where all branches are not tried yet, and the domains of the variables are restored to the domains that the variables had in that node. A current mapping of domains to variables is called a *store*:

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

- *A store s is* failed *if and only if $s(v_i) = \varnothing$ for some $v_i \in V$.*
- *A variable $v_i \in V$ is* fixed*, or* assigned*, by a store s if and only if $|s(v_i)| = 1$.*
- *A store s is an* assignment *store if all variables are fixed under s.*
- *Let c be an n-ary constraint on V. A store s is a* solution store *to c if and only if s is an assignment store and the corresponding n-tuple is a solution to c: $\forall i \in \{1, \ldots, n\}, s(v_i) = \{a_i\}$, and $\langle a_1, \ldots, a_n \rangle$ is a solution to c.*
- *A store s_1 is* stronger *than a store s_2, written $s_1 \preceq s_2$ if and only if $s_1(v) \subseteq s_2(v)$ for all $v \in V$.*

2.2 Propagation and propagators

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

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

- *A propagator p is a decreasing function: $p(s) \preceq s$ for any store s. This property guarantees that constraint propagation only removes values.*
- *A propagator p is a monotonic function: $s_1 \preceq s_2 \Rightarrow p(s_1) \preceq p(s_2)$ for any stores s_1 and s_2. This property guarantees that constraint propagation preserves the strength-ordering of stores.*
- *A propagator is correct for the constraint it implements. A propagator p is correct for a constraint c if and only if it does not remove values that are part of supports for c. This property guarantees that a propagator does not miss any potential solution store.*
- *A propagator is* checking*: for a given assignment store s, the propagator must decide whether s is a solution store or not for the constraint it implements. If s is a solution store, it must signal subsumption, otherwise it must signal failure.*

- *A propagator must be* honest*: it must be* fixpoint honest *and* subsumption honest*. A propagator p is fixpoint honest if and only if it does not signal fixpoint if it does not return a fixpoint, and it is subsumption honest if and only if it does not signal subsumption if it is not subsumed by an input store s.*

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

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

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

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 suppport 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.*

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 applying p to any input store s, the resulting store $p(s)$ 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++, distributed under the MIT license. It has a state-of-the-art performance while being modular and extensible. It supports the programming of new propagators, branching strategies, and search engines. Furthermore, Gecode is well documented and comes with a complete tutorial.

Programming a propagator in Gecode means implementing a C++ object inheriting from the base class Propagator, that complies to the correct interface. The propagator can store any data structures as instance members, for saving information about its state between executions. The interface consists of the following parts [**?**].

Posting. Typical tasks of the posting of the propagator include deciding whether the propagator really needs to be posted, performing some initial propagation and creating an instance of the propagator.

The propagator must also *subscribe* to its associated variables, subscribing to a variable ensures the propagator is scheduled for execution whenever the relevant modification event occurs for that variable. Subscription can also be handled via advisors, see below.

Disposal. The propagator must implement a dispose() method that returns the memory used by the propagator at disposal.

Copying. During search the propagator needs to be copied. This is handled by the copy() method.

Cost computation. Gecode schedules propagators for execution according to their estimated cost. Cheaper propagators execute before more expensive ones, based on the inutition that cheaper propagators might prune values or detect a failure early, so that more expensive propagators can take advantage, or not even need to be executed. Every propagator implements a cost() method that estimates the cost of executing that propagator.

Propagation. The propagator implements a propagate() method that performs the pruning of values. This method must fulfill the obligations in Definition 4.

Rescheduling. For various reasons, a propagator can be disabled, in which case it needs to be rescheduled at a later point, implemented by a reschedule() method.

Gecode provides *advisors* that can inform propagators about variable modifications. Using advisors is optional. An advisor belongs to a propagator and can store information that is needed by its propagator. The purpose of an advisor is to, as its name suggests, advise the propagator of whether it needs to be executed or not. This works as follows. Whenever the domain of the variable(s) that an advisor is subscribed to changes, its advise() method is executed. If the advisor detects that its propagator is still at fixpoint, it should not schedule the propagator. If it is not at fixpoint, or possibly not at fixpoint, the advisor must schedule its propagator for execution. Similarly, the advisor can report failure or subsumption if it detects a failure or that its propagator is subsumed. An advisor is not allowed to modify the domains of the variables. Furthermore, the advise() method can access the modification event that triggered its execution. For integer and boolean variables, some information about which values have been removed is provided.

Search in Gecode is copy-based. Before making a desicion in the search tree, the current node is copied, so that the search can continue from an equivalent state in case the decision fails, or in case more solutions are to be sought for. This implies some concerns on memory usage of the stored data structures of a propagator, since allocating memory and copying large data structures is time consuming, and a too large memory usage is not desirable.

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

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

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

Nevertheless, the TABLE constraint is an important constraint. Typical use cases? Propagator algortihms?

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 representad in a tuple set, which is a data structure in Gecode for representing a set of tuples. Among the last two one of them is based on [**?**], and consumes less memory than the other, which 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. It was first described in [**?**]. Before this project, no attempts to implement CT in Gecode were made, and consquently its performance in Gecode is unknown.

Compact table relies on bitwise operations using a new datastructure 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 bitwise 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 zero words, which makes it efficient to perform bit-wise operations with other bit-sets (such as intersecting and unioning) even though the bit-set is sparse.

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 (Gecode is not among them) use a mechanism called *trailing* to perform backtracking, 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 datastructure comes from the assumption that `words` consists of reversible ints.

In what follows, a data structure that is like 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 what follows, when we refer to an array a, $a[0]$ denotes the first element (indexing starts from 0), a.length() the number of its cells and $a[i:j]$ all its cells in the closed interval $[i, j]$, where $0 \leq i \leq j \leq a.\text{length}() - 1$.

3.1 Sparse Bit-Set

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

```
1: Class SparseBitSet

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

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

12: Method isEmpty() : Boolean
13:   return limit = −1

14: Method clearMask()
15:   Method i ← 0 to limit do
16:      offset ← index[i]
17:      mask[offset] ← 0^64

18: Method reverseMask()
19:   Method i ← 0 to limit do
20:      offset ← index[i]
21:      mask[offset] ← mask[offset]                      // birwise NOT

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

26: Method intersectWithMask()
27:   Method i ← limit downto 0 do
28:      offset ← index[i]
29:      w ← words[offset] & mask[offset]                 // bitwise AND
30:      if w ≠ words[offset] then
31:         words[offset] ← w
32:         if w = 0^64 then
33:            index[i] ← index[limit]
34:            index[limit] ← offset
35:            limit ← limit − 1

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

Algorithm 1: Pseudo code for Class SparseBitSet.

3.1.1 Fields

Todo: Add examples.

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

- `words` is an array of p 64-bit words which defines the current value of the bit-set: the ith bit of the jth word is 1 if and only if the $(j-1)\cdot 64 + i$th element of the set is present. Initially, all words in this array have all their bits set to 1, except the last word that may have a suffix of bits set to 0. Example.
- `index` is an array that manages the indices of the words in `words`, making it possible to performing operations to non-zero words only. In `index`, the indices of all the non-zero words are at positions less than or equal to the value of the field `limit`, and the indices of the zero-words are at indices strictly greater than `limit`.
- `limit` is the index of `index` corresponding to the last non-zero word in `words`. Thus it is one smaller than the number of non-zero words in `words`.
- `mask` is a local temporary array that is used to modify the bits in `words`.

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

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

3.1.2 Methods

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

- initSparseBitSet() in lines 6-11 initialises a sparse bit-set-object. It takes the number of bits as an argument and initialises the fields described in 3.1.1 in a straightforward way.
- isEmpty() in lines 12-13 checks if the number of non-zero words is different from zero. If the limit is set to -1, that means that all words are zero-words and the bit-set is empty.
- clearMask() in lines 14-21 reverses the bits in the temporary mask.
- reverseMask() in lines 18-17 clears the temporary mask. This means setting to 0 all words of `mask` corresponding to non-zero words of `words`.
- 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 unneccesary to save the index of a zero-word in a copy-based solver such as Gecode. We keep this line here though, because otherwise the invariant in (3.1) would not hold.

- intersectIndex() in lines 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 algortithm 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* $\underline{\text{dom}}(x)$ and the *current domain* $\text{dom}(x)$ 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:

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

- `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):

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

  `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. 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) : $\langle$*StatusMessage*, store$\rangle$
1: **if** the propagator is being posted **then** // executed in a constructor
2: $s \leftarrow$ InitialiseCT(s, T_0)
3: **if** $s = \emptyset$ **then**
4: **return** $\langle$**FAIL**, $\emptyset\rangle$
5: **else** // executed in an advisor
6: **foreach** variable $x \in s$ whose domain has changed since last time **do**
7: UpdateTable(s, x)
8: **if** `validTuples`.isEmpty() **then**
9: **return** $\langle$**FAIL**, $\emptyset\rangle$
10: **if** `validTuples` has changed since last time **then** // executed in the propagate function
11: $s \leftarrow$ FilterDomains(s)
12: **if** there is at most one unassigned variable left **then**
13: **return** $\langle$**SUBSUMED**, $s\rangle$
14: **else**
15: **return** $\langle$**FIX**, $s\rangle$

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, *Failed* is returned.

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

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 $t[x] \in \text{dom}(x)$ for all $x \in scp(c)$. We keep a counter, *nsupports*, for the number of valid supports for c. This is used for indexing the tuples in `supports` (we only index the tuples that are valid supports). If t is a valid support, all elements in `supports` corresponding to t are set to 1 in line 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 ∈ 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 $a \in \mathrm{dom}(x)$ 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 ∈ s such that |s(x)| > 1 do
 2:    foreach a ∈ s(x) do
 3:       index ← residues[x, a]
 4:       if validTuples[index] & supports[x, a][index] = 0 then
 5:          index ← validTuples.intersectIndex(supports[x, a])
 6:          if index ≠ −1 then
 7:             residues[x, a] ← index
 8:          else
 9:             s ← s[x ↦ s(x) \ {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) `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 (3.3). If such an index exists, we cache it

in `residues`$[x, a]$, and if it does not, we remove a from 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 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)
 1: validTuples.clearMask()
 2: if Δ_x is available ∧ |Δ_x| < |s(x)| then
 3:    foreach a ∈ Δ_x do
 4:       validTuples.addToMask(supports[x, a])
 5:    validTuples.reverseMask()
 6: else
 7:    foreach a ∈ s(x) do
 8:       validTuples.addToMask(supports[x, a])
 9: 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.

```
1: if validTuples has changed since last time then   // executed in the propagate function
2:    if (index ← validTuples.indexOfFixed()) ≠ −1 then
3:       return ⟨SUBSUMED, [x ↦ T[index][x]]_{x∈s}⟩
4:    else
5:       s ← 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 limit = 0 then
 3:    offset ← index[0]
 4:    w ← words[offset]
 5:    if (w & (w − 1)) = 0 then                          // Exactly one bit set
 6:       index_of_fixed = offset · 64 + MSB(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 (3.3), there must exists at least one tuple τ_i that is a support for (x, a) under the store $CT(s)$: $\exists i : i \in \texttt{validTuples} \ \wedge \ \tau_i[x] = a$. After UpdateTable() is perfomed 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 constratint 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

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

Describe

- Advisors
- Indexing of supports and residues
- Memory management
- Ideas from or-tools
- Profilation

5 Evaluation

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

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

The benchmarks consist of how many series with a total of 1507 CSP instances that where used in the experiments in [**?**]. The instances contain TABLE constraints only.

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 **??**. 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 is 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. The versions and their denotations are:

CT Basic version.

CT(Δ) CT using Δ_x, the set of values that has been removed from dom(x) since last time, 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 **??**.

5.1.3 Discussion

CT(Δ) outperforms the other variants.

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 nonincremental 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 [**?**]. The propagator state has the following fields:

- 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.

PROCEDURE EXTENSIONAL() : *bool*

1: **foreach** $x \in X$ **do**
2: $S[x] \leftarrow \perp$
3: **foreach** $x \in X$ **do**
4: $N \leftarrow \emptyset$
5: **foreach** $n \in D(x)$ **do**
6: **if** $S[x] = \perp$ **then**
7: $\ell \leftarrow L[\langle x, n\rangle]$
8: **while** $\ell \neq \perp \wedge \exists y \in X : \ell[y] \notin D(y)$
9: $\ell \leftarrow L[\langle x, n\rangle] \leftarrow$ next tuple for $\langle x, n\rangle$
10: **if** $\ell \neq \perp$ **then**
11: $N \leftarrow N \cup \{n\}$
12: **foreach** $y \in X$ where $y > x$ **do**
13: $S[y] \leftarrow \ell[y]$
14: **if** $N = \emptyset$ **then**
15: **return false**
16: **else**
17: $D(x) \leftarrow N$
18: **return true**

Algorithm 9: Basic positive tuple set propagator.

name	number of instances	arity	table size	variable domains
A5	50	5	12442	0..11, a few 0..1
A10	50	5	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	X	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		0..4
MDD 07	9	7		0..4
MDD 09	10	7		0..4
Mod Renault	50	2 – 10		0..41 or smaller
Nonograms	180	2		1..15 to 1..980
Pigeons Plus	40	2, some higher		0..9 or smaller
Rands JC2500	10	7		0..7
Rands JC5000	10	7		0..7
Rands JC7500	10	7		0..7
Rands JC10000	10	7		0..7
TSP 25	15	2, a few 3		Vary from singleton to 0..1000
TSP Quat 20	15	2, a few 3		Vary from singleton to 0..1000

Table 1: Benchmarks series and their characteristics.

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 therefore whose domain needs updating and whose support data need clearing. Initially empty.

```
PROCEDURE Extensional() : bool
 1: if the propagator is being posted then                    // executed in a constructor
 2:   foreach x ∈ X do
 3:     foreach n ∈ D(x) do
 4:       FindSupport(⟨x, n⟩)
 5: else                                                      // executed in an advisor
 6:   foreach ⟨x, n⟩ that has been removed since last time do
 7:     foreach t ∈ S[⟨x, n⟩] do
 8:       RemoveSupport(t, ⟨x, n⟩)
 9: while W_R ≠ ∅ ∨ W_S ≠ ∅                                   // executed in the propagator proper
10:   foreach ⟨x, n⟩ ∈ W_R do
11:     D(x) ← D(x) \ {n}
12:     if D(x) was wiped out then
13:       return false
14:   W_R ← ∅
15:   foreach ⟨x, n⟩ ∈ W_S where n ∈ D(x) do
16:     FindSupport(⟨x, n⟩)
17:   W_S ← ∅
18: return true
```

Algorithm 10: Incremental positive tuple set propagator.

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

Algorithm 11: Recheck support for literal $\langle x, n\rangle$.

PROCEDURE RemoveSupport($\ell, \langle x, n\rangle$)

1: **foreach** $y \in X$ **do**
2: $\quad S[\langle y, \ell[y]\rangle] \leftarrow S[\langle y, \ell[y]\rangle] \setminus \{\ell\}$
3: $\quad$ **if** $y \neq x \wedge S[\langle y, \ell[y]\rangle] = \emptyset$ **then**
4: $\quad\quad W_S \leftarrow W_S \cup \{\langle y, \ell[y]\rangle\}$

Algorithm 12: Clear support data 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 **??**, CT(Δ), was compared against the two existing propagators in Gecode for the TABLE constraint, as well with the propagator for the REGULAR constraint. 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 **??**.

5.2.3 Discussion

Overall performance. Comparing CT against B and I over all series, CT performs either as good as, or better than both B and I. B or I does not outperform CT on any of the series. On some of the series CT is about a factor 10 faster.

Comparing DFA against the other algorithms, on some series DFA outperforms all the other algorithms, and on some series the other algorithms outperform DFA.

Looking at the shape of the curves for the different algorithms, we see that DFA behaves differently than the other algorithms; the shape of the curves of CT, B, and I are very similar to each other for all series, while the shape of the curves of DFA is different from the other algorithms in most cases.

The impact of arity on performance. There seems to be a weak correlation between arity of the constraints and the performance. CT seems to perform better on the benchmark series that contain constraints with higher arities, than on the series that contain constraints with low aritities. However, there seems to be other factors beside the arity that affect the performance.

The impact of table size on performance.

The impact of domain size on performance. There seems to be a weak correlation between domain size and performance. CT seems to perform best on series where the domain sizes are medium sized or large. On the series that contain instances with small domain sizes ($2-3$), CT generally does not perform better than B and I. One exception to this is **BDD Large**, with domain size 2, where CT outperforms the other algorithms.

Runtime vs. Solvetime. 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 initilising CT takes longer time than initialising B and I. The reason could be that CT performs more initial propagation than B and I. Check if CT performs more initial propagation.

6 Conclusions and Future Work

For the implementation to reach industry standard 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 neccassary 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 [**?**].

References

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

[2] C. Bessière, J. Régin, R. H. C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artif. Intell.*, 165(2):165–185, 2005.

[3] 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 2016*, volume 9892 of *Lecture Notes in Computer Science*, pages 207–223. Springer, 2016.

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

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

[6] G. Pesant. A regular language membership constraint for finite sequences of variables. In M. Wallace, editor, *CP 2004*, volume 3258 of *LNCS*, pages 482–495. Springer, 2004.

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

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

[9] H. Verhaeghe, C. Lecoutre, and P. Schaus. Extending compact-table to negative and short tables. In S. P. Singh and S. Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 3951–3957. AAAI Press, 2017.

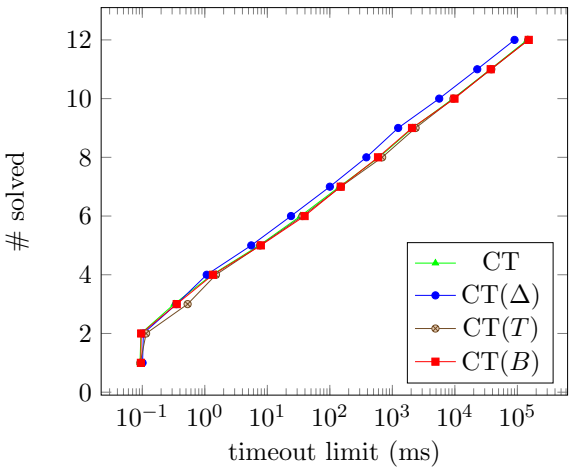

Figure 4: **Langford 4**.

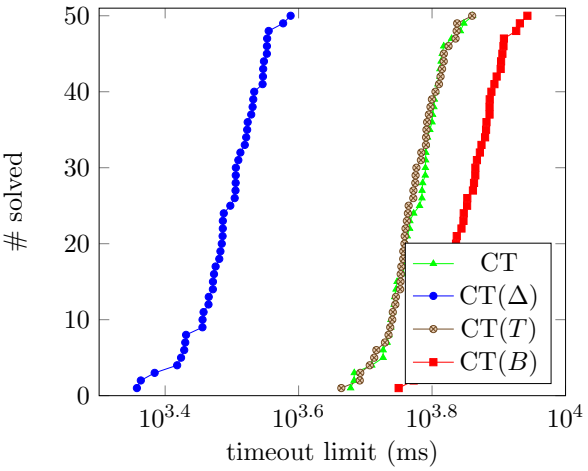

Figure 5: **A5**.

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.

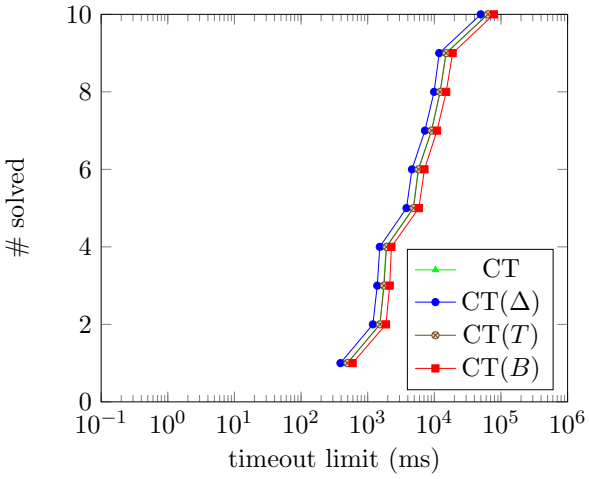

Figure 2: **Rands JC2500.**

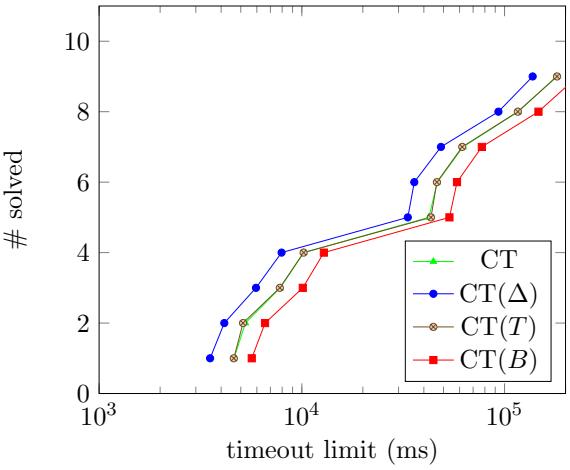

Figure 3: **Rands JC5000**.

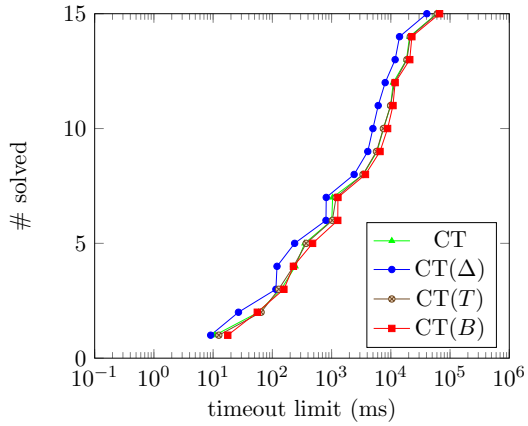

Figure 6: **TSP Quat 20**.

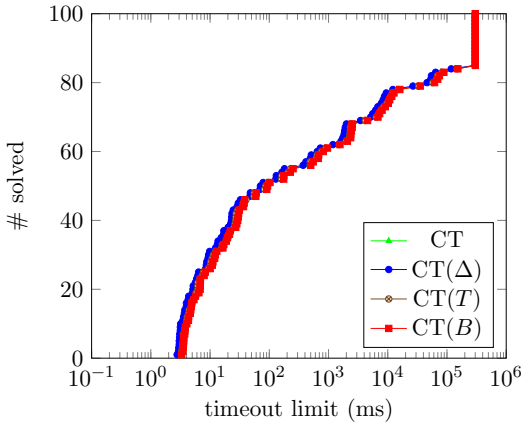

Figure 7: **Geom**.

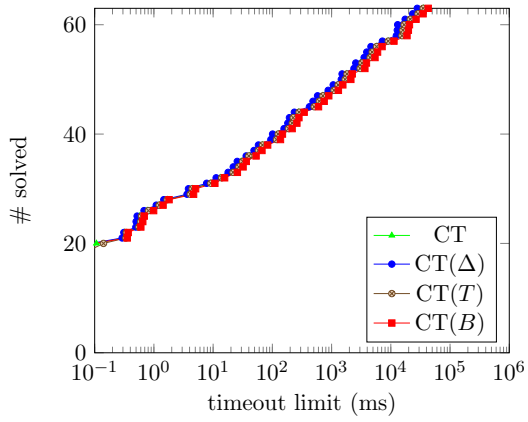

Figure 8: **Crosswords LexVG**.

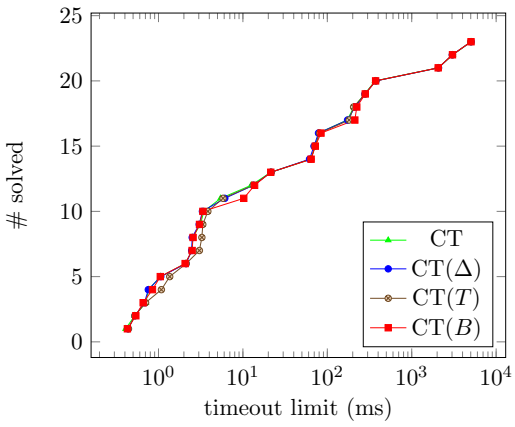

Figure 9: **AIM 50**.

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.

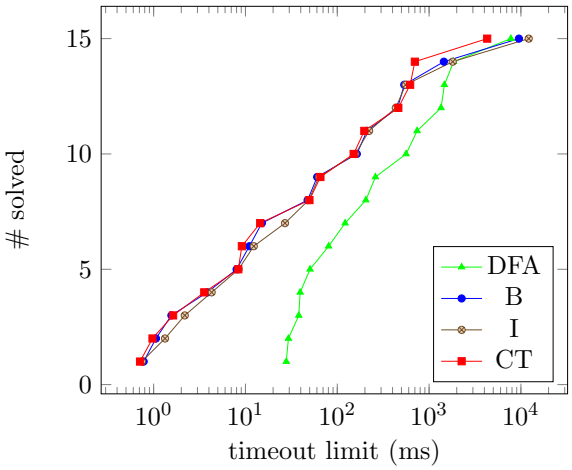

Figure 10: **Langford 2**.

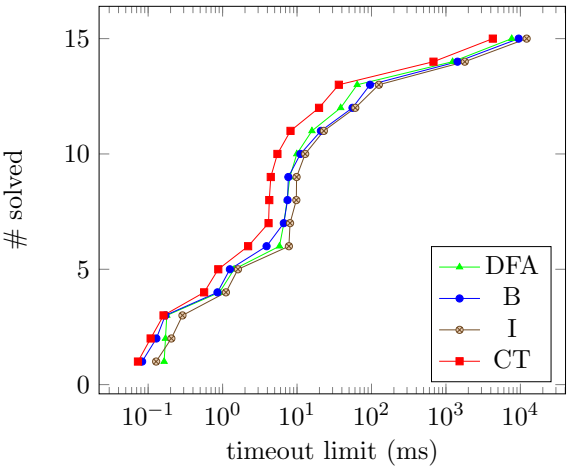

Figure 11: **Langford 2**.

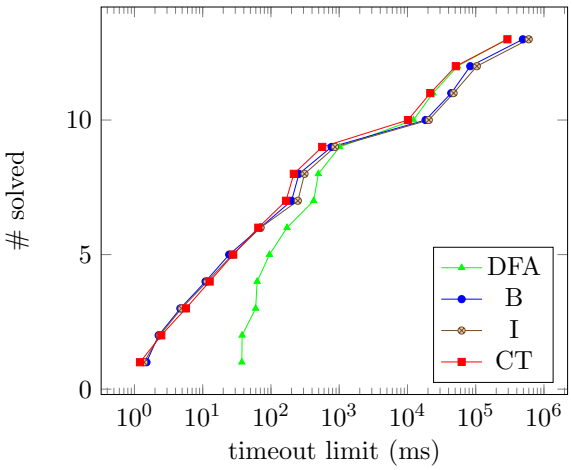

Figure 12: **Langford 3**.

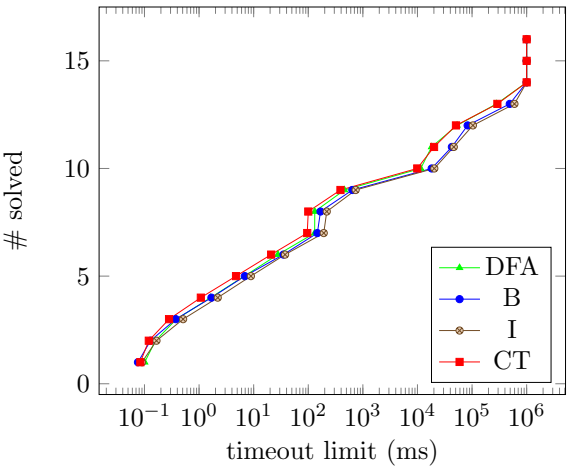

Figure 13: **Langford 3**.

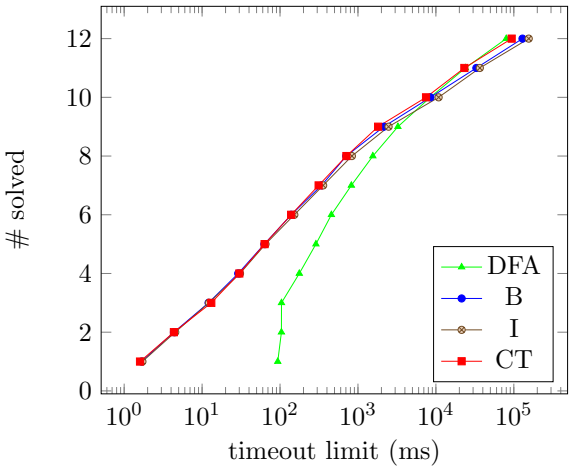

Figure 14: **Langford 4**.

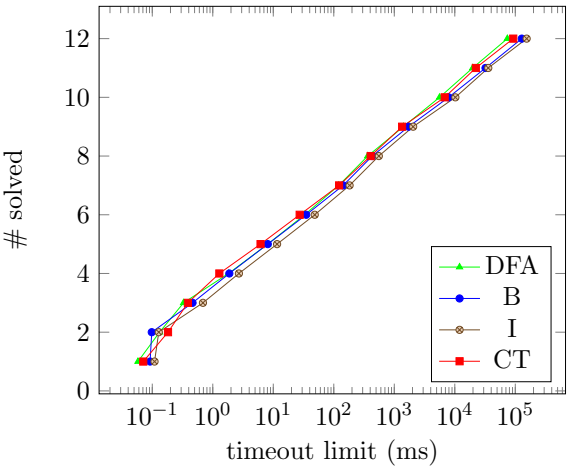

Figure 15: **Langford 4**.

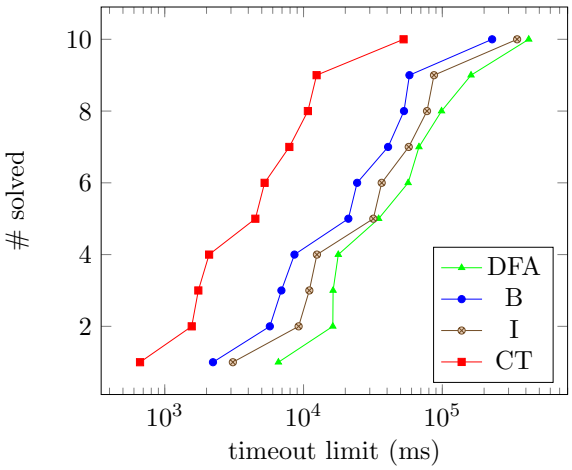

Figure 16: **Rands JC2500.**

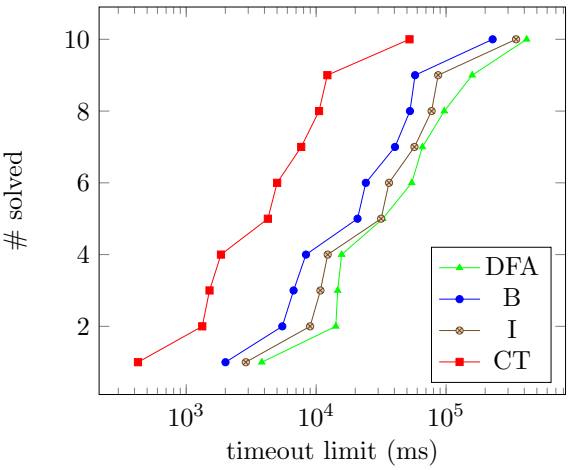

Figure 17: **Rands JC2500.**

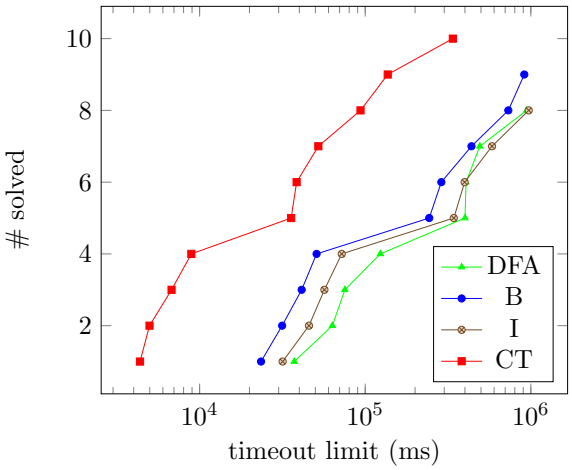

Figure 18: **Rands JC5000.**

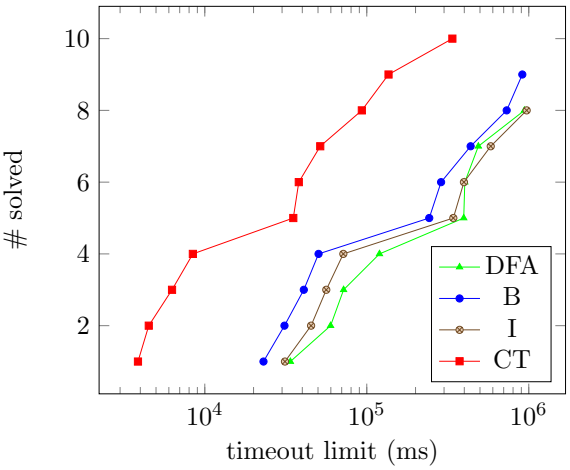

Figure 19: **Rands JC5000.**

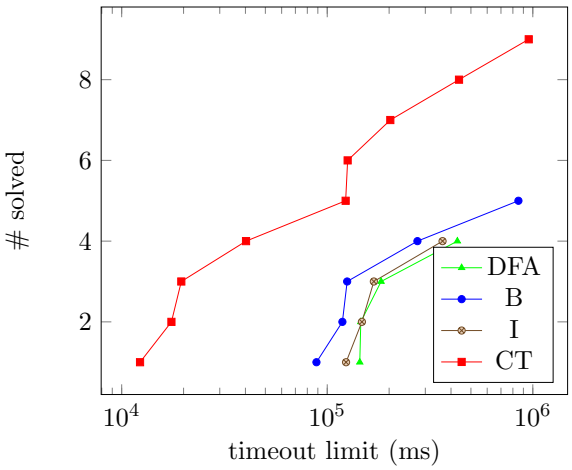

Figure 20: **Rands JC7500**.

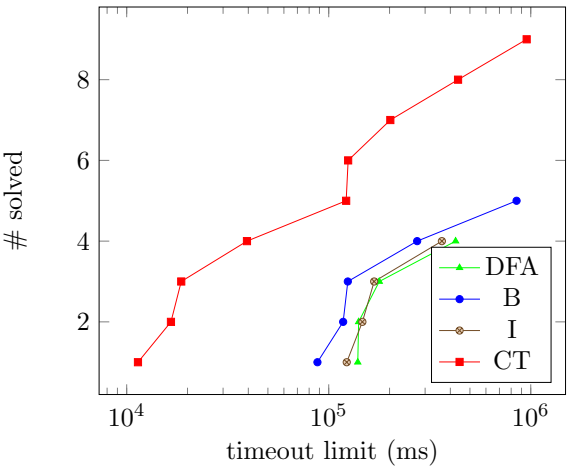

Figure 21: **Rands JC7500**.

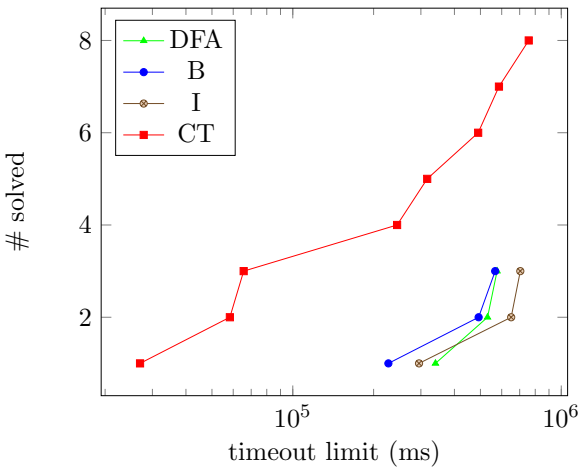

Figure 22: **Rands JC10000**.

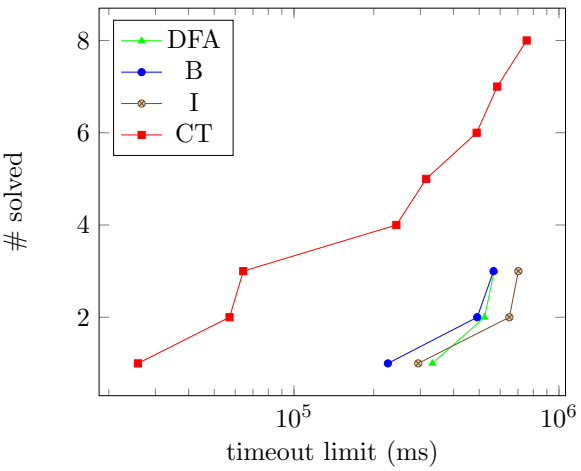

Figure 23: **Rands JC10000**.

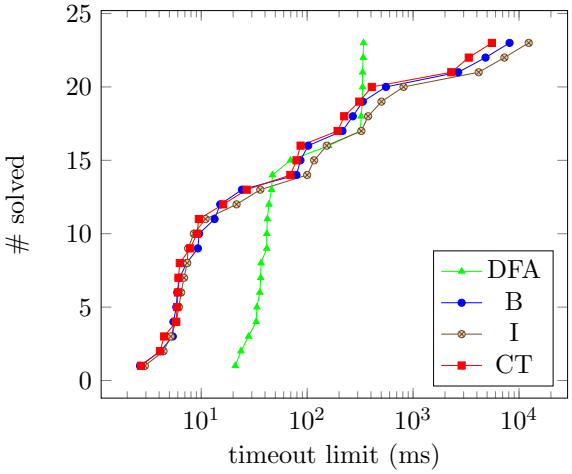

Figure 24: **AIM-50**.

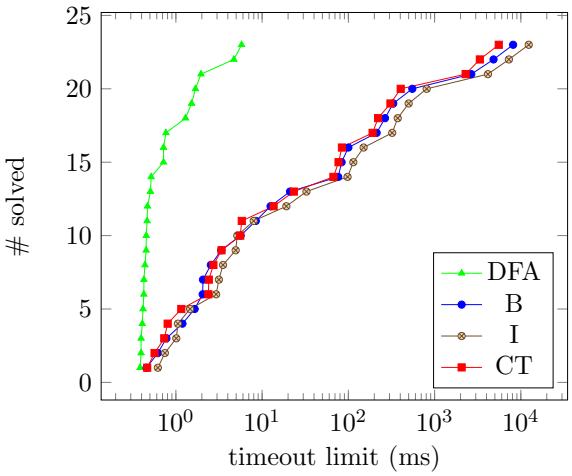

Figure 25: **AIM-50**.

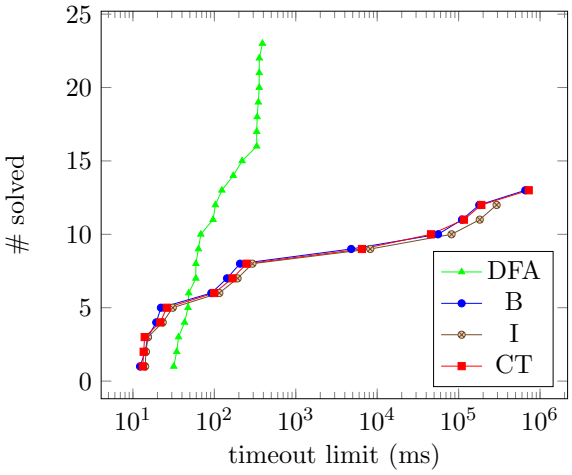

Figure 26: **AIM-100**.

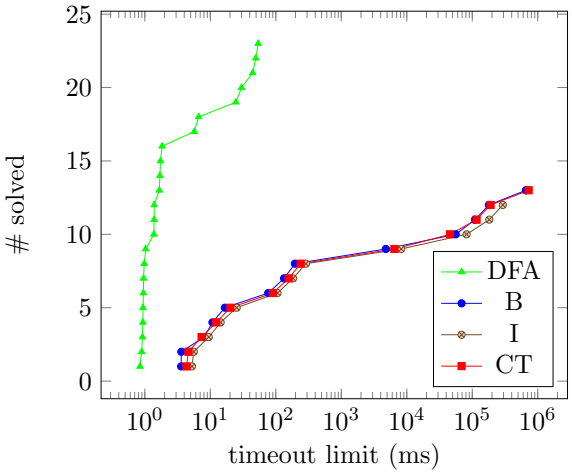

Figure 27: **AIM-100**.

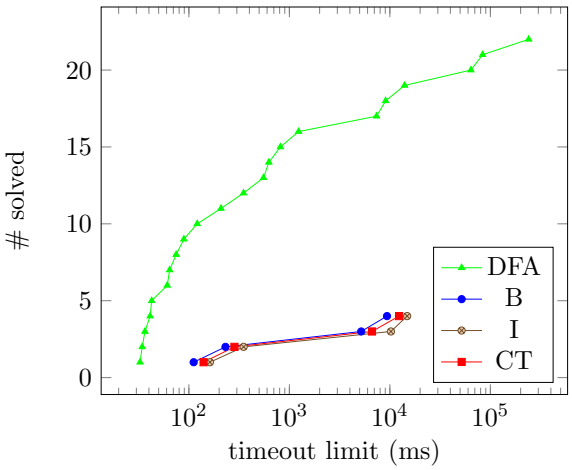

Figure 28: **AIM-200**.

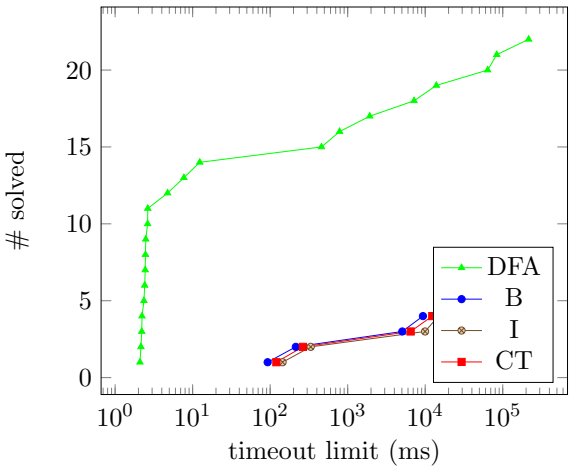

Figure 29: **AIM-200**.

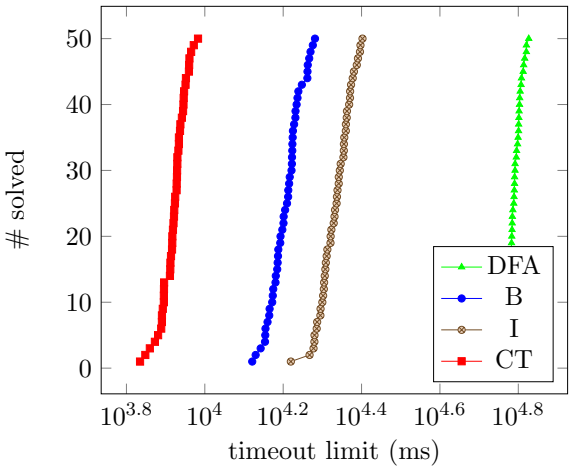

Figure 30: **A5**.

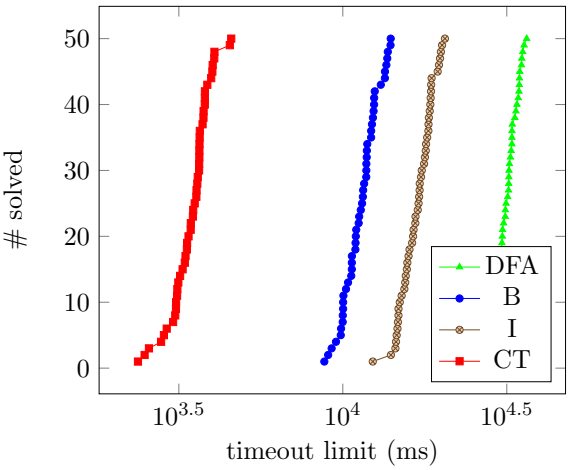

Figure 31: **A5**.

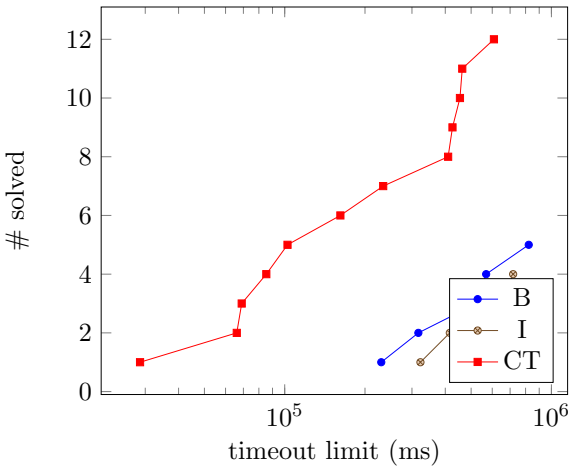

Figure 32: **A10**.

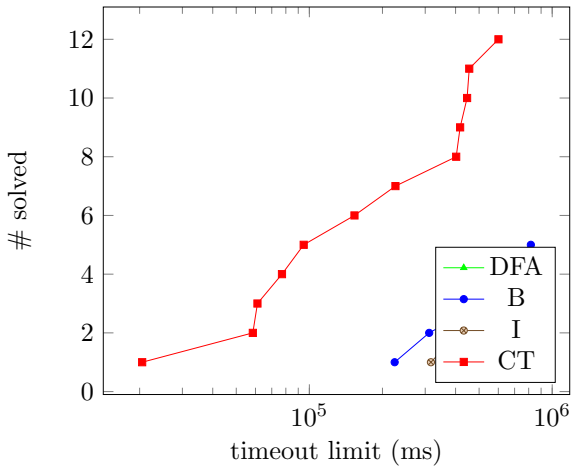

Figure 33: **A10**.

Ωb

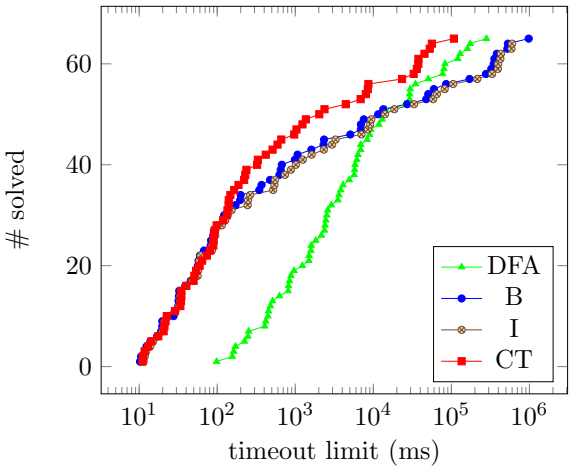

Figure 34: **Crosswords WorldVG**.

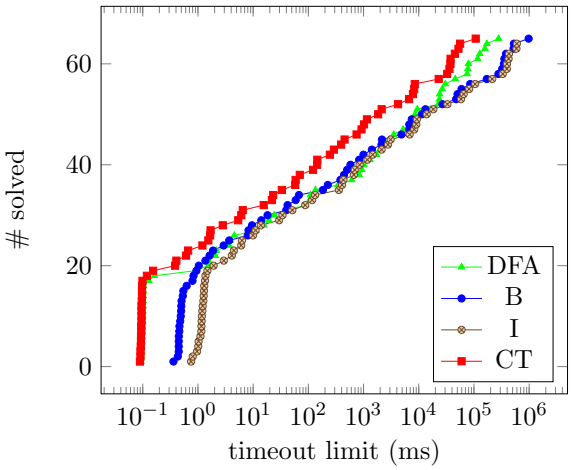

Figure 35: **Crosswords WorldVG**.

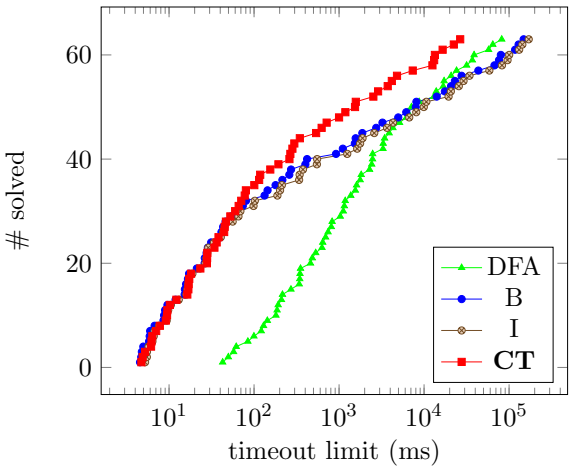

Figure 36: **Crosswords LexVG**.

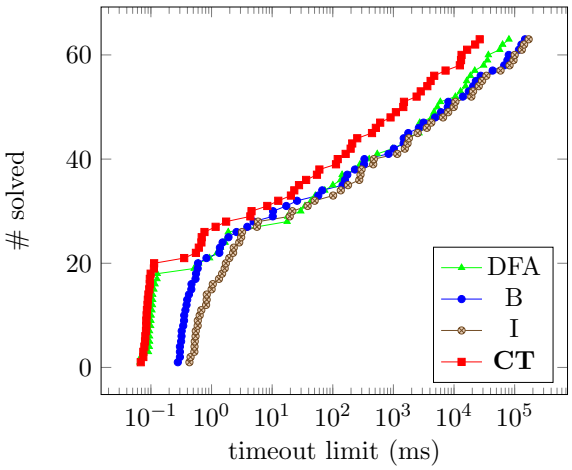

Figure 37: **Crosswords LexVG**.

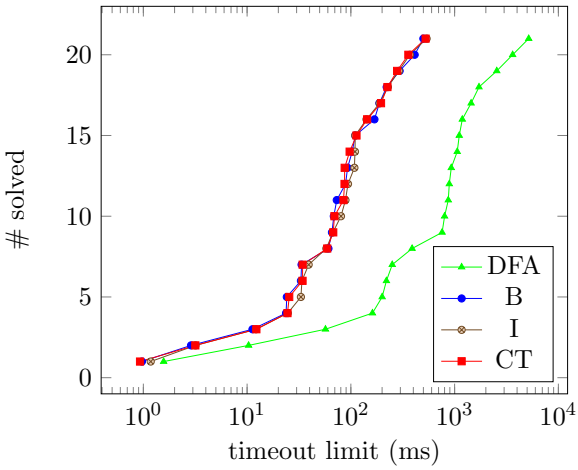

Figure 38: **Crosswords Wordspuzzle**.

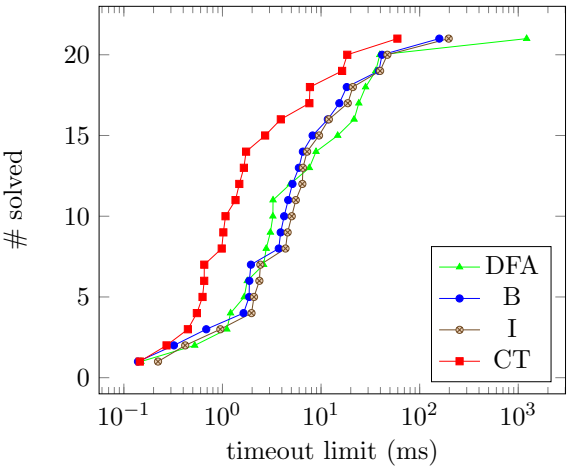

Figure 39: **Crosswords Wordspuzzle**.

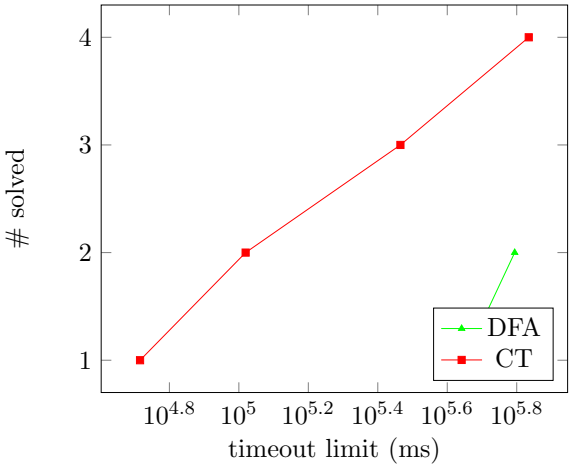

Figure 40: **MDD 05**.

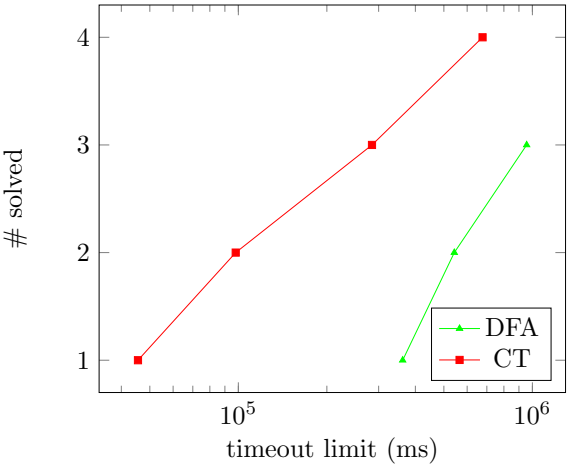

Figure 41: **MDD 05**.

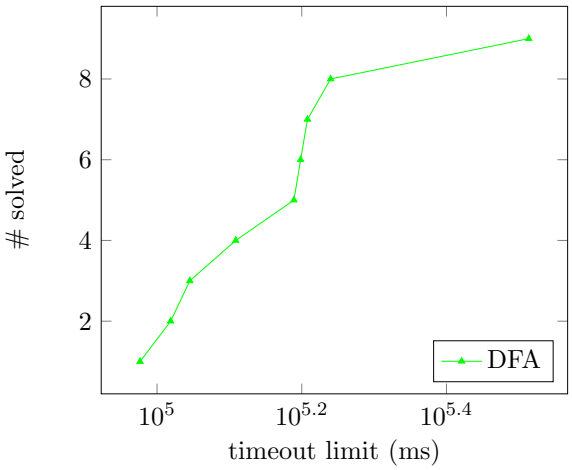

Figure 42: **MDD 07**.

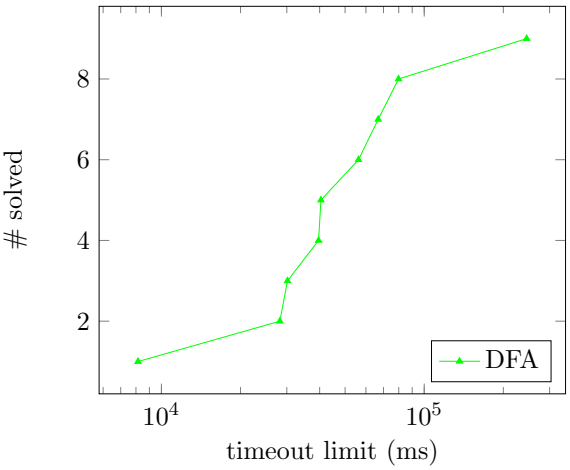

Figure 43: **MDD 07**.

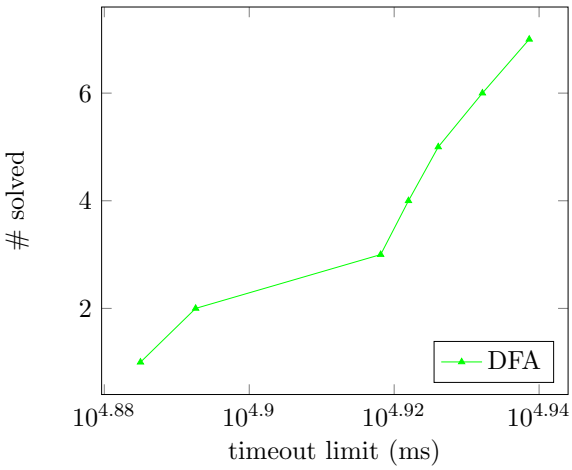

Figure 44: **MDD 09**.

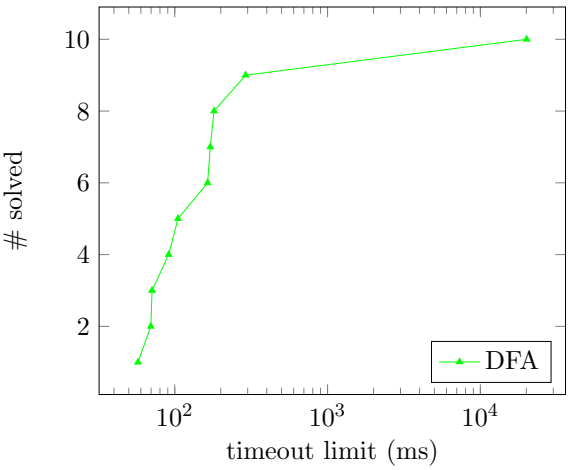

Figure 45: **MDD 09**.

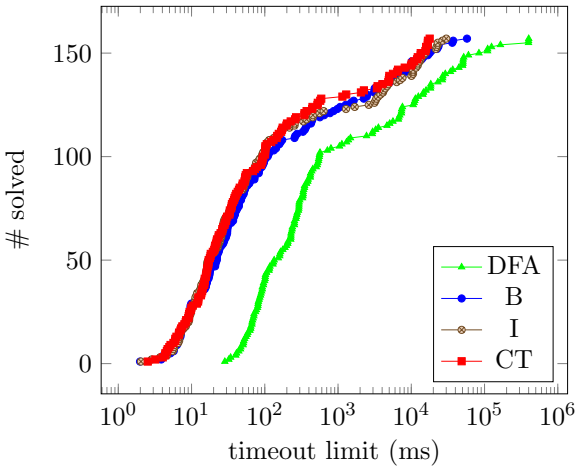

Figure 46: **Kakuro easy**.

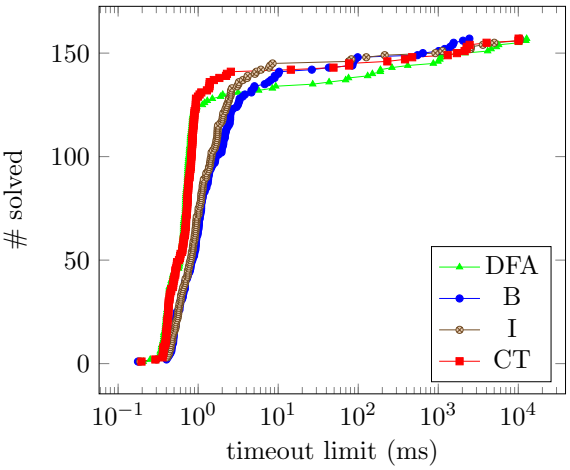

Figure 47: **Kakuro easy**.

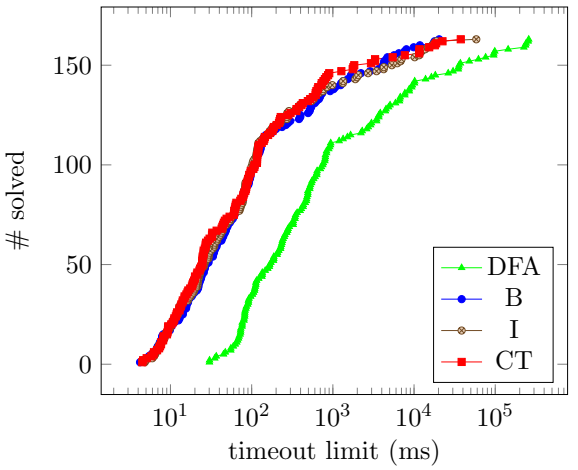

Figure 48: **Kakuro Medium**.

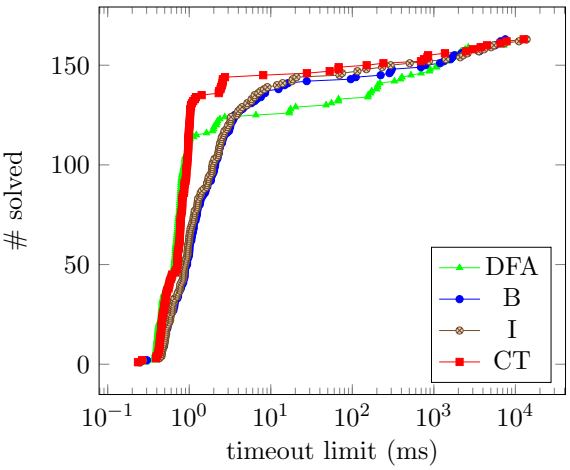

Figure 49: **Kakuro Medium**.

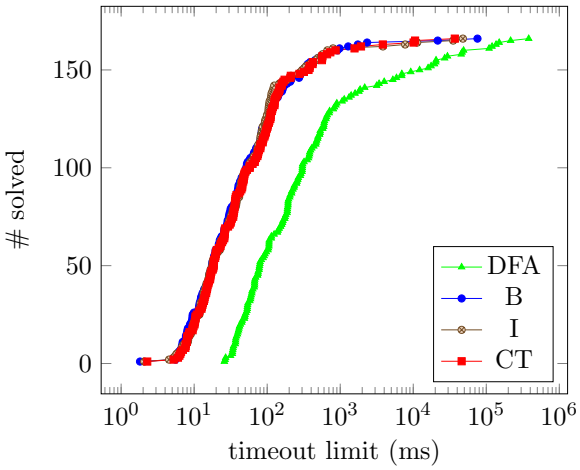

Figure 50: **Kakuro Hard**.

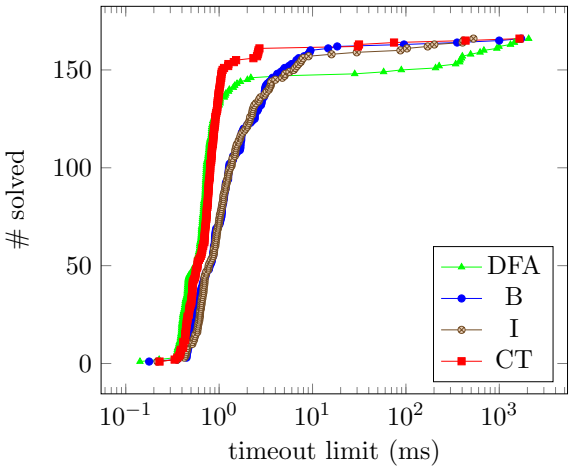

Figure 51: **Kakuro Hard**.

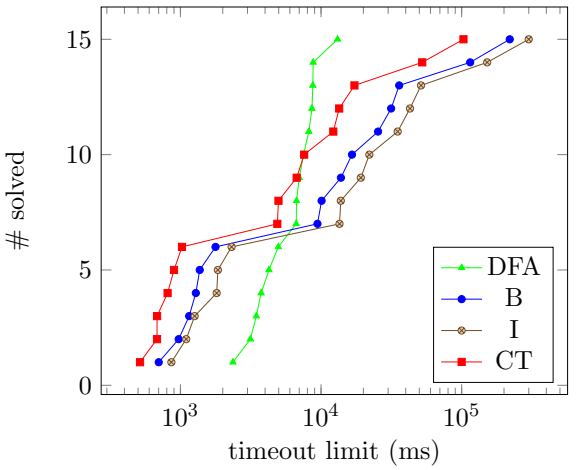

Figure 52: **TSP 25**.

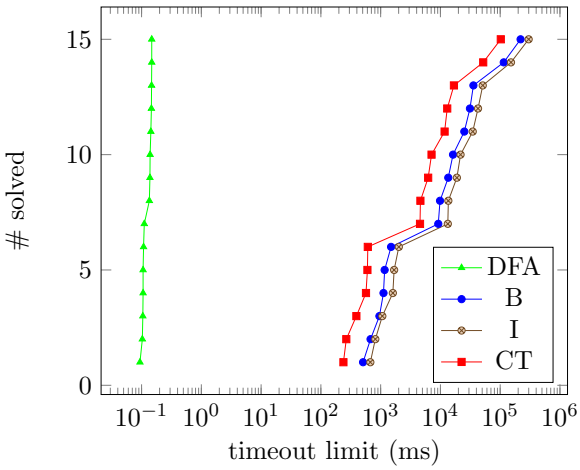

Figure 53: **TSP 25**.

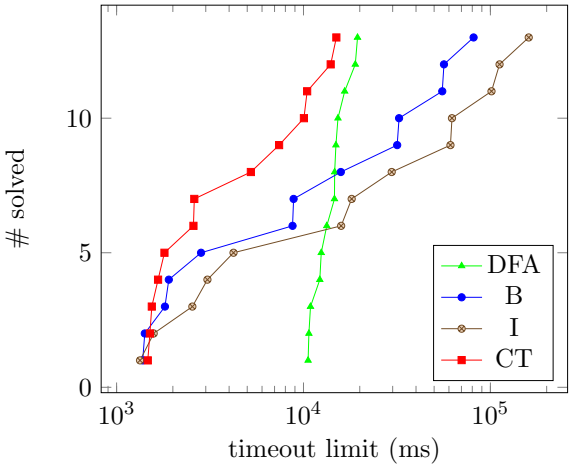

Figure 54: **TSP Quat 20**.

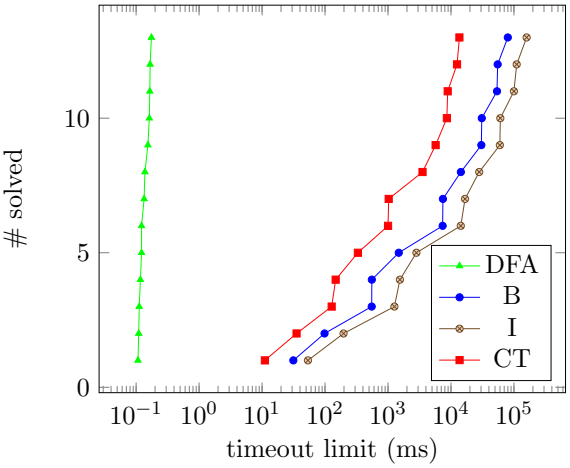

Figure 55: **TSP Quat 20**.

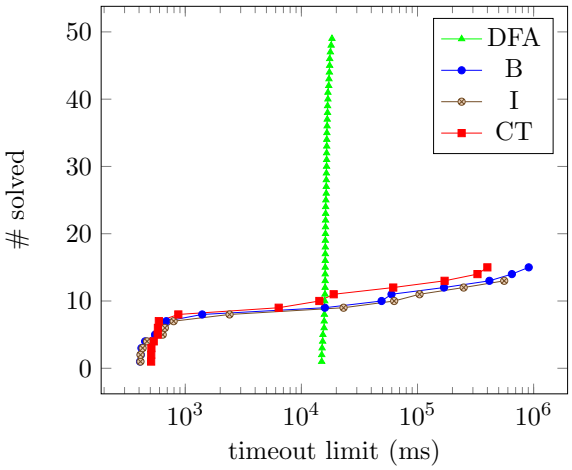

Figure 56: **Mod Renault**.

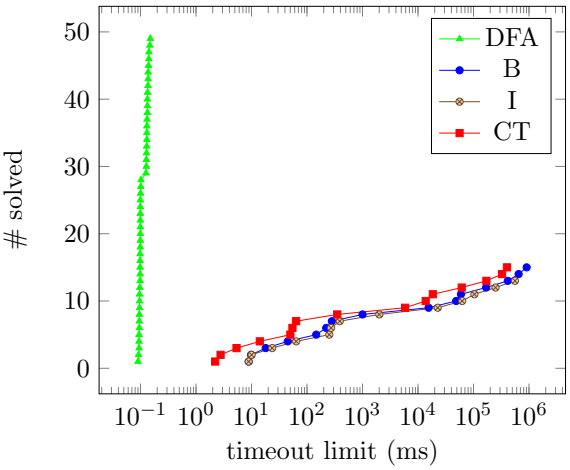

Figure 57: **Mod Renault**.

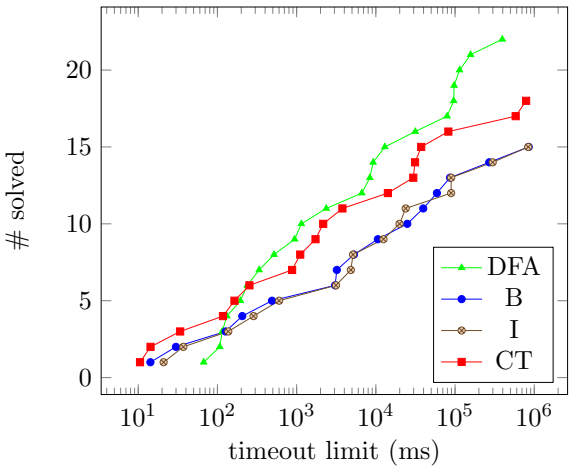

Figure 58: **Pigeons Plus**.

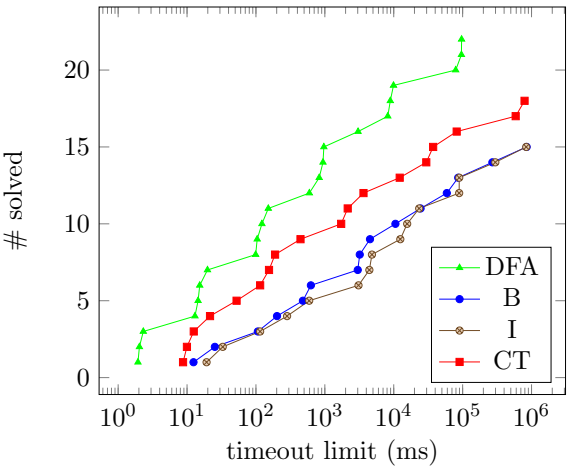

Figure 59: **Pigeons Plus**.

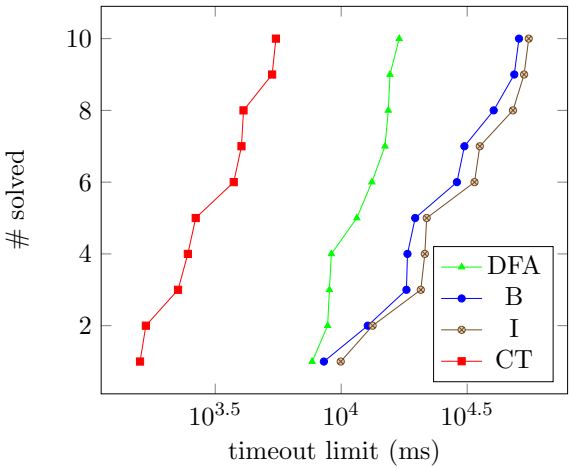

Figure 60: **K5**.

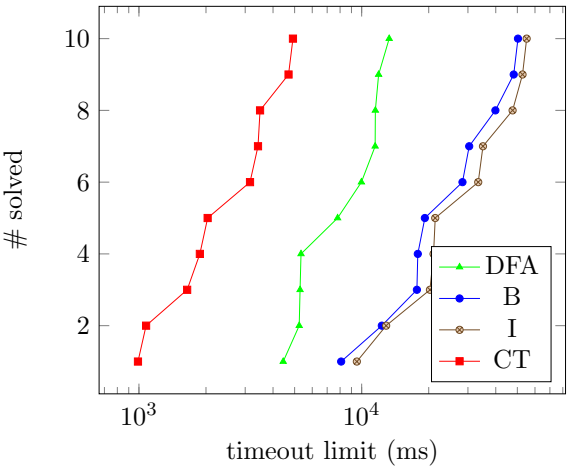

Figure 61: **K5**.

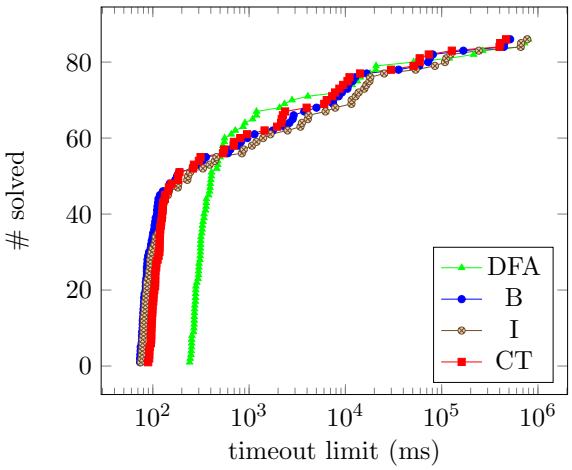

Figure 62: **Geom**.

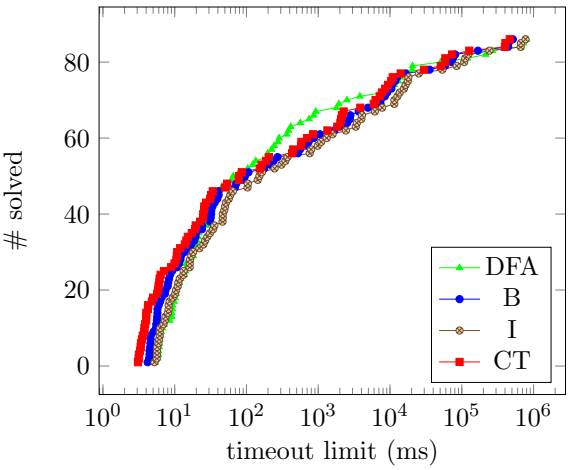

Figure 63: **Geom**.

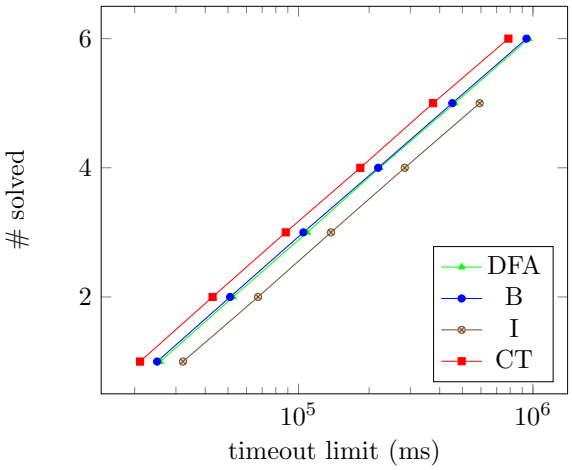

Figure 64: **Dubois**.

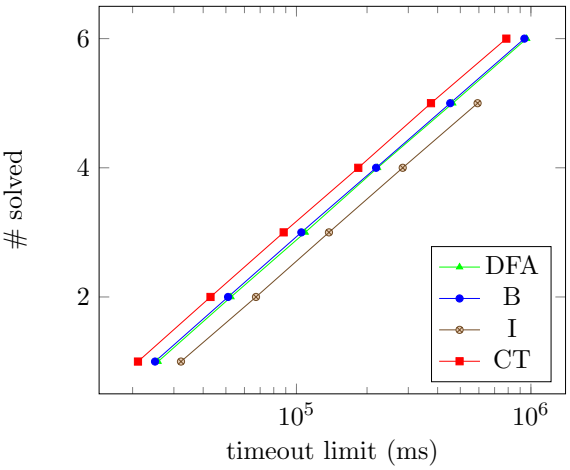

Figure 65: **Dubois**.

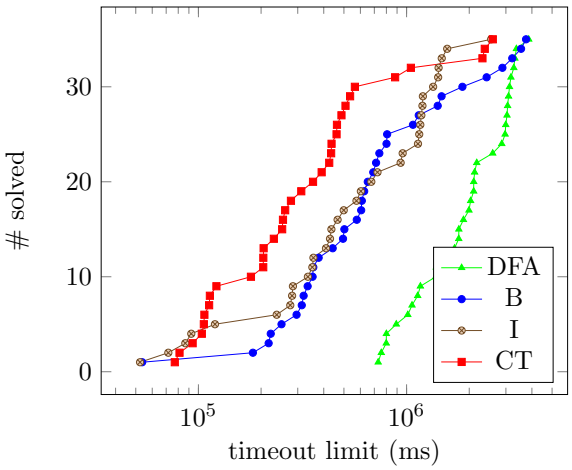

Figure 66: **BDD Large**.

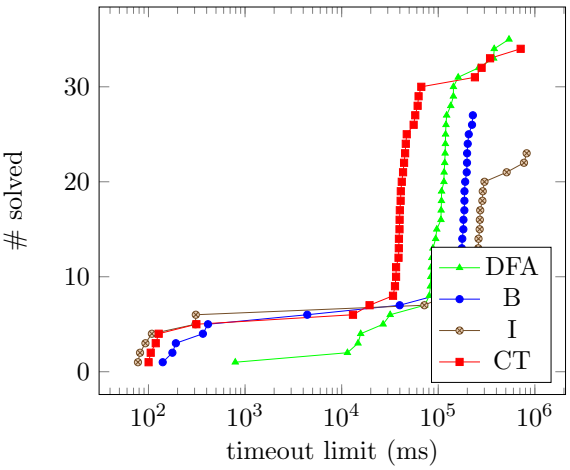

Figure 67: **BDD Large**.

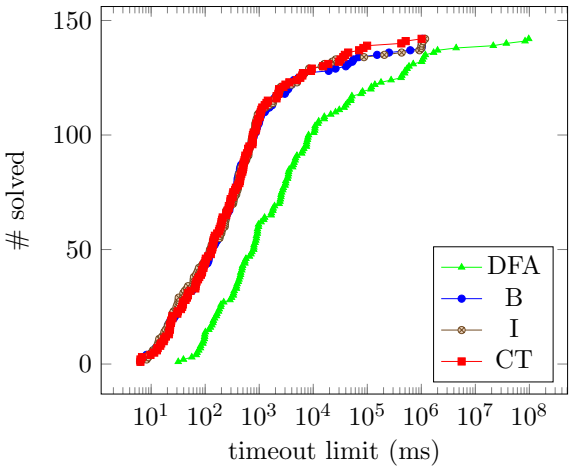

Figure 68: **Nonograms**.

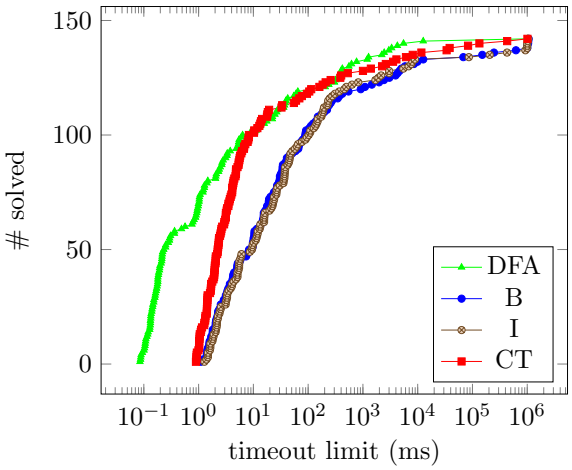

Figure 69: **Nonograms**.