Implementation and Evaluation of a Compact-Table Propagator in Gecode

Linnea Ingmar

9th June 2017

Contents

1 Introduction

Constraint programming (CP) [1] 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 form what is called the *domain* of the variable. A *solution* to a constraint problem consists of a complete assignment of values to variables, 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 on the variables.

A solution to a constraint problem is found by generating a search tree, branching on partitions of the 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 domains that are in conflict with the constraint.

The Table constraint expresses the possible combinations of values that the associated variables can take as a set 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 [5], called Compact-Table (CT). The results published in the named paper indicate that CT outperforms all previously known algorithms in terms of runtime.

A constraint programming solver (CP solver) is a software that solves constraint problems. *Gecode* [6] is a popular CP solver written in the C++ programming language that combines state-of-the-art performance with modularity and extensibility. Presently, Gecode has two existing propagators for Table, but to the best of my knowledge there have been no attempts to implement CT in Gecode before this project, and thus 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 work 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 items are the contributions made by this dissertation, while simultaneously serving as a description of the outline:

- 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 [5] have been modified to suit the target CP solver Gecode, and are presented and explained in Section 3.
- Several versions of the CT algorithm have 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 of Gecode, which suggests that CT should be included in Gecode; this is discussed in Section 6.

- Several possible improvements and known 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 2.2 discusses the concepts propagation and propagators in detail. Section 2.3 gives an overview of Gecode, a constraint programming solver. Section 2.4 introduces the Table constraint. Section 2.5 describes the main concepts of the Compact-Table (CT) propagation 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

Constraint programming (CP) [1] 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 form what is called the *domain* of the variable. A *solution* to a constraint problem consists of a complete assignment of values to variables, 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 on the variables.

A constraint programming solver (CP solver) is a software that takes constraint problems expressed in some modelling language as input, tries to solve them, and outputs the results to the user of the software. The process of solving a problem consists of generating a search tree by branching on partitions of the possible values for the variables. At each node in the search tree, the solver removes impossible values from the domains of variables. This filtering process is called *propagation.* Each constraint is associated with at least one propagation algorithm, whose purpose is to detect and remove values from the domains of the variables that cannot 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 partitioning its domain into a number of subsets, creating as many branches as subsets. Each subset is associated with one branch, along which the domain of the variable is restricted to that subset. When search moves to a new node in the tree 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

[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.

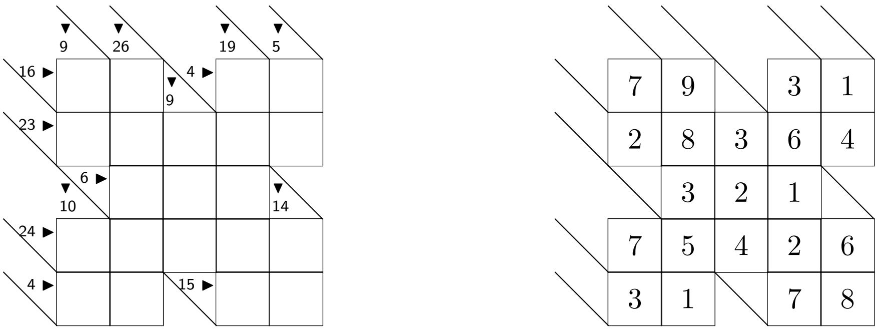

Figure 1: A Kakuro puzzle [2](left) and its solution (right).

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. If all leaves of the tree consist of failed nodes, then the problem is unsatisfiable, else there is a solution that will be found if search is allowed to go on long enough.

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 white cells (some boards also have black cells framing the white 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 white 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 are that the sum of the variables that belong to a given entry must be equal to the clue for that entry, and that 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 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 follow some formal definitions, based on [1, 10, 11].

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

Definition 1. ***Constraint.*** *Consider a finite sequence of* n *variables* $V = v_1, \ldots, v_n$*, and a corresponding sequence of finite* domains $D = D_1, \ldots, D_n$ *ranging over integers, which are possible values for the respective variable. For a variable* $v_i \in V$*, its domain* D_i *is denoted by* $dom(v_i)$*, its* domain size *is* $|dom(v_i)|$ *and its* domain width *is* $(\max(dom(v_i)) - \min(dom(v_i)) + 1)$.

- *A* constraint c *on* V *is a relation, denoted by* $rel(c)$*. The associated variables* V *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* V*, and we call those* n*-tuples* solutions *to the constraint* c*.*

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

- *For an n-tuple τ associated with V, we denote the ith value of τ by $\tau[i]$ or $\tau[v_i]$. The tuple τ is* valid *for V if and only if each value of τ is in the domain of the corresponding variable: $\forall i \in 1 \dots n, \tau[i] \in dom(v_i)$, or equivalently, $\tau \in D_1 \times \dots \times D_n$.*

- *For a constraint c on V, the n-tuple τ is a* support *on c if and only if τ is valid for V and τ is a solution to c, that is, τ is a member of $rel(c)$.*

- *For an n-ary constraint c, involving a variable x such that the value $a \in dom(x)$, the n-tuple τ is a* support for (x, a) *on c if and only if τ is a support on c and $\tau[x] = a$. If such a tuple τ exists, (x, a) is said to have a support on c.*

Note that Definition 1 restricts domains to 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: $V = v_1, \dots, v_n$ is a finite sequence of variables, $D = D_1, \dots, D_n$ is a finite sequence of domains for the respective variables, and $C = \{c_1, \dots, c_m\}$ 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* store *s is a function, mapping a finite set of variables $V = v_1, \dots, v_n$ to a finite set of domains. We denote the domain of a variable v_i under s by $s(v_i)$.*

- *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 m-ary constraint on a subsequence of V, where $m \leq n$. A store s is a* solution store *to c if and only if all variables in $vars(c)$ are assigned and the corresponding m-tuple is a solution to c: $\forall i \in \{1, \dots, m\}, s(v_i) = \{a_i\}$, and $\langle a_1, \dots, a_m \rangle$ 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 $v_i \in V$.*

- *A store s_1 is* strictly stronger *than a store s_2, written $s_1 \prec s_2$, if and only if s_1 is stronger than s_2 and $s_1(v_i) \subset s_2(v_i)$ for some $v_i \in V$.*

2.2 Propagation and Propagators

Constraint propagation is the process of removing values from the domains of the variables in a CSP that cannot participate in a solution store to the problem, because assigning them to the variables would violate the constraint. 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 the 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 [10] and [11], where we let *store* be the set of all stores.

Definition 4. ***Propagators.*** *A* propagator *p is a function mapping stores to stores:*

$$p : \mathit{store} \rightarrow \mathit{store}$$

In a CP solver, a propagator is implemented as a function that also returns a status message*. The possible status messages are* Fail*,* Subsumed*,* Fixpoint*, and* Possibly not at fixpoint*. A propagator p is at* fixpoint *on a store s if and only if applying p to s gives no further propagation: $p(s) = s$. If a propagator p always returns a fixpoint, that is, if $p(s) = p(p(s))$ for all stores s, then 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 fulfil 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: it follows the intuition that more input information (stronger input store) should give a stronger conclusion (stronger output store).*

- *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 solution stores.*

- *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, then 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 *when it does not return a fixpoint, and it is* subsumption honest *if and only if it does not signal* Subsumed *when it is not subsumed by the input store.*

This definition is not as strong as it might seem; 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 fulfils 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 Possibly not at fixpoint, except for assignment stores where it must signal either Fail or Subsumed as required by the honest property.

So why not stay on the safe side and always signal Possibly not at 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 domain of at least one of the variables changes. If p is subsumed by a store s, then 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* gives a measure of how strong the propagation of a propagator is. There are three commonly used consistencies: **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 bound 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: $\forall x \in vars(c), \forall a \in dom(x)$, (x, a) has a support on c.*

Todo: Value consistency.

A propagator p is said to have a certain consistency if after applying p to any input store s, the resulting store $p(s)$ always has that consistency. Enforcing domain consistency might remove more values from the domains of the variables compared to when enforcing value- or bounds consistency, but might be more costly.

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

2.3 Gecode

Gecode [6] (Generic Constraint Development Environment) is a popular CP 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 a CP solver, including specifically the implementation of new propagators. Furthermore, Gecode is well documented and comes with a complete tutorial [11].

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

One such data structure is called *advisors*, which can inform propagators about variable modifications. The purpose of an advisor is, as its name suggests, to advise the propagator of whether it needs to be executed or not. Whenever the domain 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 does not need to scan all the variables to see which ones have modified domains since its last invocation. Propagators that use data structures to avoid scanning all variables and/or all domains of the variables in each execution are 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. 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 usually undesirable.

2.4 The Table Constraint

The Table constraint, also called Extensional, explicitly expresses the possible combinations of values for the variables as a set of tuples:

Definition 7. ***Table constraints.*** *A (positive*[3]*)* table constraint c *is a constraint such that* $rel(c)$ *is defined explicitly by listing all the tuples that are solutions to c.*

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, because the number of possible combinations of values might be exponential 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 [10].

As an example of use case, the TABLEconstraint has proved to be useful for pre-solving sub-problems in constraint models [4].

In Gecode, the TABLE constraint and another constraint called REGULAR, which constraints a sequence of variables to form a word of a regular language, are both called EXTENSIONAL. Gecode provides one propagator for REGULAR, based on [8], and two propagators for TABLE; one which is based on [2], being more memory efficient than the other, and one that is more incremental and more efficient in terms of execution time.

2.5 The Compact-Table Algorithm

The compact-table (CT) algorithm is a domain-consistent propagation algorithm that implements the TABLE constraint. It was first implemented in OR-tools (Google Optimization Tools), a CP solver, where it outperforms all previously known algorithms, and was first described in [5]. Before this project, no attempts to implement CT in Gecode were made to the best of my knowledge, and consequently how it would perform in that framework was an open question.

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 which tuples from the input table that are currently valid. Also, for each variable-value pair, a bit-set mask is computed and stored in an array `supports`: each bit-set mask stores which of the tuples that are supports for the corresponding variable-value pair.

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, all values that no longer have a support.

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

2.6 Reversible Sparse Bit-Sets

Reversible sparse bit-sets [5] is a sparse-set [3, 9] 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), even when the bit-set is sparse; hence the name.

From hereon, we call the datastructure representing integers *int*. 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`.

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

Some CP solvers use a mechanism 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. This is true for the CP solver OR-tools, in which reversible sparse bit-sets are reversed to a previous state upon backtrack. In the following sections, a data structure that is like a reversible sparse bit-sets except that it is not reversible is called a sparse bit-set.

3 Algorithms

This section presents the algorithms that are used in the implementation of the CT propagator in Section 4. In the following, for an array a, we let $a[0]$ denote the first element (thus indexing starts from 0), a.length() the number of cells, and $a[i : j]$ all the cells in the closed index interval $[i, j]$, where $0 \leq i \leq j \leq a.\text{length}() - 1$. By the notation 0^{64} we mean a 64-bit computer word that has all its bits set to 0.

3.1 Sparse Bit-Set

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

```
Class SparseBitSet

words: array of 64-bit int                                  // words.length() = $p$
currToOrig: array of int                                    // currToOrig.length() = $p$
limit: int
mask: array of 64-bit int                                   // mask.length() = $p$

Method initSparseBitSet($nbits$: int)
    $p \leftarrow \lceil \frac{nbits}{64} \rceil$
    words $\leftarrow$ array of long of length $p$, first $nbits$ set to 1
    mask $\leftarrow$ array of long of length$p$, all bits set to 0
    currToOrig $\leftarrow [0, \ldots, p-1]$
    limit $\leftarrow p - 1$

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

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

Method flipMask()
    for $i \leftarrow 0$ to limit do
        mask[$i$] $\leftarrow \sim$ mask[$i$]                                // bitwise NOT

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

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

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

Algorithm 1: Pseudo code for the class SparseBitSet.

3.1.1 Fields

Todo: Add examples.

Lines 2–5 of Algorithm 1 show the fields of the class `SparseBitSet` and their types. Here follows a more detailed description of them:

- `words` is an array of p 64-bit words, where each word (abstractly) maps to an element in the set of words $\{w_0, w_1, \ldots, w_{p-1}\}$. Initially, $\texttt{words}[i] = w_i$ for all i. This array defines the current value of the bit-set: the ith bit of word w_j is 1 if and only if the $((j-1) \cdot 64 + i)$th element of the set is present. Upon initalisation, all words in the array have all their bits set to 1, except the last word, which may have a suffix of bits set to 0.

  When performing operations on `words`, the words are continoulsly re-ordered so that all the non-zero words are located at indices less than or equal to `limit`, and all the words that consist of only zeros are located at positions strictly greater than `limit`. Example.

- `currToOrig` is an array that manages the indices of the words in `words`, making it possible to perform operations on non-zero words only. For each word in `words`, `currToOrig` maps its *current* index to its *original* index: $\texttt{words}[i] = w_{\texttt{currToOrig}[i]}$ for all i.

- `limit` is the index of `currToOrig` and `words` 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:

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

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 Section 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 17-16 clears the temporary mask. This means setting to 0 all words of `mask` corresponding to non-zero words of `words`.

- flipMask() in lines 14-19 flips the bits in the temporary mask.

- addToMask() in lines 20–23 applies word-by-word logical bit-wise *or* operations 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 24–34 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, the word and its index are swapped with the last non-zero word and the index of the last non-zero word, respectively, and `limit` is decreased by one.

  In Section 4 we will see that the implementation can actually skip Line 31 and 33 because it is unnecessary to save information about the zerowords in a copy-based solver such as Gecode. We keep this line here though, as the invariant (3.1) would not hold otherwise.

- intersectIndex() in lines 35–40 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 38 and return the index of the word if the intersection is non-empty. If the intersection is empty for all words, then -1 is returned.

3.2 The Compact-Table Algorithm

The CT algorithm is a domain-consistent propagation algorithm for any TABLE constraint. Section 3.2.1 presents pseudo code for the CT algorithm and a few variants, and Section 3.2.2 proves that CT fulfils 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_0, and $vars(c)$, the variables that are associated with c. In what follows, we call the *initial valid table* for c the sublist $T \subseteq T_0$ 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 its *initial domain* $\underline{\mathrm{dom}}(x)$ and its *current domain* $\mathrm{dom}(x)$. 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:

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

- `supports`, a static array of bit-sets representing the supports for each variable-value pair (x, a). The bit-set `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), we have that `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.

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

Algorithm 2: Compact Table Propagator.

- `vars`, an array of variables that represent *vars*(c).

Algorithm 2 shows the CT algorithm. Lines 1-4 initialise the propagator if it is being posted (initialised). CT reports failure in case a variable domain was wiped out in InitialiseCT() or if `validTuples` is empty, meaning no tuples are valid. If the propagator is not being posted, then 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 `validTuples` is modified, then 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.

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

Lines 1–5 perform bounds propagation to limit the domain sizes of the variables, which in turn will limit the sizes of the data structures. These lines remove 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), then an empty store is returned.

Lines 6–8 initialise local variables for later use.

Lines 9–11 initialise the fields `residues`, `supports`and `vars`. The field `supports` is initialised as an array of empty bit-sets, with one bit-set for each variable-value pair, and the size of each bit-set being the number of tuples in T_0.

Lines 12–22 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 $x \in vars(c)$. 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 20. We also take the opportunity to store the word index of the found support in `residues`$[x, t[x]]$ in line 21. Line 23 increases

```
PROCEDURE InitialiseCT($s$: store, $T_0$: list of tuples, $vars(c)$: seq. of variables) : store
1:  foreach $x \in s$ do
2:      $R \leftarrow \{a \in s(x) : a > T_0.\max() \ \vee \ a < T_0.\min()\}$
3:      $s \leftarrow s[x \mapsto s(x) \setminus R]$
4:      if $s(x) = \emptyset$ then
5:          return $\emptyset$
6:  $npairs \leftarrow \text{sum}\,\{|s(x)| : x \in \texttt{vars}\}$          // Number of variable-value pairs
7:  $ntuples \leftarrow T_0.\text{size}()$                                   // Number of tuples
8:  $nsupports \leftarrow 0$                                                // Number of found supports
9:  residues $\leftarrow$ array of length $npairs$
10: supports $\leftarrow$ array of length $npairs$ with bit-sets of size $ntuples$
11: vars $\leftarrow vars(c)$
12: foreach $t \in T_0$ do
13:     $supported \leftarrow$ true
14:     foreach $x \in$ vars do
15:         if $t[x] \notin s(x)$ then
16:             $supported \leftarrow$ false
17:             break                                                       // Exit loop
18:     if $supported$ then
19:         foreach $x \in$ vars do
20:             supports$[x, t[x]][nsupports] \leftarrow 1$
21:             residues$[x, t[x]] \leftarrow \lfloor \frac{nsupports}{64} \rfloor$    // Index for the support in validTuples
22:             $nsupports \leftarrow nsupports + 1$
23: foreach $x \in$ vars do
24:     $R \leftarrow \{a \in s(x) : \texttt{supports}[x, a] = \emptyset\}$
25:     $s \leftarrow s[x \mapsto s(x) \setminus R]$
26:     if $s(x) = \emptyset$ then
27:         return $\emptyset$
28: validTuples $\leftarrow$ SparseBitSet with $nsupports$ bits set to 1
29: return $s$
```

Algorithm 3: Initialising the CT propagator.

the counter.

Lines 23–27 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 28 initialises `validTuples` as a `SparseBitSet` object with *nsupports* bits, initially with all bits set to 1 since *nsupports* tuples are initially valid supports for c. At this point $nsupports > 0$, otherwise we would have returned at line 27.

```
PROCEDURE UpdateTable($s$: store, $x$: variable)
validTuples.clearMask()
foreach $a \in s(x)$ do
    validTuples.addToMask(supports$[x, a]$)
validTuples.intersectWithMask()
```

Algorithm 4: Updating the current table. This procedure is called for each variable whose domain is modified since the last invocation.

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 store 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 in a CP solver that that runs UpdateTable() for each variable $x \in \mathit{vars}(c)$ whose domain has changed since the last invocation.

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 $x \in$ vars 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 domain value for a variable x, causing a wipe-out for x. Then, by the definition in formula (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). If it has not, then we search in line 5 for an index in `validTuples` where a valid support for the variable-value pair (x, a) is found, thereby checking the condition (3.3). If such an index exists, then we cache it in `residues`$[x, a]$, and if it does not, then we remove a from $\mathrm{dom}(x)$ 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(), then it is not necessary to attempt to filter out values from the domain of x, because every value of x will have a support in `validTuples`. Hence, in Algorithm 5, we only execute lines 2–9 for `vars` $\setminus \{x\}$.

- For `residues`, maintain as invariant that `residues`$[x, a]$ is the *highest* index in `words` where a support for (x, a) can be found, if such an index exists. This property holds upon initialisation, since `residues`$[x, a]$ will be set to the latest found index of a support for (x, a)

in line 21 in InitialiseCT(). The invariant is maintained by executing the loop in intersectIndex() from highest index to lowest index instead of the other way around. By letting intersectIndex() take an extra argument that defines the loop limit, we can skip a few iterations by starting the loop at index `residues`$[x, a]$, by passing `residues`$[x, a]$ as an argument to intersectIndex() in FilterDomains().

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

CT(Δ) – *Using delta information in* UpdateTable(). For a variable x, Δ_x is the set of values that were removed from x since the last invocation of the propagator. If the CP solver provides information about Δ_x, then that information can be used in UpdateTable(). Algorithm 6 shows a variant of UpdateTable() that uses delta information. If $|\Delta_x|$ is smaller than $|\text{dom}(x)|$, then we accumulate to the temporary mask the set of invalidated tuples, and then flip the bits in the temporary mask before intersecting it with `validTuples`, else we use the same approach as in Algorithm 4.

```
PROCEDURE UpdateTable($s$: store, $x$: variable)
validTuples.clearMask()
if $\Delta_x$ is available $\wedge$ $|\Delta_x| < |s(x)|$ then
  foreach $a \in \Delta_x$ do
    validTuples.addToMask(supports$[x, a]$)
  validTuples.flipMask()
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 an addition made to the algorithms described in [5]. If only one valid tuple is left after all calls to UpdateTable() are finished, then the domains of the variables can be fixed to the values for that tuple directly. 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 invocation then
  if ($index$ $\leftarrow$ validTuples.indexOfFixed()) $\neq -1$ then
    return $\langle$SUBSUMED, $s[x \mapsto T[index][x] : x \in$ vars$]\rangle$
  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 set bit if and only if

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

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 set bit, else −1. The method IndexOfFixed() is added to the class `SparseBitSet` and assumes access to builtin MSB which returns the index of the most significant bit of a given int.

```
Method IndexOfFixed() : int
  $index\_of\_fixed \leftarrow -1$
  if limit $= 0$ then
    $w \leftarrow$ words[0]
    if $(w \,\&\, (w - 1)) = 0$ then                    // Exactly one set bit
      $offset \leftarrow$ currToOrig[0]
      $index\_of\_fixed \leftarrow offset \cdot 64 + \mathrm{MSB}(w)$
  return $index\_of\_fixed$
```

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

3.2.2 Proof of properties for CT

We now prove 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 by $CT(s)$ 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, then all values still have a support because all tuples that were valid in the previous invocation are still valid.

So, in both cases every variable-value pair (x, a) has a support, 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 from the domain of a variable from the store $CT(s)$.

By (3.3), there must exist 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 remove 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 $\tau = \langle s(x_1), \ldots, s(x_n) \rangle \notin \mathit{rel}(c)$.

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 $(x_i, s(x_i))$ that 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, then `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 because it can never be the case that CT erroneously signals fixpoint. 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 $a \in \mathit{dom}(x)$. 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: Describe more clearly what modifications were made to the algoritms to suit the target solver.

Now the implementation of the CT algorithm presented in Section 3 will be described. This section 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 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 propagator is the same as 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 $\langle variable, value\rangle \rightarrow$ `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 depends on the domain width of `supports` and `residues`, and by storing the initial minimum value for the variable, so that `supports`$[x, a]$ and `residues`$[x, a]$ are stored at index $a - min$ in the respective array. If the domain is sparse, then the sizes of `supports` and `residues` are 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 at least $H = (2 \cdot \text{size} + \text{size} + 2 \cdot \text{size})$ memory units, because the size of the hash table is at least $2 \cdot \text{size}$. The quantities S and H are equal when $R = \frac{4}{3} \approx 1.33$. Because the hash table might have collisions, this strategy does not always take 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.

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 report failure in case `validTuples` is empty. There are several benefits to using advisors. First, without advisors, the propagator would need to scan all the variables to determine which ones have been modified since the last invocation of the propagator, and execute UPDATETABLE() on those, which would be time consuming. Second, the advisors can store the data structures that belong to its variable. This means 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. That implementation uses 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 implementation first reasons on the bounds of the domains of the variables, enforcing bounds consistency, before enforcing domain consistency. The reason for this is that iterating over domains is expensive. This candidate 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 propagator 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 profiling 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), which uses the set of values Δ_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 approach would be to generalise this condition to $|\Delta_x| < k \cdot |s(x)|$, where $k \in \mathbb{R}$ is some suitable constant; this is something that remains to be investigated.

5 Evaluation

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

The benchmarks consist of 30 groups with a total of 1507 CSP instances, involving TABLE constraints only, that were used in the experiments in [5]. These instances were chosen because they contain a large variety of instances, and the fact that they were used in [5] to evaluate CT in OR-tools suggests that they are also appropriate for evaluating CT in the context of Gecode.

The fact that the instances contain TABLE constraints only is not an issue, because since all the propagators are domain consistent, they will perform the same amount of propagation at each node in the search tree, and therefore it is only the difference in runtime that is interesting to measure. Consequently, if the intances would contain other constraints than TABLE, then a less amount of the total runtime would be spent in the propagators that are measured, which would give a weaker performance difference.

Table 1: Groups of benchmarks and their characteristics.

name	number of instances	arity	table size	variable domains
A5	50	5	12442	0..11
A10	50	10	51200	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
BDD Small	35	18	TBA	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	13	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..3 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	$2-10$	$3-48721$	Vary from 0..1 to 0..41
Nonograms	180	2	$1-1562275$	Vary from 1..15 to 1..980
Pigeons Plus	40	$2-10$	$10-390626$	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
TSP 20	15	2, a few 3	TBA, a few 1	Vary from singleton to 0..1000

The models used in [5] were originally written in XCSP2.1, an XML format used for expressing CSP models. The models were compiled to MiniZinc [7] using the compiler in [cite to compiler]. Of the 1621 instances that were used in [5], only 1507 could be used due to parse errors in the compilation process. The groups of benchmarks 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 was compared, and the winning version was compared against the existing propagators for the Table constraint in Gecode, as well as with the propagator for the Regular constraint.

A timeout of 1000 seconds was used throughout the experiments. Instances that i) could be solved within 1 s for all propagators, or ii) caused a memory-out for at least one of the propagators, were filtered out from the results. Instances that timed out were not filtered out from the results.

5.1 Comparing Different Versions of CT

5.1.1 Evaluation Setup

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

CT Basic version.

CT(Δ) CT using Δ_x, the set of values that have been removed from $\mathrm{dom}(x)$ 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, as discussed in Section 4.

5.1.2 Results

The plots from the experiments are presented in Appendix A.

5.1.3 Discussion

The results indicate that CT(Δ) outperforms the other variants. The performances of CT and CT(T) are similar, and CT(B) is overall slower than the other variants. On *AIM-50*, which only contains 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 [8].

B – Basic positive tuple set propagator This is based on [2].

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. Initialised 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 $\langle x, n\rangle$. Initialised 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 the last invocation 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 tuple 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($\langle x, n\rangle$)
$\ell \leftarrow L[\langle x, n\rangle]$
while $\ell \neq \bot \land \exists y \in X : \ell[y] \notin D(y)$
    $\ell \leftarrow L[\langle x, n\rangle] \leftarrow$ next tuple for $\langle x, n\rangle$
if $\ell = \bot$ then
    $W_R \leftarrow W_R \cup \{\langle x, n\rangle\}$
else
    foreach $y \in X$ do
        $S[\langle y, \ell[y]\rangle] \leftarrow S[\langle y, \ell[y]\rangle] \cup \{\ell\}$
```

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($\ell, \langle x, n\rangle$)
foreach $y \in X$ do
    $S[\langle y, \ell[y]\rangle] \leftarrow S[\langle y, \ell[y]\rangle] \setminus \{\ell\}$
    if $y \neq x \land S[\langle y, \ell[y]\rangle] = \emptyset$ then
        $W_S \leftarrow W_S \cup \{\langle y, \ell[y]\rangle\}$
```

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

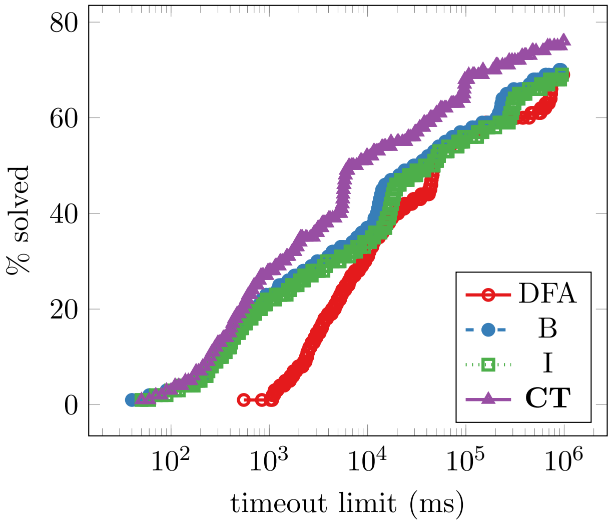

Figure 2: Percentage of instances solved as a funcion of time for the DFA, B, I, and CT propagators.

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 as with the propagator for the REGULAR constraint on the groups of benchmarks listed in Table 1. The propagators are denoted:

CT The compact-table propagator, version CT(Δ).

DFA Layered graph (DFA) propagator, based on [8].

B Basic positive tuple set propagator, based on [2].

I Incremental positive tuple set propagator.

5.2.2 Results

The final set of instances used in the results consists of 661 instances, having filtered out instances that were either solved in less than 1 s for all propagators, or that caused a memory-out for at least one propagator. Figure 2 shows the percentage of instances solved as a function of timeout limit in ms for these 661 instances. Within the timeout of 1000 s, CT could solve the highest number of instances (76 %), followed by B (70 %), I (69 %), and DFA (69 %).

Among the 846 instances that were filtered out, 207 were filtered out because of memory-outs. Among these 207 instances, DFA ran out of memory on all of them, and CT, B, and I ran out of memory on 36 of them.

The plots from each individual group of benchmarks are presented in Appendix B, except for the groups *MDD 07* and *MDD 09*, where all instances either timed out or caused a memory-out.

5.2.3 Discussion

Runtime. CT performs either as well as or better than all other propagators, on all groups except *AIM 200*, where CT was slightly slower than B and DFA on two instances, and on *BDD Small* and *BDD Large* where CT was slightly slower than B and I on the small instances. At best, CT is about a factor 10 faster than the other algorithms on some groups. CT could solve as many instances as, or more, than all other propagators, on all groups except *Pigeons Plus* where DFA could solve one more instance.

Another notable observation is that B seems to outperform I, even though I is said to be more efficient than B in terms of execution speed.

On the various groups the performance gain from CT varies, which might depend on the characteristics for the different groups of benchmarks. Here the impact of *table size*, *arity* and *domain size* on runtime performance is discussed:

Table size The increase of performance for CT compared to the other propagators is larger on the groups that contain instances with large table sizes only (see *A5*, *A10*, *K5*, *MDD 05*, and *Rands JC**), than on the groups that contain only small tables (see *AIM-**, *Dubois*, and *Geom*).

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

Arity Many groups 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 **). However, the groups with low arities also have small tables, while the groups with larger arities tend to have larger tables, which makes it hard to tell whether it is the arity or the table size that impact the performance gain.

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

Memory usage. It can be seen that CT, B, and I has about the same maximum memory usage while DFA consistently has a higher maximum memory usage.

Profiling. It can be seen that the distribution of how the time is spent between propagation, advisors and copying varies between different propagators and different groups of benchmarks.

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 result 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 seem to be largest for constraints with large tables, and more modest for constraints with small tables.

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:

- 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.

- 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 $|\Delta_x| < k \cdot |s(x)|$, for some suitable $k \in \mathbb{R}$; this is something that remains to be investigated.
- For Table constraints involving a small number (at most 64) tuples, the implementation could be simplified, which would save memory and possibly increase execution speed.
- Implement the generalisations of the CT algorithm described in [12].

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] P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM LETTERS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 2:59–69, 1993.

[4] J. J. Dekker. Sub-problem pre-solving in minizinc. Master's thesis, Department of Information Technology, Uppsala University, Sweden, November 2016. Available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-307145`.

[5] 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.

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

[7] 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`.

[8] 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.

[9] P. Schaus, C. Solnon, C. Lecoutre, and et al. Sparse-sets for domain implementation, 2013.

[10] 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.

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

[12] 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.

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.

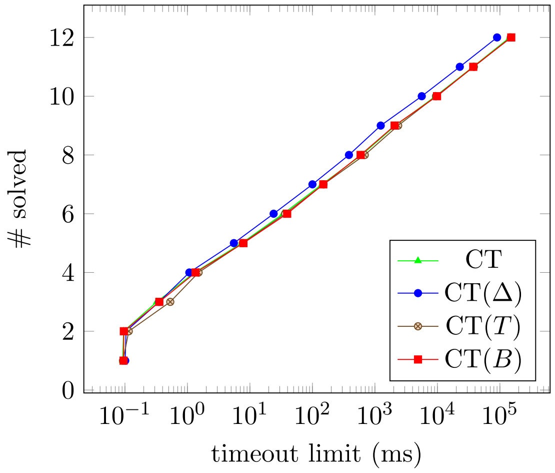

Figure 5: **Langford 4**.

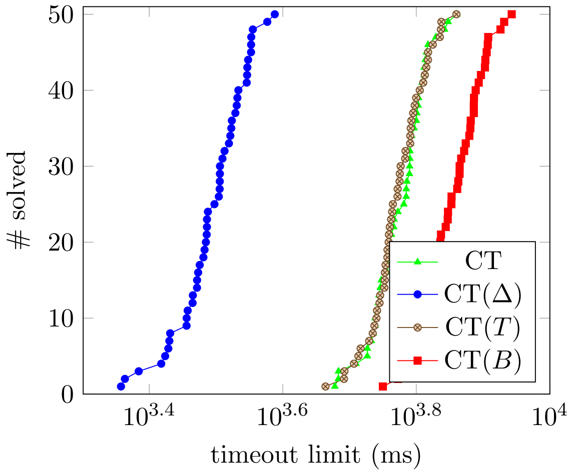

Figure 6: **A5**.

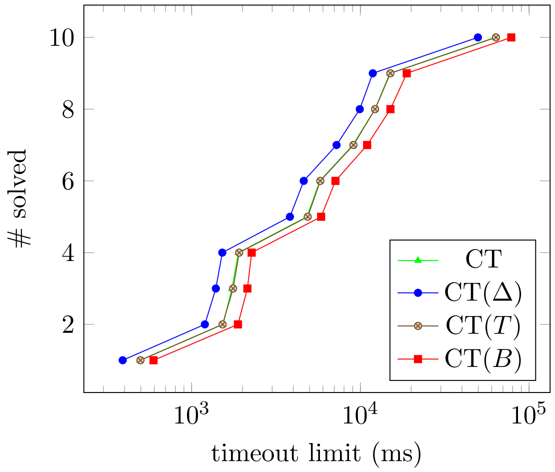

Figure 3: **Rands JC2500.**

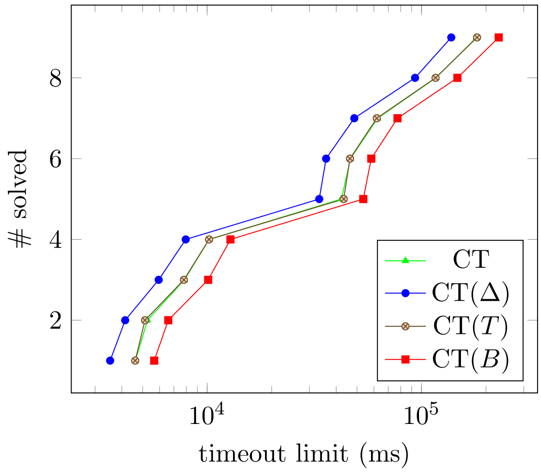

Figure 4: **Rands JC5000**.

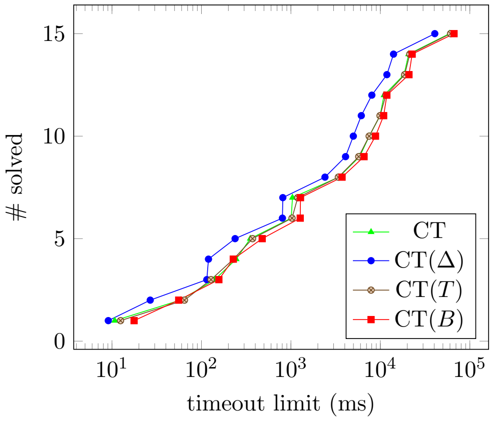

Figure 7: **TSP Quat 20**.

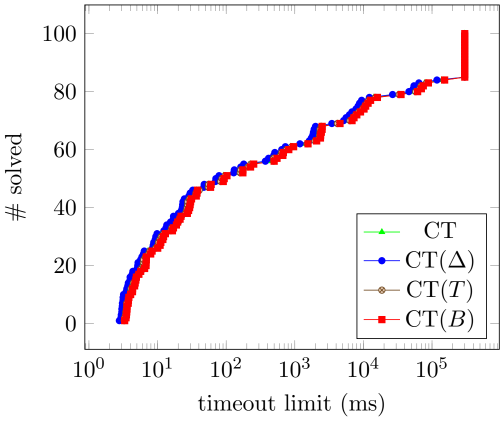

Figure 8: **Geom**.

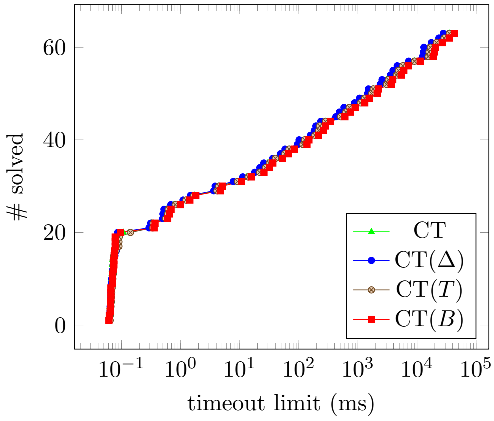

Figure 9: **Crosswords LexVG**.

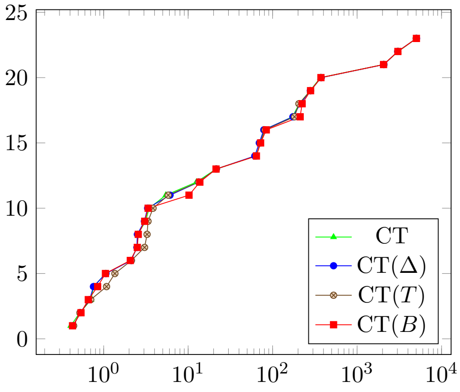

Figure 10: **AIM 50**.

B Plots from Comparison of CT against Existing Propagators

Each benchmark group has one figure, and each figure contains two plots: the first plot shows for each propagator the percentage of solved instances within that group as a function of timeout limit in milliseconds. The second plot shows the maximum memory usage, as well as how the execution time is distriubted between propagation, advisors, and copying. For the first plot, measurements from all instances that took at least 1 second to solve for all propagators, and that did not cause a memory-out for any of the propagators, are included. For the second plot, the measurements were made on one random instance within the group, such that the runtime was at least 10 s for all algorithms (the accuracy of the profiling is assumed to be too low for runtimes below 10 s). For the groups of benchmarks where all instances were solved within 10 s for at least one algorithm, only the memory usage is reported for these groups.

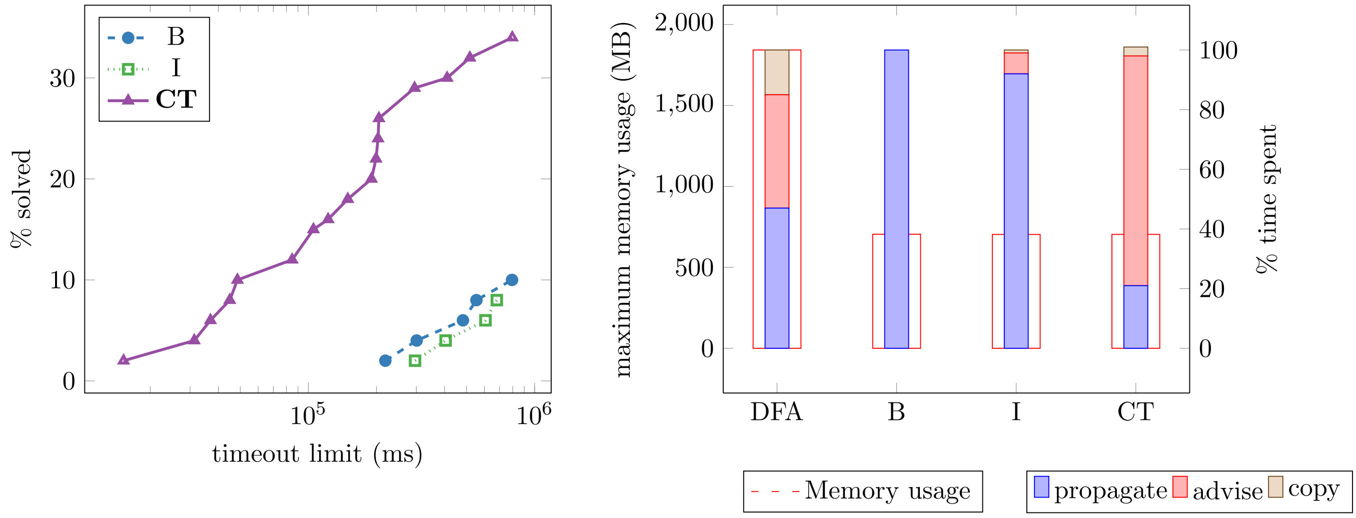

Figure 11: **A10.**

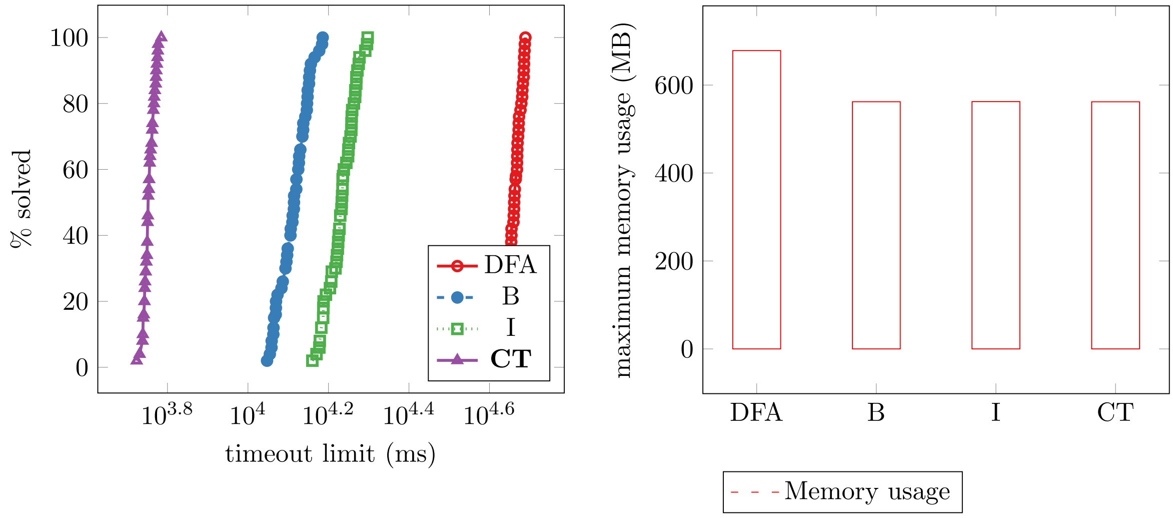

Figure 12: **A5.**

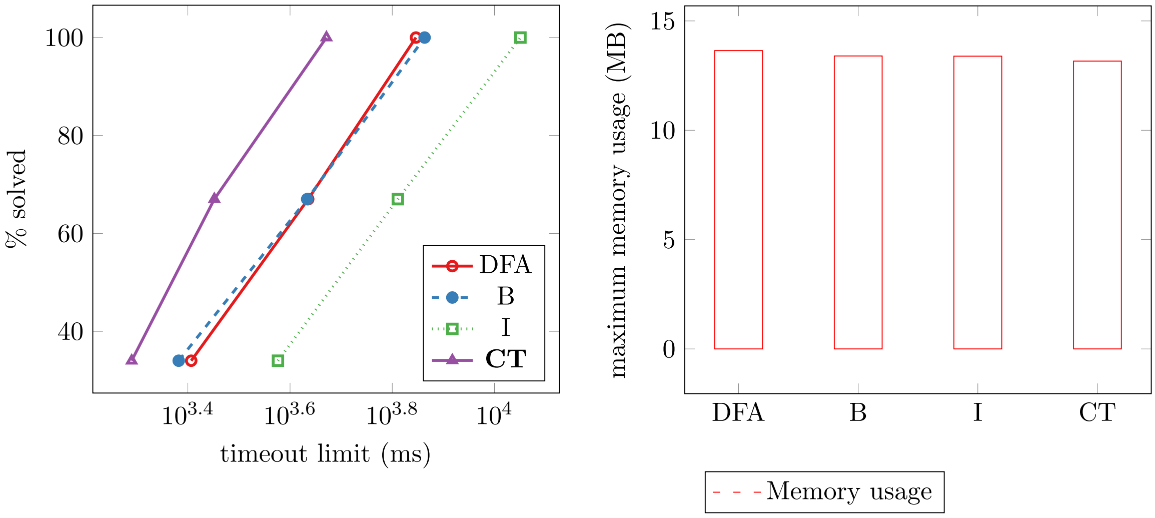

Figure 13: **AIM 50.**

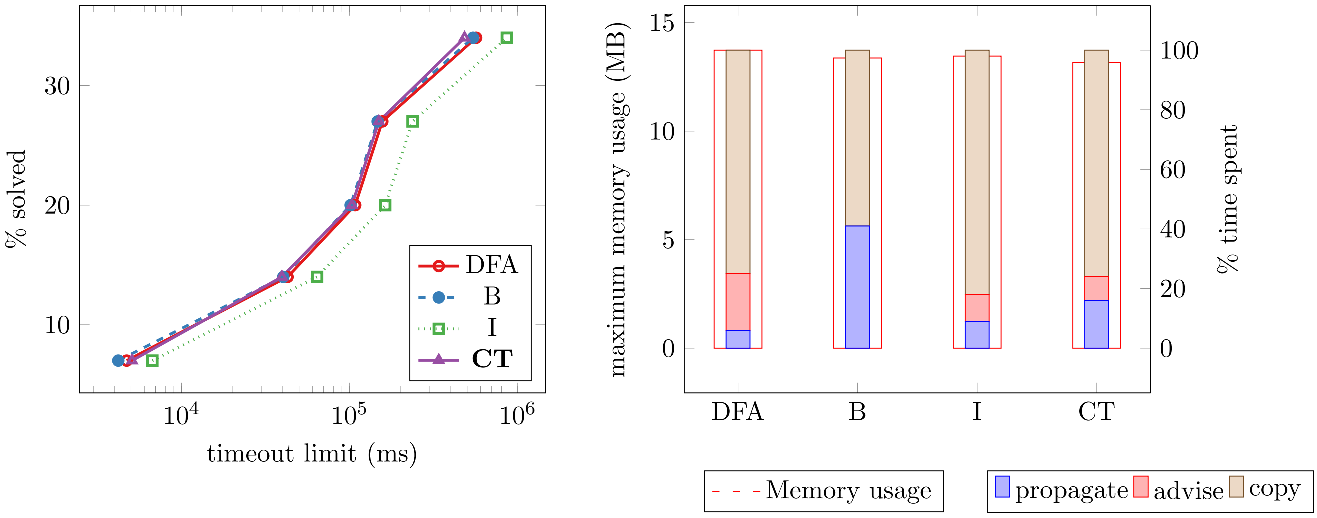

Figure 14: **AIM 100.**

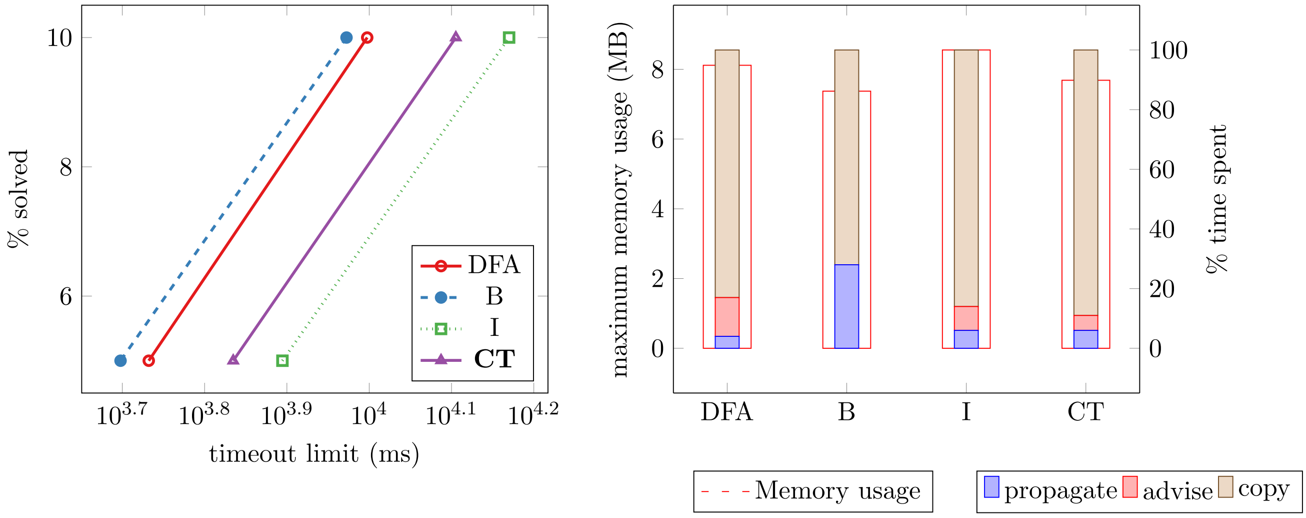

Figure 15: **AIM 200.**

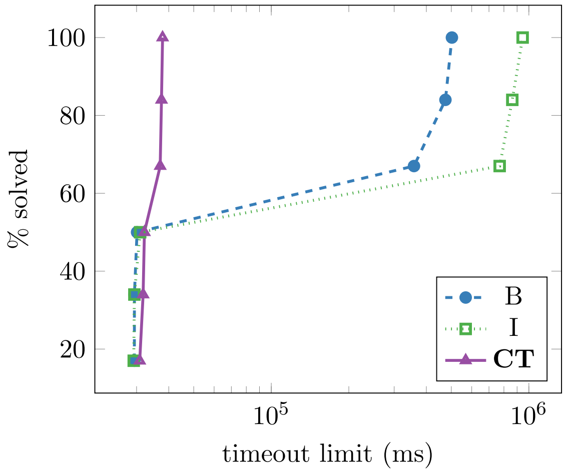

Figure 16: **BDD Small.**

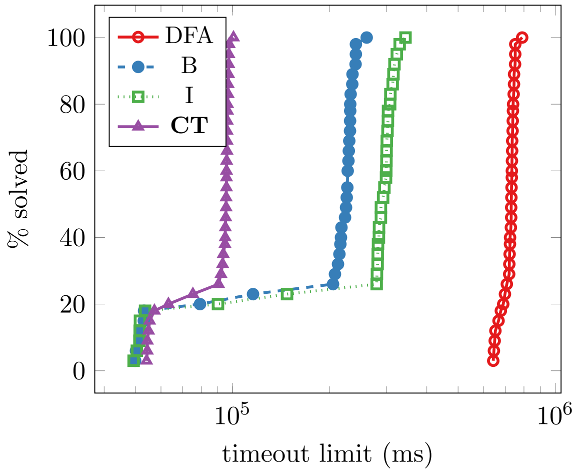

Figure 17: **BDD Large.**

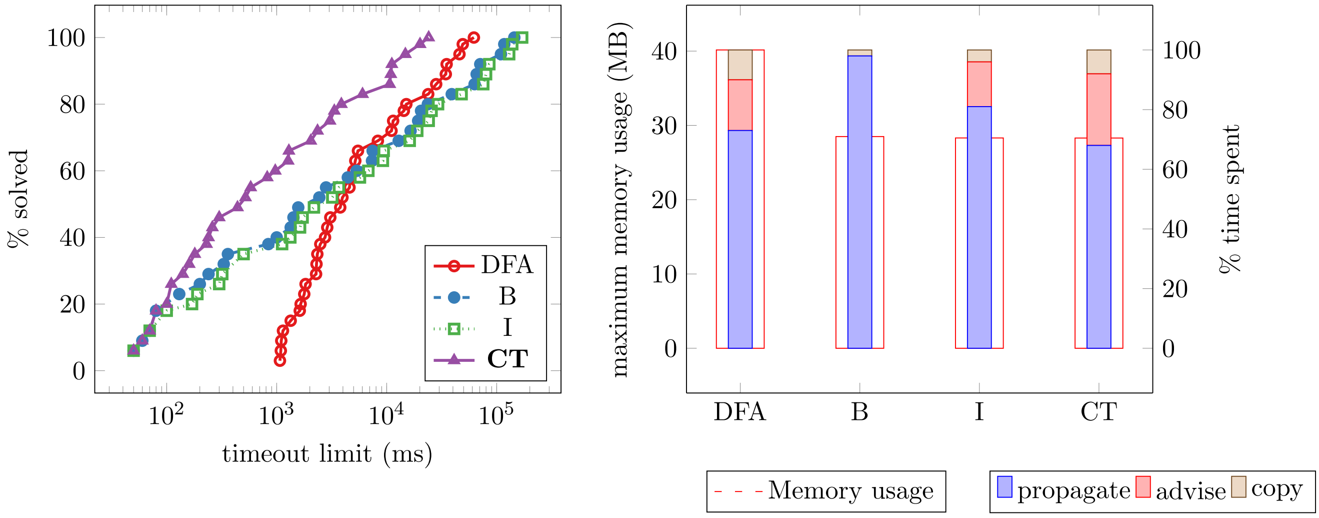

Figure 18: **Crosswords lexVg.**

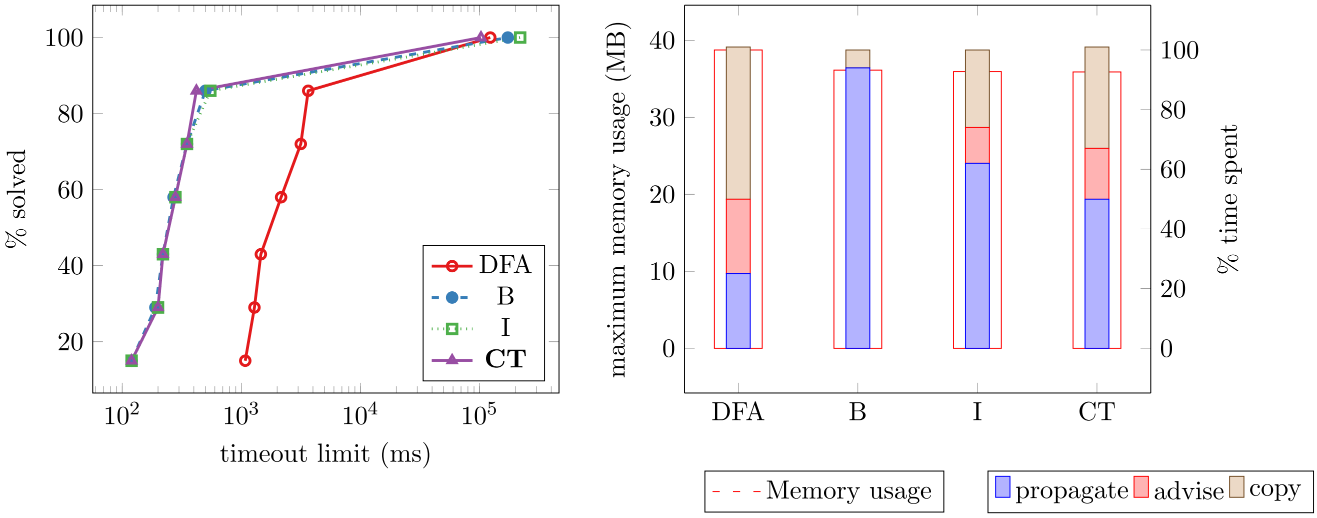

Figure 19: **Crosswords wordsPuzzle.**

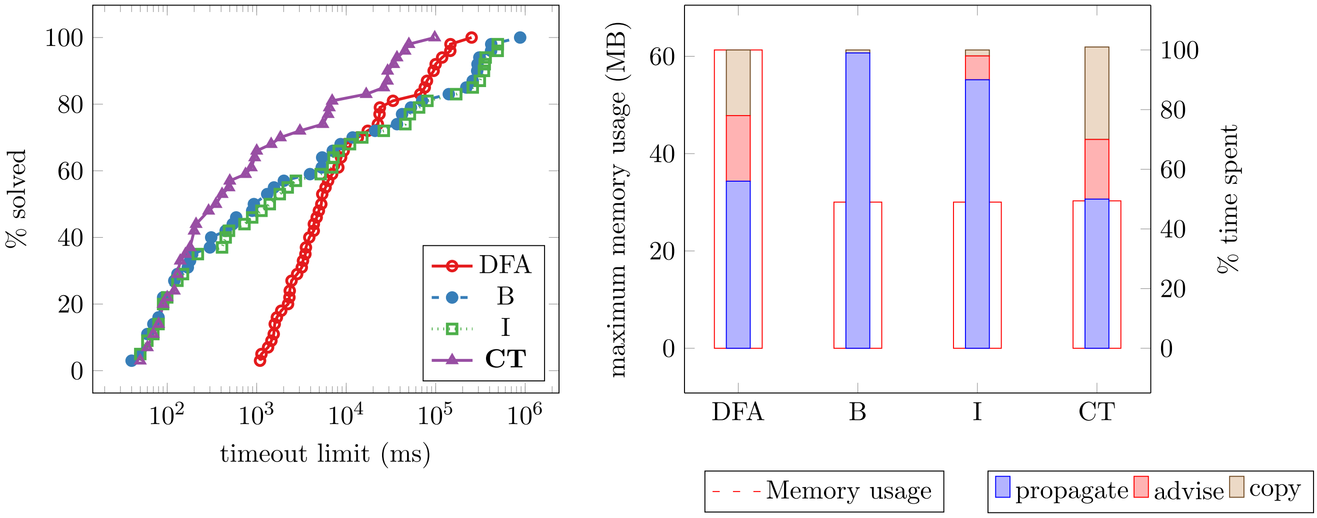

Figure 20: **Crosswords wordsVg.**

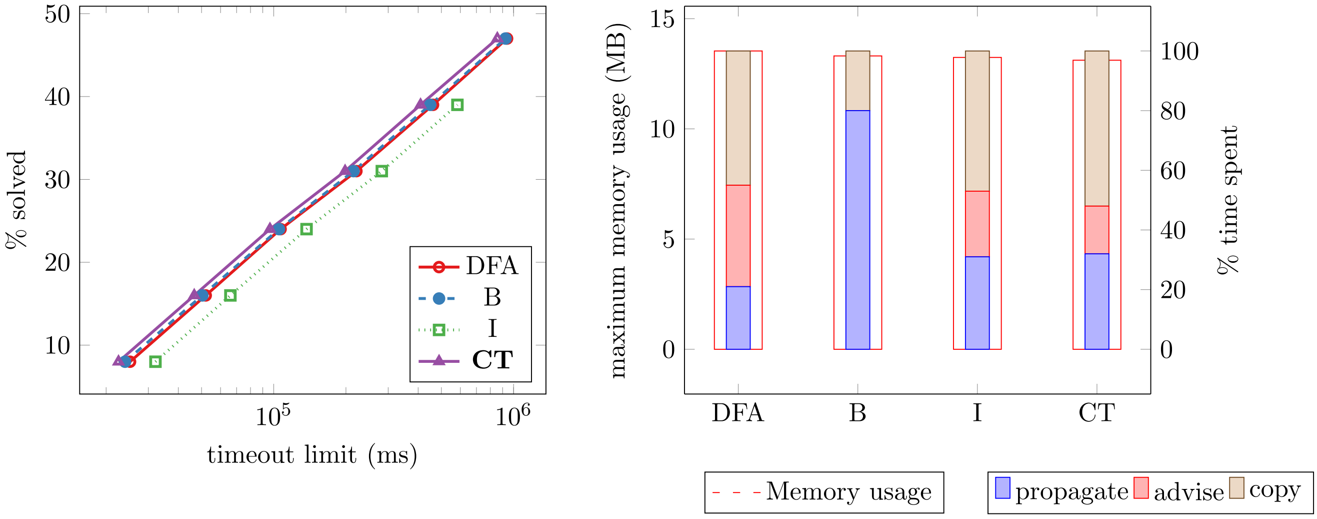

Figure 21: **Dubois.**

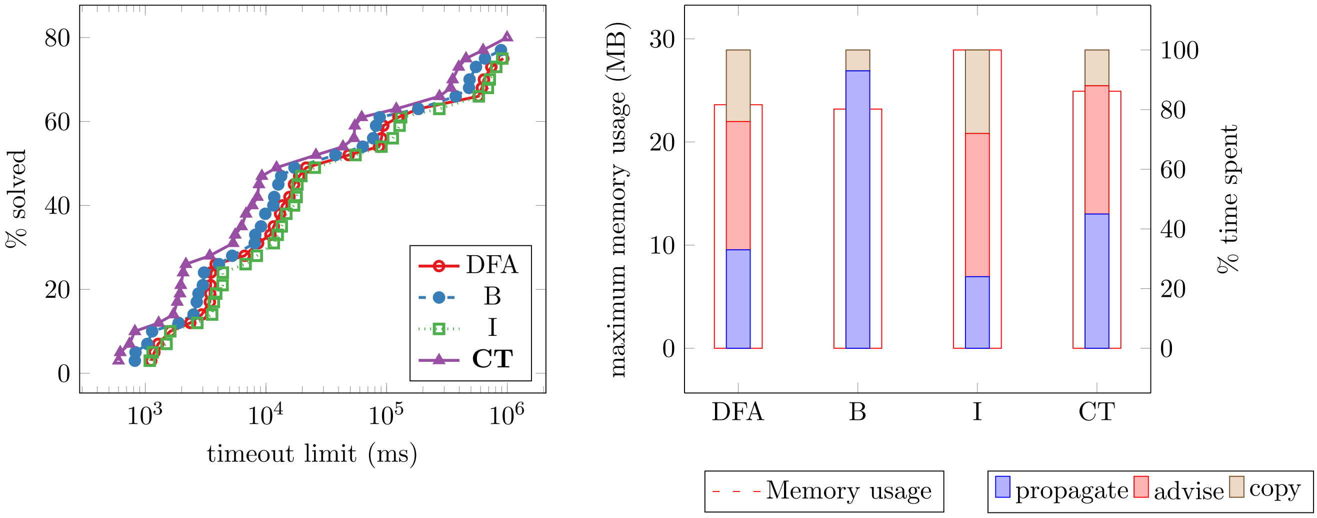

Figure 22: **Geom.**

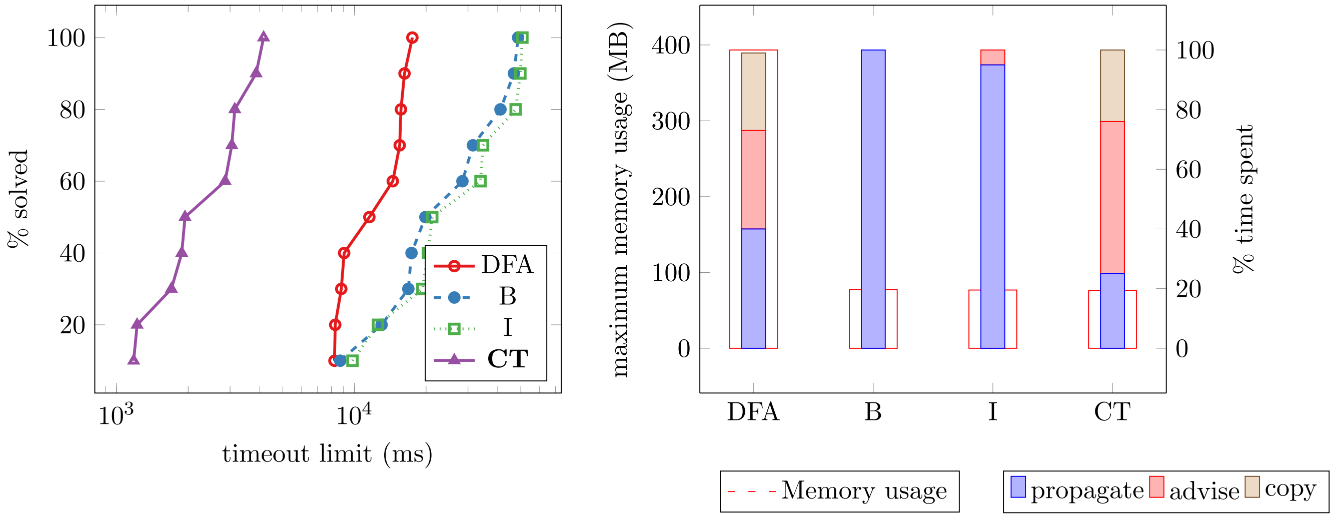

Figure 23: **K5.**

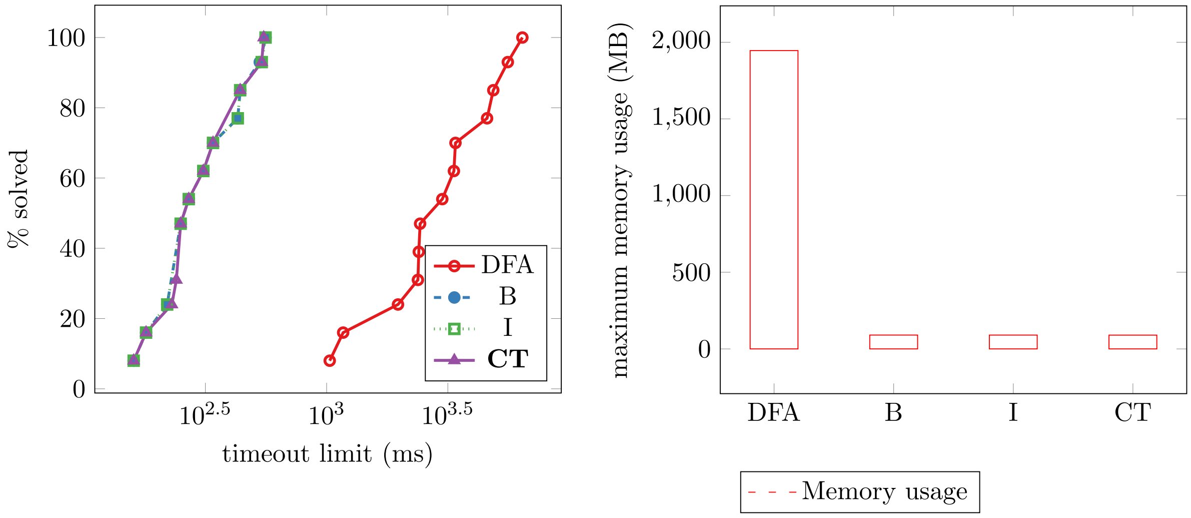

Figure 24: **Kakuro Easy.**

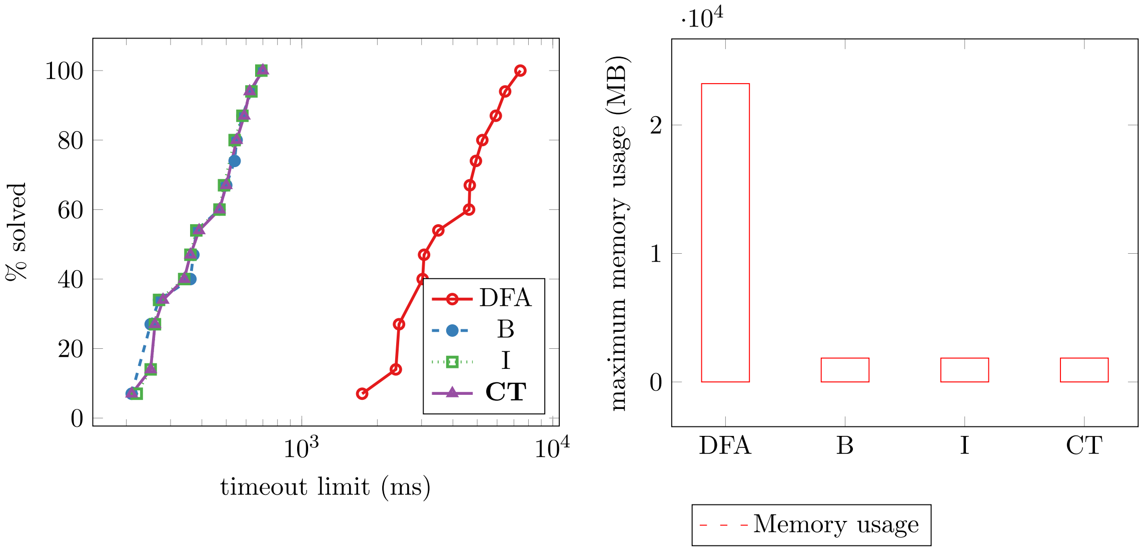

Figure 25: **Kakuro Medium.**

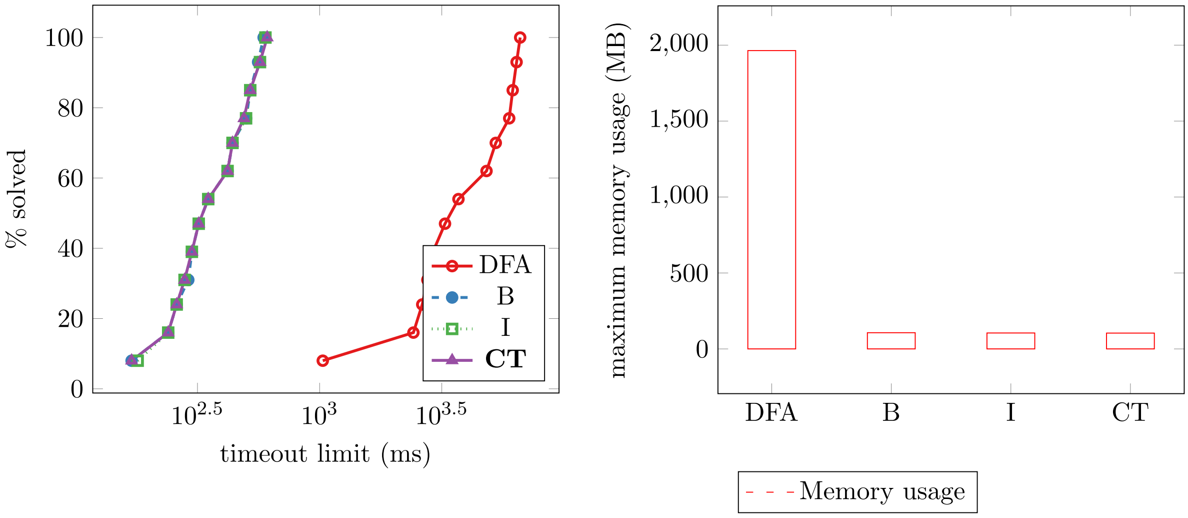

Figure 26: **Kakuro Hard.**

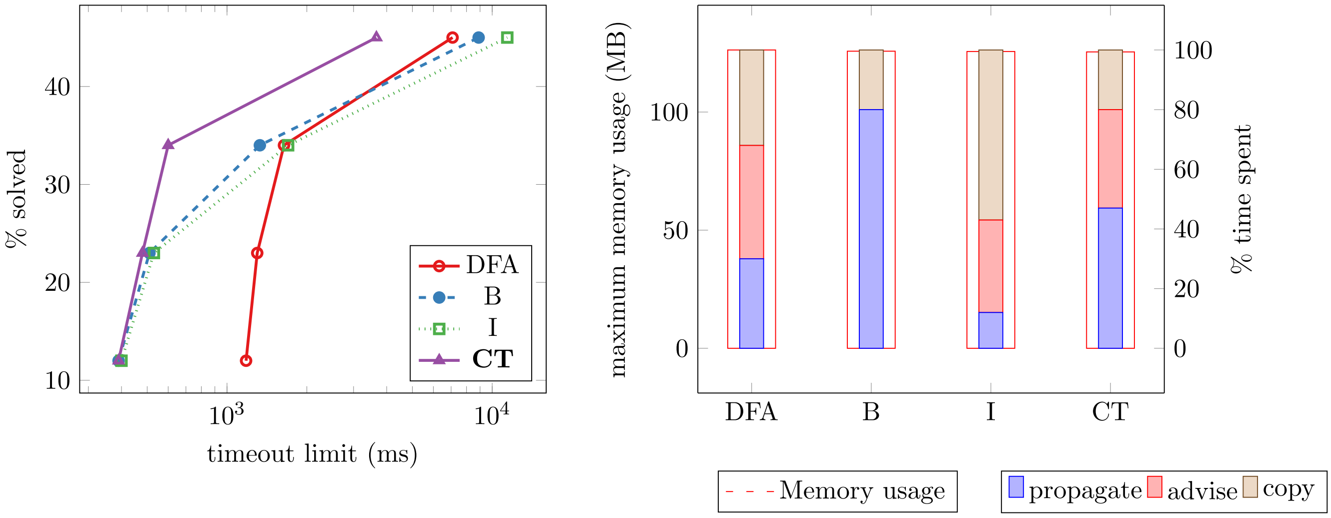

Figure 27: **Langford 2.**

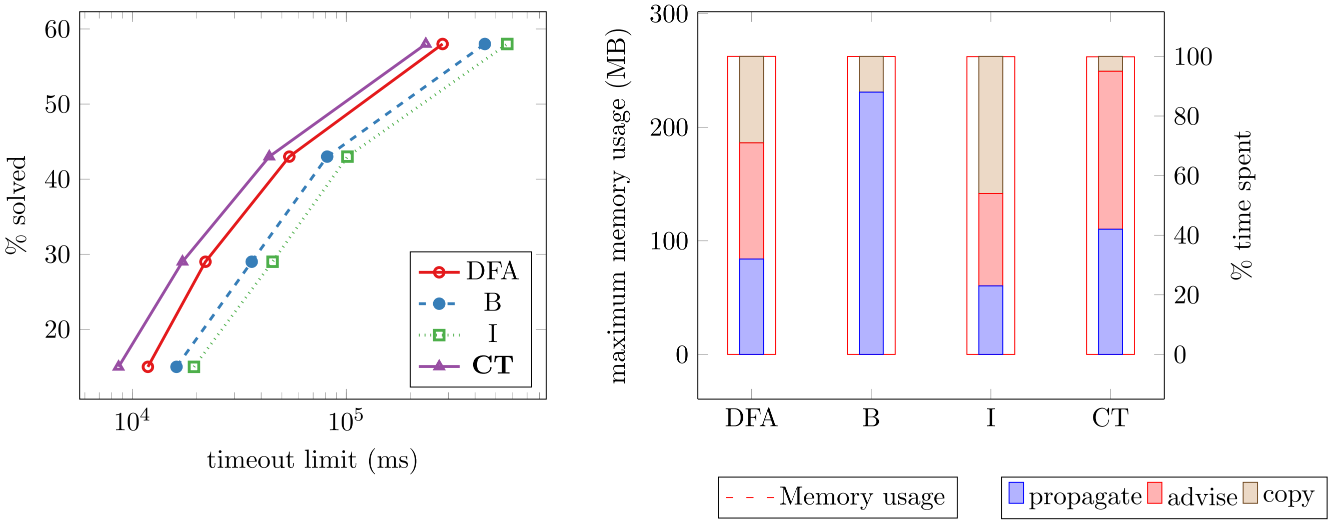

Figure 28: **Langford 3.**

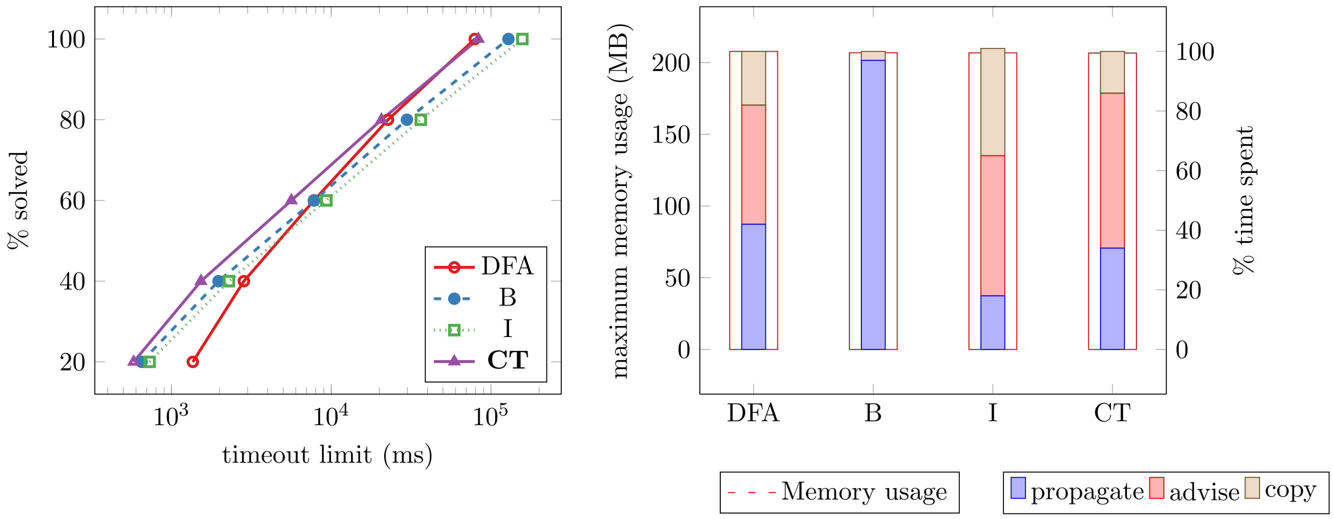

Figure 29: **Langford 4.**

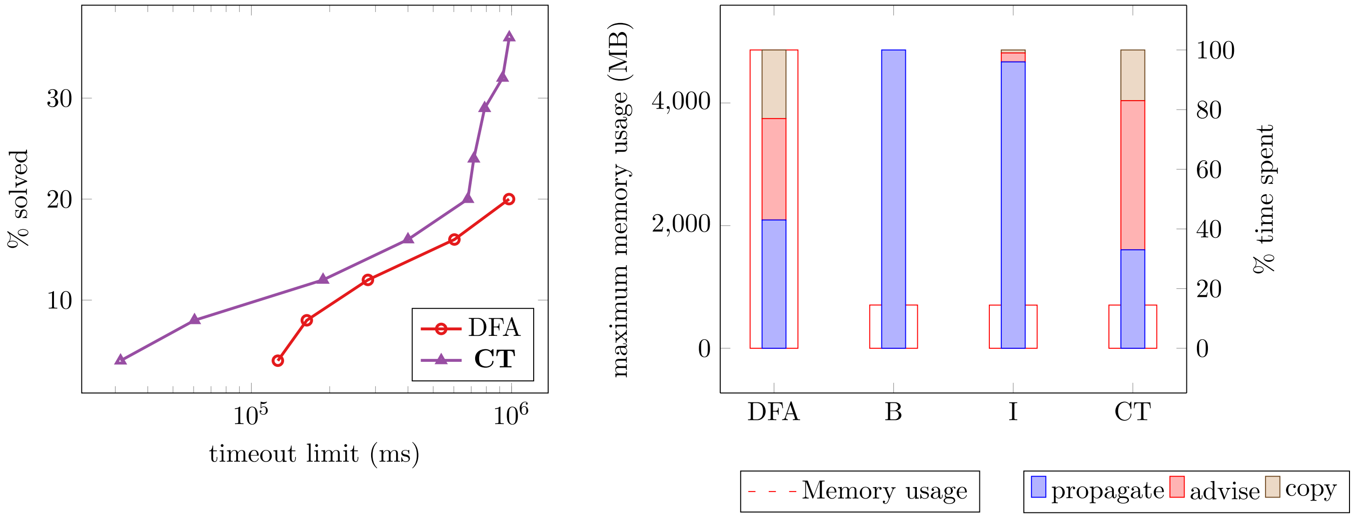

Figure 30: **MDD 05.**

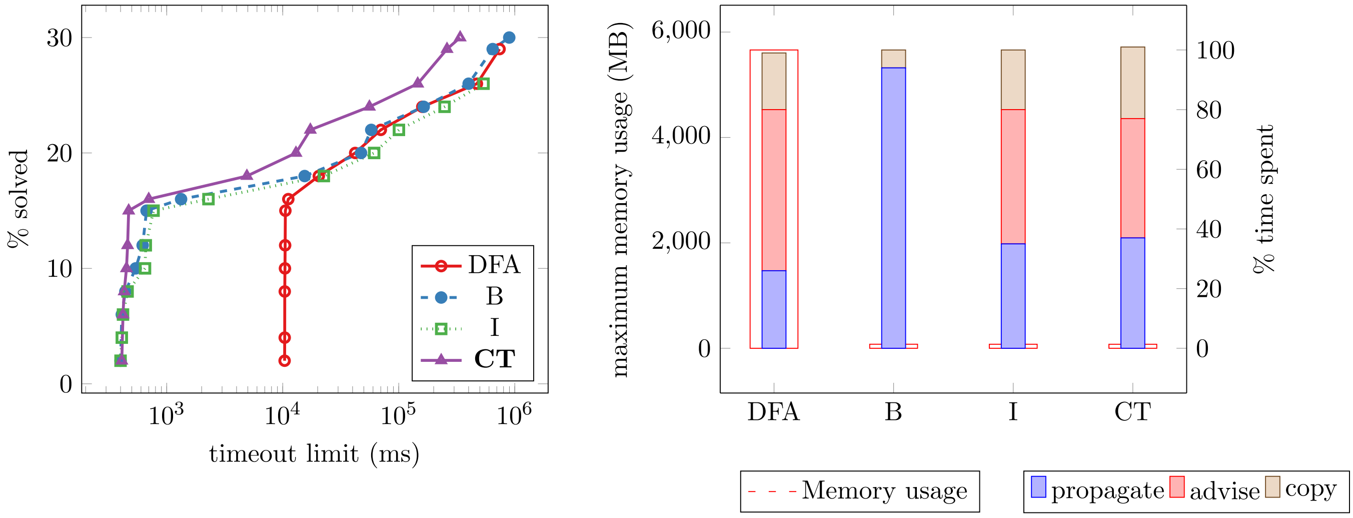

Figure 31: **Mod Renault.**

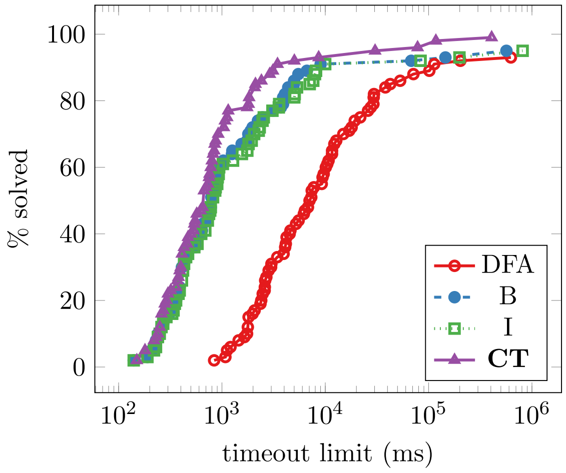

Figure 32: **Nonograms.**

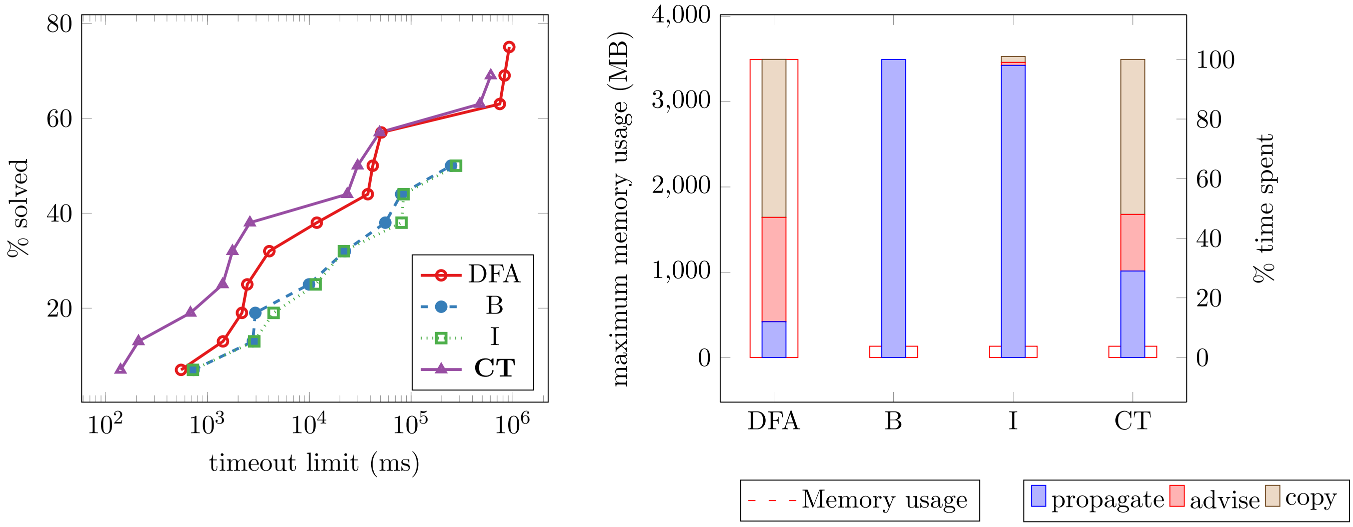

Figure 33: **Pigeons Plus.**

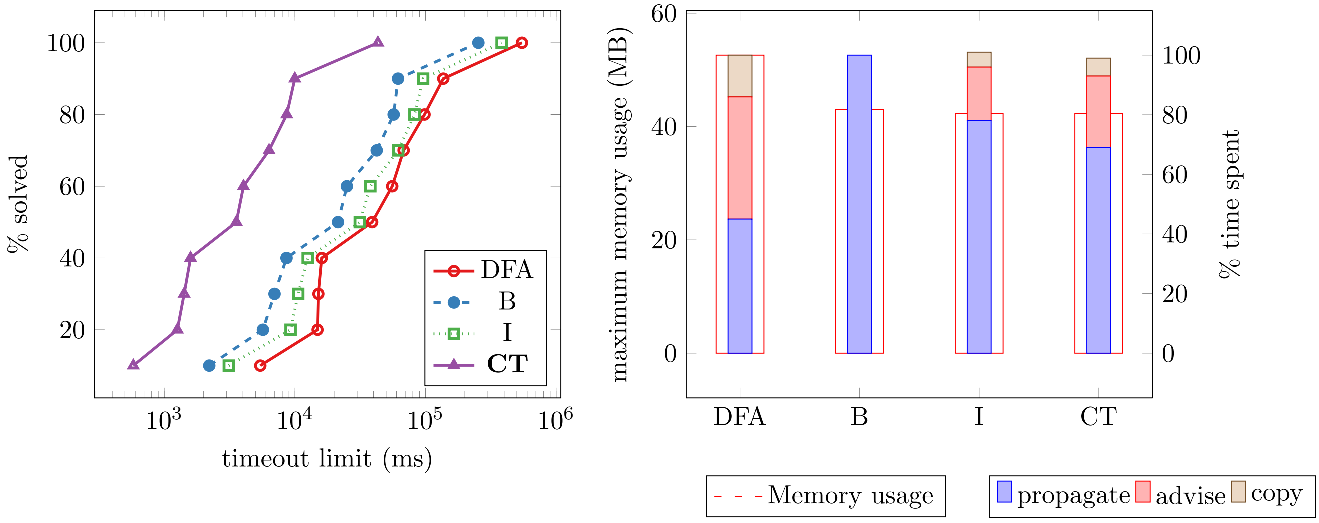

Figure 34: **Rands JC2500.**

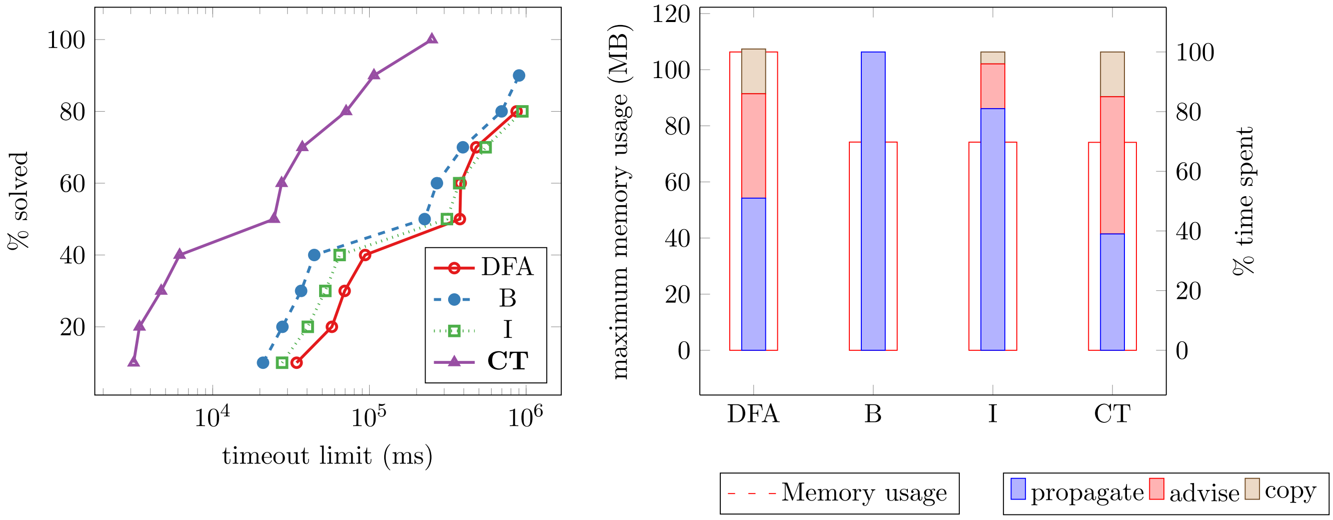

Figure 35: **Rands JC5000.**

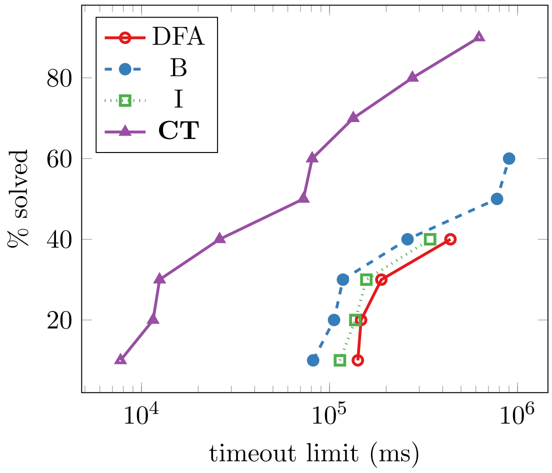

Figure 36: **Rands JC7500.**

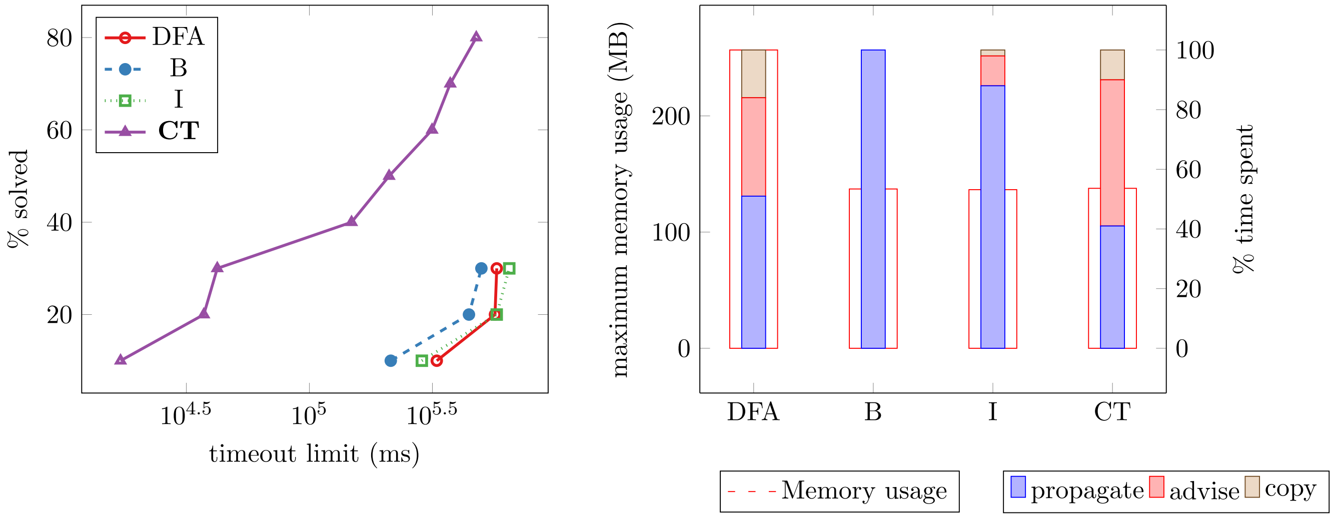

Figure 37: **Rands JC10000.**

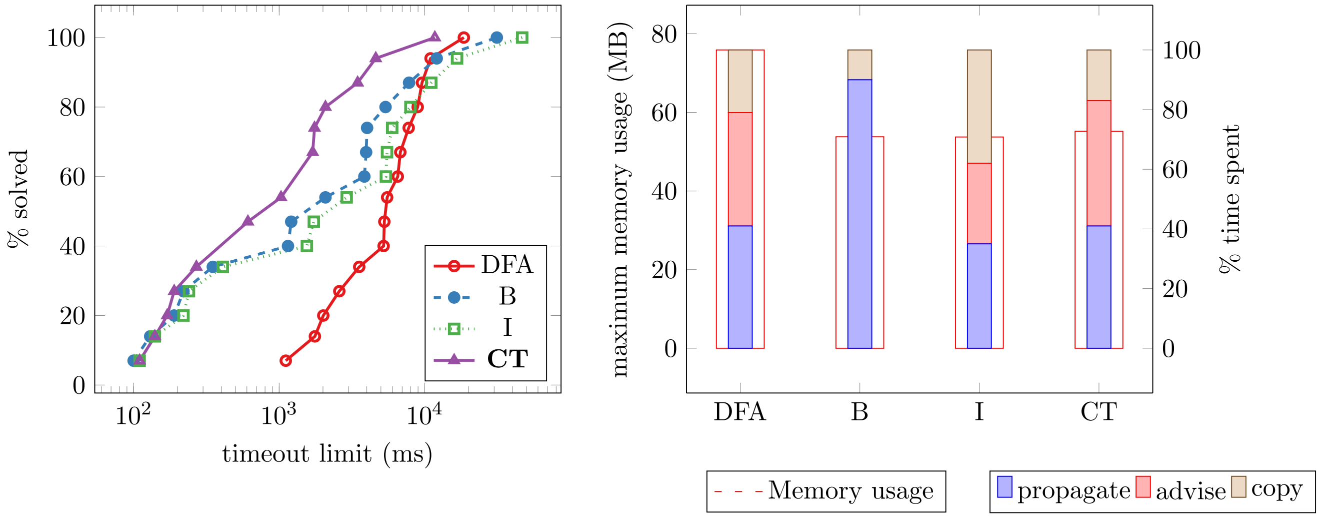

Figure 38: **TSP 20.**

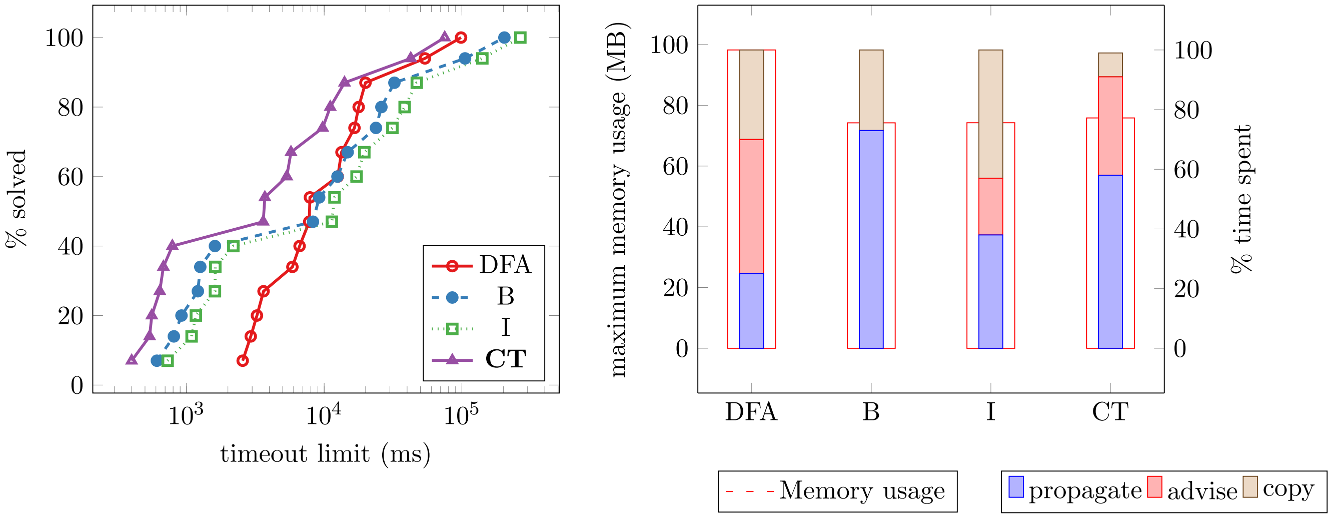

Figure 39: **TSP 25.**

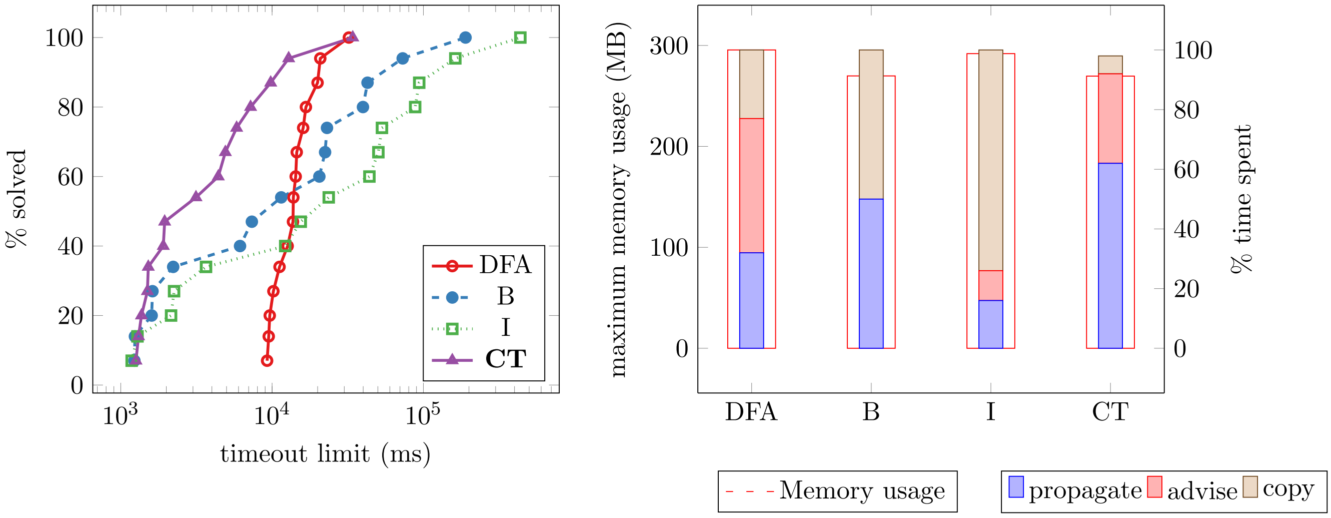

Figure 40: **TSP Quat 20.**