Implementation and Evaluation of a Compact-Table Propagator in Gecode

Linnea Ingmar

22nd June 2017

Abstract

In constraint programming, which is a programming paradigm for solving combinatorial (optimisation) problems, relations among variables are expressed using constraints — one such useful constraint is Table, which expresses the possible combinations that the variables can take as a set of tuples. All constraints have a propagation algorithm, whose purpose is to implement the reasoning for a particular constraint. Gecode [7], which is an open-source constraint-programming solver written in C++ with state-of-the-art performance, currently has two propagation algorithms for the Table constraint, and one for a closely related constraint called Regular. In this bachelor dissertation, a new propagation algorithm for the Table constraint, called compact-table, first proposed in [6], is for the first time designed and optimised for, and implemented and evaluated in Gecode. The implementation is evaluated on a large variety of benchmarks, and the results indicate that compact-table outperforms the existing propagators for Table in Gecode, as well as the propagator for the Regular constraint. These results suggest that compact-table should be included in Gecode.

Acknowledgements

I wish to express my gratitude towards the following people, and state their indispensable contributions to this work:

My supervisor Mats Carlsson, without whom I would — in several respects — never have conducted this work, and who, if supervising were an olympic sport, would be at least a twofold gold medalist in dedication and feedback loop speed, and will probably have a weekend free from work any year soon.

Pierre Flener, reviewer of this work, who has so far kept me busy for at least 533.33 hours, or 20 credits, with inspiring lectures and assignments, during which I achieved a large amount of the prerequisite for this work, including Oxford English grammar rules and what to say and what not to say about NP-completeness in a bar (serving chocolate milk), and who seems to believe more in students' potentials than they do themselves.

Christian Schulte, who gave me useful pieces of advice in an early stage of the work and who has kindly answered my questions and helped me with issues throughout the process.

Johan Öfverstedt, with whom I have had many interesting discussions, and who increased my skills of profiling and optimising C++ code, and provided useful comments on this report and some much-needed encouragement.

Albin Stjerna, who has carefully read every character of this report not only once but n times, where n is an integer larger than 3, and whose reminders of when to stop working increased the quality of this work by a factor q, and my well-being with a factor w, where q and w are constants larger than 1. More experiments are needed to determine these values with a larger accuracy.

Contents

1 Introduction **4**
1.1 Goal . . . 4
1.2 Contributions . . . 4

2 Background **5**
2.1 Constraint Programming . . . 5
2.2 Propagation and Propagators . . . 7
2.3 Gecode . . . 9
2.4 The Table Constraint . . . 10
2.5 The Compact-Table Algorithm . . . 10
2.6 Reversible Sparse Bit-Sets . . . 10

3 Algorithms **11**
3.1 Compressed Sparse Bit-Sets . . . 11
3.1.1 Fields . . . 13
3.1.2 Methods . . . 13
3.2 The Compact-Table Algorithm . . . 15
3.2.1 Pseudo Code . . . 15
3.2.2 Proof of properties for CT . . . 20

4 Implementation **22**

5 Evaluation **23**
5.1 Comparing Different Versions of CT . . . 24
5.1.1 Evaluation Setup . . . 24
5.1.2 Results . . . 24
5.1.3 Discussion . . . 25
5.2 Comparing CT against Existing Propagators . . . 25
5.2.1 Evaluation Setup . . . 30
5.2.2 Results . . . 30
5.2.3 Discussion . . . 31

6 Conclusions and Future Work **31**

A Plots from Comparison of Different Versions of CT **34**

B Plots from Comparison of CT against Existing Propagators **36**

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 [6], called Compact-Table (CT). The results in [6] 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* [7] is a popular CP solver written in the C++ programming language that combines state-of-the-art performance with modularity and extensibility, using copying during branching. 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 [6] have for the first time been optimised for a copying CP solver such as Gecode; this is described 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 the solver; 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. 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.

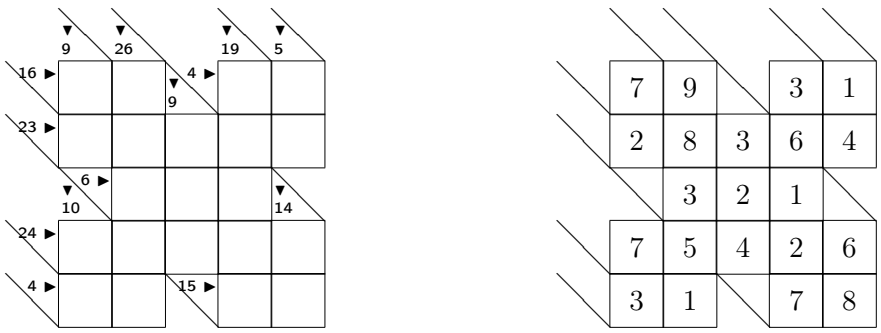

Figure 1: A Kakuro puzzle[2](left) and its solution (right).

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) 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, 14, 15]. We start by defining *constraints*, which are relations among variables.

Definition 1 (Constraints). 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 $\mathrm{dom}(v_i)$, its *domain size* is $|\mathrm{dom}(v_i)|$ and its *domain width* is $(\max(\mathrm{dom}(v_i)) - \min(\mathrm{dom}(v_i)) + 1)$.

- A *constraint* c on a subsequence of V is a relation, denoted by $\mathrm{rel}(c)$. The associated variables are denoted by $\mathrm{vars}(c)$, and the *arity* of c is $|\mathrm{vars}(c)|$. If the arity of c is r, then the relation $\mathrm{rel}(c)$ contains the set of r-tuples that are allowed for $\mathrm{vars}(c)$, and we call those r-tuples *solutions* to the constraint c.

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

- Let τ be an r-tuple of values associated with an r-ary constraint c on a subsequence of V. We denote the value corresponding to a variable v by $\tau[v]$. The tuple τ is *valid* for $\text{vars}(c)$ if and only if each value of τ is in the domain of the corresponding variable: $\forall v \in \text{vars}(c) : \tau[v] \in \text{dom}(v)$. The tuple τ is a *support on* c if and only if τ is valid for $\text{vars}(c)$ and τ is a solution to c, that is, τ is a member of $\text{rel}(c)$. For a variable $v \in \text{vars}(c)$ such that the the value $a \in \text{dom}(v)$, the tuple τ is a *support for* (v, a) on c if and only if τ is a support on c and $\tau[v] = a$. If such a tuple τ exists, then (v, 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: $V = v_1, \ldots, v_n$ is a finite sequence of variables, $D = D_1, \ldots, D_n$ is a finite sequence of domains for the respective variables, and $C = \{c_1, \ldots, c_m\}$ is a finite set of constraints, each on a subsequence of V.

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

Definition 3 (Stores). A *store* s is a function, mapping a finite sequence of variables $V = v_1, \ldots, v_n$ to a finite sequence 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 such that $s(v_i) = \emptyset$ is said to have a *domain wipe-out* under s.
- A variable $v_i \in V$ is *fixed*, or *assigned*, by a store s if and only if $|s(v_i)| = 1$.
- Let c be an r-ary constraint on a subsequence V_r of V, and assume without loss of generality that V_r contains the first r variables of V. A store s is an *assignment store* for c if all variables in V_r are fixed by s. A store s is a *solution store* to c if and only if s is an assignment store for c such that the r-tuple that the values of the variables form is a solution to c: $\forall i \in \{1, \ldots, r\} : s(v_i) = \{a_i\}$, and $\langle a_1, \ldots, a_r\rangle$ 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 [14] and [15], where we let *store* be the set of all stores.

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

$$p : store \rightarrow store$$

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) gives 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, then 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. The following defines three commonly used consistencies: **range consistency**, **bounds consistency**, and **domain consistency**, based on [2, 6].

Definition 5 (**Range consistency**)**.** Given an integer variable v, its *range* is the closed interval $[\min(\mathrm{dom}(v)), \max(\mathrm{dom}(v))]$. A constraint c is *range consistent* on a store s if and only if for all variables that are fixed under s, there exist values in the ranges of all the other variables in $\mathrm{vars}(c)$ such that the values form a solution to c.

Definition 6 (**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: $\forall v \in \mathrm{vars}(c)$, $(v, \min(\mathrm{dom}(v)))$ and $(v, \max(\mathrm{dom}(v)))$ have a support on c.

Definition 7 (**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: $\forall v \in \mathrm{vars}(c), \forall a \in \mathrm{dom}(v)$: (v, a) has a support on c.

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 enforcing range- or bounds consistency, but might be more costly.

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

2.3 Gecode

Gecode [7] (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 [15].

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 domain 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, its associated 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 8 (**Table constraints**). A (positive[3]) *table constraint* c is a constraint such that $\text{rel}(c)$ 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 [14].

As an example of use case, the Table constraint has proved to be useful for pre-solving sub-problems in constraint models [5].

In Gecode, the Table constraint and another constraint called Regular, which constrains a sequence of variables to form a word of a regular language, are both called Extensional. Gecode provides one propagator for Regular, based on [12], and two propagators for Table; one which is based on [3], 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 the CP solver OR-tools (Google Optimization Tools) [11], where it outperforms all previously known algorithms, and was first described in [6]. 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`, that stores the indices of the currently valid tuples from the input table. Also, for each variable-value pair, a bit-set mask is computed and stored in an array `supports`; each bit-set mask stores the indices of the tuple 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, first described in [6], is a sparse-set data structure [4, 8] that is a main data structure in the CT algorithm in [6]. The data structure stores a set of integers from the

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

range $0 \ldots n-1$, where n is a given number. Initially, all elements from this range are present, and the set can only become sparser — there are operations for removing values but not for adding values. Operations are performed only on non-zero words in the bit-set, which makes it efficient to perform bit-wise operations with other bit-sets (such as intersection and union), even when the set of values is sparse; hence the name.

Some CP solvers, among them OR-tools, 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. For such CP solvers, every word of a reversible sparse bit-set must be reversible, and so every time a word is modified, a trail entry may have to be pushed, so that the word can be reset to its previous contents upon backtracking.

3 Algorithms

This section presents the algorithms that are used in the implementation of the CT propagator within this project. In the following, we call *int* the data structure that represents integers. For an array a we let $a[0]$ denote the first element (thus indexing starts from 0), and a.length() the number of cells. By the notation 0^{64} we mean a 64-bit int that has all its bits set to 0.

Parts of the pseudo code and its description in this section are very similar to the corresponding content in [6], as the algorithms are based on this paper.

3.1 Compressed Sparse Bit-Sets

This section describes a new data structure called compressed sparse bit-set that is a main data structure in the CT algorithm implemented within this project. Compressed sparse bit-sets are similar to reversible sparse bit-sets described in Section 2.6; the differences are:

- Compressed sparse bit-sets are not reversible, that is, they cannot be restored to a previous state.
- Compressed sparse bit-sets have a denser representation of the active bits than reversible sparse bit-sets have — the non-zero words form a contiguous memory block.

These differences reflect the fact that a compressed sparse bit-set is a data structure customised for a copy-based solver such as Gecode, in contrast to reversible sparse bit-sets that are more suited for a trail-based solver such as OR-tools.

Algorithm 1 shows pseudo code for the class `CompressedSparseBitSet` implementing compressed sparse bit-sets. The rest of this section describes its fields and methods in detail.

```
Class CompressedSparseBitSet

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

Method initSparseBitSet(nbits: int)
  p ← ⌈nbits/64⌉
  words ← array of 64-bit int of length p, first nbits set to 1
  mask ← array of 64-bit int of length p, all bits set to 0
  index ← [0, ..., p − 1]
  limit ← p − 1

Method isEmpty() : Boolean
  return limit = −1

Method clearMask()
  for i ← 0 to limit do
    mask[i] ← 0^64

Method flipMask()
  for i ← 0 to limit do
    mask[i] ← ∼ mask[i]                                      // bitwise NOT

Method addToMask(m: array of 64-bit int)
  for i ← 0 to limit do
    offset ← index[i]
    mask[i] ← mask[i] | m[offset]                            // bitwise OR

Method intersectWithMask()
  for i ← limit downto 0 do
    w ← words[i] & mask[i]                                   // bitwise AND
    if w ≠ words[i] then
      words[i] ← w
      if w = 0^64 then
        words[i] ← words[limit]
        words[limit] ← w
        index[i] ← index[limit]
        index[limit] ← i
        limit ← limit − 1

Method intersectIndex(m: array of 64-bit int) : int
  for i ← 0 to limit do
    offset ← index[i]
    if words[i] & m[offset] ≠ 0^64 then
      return i
  return −1
```

Algorithm 1: Pseudo code for the class `CompressedSparseBitSet`.

3.1.1 Fields

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

- `words` is an array storing a permutation of p 64-bit words: $\{w_0, w_1, \ldots, w_{p-1}\}$, each word w_i representing 64 elements in the set. Initially, $\texttt{words}[i] = w_i$ for all i. The array `words` 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 initialisation, 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 continuously re-ordered so that all the non-zero words are located at indices less than or equal to `limit` (see below), and all the words that consist of only zeros are located at positions strictly greater than `limit`.

- `index` 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`, `index` maps its *current* index to its *original* index: $\texttt{words}[i] = w_{\texttt{index}[i]}$ for all i.

- `limit` is the index of `index` and `words` corresponding to the last non-zero word in `words`. Thus it is one less 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:

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

3.1.2 Methods

We now describe the methods in the class `CompressedSparseBitSet` in Algorithm 1.

- initSparseBitSet() in lines 6–11 initialises a compressed sparse bit-set-object. It takes the number of elements (*nbits*) 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 14–16 clears the temporary mask. This means setting to 0 all words of `mask` corresponding to non-zero words of `words`.

- flipMask() in lines 17–19 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 64-bit int). Once again, this operation is only applied to indices corresponding to non-zero words in `words`.

	w_0	w_1	w_2	w_3	w_4
`words`	1111	1111	1111	1111	1100
`mask`	0000	0110	0000	1110	1100
`index`	0	1	2	3	4

`limit` = 4

(a) State before applying intersectWithMask().

	w_3	w_1	w_4	w_0	w_2
`words`	1110	0110	1100	0000	0000
`index`	3	1	4	0	2

`limit` = 2

(b) Result of intersectWithMask() with mask in (a).

	0	1	2	3	4
`mask`	0000	0000	0000	1110	1100
`b`	1111	0111	1101	0001	0100
`mask`	0001	0111	0100	1110	1100
`index`	3	1	4	0	2

`limit` = 2

(c) Adding `b` to the mask.

	w_4	w_1	w_3	w_0	w_2
`words`	0100	0110	0000	0000	0000
`index`	4	1	3	0	2

`limit` = 1

(d) Result of intersectWithMask() with mask in (c).

Figure 2: Applying addToMask() and intersectWithMask() multiple times.

- 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 lines 31 and 33 because it is unnecessary to save information about the zero words in a copy-based solver such as Gecode. We keep these lines here though, as the class 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 64-bit int) 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.

Figure 2 shows an example of applying operations to a compressed sparse bit-set assuming it consists of 4-bit ints instead of 64-bit ints. We assume that initially the set contains the range 0..17, so $p = 5$ (we need 5 4-bit ints for 18 bits). In (a), `words` and `index` are still in their initial state, and `mask` has some bits set to 0 and others set to 1. In (b), we see the result of applying intersectWithMask() with the mask from (a). In the operation, all the bits in words w_0 and w_2 are set to 0, and w_0 is swapped with w_3, and w_2 is swapped with w_4. Between (b) and (c), clearMask() is assumed to have been called, so that the words up to `limit` are cleared in `mask`. In (c), the bit-set `b` is added to `mask`. Finally, (d) shows the result of applying intersectWithMask() with the mask in (c), and we see that all bits in w_3 are set to 0, and w_3 is swapped with w_4. In (d), the elements that are present are 5 and 6 (in w_1), and 17 (in w_4).

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 inputs are an initial table; a set of tuples $T_0 = \langle \tau_0, \tau_1, \ldots, \tau_{p_0-1} \rangle$ of length p_0, and the sequence of variables vars(c); the variables that are associated with c. 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 initially valid for vars(c). For a variable x, we distinguish between its *initial domain* $\underline{\text{dom}}(x)$ and its *current domain* 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 `CompressedSparseBitSet` 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 `CompressedSparseBitSet` object of initial size p, such that value i is contained (is set to 1) if and only if the ith tuple is valid:

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

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

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

  `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), 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.

- `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` has been modified since the last invocation, then FILTERDOMAINS() is called, which filters 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

```
PROCEDURE CompactTable(s : store) : ⟨StatusMessage, store⟩
if the propagator is being posted then                          // executed in a constructor
   s ← InitialiseCT(s, T_0, vars(c))
   if s = ∅ then
      return ⟨FAIL, ∅⟩
else                                                            // executed in an advisor
   foreach variable x ∈ vars whose domain has changed since last invocation do
      UpdateTable(s, x)
      if validTuples.isEmpty() then
         return ⟨FAIL, ∅⟩
   if validTuples has changed since last invocation then // executed during propagation
      s ← FilterDomains(s)
if there is at most one unassigned variable left then
   return ⟨SUBSUMED, s⟩
else
   return ⟨FIX, s⟩
```

Algorithm 2: Compact Table Propagator.

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.

In the implementation of the algorithm, InitialiseCT() is executed in the constructor of the object, UpdateTable() is executed in the advisors, and FilterDomains() is executed when the propagator is invoked for propagation; this happens after the advisors have been executed.

The procedure InitialiseCT() is described in Algorithm 3. The procedure takes the input store s, the initial table T_0, and the sequence of associated variables vars(c) as arguments.

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, then the 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 $x \in \text{vars}(c)$. 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, then 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 22 increases the counter.

Lines 23–27 remove values that are not supported by any tuple in the initial valid table. The procedure returns the empty store in case a variable has a domain wipe-out.

Line 28 initialises `validTuples` as a `CompressedSparseBitSet` 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.

The procedure UpdateTable() in Algorithm 4 filters out (indices of) tuples that have ceased to be supports for the input variable x. Line 1 clears the temporary mask. Lines 2–3

PROCEDURE INITIALISECT(s: store, T_0: set of tuples, $vars(c)$: seq. of variables) : store

1: **foreach** $x \in s$ **do**
2: $\quad R \leftarrow \{a \in s(x) : a > T_0.\text{max}() \ \vee \ a < T_0.\text{min}()\}$
3: $\quad s \leftarrow s[x \mapsto s(x) \setminus R]$
4: $\quad$ **if** $s(x) = \emptyset$ **then**
5: $\quad\quad$ **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: $\quad supported \leftarrow$ `true`
14: $\quad$ **foreach** $x \in$ `vars` **do**
15: $\quad\quad$ **if** $t[x] \notin s(x)$ **then**
16: $\quad\quad\quad supported \leftarrow$ `false`
17: $\quad\quad\quad$ **break** // Exit loop
18: $\quad$ **if** $supported$ **then**
19: $\quad\quad$ **foreach** $x \in$ `vars` **do**
20: $\quad\quad\quad$ `supports`$[x, t[x]][nsupports] \leftarrow 1$
21: $\quad\quad\quad$ `residues`$[x, t[x]] \leftarrow \lfloor \frac{nsupports}{64} \rfloor$ // Index for the support in `validTuples`
22: $\quad\quad\quad nsupports \leftarrow nsupports + 1$
23: **foreach** $x \in$ `vars` **do**
24: $\quad R \leftarrow \{a \in s(x) : \texttt{supports}[x, a] = \emptyset\}$
25: $\quad s \leftarrow s[x \mapsto s(x) \setminus R]$
26: $\quad$ **if** $s(x) = \emptyset$ **then**
27: $\quad\quad$ **return** $\emptyset$
28: `validTuples` $\leftarrow$ `CompressedSparseBitSet` with $nsupports$ bits set to 1
29: **return** s

Algorithm 3: Initialising the CT propagator.

store the union of the set of valid tuples for each value $a \in \text{dom}(x)$ in the 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 $x \in \text{vars}(c)$ 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.

We note that it is only necessary to consider a variable $x \in$ `vars` that is not assigned, because we will never filter out values from the domain of an assigned variable. To see this,

```
PROCEDURE UpdateTable(s: store, x: variable)
validTuples.clearMask()
foreach a ∈ 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.

```
PROCEDURE FilterDomains(s : store) : store
1: foreach x ∈ vars such that |s(x)| > 1 do
2:   foreach a ∈ s(x) do
3:     r ← residues[x, a]
4:     if words[r] & supports[x, a][index[r]] = 0 then
5:       r ← validTuples.intersectIndex(supports[x, a])
6:       if r ≠ −1 then
7:         residues[x, a] ← r
8:       else
9:         s ← s[x ↦ s(x) \ {a}]
10: return s
```

Algorithm 5: Filtering variable domains, enforcing domain consistency.

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 never be upon invocation of FilterDomains(), because then CompactTable() would have reported failure before FilterDomains() is called.

In lines 3–4 we check if the word at the cached index r still contains a support for (x, a). If it does 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 $\mathrm{dom}(x)$ 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(), 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`, we can make sure that `residues`$[x, a]$ is not just *any* index in `words` where a support for (x, a) was found the last time it was sought, but the *highest* such index. This means that `residues`$[x, a]$ will be an *upper bound* on the indices that contain supports for (x, a), because a word w_i residing at an index j in `words` can only move to an index smaller than j. 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 can be maintained by executing the loop in intersectIndex() from highest index to lowest index instead of the other way around. The benefit of this invariant is that we sometimes can decrease the number of iterations in intersectIndex(); more specifically, when reaching line 5 in FilterDomains(), it follows from the introduced invariant that there is no support for (x, a) at indices greater than or equal to `residues`$[x, a]$; thus we

```
PROCEDURE UpdateTable(s: store, x: variable)
validTuples.clearMask()
if Δ_x is available ∧ |Δ_x| < |s(x)| then
    foreach a ∈ Δ_x do
        validTuples.addToMask(supports[x, a])
    validTuples.flipMask()
else
    foreach a ∈ s(x) do
        validTuples.addToMask(supports[x, a])
validTuples.intersectWithMask()
```

Algorithm 6: Updating the current table using delta information.

```
if validTuples has changed since last invocation then
    if (index ← validTuples.indexOfFixed()) ≠ −1 then
        return ⟨SUBSUMED, s[x ↦ T[index][x] : x ∈ vars]⟩
    else
        s ← FilterDomains(s)
```

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

can start the iteration in intersectIndex() at index $\min(\texttt{limit}, \texttt{residues}[x, a] - 1)$. By letting intersectIndex() take an extra argument that defines the loop limit, this value can be passed to the method in line 5 in FilterDomains().

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

CT(Δ) — *Using delta information in* UpdateTable(). For a variable x, the set Δ_x contains the 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.

CT(T) — *Fixing the domains when only one valid tuple left.* If there is only one valid tuple 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.

For a word `w`, there is exactly one set bit 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 set bit, else -1. The

```
Method indexOfFixed() : int
  index_of_fixed ← −1
  if limit = 0 then
    w ← words[0]
    if (w & (w − 1)) = 0^64 then                    // Exactly one set bit
      offset ← index[0]
      index_of_fixed ← offset · 64 + MSB(w)
  return index_of_fixed
```

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

method indexOfFixed() is added to the class `CompressedSparseBitSet` and assumes access to built-in MSB which returns the index of the most significant bit of a given int.

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, $\mathrm{CT}(\mathrm{CT}(s)) = \mathrm{CT}(s)$ for any store s.

Suppose $\mathrm{CT}(\mathrm{CT}(s)) \neq \mathrm{CT}(s)$ for a store s. Since CT is monotonic and decreasing, we must have $\mathrm{CT}(\mathrm{CT}(s)) \prec \mathrm{CT}(s)$, that is CT must prune at least one value a from the domain of a variable x from the store $\mathrm{CT}(s)$. Now, by (3.3), there must exist at least one tuple τ_i that is a support for (x, a) under the store $\mathrm{CT}(s)$: $\exists i : i \in \texttt{validTuples} \ \wedge \ \tau_i[x] = a$. After UPDATE-TABLE() is performed on $\mathrm{CT}(s)$, we still have $i \in \texttt{validTuples}$, because τ_i is still valid in $\mathrm{CT}(s)$. 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 $\mathrm{CT}(\mathrm{CT}(s)) = \mathrm{CT}(s)$. □

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 for c, 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 $\tau = \langle s(x_1), \ldots, s(x_n) \rangle \notin \mathrm{rel}(c)$.

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, that is, always returns a fixpoint, trivially it will never be the case that CT signals fixpoint without having computed a fixpoint, thus 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 $a \in \mathrm{dom}(x)$. Hence, CT is indeed subsumed by s when it signals subsumption, so CT is subsumption honest. □

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

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 is available on a public repository, following this URL: `https://github.com/lingmar/CT-Gecode`.

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 closely follows the pseudo code in Section 3.2.1. 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 usage. 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 (e.g. the associated entries of `supports` and `residues`). 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 and some notable observations were made. This 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 limitations. Another implementation detail is that during propagation, the implementation in OR-tools 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 by CT(B) in the evaluation of different variant of CT (Section 5).

Memory usage and copying. Since Gecode is a copy-based solver, the state of the propagator is copied during branching. The array `supports` consists of static data (only computed once), so 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 contrast to the rest of the data structures, which are allocated in a memory area specific to the current node, because they change dynamically and therefore need to be copied.

For the compressed sparse bit-set, only the words up to `limit` inclusive are copied for `words` and `index`, since these are the only words that correspond to non-zero words. Moreover, the lines 31 and 33 in intersectWithMask() in Algorithm 1 are not executed, because the words that consist of only zeros do not need to be saved.

Furthermore, not all entries in `residues` need to be copied, because entries corresponding to assigned variables or removed values will not be used again. However, iterating over a domain of a variable is computationally expensive, while finding the minimum and maximum value of the domain is cheap, so in the current implementation all entries corresponding to values in the range $[\min(\mathrm{dom}(x)), \max(\mathrm{dom}(x))]$ for unassigned variables are copied.

Profiling. Profiling tools were used to locate the parts of the implementation 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 parts of the code that is executed very often, by moving out memory accesses from loops. The profiling shows that most of the execution time is spent in bit-wise operations. This suggests that further optimisations could be achieved by investigating methods to parallelise the bit-wise operations, for example by exploring whether it would be possible to use 128-bit operations, which some computer architectures support. Moreover, it is probable that the execution speed could be increased if the number of memory accesses could be decreased further, by revising the implementation of the data structures and trying to decrease the levels of indirection, because following pointers is costly.

Using delta information. In the version CT(Δ), which uses the set of values Δ_x that has been removed since the previous invocation, 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

We now present the evaluation of the implementation of the CT propagator, in which the performance of different versions of CT is compared, and the winning variant is compared against

the existing propagators for the TABLE constraint in Gecode, as well as with the propagator for the REGULAR constraint.

The benchmarks consist of 1507 CSP instances, divided into 30 groups, involving TABLE constraints only. The groups of benchmarks and their characteristics are presented in Table 1. This set of benchmarks was also used in the experiments in [6]. These instances were chosen because they contain a large variety of instances, and the fact that they were used in [6] 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, it is rather an advantage: if other constraints than TABLE were present, then a smaller amount of the total runtime would be spent in the propagators that are compared, which would give a weaker performance difference. Note however that if the involved propagators would not have had the same consistency — they are all domain consistent — then other constraints would have been desired, because then it would be interesting to compare not only the runtimes, but also the sizes of the search trees in the presence of other constraints.

All the benchmark instances were originally written in an XML format called XCSP2.1 [13], and before being used in this project they were translated into a modelling language called MiniZinc [10] using a tool [9] for translating XCSP2.1 instances into MiniZinc. Out of the 1621 instances that were used in [6], only 1507 could be used due to parse errors in the translation process.

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.

A timeout of 1000 seconds was used throughout the experiments. An instance was filtered out if it i) could be solved within 1 s for all propagators (as there is too much measurements noise on easier instances), or ii) caused a memory-out for at least one of the propagators (as the memory limit varies between different hardware and software platforms).

5.1 Comparing Different Versions of CT

In this section the comparison of difference versions of CT presented and discussed.

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. 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 the previous invocation of the propagator, 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. Moreover, the second best variant seems to be the basic version (CT), while it is hard to say which one of CT(T) and CT(B) is best; sometimes the first performs better than the second and sometimes the second performs better than the first. On *AIM-50*, which only contains instances with 0/1 variables, the performance of CT, CT(Δ), and CT(B) is similar, 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 propagators: one where the extension is given as a DFA (deterministic finite automaton), a non-incremental memory-efficient one where the extension is given as a tuple set, and an incremental time-efficient one where the extension is also given as a tuple set:

DFA This is based on [12].

B — Basic positive tuple set propagator This is based on [3].

The propagator state has the following fields:

- array of variables: X
- tuple set: T
- $L[x, n]$ 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.

Algorithm 9 shows the basic tuple set propagation algorithm. $S[x]$ signals whether or not a support has been found for the variable x, and is initialised to $\perp$ for each variable in lines 1 and 2. The loop in lines 3–17 will then try to find a support for each variable-value pair. The values that no longer have a support are collected in the set N, whose values are removed from the domain of x in line 15. If a variable has a domain wipe-out, then the empty store is returned in line 17, otherwise the resulting store is returned in line 18.

```
PROCEDURE EXTENSIONAL(s : store) : store
foreach x ∈ X do
    S[x] ← ⊥
foreach x ∈ X do
    N ← ∅
    foreach n ∈ dom(x) do
        if S[x] = ⊥ then
            ℓ ← L[x, n]
            while ℓ ≠ ⊥ ∧ ∃y ∈ X : ℓ[y] ∉ dom(y)
                ℓ ← L[x, n] ← next tuple for ⟨x, n⟩
            if ℓ = ⊥ then
                N ← N ∪ {n}
            else
                foreach y ∈ X where y > x do
                    S[y] ← ℓ[y]
    s ← s[x ↦ dom(x) \ {N}]
    if dom(x) = ∅ then
        return ∅
return s
```

Algorithm 9: Basic positive tuple set propagator.

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 singletons
AIM-50	23	3, a few 2	3 to 7	0..1
AIM-100	23	3, a few 2	3 to 7	0..1
AIM-200	22	3, a few 2	3 to 7	0..1
BDD Large	35	15	approx. 7000	0..1
BDD Small	35	18	approx. 58000	0..1
Crosswords WordsVG	65	2 to 20	3 to 7360	0..25
Crosswords LexVG	63	5 to 20	49 to 7360	0..25
Crosswords Wordspuzzle	22	2 to 13	1 to 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 to 9	2 to 362880	1..9
Kakuro Medium	192	2 to 9	2 to 362880	1..9
Kakuro Hard	187	2 to 9	2 to 362880	0..9
Langford 2	20	2	1 to 1722	Vary from 0..3 to 0..41
Langford 3	16	2	3 to 2550	Vary from 0..5 to 0..50
Langford 4	16	2	5 to 2652	Vary from 0..7 to 0..51
MDD 05	25	7	approx. 29000 to approx. 57000	0..4
MDD 07	9	7	approx. 40000	0..4
MDD 09	10	7	approx. 40000	0..4
Mod Renault	50	2 to 10	3 to 48721	Vary from 0..1 to 0..41
Nonograms	180	2	1 to 1562275	Vary from 1..15 to 1..980
Pigeons Plus	40	2 to 10	10 to 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 20	15	2, a few 3	20 to 23436, and a few 1	Vary from singleton to 0..1000
TSP 25	15	2, a few 3	22 to 23653, and a few 1	Vary from singleton to 0..1000
TSP Quat 20	15	2, a few 3	380 to 23436, and a few 1	Vary from singleton to 0..1000

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, "S" standing for "support", whose support data needs restoring. Initially empty.
- W_{R} is a stack of literals, "R" standing for "remove", that are longer supported, and whose domain therefore needs updating and whose support data need clearing. Initially empty.

Algorithm 10 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 the previous invocation of the propagator. 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(s : store) : store
 1: if the propagator is being posted then                    // executed in a constructor
 2:   foreach x ∈ X do
 3:     foreach n ∈ dom(x) do
 4:       FindSupport(⟨x, n⟩)
 5: else                                                       // executed in an advisor
 6:   foreach ⟨x, n⟩ that has been removed since the previous invocation do
 7:     foreach t ∈ S[⟨x, n⟩] do
 8:       RemoveSupport(t, ⟨x, n⟩)
 9: while W_R ≠ ∅ ∨ W_S ≠ ∅                                    // executed during propagation
10:   foreach ⟨x, n⟩ ∈ W_R do
11:     s ← s[x ↦ dom(x) \ {n}]
12:     if dom(x) = ∅ then
13:       return ∅
14:   W_R ← ∅
15:   foreach ⟨x, n⟩ ∈ W_S where n ∈ dom(x) do
16:     FindSupport(⟨x, n⟩)
17:   W_S ← ∅
18: return s
```

Algorithm 10: Incremental positive tuple set propagator.

Algorithm 11 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($\langle x, n\rangle$)

1: $\ell \leftarrow L[\langle x, n\rangle]$
2: **while** $\ell \neq \bot \wedge \exists y \in X : \ell[y] \notin \mathrm{dom}(y)$
3: $\quad \ell \leftarrow L[\langle x, n\rangle] \leftarrow$ next tuple for $\langle x, n\rangle$
4: **if** $\ell = \bot$ **then**
5: $\quad W_{\mathrm{R}} \leftarrow W_{\mathrm{R}} \cup \{\langle x, n\rangle\}$
6: **else**
7: $\quad$ **foreach** $y \in X$ **do**
8: $\quad\quad S[\langle y, \ell[y]\rangle] \leftarrow S[\langle y, \ell[y]\rangle] \cup \{\ell\}$

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

Algorithm 12 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($\ell, \langle x, n\rangle$)

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: $\quad\quad W_{\mathrm{S}} \leftarrow W_{\mathrm{S}} \cup \{\langle y, \ell[y]\rangle\}$

Algorithm 12: Clear support data for unsupported literal $\langle x, n\rangle$. Note: n is actually not used here.

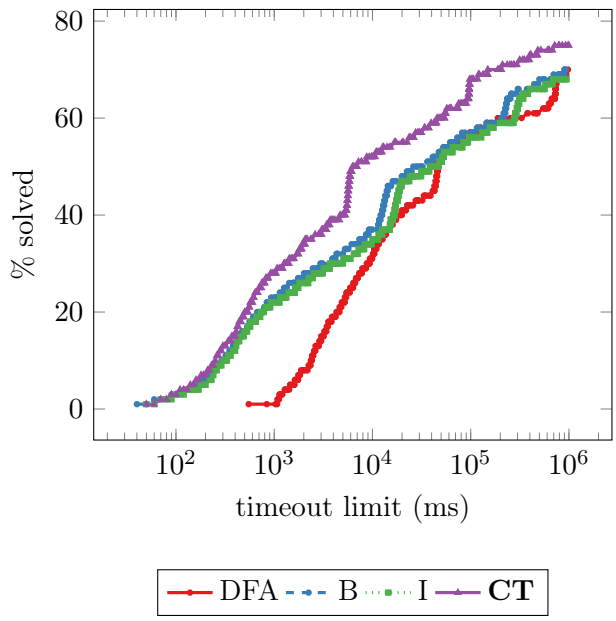

Figure 3: Percentage of instances solved as a function of time for the DFA, B, I, and CT propagators.

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 Compact-table propagator, version CT(Δ).

DFA Layered graph (DFA) propagator, based on [12].

B Basic positive tuple set propagator, based on [3].

I Incremental positive tuple set propagator.

5.2.2 Results

The final set of instances used in the results consists of 655 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 3 shows the percentage of instances solved as a function of timeout limit in ms for these 655 instances. Within the timeout of 1000 s, CT could solve the highest number of instances (75 %), followed by B (70 %), DFA (70 %), and I (68 %).

Among the 852 instances that were filtered out, 212 were filtered out because of memory-outs. Among these 212 instances, DFA ran out of memory on all of them, and CT, B, and I all ran out of memory on the same 36 of them.

The plots from each individual group of benchmarks are presented in Appendix B, except for the groups *BDD Small*, where DFA ran out of memory on all instances, and *MDD 07* and *MDD 09*, where all propagators timed out on all instances.

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 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 claimed (in p. 73 in [15]) 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 no performance gain have constraints with low arities (see *AIM-**, *Dubois*, *Geom*, and *Langford **), though there are exceptions to this (see *Pigeons Plus* and *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 impacts the performance gain.

Domain size It is hard to draw any conclusion 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-** and *Dubois*) and some have large performance gains (see *MDD 05* and *BDD Large*). The same is true for the groups with larger domain sizes; some have modest performance gains (see *Nonograms* and *Kakuro **), while some have larger performance gains (see *Rands JC** and *Crosswords **).

Memory usage. It can be seen that CT, B, and I have about the same maximum memory usage while DFA consistently has a higher maximum memory usage. This is consistent with the fact that DFA ran out of memory on the highest number of instances.

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 propagation 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 [16].

References

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

[2] C. Bessière. Constraint propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter I.3, pages 29–83. Elsevier, 2006.

[3] 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.

[4] P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2:59–69, 1993.

[5] 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`.

[6] 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 *LNCS*, pages 207–223. Springer, 2016.

[7] Gecode Team. Gecode: A generic constraint development environment, 2017. `http://www.gecode.org`.

[8] V. le Clément de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre. Sparse-sets for domain implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013. Available at `https://www.info.ucl.ac.be/~pschaus/assets/publi/trics13-sparsets.pdf`.

[9] J. Mauro and R. Amadini. Feature extractor for MiniZinc and XCSP models, 2016. Available at `https://github.com/CP-Unibo/mzn2feat`.

[10] 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`.

[11] OR-tools team. OR-tools: Google optimization tools, 2017. `https://developers.google.com/optimization/`.

[12] 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.

[13] O. Roussel and C. Lecoutre. XML representation of constraint networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009.

[14] 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.

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

[16] H. Verhaeghe, C. Lecoutre, and P. Schaus. Extending compact-table to negative and short tables. In S. P. Singh and S. Markovitch, editors, *AAAI 2017*, pages 3951–3957. AAAI Press, 2017.

A Plots from Comparison of Different Versions of CT

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

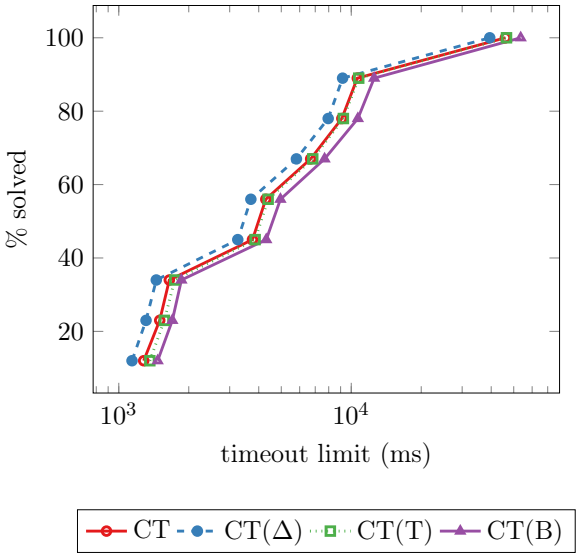

Figure 4: **Rands JC2500.**

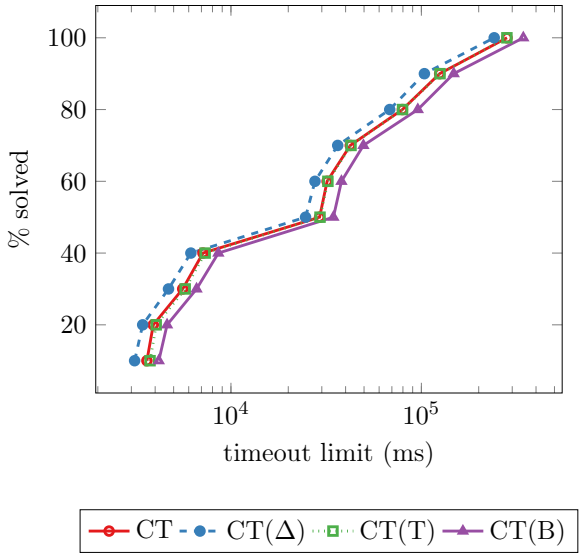

Figure 5: **Rands JC5000**.

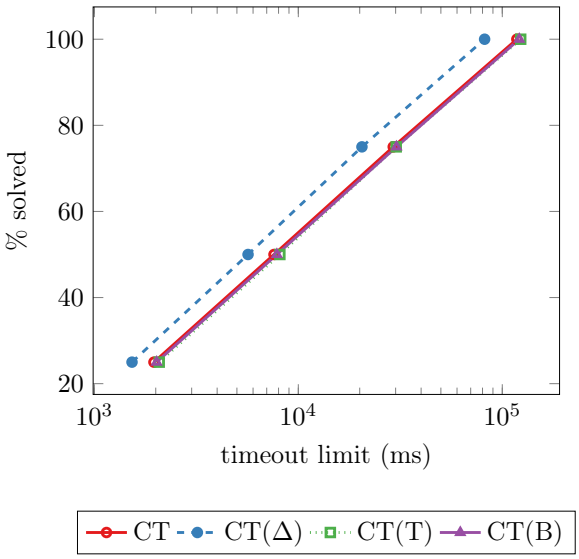

Figure 6: **Langford 4**.

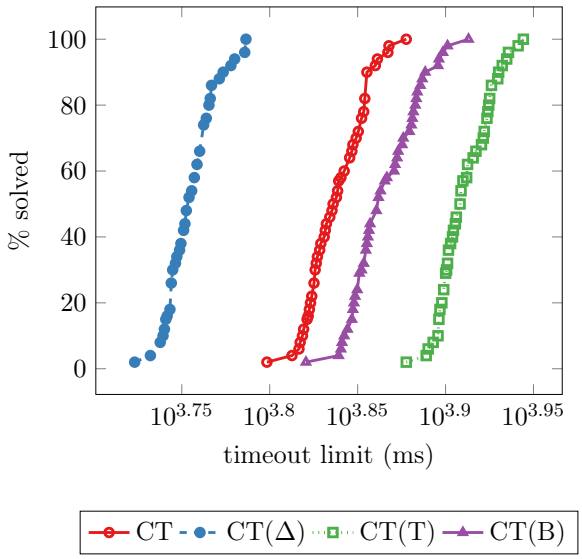

Figure 7: **A5**.

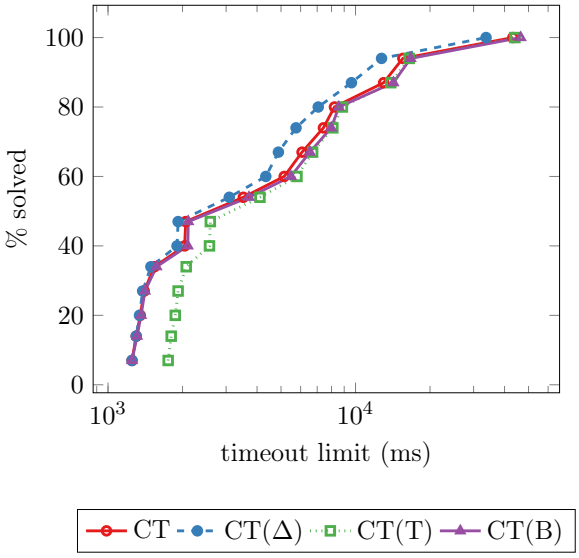

Figure 8: **TSP Quat 20**.

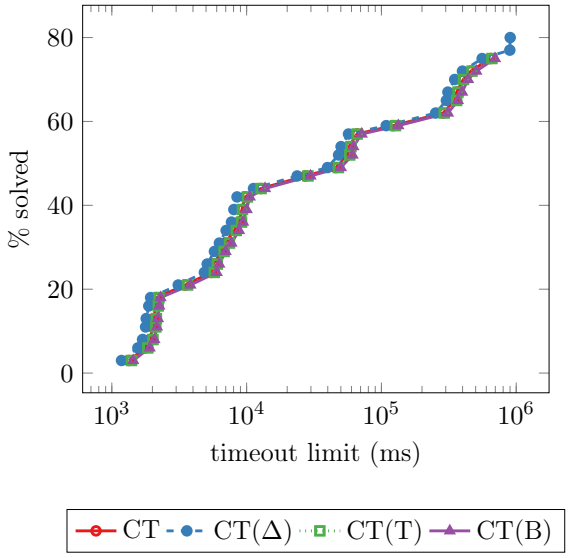

Figure 9: **Geom**.

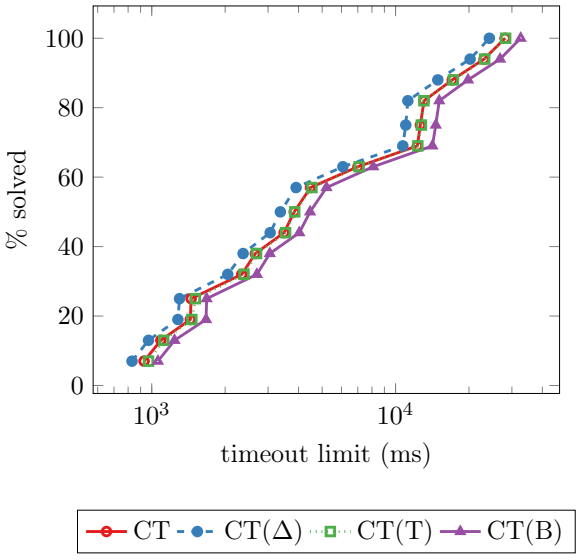

Figure 10: **Crosswords LexVG**.

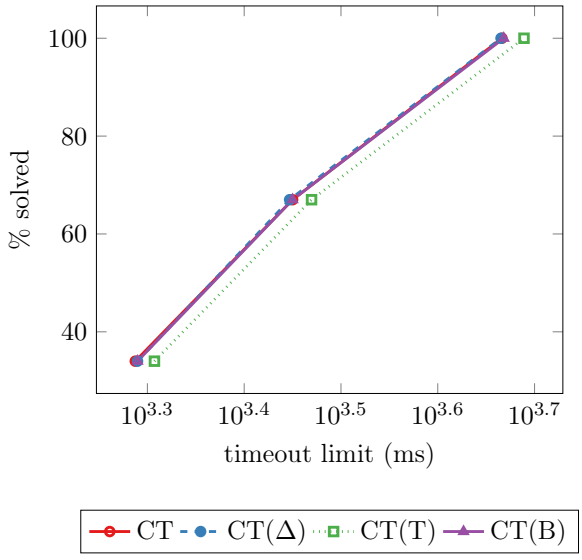

Figure 11: **AIM 50**.

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 distributed 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.

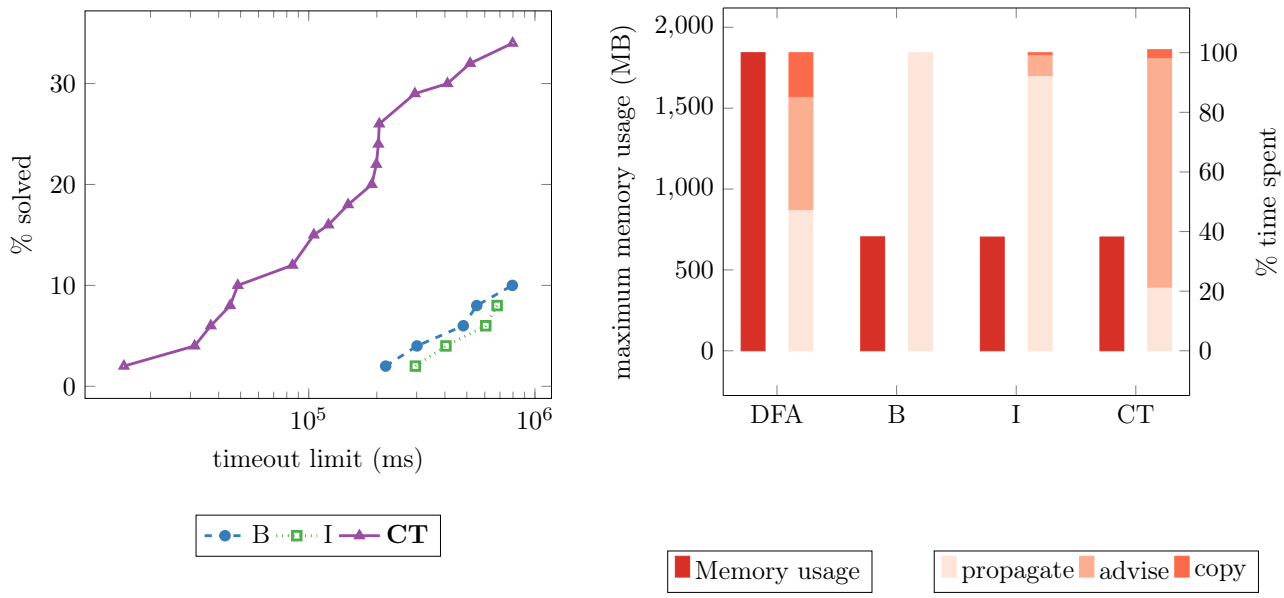

Figure 12: **A10.**

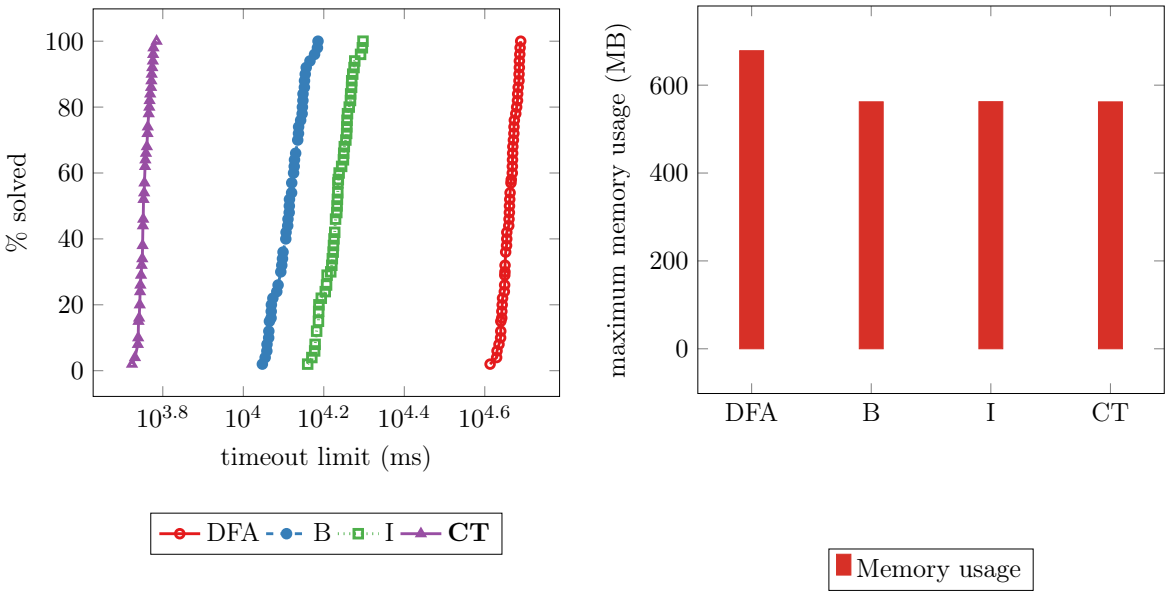

Figure 13: **A5.**

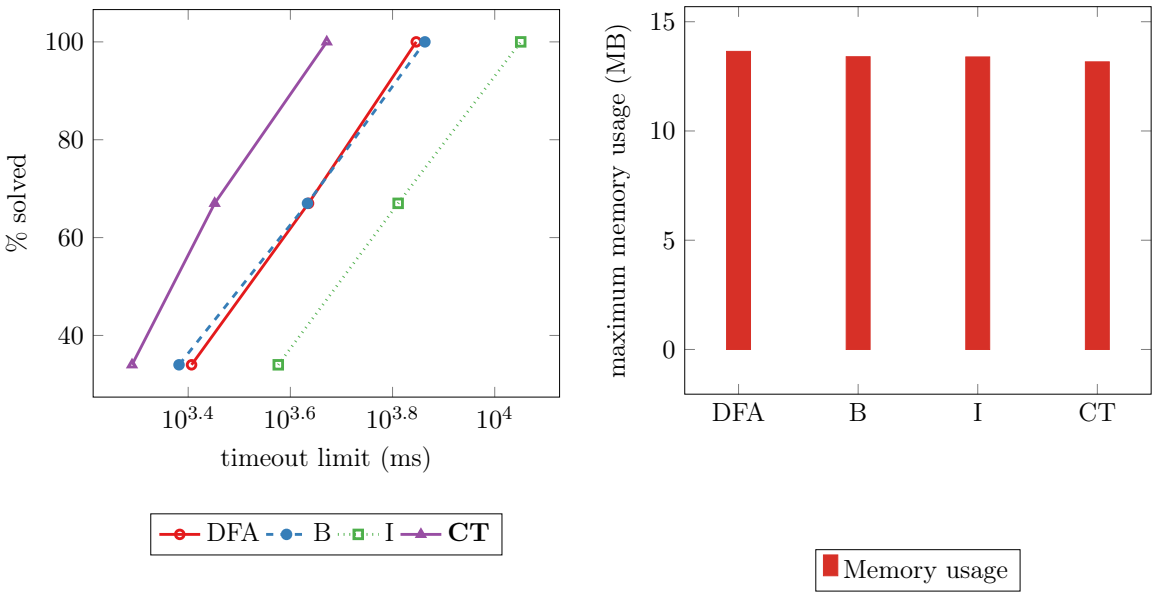

Figure 14: **AIM 50.**

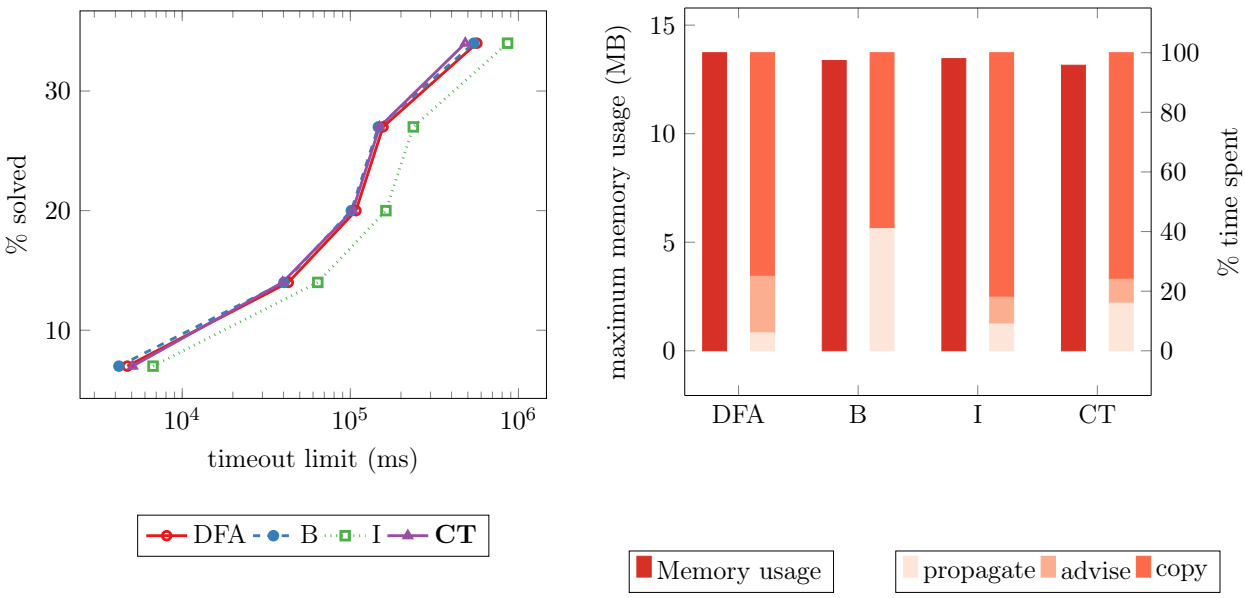

Figure 15: **AIM 100.**

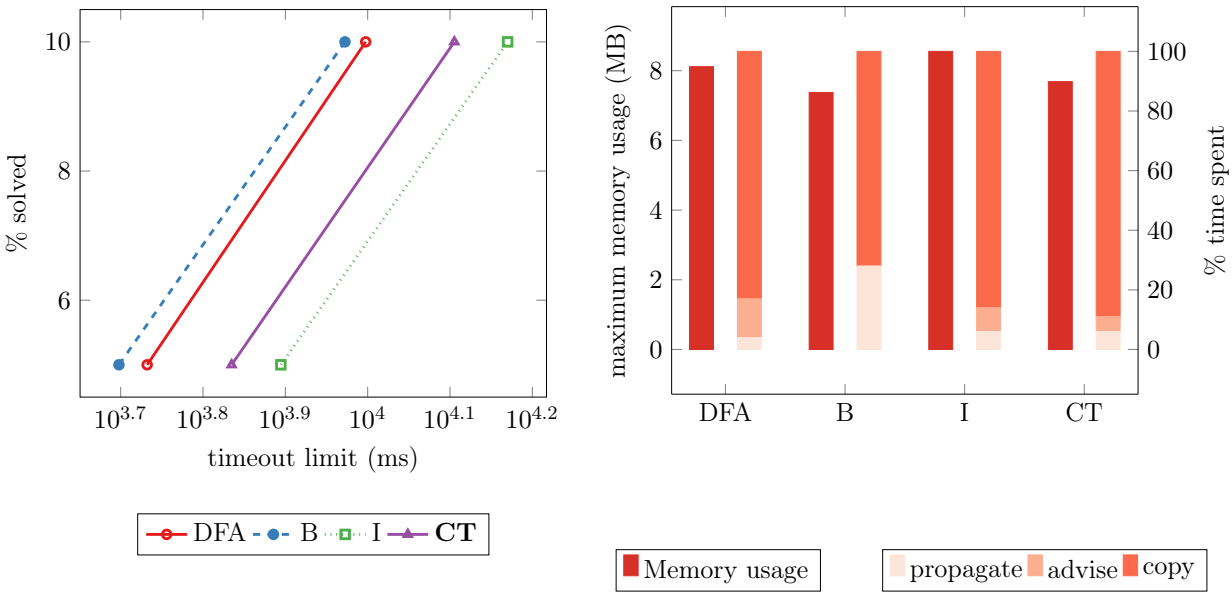

Figure 16: **AIM 200.**

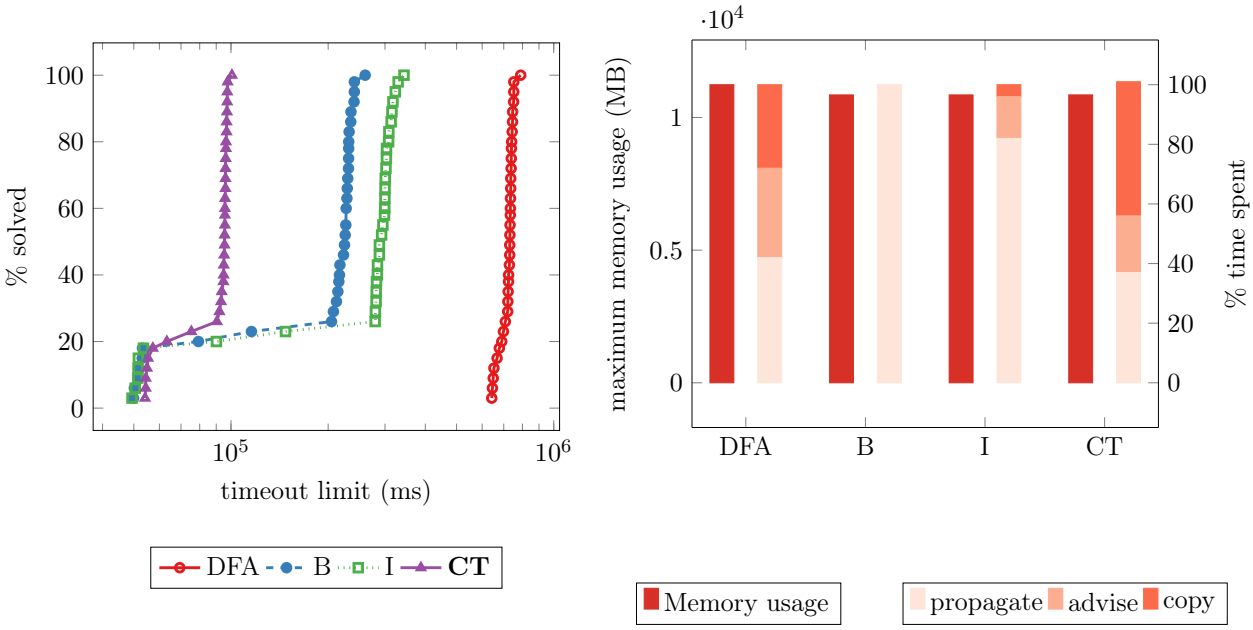

Figure 17: **BDD Large.**

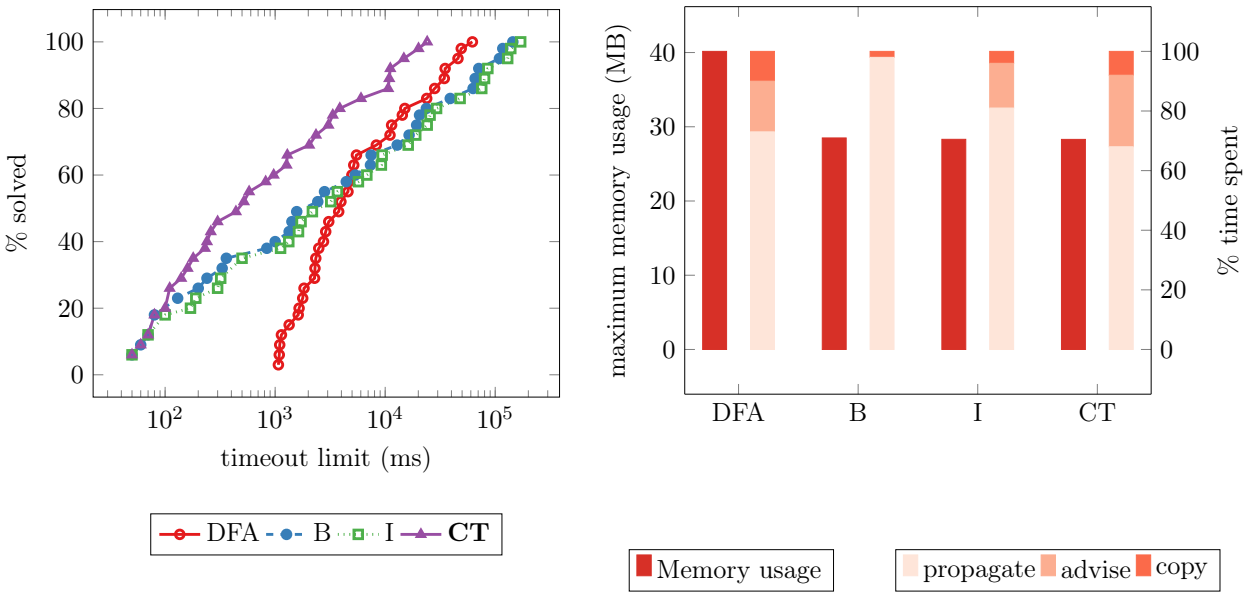

Figure 18: **Crosswords lexVg.**

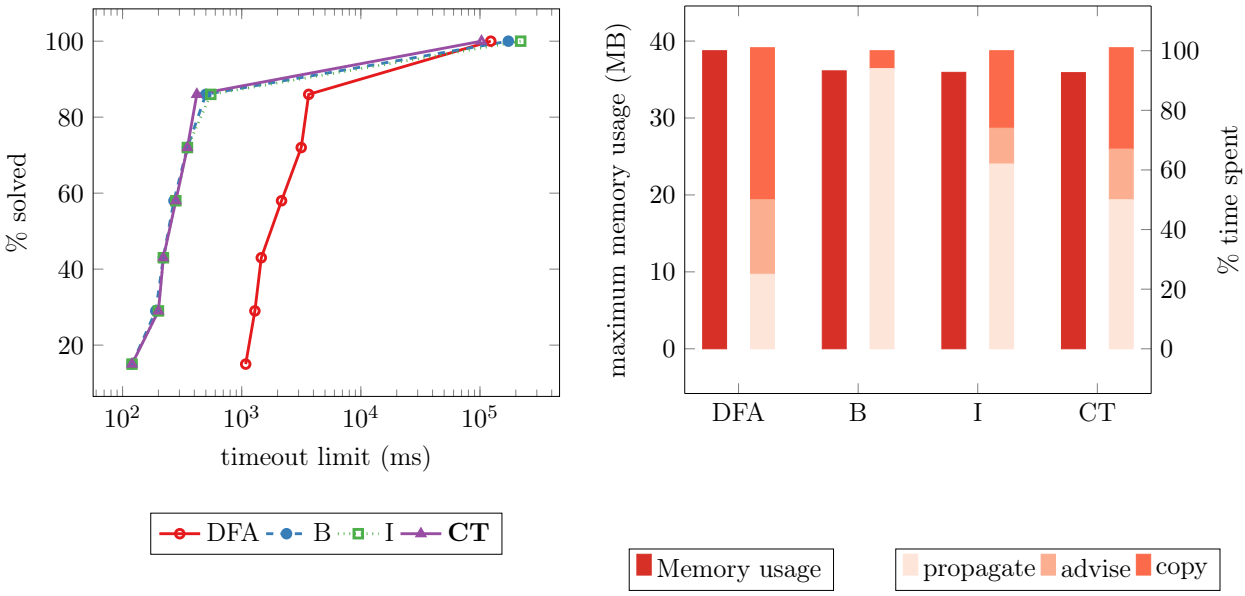

Figure 19: **Crosswords wordsPuzzle.**

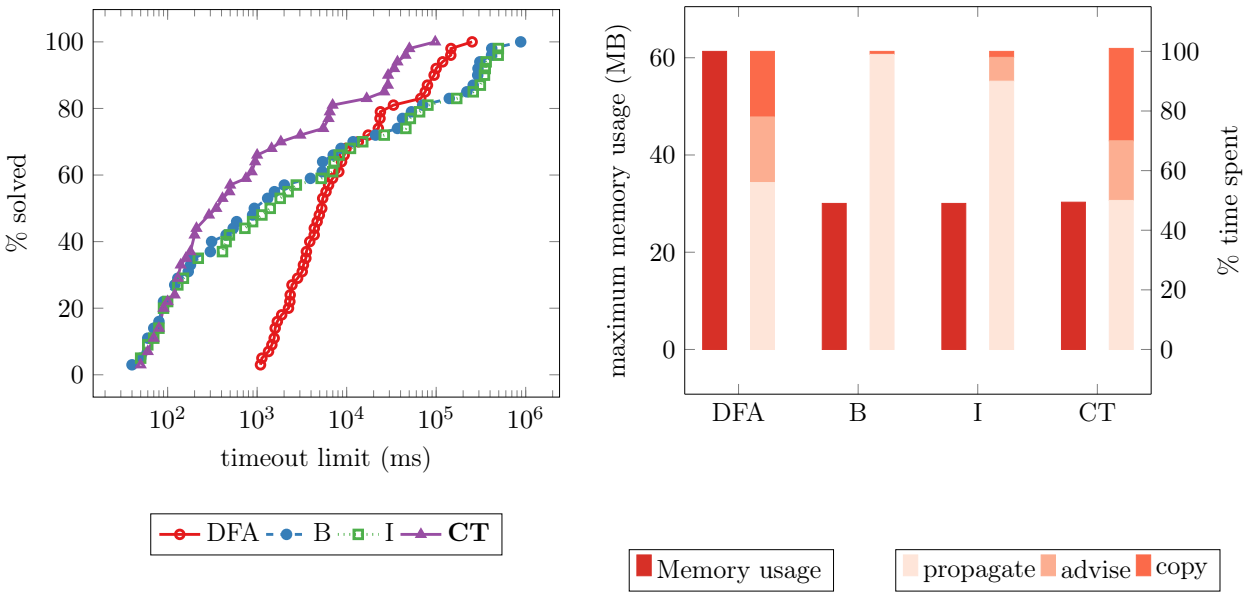

Figure 20: **Crosswords wordsVg.**

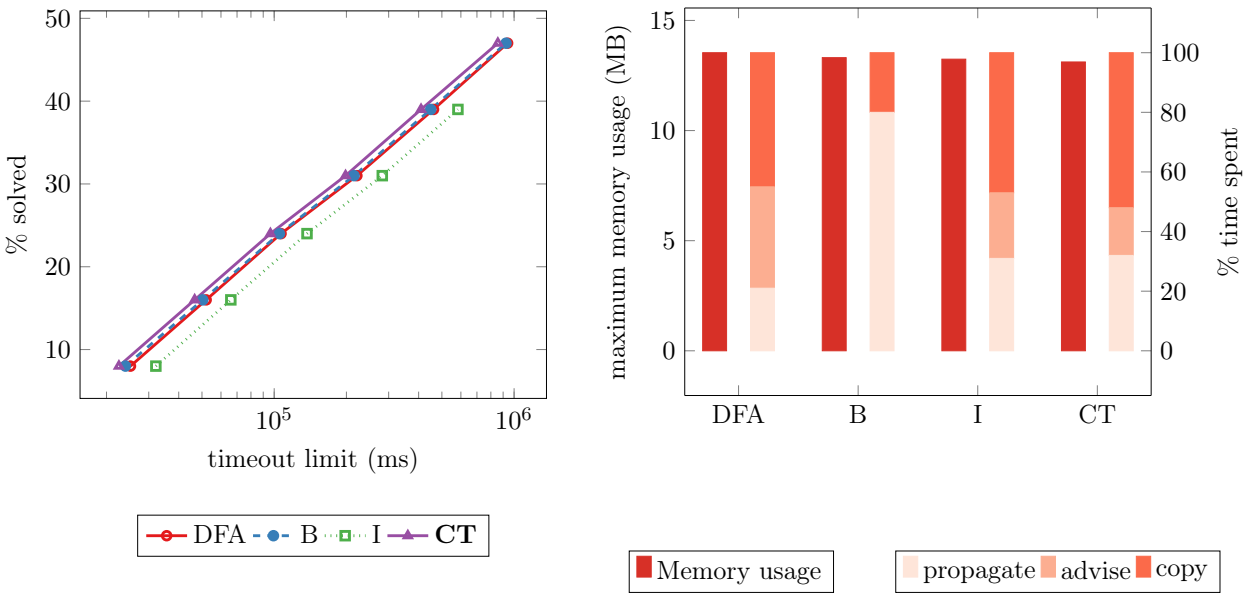

Figure 21: **Dubois.**

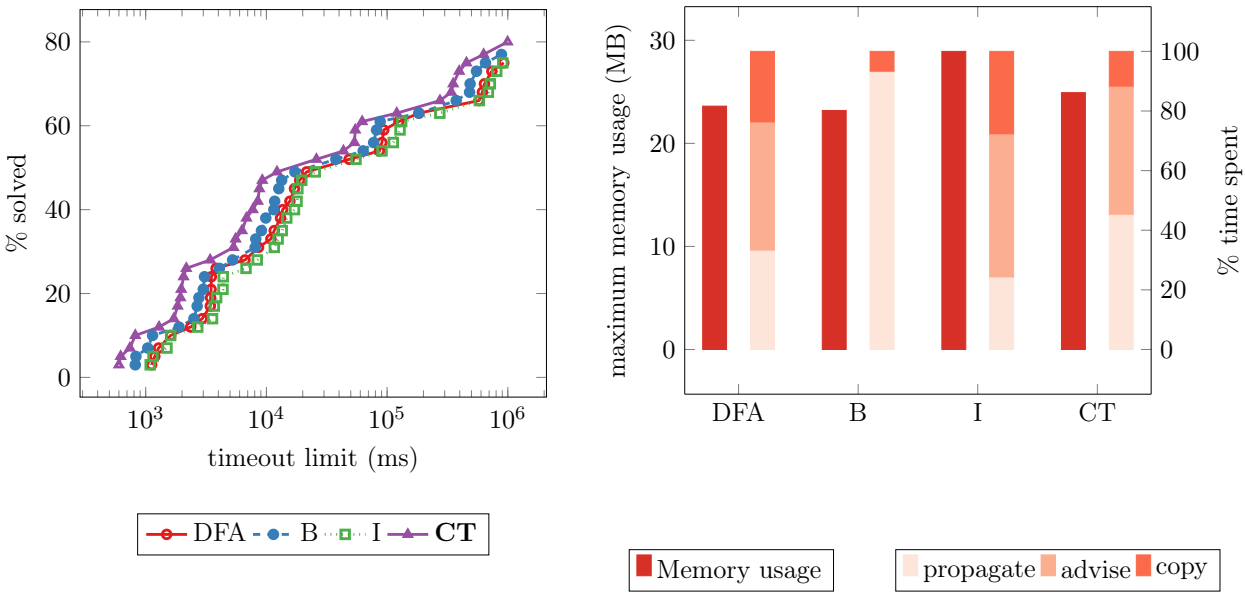

Figure 22: **Geom.**

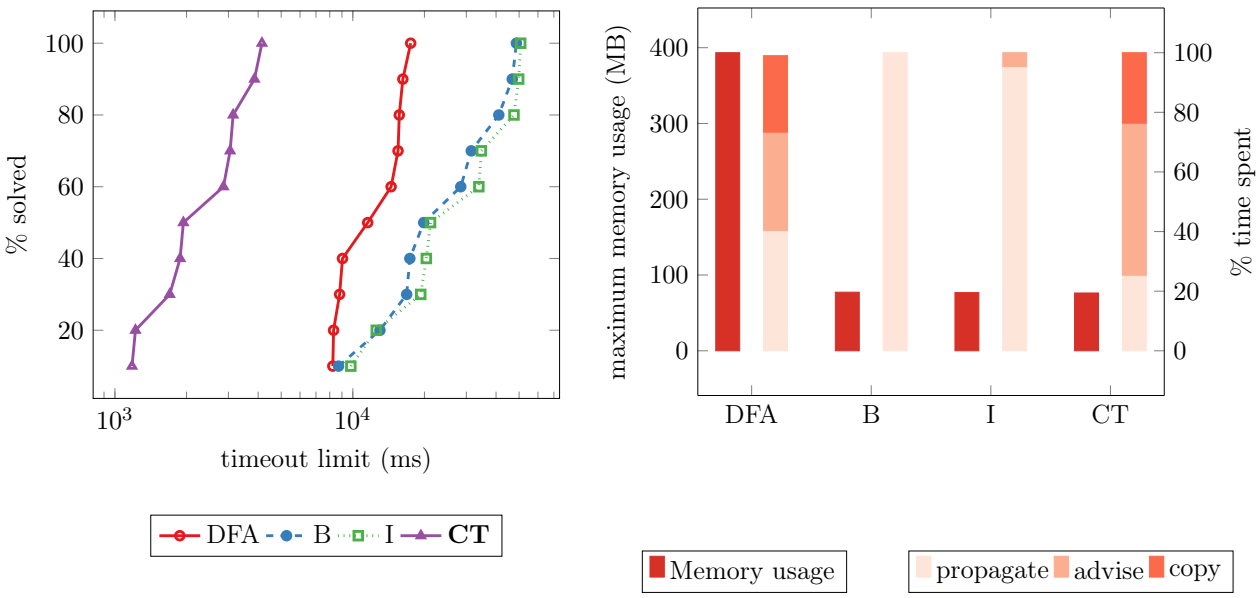

Figure 23: **K5.**

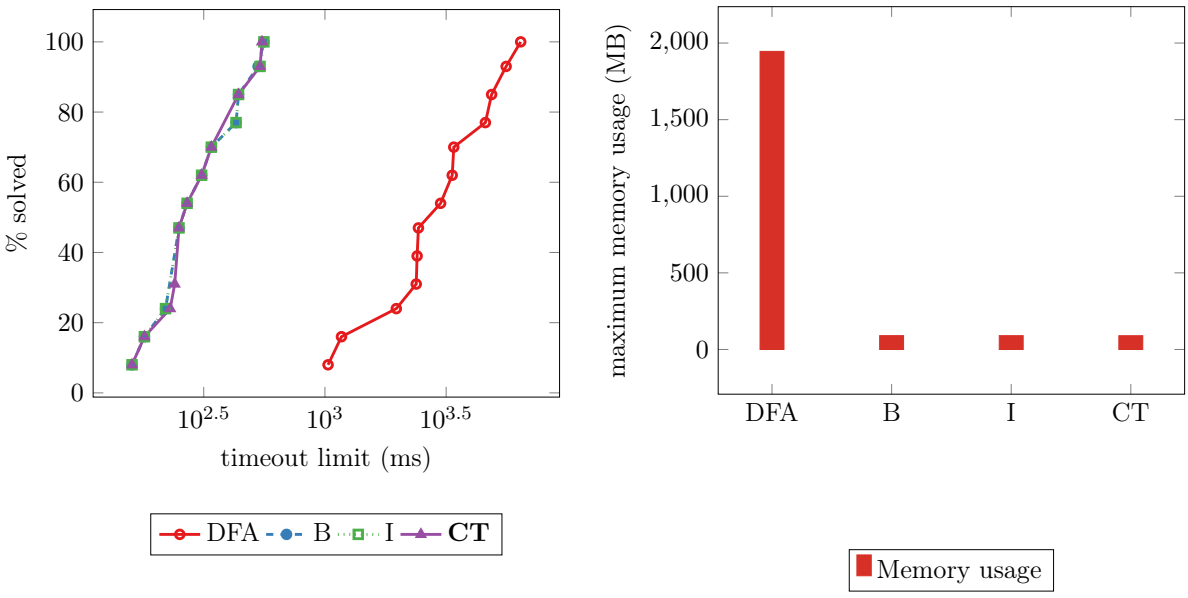

Figure 24: **Kakuro Easy.**

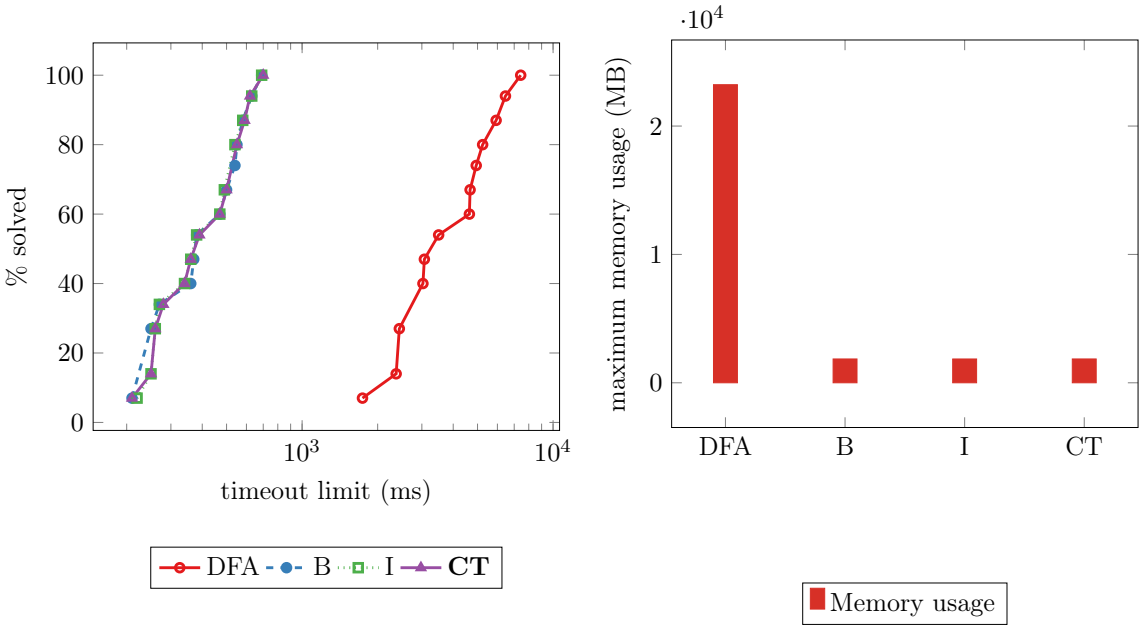

Figure 25: **Kakuro Medium.**

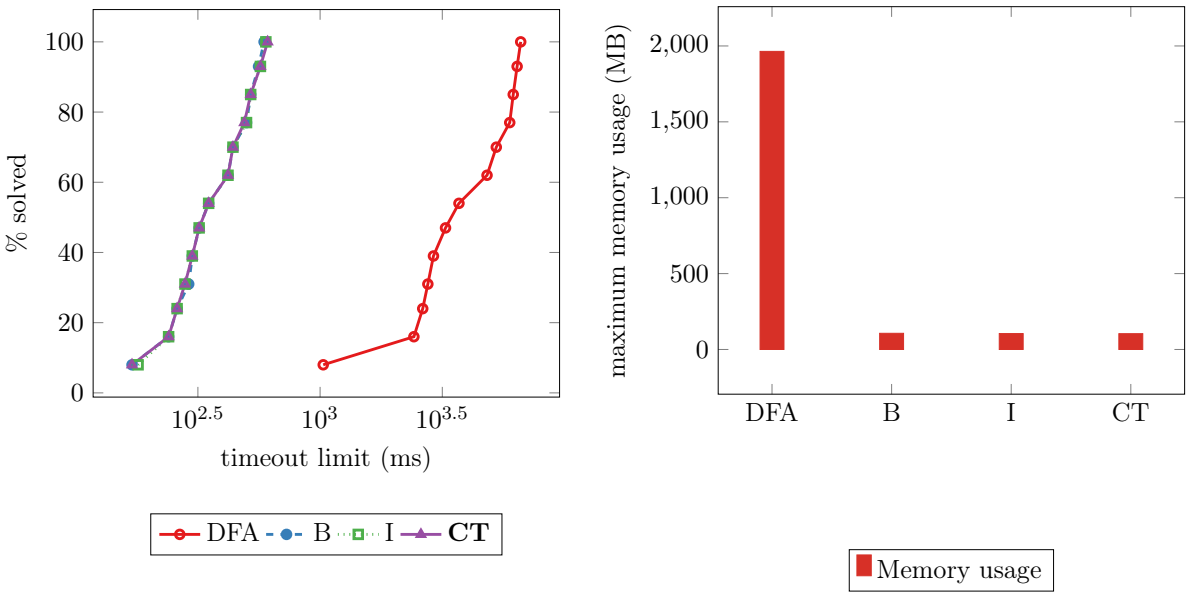

Figure 26: **Kakuro Hard.**

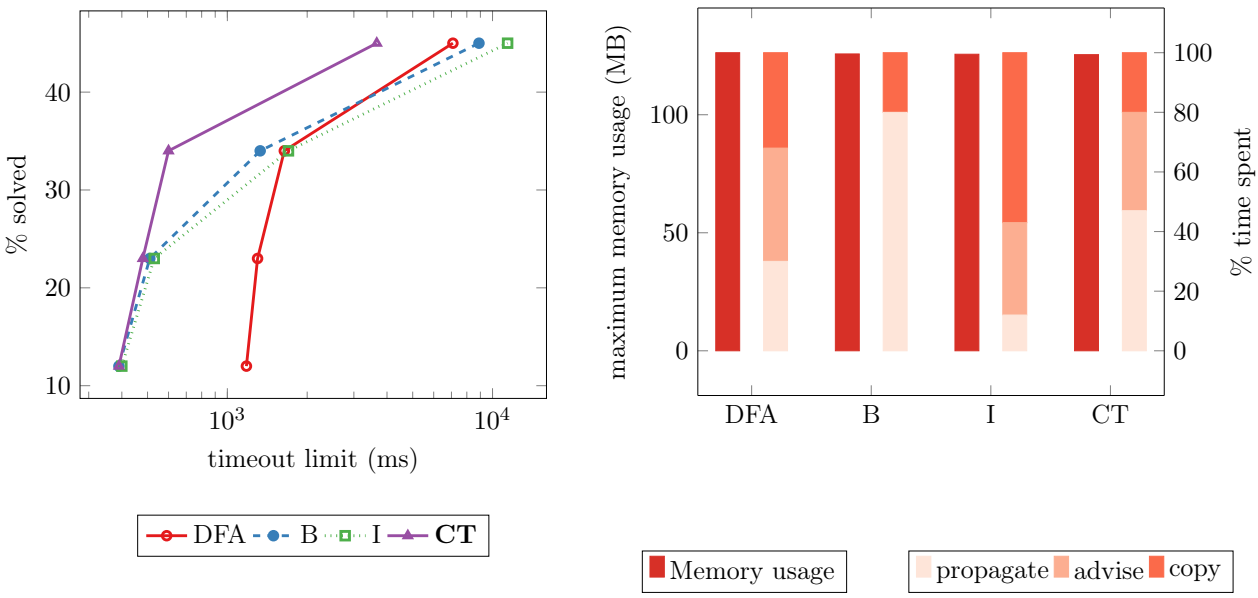

Figure 27: **Langford 2.**

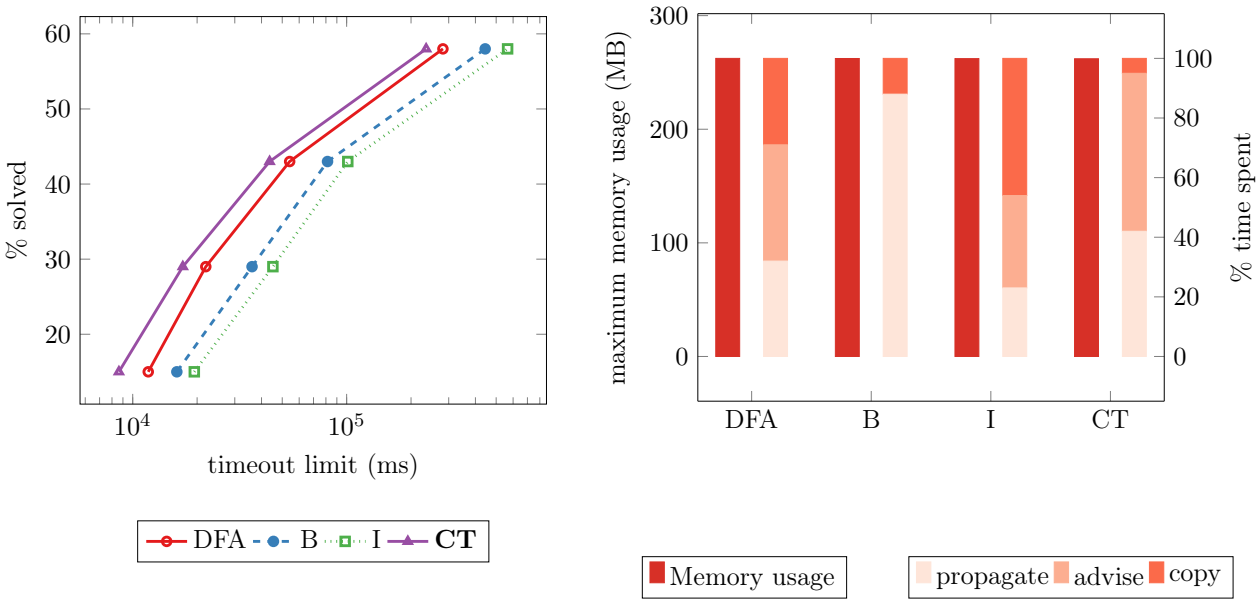

Figure 28: **Langford 3.**

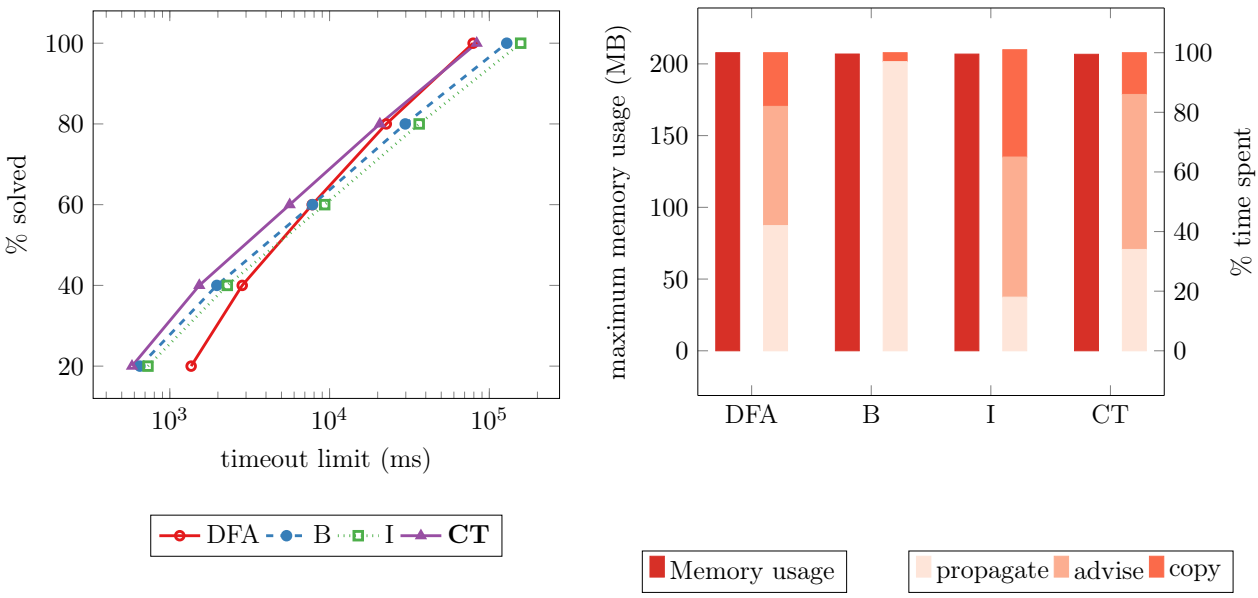

Figure 29: **Langford 4.**

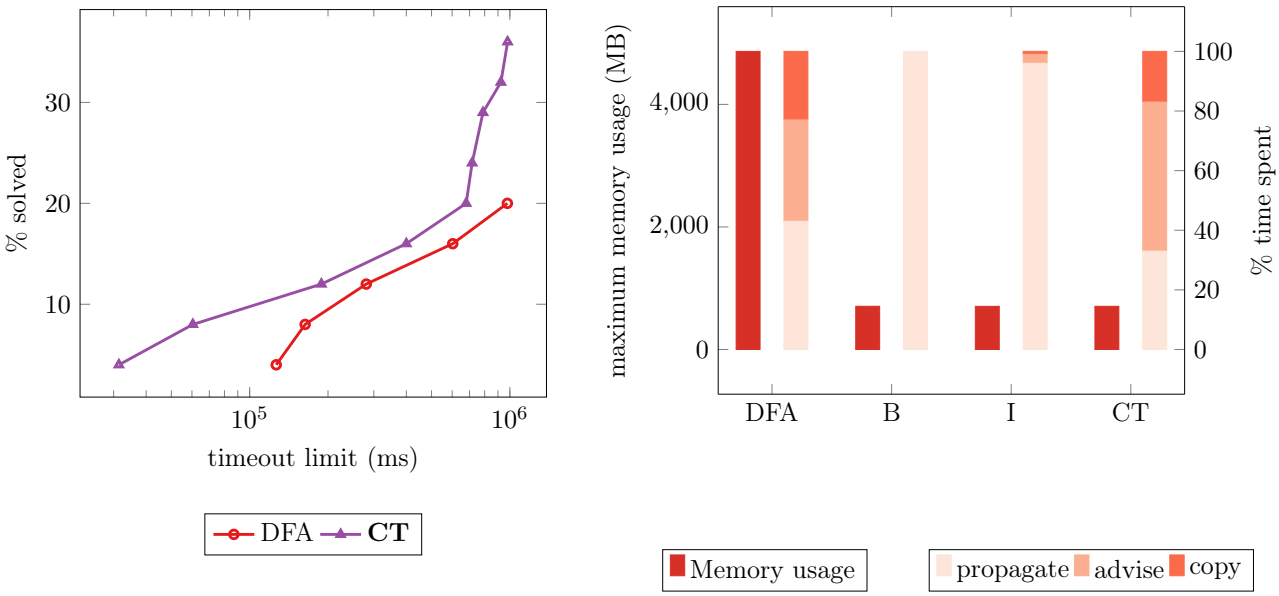

Figure 30: **MDD 05.**

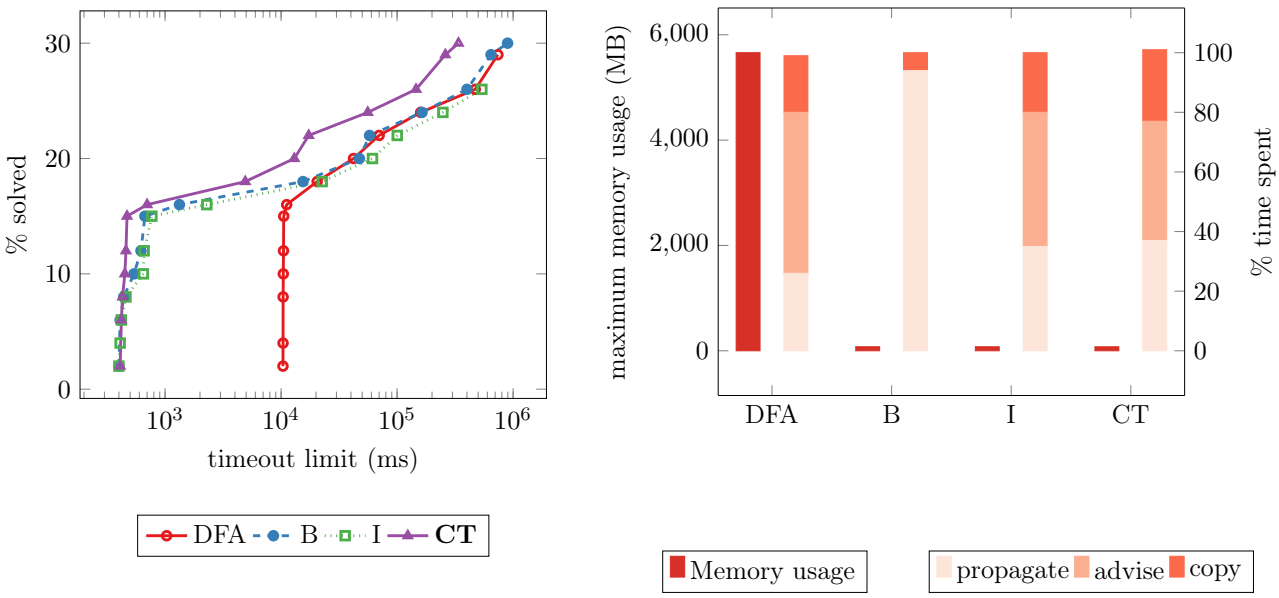

Figure 31: **Mod Renault.**

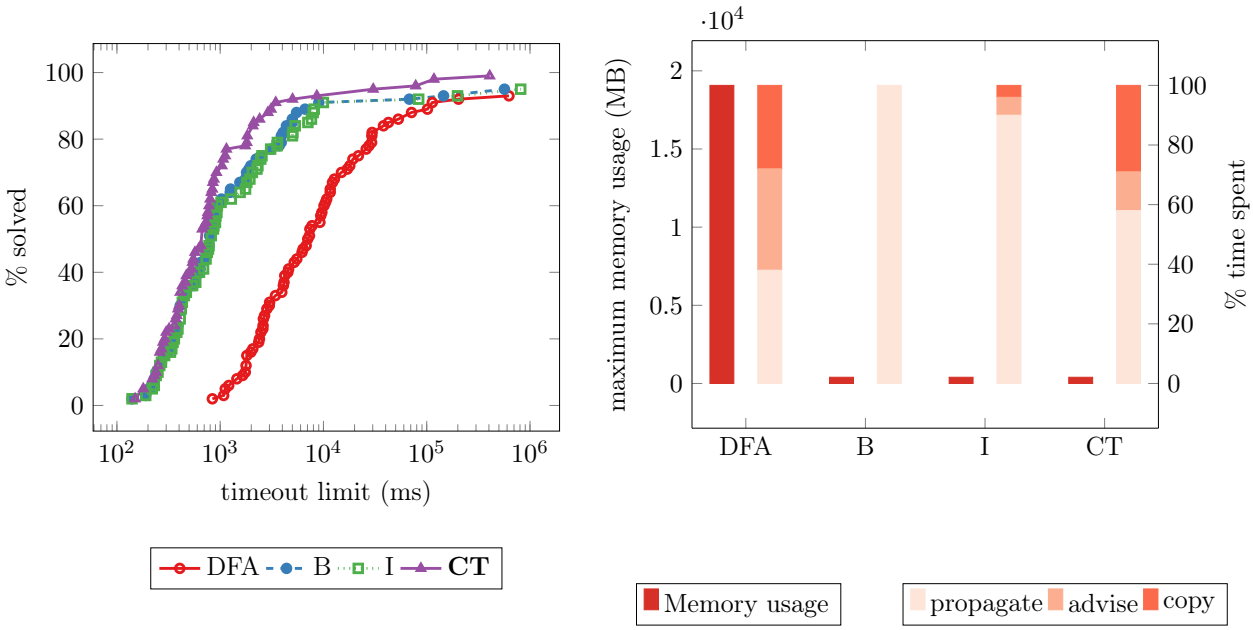

Figure 32: **Nonograms.**

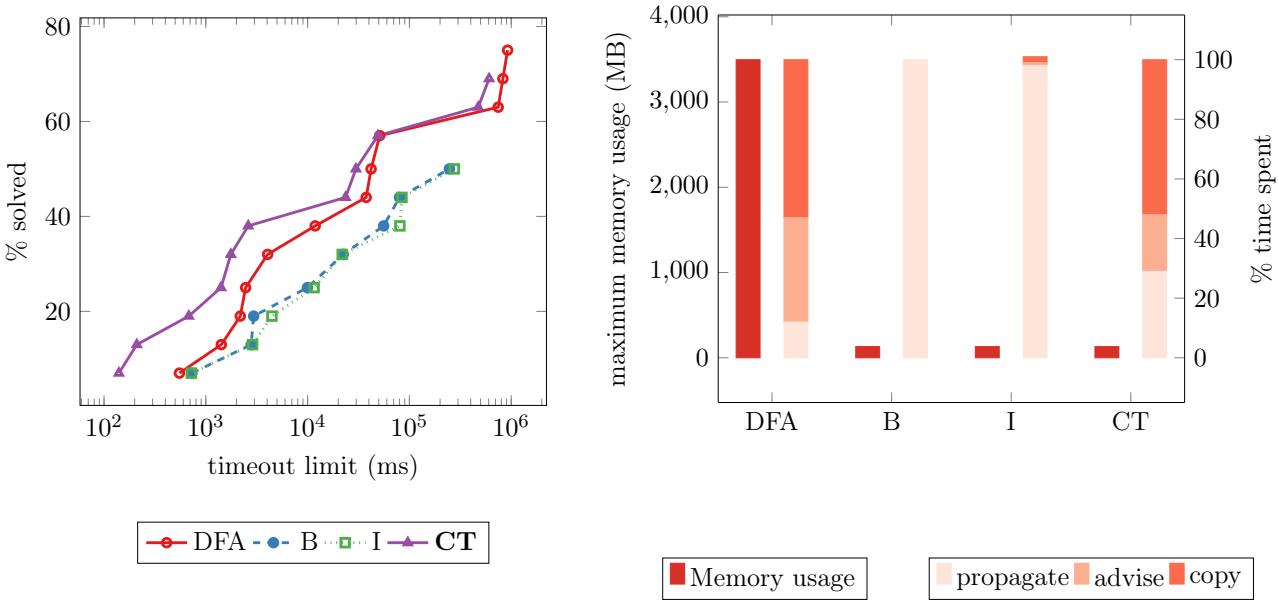

Figure 33: **Pigeons Plus.**

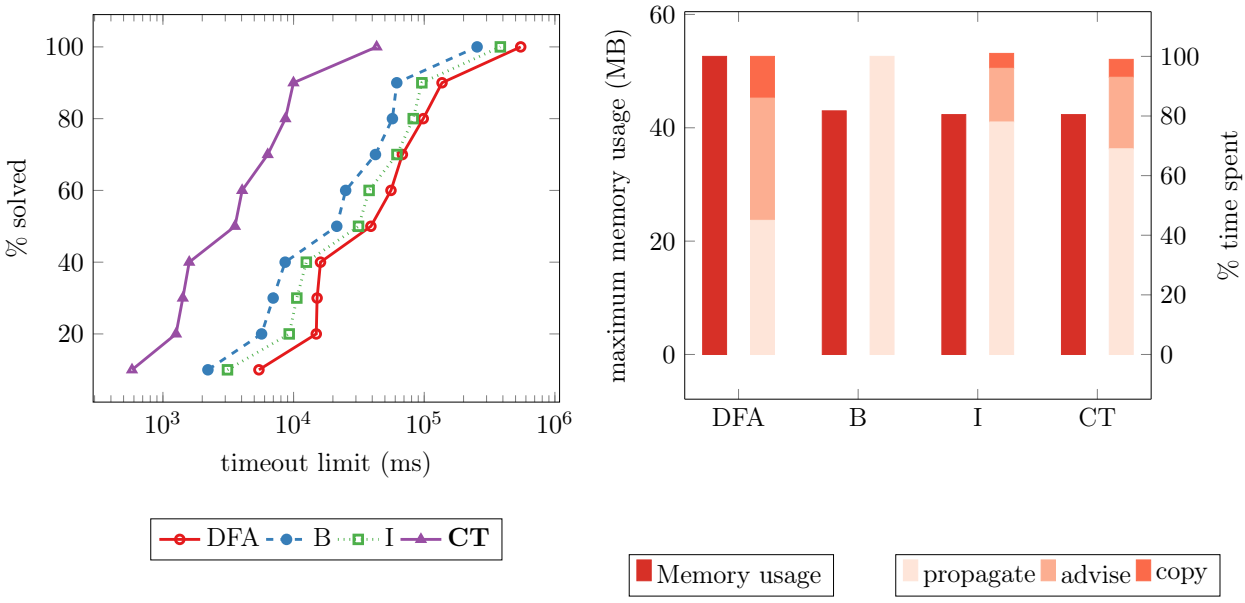

Figure 34: **Rands JC2500.**

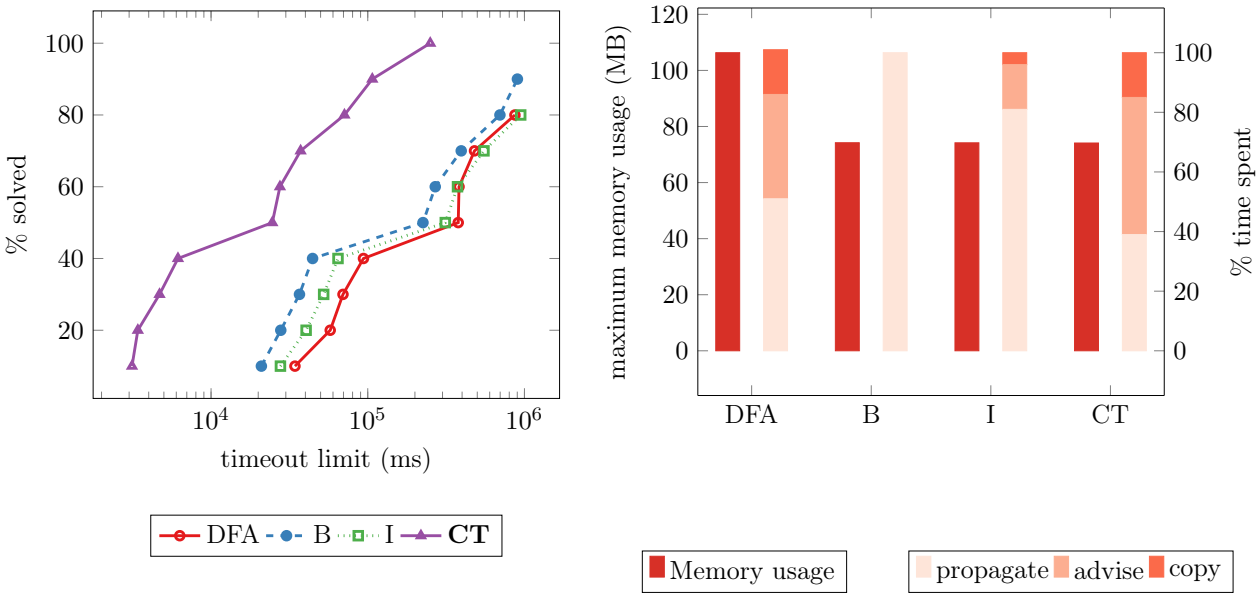

Figure 35: **Rands JC5000.**

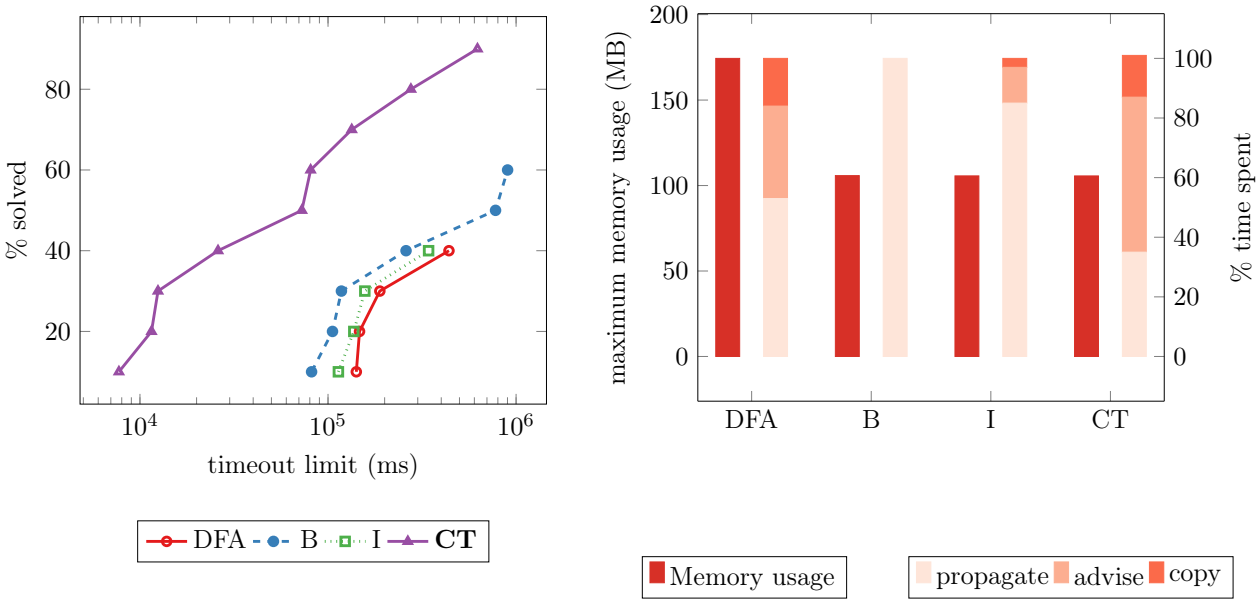

Figure 36: **Rands JC7500.**

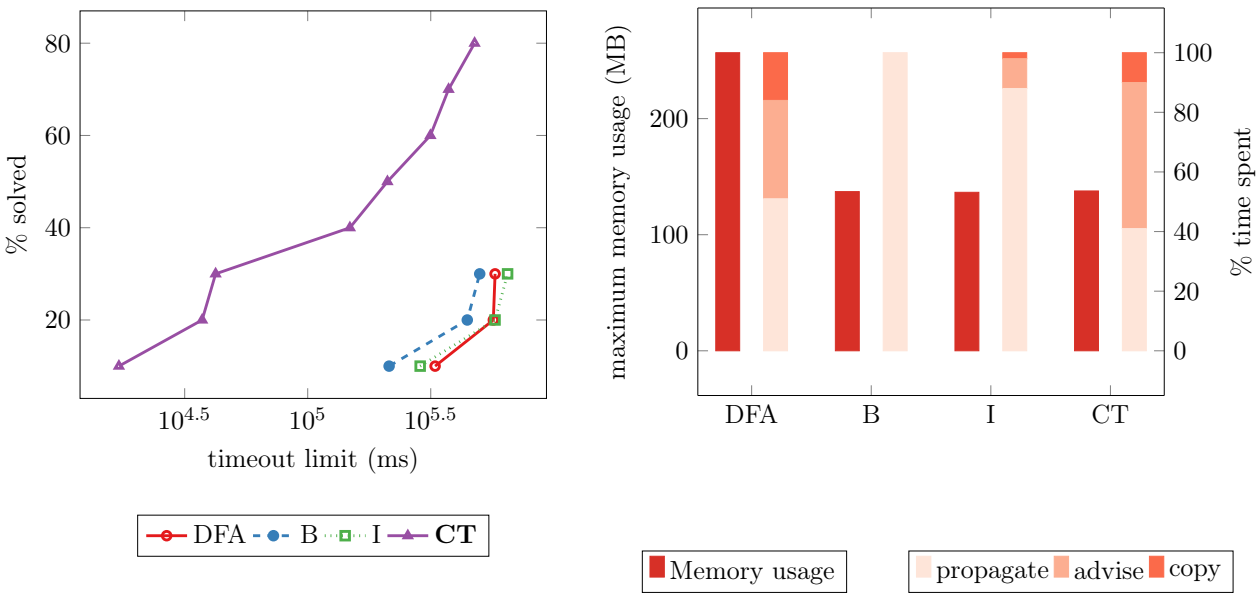

Figure 37: **Rands JC10000.**

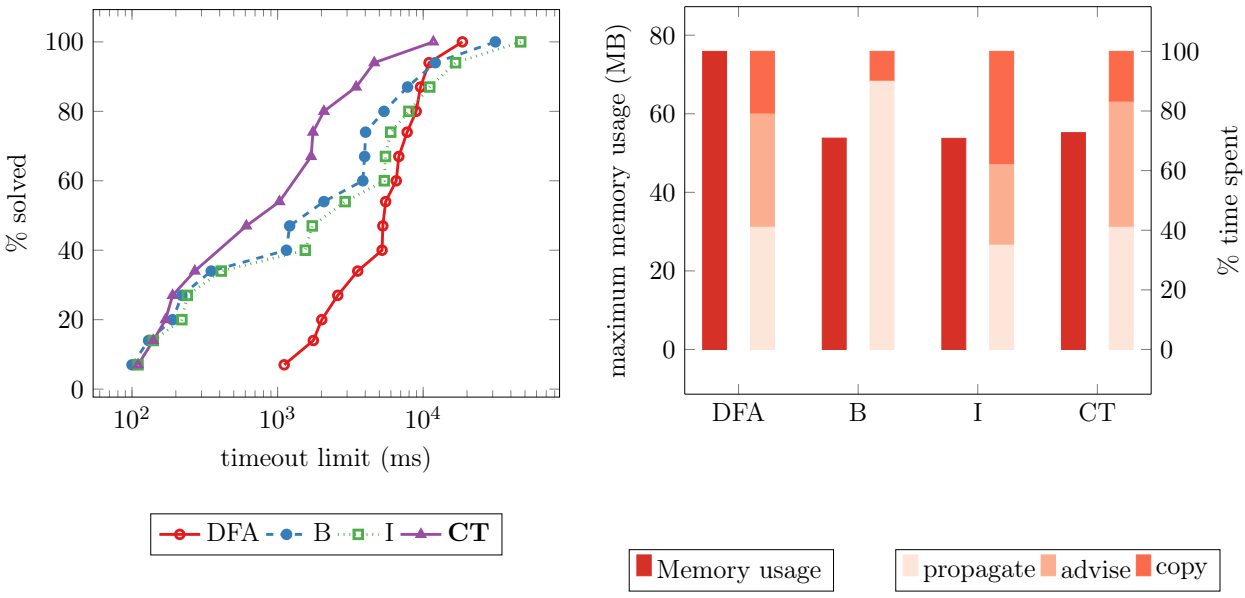

Figure 38: **TSP 20.**

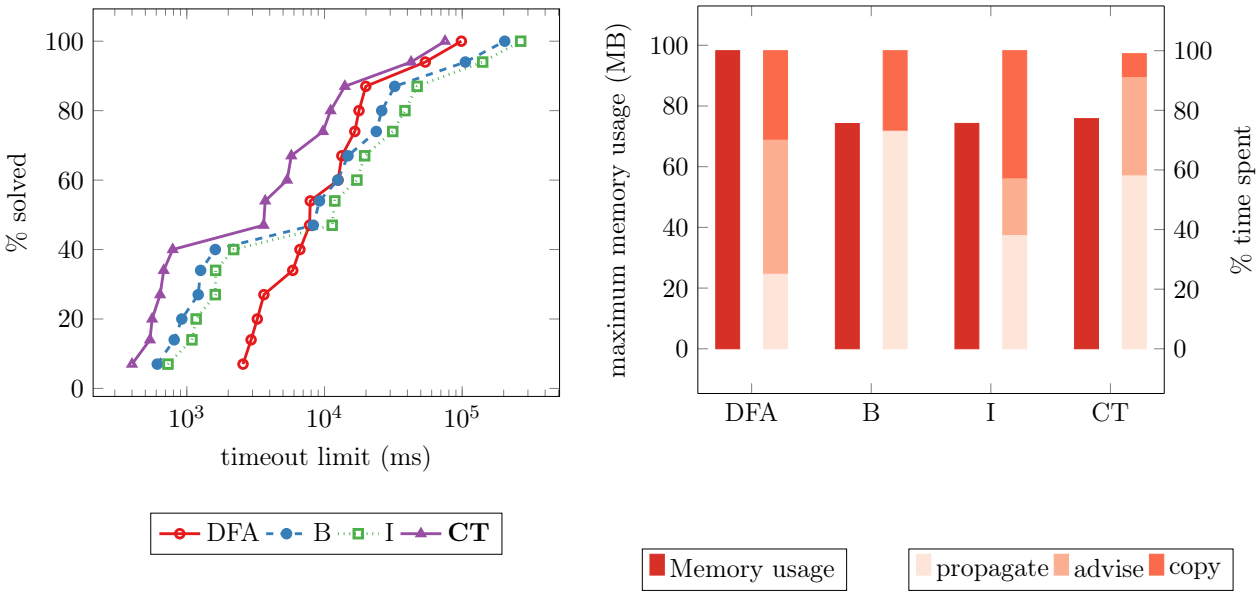

Figure 39: **TSP 25.**

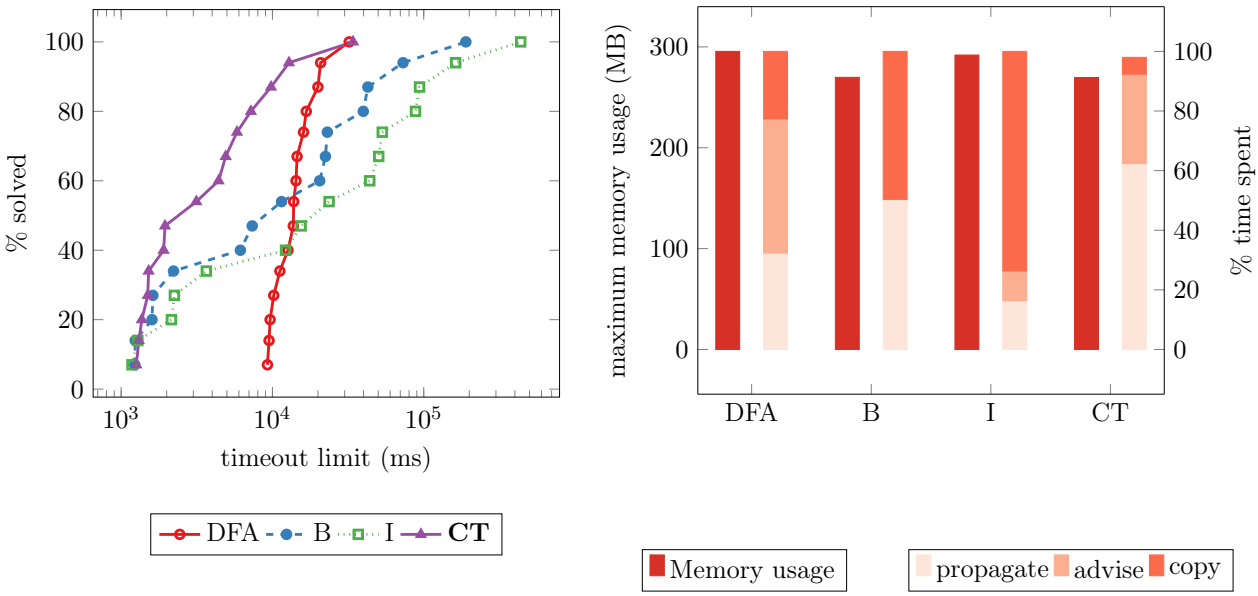

Figure 40: **TSP Quat 20.**