Implementation and Evaluation of a Sweep-Based Propagator for Diffn in Gecode

Mikael Östlund

Institutionen för informationsteknologi
Department of Information Technology
Abstract

Implementation and Evaluation of a Sweep-Based Propagator for Diffn in Gecode

Mikael Östlund

This thesis builds upon Beldiceanu and Carlsson’s sweep-based propagator for a non-overlapping-rectangle constraint. I design and implement a sweep-based propagator for the Diffn constraint, which deals with rectangles generalised to any number of dimensions. Such a constraint is useful in modelling scheduling, assignment, and packing problems. The work is carried out in the context of the copying constraint programming solver Gecode. Different algorithm optimisations are explored and evaluated across a range of benchmarks in terms of inference strength and execution time. The best optimisation configuration is compared against the propagator for Gecode’s current two-dimensional counterpart to Diffn: NoOverlap. The results show that the sweep-based Diffn propagator yields smaller search trees than the NoOverlap propagator in models where non-overlapping constraints dominate the propagation phase, as the sweep-based propagator yields stronger bounds tightening. As other constraints are introduced into the models, the difference in search-tree size becomes smaller, and in cases where the two propagators yield identical search trees, the NoOverlap propagator performs best. While the sweep-based approach shows great potential in some of the benchmarks, the stronger inference is often dwarfed in models with several different constraints.

Handledare: Mats Carlsson (RISE SICS AB)
Ämnesgranskare: Pierre Flener
Examinator: Mats Daniels
IT 17 025
Tryckt av: Reprocentralen ITC
Acknowledgements

I would like to thank my supervisor Mats Carlsson for his guidance, assistance, and elaborate feedback and suggestions on the many drafts of this thesis.

I would also like to thank Pierre Flener of the ASTRA Research Group of Uppsala University for reviewing this thesis and helping improve it, and for his courses related to the field through which he sparked my interest in constraint programming.

Thanks also to Christian Schulte, for providing initial guidelines on how to approach the implementation and explaining some of the underlying architecture of Gecode.
1 Introduction

When solving combinatorial optimisation problems it is often impractical to explore the whole search space of a problem, as such problems tend to be NP-hard and often associated with large search spaces. A way to avoid exploring the complete search space is to prune parts of it using constraint propagation, removing infeasible parts of the search space and thus avoiding redundant exploration.

A particular kind of combinatorial optimisation problems consists of placing objects that must not overlap. Two such sets of problems are packing problems and scheduling problems. To enforce that objects do not overlap, the objects are constrained by a constraint, specifying rules requiring that only assignments that consist of no objects overlapping are allowed, other assignments of the object positions are forbidden. During search space exploration, it can often be inferred that the current search path of the tree has failed before the positions of all objects are fixed. A common constraint for ensuring that $k$-dimensional hyperrectangles do not overlap is often denoted $\text{DIFFN}$, specifying different rules that the objects must obey in order to reside on a feasible position. A propagator is a procedure that, for the $\text{DIFFN}$ constraint, implements the rules and removes (prunes) infeasible positions for the rectangles. It also reports if the rectangles are guaranteed to overlap, in which case no further exploration is needed through the current search path, enabling a part of the search space to be pruned and safely ignored.

In their 2001 paper, Beldiceanu and Carlsson showed how to exploit a sweep-based propagator for a $\text{DIFFN}$ constraint in order to obtain stronger inference and convenient integration with arithmetic constraints on the relevant variables [1]. The sweep idea allows candidate locations for each rectangle to be pruned with regards to the aggregated set of all potentially conflicting rectangles. In their 2007 paper, Beldiceanu et al. proposed $\text{GEOST}$, a generalisation of the $\text{DIFFN}$ constraint for polymorphic $k$-dimensional objects made up of $k$-dimensional hyperrectangles [2]. There has been no attempt to implement a sweep-based algorithm for $\text{DIFFN}$ (or otherwise) in the constraint programming solver Gecode, and consequently its performance in Gecode is unknown.

1.1 Contributions

The contributions of this thesis are the following:

- The sweep-based algorithm proposed by Beldiceanu and Carlsson was adapted and implemented as a new $k$-dimensional $\text{DIFFN}$ constraint in the copying constraint programming solver Gecode. Previously, the sweep-based algorithm has only been used in context of trailing-based solvers.

- Several optimisation ideas for the implementation were implemented and evaluated, microbenchmarks as well as macrobenchmarks were considered, most, but not all, of which were partially or completely derived from the work of Beldiceanu and Carlsson in [1].
The implementation was evaluated and compared against the Gecode’s current 2-dimensional counterpart NOOVERLAP across several benchmarks, including benchmarks featuring more than only the DIFFN constraint.

The inference strength of the sweep-based algorithm was evaluated by examining the number of failures in the search tree.

An examination was made where time was spent inside the propagator’s components over the different algorithm optimisation configurations for particular instances of the benchmarks. A brief examination of the memory utilisation of the optimisations was also performed.

1.2 Outline

This thesis is structured as follows. Section 2 covers background material, specifically an overview of constraint programming and Gecode, followed by a description of the DIFFN constraint and an overview of a value sweep algorithm, proposed by Beldiceanu and Carlsson. Section 3 covers both the current algorithm for Gecode’s DIFFN 2-dimensional counterpart NOOVERLAP as well as the algorithms used in the implementation of the sweep-based DIFFN propagator. The DIFFN implementation is then covered in Section 4 in-depth, including design choices and the choice of data structures. The implementation of the propagator is then evaluated in Section 5, comparing its performance with its current 2-dimensional Gecode counterpart NOOVERLAP. Section 5 also includes a discussion of the evaluation results and tackles three research questions. The thesis is then concluded in Section 6 with a summary of the results and provides recommended future work.

2 Background

This section gives an overview over the relevant preliminaries. First, notation used throughout the thesis is defined in Section 2.1. Constraint programming is then covered in Section 2.2 followed by a breakdown of the constraint programming solver Gecode in Section 2.3. In Section 2.4 the DIFFN constraint is described. A sweep algorithm for pruning the minimum and maximum of variable domains proposed by Beldiceanu and Carlsson is explained in Section 2.5.

2.1 Notation and Terminology

- The notation $x..y$ is used to represent the integer interval corresponding to the set of integers $\{x, x+1, x+2, \ldots, y\}$.

- The notation $\mathcal{D'} \subseteq \mathcal{D}$ denotes that $\mathcal{D'}(x) \subseteq \mathcal{D}(x)$ for every $x$ defined by functions $\mathcal{D}$ and $\mathcal{D'}$. 
• The terminology *tightening the lower* or *upper bound* of a variable is a shorthand for increasing the lower bound and decreasing the upper bound of the domain of that variable, respectively. Similarly, *pruning an object* o acts as a shorthand for pruning some variable domain stored in the object.

### 2.2 Constraint Programming

Constraint programming (CP) is a paradigm that uses knowledge from fields such as artificial intelligence, computer science and operational research to solve combinatorial problems [3, Chapter 1]. Much like in *declarative* programming, a CP problem is formulated at a high level specifying what problem to solve, in contrast to an *imperative* paradigm, where the programmer instead states how to solve a problem. A problem specification in CP is called a *model*, specifying mathematical variables relevant to the problem, as well as constraints over the variables. A CP model of a combinatorial problem is solved by a CP solver, taking the model, possibly alongside instance data, as input.

**Definition 2.1.** A *domain* \( D \) is a complete mapping from a fixed set of variables \( V \) to finite sets of integers. \( D(v) \) denotes the variable domain of a variable \( v \in V \). If \( D(v) = \emptyset \) for any such \( v \), then the domain \( D \) is failed. A variable \( v \in V \) is fixed by \( D \) if \( |D(v)| = 1 \), i.e., if its variable domain is a singleton. If \( D' \sqsubseteq D \), then \( D' \) is stronger than \( D \) [3, Chapter 14].

Two types of constraint problems often occur; the following definitions are in part taken from [3, Chapter 2].

**Definition 2.2.** A *constraint satisfaction problem* (CSP) is a triplet \( P = \langle V, D, C \rangle \), where \( V \) is a set of variables and \( D \) their corresponding domain such that every \( v_i \in V \) has a domain \( D(v_i) \). Further, \( C \) is a set of constraints over the variables in \( V \), defining relations between the variables that must hold. An assignment to the CSP \( P \) is a triplet \( P' = \langle V, D', C \rangle \) such that \( D' \sqsubseteq D \) and where all variables \( v \in V \) are fixed by \( D' \). If such an assignment exists and does not violate any constraint in \( C \), then we say that the problem is satisfiable and that the assignment is a solution, else we say that the problem is unsatisfiable.

**Definition 2.3.** A *constraint optimisation problem* (COP) is a 4-tuple \( P = \langle V, D, C, f \rangle \) that is a CSP with an additional objective function \( f \), to be optimised. The function \( f \) is defined over variables in \( V \) such that it denotes a cost value for each solution, indicating the solution’s quality. This value is then either minimised or maximised, depending on the type of problem. If the COP is satisfiable, then a solver returns a solution corresponding to the optimal value of \( f \) if an optimal solution is found before any user-defined timeout, else if the optimal solution is not found yet, then the solution corresponding to the best objective value found thus far is returned, if any. In the case that the COP is unsatisfiable, no solution is given.

**Example 2.1.** Consider the problem of solving a Sudoku puzzle, illustrated in Figure 1. This combinatorial problem can be modelled in CP as a CSP \( P = \langle V, D, C \rangle \) using \( |V| = 81 \) variables, where \( D(v_i) = 1..9 \), for each variable \( v_i \in V \). Each respective variable represents a unique field of the Sudoku board. The constraints posted on the variables are the following:
Figure 1: Illustration of a Sudoku puzzle along with its unique solution.

- All rows must have distinct values
- All columns must have distinct values
- Each disjoint $3 \times 3$ region, highlighted by bold lines, must have distinct values

The model alongside a Sudoku instance is then given to a solver that interleaves inference (propagation) and search to find a solution to the Sudoku puzzle. The former removes values violating any of the constraints from the corresponding variable domains. The latter creates subtrees of the current node when no propagation can be performed, by partitioning some of the variable domains contained in the parent node.[4]

2.2.1 Propagation

The act of removing values that violate a constraint from the domain of a variable is called pruning the domain of a variable. In order to infer what values to prune on a constraint-to-variable basis, a propagator is used. A propagator for a constraint $C$ is responsible for pruning values from variable domains that violate the constraint $C$.[4]. Each propagator only knows of one of the constraints in a CSP or COP, and reasons about a combinatorial problem locally. Communication between propagators is done by the pruning of values, as propagators may share variables. A propagator subscribes to domain changes to its variables, thus it is aware of external changes to its variable domains.[4]

Each constraint is implemented as one or several propagator algorithms, which prune variable domains according to the constraint. A propagator has no obligation

---

1Bisection is often used, but not required. As such, the resulting search tree need not be binary.
to remove all values of a variable domain that violate the constraint: it all depends on
the strength of a propagator [4]. The strength, or more formally, the consistency of a
propagator is often chosen by the modeller at the modelling stage and is often specified
as one of two choices. The two following definitions are in part from [4].

**Definition 2.4.** Under domain consistency, every value of each variable domain participate in
some assignment satisfying the constraint.

**Definition 2.5.** Under bounds consistency, the lower- and upper-bound values of each variable
domain participate in some assignment satisfying the constraint.

A bounds-consistent propagator often results in a lower computational complexity
class but may achieve weaker inference compared to domain consistency.

**Definition 2.6.** A propagator $p$ is a function from domains to domains with several obliga-
tions [5, Chapter 23]:

- **Correct.** It is required that a propagator never prunes values that can appear in a solution
  of the propagator’s corresponding constraint.

- **Checking.** It is required of a propagator that it can distinguish a solution and a non-
solution. A propagator reports failure when it can be determined that no assignment of
  the variable domains satisfies the constraint. When it can be inferred that no assignment of
  the variable domains can violate the constraint, the propagator reports subsumption,
  indicating that all future assignments of the variable domains will fulfil the constraint [4].
  Given an assignment, the propagator must report failure if the assignment violates the
  constraint, else subsumption.

- **Contracting** (sometimes called decreasing). A propagator is only allowed to remove
  (prune) values from a domain; it is not allowed to introduce new values to a domain.

- **Monotonic.** Preferably, while not mandatory [4], a propagator $p$ is monotonic, meaning
  that when applied to stronger domains, it also yields stronger domains. I.e., $D' \subseteq D \Rightarrow
  p(D') \subseteq p(D)$, for any domains $D$ and $D'$.

- **Fixpoint and subsumption honest.** A propagator must not report fixpoint or subsump-
tion unless it is indeed the case. E.g., a propagator may not report fixpoint when it can
  perform more pruning.

A design decision regarding a propagator $p$ is whether it is idempotent, meaning
that it calculates a fixpoint of $p$ when it is executed. I.e., a propagator $p$ is idempotent
if $p(p(D)) = p(D)$ for any domain $D$ [3, Chapter 14]. The DIFFN propagator imple-
mented in the context of this thesis is an idempotent propagator. While non-idempotent
propagators do not necessarily reach a fixpoint, they are normally designed to report
whether they have reached fixpoint after execution.

When a solver performs inference on the search space, it executes propagators for
the constraints of the problem, until each propagator reaches fixpoint. When no more
pruning can be performed on the variable domains by any of the propagators, we say that the algorithm itself has reached fixpoint. If all values are assigned (i.e., all variable domains are singletons) at that point, then a solution has been found. Otherwise, search is required.

**Example 2.2.** Consider Figure 2. There are 3 variables, namely $x \in 2..10$, $y \in 0..10$, and $z \in 3..10$. Three constraints, each with its corresponding bounds-consistent propagator are included:

- $c_1 : x < y$
- $c_2 : x + y < 8$
- $c_3 : y > z$

Figure 2 shows the propagation of the corresponding propagators $p_i$ for $c_i$. In the first iteration, $p_1$ will prune $x$ and $y$ by inferring that $x$ must be at most 9, since $x > 9$ would violate the constraint $x < y$. $p_1$ also prunes $y$ by restricting its lower bound to be no lower than 3. In the same iteration of propagation, $p_2$ infers that $x$ can be at most 4 in order to fulfill the $c_2$ constraint. It also infers that $y$ can be no larger than 5. Lastly, still in the first iteration of propagation, $p_3$ infers that $y$ must be at least 4 and at most 5 and $z$ must be at least 3 and at most 4. In the second iteration, $p_2$ is called to action and prunes away the value 4 from $x$. In the third iteration no further propagation is possible, and the propagation algorithm is at fixpoint.

The observant reader may notice that the behaviour is identical no matter if domain or bounds-consistent propagators are used. This is due to the nature of the included constraints, as they are only concerned about limits of variables.

### 2.2.2 Search

When no more values can be propagated and not all variables have been assigned a value, search is needed. In CP, search is often performed using a backtracking method [3, Chapter 4]. This method assures that all solutions will be found if solutions exist, i.e., the search is total. Further, since the method guarantees to find all solutions, it can be used to prove optimality by finding the globally best solution in the entire search space. One advantage of using backtracking search over another search strategy, like dynamic programming, is that it uses less memory in general, and needs only a polynomial amount of space [3, Chapter 4].
One can view a backtracking search as a depth-first search (DFS) of a tree, representing the search space. The tree is generated during search, by partitioning the domain of a chosen variable $v$, creating subtrees, each with its own partition of $v$. The choice of variable and how to partition the domain is done by a brancher, using a branching heuristic. However, branching heuristics are out of the scope of this thesis.

A backtracking algorithm will visit a node in the search tree if it has been generated sometime during the algorithm execution. Constraints are used to infer whether a branch (subtree) of the search tree is viable, i.e., whether the branch potentially contains a solution [3, Chapter 4]. Subtrees with a root node that is unsatisfiable according to some constraint contain no solutions and are pruned; such a subtree is called a dead end.

An example of a search tree is found in Figure 3. Note that a search tree is not necessarily binary, as the number of children for each node depends on the branching heuristic chosen.

For COPs, a branch-and-bound (BAB) search heuristic is applied. It is similar to the regular backtracking algorithm but stores the current best solution during search, enabling it to prune branches that will certainly not lead to a better solution. This is done by posting a constraint each time a solution is found; the objective value of any new solution must be better than the objective value of the previous solution [3, Chapter 11].

2.3 Gecode

Gecode is an open and free constraint programming library, written in C++ [5, Chapter 1]. Gecode enables programmers to write their own propagators, branching heuristics and search engines, among other things. Gecode is also an efficient CP solver that won all gold medals in the MiniZinc Challenge in 2012, 2011, 2010, 2009, and 2008 [5, Chapter 1]. It also received a bronze medal in 2013, but was later ineligible to compete.
due to conflict of interest, as one of Gecode’s coauthors got involved in the MiniZinc project.

Propagators in Gecode modify views rather than variables directly [5, Chapter 23]. When a variable is created for use in modelling, a variable implementation is simultaneously created, corresponding to said variable. The variable itself is to be viewed as a read-only interface to the variable implementation; a view is also an interface to a variable implementation, but more extensive, as it enables value access as well as removal.

Search requires that previous computation states (search tree nodes) must be available at a later state, as they may be needed for further exploration of unexplored branches of the search tree [6]. Gecode utilises copying of variable domains to store previous computation states, enabling (search tree) exploration [4]. This has mainly two implications. The first is that the search procedure and the propagation procedure through the fixpoint algorithm are orthogonal, meaning that exploration does not affect the propagation procedure. While copying enables components of the solver to be independent, it does come with a twist, the second implication: memory usage. Since the data structure for representing variable domains must be copied each time a decision is made during search, it is crucial that its memory footprint be minimal. Data structures for storing states of other solver components should also have a minimal memory footprint. Exploration through copying is complete, meaning that it will find all solutions to a problem, if any exists [4].

Gecode also supports rendering the search tree by using its interactive search tool GIST, obtaining a result similar to the illustrated search tree in Figure 3.

2.4 The DIFFN Constraint

The DIFFN constraint requires a set of rectangles (or, in general, hyperrectangles, or orthotopes) to be mutually non-overlapping in at least one of $k$ dimensions. Two hyperrectangle objects $A$ and $B$ are non-overlapping in $k$ dimensions if they do not share a space of volume $V > 0$. The constraint has several applications, mainly related to packing and scheduling problems as well as code generation [7, 8, 9].

Gecode currently features the DIFFN constraint under the name NOOVERLAP: the latter currently supports only 2-dimensional rectangles and its propagator reasons about one pair of rectangles at a time. In Figure 4, an illustration of DIFFN is shown, illustrating a 2-dimensional placement of squares. Figure 4a illustrates violation of the constraint, where the dashed area between the squares 4 and 5 highlights the overlap. Figure 4b illustrates a feasible square placement with no overlap, using the same squares.

In 2004, Thiel showed that finding out whether a non-overlapping constraint, such as DIFFN, over a set of rectangles has a solution is NP-hard. The proof was performed by a reduction from Sequencing with release times and deadlines [10]. As a consequence, enforcing bounds or domain consistency of a propagator for DIFFN would re-
Figure 4: A visualisation of a placement of 5 squares in 2 dimensions under the DIFFN constraint. A dashed area indicates an overlap.

qure exponential time, unless P is equal to NP.

**Definition 2.7.** A rectangle $R_i = (X_i, w_i, Y_i, h_i)$ is defined by its origin on the x-axis $X_i$ and y-axis $Y_i$ as well as its width $w_i > 0$ and height $h_i > 0$.

The origin of a rectangle is defined as its lower left corner.

**Definition 2.8.** The DIFFN constraint takes several (hyper-)rectangle objects as input and constrains the objects so that they do not overlap.

### 2.5 Value Sweep Algorithm

In [1], Beldiceanu and Carlsson presented a sweep algorithm and applied it in a propagator for a NOOVERLAP (DIFF2) constraint. The algorithm was used to propagate a constraint over rectangle domains in a more efficient way compared to using pair-wise reasoning. This was done by considering the aggregation of overlaps between each rectangle $r$ and every other rectangle $q$ and pruning the values in a sweep-like fashion.

The value sweep algorithm idea is based on the notion of forbidden regions, denoting rectangular regions in which the origin of a rectangle $r$ cannot reside.

**Definition 2.9.** Given rectangles $R_a = (X_a, w_a, Y_a, h_a)$ and $R_b = (X_b, w_b, Y_b, h_b)$ that occur in a DIFFN constraint, $F_{ab} = (f_x^-, f_x^+, f_y^-, f_y^+)$ is a forbidden region (FR) of $R_a$ if for each $x \in f_x^-, f_x^+, y \in f_y^-, f_y^+$ the assignment $X_a = x$ and $Y_a = y$ causes $R_a$ and $R_b$ to overlap in the plane. $F_{ab}$ is defined by:

$$
\begin{align*}
  f_x^- &= \max(X_b) - w_a + 1 \\
  f_x^+ &= \min(X_b) + w_b - 1 \\
  f_y^- &= \max(Y_b) - h_a + 1 \\
  f_y^+ &= \min(Y_b) + h_b - 1
\end{align*}
$$

3Note that there exist variations of DIFFN that allow side-lengths of size 0.
The forbidden region $F_{ab}$ exists (is nonempty) if and only if $f_x^- \leq f_x^+ \wedge f_y^- \leq f_y^+$. Note that there exists at most one forbidden region for each ordered pair of rectangles.

**Definition 2.10.** Given rectangles $R_i = (X_i, w_i, Y_i, h_i)$ in a DIFFN constraint, an assignment $(X_i, Y_i) = (x, y)$ is said to be feasible if and only if $(x, y)$ does not belong to any forbidden region of $R_i$. An assignment to a single coordinate $X_i = x$ is said to be feasible if and only if there exists an assignment $Y_i = y$ such that $(x, y)$ does not belong to any forbidden region of $R_i$.

It follows from Definition 2.10 that a necessary condition for the DIFFN constraint is that no assignment of any rectangle belongs to any of its forbidden regions.

Beldiceanu and Carlsson’s value sweep-based algorithm reasons in terms of forbidden regions when tightening $\min(X_i)$ of a rectangle object $R_i = (X_i, w_i, Y_i, h_i)$. Since the algorithm behaves analogously for tightening the minimum and the maximum of an object, without loss of generality, the algorithm is presented in terms of tightening $\min(X_i)$. The algorithm behaves analogously in any dimension, thus tightening of $\min(Y_i)$ is not described. Omitting some details of the algorithm, the algorithm starts with a sweep line position $\Delta = \min(X_i)$: if the assignment $X_i = \Delta$ is feasible, then $\min(X_i) \leftarrow \Delta$. If $X_i = \Delta$ is not feasible, then the sweep line $\Delta$ is incremented, using inferred information from the forbidden regions $F_{ij}$, until the assignment $X_i = \Delta$ is feasible. In each iteration of the algorithm, all forbidden regions $F_{ij}$ are considered when pruning $X_i$, for all other rectangles $R_j$. If no feasible $\Delta$ exists, then the propagator reports failure.

The sweep algorithm idea was later generalised by Beldiceanu et al. in [2] into a sweep kernel, focusing on the GEOST constraint, a constraint similar to DIFFN that restricts polymorphic $k$-dimensional objects, requiring that no pair of objects overlap in a $k$-dimensional space. The majority of the ideas presented by Beldiceanu et al. are applicable to the DIFFN constraint through minor modifications, leading to a $k$-dimensional DIFFN propagator implementation based on the sweep algorithm. The extension reasons in terms of next viable position for the sweep line, or rather, in the extension, a $k$-dimensional sweep point $c$ when pruning the minimal (maximum) domain of an object’s dimension. The forbidden regions are also generalised to $k$ dimensions.

The sweep kernel also enables the integration of other constraints by representing the constraints as forbidden regions. E.g., by integrating the minimum-distance constraints that must hold between containers in the Vessel Loading problem [11], where containers are to be placed on a deck area, some containing unstable chemicals that need to be on a minimum distance from other, similar containers. One motivation behind aggregating several constraints in the kernel is that it opens up for further potential pruning, as they may share information not only through domain changes.

When pruning the domain of a $k$-dimensional object $o$ in dimension $d$, the algorithm searches for the lexicographically smallest point $^4c$ such that $c$ is not inside any forbidden region and $\forall d \in \{1, \ldots, k\} : c[d] \in D(o[d])^5$. If such a sweep point $c$ exists, then

---

^4Lexicographically smallest with regards to the dimensions $d, (d + 1) \mod k, \ldots, (d - 1) \mod k$.

^5$c[d]$ denotes $c$’s position in dimension $d$. $D(o[d])$ denotes $o$’s domain in dimension $d$. 

18
min(o[d]) ← c[d], else the algorithm reports failure. The algorithm maintains two vectors for bookkeeping. One for the sweep point c and the other for a jump vector j, storing information gained from encountered forbidden regions (What is the next possibly feasible sweep point c to try?). The latter enables more efficient search by skipping points known to be inside forbidden regions.

Example 2.3. Figure 5 illustrates the sweep algorithm performing a sweep over the x-axis when pruning the minimum of the rectangle object \( R_0 = (0..5, 4, 0..5, 3) \). Three other rectangle objects participate in the problem and no other rectangle is allowed to overlap with \( R_0 \) in the plane. The three other objects are defined as:

- \( R_1 = (1..3, 1..2, 1) \)
- \( R_2 = (1..3, 2, \{4\}, 2) \)
- \( R_3 = (3..5, 2, \{2\}, 1) \)

Three forbidden regions relative to \( R_0 \) are generated for the rectangles \( R_1, R_2, \) and \( R_3 \) and are denoted \( FR_{01}, FR_{02}, \) and \( FR_{03} \) in the figure. Step (a) shows \( R_0 \)'s relative forbidden regions. Step (b) shows the first jump taken by the sweep point c due to being in conflict with \( FR_{01} \). Step (c) illustrates c committing to a jump to arrive outside \( FR_{02} \), by doing so it also arrives outside the domain of \( R_0 \), causing the y-dimension to become exhausted. The y-coordinate of c is reset to \( R_0 \)'s minimum value as a consequence and the x-coordinate of c is assigned to the smallest (potentially) feasible x-position, based on information stored in the jump vector. Step (d) shows c escaping \( FR_{03} \). Step (e) illustrates c being in conflict with \( FR_{02} \), committing to a jump outside \( FR_{02} \), thus, once again, exhausting the y-dimension and being forced to position \( c = (3, 0) \) using information stored in the jump vector. The final step (f) illustrates c escaping \( FR_{03} \) and arriving at a feasible point. The sweep algorithm then modifies \( R_0 \) so that the minimum value of \( R_0 \)'s origin in the x-dimension is equal to 3.

3 Algorithms

This section covers the algorithms that the implementations of the DIFFN and NOOVERLAP propagators build upon. The algorithms are presented in a top-down fashion, first presenting the caller, then followed by the callee(s). The notation used is covered in Section 3.1 followed by a description of the pairwise NOOVERLAP propagator in Section 3.2. The core components making up the filtering algorithm used for pruning the domains of the variables of a hyperrectangle object are then covered in Section 3.3.

3.1 Notation

To understand the pseudocode describing the different algorithms, it is important to understand the notation. The terminology and notation used throughout this section are as follows:
Figure 5: A visualisation of the sweep algorithm for pruning the minimum in the x-dimension. The red dot represents the current sweep point $c$, while the blue point(s) illustrate the new position of the sweep point, obtained either by jumping using the jump vector or following a dimension becoming exhausted (in which case there are two blue points). The arrows highlight the direction and order the sweep point jumps to reach its new position. A highlighted forbidden region (red and thick dashed lines) indicates that the forbidden region is in conflict with the current sweep point.
The variable corresponding to the origin coordinate of a \( k \)-dimensional hyper-rectangle object \( o \) in dimension \( d \) is denoted by \( o.x[d] \). Thus, \( o.x \) contains \( k \) such variables and as a whole it represents \( o \)'s position in the \( k \)-dimensional space.

The minimum (respectively maximum) value of a hyperrectangle object \( o \)'s \( d \)-coordinate domain is denoted by \( o.x[d] \) (respectively \( o.x[d] \)).

The side-length of a \( k \)-dimensional object \( o \) in dimension \( d \) is denoted by \( o.\ell[d] \).

\( f^+ \), respectively \( f^- \), for a \( k \)-dimensional forbidden region \( f \) denotes the vector \((f_0^+, f_1^+, \ldots, f_k^+)\), respectively \((f_0^-, f_1^-, \ldots, f_k^-)\).

\( \min(v_0, v_1) \), applied to two vectors \( v_0 \) and \( v_1 \), uses component-wise \( \min \) over the components of the two vectors to create a new vector, \( \max(v_0, v_1) \) is defined analogously.

\( [x = y] \), applied to numerals \( x \) and \( y \), is a comparison operator that returns 1 (denoting the Boolean true) if \( x \) is equal to \( y \), else 0 (denoting the Boolean false).

The domain of an object \( o \) refers to the \( k \) variable domains constituting the object \( o \). An object's domain is then fixed if all its variable domains are fixed.

The Constraint keyword is used to constrain a variable by pruning infeasible values from its domain so that the condition following the keyword is satisfied. When executed, it returns a Boolean value indicating whether or not it was possible to constrain the variables: if pruning was successful, then true, else false.

Performing the append operation on an ordered collection \( C \) with an element \( e \) as parameter has the effect of adding \( e \) at the start of \( C \), while the popLast operation on \( C \) deletes the element located at the very end of the ordered collection.

3.2 Pairwise

As briefly mentioned in Section 2.4, Gecode currently features a NOOVERLAP constraint, whose propagator uses pairwise reasoning between rectangles to ensure that no rectangles overlap. This procedure is detailed in Algorithms 1, 2, 3, and 4.

The algorithm uses the concept of disjoint boxes to reason about how many rectangles a given rectangle object \( o \) can possibly overlap with. Each rectangle stores a disjoint-box counter and when said counter reaches 0 the rectangle object is removed from the collection of rectangle objects \( O \) (lines 17-20), as it can no longer overlap with any other object.

The propagator is a non-idempotent propagator and returns either NOFIX, indicating that it should be rescheduled, or SUBSUMED if it reaches subsumption. This occurs when all objects in \( O \) cannot overlap with each other, or when \( O \) contains at most one object. In both cases the NOOVERLAP constraint cannot be violated.
If two objects can overlap, then the algorithm will prevent potential overlap between the two using the `PreventOverlap` function. If pruning is needed to avoid overlap, then the `PruneOverlap` function will be used, which will remove infeasible values from the objects’ variable domains. Note that `PruneOverlap` does not only tighten the upper and lower bounds of the variables, it may also create holes inside object domains by pruning values that are infeasible inside the domain of the variables (line 8 of Algorithm 4).

```
Function NoOverlap(O, k)
    Input: Collection of objects to prune, O, and the number of dimensions, k.
    Output: An execution status with three cases: NOFIX if all objects in O have a feasible origin, SUBSUMED if all objects in O have a feasible origin and no assignment to the variable domains can cause overlap. If none of the previous are true, then FAILED is returned, indicating failure.
    n ← |O|
    d ← array of n elements  // d[i] = # of disjoint boxes for rectangle i
    e ← 0  // Number of disjoint boxes to be eliminated
    for i ∈ 0..(n − 1) do
        d[i] ← n − 1
    for i ∈ 0..(n − 1) do
        for j ∈ (i + 1)..(n − 1) do
            if CantOverlap(O[i], O[j], k) then
                d[i] ← d[i] − 1
                d[j] ← d[j] − 1
                e ← e + [d[i] = 0] + [d[j] = 0]
            else if ¬PreventOverlap(O[i], O[j], k) then
                return FAILED
    if e = n then
        return SUBSUMED  // No overlap between any boxes possible
    for i ∈ 0..(n − 1) while e > 0 do
        if d[i] = 0 then
            e ← e − 1
            O ← O \ {O[i]}
    if |O| ≤ 1 then
        return SUBSUMED  // No overlap between any boxes possible
    else
        return NOFIX
```

Algorithm 1: Pairwise toplevel.
Algorithm 1: Checks whether two rectangles cannot overlap.

Function CantOverlap(o,o',k)

Input: Rectangle objects o and o', and the number of dimensions, k.

Output: false if the two rectangles o and o' can overlap.

for i ∈ 0..(k-1) do
    if o.x[i] + o.ℓ[i] ≤ o'.x[i] ∨ o'.x[i] + o'.ℓ[i] ≤ o.x[i] then
        return true
    return false

return false

Algorithm 2: Checks whether two rectangles cannot overlap.

Function PreventOverlap(o,o',k)

Input: Rectangle objects o and o', and the number of dimensions, k.

Output: false if the two rectangles o and o' overlap for all values of their domains.

for i ∈ 0..(k-1) do
    if o.x[i] + o.ℓ[i] ≤ o'.x[i] ∨ o'.x[i] + o'.ℓ[i] ≤ o.x[i] then
        for j ∈ (i+1)..<(k-1) do
            if o.x[j] + o.ℓ[j] ≤ o'.x[j] ∨ o'.x[j] + o'.ℓ[j] ≤ o.x[j] then
                return true
        return false

if ¬PruneOverlap(o,o',k) ∨ ¬PruneOverlap(o',o,k) then
    return false
else
    return true

return false

Algorithm 3: Prevent two rectangles from overlapping.
Function PruneOverlap(o, o', k)

Input: Rectangle objects o and o', and the number of dimensions, k.

Output: false iff no feasible origin for o was attainable wrt object o'.

Side-effects: Prunes o.x and o'.x so that the objects do not overlap, if possible.

1. if o'.x[i] + o'.ℓ[i] > o.x[i] then
2.   if ¬constraint o.x[i] + o.ℓ[i] ≤ o'.x[i] then
3.     return false
4.   else if ¬constraint o'.x[i] ≥ o.x[i] + o.ℓ[i] then
5.     return false
6. else if o'.x[i] < o'.x[i] + o'.ℓ[i] then
7.   if ¬constraint o.x[i] ∉ (o'.x[i] − o.ℓ[i] + 1..o'.x[i] + o'.ℓ[i] − 1) then
8.     return false

Algorithm 4: Prunes rectangle object o with regards to rectangle o'.

3.3 Core Sweep Algorithm

The algorithms detailed in this section build upon the ideas by Beldiceanu and Carlsson from their 2007 paper [2], with minor differences such as choice of data structures, as explained further in Section 4.2.

The main component of the DIFFN propagator implementation is a filtering algorithm detailed in Algorithm 5. The filter function tightens the lower and upper bounds of every k-dimensional object o’s domain, using the PruneMin (Algorithm 6) and PruneMax (Algorithm 7) functions, such that it does not overlap with other objects. The act of running the Filter function over an object o will commonly be referred to as filtering o throughout the thesis. Filter uses a nonfix Boolean variable to detect fixpoint by concluding that if no object’s domain was pruned during iteration over all objects, then the function cannot prune anything at that point. For each object, the algorithm generates its corresponding forbidden regions relative to the other objects (line 7). The current object o is then pruned in all k dimensions, one at a time. If, for some dimension, no feasible point of origin exists, then the function fails and reports failure by returning false. Note that the two pruning functions not only return a Boolean value, but also prune the domain of o in dimension d as a side-effect, where possible. If any variable domain of o was pruned then the algorithm is re-run over all objects after the current outer iteration. The filter algorithm skips some calls to the pruning functions for objects that are fixed (assigned) in the current dimension. The filter algorithm will still detect failure as other dimensions will fail if a skipped dimension was assigned to an unfeasible value. If all dimensions of an object are fixed however, it must check if the object’s (fixed) position is feasible.

To advance the sweep point, the Adjust function is used, detailed in Algorithm 8. The procedure uses information from the jump vector j to move the sweep point c
forward, as this vector stores the lexicographically smallest (respectively largest if the upper bound is being tightened) known feasible position in dimension $r$. After committing to a jump, the jump vector is reset to its default value in dimension $r$.

To generate the forbidden regions $\mathcal{F}$, the $\text{GenOutBoxes}$ function is used. It generates forbidden regions relative to other objects and is detailed in Algorithm 9. When generating forbidden regions for an object $o$, it inspects every object $o' \neq o$ to generate a forbidden region $f$ relative to $o'$. The logic is analogous to the one of creating forbidden regions in 2 dimensions. A $k$-dimensional forbidden region is composed of $k$ intervals, each with an origin and an endpoint. A forbidden region $f$ then exists if all such intervals are non-empty and if the domain of $o$ overlaps $f$ according to the $\text{Overlaps}$ function (detailed in Algorithm 10). Note that no forbidden regions for domain holes are needed. This is due to the nature of how $o.x$ is pruned. If the sweep point is positioned in a hole of $D(o.x[d])$, for any dimension $d$, then the pruning will still be performed correctly since the new minimum (respectively maximum) value of $D(o.x[d])$ will be the first larger (respectively smaller) value $v \in D(o.x[d])$. Such a $v$ must exist if pruning occurs, as the sweep point $c$ cannot be feasible otherwise.

To obtain a forbidden region given a sweep point $c$ and a set of forbidden regions $\mathcal{F}$, the $\text{GetFR}$ function is used, detailed in Algorithm 11. This function returns a forbidden region such that $c$ is inside that forbidden region ($c$ is infeasible). The $\text{IsFeasible}$ function, seen in Algorithm 12, is used to check whether a point $c$ is feasible according to the given forbidden region $f$. 

25
Function $\text{Filter}(\mathcal{O}, k)$

**Input**: Collection of objects to prune $\mathcal{O}$ and the number of dimensions $k$.

**Output**: An execution status with three cases: **FIXPOINT** if all objects in $\mathcal{O}$ have a feasible origin, **SUBSUMED** if all objects in $\mathcal{O}$ have a feasible origin and all objects are fixed. If none of the previous are true, then **FAILED** is returned, indicating failure.

1. $\text{nonfix} \leftarrow \text{true}$
2. while $\text{nonfix}$ do
3.    $\text{nonfix} \leftarrow \text{false}$
4.    $\text{allFixed} \leftarrow \text{true}$
5.    for $o \in \mathcal{O}$ do
6.        $\mathcal{F} \leftarrow \text{GenOutBoxes}(\mathcal{O}, k, o)$
7.        if $o.x$ is assigned in all dimensions then
8.            if $|\mathcal{F}| > 0$ then
9.                return **FAILED**
10.       else
11.           for $d \in 0..(k - 1)$ do
12.             if $\neg o.x[d].\text{assigned()} \land \neg \text{PruneMin}(o, d, k, \mathcal{F})$ then
13.                 return **FAILED**
14.             else if $\neg o.x[d].\text{assigned()} \land \neg \text{PruneMax}(o, d, k, \mathcal{F})$ then
15.                 return **FAILED**
16.             else if $o.x$ was modified then
17.                 $\text{nonfix} \leftarrow \text{true}$
18.                 if $o.x$ is not assigned then
19.                     $\text{allFixed} \leftarrow \text{false}$
20.        if $\text{allFixed}$ then
21.            return **SUBSUMED**
22.        else
23.            return **FIXPOINT**

Algorithm 5: The Filter function for pruning the minimum and maximum of object domains.
Function PruneMin(o, d, k, F)

Input: The object \( o \) that is being filtered, dimension in which filtering is being performed, \( d \), number of dimensions \( k \), and forbidden regions \( F \).

Output: A Boolean \( b \), indicating failure (false) or success (true). The function returns false if no feasible minimum point is obtainable in dimension \( d \) for \( o \), true otherwise.

Side-effects: Tightens the lower bound of \( o.x[d] \) so that \( o \) and \( o' \) do not overlap, if possible.

1. \( b \leftarrow true \)
2. \( c \leftarrow o.x \) // Sweep point is initialised to the lower bound of \( o.x \)
3. \( j \leftarrow o.x + 1 \) // Jump vector is set to the upper bound of \( o.x + 1 \)
4. \( \langle \text{infeasible}, f \rangle \leftarrow \text{GetFR}(k, c, F) \)

while \( b \land \text{infeasible} \) do
5. \( j \leftarrow \min(j, f^+ + 1) \)
6. \( \langle c, j, b \rangle \leftarrow \text{Adjust}(c, j, o, d, k, true) \) // Move up \( c \) to next feasible point
7. \( \langle \text{infeasible}, f \rangle \leftarrow \text{GetFR}(k, c, F) \) // Is \( c \) still infeasible?
8. if \( b \) then
9. constraint \( o.x[d] \geq c[d] \) // Prune \( o \)
10. return \( b \)

Algorithm 6: The PruneMin function, used for pruning the lower bound of the \( d^{th} \) coordinate of a hyperrectangle object \( o \).
**Function** \texttt{PruneMax}(\(o, d, k, F\))

**Input:** The object \(o\) that is being filtered, dimension in which filtering is being performed, \(d\), number of dimensions \(k\), and forbidden regions \(F\).

**Output:** A Boolean \(b\), indicating failure (false) or success (true). The function returns \texttt{false} if no feasible maximum point is obtainable in dimension \(d\) for \(o\), \texttt{true} otherwise.

**Side-effects:** Tightens the upper bound of \(o.x[d]\) so that \(o\) and \(o'\) do not overlap, if possible.

\[
\begin{align*}
&b \leftarrow \texttt{true} \\
c \leftarrow o.x \quad \text{// Sweep point is initialised to the upper bound of } o.x \\
j \leftarrow o.x - 1 \quad \text{// Jump vector is set to the lower bound of } o.x - 1 \\
\langle \text{infeasible}, f \rangle \leftarrow \text{GetFR}(k, c, F) \\
\textbf{while } b \land \text{infeasible } \textbf{do} \\
&j \leftarrow \max(j, f - 1) \\
&\langle c, j, b \rangle \leftarrow \text{Adjust}(c, j, o, d, k, \text{false}) \quad \text{// Move up } c \text{ to next feasible point} \\
&\langle \text{infeasible}, f \rangle \leftarrow \text{GetFR}(k, c, F) \quad \text{// Is } c \text{ still infeasible?} \\
\textbf{if } b \textbf{ then} \\
&\text{constraint } o.x[d] \leq c[d] \quad \text{// Prune } o \\
\textbf{return } b
\end{align*}
\]

**Algorithm 7:** The \texttt{PruneMax} function, used for pruning the upper bound of the \(d\)th coordinate of a hyperrectangle object \(o\).
Function `Adjust(c, j, o, d, k, minimum)`

**Input**: The sweep point `c`, the jump vector `j`, the object being filtered `o`, the dimension in which filtering occurs `d`, the total number of dimensions, `k`, and whether the minimum or maximum is being tightened, `minimum`.

**Output**: A triplet consisting of the updated sweep point `c`, the reset jump vector `j`, and a Boolean, indicating whether a candidate new sweep point was found or not.

1. **if** `minimum` **then**
   2. for `i ← k − 1` **downto** 0 do
   3.     `r ← (i + d) mod k` // rotation wrt `d,k`
   4.     `c[r] ← j[r]` // Jump using jump vector `j`
   5.     `j[r] ← o.x[r] + 1` // Reset coordinate of `j`
   6.     if `c[r] ≤ o.x[r]` then
   7.         return `⟨c, j, true⟩`
   8.     else
   9.         `c[r] ← o.x[r]`
10.     return `⟨c, j, false⟩`
11. **else**
12.    for `i ← k − 1` **downto** 0 do
13.     `r ← (i + d) mod k` // rotation wrt `d,k`
15.     `j[r] ← o.x[r] − 1` // Reset coordinate of `j`
16.     if `c[r] ≥ o.x[r]` then
17.         return `⟨c, j, true⟩`
18.     else
19.         `c[r] ← o.x[r]`
20.     return `⟨c, j, false⟩`

**Algorithm 8**: The `Adjust` function moves the sweep point `c` to the next feasible position based on the jump vector `j`. 

---

29
Function GenOutBoxes(\(O, k, o\))

Input: Collection of objects \(O\), number of dimensions \(k\), and the object \(o\) whose forbidden regions are to be calculated.

Output: A set of \(k\)-dimensional forbidden regions \(F\) for the object \(o\) relative to the objects in \(O\).

\[
F \leftarrow \emptyset
\]

for \(o' \in O: o \neq o'\) do

\(
\begin{align*}
    f & \leftarrow \text{empty forbidden region} \\
    \text{exists} & \leftarrow \text{true} \\
    \text{for } d \in 0..(k-1) \text{ do} \\
    \quad \text{if } o'.x[d] - o.\ell[d] + 1 \leq o'.x[d] + o'.\ell[d] - 1 \text{ then} \\
    \quad \quad f_d^- & \leftarrow o'.x[d] - o.\ell[d] + 1 \\
    \quad \quad f_d^+ & \leftarrow o'.x[d] + o'.\ell[d] - 1 \\
    \quad \text{else} \\
    \quad \quad \text{exists} & \leftarrow \text{false} \\
    \text{if exists \& Overlaps(o, f, k) then} \\
    \quad F & \leftarrow F \cup \{f\}
\end{align*}
\)

return \(F\)

Algorithm 9: The GenOutBoxes function for generating the forbidden regions of an object \(o\), according to other objects.

Function Overlaps(\(o, f, k\))

Input: Object \(o\), forbidden region \(f\), and the number of dimensions \(k\).

Output: true iff the domain of \(o\) overlap the forbidden region \(f\).

for \(d \in 0..(k-1)\) do

\[
\begin{align*}
    \text{if } o.x[d] < f_d^- \lor o.x[d] > f_d^+ & \text{ then} \\
    \quad \text{return false}
\end{align*}
\]

return true

Algorithm 10: The Overlaps function, used for checking if the domain of object \(o\) overlaps the forbidden region \(f\).
Function $\text{GetFR}(k, c, \mathcal{F})$

Input: The number of dimensions $k$, sweep point $c$, and a collection of active forbidden regions $\mathcal{F}$.

Output: A tuple $(\text{infeasible}, f)$, where $\text{infeasible}$ is $\text{true}$ iff $c$ overlaps with the forbidden region $f$.

1. if $\exists f \in \mathcal{F} | \neg \text{IsFeasible}(f, k, c)$ then
2. return $(\text{true}, f)$
3. else
4. return $(\text{false}, _)$

Algorithm 11: The $\text{GetFR}$ function, used for obtaining a forbidden region for the according to the current sweep point $c$ and the object being pruned, $o$.

Function $\text{IsFeasible}(f, k, c)$

Input: Forbidden region $f$, number of dimensions $k$, and the sweep point $c$.

Output: $\text{true}$ iff $c$ does not overlap with the input forbidden region $f$.

1. for $j \in 0..(k - 1)$ do
2. if $c[j] < f_j^- \vee c[j] > f_j^+$ then
3. return $\text{true}$
4. return $\text{false}$

Algorithm 12: The $\text{IsFeasible}$ function, used for checking if $c$ is feasible according to the forbidden region $f$.

### 3.4 Optimised Sweep Algorithm

This section presents optimisations, some devised by myself\(^6\) and the majority devised by Beldiceanu and Carlsson in [1], that have been implemented with the goal of reducing the execution time of the implementation, mainly by reducing the total number of cycles spent during propagation.

Let bounding box $B(o)$ of a $k$-dimensional hyperrectangle object $o$ be the convex hull of all feasible instances of $o$. It is defined as

$$B(o)_d^- = o.x[d]$$

$$B(o)_d^+ = o.x[d] + o.ℓ[d] - 1$$

for dimension $0 \leq d < k$. A combined bounding box $B$ over all objects $O$ is defined as the following:

$$B_d^- = \min(B(o)_d^- | o \in O)$$

$$B_d^+ = \max(B(o)_d^+ | o \in O)$$

\(^6\)Specifically: the idea of populating the bounding box $B$ incrementally in context of the Incrementality optimisation.
**Merge.** For this optimisation, both the $\text{GenOutBoxes}$ function and the $\text{Filter}$ function were modified. The optimised $\text{GenOutBoxes}$ function is denoted by $\text{GenOutBoxesOpt}$ and can be viewed in Algorithm 13. It implements an optimisation that merges the most recently generated forbidden regions when possible. $\text{GenOutBoxesOpt}$ utilises an auxiliary function to accomplish the merging of forbidden regions, namely the $\text{Coalesce}$ function, detailed in Algorithm 14. The function merges two forbidden regions if one of them encloses the other, as illustrated in Figure 6, or if one of two forbidden regions $A$ and $B$ can be extended in some dimension to create $A \cup B$, as illustrated in Figure 7. The goal of the merge optimisation is to minimise the amount of forbidden regions having to be considered in $\text{GetFR}$.

**Separate.** During propagation the domain of some objects may become fixed and as a consequence they can be ignored during filtering. To obtain this functionality, the $\text{Filter}$ function was modified into $\text{FilterOpt}$, detailed in Algorithm 15, along with its auxiliary functions $\text{SourceCheck}$ and $\text{Disjoint}$, detailed in Algorithms 16 and 17. The optimisation involves having two collections of objects instead of a single collection. The $O$ collection is identical to the one in $\text{Filter}$, storing all objects participating in

![Figure 6: A visualisation of merging where $A$ encloses $B$.](image)

![Figure 7: A visualisation of merging where $A$ can be extended into $A \cup B$.](image)
the DIFFN constraint. The new collection $\mathcal{N}$ is used for storing non-fixed objects, i.e., objects that are not yet known to be fixed at a feasible origin. Objects in $\mathcal{O} \setminus \mathcal{N}$ are said to be checked and fixed, meaning that the domains of the objects are fixed. A checked object is an object for which the Filter function has checked its (fixed) position for overlap relative to other objects. Checked objects will still be considered when generating forbidden regions for other objects however, as other objects must still avoid them. The goal of this optimisation is to remove some redundant work of the propagator. The idea of fixed and checked objects is identical to the target property explored in \cite{1}, where fixed and checked objects in this thesis are identical to objects without the target property in \cite{1}.

Another part of the separate optimisation included in FilterOpt is identical to the optimisation denoted source in \cite{1}. Here, all objects are initially assigned a source property, indicating that they can lead to some pruning of other objects. I.e., they need to be considered during forbidden-region generation by other objects. To conclude that the source property of an object $o$ has been lost, a bounding box $B$ is first calculated over all non-fixed objects. Then, if $o$ is disjoint from $B$, then the object can be removed from $\mathcal{O}$ and need not be considered during filtering. The goal of this optimisation is to minimise the amount of objects to consider during filtering. Objects without a source property must also be fixed, implying that they can be completely ignored during filtering. This part of Separate uses the same reasoning used for disjoint boxes in the pairwise algorithm.

**Support.** The functions PruneMin and PruneMax were optimised by adding two attributes to the objects being filtered. The supportedMin (respectively supportedMax) attribute stores a previously known smallest (respectively largest) feasible $k$-dimensional point for each pruning dimension $d$. I.e., each object now stores two $k$-dimensional points; one for use in the optimised PruneMinOpt function, the supportedMin attribute, and one for use in the optimised PruneMaxOpt function, namely supportedMax. The optimised pruning functions are detailed in Algorithm 18 and Algorithm 19. The goal of this optimisation is to avoid potentially redundant work in the case that the previously feasible position of the object is still feasible.

**Incrementality.** When generating forbidden regions for an object $o$, the algorithm needs to consider all relative objects $o'$ to $o$. The formula for forbidden regions for $o$ relative to $o'$ can be rephrased from Definition 2.9 as follows:

$$
\begin{align*}
  f_d^- &\leftarrow o'.x[d] - o.ℓ[d] + 1 \\
  f_d^+ &\leftarrow o'.x[d] + o'.ℓ[d] - 1
\end{align*}
$$

The forbidden region $f$ is empty if:

$$
\begin{align*}
  o'.x[d] - o.ℓ[d] + 1 &> o'.x[d] + o'.ℓ[d] - 1
\end{align*}
$$
which can be reformulated as

\[
\overline{o'.x[d]} - o'.x[d] - o'.\ell'[d] > o.\ell[d] - 2
\]

Let \(\text{maxl}[d]\) be the maximal length in dimension \(d\) over all objects, and let:

\[
o'.dsize[d] \leftarrow \overline{o'.x[d]} - o'.x[d] - o'.\ell[d]
\]

Then, every forbidden region relative \(o'\) becomes empty if:

\[
\exists d: o'.dsize[d] > \text{maxl}[d] - 2
\]  

This knowledge can be used when generating forbidden regions by skipping objects \(o'\) whose condition (2) holds: they do not need to be considered during forbidden region generation for any other object \(o\). This part of the incrementality optimisation is found at line 3 of GenOutBoxesOpt in form of an object attribute \(\text{isSkippable}\). This attribute is initialised to true during object creation and is changed to false once the expression

\[
o'.dsize[d] > \text{maxl}[d] - 2
\]

no longer holds for any dimension \(0 \leq d < k\). This is monitored incrementally behind the curtains by the propagator.

When a hyperrectangle object is pruned it often does not affect a large portion of the other objects. Thus, when the propagator is executed, it is often redundant to filter all objects as most are likely not affected by external events, i.e., pruning performed by other propagators or via exploration. The idea behind this part of the Incrementality optimisation is inspired by the work of Beldiceanu and Carlsson [1] but does not loop through all objects to calculate the bounding box \(B\) in each invocation of the propagator. Here, \(B\) is computed incrementally and updated only when a bound of an object’s domain is modified externally, through another constraint or by exploration. The rationale is that, typically, only a few objects were pruned since the previous invocation of the propagator. Apart from utilising \(B\) to store external events, the optimised FilterOpt function also stores internal events by using another bounding box \(I\), acting as a buffer for events. It does so by copying the contents of \(B\), which is then immediately reset. This enables \(B\) to store information about internal events during the filtering process, without the internal events being drowned in external events. This can potentially avoid a lot of work for other objects that are not affected by the events. This part of the Incrementality optimisation is reflected in lines 5–6, 7, and 19 of the FilterOpt function. Here, RefreshB (detailed in Algorithm 20) is used to update the bounding box \(B\).
Function GenOutBoxesOpt(O, k, o)

Input: Collection of objects O, number of dimensions k, the object o whose forbidden regions are to be calculated.
Output: A set of k-dimensional forbidden regions \( \mathcal{F} \) for the object \( o \) relative to the objects in \( O \).

\[ \mathcal{F} \leftarrow \emptyset \]

for \( o' \in O; o \neq o' \land \neg o'.isSkippable \) do

\( f \leftarrow \) empty forbidden region

\( \text{exists} \leftarrow \) true

for \( d \in 0..(k-1) \) do

if \( o'.x[d] - o.\ell[d] + 1 \leq o'.x[d] + o'.\ell[d] - 1 \) then

\( f^- d \leftarrow o'.x[d] - o.\ell[d] + 1 \)

\( f^+ d \leftarrow o'.x[d] + o'.\ell[d] - 1 \)

else

\( \text{exists} \leftarrow \) false

if \( \text{exists} \land \text{Overlaps}(o, f, k) \) then

while \( \text{exists} \land |\mathcal{F}| > 0 \) do

\( f' \leftarrow \mathcal{F}.\text{popLast}() \)

switch Coalesce\( (f', f, k) \) do

case 0

\( \mathcal{F}.\text{append}(f') \)

\( \text{exists} \leftarrow \) false

case 1 // \( f \) subsumed

\( f \leftarrow f' \)

case 2 // \( f' \) subsumed

\( \) no-op

\( \mathcal{F}.\text{append}(f) \)

return \( \mathcal{F} \)

Algorithm 13: The optimised GenOutBoxesOpt function for generating forbidden regions of an object, supporting the merging of forbidden regions.
Function `Coalesce(A, B, k)`

Input: FR A, FR B, and number of dimensions k.

Output: Attempts to coalesce A and B. Returns 0 if no coalescing is possible, 1 if A includes B, either by extension of A in one dimensions or by subsumption. Returns 2 if B subsumes A.

```
trend ← 0
for d ∈ 0..(k - 1) do
    if $A_d^+ + 1 < B_d^- \lor A_d^- > B_d^+ + 1$ then
        return 0
    else if $A_d^- = B_d^- \land A_d^+ = B_d^+$ then
        no-op
    else if $A_d^- \leq B_d^- \land A_d^+ \geq B_d^+$ then
        if trend = 0 \lor trend = 1 then
            trend ← 1
        else
            return 0
    else if $A_d^+ \geq B_d^- \land A_d^- \leq B_d^+$ then
        if trend = 0 \lor trend = 2 then
            trend ← 2
        else
            return 0
    else
        e ← d // Which dimension is extendable?
        if trend = 0 then
            trend ← 3
        else
            return 0

switch trend do
    case 0 \lor 1
        return 1
    case 2
        return 2
    case 3
        $A_e^- ← \min(A_e^-, B_e^-)$
        $A_e^+ ← \max(A_e^+, B_e^+)$
        return 1
```

Algorithm 14: The `Coalesce` function, used for coalescing two forbidden regions, when possible.
Function FilterOpt(\(\mathcal{N}, \mathcal{O}, B, k\))

**Input**: Collection of non-fixed objects to prune \(\mathcal{N}\), collection of all objects \(\mathcal{O}\), bounding box \(B\) of all objects in \(\mathcal{N}\), and the number of dimensions \(k\).

**Output**: An execution status with three cases: **FIXPOINT** if all objects in \(\mathcal{O}\) have a feasible origin, **SUBSUMED** if all objects in \(\mathcal{O}\) have a feasible origin and all objects are fixed. If none of the previous are true, then **FAILED** is returned, indicating failure.

1. nonfix ← true
2. while nonfix do
3.     nonfix ← false
4.     \(I ← B\)
5.     \(B ← \text{region with } \infty \ldots \infty \text{ in all dimensions}\)
6.     for \(o ∈ \mathcal{N}: \neg\text{Disjoint}(I, o, k)\) do
7.         \(F ← \text{GenOutBoxesOpt}(\mathcal{O}, k, o, \max l)\)
8.         if \(o.x\) is assigned in all dimensions then
9.             if \(|F| > 0\) then
10.                return FAILED
11.         else
12.             for \(d ∈ 0..(k - 1)\) do
13.                 if \(\neg.o[x][d].\text{assigned()} \land \neg\text{PruneMinOpt}(o, d, k, F)\) then
14.                     return FAILED
15.                 else if \(\neg.o[x][d].\text{assigned()} \land \neg\text{PruneMaxOpt}(o, d, k, F)\) then
16.                     return FAILED
17.                 else if \(o.x\) was modified then
18.                     \(B ← \text{RefreshB}(B, o, k)\)
19.                     nonfix ← true
20.         end
21.     end
22. end
23. // Remove objects that have become fixed
24. for \(o ∈ \mathcal{N}: o.x\) is assigned do
25.     \(\mathcal{N} ← \mathcal{N} \setminus \{o\}\)
26. end
27. \(O ← \text{SourceCheck}(\mathcal{N}, \mathcal{O}, k)\) // Remove disjoint objects from \(\mathcal{O}\)
28. if \(|\mathcal{N}| = 0\) then
29.     return SUBSUMED
30. else
31.     return FIXPOINT

**Algorithm 15**: Optimised version of the filter function, FilterOpt.
Function SourceCheck($\mathcal{N}, \mathcal{O}, k$)

**Input**: Collection of non-fixed objects to prune $\mathcal{N}$, collection of all objects $\mathcal{O}$, and the number of dimensions $k$.

**Output**: $\mathcal{O}$ with all objects disjoint from the bounding box of all non-fixed objects removed.

if $|\mathcal{O}| > |\mathcal{N}| > 0$ then

$f \leftarrow$ region with $\infty, -\infty$ in all dimensions

for $o \in \mathcal{N}$ do

for $d \in 0..(k-1)$ do

$f_d^- \leftarrow \min(f_d^-, o.x[d])$

$f_d^+ \leftarrow \max(f_d^+, o.x[d] + o.\ell[d] - 1)$

for $o \in \mathcal{O} \setminus \mathcal{N}$: Disjoint($f, o, k$) do

$\mathcal{O} \leftarrow \mathcal{O} \setminus \{o\}$

return $\mathcal{O}$

**Algorithm 16**: The SourceCheck function – used for removing objects that have lost their source property from $\mathcal{O}$.

Function Disjoint($B, o, k$)

**Input**: Bounding box $B$, object $o$, and the number of dimensions, $k$.

**Output**: true iff $o$ is disjoint from $B$.

for $i \in 0..(k-1)$ do

if $o.x[i] + o.\ell[i] - 1 < B_i^- \lor o.x[i] > B_i^+$ then

return true

return false

**Algorithm 17**: The Disjoint function – used for checking if an object is disjoint from the bounding box of all non-fixed objects, i.e., if it has lost its source property.
Function PruneMinOpt(o, d, k, F)

Input: The Object o that is being filtered, dimension in which filtering is being performed, d, number of dimensions k, and forbidden regions F.

Output: A Boolean b, indicating failure (false) or success (true). The function returns false if no feasible minimum point is obtainable in dimension d for o, true otherwise.

Side-effects: Tightens the lower bound of o.x[d] so that o and o' do not overlap, if possible.

\[
supported \leftarrow true \quad //\text{ Assume } o.\text{supportMin}[d] \text{ is a feasible position}
\]

\[
\text{for } j \in 0..(k - 1) \text{ do}
\]

\[
\quad \text{if } o.\text{supportMin}[d][j] \notin o.x[j] \text{ then}
\]

\[
\quad \quad supported \leftarrow false
\]

\[
\quad \text{break}
\]

\[
\text{if } supported \land \neg \text{GetFR}(k, o.\text{supportMin}[d], F) \text{ then}
\]

\[
\quad \text{return true}
\]

\[
b \leftarrow true
\]

\[
c \leftarrow o.x
\]

\[
j \leftarrow o.x + 1 \quad //\text{ Jump vector is set to the upper bound of } o.x + 1
\]

\[
\langle \text{infeasible, } f \rangle \leftarrow \text{GetFR}(k, c, F)
\]

\[
\text{while } b \land \text{infeasible do}
\]

\[
j \leftarrow \min(j, f^+ + 1)
\]

\[
\langle c, j, b \rangle \leftarrow \text{Adjust}(c, j, o, d, k, true) \quad //\text{ Move up } c \text{ to next feasible point}
\]

\[
\langle \text{infeasible, } f \rangle \leftarrow \text{GetFR}(k, c, F) \quad //\text{ Is } c \text{ still infeasible?}
\]

\[
\text{if } b \text{ then}
\]

\[
\text{for } j \in 0..(k - 1) \text{ do}
\]

\[
\quad o.\text{supportMin}[d][j] \leftarrow c[j]
\]

\[
\text{constraint } o.x[d] \geq c[d] \quad //\text{ Prune } o
\]

\[
\text{return } b
\]

Algorithm 18: The optimised PruneMinOpt function for pruning the lower bound of an object.
Function PruneMaxOpt \( (o, d, k, F) \)

**Input:** The object \( o \) that is being filtered, dimension in which filtering is being performed, \( d \), number of dimensions \( k \), and forbidden regions \( F \).

**Output:** A Boolean \( b \), indicating failure (false) or success (true). The function returns false if no feasible maximum point is obtainable in dimension \( d \) for \( o \), true otherwise.

**Side-effects:** Tightens the upper bound of \( o.x[d] \) so that \( o \) and \( o' \) do not overlap, if possible.

\[
supported \leftarrow true \quad // \text{Assume } o.supportMax[d] \text{ is a feasible position}
\]

for \( j \in 0..(k - 1) \) do

\[
\text{if } o.supportMax[d][j] \notin o.x[j] \text{ then}
\]

\[
supported \leftarrow false
\]

\[
\text{break}
\]

if \( supported \land \neg \text{GetFR}(k, o.supportMax[d], F) \) then

\[
\text{return true}
\]

\[
b \leftarrow true
\]

\[
c \leftarrow o.x
\]

\[
j \leftarrow o.x - 1 \quad // \text{Jump vector is set to the lower bound of } o.x - 1
\]

\[
\langle \text{infeasible}, f \rangle \leftarrow \text{GetFR}(k, c, F)
\]

while \( b \land \text{infeasible} \) do

\[
\text{if } b \text{ then}
\]

\[
\text{for } j \in 0..(k - 1) \text{ do}
\]

\[
\text{o.supportMax}[d][j] \leftarrow c[j]
\]

\[
\text{constraint } o.x[d] \leq c[d] \quad // \text{Prune } o
\]

\[
\text{return } b
\]

Algorithm 19: The optimised PruneMaxOpt function for pruning the upper bound of an object.
Algorithm 20: The RefreshB function – used for updating the bounding box $B$ so that $B(o)$ is included in $B$.

3.5 Propagator Obligations

The sweep-based propagator does indeed fulfil the obligations of a propagator:

- **Correct.** As only values of variables that are within forbidden regions are removed, the correctness of the sweep-based propagator boils down to the correctness of the creation and usage of the forbidden regions. While no formal proof is given here (as it would be very involved), the algorithm of \texttt{genOutboxes} does indeed generate forbidden regions according to Definition 2.9. The sweep point $c$ is always outside a forbidden region, at the smallest possible lexicographic point, when pruning of the object’s domain occurs. No feasible, lexicographically smaller values than $c$ exist between any forbidden points and $c$, and as such, the correctness of the sweep propagator holds.

- **Checking.** By construction of forbidden regions, no assignment with overlapping hyperrectangles has a feasible sweep point. Thus, the \texttt{Filter} function will always report failure if given an infeasible assignment. Given an assignment (or reaching an assignment during propagation), the propagator will report subsumption if all (fixed) hyperrectangles have a feasible sweep point.

- **Contracting.** This holds vacuously as no modification operation in any of the functions adds values to a variable domain.

- **Monotonic.** We first note that forbidden regions increase in size as the bounds of variable domains become tighter. This follows from the minimum value of a forbidden region $f_{ab}$, in dimension $d$, decreasing as the upper bound of $b.x[d]$ is tightened, and the maximum value of $f_{ab}$, in dimension $d$, increasing as the lower bound of $b.x[d]$ is tightened. Let $p$ be the sweep-based propagator and assume, for contradiction, that $p$ is not monotonic.

Under the assumption of $p$ not being monotonic, there exist domains $D$ and $D'$ such that:

$$D' \subseteq D \land \neg(p(D') \subseteq p(D))$$
That is, there exists a variable \( x \) such that:

\[
D'(x) \subseteq D(x) \land \neg(p(D')(x) \subseteq p(D)(x))
\]

or equivalently:

\[
D'(x) \subseteq D(x) \land |p(D')(x) \setminus p(D)(x)| \geq 1
\]

This implies that there exists some value \( v \) that \( p \) removes from \( x \) given the domain \( D \) while preserving it in \( x \) given the domain \( D' \). However, this is impossible as it would imply that some forbidden region \( f \) is calculated strictly larger given \( D \) compared to when given \( D' \). This contradicts the definition of a forbidden region, as forbidden regions cannot decrease in size as domains grow stronger. Thus, since the assumption that \( p \) is not monotonic leads to a contradiction, we must have that \( p \) is monotonic.

- **Fixpoint and subsumption honest.** The algorithm is by design a fixpoint algorithm and is idempotent. The fixpoint loop inside the Filter function ensures that it will never report being at a fixpoint unless all hyperrectangles have been checked and no pruning occurred. Since the algorithm does not report subsumption unless an assignment has no relative forbidden regions, it cannot incorrectly report subsumption, assuming the region generation is correct.

## 4 Implementation

This section covers an implementation of a propagator for the DIFFN constraint, based on the value sweep algorithm, explained in Section 2.5 and detailed in Section 3. C++ was used for implementing the propagator since the target solver Gecode is written in C++. The implementation makes use of several Gecode features, the ones of interest in the context of this thesis are covered in this section.

The implementation is available at [www.github.com/eastlund/gecode-diffn](http://www.github.com/eastlund/gecode-diffn) and the evaluated version corresponds roughly to commit 4bad5b. The exact version used is not available, as the evaluated version of the propagator fixed the dimension during compilation, as explained further in Section 5.2.

### 4.1 Overview

The propagator implementation of DIFFN features several constructors to support multiple modelling scenarios, three of which are custom-made for specific dimensions. One constructor for one-dimensional modelling, one for two-dimensional, and one for three-dimensional modelling. The implementation also features a constructor for k-dimensional modelling. When one of the propagator’s constructors is called, the propagator is scheduled for execution and subscribes all views under bounds propagation,
implying that whenever a view’s bound is changed, the propagator is scheduled for execution. Despite the propagator using bounds propagation it is neither bounds consistent nor domain consistent as it would require exponential time to enforce, as explained in Section 2.4.

The implementation utilises Gecode’s space memory regions to allocate persistent objects such as hyperrectangle objects [5, Chapter 31], containing variable views representing its k-dimensional origin coordinate as well as its lengths, identification number and more optimisation-related attributes. Such objects including their attributes must be copied each time the solver makes a branching decision. Gecode regions are used for storing volatile objects, making them suitable for storing forbidden regions in the propagator [5, Chapter 31]. Gecode regions feature implicit deallocation. When a Gecode region is no longer in the scope of the current execution path is is destroyed and the corresponding allocated memory is freed. This implies that when the forbidden regions are no longer in scope, they are implicitly deallocated as they are allocated at roughly the same time as the region containing them is created.

Each time the propagator is executed, it runs the fixpoint loop detailed in Algorithm 15. During this loop the propagator can reach three states:

- **Failure.** If the propagator reaches failure, then it will return ES_FAILED, one of the values available among Gecode’s execution statuses, indicating to the solver that failure was reached. This occurs analogously to where it occurs in the algorithms detailed in Section 3.

- **Fixpoint.** If the propagator reaches fixpoint, then it will return ES_FIX, indicating to the solver that it has reached fixpoint, meaning that unless any domain changes, it can not prune anything further. This occurs whenever the corresponding Filter (or FilterOpt, if using the optimised version) returns FIXPOINT.

- **Subsumption.** If the propagator concludes subsumption it will return home.ES_SUBSUMED, indicating to the solver that the home space has reached subsumption. This occurs when all objects are fixed after iterating through the objects using the Filter function, without failure.

Since Gecode utilises cloning, the propagator also features a cloning function that copies the contents of all hyperrectangle objects to new, corresponding objects in a new Gecode space.

Some care has been taken to reduce the amount of allocation calls associated with the creation of relative forbidden regions of a hyperrectangle object o. In a problem with n objects, an object o can only generate at most n − 1 relative forbidden regions. Thus, one can allocate space for n − 1 forbidden regions at once, and use the memory chunk as a stack of forbidden regions, avoiding several allocation calls while potentially using more temporary memory. While memory utilisation should be kept at a minimum in copying solvers such as Gecode, it should be reiterated that forbidden regions are allocated inside Gecode regions, meaning that they will be implicitly deallocated and will not be copied during exploration.
With the goal of optimising cache locality with regards to forbidden regions, all forbidden regions are stored in place in a contiguous fashion as illustrated by Figure 8. This placement was chosen for two reasons: readability and the fact that it is very likely that all bounds of a specific forbidden region will be accessed in quick succession.

Figure 8: Illustration of 2-dimensional forbidden region storage in memory, where \(a, b, \ldots, e\) all represent forbidden regions.

Gecode advisors were used to enable the Incrementality optimisation. Advisors in Gecode are used to gain more fine-grained information regarding view changes and use said information to make decisions such as whether to schedule the propagator. They can also be used to update values incrementally as variable domains are modified by other propagators or through exploration [5, Chapter 27]. An advisor subscribes to a single variable view and advises each time the view changes by implementing an advise method that is called each time a modification of the view occurs. In the current implementation of the DIFFN constraint, the advise method mainly updates the values occurring in the context of the Incrementality optimisation and maintains the bounding box \(B\), keeping it up to date as its view changes. However, it also makes sure to only schedule the propagator whenever a bound of its view is changed, or if its view is assigned a value, i.e., becomes fixed (in which case it is also disposed). As the propagator is only concerned about the bounds of the variable views, it is redundant to schedule the propagator when other values are pruned.

4.2 Justifications

The decision to store the forbidden regions using an array despite the original paper [1] using a heap structure for storage was based on input from one of the authors, Mats Carlsson. Experiments performed by him showed that using complex data structures to store the forbidden regions were of no use for performance, and going with a simple structure such as an array proved most efficient.

The decision whether to make the propagator idempotent was done based on evaluations over the benchmark suite where the idempotent version of the propagator outperformed the non-idempotent one in the microbenchmarks, and tied with it in the macrobenchmarks. The results of the experiment can be viewed in Section 5.

5 Evaluation

This section focuses on the evaluation of the propagator implementation presented in Section 4. It aims to answer the following questions:

1. How much extra pruning does the sweep-based algorithm achieve compared to the pairwise algorithm?
2. What is the performance of the sweep-based algorithm compared to the pairwise algorithm?

3. What is the performance gain of the various optimisations applied to the sweep-based algorithm?

5.1 Notation

- **Microbenchmark**: A benchmark focusing on a single component. In this thesis: models of combinatorial optimisation problems consisting of only one constraint, specifically the DIFFN constraint.

- **Macrobenchmark**: A benchmark focusing on several components, representing real-life scenarios. In this thesis: models of combinatorial optimisation problems featuring several constraints.

5.2 Evaluation Setup

The experiments were run under Linux Mint 17.3 (64 bit) on an Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz with an 3 MB L3 cache and 8 GB of RAM. The reported elapsed time of every instance is the averaged execution time over 5 runs by measuring the run time reported by Gecode’s FlatZinc interpreter, available in the Gecode distribution. MiniZinc models were used to represent the benchmark scenarios and were executed using Gecode’s FlatZinc interpreter for Gecode version 5.0.0. MiniZinc is a solver-independent modelling language, enabling the modeller to model a problem once and run the same model using different solvers and techniques [12]. To solve a MiniZinc model, the model must first be translated to FlatZinc, an intermediate language between MiniZinc and the input language of a given solver. Each solver is required to provide a MiniZinc-to-FlatZinc front-end (interpreter), responsible for interpreting the FlatZinc model and executing it. By default in Gecode version 5.0.0, the FlatZinc interpreter features two redundant constraints\(^7\) that are posted alongside the DIFFN constraint; these redundant constraints were removed in order to reduce measurement noise while potentially reducing overall performance.

Several configurations of the DIFFN propagator implementation were considered in the evaluations, namely:

- **Basic**: The basic sweep algorithm, with no optimisations added.

- **Separate**: Basic with the Separate optimisation.

- **Support**: Basic with the Support optimisation.

- **Incrementality**: Basic with the Incrementality optimisation.

\(^7\)Redundant constraints are constraints implied by other constraints that do not remove any solutions from the search space. They can contribute a lot to propagation by removing infeasible assignments from the search space, potentially reducing the overall execution time.
• **Merge**: Basic with the **Merge** optimisation.

• **Combined**: Basic with all optimisations except Merge

While the implementation of the DIFFN propagator allows choosing the dimensionality by using a parameter post-compilation, the current implementation of NOOVERLAP in Gecode fixes the dimensionality to two dimensions during compilation. This opens up several potential compiler optimisations such as loop-unrolling but also enables fewer allocation calls for the object attributes residing in memory as the size of an object can be computed during compilation. To make the evaluation as fair as possible, the DIFFN propagator implementation was also fixed to two dimensions during compilation for the evaluation.

Gecode allows the propagator designer to define the cost of running the propagator, which the Gecode solver uses to perform scheduling decisions among the propagators. The cost of the DIFFN and NOOVERLAP was equal throughout the experiments, defined as \( O((km)^2) \), where \( m \) is the amount of rectangles and \( k \) the number of dimensions of the problem.

### 5.3 Experiments

This subsection covers the benchmark scenarios explored in this thesis and presents the performance evaluations of them. Microbenchmarks as well as macrobenchmarks are explored. Execution time was only considered for instances that finished in no less than one second under at least one configuration, as there is too much measurement noise in easier instances. Failure count is sometimes reported for easier instances however, as static search heuristics were used. Static search heuristics enable inference strength comparisons in the different macrobenchmarks and were the only things modified in the referenced MiniZinc models, unless explicitly noted otherwise.

#### 5.3.1 Microbenchmarks

The microbenchmarks examined consist of 6 problem sets, similar to the ones used by Beldiceanu and Carlsson in their evaluation but with minor changes to allow for larger problem instances, such as increasing the domain sizes and modifying the widths and heights based on the problem size. Each problem set is evaluated over \( n \in \{100, 200, 400, 800\} \) rectangles that must not overlap. Table 1 shows the specifics of the different problem sets. Set 6 is 95% ground, meaning that 95% of the rectangles are fixed before the propagator is first initialised. It was created by taking a solution to problem set 4 and restoring 5% of the fixed rectangles’ domains to their corresponding initial domains. The model can be viewed in Listing 1 in Appendix B.

The only constraint used for solving the problem sets was DIFFN.

---

8Merge was excluded due to lacklustre performance results in the measurements; it was conjectured that it would most likely damage the performance of Combined.

9The actual performance impact of this change was miniscule.
Set 1

<table>
<thead>
<tr>
<th>$X_i$</th>
<th>$Y_i$</th>
<th>$w_i$</th>
<th>$h_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$\text{rnd}(1.20)$</td>
<td>$\text{rnd}(1.20)$</td>
</tr>
</tbody>
</table>

Set 2

<table>
<thead>
<tr>
<th>$X_i$</th>
<th>$Y_i$</th>
<th>$w_i$</th>
<th>$h_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{rnd}(1.2000)$</td>
<td>$\text{rnd}(1.20)$</td>
<td>$i$</td>
<td>$\text{rnd}(1.101 - 2h_i)$</td>
</tr>
</tbody>
</table>

Set 3

<table>
<thead>
<tr>
<th>$X_i$</th>
<th>$Y_i$</th>
<th>$w_i$</th>
<th>$h_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$\text{rnd}(1.20)$</td>
<td>$\text{rnd}(1.20)$</td>
</tr>
</tbody>
</table>

Set 4

<table>
<thead>
<tr>
<th>$X_i$</th>
<th>$Y_i$</th>
<th>$w_i$</th>
<th>$h_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$\text{rnd}(1.20)$</td>
<td>$\text{rnd}(1.20)$</td>
</tr>
</tbody>
</table>

Set 5

<table>
<thead>
<tr>
<th>$X_i$</th>
<th>$Y_i$</th>
<th>$w_i$</th>
<th>$h_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$\text{rnd}(1.20)$</td>
<td>$\text{rnd}(1.20)$</td>
</tr>
</tbody>
</table>

Table 1: Rectangle $R_i$ for different problem sets.

Where:

\[
B = 1.25 \cdot \max(n, \sum_{j=1}^{n} \frac{j^2}{10^4})
\]

\[
C = 1.25 \cdot \max(h_i, \sum_{j=1}^{n} \frac{w_j \cdot h_j}{10^4})
\]

\[
f(i) = \begin{cases} 
\frac{n+3-i}{2} & \text{if odd}(i) \\
\frac{n+2+i}{2} & \text{otherwise}
\end{cases}
\]

\[
g(i) = \begin{cases} 
\frac{n+1+i}{2} & \text{if odd}(i) \\
\frac{n+2-i}{2} & \text{otherwise}
\end{cases}
\]

From the results of the microbenchmarks in Figure 9, along with the failure counts in Figure 10, it can be seen that the best configuration is Combined, utilising every optimisation excluding Merge. All configurations involving either the Separate or the Incrementality optimisation are non-surprisingly performing best, by far, on problem set 6. Problem set 6 consists of 95% ground rectangles, meaning that only 5% are non-fixed. In Separate’s case this implies that 95% of the objects can be skipped during filtering and in Incrementality’s case it implies that the bounding box of modified objects is unlikely to affect most other objects, i.e., most can be skipped during filtering.

While the Support optimisation only aids performance on some instances, it decreases performance relative to Basic on other instances. It does however aid the Combined configuration a bit in some of the instances of the microbenchmarks, as witnessed in Figure 24. Merge on the other hand is never worth using according to the results of the microbenchmarks.

The optimisation aiding scaling (in terms of number of squares) the most is the Incrementality optimisation, as is most prevalent for datasets 2 and 5 of the microbenchmarks. The Separate optimisation also aids in this aspect in some of the instances. Overall, these two optimisations are the best by far, both in scaling performance and by reaching the lowest overall execution time.
Figure 9: Microbenchmark results over 6 sets of instances of the elapsed time for each configuration. (Lower is better)
5.3.2 Rectangle Packing Problem

The Rectangle Packing Problem (RPP) is a packing problem from the MiniZinc Challenge [13] with the objective of packing $n$ 2-dimensional squares of increasing sizes $1 \times 1, 2 \times 2, \ldots, n \times n$ inside a rectangle of minimal area. RPP is sometimes used to model scheduling problems but can also be used in the design of VLSI chips, as well as circuit blocks inside processors and memory [14]. Figure 11 shows an optimal solution for $n = 5$ of RPP.

This optimisation problem can be modelled as either a COP by minimising the enclosing rectangle’s area, or a sequence of CSPs, by starting with a lower bound of the enclosing area and incrementing it following an appropriate search heuristic, implying that once a solution is found, it is also optimal. The benchmark scenario included in this thesis builds upon a CSP model, featured in the MiniZinc benchmark suite [15]. The model is measured over $n = 18, 19, 20, 21$ under two problem sets, one where the smallest $1 \times 1$ square is ignored (as it can always be placed), and one where it is considered.
when solving.

In RPP, the DIFFN constraint is used to make sure that the squares do not overlap. Apart from DIFFN, the RPP model also features two redundant CUMULATIVE constraints that make sure that the squares can fit horizontally and vertically based on their widths and heights. Further, the model features an arithmetical constraint that ensures that all rectangles are inside the enclosing rectangle and one symmetry-breaking constraint by restricting the position of the largest square. Lastly, the RPP model features an adaptation of Empty Strip Dominance from [16] posted through various arithmetical constraints.

DIFFN accounts for 8–30% of the total run time, depending on the RPP instance.

Looking at the results of the RPP benchmark in Figure 12 along with the failure counts in Tables 2 and 3, it can be witnessed that the Incrementality and Combined configurations were the best performing configurations, whereas Merge led to a performance loss, as did Support in all but one instance.

![Figure 12: Slowdown percentages for different DIFFN optimisations relative to the pairwise implementation over 4 of the instances of the RPP benchmark. (Lower is better)](image)

Table 2: Number of failures for different RPP instances when the 1-unit square is not considered.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Sweep</th>
<th>Pairwise</th>
<th>Pairwise Sweep</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>121028</td>
<td>121070</td>
<td>1.00035</td>
</tr>
<tr>
<td>19</td>
<td>172588</td>
<td>172588</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>138630</td>
<td>138630</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1382272</td>
<td>1382272</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 3: Number of failures for different RPP instances when the 1-unit square is considered.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Sweep</th>
<th>Pairwise</th>
<th>( \frac{Pairwise}{Sweep} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>275393</td>
<td>275435</td>
<td>1.00015</td>
</tr>
<tr>
<td>19</td>
<td>191557</td>
<td>191557</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>296863</td>
<td>296863</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1422923</td>
<td>1422923</td>
<td>1</td>
</tr>
</tbody>
</table>

5.3.3 Perfect Square Packing

Perfect Square Packing (PSP), illustrated in Figure 13, tackles the problem of placing \( n \) squares, all having unique integer sizes, into another bigger square. The sum of the squares’ areas is equal to the area of the enclosing square, implying that there are no unused spaces inside the enclosing square. The model, initially created by Håkan Kjellerstrand and significantly modified by Mats Carlsson, can be found in Listing 3 in Appendix B.

The benchmark utilises as few redundant constraints as possible in order to better highlight the differences between the sweep and pairwise configurations, while still keeping the problem manageable for the solver.

The PSP benchmark was executed over 9 different instances, identical, before modification, to the instances 2 . . 10 available in Bouwkamp and Duijvestijn’s paper [17]. All

Figure 13: A solution to instance 9 of the PSP problem.
9 instances consists of \( n = 22 \) squares and were modified by fixing some of the initial positions of the squares to make the problem manageable without introducing redundant constraints. The amount of squares that are fixed in each instance varies between 6 and 12 squares. Note that this modification likely introduces bias toward the \textbf{Separate} optimisation, and, by extension, the \textbf{Combined} configuration.

\textsc{DiffN} accounts for roughly 30–80% of the total run time, depending on the PSP instance.

From the results in Figure 14 along with the failure counts in Table 4 it can be witnessed that \textbf{Separate} is the best performing configuration among the sweep-based configurations, but also the best configuration overall, when comparing against the pairwise configuration as well. \textbf{Incrementality} and \textbf{Combined} also measure lower execution times in relation to the pairwise configuration while the other configurations are all outperformed by the latter. The failure counts differ between the sweep-based and pairwise configurations, where the pairwise measures up to 3.38 times more failures. When the difference of failure counts between the two decreases, so does the difference in execution time.

![Figure 14: Elapsed time (ms) for different \textsc{DiffN} optimisations and the pairwise propagator over 9 of the instances of the PSP benchmark. (Lower is better)](image-url)
Table 4: Number of failures for different PSP instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Sweep</th>
<th>Pairwise</th>
<th>Pairwise Sweep</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2638036</td>
<td>8347454</td>
<td>3.16</td>
</tr>
<tr>
<td>3</td>
<td>1103167</td>
<td>3738369</td>
<td>3.38</td>
</tr>
<tr>
<td>4</td>
<td>13060718</td>
<td>33209827</td>
<td>2.54</td>
</tr>
<tr>
<td>5</td>
<td>3395520</td>
<td>7637965</td>
<td>2.24</td>
</tr>
<tr>
<td>6</td>
<td>13583655</td>
<td>20211166</td>
<td>1.48</td>
</tr>
<tr>
<td>7</td>
<td>2557333</td>
<td>4148363</td>
<td>1.62</td>
</tr>
<tr>
<td>8</td>
<td>3172463</td>
<td>5366458</td>
<td>1.69</td>
</tr>
<tr>
<td>9</td>
<td>3429152</td>
<td>5069769</td>
<td>1.47</td>
</tr>
<tr>
<td>10</td>
<td>4953336</td>
<td>9359972</td>
<td>1.89</td>
</tr>
</tbody>
</table>

5.3.4 2D Strip Packing

The 2D Strip Packing (2DSP) problem deals with packing a list of items (2-dimensional rectangles) into a bin (strip) of fixed width and infinite height [18]. The problem is a COP with the goal of minimising the total packing height in the strip while requiring that the participating rectangles do not overlap. The model, by Mats Carlsson, can be found in Listing 2 in Appendix B.

In 2DSP, DIFFN makes sure that the items do not overlap in the strip. Just like RPP, 2DSP also features two redundant CUMULATIVE constraints, enabling early pruning of item placements that does not contain enough space to hold the items, one CUMULATIVE constraint for the heights, and one for the widths. It also features some redundant and symmetry breaking constraints.

The benchmark scenarios explored are of varying difficulty and size. The instances following the prefix NGCUT are derived from Besley’s paper [19]. The instance HT05 is the fifth instance proposed by Hopper and Turton in [20].

DIFFN accounts for about 20% of the total run time of the 2DSP instances.

The results of the 2DSP benchmark are found in Figure 15 along with their failure counts in Table 5. In three of the four included instances the optimisations do more harm than good, and lead to slowdowns in relation to both Basic and the pairwise configuration. However, in instance HT05 Incrementality yields a performance increase over both aforementioned configurations despite measuring an identical failure count.
Figure 15: Slowdown percentages for different DIFFN optimisations relative to the pairwise configuration over 4 of the instances of the 2DSP benchmark. (Lower is better)

<table>
<thead>
<tr>
<th>Instance</th>
<th>Sweep</th>
<th>Pairwise</th>
<th>Pairwise/Sweep</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGCUT02</td>
<td>56213</td>
<td>56213</td>
<td>1</td>
</tr>
<tr>
<td>NGCUT06</td>
<td>406827</td>
<td>4086116</td>
<td>1.0043</td>
</tr>
<tr>
<td>NGCUT11</td>
<td>314348</td>
<td>314348</td>
<td>1</td>
</tr>
<tr>
<td>HT05</td>
<td>110252</td>
<td>1102520</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5: Number of failures for different 2DSP instances.

5.3.5 Filter

The Filter problem, illustrated in Figure [16], is a scheduling and resource assignment problem frequently featured in the annually MiniZinc Challenge (http://www.minizinc.org/challenge). Filter is a resource-constraint scheduling benchmark that utilises high-level synthesis (HLS) to model different signal processing filters [21]. The Filter problem is a COP and the task is to schedule \( n \) different operations, each with a specific duration and dependencies (i.e., an operation might depend on the result of another operation), such that the total execution time of the constructed filter is minimised. An operation is either an addition or a multiplication and there can be no more operations of a certain type \( t \) than there are available resources for calculating \( t \) during any time slot.
Figure 16: Illustration of an optimal schedule when using 2 adders and 1 multiplier. Here, \( t_i \) represents a time slot and each directed arrow represents a dependency.

The type of filter being modelled depends on the instance. For the evaluations, only finite impulse response (FIR) filters were considered as instances for the model [22], as Gecode was able to run those instances using a timeout of 5 minutes. Other instances either did not finish under 5 minutes or were solved almost instantly. The different FIR instances consists of varying number of available resources, in this specific case the number of adders and multipliers available when constructing the filter. The number of operations to schedule is equal throughout the FIR instances, namely \( n = 23 \) operations.

The naming convention for the FIR instances is \( \text{fir}_a_m \) where \( a \) is the amount of available adders and \( m \) the amount of multipliers available at any given time slot.

The 2-dimensional \textsc{diffn} is mainly used for resource assignment constraints in the Filter problem by having each operation represented by a rectangle, consisting of the start time of the operation as its origin on the \( x \)-axis, its \( y \)-axis domain is set to \( 1 \ldots r \), where \( r \) is the amount of the corresponding resource that is available (equal to either \( a \) if the operation is an addition or \( m \) if it is a multiplication). The duration of executing the operation is assigned to its width, and a fixed 1 is assigned to its height.

Other constraints used are exclusively arithmetical constraints to ensure that dependencies are not violated and making sure that the time when the last operation finishes is equal to the objective.

\textsc{diffn} accounts for 30–75\% of the total run time of the Filter instances.

Witnessed from the results of the Filter benchmark in Figure [17] coupled with the
Table 6: Number of failures for different Filter instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Sweep</th>
<th>Pairwise</th>
<th>Pairwise/Sweep</th>
</tr>
</thead>
<tbody>
<tr>
<td>fir-1-1</td>
<td>326183</td>
<td>326183</td>
<td>1</td>
</tr>
<tr>
<td>fir-1-2</td>
<td>930628</td>
<td>930628</td>
<td>1</td>
</tr>
<tr>
<td>fir-1-3</td>
<td>930630</td>
<td>930630</td>
<td>1</td>
</tr>
<tr>
<td>fir-2-3</td>
<td>49889257</td>
<td>49889257</td>
<td>1</td>
</tr>
</tbody>
</table>

The failure counts in Table 6 is that all sweep-based configurations yield higher execution times than the pairwise configuration, throughout all instances. Separate yields the best result in three of the four instances, while Incrementality and Combined outperform Separate in the fourth instance, fir-2-3, where more resources are available when constructing the FIR filter.\(^\text{10}\) The failure counts are identical between the sweep-based and pairwise configurations.

5.4 Discussion

The results of the macrobenchmarks show that using sweep often leads to a slowdown compared to using the default pairwise algorithm of DIFFN, while the microbench-
marks suggest that the **Combined** configuration of sweep (using all optimisations excluding **Merge**) can lead to tremendous speedups over the pairwise algorithm. To investigate why the sweep-based algorithm often yields slower execution times compared to pairwise in the macrobenchmarks, the tool **perf** [23] was used to sample where time was spent during execution. Figure 18 highlights how much of the total run time for instance $n = 20$ of the RPP benchmark was spent performing copying, advising, disposing and propagating (filtering) inside the **DIFFN** propagator. From the figure it can be observed that the pairwise configuration is faster in terms of copying, but is tied with the **Combined** configuration of the sweep implementation when it comes to propagation. However, the **Combined** configuration has a lot more overhead in terms of copying but also the additional advise and dispose methods. **Incrementality** spends less time copying and advising compared to the **Combined** configuration, but spends more time propagating.

From the results of the 2DSP benchmark it is clear that the performance of the **DIFFN** optimisations is heavily instance dependent. For two of the instances of the 2DSP benchmark, NGCUT6 and NGCUT11, the optimisations do more harm than good, but for NGCUT2 and HT05, they improve performance, and **Incrementality** yields a speedup in relation to the pairwise configuration on instance HT05.

Despite the Filter instances having a bit different characteristics in terms of available resources, the different optimisations perform equally across three of the four instances. Figure 21 shows where time is spent inside the **DIFFN** propagator for the different optimisation configurations of the sweep algorithm and the corresponding **NOOVERLAP** propagator for the pairwise configuration. Not visible in the figure is that memory operations such as allocation and memory disposal took more time for the **Combined** configuration compared with the **Separate** optimisation, potentially answering the question why the former yields a higher execution time but spends less time in different components of **DIFFN**.

The performance profile of the PSP benchmark in Figure 22 looks similar to the performance profile of the Filter benchmark: the proportions of the different bars are similar between the two benchmarks, apart from the **Separate** and **Combined** optimisations, a difference that is likely explained by some objects being fixed in the instances of the PSP benchmark. Like in all of the performance profiles examined previously, the pairwise configuration spends the least time performing propagation and copying out of the different configurations. Thus, the sweep-based propagator must be able to gain an advantage from its stronger pruning in order to reach lower execution times compared to the pairwise configuration. Such is the case in the PSP benchmark, where the sweep-based algorithm requires at least 32% fewer failures relative to the pairwise configuration for the **Incrementality** optimisation to win out over it in the evaluations.

Another common trend throughout the profiling results in Figures 18, 19, 20, 21, and 22 is that **Support**, **Incrementality**, and therefore also the **Combined** configuration, spend more time copying compared to the other configurations. This is non-surprising as **Support** needs to store more data inside the objects and the **Incrementality** configuration needs to maintain a collection of advisors.
The internal functions of the sweep-based algorithm were profiled mainly in order to investigate the reason behind Merge’s lacklustre performance. The profiling results of \( n = 20 \) of the RPP benchmark are presented in Figure 23, where there is only a minor improvement in time spent in the GetFR function while the overhead of generating the forbidden regions is increased a lot when using Merge. Worth noting is also that the propagator tends to spend a relatively small amount of time in the GetFR function, that is the function that Merge aims to optimise. When investigating other instances from other benchmarks the trend is clear: GenOutboxes dominates GetFR, and the propagator often spends at least 5 times the amount of time in GenOutboxes relative to GetFR. Thus, to see any performance benefits from Merge, the optimisation would need to have minimal impact on the run time of GenOutboxes, while also decreasing the time spent in GetFR.

Figure 23 also shows that the main reason behind the performance gain of Incrementality is the reduced time spent generating forbidden regions (GenOutboxes), due to objects being skipped during filtering and as a consequence not generating relative forbidden regions.

While DIFFN does not seem to lend much extra help in terms of pruning in most of the macrobenchmarks when investigating the amount of failures in the search tree, one interesting thing does occur when investigating Figure 18 and comparing it to the slowdowns in Figure 12: DIFFN spends twice as much time propagating compared to NOOVERLAP, but the slowdown is only roughly 5%. Investigating further shows that the time spent inside the redundant CUMULATIVE propagators is reduced when using DIFFN under this instance of the RPP benchmark. When running the pairwise configuration, roughly 3800 ms were spent propagating inside the CUMULATIVE propagators while the corresponding time for the DIFFN configurations was about 3500 ms, suggesting that CUMULATIVE requires less work when used in combination with DIFFN.

The memory utilisation was also measured briefly across all the configurations over the three benchmarks RPP, 2DSP, and Filter, by sampling the memory usage during execution and reporting the highest measured amount of memory during any sample. The differences between the configurations were microscopic (≤ 1%) and sometimes counter-intuitive in the sense that Basic could sometimes allocate more memory compared to Support. This begs the question that if Gecode allocates more memory than it uses, then the differences cannot be observed using this method. One thing that can be concluded however is that no configuration yields any spikes in memory in relation to another configuration.
Figure 18: Stacked percentages of where time is spent in the DIFFN propagator under the different optimisations for the instance $n = 20$ of the RPP benchmark, when considering the 1-unit square.

Figure 19: Stacked percentages of where time is spent in the DIFFN propagator under the different optimisations for the instance NGCUT11 of the 2DSP benchmark.
Figure 20: Stacked percentages of where time is spent in the DIFn propagator under the different optimisations for the instance HT05 of the 2DSP benchmark.

Figure 21: Stacked percentages of where time is spent in the DIFn propagator under the different optimisations for the instance fir-1-3 of the Filter benchmark.
Figure 22: Stacked percentages of where time is spent in the DIFFN propagator under the different optimisations for the instance 9 of the PSP benchmark.

Figure 23: Stacked percentages of where time is spent during the propagation phase of the sweep-based DIFFN propagator over different optimisations for the instance $n = 20$ of the RPP benchmark, when considering the 1-unit square.
5.5 Findings

This subsection discusses and answers the research questions posted in the preface of this section using the results from the measurements.

How much extra pruning does the sweep-based algorithm achieve compared to the pairwise algorithm? The sweep-based algorithm achieves significantly more pruning compared to the pairwise algorithm in benchmarks where the DIFFN constraint is used exclusively or when only a few other constraints are involved. In the microbenchmarks the former is able to find a solution without encountering failures across all instances while the pairwise algorithm can have up to 61421 failures on some. The sweep-based algorithm also yields up to 3.38 times fewer failures in the PSP benchmark, involving only a few lightweight constraints apart from the DIFFN constraint.

When other heavyweight constraints, in particular CUMULATIVE, apart from DIFFN enter the picture the results change, and the additional pruning observed in the benchmarks mentioned previously is no longer visible, as if the impact were dwarfed by those other constraints.

What is the performance of the sweep-based algorithm compared to the pairwise algorithm? If the number of failures is equal or similar for some benchmark instance, then the pairwise algorithm often yields a lower execution time compared to the sweep-based algorithm in real applications, where the best configuration of the sweep-based algorithm, Combined, sees a slowdown up to 60%, varying between benchmarks and instances. The sweep-based algorithm is superior in the microbenchmarks however, where it has performance increases up to 4000% over the pairwise algorithm.

Overall the sweep-based algorithm is dependent on achieving stronger inference relative to the pairwise algorithm in order to have a lower execution time. Such is the case in the PSP benchmark results, where the improvement in execution time is most likely a consequence of the notable difference in number of nodes explored.

What is the performance gain of the various optimisations applied to the sweep-based algorithm?

- **Separate** is one of the most efficient optimisations as it improved performance a lot for the microbenchmark instances and for the Filter benchmark. The optimisation also improved performance slightly in some of the RPP instances but decreased performance in the 2DSP benchmark.

- **Support**: While often decreasing performance relative to the basic configuration in the macrobenchmarks and for some of the instances of the microbenchmarks, Support seems to aid the performance of the combined configuration according to the microbenchmark results. The results of the macrobenchmarks, while not presented, show no notable difference in performance between including and excluding the Support optimisation from the combined configuration.
- **Incrementality** is the most efficient optimisation, as it outperformed **Separate** with a slight margin when compiling the results of all benchmarks. This optimisation improved the performance across all benchmarks except the 2DSP benchmark. It is likely the most efficient optimisation in the PSP benchmark, despite **Separate** measuring a lower time, since the instances are biased towards the latter.

- **Merge** is the worst performing optimisation, as it performed worse than the basic sweep-algorithm on all instances of all benchmarks, mostly due to the time spent generating forbidden regions dominating the time spent retrieving infeasible forbidden regions.

### 6 Conclusion and Future Work

This section provides a summary of the results and suggests future work.

#### 6.1 Conclusion

This thesis implements and evaluates a sweep-based algorithm for a **DIFFN** propagator in Gecode. The results show that the implementation outperforms its 2-dimensional counterpart **NOOVERLAP**, currently available in Gecode, in benchmarks where **DIFFN** dominates the propagation phase. In 3 of the 4 macrobenchmarks explored, other participating constraints dwarfs the difference in inference strength between the two algorithms. As a consequence, the sweep-based algorithm yields a performance decrease relative to the pairwise algorithm due to additional overhead involved in enabling the former’s more complex logic.

Different optimisations were evaluated for the sweep-based algorithm and the best ones are **Separate** and **Incrementality**. The former enables fixed objects to be ignored and removes objects that can no longer overlap other objects. The latter incrementally book-keeps variables used to infer if a specific object was possibly affected by a recent pruning (performed by either another propagator or through exploration, or by the sweep algorithm itself), this enables some objects to be ignored during propagation of the **DIFFN** constraint.

#### 6.2 Future Work

Currently, the implementation of the **DIFFN** propagator does not support defining the rectangle sizes as variables as opposed to static values and as such it does not support some modelling approaches. Future work involves implementing this functionality and evaluating the propagator’s performance when using variable lengths, using the Cargo and Carpet Cutting problems from the MiniZinc Challenge [24][25], for instance.

---

\[^{11}\text{It was never examined whether **Separate** outperforms **Incrementality** in the PSP benchmark when not initially fixing the position of some rectangles as the modified instances did not finish under 10 minutes of execution time.}\]
The thesis concludes that other constraints, such as CUMULATIVE, tend to assist the NOOVERLAP propagator by efficiently pruning values that the pairwise algorithm cannot on its own detect as infeasible. It would be interesting to find what characteristics inside the problems cause the sweep-based algorithm to be exclusively able to prune some of the values. Such characteristics are likely to exist following that the sweep-based algorithm is able to prune more, albeit very few, values in RPP and 2DSP.

Some experiments were made on modifying the cost of the DIFFN propagator during the thesis project’s time span, isolated to a few instances of the RPP benchmark and only a few variations of the cost. Although no improvement of the execution time was seen when modifying the cost under these instances, more rigorous experiments of different cost configurations could perhaps yield other results.

Some of the optimisations evaluated in Section 5 contain several parts that could be partitioned into a unique optimisation. This would enable more fine-grained comparisons between the optimisations and could potentially lead to a better Combined configuration. Future work involves evaluating the individual parts of the optimisations and combining the best-performing components into a Combined configuration.

The Resource-Constrained Project Scheduling Problem (RCPSP) is a scheduling problem similar to the Filter problem explored in this thesis, featuring resource, time, and dependency constraints over a number of tasks [26]. While the resource constraints usually manifest themselves as CUMULATIVE constraints in most models of the problem, one could modify a model so that the resource constraints are modelled using DIFFN, as done in the Filter problem. Hundreds of instances of varying sizes exist for the problem online [27] and future work involves modelling RCPSP using DIFFN and using it to evaluate how well the sweep-based configuration scales compared to the pairwise configuration as the number of tasks increases.

The DIFFN propagator was only evaluated using problems in a 2-dimensional space. It would be interesting to examine the propagator’s performance under one dimension and three dimensions as well. The latter could be evaluated by modelling the Conway packing problem [28, Section 3.7.64] and generating appropriate instances.

Commonly in top-performing models for combinatorial optimisation problems a dynamic search heuristic is used rather than a static one, as dynamic heuristics tend to outperform the latter. In 2015 Cauwelaert et al. proposed a method to measure the potential of propagators and preserving fairness of the comparisons under any chosen search heuristic [29]. Future work involves examining if this approach can be applied when comparing propagators for the DIFFN constraint and if applicable, using it to evaluate the potential of the sweep-based propagator compared to the pairwise NOOVERLAP propagator.
Appendices

A Miscellaneous Benchmark Results

Figure 24: Microbenchmark results over 6 sets of instances for speedups relative to the pairwise configuration for the Combined configuration as well as the Combined configuration without the Support optimisation. (Higher is better)
## MiniZinc Models

### Listing 1: The MiniZinc model of the microbenchmark

```plaintext
% Author: Mats Carlsson

% Micro benchmarks for Diffn aka. NoOverlap.
% Slightly generalized from "Sweep as a generic pruning technique
% applied to the non-overlapping rectangles constraint", N Beldiceanu, M

% Set 1: wi = rand(1..20), hi = rand(1..20), Xi in 1..10000, Yi in
% 1..(101-hi)

% Set 2: wi = rand(1..20), hi = rand(1..20), Xi in rand(1..2000)..10000, Yi
% in rand(1..101-2hi)..<(101-hi)

% Set 3: wi = i, hi = i, Xi in 1..10000, Yi in 1..(B-i)
% where
% B = 1.25 * max(N,sum(j in 1..N)(j^2)/10000)

% Set 4: wi = f(i), hi = g(i), Xi in 1..10000, Yi in 1..(B-i)
% where
% B = 1.25 * max(max(j in 1..N)(hj),sum(j in 1..N)(wj*hj)/10000)

% Set 5: wi = hi = 10000/sqrt(N), Xi in 1..10000, Yi in 1..10000

% Set 6 is a solution to Set 4 with 5% of the rectangles restored to their
% domains in Set 4

% All datasets are sorted by decreasing area

include "globals.mzn";

array[int,int] of int: table1;
array[int,int] of int: table2;
array[int,int] of int: table3;
array[int,int] of int: table4;
array[int,int] of int: table5;
array[int,int] of int: table6;

int: N = max(index_set_1of2(table1));
int: key;
array[1..N] of var int: X;
array[1..N] of var int: Y;
constraint
  let {array[int,int] of int: thetable =
    if key=1 then table1
    else if key=2 then table2
```
else if key=3 then table3
else if key=4 then table4
else if key=5 then table5
else table6
endif endif endif endif endif}
in forall(i in 1..N)(
let {int: xmin = thetable[i,1],
int: xmax = thetable[i,2],
int: ymin = thetable[i,4],
int: ymax = thetable[i,5]}
in X[i] in xmin..xmax /
Y[i] in ymin..ymax
) /

diffn(X, Y, [thetable[i,3] | i in 1..N], [thetable[i,6] | i in 1..N])
);
solve :: int_search([if k=1 then X[i] else Y[i] endif | i in 1..N, k in 1..2], input_order, indomain_split, complete)
satisfy;
output [show(X)++show(Y)];

Listing 2: The MiniZinc model of 2DSP problem
% domination
constraint
define(r in rectangle)(
  member([0] ++ [x[s] + width[s] | s in rectangle where s != r], x[r]) /
  member([0] ++ [y[s] + height[s] | s in rectangle where s != r], y[r])
);
%
% symmetry breaking
constraint
let {int: o1 = order[1], int: o2 = order[2]} in
  lex_less([y[o1], x[o1]], [y[o2], x[o2]]);
%
% symmetry breaking
constraint
forall(r in rectangle)(
  let {array[int] of int: I = [i | i in rectangle where width[i]=width[r] /
    height[i]=height[r]]}
  in if length(I)>1 /
    min(I)=r then
    forall(i in index_set(I) where i>1)(
      lex_less([y[I[i-1]], x[I[i-1]]], [y[I[i]], x[I[i]]])
    )
    else true endif
);
%
solve ::
int_search([strip_height], input_order, indomain_min, complete)
:: seq_search([
  int_search([y[r] | r in order], input_order, indomain_min, complete),
  int_search([x[r] | r in order], input_order, indomain_min, complete),
  int_search([strip_height], input_order, indomain_min, complete),
])
minimize strip_height;

Listing 3: The MiniZinc model of PSP problem
16 set of int: rectangle = index_set(a);
17 array[int] of int: order = sort_by(rectangle, [-a[r] | r in rectangle]);
19 constraint
diffn(x,y,a,a)
;
24 constraint
% keep the squares inside the big square
forall(i in 1..size) {
26 x[i] + a[i] <= n+1 /
28 y[i] + a[i] <= n+1
}

32 % symmetry breaking
constraint
x[size] <= y[size] /
35 x[size] <= n-x[size]-a[size]+2 /
36 y[size] <= n-y[size]-a[size]+2;

38 array[1..n] of int: lh_table;
40 int: mingap(int: sz) =
41 min([gap | gap in 1..n where lh_table[gap]-(if gap=sz then sz else 0
44 endif)>=sz]);

50 % prevent holes near the borders using Longest Hole info
constraint
forall(i in 1..size)(
56 not(x[i]-1 in 1..mingap(a[i])-1) /
57 not(n-a[i]+1-x[i] in 1..mingap(a[i])-1) /
58 not(y[i]-1 in 1..mingap(a[i])-1) /
59 not(n-a[i]+1-y[i] in 1..mingap(a[i])-1)
);

63 array[int, 1..3] of int: seed;

69 solve :: seq_search([int_search([x[r] | r in order], input_order,
67 intdomain_min, complete),
71 int_search([y[r] | r in order], input_order,
75 intdomain_min, complete)])
77 satisfy;
References


