Treewidth and Indexicals: Applying Results in Tractability to Propagators

Hans Koberg
Abstract

**Treewidth and Indexicals: Applying Results in Tractability to Propagators**

*Hans Koberg*

I simplify a small part of tractability research in the field of time complexity in constraint programming. Treewidth and strong k-consistency can be combined in order to achieve tractability. The result is partially applied to Extended Indexicals by implementing treewidth in Extended Indexicals. Preliminary tests of the treewidth of the most common constraints have been carried out and the result is promising with the majority of the constraints having a low treewidth.
Contents

1 Introduction 4

2 Preliminaries and theoretical work 5
   2.1 Constraint programming introduction 5
   2.2 Complexity of constraint programming 6
   2.3 Treewidth 7
   2.4 Consistency 10
   2.5 Treewidth and consistency 11
   2.6 Extended Indexicals 12

3 Constructing a constraint graph in Extended Indexicals 13
   3.1 Syntax of Extended Indexicals 14
   3.2 Extended Indexicals internals 15
   3.3 Preprocessing 16
   3.4 Solution 16
   3.5 Limitations to the constraint graph 17

4 Computing treewidth 18

5 Results 18

6 Discussion 22

7 Future work 23

8 Conclusions 24

9 Acknowledgements 24
1 Introduction

The idea behind Constraint programming [1,2] is to solve combinatorial search problems efficiently. This is done by modelling a combinatorial search problem as a constraint satisfaction problem. A constraint satisfaction problem is specified by a set of variables, a domain and a set of constraints. Every variable has to be assigned a value from the domain such that every constraint is satisfied.

One example of a constraint is the \texttt{AllDifferent}(x_1,...,x_n) constraint where the constraint variables $x_1$ up to $x_n$ have to be assigned distinct values. Even with the simple \texttt{AllDifferent} constraint, real problems can be solved. One excellent example is the sudoku puzzle. The rules in a $9 \times 9$ sudoku are that every row, every column and each of the nine $3 \times 3$ sub-grids is assigned unique values in the range $1 \ldots 9$. A sudoku can therefore be modelled as a constraint satisfaction problem with the \texttt{AllDifferent} constraint on every row, every column and every $3 \times 3$ sub-grid.

To solve the sudoku with constraint programming, one would specify the usage of 81 variables, the domain $1 \ldots 9$ and the 27 \texttt{AllDifferent} constraints with their respective variables. A constraint solver [1] would then do the rest. A constraint solver is a programming library or system where one can model constraint satisfaction problems and automatically get a solution given enough time and space, if a solution exists. The constraint solver’s job is to prune the domain of every variable until one or no solution is found. The pruning is done both with propagation (logical elimination) and search. The logical elimination is carried out by something called propagators. Propagators are efficient implementations of constraints, and every constraint can have one or more propagators. When constraint propagation is exhausted or too much time is consumed the constraint solver turns to searching in order to find a solution. The search is interleaved by propagation in order to get a more efficient solver.

Since the birth of constraint programming there has been much research [1,3,4,5,6] in the area of computational complexity of constraint programming in order to improve performance. The class of all constraint satisfaction problems is NP-complete [2,7]. This means that all known algorithms for solving general constraint satisfaction problems take exponential time in the size of the input in the worst case. Because the class of all constraint satisfaction problems is NP-complete, there is a very low possibility of finding a general algorithm that can solve all constraint satisfaction problems efficiently. Research has therefore been focused on finding special cases that are guaranteed to be solved in polynomial time and cases that improve the exponential running time. One of those areas is the area of treewidth [8], where a considerable amount of research has been conducted [5,6,9,10].

Treewidth is a measure of how far a graph is from being a tree [8], or, in other words, treewidth measures the degree of cyclicity of a graph [11]. A constraint satisfaction problem can be associated with a graph where each variable is a vertex and each constraint an edge. The treewidth of the graph can then be computed and results in tractability be applied. For example, if the treewidth of a class of constraint satisfaction problems is bounded, then any constraint
satisfaction problem from that class can be solved in polynomial time [12].

When constraints are implemented, they are implemented in a specific solver, making it difficult to share constraint implementations between solvers. The implementation is often both time consuming and difficult to do efficiently. Extended Indexicals [13] is a recent research project where some of the above problems have been addressed. The Extended Indexicals language is a higher-level solver language that makes writing and prototyping constraints much faster. The key feature is that propagators for specific solvers are generated automatically from the higher-level language. The research suggests that Extended Indexicals might not be significantly slower than a traditional solver.

The goal of this thesis is to start the work of combining the Extended Indexicals with treewidth in order to make the Extended Indexicals implementation faster. To do this, the theoretical foundation of treewidth and tractability using treewidth is presented. This thesis will implement treewidth in the Extended Indexicals project for future use and give some preliminary results of treewidth in the Extended Indexicals project.

2 Preliminaries and theoretical work

In the following subsections all necessary theory to understand constraint programming, the complexity of constraint programming, treewidth, consistency, treewidth and consistency combined, and the Extended Indexicals project is presented.

2.1 Constraint programming introduction

A constraint satisfaction problem is specified by a set of variables, a set of values and a set of constraints.

Definition 2.1 [1, 3]. A constraint satisfaction problem $P$ is defined as a triple $P = (V, D, C)$

where

- $V$ is a finite set of variables, $V = \{v_1, \ldots, v_n\}$
- $D$ is a finite set of values, $D = \{d_1, \ldots, d_t\}$ such that each $v_i \in D$
- $C$ is a finite set of constraints, $C = \{R_1(S_1), \ldots, R_m(S_m)\}$

so that in each constraint $R_i(S_i)$

- $S_i$ is an ordered list of $k_i$ variables, called the constraint scope
- $R_i$ is a set of tuples over $D$, each of size $k_i$

where $1 \leq k_i \leq n$
The tuples in $R_i$ are the allowed values for the variables in $S_i$, in the constraint $R_i(S_i)$. $D$ is called the domain of $P$.

**Example 1.** Let a constraint satisfaction problem $P = (V, D, C)$ be defined as $V = \{v_1, v_2, v_3\}$, $D = \{1, 2, 3\}$ and $C = \{R_1(v_1, v_2), R_2(v_2, v_3), R_3(v_3, v_1)\}$ where $R_1 = R_2 = R_3 = \{(1,1), (2,2), (3,3)\}$, commonly known as the Equal (=) constraint.

Note that in the definition above of a constraint satisfaction problem, every variable will have the same initial domain. If this is not desirable, constraints can be added that limit the domain of a specific variable. In practice, in a constraint $R_i(S_i)$, where $S_i$ is of size $k_i$, the relation $R_i$ is often not a set of tuples but instead a Boolean function. The Boolean function has $k_i$ arguments and determines if an assignment of the variables satisfies the constraint $R_i(S_i)$.

**Definition 2.2**[3]. A solution to a constraint satisfaction problem $P = (V, D, C)$ is an assignment of the variables in $V$ with values from the domain $D$, such that all constraints are satisfied.

Formally, a solution is a map $f : V \to D$ such that $f(S_i) \in R_i$, for all $i$, where $f(S_i)$ is applied coordinate-wise. That is, if $S_i = (v_1, \ldots, v_k)$, then $f(S_i) = (f(v_1), \ldots, f(v_k))$ and a solution thus satisfies $(f(v_1), \ldots, f(v_k)) \in R_i$, for all $i$.

**Example 2.** Let $P = (V, D, C)$ be defined as in Example 1. One solution to the problem is $f(v_1) = 1$, $f(v_2) = 1$ and $f(v_3) = 1$ since $(1,1) \in R_1$, $(1,1) \in R_2$ and $(1,1) \in R_3$. The set of all solutions is \{\{(1,1,1), (2,2,2), (3,3,3)\}\}

Many constraints already exist, but there is no limit to what sort of constraints can be created. However, if a constraint that is not yet implemented is needed, there are currently two solutions to the problem: the first solution is to remodel the problem to use only existing constraints, but this can often not be done without loss of generality and efficiency [1]; the second option is to implement the constraint, but it is often both hard and time consuming to implement a constraint efficiently [13].

Extended Indexicals [13] lets the user specify a constraint in a higher-level language and get efficient constraints automatically.

### 2.2 Complexity of constraint programming

The class of all constraint satisfaction problems is NP-complete [2 7]. Informally, in order to show that a problem is NP-complete, one has to show that the problem is in NP, the class of problems where a solution can be checked in polynomial time, and that one existing NP-complete problem can be reduced to the problem in polynomial time. The reduction is done to show that the problem is at least as hard as the NP-complete problem [14].

A constraint satisfaction problem is in NP since for every constraint, checking whether it is satisfied or not means comparing the solution to every tuple in $R_i$, of allowed values, which can clearly be done in polynomial time in the size of
each constraint, where the size is the number of tuples in $R_i$. Checking if all constraint variables are in the domain can also be done in polynomial time by comparing the assigned value to all values in the domain. If $R_i$ instead is a Boolean function then it has to execute in polynomial time in order for us to show that a constraint satisfaction problem is in NP.

The reduction can be done from the existing NP-complete problem 3-SAT [15]. 3-SAT is the problem of determining if there exists a Boolean assignment such that a logic formula in conjunctive normal form where each clause is of at most size three, is true. For example $(x \lor y \lor z) \land (\neg x \lor y)$ is in conjunctive normal form. 3-SAT can be reduced to a constraint satisfaction problem in the following way: Let each variable in 3-SAT become a variable in the constraint satisfaction problem and let each clause be a constraint where $n$ is the size of the clause and $L_i$ the $i$th literal in the clause. The constraint is:

$$\sum_{i=1}^{n} L_i \geq 1 \text{ where } L_i = \begin{cases} L_i & \text{if } L_i \text{ is a positive literal} \\ 1 - L_i & \text{if } L_i \text{ is a negated literal} \end{cases}$$

The transformation can clearly be done in polynomial time in the size of the 3-SAT since each clause represents a new constraint that will not depend on any other constraint.

**Example 3.** Let $(x \lor y \lor z) \land (\neg x \lor y)$ be a 3-SAT problem. Transforming it into a constraint satisfaction problem $P = (V, D, C)$ can be done in the following way: let $V = \{x, y, z\}$, $D = \{0, 1\}$ and $C = \{R_1(x, y, z), R_2(x, y)\}$ where $R_1$ is the constraint $x + y + z \geq 1$ and $R_2$ is the constraint $1 - x + y \geq 1$ and formally $R_1 = \{(1, 1, 1), (0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)\}$ and $R_2 = \{(1, 1), (0, 1), (0, 0)\}$.

Specific instances of a constraint satisfaction problem or specific classes of constraint satisfaction problems can be solved in polynomial time. These problems and classes are called tractable problems and classes.

There has been much research [1, 3, 4, 5, 6] on properties that make classes of problems tractable. These properties can roughly be divided into two fields, structural conditions and semantic conditions.

Structural conditions restrict how constraints can be combined together, for example when the constraints are represented with a graph and the graph has certain properties. One example is when the resulting graph is a tree, then the constraint satisfaction problem is tractable [3].

Semantic conditions restrict which constraints and domains can be used [16]. For example, deciding if a graph is 3-colourable is NP-complete while deciding if a graph is 2-colourable is tractable [1].

This thesis is about the first class, structural conditions, and more precisely about treewidth, discussed next.

### 2.3 Treewidth

Treewidth is a measure of how far a graph is from being a tree [8], or, in other words, treewidth measures the degree of cyclicity of a graph [11]. In order to
define treewidth, graphs and constraint graphs have to be defined first.

**Definition 2.3** [2]. A graph is a pair \( G = (V, E) \), where \( V = \{v_1, \ldots, v_n\} \) is a set of vertices, and \( E \subseteq \{(u, v) \mid u, v \in V\} \) is a set of edges.

A hypergraph is a pair \( H = (V, E) \), where \( V = \{v_1, \ldots, v_n\} \) is a set of vertices, and \( E = \{e_1, \ldots, e_k\} \) is a set of hyperedges where each hyperedge \( e_i \subseteq V \). A tree is a graph with no cycles.

See Figure 1 for an example of a hypergraph.

**Definition 2.4** [1]. A constraint graph is the hypergraph \( H = (N, E) \) of a constraint satisfaction problem \( P = (V, D, C) \), with vertices \( N \) being the set of variables \( V \) and for each \( R_i(S_i) \in C \), let \( S_i \in E \).

We assume that all constraint graphs are connected and hence also that all variables in a constraint satisfaction problem participate in some hyperedge. If a constraint graph is not connected, several constraint satisfaction problems can be constructed that are independent of each other.

**Example 4.** Let a constraint satisfaction problem \( P = (V, D, C) \) be defined as \( V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\} \), \( C = \{R_1(v_1, v_2), R_2(v_2, v_3), R_3(v_3, v_4), R_4(v_4, v_5), R_5(v_5, v_1), R_6(v_5, v_6, v_7)\} \) where each \( R_i \) is any constraint, and let \( D \) be any domain. The constraint graph of \( P \) can be seen in Figure 1.

Before defining treewidth a more intuitive upper bound on treewidth is defined in **Definition 2.5** and **Theorem 1** while treewidth is defined in **Definition 2.6**.

**Definition 2.5** [17, 9]. Cops and robber is a game played on a graph. There are \( k + 1 \) cops trying to catch a robber who wins if the cops cannot catch him in sufficient time. The robber can move arbitrarily fast along the edges as long as the vertices are not occupied by a cop. The cops move by flying with helicopters between any vertices but as soon as a vertex is not occupied, the robber can move to it before any cop can land on it, if a path without a cop to the vertex exists. Both the cops and the robber can see each other at all time and the robber can always switch vertex before a cop can land on it.

**Theorem 1** [17, 9]. A graph has treewidth at most \( k \) if and only if \( k + 1 \) cops have a winning strategy in cops and robber.
Example 5. Using the cops and robber upper bound, one can show that the treewidth of a tree is at most 1 and that the treewidth of a cycle is at most 2. One can also show that the treewidth of an $n \times n$ grid is at most $n$; Place $n$ cops on the $n$ vertices in column one and the last cop on a vertex in column two. The cop to the left of the cop in column two is now free to move. Move it somewhere on the second column and repeat this strategy until all vertices on the second column are occupied. The cops can repeat this strategy until they capture the robber on column $n$.

Definition 2.6 [10]. A tree decomposition of a graph $G$ is a tree $T$ where every vertex of $T$ is a subset of $G$’s vertices and the tree has to satisfy the following three conditions:

- Every vertex of $G$ is in at least one set of $T$
- For every edge $(e_1, e_2)$ of $G$, the set $\{e_1, e_2\}$ is in at least one set of $T$
- For every set $x_i$ and $x_j$ of $T$, if both contain the same vertex of $G$, then the sets on the path between $x_i$ and $x_j$ must also contain the same vertex

The width of a tree decomposition is the size of its largest set minus one. The treewidth of a graph is the minimum width of all possible tree decompositions. So, for instance, the treewidth of a tree is 1.

Constructing a tree decomposition and computing the treewidth is not a trivial task and is in fact an NP-hard problem [10], but many problems with bounded treewidth can be solved efficiently [10].

Example 6. A graph can have many different tree decompositions, but only a few are of minimal width. Both Figure 3 and Figure 4 are valid tree decompositions of the graph of Figure 2, but only the later one is a minimal tree decomposition, of width 2. Moving the rightmost vertex $\{v_5, v_6, v_7\}$ of Figure 4 to be attached to the upper-left vertex instead would also yield a minimum tree decomposition.

\[9\]
A tree decomposition can also be seen as a winning strategy in cops and robber. The cops start at all vertices in a set of the tree decomposition. The cops can then advance to the next set in the direction of the robber in the tree decomposition. Because the tree decomposition is a tree there will always be a path to the robber [10].

Observant readers may have noticed that the treewidth only is defined for graphs, not for hypergraphs. This can be solved with the use of primal graphs.

**Definition 2.7** [10]. The primal (Gaifman) graph of a hypergraph \( H = (V, E) \) is a graph \( G = (N, A) \) where \( N = V \) and for every hyperedge \( e_i \in E \), we have that \( \{(u, v) \mid u, v \in e_i, u \neq v\} \in A \). In other words, all variables in the same edge are connected pairwise.

The primal graph, also called constraint graph or primal constraint graph in the literature [2], is a graph and hence treewidth is defined. Figure 2 is a primal graph of Figure 1.

### 2.4 Consistency

Informally, consistency is a property of constraint satisfaction problems where the idea is to assign all possible values from the domain to all possible subsets of variables and then remove assignments that conflict with the constraints. Consistency is different from constraint propagation since consistency is done within and between constraints, as opposed to propagators which only prune within constraints.

**Definition 2.8** [12]. A partial solution to a constraint satisfaction problem \( P = (V, D, C) \) is a partial map \( f : V' \rightarrow D \) where \( V' \subseteq V \) and for each \( R_i(S_i) \in C \) if \( S_i \subseteq V' \) then \( f(S_i) \in R_i \).

In other words, only the constraints where every variable is assigned a value have to be satisfied. Also note that a partial solution does not necessarily extend to a full solution.

**Example 7.** Let a constraint satisfaction problem \( P = (V, D, C) \) be defined as \( V = \{v_1, v_2, v_3, v_4, v_5\} \), \( D = \{1, 2, 3\} \) and \( C = \{R_1(v_1, v_2), R_2(v_2, v_3), R_3(v_3, v_4), R_4(v_4, v_5)\} \) where \( R_1 = R_2 = R_3 = R_4 = \{(1, 1), (2, 2), (3, 3)\} \).

One partial solution is \( f(v_1) = 3, f(v_2) = 3 \) and \( f(v_4) = 1 \) since \( R_3 \) does not need to be satisfied since not every variable of \( R_3 \) is assigned a value. All solutions are \( \{(1, 1, 1, 1, 1), (2, 2, 2, 2, 2), (3, 3, 3, 3, 3)\} \).

**Definition 2.9** [12]. Let \( g : V_g \rightarrow D \) and \( f : V_f \rightarrow D \) be partial solutions of a constraint satisfaction problem \( P \). We say that \( g \) extends \( f \) if \( V_f \subseteq V_g \) and \( f(V_f) = g(V_f) \).

**Example 8.** The partial solution \( g(v_1) = 3, g(v_2) = 3, g(v_4) = 1 \) and \( g(v_5) = 1 \) extends the partial solution in Example 7.
Definition 2.10 [12]. A constraint satisfaction problem \( P = (V, D, C) \) is \( k \)-consistent if for every partial solution on \( k - 1 \) variables \( (v_1, \ldots, v_{k-1}) \), and for every variable \( v_k \notin \{v_1, \ldots, v_{k-1}\} \), there exists a partial solution on \( k \) variables \( (v_1, \ldots, v_{k-1}, v_k) \) extending the partial solution on \( k - 1 \) variables.

Example 9. The constraint satisfaction problem of Example 7 is 2-consistent since every partial solution on one variable can be extended to another variable. The constraint satisfaction problem of Example 7 is not 3-consistent since if \( f(v_1) = 1 \) and \( f(v_3) = 2 \) then \( f(v_2) \) has no solution.

Definition 2.11 [12]. A constraint satisfaction problem is strongly \( k \)-consistent if it is \( i \)-consistent for every \( i \in \{1 \ldots k\} \).

Example 10. Example 7 is strongly 2-consistent since it is both 1-consistent and 2-consistent.

Definition 2.12 [3]. A constraint satisfaction problem is globally consistent if any partial solution can be extended to a solution.

Example 11. Consider the constraint satisfaction problem of Example 7 with the extra constraints \( R_5(v_1, v_3), R_6(v_1, v_4), R_7(v_1, v_5), R_8(v_2, v_4), R_9(v_2, v_5), R_{10}(v_3, v_5) \) where \( R_5 = R_6 = R_7 = R_8 = R_9 = R_{10} = \{(1, 1), (2, 2), (3, 3)\} \). The new constraint satisfaction problem is strongly 5-consistent. By adding more constraints some of the previous partial solutions are not partial solutions any more because they do not satisfy all the new constraints. All solutions are still \{\( (1, 1, 1, 1, 1), (2, 2, 2, 2, 2), (3, 3, 3, 3, 3) \)\}.

A constraint satisfaction problem will most likely not have any level of \( k \)-consistency. The assignments that prevent the problem from having a certain level of consistency then have to be removed in order to achieve the consistency. Removing the conflicting assignments is easy for \( k = 1, 2 \) since it is just a matter of pruning domains. When \( k \geq 3 \) that means a partial solution on \( k - 1 \) variables should be extended to a \( k \)th variable. If this is not possible, the assignment of those \( k - 1 \) variables should be removed. This often means that a constraint has to be added that explicitly removes just that assignment.

Theorem 2 [2]. A globally consistent constraint satisfaction problem can be solved in time linear in the size of the problem, where the size of a problem is defined as the number of variables plus the size of the constraints as given by the tuples of allowed variables.

2.5 Treewidth and consistency

Treewidth and consistency can be combined to form a class of tractable problems.

Theorem 3 [12]. Let a constraint satisfaction problem have a treewidth of at most \( k - 1 \) and be strongly \( k \)-consistent, then the constraint satisfaction problem is globally consistent.
If a constraint satisfaction problem satisfies the conditions of Theorem 3, then it also satisfies those of Theorem 2 and the constraint satisfaction problem can be solved in time linear in the size of the problem.

**Example 12.** Take the graph in Figure 2 with arbitrary constraints but assume the constraint satisfaction problem is strongly 3-consistent. This graph has a treewidth of 2 and thus Theorem 3 is applicable. One can show that the constraint satisfaction problem is globally consistent. For example: Assign \( v_6 \) a value, then assign \( v_7 \) a value; this can be done since the problem is 2-consistent and for every value of \( v_6 \) there is a value for \( v_7 \). Assign \( v_5 \) a value; this can be done since the problem is 3-consistent. Both \( v_1 \) and \( v_4 \) can be assigned a value since the problem is 2-consistent. Variable \( v_3 \) can then be assigned a value for the same reason. Variable \( v_2 \) can at last be assigned a value since the problem is 3-consistent, because for every assignment of \( v_1 \) and \( v_3 \), variable \( v_2 \) has at least one possible assignment.

### 2.6 Extended Indexicals

The Extended Indexicals language [13] is a high-level language for writing and prototyping propagators. These propagators can then automatically be transformed into solver-specific propagators. This makes sharing propagators between different solvers much easier since the propagator only has to be written once. The higher-level language makes prototyping and writing propagators faster.

The Extended Indexicals language is based on the previous work on Indexicals [18] but has been extended to a higher level with many features of a modern programming language. Some of the features are constraint arrays of arbitrary size and operations on such arrays.

The drawback with a more general and higher-level language is worse execution time of propagators compiled with Extended Indexicals compared to built-in propagators [18]. The initial research however suggests that propagators from Extended Indexicals might not be significantly slower than built-in language-specific propagators.

The currently supported target solvers are Gecode [19] and OscaR [20]. The Extended Indexicals compiler is written in Java.

The Extended Indexicals language should be distinguished from MiniZinc [21] and similar languages where the purpose is to state problem instances in a higher-level language whereas the Extended Indexicals state and implement propagators in a higher-level language.

**Example 13 [22].** Here is how the constraint AllDifferent can be written in Extended Indexicals:

```plaintext
def Alldiff(vint[] X){
  desc("all variables take a different value")
  prop(decomp){
    forall (i in rng(X)){
```
forall(j in rng(X)){
    once(i<j){
        X[i] in U minus {val(X[j])} ;
        X[j] in U minus {val(X[i])} ;
    }
}
}
}
}
}
}

The code should be read as: For each unique index pair i and j in the array X, let X[i] not be any value of the domain of X[j] and let X[j] not be any value of the domain of X[i]. Especially note the use of an arbitrary-sized array as input, vint[] X, and its usage, i in rng(X) and j in rng(X), where “rng” stands for index range.

3 Constructing a constraint graph in Extended Indexicals

The first step in applying the concept of treewidth to Extended Indexicals is to compute the treewidth. In order to do this, a constraint graph has to be constructed. A constraint graph is normally constructed from a constraint satisfaction problem instance where everything is known. In Extended Indexicals this is not the case, since Extended Indexicals implement propagators, that are not instance-specific. This means that the constraint graph cannot be created as usual but needs an alternative interpretation.

The only difference between an instance and a non-instance is formed by the input arguments. In Extended Indexicals these are regular constraint variables together with the possibility of constraint-variable arrays of unknown size: see Example 13 again. There are two problems with this: the first one is that the domains of the constraint variables are unknown and the second one is that the sizes of the arrays are unknown.

There are three possible approaches to this problem. The first approach is to derive the exact treewidth as a function of the input arguments, both from the possible constraint-variable arrays of unknown size and the domains of the input arguments. The second approach is to derive only the exact treewidth as a function of the constraint-variable arrays sizes. The third approach is to assign the possible constraint-variable arrays a few different sizes and derive a likely treewidth.

The first approach requires a complete proof of the treewidth as a function of the arguments, the second one requires only a complete function of the argument sizes, and the last one requires a very rough estimate of the treewidth. Both the first and second approaches require substantial work in order to produce a function that with certainty can provide the treewidth for any arguments, if even possible. The third approach is simple enough to implement and is the
approach chosen. It will probably give a good result in most cases with only a small time cost.

3.1 Syntax of Extended Indexicals

```
CSTR ::= def CNAME(ARG)( PROPAG+ CHECKER?)
PROPAG ::= propagator(PNAME?){ INSTR* }
CHECKER ::= checker { BOOL }
INSTR ::= VAR in SET ; | post(CINVOKES,PNAME?); | fail; | once(BOOL){ INSTR* } | forall(ID in SET){ INSTR* }
SET ::= univ | emptyset | ID | INT..INT | rng(ID) | dom(VAR) | NSETOP(ID in SET)(SET) | -SET | SET BSETOP SET |
{INT+} | {ID in SET:BOOL}
INT ::= inf | sup | NUM | ID | card(SET) | min(SET) | max(SET) | min(VAR) | max(VAR) | val(VAR) | ~INT | INT BINTOP INT | b2i(BOOL) | NINTOP(ID in SET)(INT)
BOOL ::= true | false | ID | INT INTCOMP INT | INT memberOf SET | SET SETCOMP SET | not BOOL | BOOL BBOOLOP BOOL | NBOOLOP(ID in SET)(BOOL) | entailed(CINVOKES) | satisfiable(CINVOKES) | check(CINVOKES)
BINTOP ::= + | - | * | / | mod
NINTOP ::= sum | min | max
BSETOP ::= union | inter | minus | + | - | * | / | mod
NSETOP ::= union | inter | sum
INTCOMP ::= = | != | <= | < | >= | >
SETCOMP ::= = | subseteq
BBOOLOP ::= and | or | =
NBOOLOP ::= and | or
CINVOKES ::= CNAME | CNAME(ARG)
```

Figure 5: A BNF-like grammar of Extended Indexicals [13]. Starting at the top, an Extended Indexical is defined with a name, one or more propagators, and a checker. The checker is a function that tests whether the constraint is satisfied or not. An instruction is an Indexical, posting another propagator, failing the propagator, an if statement (once), or a for loop (forall). The rest of the instructions are modifying and creating integers, sets and Booleans. The rules for ARGs (list of arguments), CNAME (current propagator name), PNAME (propagator name), ID (constant name), VAR (variable name) and NUM (integer literal) are shown.
A BNF-like grammar of the Extended Indexicals language can be found in Figure 5 [13]. An almost full definition and semantics of the language can be found in the program’s user manual, which is available on demand by the Extended Indexicals author [13].

An Indexical is an expression of the form $X \text{ in } \sigma$, where $X$ is a constraint variable and $\sigma$ a set. This is the basic expression in Indexicals and means that the new domain of $X$ is the intersection of the old domain of $X$ and $\sigma$. The Indexical expression is the only way to manipulate the domain of constraint variables.

Example 14. Let $X$, $Y$ and $Z$ be three constraint variables and consider the following lines of code to be in the Extended Indexicals language:

- $X \text{ in } 1..10$
- $Y \text{ in } 8..16$
- $Y \text{ in } X$
- $Z \text{ in } 0..\min(X)$

which means that $X$ should be in the intersection of its old values and the range 1..10. $Y$ should be in the intersection of its old values and the range 8..16. $Y$ should be in the intersection of its old values and the current values of $X$. Lastly, $Z$ should be in the intersection of its old values and the range 0 to the minimum value of $X$.

The constraint graph was previously only defined on constraint satisfaction problem instances but can easily be used on Extended Indexicals: let each Indexical expression be seen as a single constraint. Imagine that a whole propagator in Extended Indexicals is the constraint satisfaction problem instance and that each Indexical expression is a constraint. When creating a constraint graph, the left-hand side of $\text{in}$, the constraint variable, is connected by an edge with every constraint variable on the right-hand side, if any. The variables on the right-hand side, if several, are not connected with each other since they only constrain the left-hand side, not each other.

Example 15. The constraint graph of the code in Example [14] is:

![Constraint Graph]

3.2 Extended Indexicals internals

Behind the scenes of Extended Indexicals is a fairly complicated compiler with a lot of source code; Extended Indexicals is after all a programming language. However, only a small part of the source code needs to be tapped into in order to compute the constraint graph.
When the Extended Indexicals compiler reads in new constraints they are parsed into an abstract syntax tree where various modifications and simplifications can be done easily. The abstract syntax tree in Extended Indexicals is of the type \textit{Node}. \textit{Node} is a basic tree structure for representing an abstract syntax tree with five important fields. The fields are \textit{Type}, \textit{Form}, \textit{SubForm}, \textit{Text} and \textit{Children}. The \textit{Type} indicates which Extended Indexicals data type the \textit{Node} is of, the \textit{Form} indicates what the \textit{Node} is in a more general sense, the \textit{SubForm} specifies exactly what the \textit{Node} is, the \textit{Text} is the text representation of the \textit{Node} and \textit{Children} is a list of \textit{Nodes} of all the children of \textit{Nodes}.

The abstract syntax tree is an excellent part to work on to generate a constraint graph since all constraints are represented internally with the abstract syntax tree and all internal work is performed on the abstract syntax tree. This also means that some basic methods for modifying the abstract syntax tree exist. Since Extended Indexicals is a programming language there is quite an extensive collection of simplification rules that simplify the abstract syntax tree. Everything from variable substitution to more simple logical simplifications and integer simplifications exists. This means that expressions that can be simplified are simplified automatically; for example Boolean statements are evaluated to true or false if possible.

3.3 Preprocessing

To make the construction of the constraint graphs easier some preprocessing is done. First a copy of each constraint is done not to mess up the original constraints and each constraint with more than one propagator is split into separate constraints since two propagators for the same constraint can have different constraint graphs.

To avoid iterating \texttt{forall} loops and other \texttt{n}-ary loop operations, such as \texttt{sum}, \texttt{min} and \texttt{union}, when creating the constraint graph, they are unfolded into explicit expressions with the loop index variables replaced.

Since the plan is to evaluate each constraint with an array argument with a few different sizes of the array, it can from now on be assumed that the array sizes are fixed to a value. The point of knowing the array size is to be able to replace the call to the \texttt{rng()} operator of an array with the actual length of the array. Replacing the \texttt{rng} makes almost all \texttt{forall} loops possible to unfold.

3.4 Solution

After preprocessing is done, the abstract syntax tree is ready to be traversed in order to construct a constraint graph per constraint. This is done by traversing the abstract syntax tree and executing the appropriate action on each \textit{Node}. Since the constraint graph is constructed from the expressions of the form \texttt{X in \sigma} and only those expressions, traversing each \textit{Node} is only a way of finding those expressions. The Extended Indexicals language is fairly complex and

\footnote{An abstract syntax tree is source code represented as a tree.}
constructing the constraint graph is not as straightforward as described above. Every type of Node (syntax) needs a rule of how to interpret it as a constraint graph. The most important syntax and cases are explained below.

Sometimes there are expressions that cannot be evaluated; for example forall (i in dom(X)), where the domain of the constraint variable X is used. Since the domains of all constraint variables are unknown no constraint graph can be made.

A once statement can sometimes not be evaluated to either true or false. This will make it impossible to compute a constraint graph for the whole constraint. There is however an alternative solution that is used in this thesis: Always assume that the expression is true if it is undecidable. Assuming such a thing will possibly give a slightly larger constraint graph, depending on the once actually is true or false.

The post statement, which works just like a function call and calls another propagator, is possibly inlined by the Extended Indexicals compiler. If a post cannot be inlined for some reason, then it is assumed to have the maximal constraint graph, with all variables connected to all variables.

In the expression X in σ, the set expression σ can be quite complicated but since the goal is to find the constraint variables of the expression, the meaning of σ is not important. Only the constraint variables are important.

Most other syntax is not really interesting; either all child nodes are visited or the whole node can be skipped.

3.5 Limitations to the constraint graph

Since the Extended Indexicals language is rather complex, there are some limitations to the constraint graph.

No constraint graph is constructed for constraints that have more than one array, since it is too complicated to get a comprehensible result.

In the case where an array is accessed by an unknown index, for example Y in X[i] after simplifications, then no constraint graph is made.

The keyword failed is used to indicate that the constraint cannot be solved but is ignored when creating a constraint graph. This might not be the best solution.

If the post statement does not get inlined by the compiler and has at least one array argument of unknown size, then no constraint graph is constructed.

No special care has been taken about local variable declarations, since Extended Indexicals should simplify these expressions.

Complex expressions like once(\min(y)>10)\{x \in 3..5\} have not been considered due to the complexity and time limitations. The variables x and y should be connected in the constraint graph since once is an implication constraint. However, once is mostly used to filter simple forall expressions where no extra edge should be added anyway.

Example 16. The constraint graph of the Alldiff constraint in Example 15 is shown for input sizes 4 and 8 in Figures 6 and 7 respectively, where each edge
4 Computing treewidth

To compute the actual treewidth, the libTW library \cite{23} is used. It is, at the time of writing this thesis, the only Java implementation available for use. There are faster implementations in other languages but since everything in Extended Indexicals is done in pre-processing there is no need for the fastest implementation. It has a lot of lower and upper-bound algorithms implemented in order to decrease the running time, along with two exact algorithms on treewidth. Exact details and suggested algorithms to use can be found in \cite{23}.

Since libTW is used, only some piping between the constraint graph and libTW has to be done. LibTW primarily reads from files but a tiny modification lets it read from strings instead. The format is very simple, namely “\textit{a b}” separated by a line break, where \textit{a} and \textit{b} are vertices in a graph and where \textit{e} is a token indicating an edge. libTW cannot handle self-loops in graphs.

5 Results

The results of this thesis are both theoretical and practical.

The result of the theoretical work is the structuring and simplifying of available papers. This thesis gives a theoretical introduction to constraint programming, the complexity of constraint programming, treewidth, consistency, and finally treewidth and consistency combined.

The practical results of this thesis are that the Extended Indexicals compiler can compute the treewidth of most constraints that are fed into it. Every constraint can consist of several propagators. In that case, one new constraint is made for each propagator since the treewidth can be different for each propagator. The result is divided into three groups: constraints with zero, one or more arrays of constraint variables. In the first group the treewidth is reported directly. In the second group, where one array is used, the treewidth is reported for the sizes 2, 4, 8 and 16 of the constraint array. In the last group no treewidth
is reported. If the treewidth for some reason cannot be computed, an error is shown. A treewidth of 0 means that the treewidth could be computed and was 0; for example the empty graph or the graph with no edges.

Some preliminary tests of treewidth have been carried out on the file `/examples/CP2012/all_cstrs_CP2012.idx` that is included with Extended Indexicals. The file contains 47 constraints with 73 propagators in total. The constraints are of various types, including the most common constraints. Computing the treewidth of the file results in 31 propagators with no array as argument, 34 propagators with one array as argument and 8 propagators with more than one array as argument. The result for the propagators with no array arguments is shown in Table 1. The result for the propagators with one array argument is shown in Table 2. The propagators with two or more arrays as arguments where no treewidth was obtained are listed in Table 3. An example of a constraint with two propagators that are of different treewidth will be seen in Example 17.

Using four array sizes (2, 4, 8 and 16) for the constraints in `/examples/CP2012/all_cstrs_CP2012.idx` and computing their treewidths takes below one hundred milliseconds per propagator on an Intel Core i3 350M. It should be noted that the majority of the time consumed is from the Extended Indexicals compiler preprocessing and not from computing the constraint graph or treewidth.
<table>
<thead>
<tr>
<th>Constraint Name</th>
<th>Treewidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0</td>
</tr>
<tr>
<td>EXACTLY_IND</td>
<td>0</td>
</tr>
<tr>
<td>IMPLY</td>
<td>0</td>
</tr>
<tr>
<td>INSET</td>
<td>0</td>
</tr>
<tr>
<td>IsTrans</td>
<td>0</td>
</tr>
<tr>
<td>IsUniqueTrans</td>
<td>0</td>
</tr>
<tr>
<td>NOT</td>
<td>0</td>
</tr>
<tr>
<td>NOTINSET</td>
<td>0</td>
</tr>
<tr>
<td>NotTrans</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>0</td>
</tr>
<tr>
<td>REIFY</td>
<td>0</td>
</tr>
<tr>
<td>Trans</td>
<td>0</td>
</tr>
<tr>
<td>ABS_prop_AC</td>
<td>1</td>
</tr>
<tr>
<td>ABS_prop_BC</td>
<td>1</td>
</tr>
<tr>
<td>ABS_prop_FC</td>
<td>1</td>
</tr>
<tr>
<td>DIST_prop_BC</td>
<td>1</td>
</tr>
<tr>
<td>EQ_prop_AC</td>
<td>1</td>
</tr>
<tr>
<td>EQ_prop_BC</td>
<td>1</td>
</tr>
<tr>
<td>GEQ</td>
<td>1</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>LEQ</td>
<td>1</td>
</tr>
<tr>
<td>LT</td>
<td>1</td>
</tr>
<tr>
<td>MAX</td>
<td>1</td>
</tr>
<tr>
<td>NEQ</td>
<td>1</td>
</tr>
<tr>
<td>Reif_EQ</td>
<td>1</td>
</tr>
<tr>
<td>DIST_prop_AC</td>
<td>2</td>
</tr>
<tr>
<td>DIST_prop_decomp</td>
<td>2</td>
</tr>
<tr>
<td>PLUS_prop_AC</td>
<td>2</td>
</tr>
<tr>
<td>PLUS_prop_BC</td>
<td>2</td>
</tr>
<tr>
<td>PLUS_prop_FC</td>
<td>2</td>
</tr>
<tr>
<td>PLUSLEQ</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: Constraint name and corresponding over-approximated treewidth of constraints with no array as argument from file `all_cstrs_CP2012.idx`. The suffix `_prop_name` means that a constraint has several propagators where name is the propagator name.
<table>
<thead>
<tr>
<th>Constraint Name</th>
<th>size 2</th>
<th>size 4</th>
<th>size 8</th>
<th>size 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Contiguity_prop_</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Global Contiguity_prop_v2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ith_POS_DIFF_ZERO_prop_veryshort</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SUM_prop_checking</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AMONG_prop_</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AMONG_prop_meta</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AMONG_prop_withPost</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>COUNT</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>EXACTLY_prop_</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>EXACTLY_prop_meta</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>INCREASING_prop_local</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>INCREASING_prop_decomp</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ITH</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MAXIMUM_prop_AC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MAXIMUM_prop_BC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NON_DECREASING_prop_local</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NON_DECREASING_prop_decomp</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SOME_EQ_prop_</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SOME_EQ_prop_reshave</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHANGE</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>EXACTLYSEQ</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INCR_NVALUE_prop_</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INCR_NVALUE_prop_meta</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INCR_NVALUE_prop_withPost</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INCR_NVALUE_prop_withPost2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SEQ_BIN</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Alldiff</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>INCREASING_prop_global</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>NON_DECREASING_prop_global</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>SUM_prop_AC</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>SUM_prop_BC</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>ELEMENT</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FIRST</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Ith_POS_DIFF_ZERO_prop_</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2: Constraint name and corresponding over-approximated treewidth of constraints with arrays of size 2, 4, 8 and 16 as argument from file all_cstrs_CP2012.idx. Here '-' means that no treewidth could be computed for some reason. The suffix _prop_name means that a constraint has several propagators where name is the propagator name.
Table 3: Constraint name of constraints with two or more arrays as argument from file all_cstrs_CP2012.idx. No treewidth (-) was computed due to the difficulty and complexity of handling two arrays at the same time. The suffix _prop_name means that a constraint has several propagators where name is the propagator name.

Example 17 \[22\]. Constraint INCREASING with the two propagators local and global (INCREASING_prop_local respectively INCREASING_prop_global in Table 3) and their constraint graphs in Figure 8 and Figure 9 for argument size 4:

```python
def INCREASING(vint[] X):
    desc("X is in strictly increasing order.")
    prop(local){
        forall(i in {i in rng(X):i<max(rng(X)))}{
            X[i] in inf .. (max(X[i+1]) - 1);
            X[(i+1)] in (min(X[i]) + 1) .. sup;
        }
    }
    prop(global){
        forall(i in {i in rng(X):i<max(rng(X)))}{
            X[i] in inf .. min(j in {j in {j in rng(X):j<max(rng(X)):j>=i}((max(X[(j+1)]) - ((j+1) - i))));
            X[(i+1)] in max(j in {j in {j in rng(X):j<max(rng(X)):j<=i}((min(X[j]) + ((i+1)-j))) .. sup;
        }
    }
}
```

6 Discussion

The theoretical results are quite modest but provide a stable groundwork for future work.
The initial results of treewidth are promising since most of the constraints have a low treewidth and the treewidth cannot be computed only for a few constraints. However, as mentioned for the limitations of the constraint graph, there are a few problems with the constraint graph that make the results somewhat unreliable in some cases. For example the case with once where if it is undecidable, it is assumed to be true. A warning should be shown in such a case so one can know which constraint graphs and treewidths can be trusted and which are over-approximations. The post statement should be handled the same way.

It is also quite difficult to ensure that the creation of the constraint graph is correct since the syntax of Extended Indexicals is quite advanced.

There is a clear pattern in all cases with one array argument where the treewidth is either 0, 1, 2, the size of the array argument minus one 1, or the size of the array argument. A conclusion of the treewidth of each constraint could probably be done with great confidence.

The three constraints with one array argument that cannot be computed use access of the type \texttt{for i in domain of Y, do X[i]}. Since the domain is unknown the treewidth cannot be computed.

The eight constraints with two or more arrays as argument leaves little to discuss since no result was attempted to be obtained.

7 Future work

The first thing that should be addressed are the various problems involved in creating the constraint graph. A few improvements and reflections about the constraint graph should be carried out in order to ensure that the constraint graph is correctly computed. A good improvement would be to inform the user.
when the treewidth may not be correct, as once for cases that are assumed to be true. There may also be other situations that this thesis has not mentioned as a problem due to the advanced syntax of Extended Indexicals. The eight constraints with two or more arrays as argument can also be tried with a few different array sizes just to see if there exists a pattern in the treewidth there too.

The theoretical work also needs some future work. Theorem 3 can be proved with a much shorter proof; at the moment the proof is somewhat spread out and hidden between the lines in the referenced article.

The single biggest future work direction is the theoretical and practical work of improving the Extended Indexicals compiler with the help of treewidth.

8 Conclusions

This thesis has provided a groundwork for improving the Extended Indexicals with Treewidth. Unfortunately, for time reasons, no work was done to actually improve the Extended Indexicals compiler, nor was a plan written on how it could be done concretely. The majority of the investigated constraints have a low treewidth, which makes further research in improving Extended Indexicals with the help of treewidth promising. It should be noted that the correctness of the treewidth is difficult to check. Only future research can tell if Extended Indexicals can be improved and how much.

9 Acknowledgements

I want to thank Justin Pearson for taking his time to be the supervisor of this thesis. I want to thank the author of Extended Indexicals, Jean-Noël Monette, for sharing his knowledge about the Extended Indexicals and answering questions about the Extended Indexicals. I want to thank Pierre Flener for reviewing this thesis. Lastly I want to thank Jonne Mickelin Sätherblom for correcting the language and typography of this thesis.

References


