

Oxiflex - A Constraint Programming Solver for MiniZinc written in Rust

Bachelor's thesis

University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence

Examiner: Prof. Dr. Malte Helmert
Supervisor: Simon Dold

Gianluca Klimmer
gianluca.klimmer@stud.unibas.ch
2019-915-594

15. July 2024

Abstract

Constraint Satisfaction Problems (CSPs) are typical NP-complete combinatorial problems in the field of Artificial Intelligence. As part of this thesis we introduce Oxiflex, a CSP solver written from scratch in Rust. Oxiflex is built on the MiniZinc toolchain and supports a subset of FlatZinc constraint builtins. Starting with a naive backtracking approach, we enhance Oxiflex by applying variable ordering and inference. Both forward checking and arc consistency enforcing algorithms using AC-1 and AC-3 are used for inference. Results show that variable ordering and forward checking have a positive impact on time measurements but AC-1 and AC-3 do not. However by measuring number of iterations results show that AC-1 and AC-3 can significantly reduce the number of iterations needed for backtracking. This work shows that inference works, but careful implementation is needed to make it fast.

Table of Contents

1	Introduction	1
2	Constraint Satisfaction Problems	3
2.1	Overview	3
2.2	MiniZinc	4
2.2.1	FlatZinc	5
2.3	Queens Problem	7
3	Solving Constraint Satisfaction Problems	9
3.1	Backtracking	9
3.1.1	Variable Ordering	10
3.2	Inference	11
3.2.1	Forward Checking	12
3.2.2	Arc Consistency	13
3.2.2.1	Enforcing Arc Consistency	13
4	Implementation	15
4.1	Oxiflex	15
4.2	Rust	15
4.2.1	Limitations	15
4.3	Dependencies	16
4.3.1	flatzinc	16
4.3.2	structopt	16
4.4	Architecture	16
4.4.1	parser	16
4.4.2	model	16
4.5	Solver	17
4.5.1	Value Ordering	17
4.5.2	Forward Checking	18
4.5.3	Arc Consistency	18
5	Results	19
5.1	Method	19
5.2	N-Queens	20

Table of Contents	iv
5.3 Slow Convergence	21
5.4 Gecode	23
6 Conclusion	26
6.1 Discussion	26
6.2 Future work	27
Bibliography	28

1

Introduction

Constraint Satisfaction Problems (CSPs) are typical NP-complete combinatorial problems in the field of Artificial Intelligence, characterized by a set of variables and constraints that define the relationships between these variables. Our main goal of this thesis is the development of a CSP solver entirely from scratch.

In Chapter 2 we will discuss how we can write down CSPs in both a formal and an useful way. For the latter we will be using MiniZinc [NSB⁺07], a CSP modeling language developed at and by Monash University in Australia. MiniZinc is a toolchain with tools for users that want to solve CSPs and tools for solvers. The main idea of MiniZinc is to translate to a simpler language called FlatZinc that solvers are able to use to solve the problem. MiniZinc is only the language and does not provide a solver. This makes a MiniZinc problem independent from the solver and makes it really easy to give the same problem to multiple solvers. At the of this chapter we will showcase an example problem domain called N-Queens as an easy to understand CSP that will also be used for benchmarking the solver.

In Chapter 3 we will explore how we can actually solve CSPs. We will start from a naive backtracking algorithm approach that resembles depth first search. Next we will improve on this algorithm by changing which variables the solver tries out first. This is called variable ordering. In a last step we will introduce a method to enhance solving CSPs by using inference. Inference is a way to tighten the problem size and reduce the amount of search we have to do with backtracking. To apply inference there will be two methods discussed: Forward checking and enforcing arc consistency. The former beeing a simpler approach that is a special case of enforcing arc consistency. The latter can be done in multiple ways, we will be discussing the algorithms called AC-1 and AC-3.

In Chapter 4 as part of this thesis we introduce a CSP solver called Oxiflex written from scratch in rust. Rust [MKI14] is a general purpose programming language focused on security and, importantly for Oxiflex, performance. Oxiflex supports a subset of FlatZinc builtin constraints. Builtin constraints are a type of constraint that has to be supported by any solver in order to support FlatZinc and in turn MiniZinc. The architecture of Oxiflex is made up of tree parts: parser, model and solver. Building a solver from scratch will enable us to build it in a way so that we can build in flags to turn on each improvement for the algorithm separately. This structure will allow us to measure each algorithm combination by its own.

This chapter will also explore attempts that were made to make Oxiflex computationally faster by changing used datastructures.

In Chapter 5 we will include various experiments done with Oxiflex. There are two problem domains used: N-Queens (introduced in Chapter 2) and Slow Convergence which is a problem from the MinZinc benchmarks repository [Min18]. The improvements to the solver algorithm discussed in Chapter 3 will be compared to each other and visualized using diagrams. Both time and number of iterations are part of the measurements. Time is great to showcase how fast the solving is for the user and iterations is great to see how much we tightened the problem size after each inference step. As we will see those can vary a lot. Results will show that arc consistency enforcing algorithms in Oxiflex do not outperform forward checking or even the naive approach of the algorithm in time measurements. However they do decrease iterations significantly. We will also see that variable ordering improves the N-Queens problem solving by a lot and even enables Oxiflex to solve the Slow Convergence problem at all.

In Chapter 6 we will finish by a discussion of the thesis and explore possibilities for future work.

2

Constraint Satisfaction Problems

2.1 Overview

Constraint Satisfaction Problems (CSP) [Mac87] are mathematical questions defined as a finite set of variables whose value must satisfy a number of constraints or limitations. When solely talking about the problem without the algorithmic finding of a solution, these are called Constraint Networks. CSPs are typical NP-complete combinatorial problems in the field of Artificial Intelligence. See Example 2.1.1 for an simple constraint network.

Example 2.1.1: Simple Constraint Network

$w = \{1, 2, 3, 4\}$	where:
$y = \{1, 2, 3, 4\}$	$w = 2 \cdot x$
$x = \{1, 2, 3\}$	$w < z$
$z = \{1, 2, 3\}$	$y > z$

We define variables w , y , x and z . Variables w and y can both have one value from $\{1, 2, 3, 4\}$ and variables x and z can have one value from $\{1, 2, 3\}$. The constraints then restrict which values are valid from their respective domains. Here $w = 2 \cdot x$ restrict the value of x to be double of w for example. If there are no constraints for variables, the constraints are still there but they allow every assignment. These constraints are called trivial constraints and are usually omitted.

In this example we define constraints in a mathematical notation. There are no formal restrictions on stating constraints neither by their complexity nor by the number of variables involved. To make it easier to reason about and easier to understand, we model constraints as binary constraint sets within this explanation. This is not needed when implementing the constraints later. Constraints are then sets of valid value pairs for two specific variables. Instead of stating the desired relation between any variables, we list all valid value pair tuples in a set. Constraint $w < z$ then becomes $(R_{wz} = \{(1, 2), (1, 3), (2, 3)\})$ which contains all possible value pairs for the two variables w and z .

We define constraint networks formally:

A (binary) constraint network is a 3-tuple $C = \langle V, \text{dom}, (R_{uv}) \rangle$ such that:

- V is a non-empty and finite set of variables,
- dom is a function that assigns a non-empty and finite domain to each variable $v \in V$, and
- $(R_{uv})_{u,v \in V, u \neq v}$ is a family of binary relations (constraints) over V where for all $u \neq v : R_{uv} \subseteq \text{dom}(u) \times \text{dom}(v)$

And we define our example formally:

$C = \langle V, \text{dom}, (R_{uv}) \rangle$ with

- variables:
 $V = \{w, x, y, z\}$
- domains:
 $\text{dom}(w) = \text{dom}(y) = \{1, 2, 3, 4\}$
 $\text{dom}(x) = \text{dom}(z) = \{1, 2, 3\}$
- constraints:
 $R_{wx} = \{(2, 1), (4, 2)\}$
 $R_{wz} = \{(1, 2), (1, 3), (2, 3)\}$
 $R_{yz} = \{(2, 1), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3)\}$

The goal in CSP is then to find a assignment that satisfies all constraints. For this simple example a possible assignment would be $(w \mapsto 2), (x \mapsto 1), (y \mapsto 4), (z \mapsto 3)$.

2.2 MiniZinc

MiniZinc [NSB⁺07] is a free and open-source constraint modeling language developed at and by Monash University in Australia. It allows us to express Constraint Satisfaction Problems in a mathematical notation-like way. MiniZinc also holds an annual competition of constraint programming solvers on a variety of benchmarks. Here we will be talking about the modeling language. See our simple example 2.1.1 written in the MiniZinc language with example 2.2.1.

Example 2.2.1: MiniZinc Translation

```
var 1..4: w;  
var 1..4: y;  
var 1..3: x;  
var 1..3: z;  
  
constraint w = 2 × x;  
constraint w < z;  
constraint y > z;  
  
solve satisfy;
```

MiniZinc is only the language to express a problem domain. Once a problem domain is specified in MiniZinc we can give the problem to multiple solvers to solve them. In this way we can compare the performance of various solvers on the same problem domain. Note that in MiniZinc we also specify how we want the problem to be solved. With `solve satisfy`; we can tell the solver to give us any solution that satisfies the constraints. MiniZinc also supports `solve maximize` and `solve minimize` for optimization problems. We will be focusing on finding any solution.

MiniZinc also provides a way to parametrize a problem domain. This is a great way to scale a problem size up and see how increasing the problem size affects the solving speed. A great example for this is the Queens Problem (See Section 2.3). We define the Queens Problem domain once and can then run specific problem instances for different n . This makes it really easy to compare the solving speed for the queens problem when $n = 8$, $n = 16$ or $n = 32$. Those files where we specify parameters for MiniZinc files are called data files and have the extension `dzn`. Files where we define the problem domain like in Example 2.2.1 are called MiniZinc files and have the file extension `mzn`. We can combine `mzn` files with `dzn` files to create FlatZinc files that a solver is able to read and solve.

2.2.1 FlatZinc

FlatZinc is a simpler problem specification language provided by the MiniZinc toolchain. It is designed to be used by solvers directly. MiniZinc files in combination with data files are translated to FlatZinc files in a pre-solving step. FlatZinc files have the file extension `fzn` and can directly be read by solvers.

Translating from MiniZinc to FlatZinc maps more advanced instructions from MiniZinc to primitives supported in FlatZinc. An analogy to this translation is compiling a C program to Assembly where MiniZinc is C and FlatZinc is Assembly. FlatZinc therefore requires solvers to support a set of standard constraints called FlatZinc builtins. Builtins need to be implemented to be a fully compatible FlatZinc solver. See Example 2.2.2 for an FlatZinc translation using our Simple Example 2.2.1.

Example 2.2.2: FlatZinc Translation (Simplified)

```

array [1..2] of int: x_introduced_2_ = [1,-2];
array [1..2] of int: x_introduced_3_ = [1,-1];
array [1..2] of int: x_introduced_4_ = [-1,1];
var 2..4: w:: output_var;
var 1..4: y:: output_var;
var 1..3: x:: output_var;
var 1..3: z:: output_var;
constraint int_lin_eq(x_introduced_2_,[w,x],0);
constraint int_lin_le(x_introduced_3_,[w,z],-1);
constraint int_lin_le(x_introduced_4_,[y,z],-1);
solve satisfy;

```

The translation of the variable declarations is straight forward. For the constraints, MiniZinc translated all constraints into FlatZinc builtin constraints. For our simple example MiniZinc used two builtins: `int_lin_eq` and `int_lin_le`. See the lines that start with `constraint`. We will look at `int_lin_eq` further to see how FlatZinc builtins work. Example 2.2.3 shows the signature of the builtin `int_lin_eq` that was used for the constraint $w = 2 \cdot x$.

Example 2.2.3: FlatZinc builtin: `int_lin_eq`

```

predicate int_lin_eq(array [int] of int: as,
                    array [int] of var int: bs,
                    int: c)

```

Note that the builtin `int_lin_eq` expects 3 parameters. The first `as` is an array of `int` constants. This is what the example FlatZinc translation named `x_introduced_2_`. This array is called a parameter, because it has concrete values assigned to it. Here `x_introduced_2_` has the value `[1,-2]` assigned. The second parameter `bs` is an array of `int` variables, that is an array of variables that we want to solve for. Here the variables w and x are passed in also as an array `[w,x]`. The third parameter `c` is also a parameter because it is also a constant value that needs to be passed. Here the value for `c` is 0.

Every FlatZinc builtin also has a description for when the constraint is valid or violated respectively. For `int_lin_eq` the description is given with Eq. (2.1).

$$c = \sum_i as[i] \cdot bs[i] \quad (2.1)$$

For this builtin, MiniZinc translated our constraint into a linear combination. With our example we can fill in the passed parameters to the constraint and we get $0 = w - 2x$ which can be rearranged to $w = 2 \cdot x$.

Note that MiniZinc created these parameter arrays by itself. The `x` within `x_introduced_2_`

is not the same as our variable x that we defined within MiniZinc. Also does the 2 in the name have nothing to do with our model but is instead defined by the MiniZinc translation. Also note that for the translation MiniZinc already does some basic level of inference. The FlatZinc variable w can only have values between 2 and 4 in the translated FlatZinc. Whereas in the MiniZinc version we defined w with the domain $\{1, 2, 3, 4\}$. This means MiniZinc infers that w can not be value 1 and removes it from its domain declaration. Due to the constraint $w = 2 \times x$, the variable w has to be double of x and x must have at least value 1. Therefore excluding 1 as possible value for w .

2.3 Queens Problem

Also called the Eight Queens Puzzle, the Queens Problem is an example of a classic constraint satisfaction problem that involves placing eight queens on an 8×8 chessboard in such a way that no two queens threaten each other. That is, no two queens can share the same row, column, or diagonal. See Fig. 2.1 for an example solution to the 8-Queens Problem.

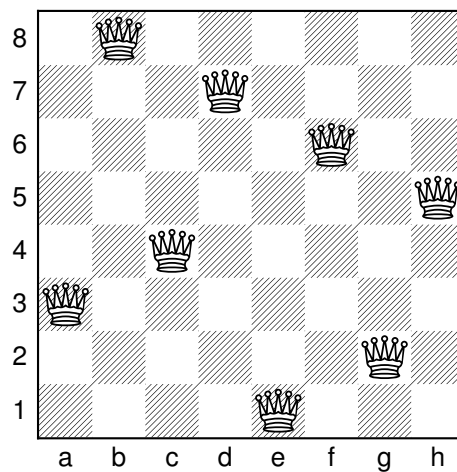


Figure 2.1: Possible solution to the 8-Queens problems.

The Eighth Queens Puzzle is really good suited as an example problem domain for constraint satisfaction problems because it is easy to understand and can also easily be scaled up to increase complexity for a solver. By generalizing the problem from a fixed 8×8 grid size to an $n \times n$ grid with n queens, the problem remains the same in principle, but gets way harder to solve. See Example 2.3.1 [RR06] for the N-Queens Problem modeled in MiniZinc.

Example 2.3.1: N-Queens Problem MiniZinc Model

```
int: n;

array [1..n] of var 1..n: q;

predicate
  noattack(int: i, int: j, var int: qi, var int: qj) =
    qi != qj /\
    qi + i != qj + j /\
    qi - i != qj - j;

constraint
  forall (i in 1..n, j in i+1..n) (
    noattack(i, j, q[i], q[j])
  );

solve satisfy;
```

This MiniZinc model defines an array of variables q where each index corresponds to a column on the chessboard and the value at each index represents the row position of the queen in that column. The constraints ensure that no two queens are on the same row, column or diagonal. Note that this model receives a parameter n and is therefore not specific to 8 queens.

3

Solving Constraint Satisfaction Problems

Constraints satisfaction problems on finite domains are typically solved using a form of search. We search for a solution to the constraint network by trying out possible values until a solution is found or we find that there is no solution. A solution is a valid assignment of all variables with a value of their respective domain satisfying all constraints within the constraint network.

3.1 Backtracking

Backtracking is a technique used to search a problem space for potential solutions [BR75]. It systematically organizes the search process by attempting to extend a partial solution step-by-step. If an extension of the current partial solution proves to be leading to no solution, the algorithm "backtracks" to a previous, shorter partial solution and tries a different path. This method is particularly useful in solving constraint satisfaction problems, as we can do partial assignments of variables and expand them step by step by assigning more variables. We can start by using backtracking without any enhancements. See function 3.1.1 for reference with the following notes:

Input: constraint network C and partial assignment α for C . On first invocation of Naive-Backtracking we pass an empty assignment $\alpha = \emptyset$.

Result: Total assignment (solution) of C or **inconsistent**.

function 3.1.1: NaiveBacktracking(C, α)

```

 $\langle V, \text{dom}, (R_{uv}) \rangle := C$ 
if  $\alpha$  is inconsistent with  $C$ :
    return inconsistent

if  $\alpha$  is a total assignment:
    return  $\alpha$ 

select some variable  $v$  for which  $\alpha$  is not defined
for each  $d \in \text{dom}(v)$  in some order:
     $\alpha' := \alpha \cup \{v \mapsto d\}$ 
     $\alpha'' := \text{NaiveBacktracking}(C, \alpha')$ 
    if  $\alpha'' \neq \text{inconsistent}$ :
        return  $\alpha''$ 

return inconsistent

```

This algorithm corresponds to Depth First Search (DFS). It assigns values to variables from their domains to form a partial assignment. This process continues until either all variables are assigned and a solution is found, or a constraint is violated. If a constraint is violated, the algorithm backtracks and tries a different value from the domain until a solution is found. If all possible assignments violate constraints, there is no solution. Finding a total assignment, that is an partial assignment that gives each variable a valid value from its domain, is finding a solution.

Backtracking is great as an easy to understand solving technique, but is far from the best way to solve CSPs [BG95].

3.1.1 Variable Ordering

Backtracking in general does not specify in which order the search is done. For constraint satisfaction problems we want to assign critical variables early. Critical variables are variables that tighten the search space the most by their assignment. This can be done in multiple ways:

- **static order**
Fixed order defined prior to search.
- **dynamic order**
Order depends on current search state and is calculated after each assignment.

Dynamic ordering is more powerful but also requires computational overhead during search for each iteration. The following are two commonly used variable ordering criteria:

- **minimum remaining values:**
prefer variables that have small domains

- **most constraining variable:**
prefer variables contained in many constraints

Dynamic variable ordering is usually more effective combined with inference.

3.2 Inference

Inference allows us to modify our constraint network by tightening the constraint network. Tightening works by excluding values from domains of variables that we know are not possible. For example in the Queens Problem (See 2.3) if we place a Queen on *d4*, we can exclude the value 4 from all other files (chess term for column). We can also exclude all diagonally positioned squares like *a1*, *b2*, *c3* and so forth. See Fig. 3.1 for reference. Note that we do not need to do anything with the file the queen is on because we modeled the file to be a variable to solve for and a variable can only be one value anyways.

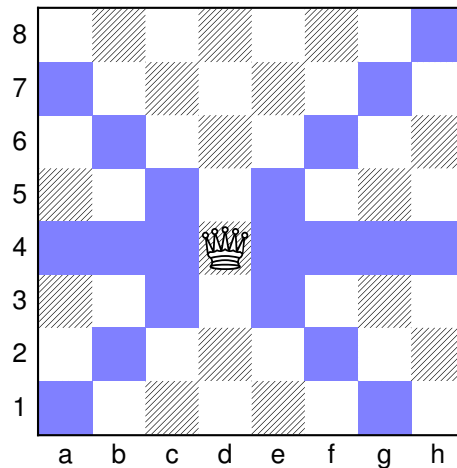


Figure 3.1: Possible solution to the 8-Queens problems.

By removing impossible values from the remaining domains, we can tighten the resulting constraint network and have a smaller search space. We adjust our NaiveBacktracking approach by applying inference after each assignment of a variable. See function 3.2.1 for the adjusted algorithm.

function 3.2.1: BacktrackingWithInference(C, α)

```

 $\langle V, \text{dom}, (R_{uv}) \rangle := C$ 
if  $\alpha$  is inconsistent with  $C$ :
    return inconsistent

if  $\alpha$  is a total assignment:
    return  $\alpha$ 

 $C' := \langle V, \text{dom}', (R'_{uv}) \rangle := \text{copy of } C$ 
apply inference to  $C'$ 
if  $\text{dom}'(v) \neq \emptyset$  for all variables  $v$ :
    select some variable  $v$  for which  $\alpha$  is not defined
    for each  $d \in \text{copy of } \text{dom}'(v)$  in some order:
         $\alpha' := \alpha \cup \{v \mapsto d\}$ 
         $\text{dom}'(v) := \{d\}$ 
         $\alpha'' := \text{BacktrackingWithInference}(C', \alpha')$ 
        if  $\alpha'' \neq \text{inconsistent}$ :
            return  $\alpha''$ 

return inconsistent

```

Note that we now have to copy the constraint network after each assignment which can introduce significant overhead for large problems. The coping is needed because we still have to backtrack if we find an inconsistent assignment. When backtracking we have to restore the domain for each variable again because it is possible that the domain had values that are possible again after backtracking.

3.2.1 Forward Checking

We start with a simple inference method called Forward Checking [HE80]. See function 3.2.2 for reference.

function 3.2.2: ForwardChecking(C, α)

```

 $\langle V, \text{dom}, (R_{uv}) \rangle := C$ 
for each  $v \in \text{unassigned variable in } \alpha$ :
    for each  $R_{vx}$  and  $R_{xv}$  in  $C$ , any  $x$ :
        if any  $d \in \text{dom}(v)$  and any  $d' \in \text{dom}(x)$ :  $(d, d') \notin R_{vx}$  or  $(d', d) \notin R_{xv}$ 
             $\text{dom}(v) = \text{dom}(v) \setminus d$ 

```

Forward checking is basically looking ahead in the future to see which values can be excluded from search after an assignment. By looking ahead we can omit the backtracking part that would result by finding a dead end. We check each value for each variable with the

new assignment and remove all values that are not possible anymore from their respective domain.

3.2.2 Arc Consistency

Originally developed to address vision problems, arc consistency represents a generalization of forward checking [Wal72]. Forward checking enforces arc consistency for all variables with respect to the just assigned variable. Arc consistency does this for all variables. This makes forward checking a special case of arc consistency. We can define arc consistency formally:

Let $C = \langle V, \text{dom}, (R_{uv}) \rangle$ be a constraint network.

- The variable $v \in V$ is arc consistent with respect to another variable $v' \in V$, if for every value $d \in \text{dom}(v)$ there exists a value $d' \in \text{dom}(v')$ with $\langle d, d' \rangle \in R_{vv'}$.
- The constraint network C is arc consistent, if every variable $v \in V$ is arc consistent with respect to every other variable $v' \in V$.

Note that for a variable pair the definition is not symmetrical. That means if v is arc consistent with respect to v' , v' does not have to be arc consistent with respect to v .

3.2.2.1 Enforcing Arc Consistency

There are multiple algorithms to enforce arc consistency [Mac77] [Bes94]. The simplest is called AC-1. It works by making use of a function called revise. The function 3.2.3 revise ensures arc consistency in one direction between two variables.

function 3.2.3: revise(C, v, v')

```

 $\langle V, \text{dom}, (R_{uv}) \rangle := C$ 
for each  $d \in \text{dom}(v)$ :
    if there is no  $d' \in \text{dom}(v')$  with  $\langle d, d' \rangle \in R_{vv'}$ :
        remove  $d$  from  $\text{dom}(v)$ 

```

The function 3.2.4 called AC-1 iterates over each constraint and applies revise in both directions to each variable pair for each constraint until there was no change within its iteration of using revise.

function 3.2.4: AC-1(C)

```

 $\langle V, \text{dom}, (R_{uv}) \rangle := C$ 
repeat
    for each nontrivial constraint  $R_{uv}$ :
        revise( $C, u, v$ )
        revise( $C, v, u$ )
until no domain has changed in this iteration

```

Building on AC-1, AC-3 tries to save redundant checks made by AC-1. Instead of repeatedly going over all constraints, AC-3 iterates over all constraint once and revises variable pairs again only if needed. We can achieve this by using a queue. See function 3.2.5 for reference.

function 3.2.5: AC-3(C)

```

 $\langle V, \text{dom}, (R_{uv}) \rangle := C$ 
queue :=  $\emptyset$ 
for each nontrivial constraint  $R_{uv}$ :
    insert  $\langle u, v \rangle$  into queue
    insert  $\langle u, v \rangle$  into queue

while queue  $\neq \emptyset$ :
    remove an arbitrary element  $\langle u, v \rangle$  from queue
    revise( $C, u, v$ )
    if dom( $u$ ) changed in the call to revise:
        for each  $w \in V \setminus \{u, v\}$  where  $R_{wu}$  is nontrivial:
            insert  $\langle w, u \rangle$  into queue

```

4

Implementation

4.1 Oxiflex

As part of this thesis we present **Oxiflex**, a minimal constraint satisfaction problem solver from scratch for MiniZinc written in Rust. Oxiflex is a FlatZinc solver that can be used as an backend to MiniZinc. This means Oxiflex minimally supports the requirements for a solver to take advantage of the MiniZinc toolchain. The goal is to have a minimal solver that is able to measure the impact of our improvements like forward checking and enforcing arc consistency on constraint satisfaction problem solvers.

Oxiflex is open-source, licensed under the MIT license and available on Github¹.

4.2 Rust

Rust [MKI14] is a general purpose systems programming language focused on safety and performance. It achieves these goals without using a garbage collector by ensuring memory safety through a system of ownership with strict compile-time checks enforced by the borrow checker. This makes Rust particularly well-suited for creating performance-critical applications like CSP solvers where control over resources is crucial. This makes Rust an ideal choice for developing Oxiflex.

4.2.1 Limitations

Not all FlatZinc builtins are supported in Oxiflex. The idea is to implement just the needed builtins for any given interesting problem domain. Further are only `ints` supported, no floating point values. With that there are no optimization solving structures available in Oxiflex to be able to search for an optimal solution instead of just any solution.

¹ <https://github.com/glklimmer/oxiflex>

4.3 Dependencies

4.3.1 flatzinc

The library flatzinc [Thi20] is a FlatZinc parser for Rust. It parses the FlatZinc format into Rust structures and variables. Oxiflex uses version 0.3.20 of flatzinc.

4.3.2 structopt

The library structopt [P.20] is utilized to parse command-line arguments in Oxiflex. This library simplifies setting up custom commands and flags for Oxiflex. Oxiflex uses version 0.3.26 of structopt.

4.4 Architecture

Oxiflex is made up from three parts: parser, model and solver. The solver part can be fine tuned from outside by using command line flags that enable different solving strategies. The output is printed to standard output in a format given by the MiniZinc toolchain.

4.4.1 parser

Using the library flatzinc 4.3.1 Oxiflex reads an FlatZinc `.fzn` file and collects all parts needed to then construct a constraint satisfaction network. These include a list for parameters, variables and constraints. In order to also output the solution after solving the problem, MiniZinc makes use of annotations on FlatZinc elements. Variables that are needed for the output are annotated as `output_var`. There are two possible output annotations in FlatZinc: `output_var` and `output_array`.

4.4.2 model

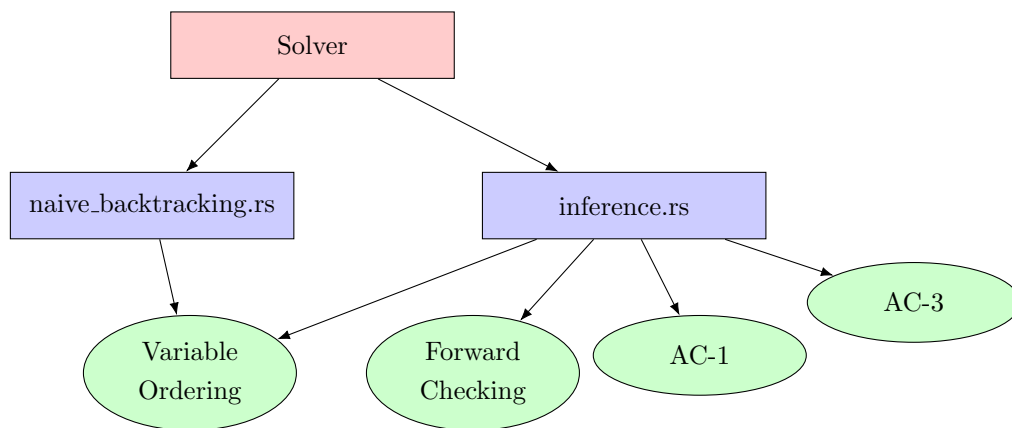
After parsing a FlatZinc file into Rust structures that can be used directly, Oxiflex starts to build useful structures to solve a given CSP. This is where Oxiflex creates a model containing variables with their respective domains and constraints. Models use HashMaps to keep track of its variables and their respective domains. This allows for constant access time to domains to either read or modify them after inference (3.2) for example. Constraints are saved by the model as a list (In rust this is a pointer, capacity, length triplet). Usually when checking if constraints are violated we either want all constraints or all constraints related to a variable. For this reason an additional HashMap is created called `constraint_index`, that uses variable ids as key and points to a list of constraints on the heap. In Rust this can be done by using reference counting. This results in two ways to access constraints. One that is just a list to iterate over all constraints and one where a HashMap is used to get all constraints involved by a specific variable.

Variables all have an id. All variable ids are strings. Oxiflex also uses reference counting to store variable ids. As it is often also needed to pass variable ids around, we can mitigate the cost of calling `clone` on variable ids by using reference counting. Instead of actually cloning variable ids, we just pass a pointer to the variable id needed. With reference counting we

can ensure the actual memory for the variable `id` is freed after all pointers to it have been deleted.

4.5 Solver

The solver is the core part of Oxiflex. By allowing control over what optimization is turned on or off we can measure the impact of each optimization individually. As discussed in Chapter 3 there are various optimizations for solving CSPs. See the following diagram to see all optimizations implemented in Oxiflex.



By default each optimization in Oxiflex is turned on. By passing flags named after each optimization we can disable the respective optimization. The help menu can be printed using `Oxiflex --help` from the console.

FLAGS:

```

-f, --forward-checking
Use forward checking as inference

-n, --naive-backtracking
Use naive backtracking, e.g. no forward_checking

-r, --random-variable-order
Use random order for variable ordering.

-a, --arc-consistency <arc-consistency>
Specify arc consistency version [default: 3]
```

4.5.1 Value Ordering

Oxiflex is able to use dynamic ordering of variables during search based on the number of constraints. Enabled by default, Oxiflex orders variables from most constraints involvement to least for assignment. So variables that are involved with the most constraints are chosen

first to be assigned. This fail early approach to ordering can be used both for NaiveBacktracking and Inference based algorithms. The calculation for which variable has the most constraints we use the HashMap called `constraint_index` (See 4.4.2: model) mentioned in section 4.4.2: model.

4.5.2 Forward Checking

Forward checking in Oxiflex works by removing values of domains that are no longer valid for some constraints. Domains in Oxiflex are of type `Vec`, which are pointer, capacity and length triplets. When Oxiflex was started, the removal of values in domains was done in a immutable manner. That is, before removing values the whole domain was copied. After copying, the values were removed from the copied domain. Finally the old domain was replaced with the new one in the model. This was easier to implement, but more inefficient. Forward checking also uses `constraint_index` (See 4.4.2: model) to only get the constraints that are needed instead of checking all constraints.

4.5.3 Arc Consistency

Both AC-1 and AC-3 use the function `revise` which ensures arc consistency in one direction for two variables. The main computational work to ensure arc consistency happens within this function. The role of AC-1 and AC-3 is to arrange the calls to `revise`. Within `revise` we also use the `constraint_index` (See 4.4.2: model) to get only constraints that are involved with the given variable for `revise`.

Checking constraints works by checking a `PartialAssignment`. The actual type for `PartialAssignment` is a HashMap with variable id for keys and assignments of variables as value. Therefore within `revise` in order to values for two variables, `PartialAssignment` is created for each combination of values. That means for each value pair within the two domains a new HashMap is created with two elements. The first element is an assignment of the first variable and the second an assignment of the second variable.

5

Results

5.1 Method

For benchmarking two metrics were measured:

- Time
Time until a solution is found.
- Iterations
How many times the recursive algorithm was called.

For time measurements hyperfine [Pet23] is used. Each time benchmark includes 3 warmup runs and is averaged. The longer the solver takes, the fewer runs are done. This is default behaviour of hyperfine. At least 10 runs were made for each benchmark. Time is given in seconds. Iterations were measured by doing 5 runs and averaging the number of iterations used.

A benchmark is a pair of problem size and a combination of algorithmic modifications to the solver. As Oxiflex allows to enable each optimization we can create 8 optimization combinations.

- `-n -r`
NaiveBacktracking
- `-n`
NaiveBacktracking with variable ordering
- `-f -r`
Inference with forward checking
- `-f`
Inference with forward checking and variable ordering
- `-a 1 -r`
Inference with AC-1

- `-a 1`
Inference with AC-1 and variable ordering
- `-r`
Inference with AC-3
- `no flags`
Inference with AC-3 and variable ordering

The following Problem Domains where measured:

- N-Queens
- Slow Convergence

All benchmark were performed on the same machine.

CPU:	Intel i7-6700K (8) @ 4.200GHz
Memory:	6051MiB / 32021MiB
hyperfine version:	1.16.1
Rust version:	1.76.0
Operating System:	Pop!_OS 22.04 LTS

5.2 N-Queens

Fig. 5.1 shows the runtime comparison for the N-Queens Problem (See Section 2.3: Queens Problem) with $n = 8..20$ for all combinations. It shows that arc consistency enforcing methods AC-1 and AC-3 in Oxiflex take much more time then the naive and forward checking approach. We can also see that variable ordering seems to improve all approaches significantly.

Fig. 5.2 does not include arc consistency enforcing methods making it clear to see the impact of forward checking on time measurements for the N-Queens Problem. This shows that although the naive approach without variable ordering performs much worse then forward checking without variable ordering, it outperforms forward checking with variable ordering. Fig. 5.3 shows benchmarks with the same parameters for measurements of iterations. It shows that the number of iterations grows significantly up from $n = 20$. And although we could see that arc consistency enforcing methods worsen the performance time wise, they provide huge improvements for number of iterations.

In Fig. 5.4 we provide benchmarks for iterations without the naive approach with $n = 14$ up to $n = 24$. As expected forward checking performed the worst out of the three.

Table 5.1 and table 5.2 contain results for time and iterations. Time values are rounded to 2 decimal places. If there is a standard deviation for the measurements it is given after the \pm symbol. If there is no variable ordering the measurments can vary greatly because variables are chosen randomly to be assigned a value.

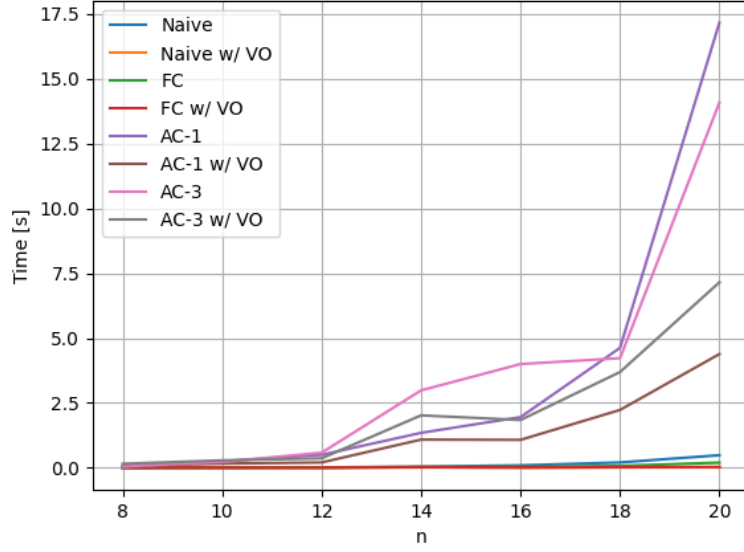


Figure 5.1: N-Queen time measurements averaged over multiple runs.

n	14	16	18	20
Naive	0.05 ± 0.08	0.09 ± 0.12	0.21 ± 0.45	0.49 ± 1.29
FC	0.03 ± 0.03	0.04 ± 0.04	0.07 ± 0.10	0.20 ± 0.66
AC-1	1.35 ± 0.73	1.95 ± 1.06	4.62 ± 3.25	17.17 ± 31.87
AC-3	2.99 ± 1.92	4.00 ± 2.67	4.23 ± 0.97	14.09 ± 9.93
Naive w/ VO	0.03	0.00	0.02	0.02
FC w/ VO	0.02	0.01	0.02	0.03
AC1 w/ VO	1.09 ± 0.02	1.08 ± 0.03	2.23 ± 0.04	4.39 ± 0.06
AC3 w/ VO	2.02 ± 0.04	1.84 ± 0.05	3.69 ± 0.06	7.15 ± 0.05

Table 5.1: N-Queens time in seconds over multiple runs, Mean \pm SD, w/ VO: with variable ordering.

n	14	16	18	20
Naive	3538 ± 1148	1749 ± 907	2760 ± 1145	21479 ± 11183
FC	110 ± 51	415 ± 76	79 ± 16	270 ± 63
AC-1	51 ± 9	396 ± 343	195 ± 67	104 ± 19
AC-3	103 ± 41	34 ± 6	91 ± 68	86 ± 30
Naive w/ VO	3536	137	1018	1011
FC w/ VO	112	17	46	52
AC1 w/ VO	52	17	25	36
AC3 w/ VO	49	17	24	34

Table 5.2: N-Queens iterations over 5 runs, Mean \pm SD, w/ VO: with variable ordering.

5.3 Slow Convergence

The Slow Convergence Problem is from the minizinc benchmarks repository [Min18]. Benchmarks for this problem without variable ordering took over 300 seconds and are therefore omitted. Fig. 5.5 shows benchmarks for $n = 2..10$ with combinations using variable order-

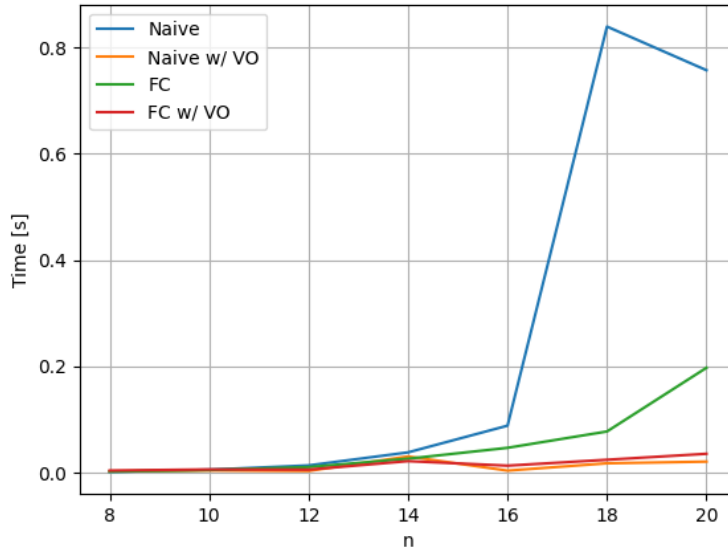


Figure 5.2: N-Queens time measurements without arc consistency averaged over multiple runs.

ing. We can see that time measurements for AC-1 and AC-3 grow exponentially and it seems that the naive and forward checking approach are growing linearly.

Fig. 5.6 shows benchmarks with the same parameters for measurements of iterations. Note the huge spike in iterations at $n = 3$. This spike is not a measurement error. The benchmarks were run multiple times and provided the same results. We can see that when measuring iterations, inference methods help reducing the number of iterations.

Next, increasing n for non arc consistency enforcing benchmarks. Fig. 5.7 shows time measurements on the left and iteration measurements on the right. Note the inverse correlation between iterations and time for the two options. Although naive backtracking ($-n$) takes more iterations, it is still faster than forward checking ($-f$). We can also observe that it looked like forward checking was growing linearly in Fig. 5.5, and Fig. 5.7 on the left clearly show also exponential growth. Further we can again observe that the naive approach seems to be growing linearly, but as we will see, it also grows exponentially.

Table 5.3 and Table 5.4 contain results for time and iterations for the $n = 10..60$ range.

n	Naive w/ VO	FC w/ VO
10	0.00	0.01
20	0.00	0.13
30	0.01	0.54 ± 0.01
40	0.01	1.58 ± 0.08
50	0.02	5.29 ± 0.14
60	0.04	21.94 ± 0.5

Table 5.3: Slow Convergence time in seconds over multiple runs, Mean \pm SD, w/ VO: with variable ordering.

n	Naive w/ VO	FC w/ VO
10	77	23
20	252	43
30	527	63
40	902	83
50	1377	103
60	1952	123

Table 5.4: Slow Convergence number of iterations over 5 runs, w/ VO: with variable ordering.

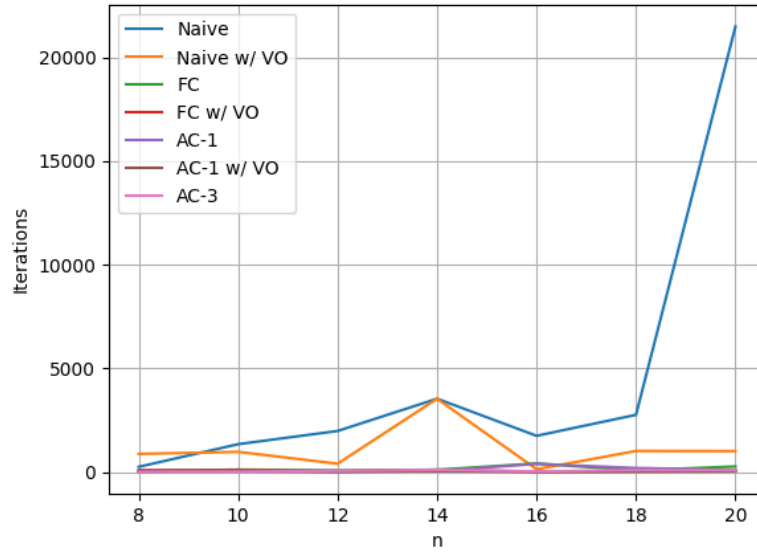


Figure 5.3: N-Queens iteration measurements averaged over 5 runs.

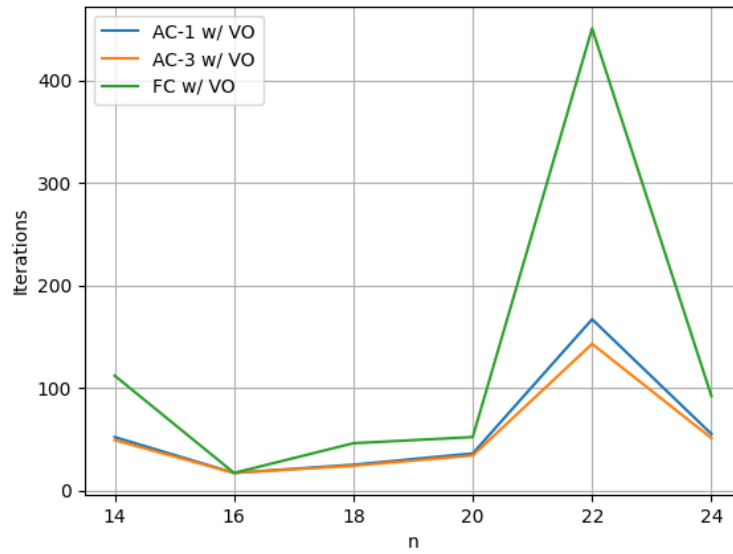


Figure 5.4: N-Queens iterations only inference averaged over 5 runs.

5.4 Gecode

For comparison Fig. 5.8 shows Oxiflex compared to Gecode [Gec06]. Gecode is a constraint satisfaction problem solver compatible with MiniZinc with state-of-the art performance. Note the steep increase in $n = 100..600$.

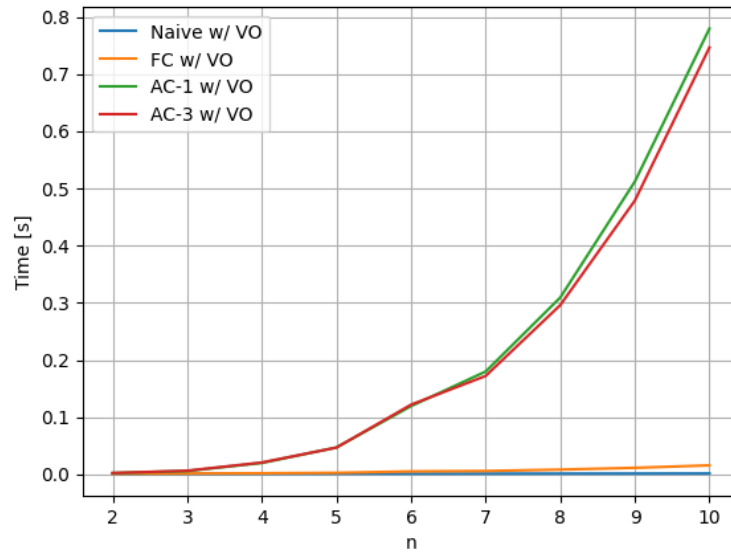


Figure 5.5: Slow Convergence time measurements averaged over multiple runs.

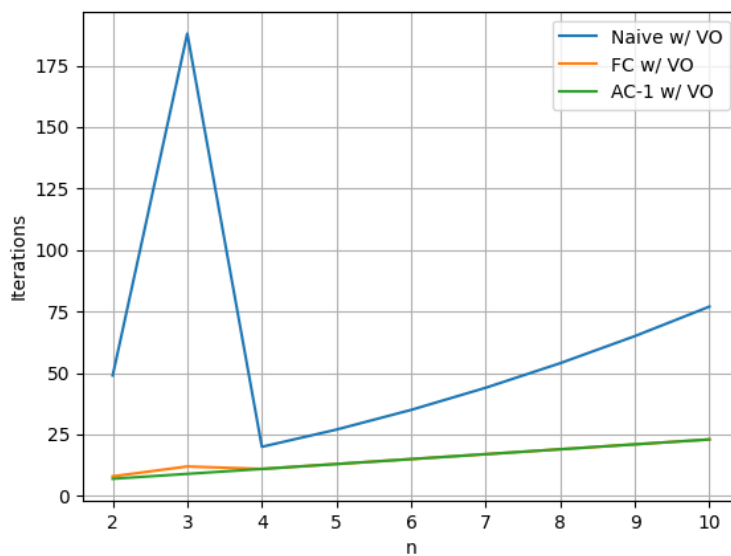


Figure 5.6: Slow Convergence iteration measurements averaged over 5 runs.

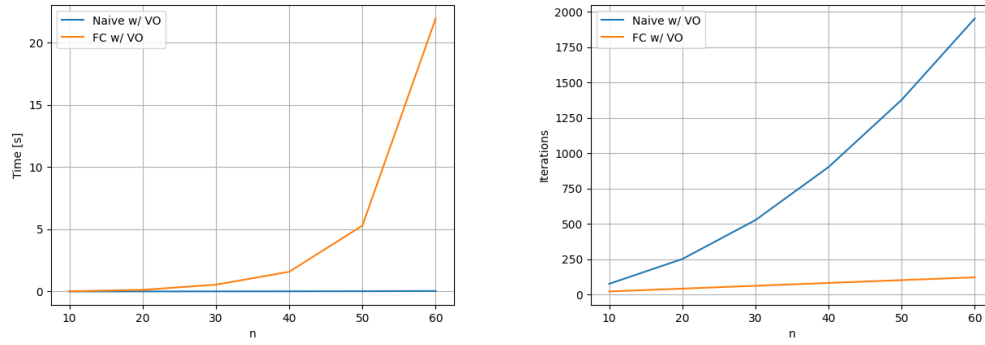


Figure 5.7: Comparison for higher $n = 10..60$. Left: time, right: iterations.

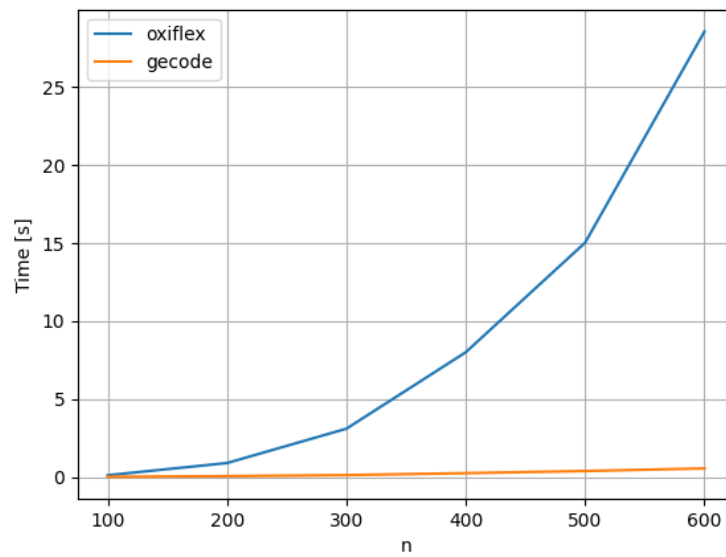


Figure 5.8: Slow Convergence time measurements averaged over multiple runs.

6

Conclusion

6.1 Discussion

The original goal of this thesis and Oxiflex was to showcase how much of an impact forward checking and arc consistency would make. Although they did have an impact, the impact is in the wrong direction. Enforcing arc consistency made Oxiflex in some cases slower by an order of magnitude. At least forward checking made the N-Queens problem be solved faster.

It is great to see the tradeoff between search and inference though. As we could see in Fig. 5.7, although it took longer to solve, inference did reduce the number of iterations significantly. It is interesting to see the effect that variable ordering has for the slow convergence problem. In fact it made it even possible to solve the problem at all. It can be useful to measure other things than time (like iterations) to gather insights like these.

The performance measured at the point of writing this is already significantly better compared to previous versions of Oxiflex. Mainly by removing values from domains inplace and using the `constraint_index` hashmap to only check constraints that are needed. Making improvements like this included trial and error, benchmarking and profiling. Making a change like using a different datastructure could feel like it could improve performance but without measuring before and after, the change can make the code more complex without providing performance gain.

Currently the main performance loss for arc consistency enforcing algorithms in Oxiflex lie within the `revise` function. After profiling the main issue seems to be around creating hashmaps that act as `PartialAssignment`. It seems that although hashmaps have an insertion complexity of $\mathcal{O}(1)$ that the most time within `revise` was spent on creation of and insertions made to the hashmap used to then check constraints. It feels like a good idea to not use hashmaps and just pass the needed data directly to be checked within a simple struct. This is easily possible to refactor because within `revise` only two variables are checked by their constraints. But this modification turned out to make it worse and degrade the performance even further.

Both inference methods repeatedly go over domains of variables and remove values. Domains are implemented using a simple `Vec`. As already mentioned those are pointer, capacity and length triplets. So removing a single element has complexity of $\mathcal{O}(n)$. As we remove values

from domains often, it would make sense to use a `LinkedList` instead as it has complexity of $\iota(1)$ for removing a single element. After trying this out, the performance stayed the same. Although removing elements from `Vec` takes $\mathcal{O}(n)$ we always have to go every value in a domain anyways to check all values against the constraints. Not on purpose this removal of values in domains was done using the function `retain`. `retain` accepts a predicate and goes over all values in the `vec` and only keeps values that the predicate allows. After benchmarking this was just as efficient as using `LinkedList` for domains.

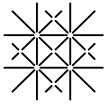
Therefore the main takeaway for this thesis is that datastructures matter. Although an algorithm performs better in theory, the right datastructures have to be used to make it really go faster. Using hashmaps for everything might not be the best approach if performance is the main criteria of a program. This also underlines that just using a fast programming language is not sufficient to make a program go fast.

6.2 Future work

Bibliography

- [Bes94] Christian Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.
- [BG95] Fahiem Bacchus and Adam Grove. On the forward checking algorithm. In Ugo Montanari and Francesca Rossi, editors, *Principles and Practice of Constraint Programming — CP '95*, pages 292–309, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [BR75] James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Commun. ACM*, 18(11):651–656, nov 1975.
- [Gec06] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from <http://www.gecode.org>.
- [HE80] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.
- [Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
- [Mac87] Alan K. Mackworth. Constraint satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. J. Wiley and Sons, NY, March 1987.
- [Min18] MiniZinc. Slow convergence problem, minizinc benchmarks, 2018.
- [MKI14] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.
- [NSB⁺07] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. Minizinc: Towards a standard cp modelling language. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
- [P.20] Guillaume P. structopt, January 2020.
- [Pet23] David Peter. hyperfine, March 2023.
- [RR06] Peter Stuckey Reza Rafeh. N-queens problem, minizinc benchmarks, 2006.
- [Thi20] Sven Thiele. flatzinc, April 2020.

-
- [Wal72] David Waltz. Understanding line drawings of scene with shadows. *The Psychology of Computer Vision*, pages 19–91. McGraw Hill, 1972.



Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)

Translation from German original

Title of Thesis: _____

Name Assessor: _____

Name Student: _____

Matriculation No.: _____

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Place, Date: _____ Student: _____

Will this work, or parts of it, be published?

No

Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: _____

Place, Date: _____ Student: _____

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis.