University of Basel

Department of Mathematics and Computer Science

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

This thesis discusses the thesis template using some examples of the Turing Machine.

Table of Contents

1
Introduction

test

2

Constraint Satisfaction Problems

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 AI.

<div>

Example:

$$w = \{1, 2, 3, 4\}$$
$$y = \{1, 2, 3, 4\}$$
$$x = \{1, 2, 3\}$$
$$z = \{1, 2, 3\}$$

where:
$$w = 2 \cdot x$$
$$w < z$$
$$y > z$$

</div>

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 a 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 mathmatical 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 within this text, we model constraints as binary constraint sets. This is only for simplification. 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 varaibles w and z.

We define Constraint Networks formally:

A (binary) constraint network is a 3-tuple $C = <V, \text{dom}, (R_{uv})>$ 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 $(w \mapsto 2), (x \mapsto 1), (y \mapsto 4), (z \mapsto 3)$.

2.1 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 mathmatical notation-like way.

```
MiniZinc example

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. MiniZinc Domain files have the file extension `.mzn`.

MiniZinc also provides a way to parametrize a problem domain. This is a great way to scale a problem space up and see how increasing the problem size affects the solving speed. A great example for this is the Queens Problem (See Section 2.2). 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 $n = 32$. Files where we specify parameters for MiniZinc files are called data files and have the extension `.dzn`. We can than combine `.mzn` files with `.dzn` files to created FlatZinc files.

2.1.1 FlatZinc

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

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 contraints called FlatZinc builtins. Builtins need to be implemented to be a fully compatible FlatZinc solver. See the following for an FlatZinc translation using our simple example.

Simple example translated to FlatZinc (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 declations 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. The following is the signature of the builtin `int_lin_eq` that was used for the constraint $w = 2 \cdot x$.

---

int_lin_eq builtin signature

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 as follows.

---

int_lin_eq builtin constraint

$$c = \sum_i \text{as}[i] \cdot \text{bs}[i] \tag{2.1}$$

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 FlatZinc 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 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.

2.2 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 8x8 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 the following for an example solution to the 8-Queens Problem.
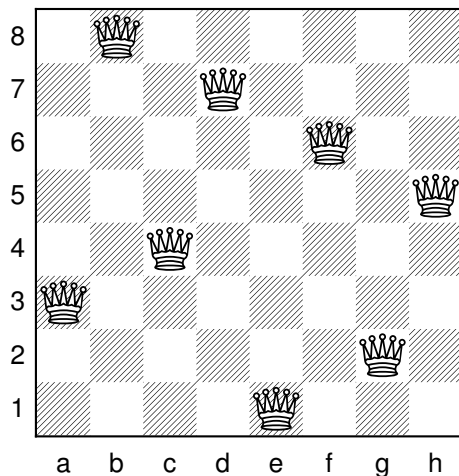
The Eigth Queens Puzzle is really good suited as an example problem domain for constraint satisfaction problems because it is easy to understand and can also easily be scaled up to increase complexity for a solver. By generalizing the problem from a fixed 8×8 grid size to an $n \times n$ grid with n queens, the problem remains the same in principle, but gets way harder to solve. See the following example of the queens problem modeled in MinZinc [RR06].

```
MiniZinc Model for N-Queens Problem

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.

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 all possible values until a solution is found. That is, a valid assignment of all variables with a value of their respective domain satisfing all constraints within the constraint network.

3.1 Naive 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 unfeasible at any stage, 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 demonstrated by the algorithm called NaiveBacktracking.

---

function NaiveBacktracking)

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

if α is a total assignment:
return α

select some variable v for which α 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 $\alpha'' \neq$ **inconsistent**:
return α''

return inconsistent

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**.

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 far from the best way to solve CSPs.

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 thightening the constraint network. Thightening works by excluding values from domains of variables that we know are not possible. For example in the Queens Problem (See 2.2) 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 3.2 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 how one value anyways.
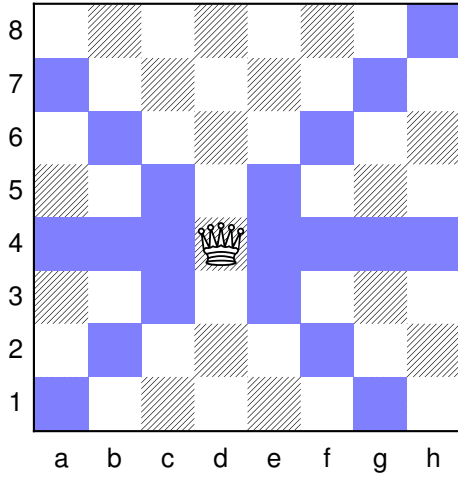
By removing impossible values from the remaining domains, we can thighten the resulting constraint network and have a smaller search space. We adjust our NaiveBacktracking approach by applying inference after each assignment of an variable. See the following for the NaiveBacktracking algorithm.

```
function BacktrackingWithInference(C, α)

⟨V, dom, (R_uv)⟩ := C
if α is inconsistent with C:
return inconsistent


if α is a total assignment:
return α


C' := ⟨V, dom', (R'_uv)⟩ := copy of C
apply inference to C'
if dom'(v) ≠ ∅ for all variables v:


select some variable v for which α is not defined
for each d ∈ copy of dom'(v) in some order:
α' := α ∪ {v ↦ d}
dom'(v) := {d}
α'' := BacktrackingWithInference(C', α')
if α'' ≠ inconsistent:
return α''


return inconsistent
```

Note that we now have to copy the constraint network after each assignment which can introduce significant overhead for simple 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 the following algorithm.

> function ForwardChecking(C, α)
>
> for each $v \in$ unassigned variables in α:
> for each constraint:
> **if** any $d \in \text{dom}(v)$ in conflict: $\text{dom}(v) = \text{dom}(v) \setminus d$

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 ommit 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

Arc consistency excludes all impossible values from all domains of all variables in a given constraint network. Originally developed for vision problems [Wal72] arc consistency is the generalization of forward checking. 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. See the following for formal definition.

> Definition: Arc Consistent
>
> 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 following function revise ensures arc consistency in one direction between two variables.

> **function** 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 following algorithm AC-1 goes 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** 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 just going over all constraints over and over, we go at least over all constraint once and revise variable pairs again only if needed. In AC-3 this can be achived by using a queue. See the following for reference.

> **function** 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** $\text{dom}(u)$ 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

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 and be able measure the impact of various improvements like forward checking on constraint satisfaction problem solvers.

Oxiflex is open-source and licensed under the MIT license. It is available at Github https://github.com/glklimmer/oxiflex.

4.1 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 Dependecies

This work depends on previous work by others. This section highlights the components used by oxiflex.

4.2.1 flatzinc

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

4.2.2 structopt

The library structopt [P.20] is utilized to parse command-line arguments in oxiflex. This library simplifies setting up custom commands and options for oxiflex.

4.2.3 hyperfine

The librabry hyperfine [Pet23] is a command-line benchmarking tool. We use hyperfine to measure and compare the performance of different solver strategies and optimizations.

4.3 Architecture

There are three main parts of oxiflex.

- parser

- model

- solver

4.3.1 parser

Using the library flatzinc 4.2.1 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 annonations 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`.

4.3.2 model

After parsing the FlatZinc file into Rust structures that can be used directly, oxiflex starts to build useful structures to solve any given problem. This is where oxiflex creates a model containing variables with their respectice 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.

4.3.3 Limitations

There are some limitations due to time constraints that currently limit oxiflex as a universal
MiniZinc solver.

Not all FlatZinc builtins are supported. The idea is to implement just the needed builtins
for any given insteresting problem domain.

4.4 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: Solving Constraint Satisfaction Problems, there are various optimizations for
solving CSPs. See the following diagram to see all optimizations implemented in oxiflex.
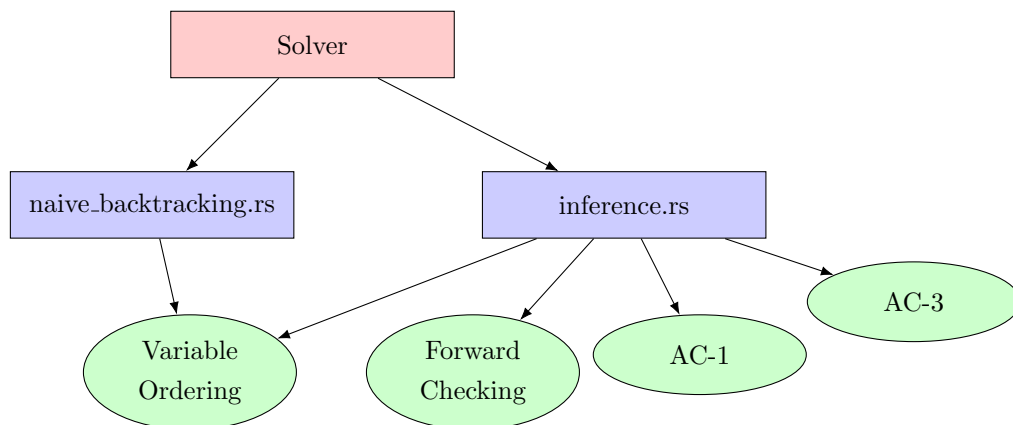
By default each optimization in oxiflex is turned on. By passing flags named after each
optimization we can disable the respective optimization. The help menu can be printed
using `oxiflex --help` from the console.

```
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]
```

4.4.1 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 4.3.2: model) mentioned in section 4.3.2: model.

4.4.2 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 where 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 4.3.2: model) to only get the constraints that are needed instead of checking all constraints.

4.4.3 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 4.3.2: 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 firt variable and the second an assignment of the second variable.

5

Results

5.1 Method

For benchmarking two metrics where measured:

- Time

  Time until a solution is found.

- Iterations

  How many times the recursive algorithm was called.

For time measurments hyperfine is used. Each 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 made were made for each benchmark. Time is usually given in seconds. More time is worse. Iterations were measured by doing 5 runs and averaging the number of iterations used. More iterations is worse.

A benchmark is a pair of problem size with a combination of algorithmic modifications to the solver. As oxiflex allows to enable each optimization we can create 8 optimization combinations. Note that by default the all optimizations are enabled and by passing flags to oxiflex we disable them.

- `-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

5.2   N-Queens

Figure 5.1 shows the N-Queens Problem (See 2.2: Queens Problem) with $n = 4..14$. Figure 5.2 shows benchmarks with the same parameters for measurements of iterations.
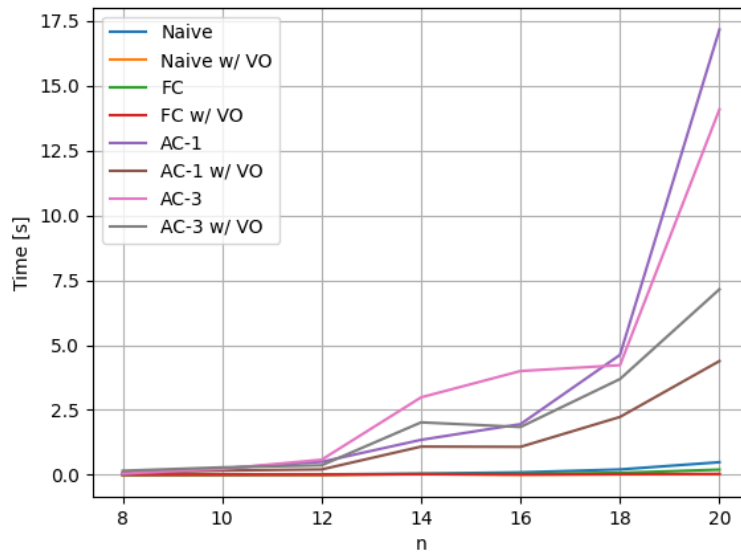

Figure 5.1: Time measurements with all possible flag combinations

It appears that the time measuments in Figure 5.1 for runs that do not enforce arc consistency grow linearly. Figure 5.3 shows benchmarks only for runs that do not enfore arc consistency using the same steps for n.
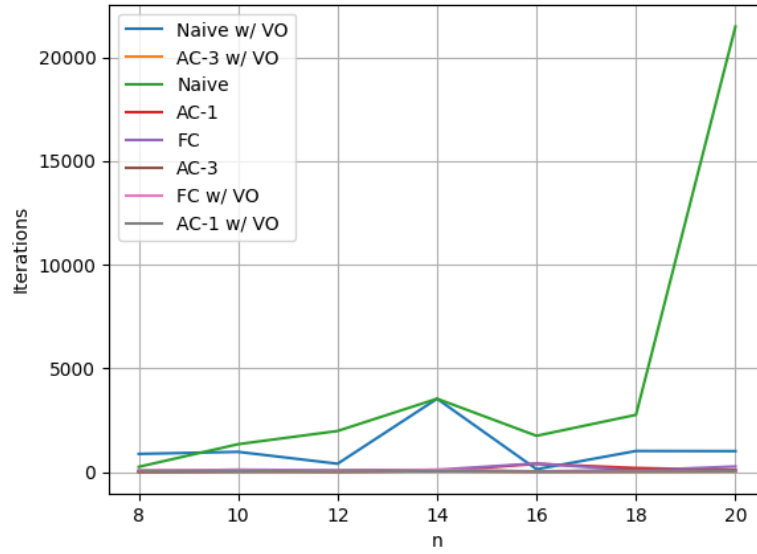
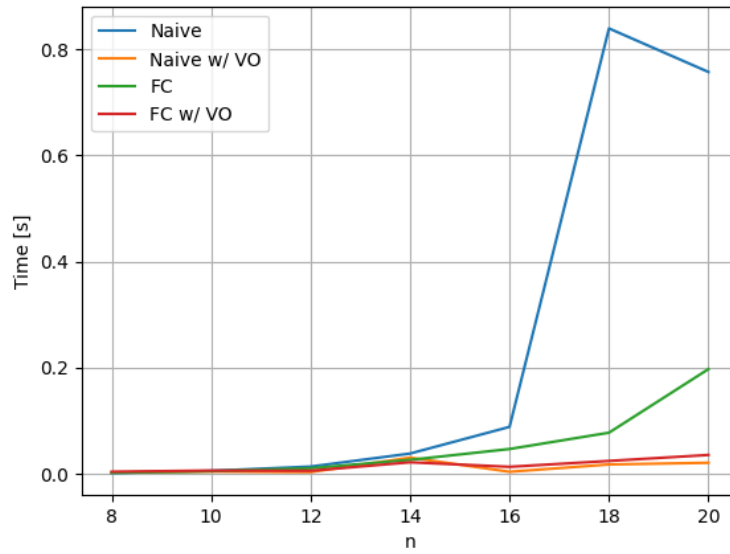Figure 5.2: Iteration measurements with all possible flag combinations

Figure 5.3: Time measurements without arc consistency enforcing flags

Table 5.1 and table 5.2 contains results for time and iterations. Time values are rounded to 2 decimal places.
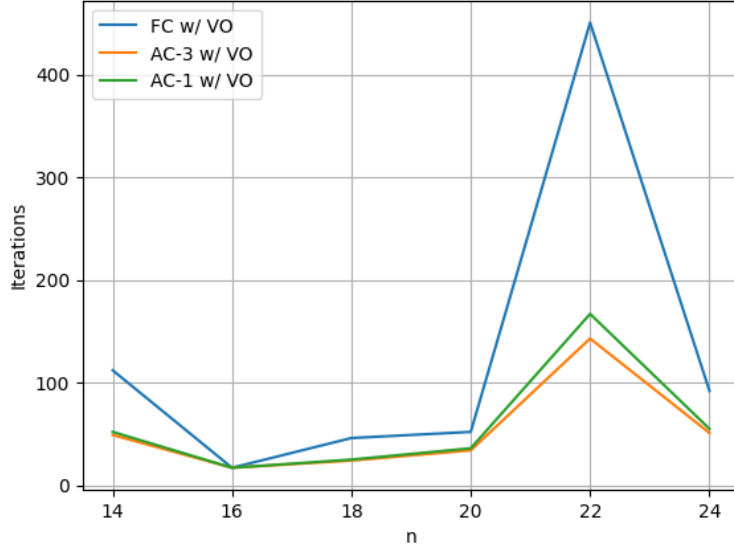
Figure 5.4: Iterations only for inference methods

n	8	10	12	14
Naive w/ VO	0.00	0.00	0.00	0.03
FC w/ VO	0.00	0.01	0.01	0.02
AC1 w/ VO	0.54 ± 0.02	1.25 ± 0.04	1.00 ± 0.04	12.55 ± 0.19
AC3 w/ VO	0.54 ± 0.02	1.24 ± 0.04	1.07 ± 0.13	12.51 ± 0.17
Naive	0.00	0.01 ± 0.01	0.02 ± 0.02	0.04 ± 0.07
FC	0.00	0.01	0.01 ± 0.01	0.03 ± 0.04
AC-1	0.16 ± 0.14	0.87 ± 0.81	9.26 ± 11.91	8.84 ± 6.22
AC-3	0.18 ± 0.13	1.08 ± 1.30	3.27 ± 3.83	33.31 ± 44.64

Table 5.1: N-Queens Time in seconds

n	8	10	12	14
Naive w/ VO	877	976	403	3536
FC w/ VO	89	84	31	112
AC1 w/ VO	877	976	403	3536
AC3 w/ VO	877	976	403	3536
Naive	401 ± 251	1274 ± 441	1387 ± 996	3160 ± 2321
FC	28 ± 5	63 ± 30	154 ± 116	424 ± 177
AC-1	235 ± 92	946 ± 478	1886 ± 1008	25653 ± 20496
AC-3	201 ± 62	1212 ± 565	6527 ± 2522	2578 ± 717

Table 5.2: N-Queens Iterations

5.3   Slow Convergence

The Slow Convergence Problem is from the minizinc benchmarks repository [Min18]. Benchmarks without variable ordering took way to long and are therefore omitted. Figure 5.5 therefore only shows benchmarks with variable ordering enabled for $n = 1..10$, as higher values for n took way to long to measure. Note that the benchmark repo suggests dzn files

for n values from 100 up to 1000.

Figure 5.6 shows benchmarks with the same parameters for measurments 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.
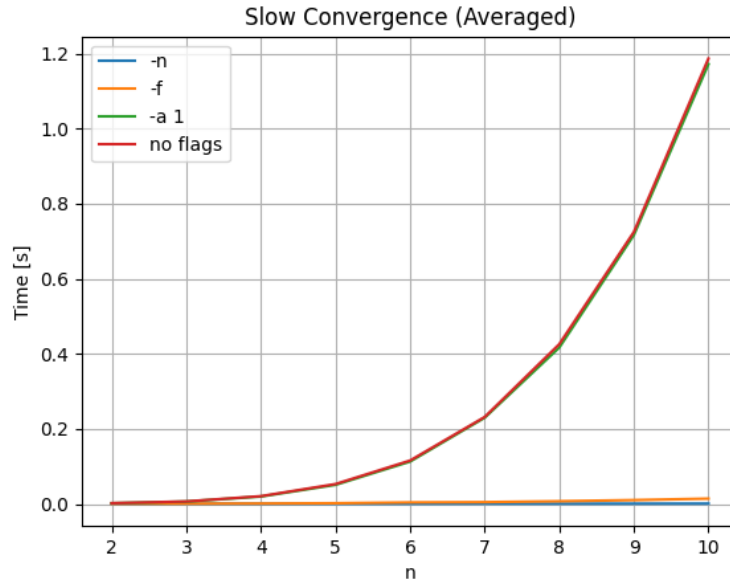

Figure 5.5: Time measurements only with varaible ordering combinations
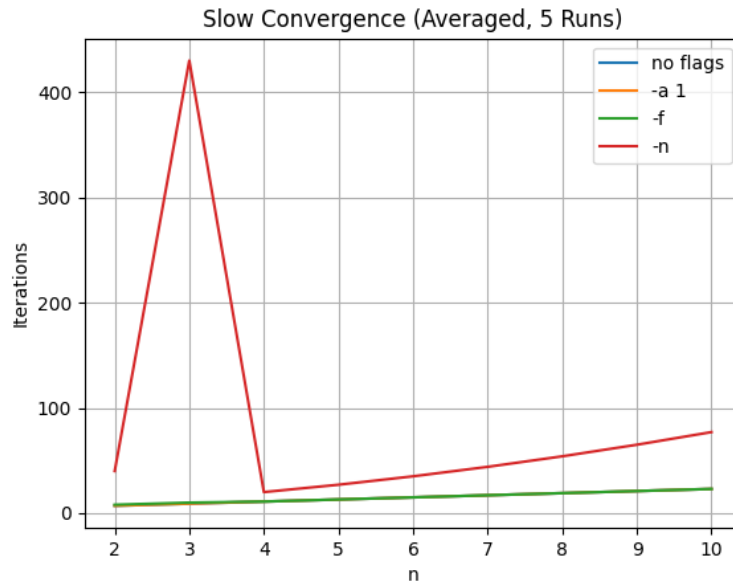

Figure 5.6: Iteration measurements only with varaible ordering combinations

Next increasing n for non arc consistency enforcing benchmarks. Figure 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. Altough naive backtracking (-n) takes more iterations, it is still faster than forward checking (-f).
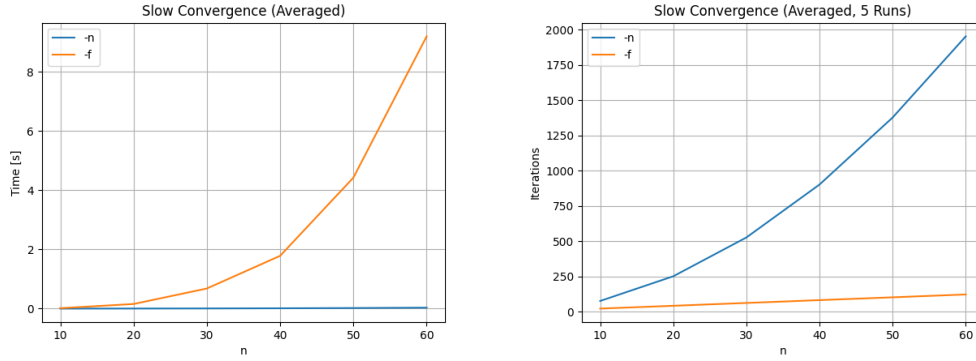

Figure 5.7: Comparison for higher $n = 10..60$. Left: time, right: iterations.

Table 5.3 contains results for time and iterations for the $n = 10..60$ range. Time values are rounded to 2 decimal places.

Slow Convergence Time			Slow Convergence Iterations		
n	-n	-f	n	-n	-f
10	0	0.02	10	77	23
20	0	0.16	20	252	43
30	0.01	0.68	30	527	63
40	0.01	1.79	40	902	83
50	0.02	4.43	50	1377	103
60	0.03	9.21	60	1952	123

Table 5.3: Comparative data for Slow Convergence: Time and Iterations.

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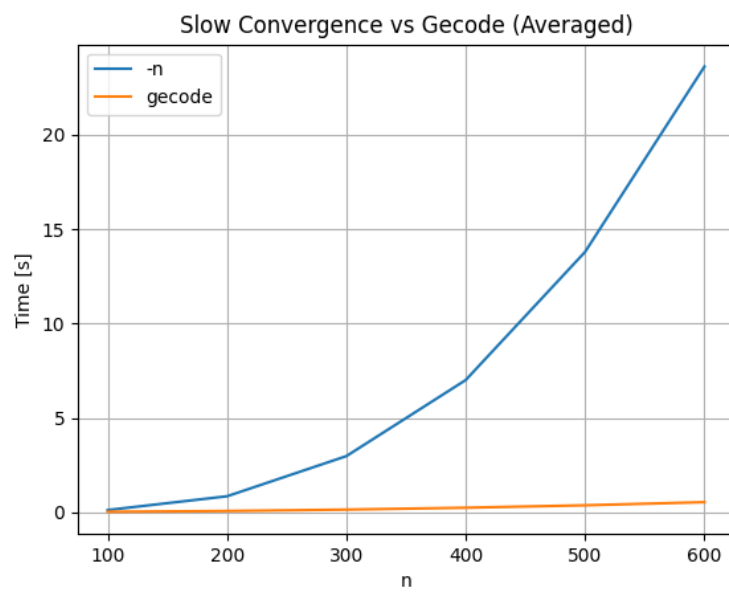
Figure 5.8: oxiflex vs gecode

6

Conclusion

The original goal of this thesis and oxiflex was to showcase how much of an impact forward checking and arc consistency would make. Altough 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, altough 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 seemes to be around creating hashmaps that act as `PartialAssignment`. It seemes that altough 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 $\prime(1)$ for removing a single element. After trying this out, the performance stayed the

same. Altough 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. Altough an algorithm performs better in theory, the right datastructures have to be used to make it really go faster. Using hashmaps for everything migth 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.

Bibliography

[Bes94]   Christian Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[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.

A

Appendix

University of Basel

Faculty of Science
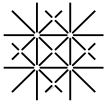
Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)
Translation from German original

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.