University of Basel

Oxiflex - A Constraint Programming Solver for MiniZinc written in Rust

Bachelor's thesis

Faculty Artificial Intelligence of the University of Basel
Department of Mathematics and Computer Science
Research Group
https://ai.dmi.unibas.ch/

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

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

Hand-In-Date

Abstract

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

Table of Contents

1

Introduction

2

Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSP) 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.

Example:

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

where:

$$w = 2 * x$$
$$w < z$$
$$y > z$$

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 a value from $\{1, 2, 3\}$. The constraints then restrict which values are valid from their respective domains. Here $w = 2 \times x$ 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, we model constraints as binary constraint sets. 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 $C =< V, \mathrm{dom}, (R_{uv}) >$ 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 $u \neq v : R_{uv} \subseteq \mathrm{dom}(u) \times \mathrm{dom}(v)$

And we define our example formally:

$C =< V, \mathrm{dom}, (R_{uv}) >$ with

- variables:
  $V = \{w, x, y, z\}$
- domains:
  $\mathrm{dom}(w) = \mathrm{dom}(y) = \{1, 2, 3, 4\}$
  $\mathrm{dom}(x) = \mathrm{dom}(z) = \{1, 2, 3\}$
- 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 [1] 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 it each. 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 space 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.

Simple example translated to FlatZinc

```
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:: is_defined_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):: defines_var(w);
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. But our constraints are each split into two parts and translated into linear combinations. For example is constraint $w = 2 \times x$ translated to the builtin called `int_lin_eq` which is defined as follows:

int_lin_eq builtin

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

With the restiction on given parameters to the constraint.

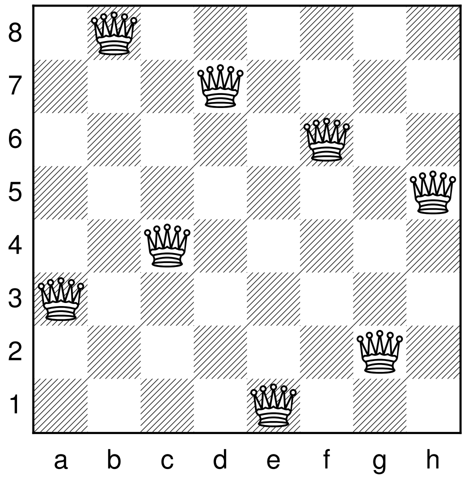
int_lin_eq builtin constraint

$$c = \sum_i \mathrm{as}[i] \times \mathrm{bs}[i] \tag{2.1}$$

Note that MiniZinc already does some basic level of inference. The FlatZinc variable w can only have values between 2 and 4 in the translated FlatZinc file. But in the MiniZinc file 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.



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.

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.

This model is from a past MiniZinc challenge by Reza Rafeh July 2005 and Peter Stuckey September 30 2006. See https://github.com/MiniZinc/minizinc-benchmarks/blob/master/

queens/queens.mzn for reference.

3
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 supports the minimal requirements for a solver to take advantage of the MiniZinc toolchain. The idea is to have a minimal solver and excatly measure the impact of various improvments on it like forward checking or arc consistency.
Oxiflex is open-source and available under the MIT license under https://github.com/glklimmer/oxiflex.

3.1 Rust

Rust 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.

3.2 Dependecies

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

3.2.1 flatzinc

The library flatzinc [4] is a FlatZinc parser for Rust. It parses the FlatZinc format into Rust structures and variables.

3.2.2 structopt

The library structopt [2] is utilized to parse command-line arguments in oxiflex. This library simplifies setting up custom commands and options for oxiflex.

3.2.3 hyperfine

The librabry hyperfine [3] is a command-line benchmarking tool. We use hyperfine to measure and compare the performance of different solver strategies and optimizations.

3.3 Architecture

There are three main parts of oxiflex.

- parser
- model
- solver

3.3.1 parser

Using flatzinc 3.2.1 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`.

3.3.2 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 (3.5) 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 that uses variable ids as key and points to a list in 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 constraint and a hashmap to get all constraints that use a specific variable.

3.3.3 Limitations

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

Only binary constrains are supported. The reason for this is that the theory used to implement oxiflex is based on binary constraints. Therefore it made sense to also just support them at first.

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

3.4 Solver

Constraints satisfaction problems on finite domains are typically solved using a form of search. For oxiflex we use backtracking.

3.4.1 Naive Backtracking

function NaiveBacktracking)

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

if α is a total assignment:
return α

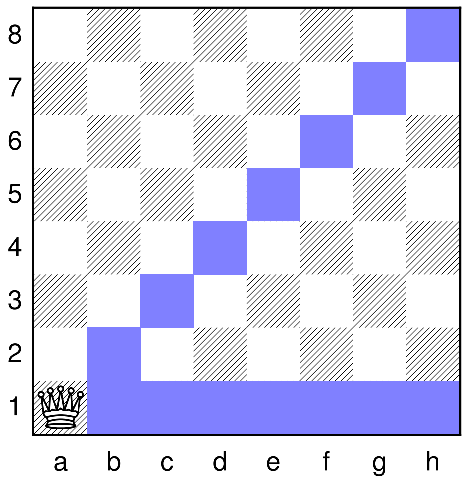
select some variable v for which α is not defined
for each $d \in \mathrm{dom}(v)$ in some order:
$\alpha' := \alpha \cup \{v \mapsto d\}$
$\alpha'' :=$ NaiveBacktracking(C, α')
if $\alpha'' \neq$ **inconsistent**:
return α''

return inconsistent

Input: constraint network C and partial assignment α for C. On first invocation of NaiveBacktracking 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 any value to any variable from its domain. Repeating this until either all variables are set and a solution is found or a constraint is violated. If a constraint is violated, the algorithm goes back and tries an different value from the domain until a solution is found. If all possible assignments violate a constraint, then there is no solution.

3.5 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 anymore after an assignment. For example in the Queens Problem (See 2.2) if we place a Queen on $a1$, we can exclude the value 1 from all other files (columns in chess-lingo). We can also exclude all diagonally positioned squares like $b2$, $c3$ an so forth because the queens is on $a1$. See 3.5 for reference.



function BacktrackingWithInference)

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

if α is a total assignment:
return α

$C' := \langle V, \text{dom}', (R'_{uv})\rangle :=$ copy of C
apply inference to C'
if $\text{dom}'(v) \neq \emptyset$ for all variables v:

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

return inconsistent

3.5.1 Forward Checking

3.5.2 Arc Consistency

4
Results

Results, Graphs and stuff.

5 Conclusion

Time for some interpretation.

Bibliography

[1] 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.

[2] Guillaume P. structopt, January 2020. URL https://github.com/TeXitoi/structopt.

[3] David Peter. hyperfine, March 2023. URL https://github.com/sharkdp/hyperfine.

[4] Sven Thiele. flatzinc, April 2020. URL https://github.com/potassco/flatzinc.

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.