



MASTER  
**INFORMATIQUE**



UNIVERSITÉ  
**CÔTE D'AZUR**

MASTER IN COMPUTER SCIENCE

COURSE : CONSTRAINT PROGRAMMING

---

## Mini-solver in OCaml based on binary Constraint Satisfaction Problem

---

*Author:*  
Fissore Davide

*Supervisor:*  
Jean-Charles Régim

Februray 2023



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Binary constraints and arc consistency</b>	<b>1</b>
<b>3</b>	<b>My Implementation</b>	<b>2</b>
3.1	Doubly linked lists . . . . .	2
3.2	Graph, domains and constraints representation . . . . .	3
3.3	The solver . . . . .	3
3.4	The Arc Consistency Algorithms . . . . .	4
3.4.1	<i>AC-3</i> . . . . .	5
3.4.2	<i>AC-4</i> . . . . .	5
3.4.3	<i>AC-6</i> . . . . .	5
3.4.4	<i>AC-2001</i> . . . . .	6
<b>4</b>	<b>Run the project</b>	<b>6</b>
4.1	The parser . . . . .	6
4.2	Example of commands . . . . .	7
<b>5</b>	<b>Benchmark</b>	<b>7</b>
5.1	The <i>All Interval Series</i> problem . . . . .	7
5.1.1	Generation of the problem . . . . .	7
5.1.2	Stats . . . . .	8
5.2	The <i>N-Queens</i> problem . . . . .	8
5.2.1	Generation of the problem . . . . .	8
5.2.2	Stats . . . . .	9
<b>6</b>	<b>Conclusion</b>	<b>9</b>
<b>A</b>	<b>An output example</b>	<b>10</b>
A.1	Example file to parse . . . . .	10
A.2	Example of output . . . . .	10
<b>B</b>	<b>Concrete problems' inputs</b>	<b>11</b>
B.1	A <i>All Interval Series</i> input . . . . .	11
B.2	A <i>N-Queens</i> input . . . . .	13

# 1 Introduction

The goal of this project is to realize a binary constraint solver based on the arc consistency filtering algorithms *AC-3*, *AC-4*, *AC-6*, *AC-2001*. As we have seen in the Constraint Programming course provided by Mr. Régin, a constraint satisfaction problem is a problem made of a set of variables  $V = \{V_1, \dots, V_n\}$  such that each variable  $V_i$  is defined over a domain  $D_i$ . A domain is a set of values that can be assigned to the corresponding variable.

A constraint is a relations between a set of values taken from some of the given domains such that each domain of each constraint appear at most one time.

Let  $n$  be the number of variables of the problem, the constraint problem can be represented as an undirected  $n$ -partite hyper-graph  $G = (V, E)$  where each partition of the graph is made of the values of each domain.  $V$  is made of all the values of each variable and  $E$  represent the constraints of the graph.

**Example 1.** If we have the variables  $V_1 = \{1, 2, 3\}$  and  $V_2 = \{1, 2\}$  and  $V_3 = \{0, 1, 2\}$  and the constraint  $C_1 \triangleq |v_1 - v_2| = v_3$ , we build the hyper-graph  $G$  made of the vertices  $\{1_{V_1}, 2_{V_1}, 3_{V_1}, 1_{V_2}, 2_{V_2}, 0_{V_3}, 1_{V_3}, 2_{V_3}\}$  and the hyper-edges are made such that the constraint  $C_1$  is respected, for instance we can build the multi-edge  $e = \{1_{V_1}, 1_{V_2}, 0_{V_3}\}$  since the absolute value of the difference between the value 1 from  $V_1$  and the value 1 from  $V_2$  gives 0 in  $V_3$ .

A solution of a constraint satisfaction problem is a simple-path passing exactly one time through each partition of the hyper-graph, this is an equivalent for the classic definition which says that a solution is a the choice of a value for each variable such that every constraint are satisfied.

A value  $v_i \in D_i$  of the variable  $V_i$  is supported in the hyper-graph if for each constraint  $c_1$  involving the domains  $D(C) = \{d_1, \dots, d_n\}$  we have  $d_i \in D(C)$  and for each domain  $d_j \in D(C)$  there exists a value  $v_j \in D_j$  having a relation with  $v_i$ . A not-supported value can be removed from its domains since it cannot be part of a solution of the problem.

We can find a solution of a constraint problem by choosing an arbitrary value  $v_i$  from a domain  $D_i$  and removing all the other values in  $D_i$ . We look for all the domains  $D_j$  linked to  $D_i$  and remove all the values in  $D_j$  that are no more supported; these values are called *delta domain*. We repeat this operation until no modification can be performed. This operation is called *propagation*. If after the propagation there exists an empty domain it means that there does not exist a solution containing  $v_i$ . We repeat the procedure with backtracking the state of the problem before  $v_i$  was chose and we choose a new value in  $D_i$  different from  $v_i$ . If on the other hand, after propagation we have no empty domains, we take a value  $v_j$  from another domain  $D_j$  and repeat the procedure. If we are able to select a value for each value which not produce an empty domain, it means we have found a solution.

## 2 Binary constraints and arc consistency

An interesting property of constraint satisfaction problems is that they can always been rewritten in an equivalent problem having only binary constraint. A binary constraint is a constraint linking only two variables and thanks to this strategy the corresponding graph will have no more hyper-edges.

**Example 2.** If we retake the problem depicted in [Example 1](#), we can change its model by adding an auxiliary variable  $V_{aux}$  representing the “index” of each multi-relation of the original problem. For example, if we take  $\{1_{V_1}, 1_{V_2}, 0_{V_3}\}$ , we can say that  $1_{V_{aux}}$  is the index of this tuple of values. The constraint  $C_1$  is split in 3 sub-constraints:  $C_1^1$  representing the link between  $V_{aux}$  and  $V_1$ ,  $C_1^2$  representing the link between  $V_{aux}$  and  $V_2$ ,  $C_1^3$  representing the link between  $V_{aux}$  and  $V_3$ . Note that constraint  $C_1^3$  is made in order to respect the original constraint  $C_1$ . A more detailed example will be provided in [Section 5.1.1](#)

In the state of the art we can find a lot of algorithms aiming to filter the values of a domains returning the set of not-supported values in a binary constraint satisfaction problem after deletion of a value  $v_i$  in a domain  $D_i$ .

In the following paragraph we will sketch the main ideas behind the algorithms *AC-3*, *AC-4*, *AC-6*, *AC-2001*.

**AC-3** In the *AC-3* algorithm, after the deletion of a value  $v_i$  from a domain  $D_i$ , *AC-3* will iterate over each domain  $D_j$  with a relation with  $D_i$  and for each value of  $v_j \in D_j$ , if there does not exist a value in  $D_i$  supporting  $v_j$ ,  $v_j$  will be returned.

**AC-4** This algorithm has an internal data structure in order to improve the search of not-supported variables. Each value of each domain is associated to the list of the values supporting it. When we remove a value  $v_i \in D_i$ , we can directly know which variables  $v_j$  depend on  $v_i$  and if  $v_j$  has no other variable in  $D_i$  supporting it,  $v_j$  is returned.

**AC-6** In *AC-6*, the internal data structure is similar to the one of *AC-6*, but instead of associating each value  $v_i$  to all the value  $v_j$  supporting  $v_i$ , we only store the first value in each domain supporting  $v_i$ . In this way, we reduce the amount of data to store in memory and when a value  $v_i \in D_i$  is removed, we look for the values  $v_j$  supporting it and if the support of  $v_j$  in  $D_i$  is different from  $v_i$  nothing is done. Otherwise, we look for a new support in  $D_i$  starting from the value  $v_i$ . Note that in this algorithm it is important to give an order to the values in the domain.

**AC-2001** In order to use the minimum amount of space, *AC-2001* stores, for each value  $v_i \in D_i$ , the first element  $v_j$  in the related domain supporting it. When  $v_i$  is removed, we only look for the values depending on  $v_j$  and for them we look for a new support starting from  $v_i$  in  $D_i$ , if such value doesn't exist it means that  $v_j$  can be removed from  $D_j$ .

### 3 My Implementation

I have developed my solver in OCaml (v. 4.13.1) a functional programming language using pointer and the *Base* library since I have noticed better speed performances compared of the standard OCaml modules. In the following subsections I will provide a brief explanation of some of the most important modules I have implemented.

#### 3.1 Doubly linked lists

A doubly linked list (*dll*) is a list whose elements have a pointer to their corresponding following and preceding element. One can note that the predecessor (resp. successor) of the first (resp. the last) element of a doubly linked list are represented by a fictive object: in my case I have used the *None* option type. Doubly linked lists are particularly useful since the insertion and the deletion of an element of a *dll* can be done in constant time: this is particularly useful to backtrack a list to a previous state.

```
type 'e node = {
  value : 'e;
  id : int;
  dll_father : 'e t;
  mutable prev : 'e node option;
  mutable next : 'e node option;
  mutable is_in : bool;
}

and 'e sentinel = { mutable first : 'e node; mutable last : 'e node }
and 'e t = { id_dom : int; name : string; mutable content : 'e sentinel option }
```

We can see that the type *node* has a *prev* and a *next* field which are of type optional. The doubly linked list, itself, is represented by the type *t* (following the OCaml convention) and it contains a sentinel pointing on the first and the last element of the *dll*.

To represent in a unique way nodes and *dll*, I have added the field *id* (resp. *id\_dom*) in order to find them quickly when looking inside *Hash-Tables*.

The id of those records are generated through the generator:

```
let gen =
  let x = ref 0 in
  fun () -> incr x; !x
```

Inside the module *DoublyLinkedList*, I have added all of those utility functions allowing to modify the content of a *dll*. In particular, I can create, remove, insert, append or prepend a node inside a *dll* modifying correspondingly the *prev* and the *next* fields.

Moreover, I took inspiration from the *List* module in OCaml and I have added some higher-order functions in order to check if an element belongs to a *dll*, if a property is verified for every element in the *dll* (a kind of *foreach* in Java streams), *etc.*

### 3.2 Graph, domains and constraints representation

The graph, the domains and the constraints are all implemented in the *Graph* module since they contains the information about the problem.

```
open Base
module DLL = DoublyLinkedList

type 'a domain = 'a DLL.t
type 'a value = 'a DLL.node
type 'a relation = 'a value -> 'a value -> bool
type 'a table_type = (int * int) Hash_set.t

type 'a graph = {
  tbl : 'a table_type;
  relation : 'a relation;
  constraint_binding : ('a domain, 'a domain DLL.t) Hashtbl.t;
  domains : 'a domain Hash_set.t;
}
```

The type *graph* is a record containing a *Hash\_set* of pairs of integers, that are the *id* of two values supporting each other; a *relation* taking two node and returning if they are linked in the constraint graph (it can be seen as the edges of the graph). Finally the *constraint\_binding* is a *Hash-Table* associating to the *id* of each domain the set of domain linked through a constraint.

We can add constraints between values through the auxiliary function:

```
let add_constraint (graph : 'a graph) d1 v1 d2 v2 =
  let add_if_absent (d1 : 'a domain) (d2 : 'a domain) =
    let dom =
      Hashtbl.find_or_add graph.constraint_binding d1 ~default:(fun _ ->
        Hash_set.add graph.domains d1;
        DLL.empty "")
    in
    DLL.add_if_absent (fun e -> phys_equal e.value d2) d2 dom
  in
  let get d v = DLL.find_by_value v d in
  let a, b = (get d1 v1, get d2 v2) in
  Hash_set.add graph.tbl (a.id, b.id);
  add_if_absent d1 d2;
  add_if_absent d2 d1
```

This function take in parameter a graph, the name of a variable  $v_1$  followed by the name of its domain  $d_1$  and a second variable  $v_2$  with the name of its corresponding domain  $d_2$ . In this function,  $d_1$  and  $d_2$  are inserted to the list of domains and the constraint between the node  $v_1$  and  $v_2$  is added.

### 3.3 The solver

The solver is the engine behind the resolution of a CP problem. The solver select the values of the domains and each time a selection is performed, the AC algorithm is asked to give back the delta domains.

My solver in OCaml is a *functor* taking in parameter a module of type *Arc\_consistency*.

The solver has two public functions:

```
module type Solver = sig
  module DLL = DoublyLinkedList

  val initialization : ?verbose:bool -> string Constraint.graph -> unit

  val find_solution :
    ?debug:bool ->
    ?count_only:bool ->
```

```

?only_valid:bool ->
?verbose:bool ->
?one_sol:bool ->
unit ->
unit
end

```

These function aim to initiate the problem inside the solver taking a graph  $g$  in entry, and to find one or all the solutions obtainable from  $g$ . All the optional argument of the *find\_solution* method are detailed in [Section 4](#).

Inside the functor we can find all the auxiliary attributes and functions allowing to solve the given problem.

```

type 'a stack_type :
  (string AC.stack_operation * string Graph.value) option Stack.t
val backtrack_mem : 'a stack_type
val stack_op : 'a stack_type
val remove_by_node : ?verbose:bool -> string Graph.value -> unit
val propagation_remove_by_node : ?verbose:bool -> string Graph.value -> unit
val propagation_select_by_node : ?verbose:bool -> string Graph.value -> unit
val back_track : unit -> unit

```

**stack\_op:** is the stack containing all the operation made inside the Arc-Consistency algorithm.

**backtrack\_mem:** is the stack containing all the pointers to a previous state in the exploration tree in order to backtrack.

**remove\_by\_node:** when we remove a value  $v_i$  from a domain  $D_i$ , we call the Arc-Consistency algorithm passed to the solver *functor* and we add to the stack of undo operation the delta domain the set of values to remove for propagation in a second moment. Note that if after remove  $v_i$  from  $D_i$ , we throw the *Empty\_domain* exception which will be caught in order to backtrack and find other solutions.

**propagation\_remove\_by\_node:** this is an recursive function which propagates the deletion of a value  $v_i$  which keep to remove all the value inside the delta domain until it is not empty.

**propagation\_select\_by\_node:** is selected is a function that calls the *propagation\_remove\_by\_node* for all the values inside  $D_i$  that are different from the current value  $v_i$ . At each selection of a value  $v_i$ , we add to the *backtrack\_mem* a pointer to the actual state of the solver in order to get it back during the backtrack step.

### 3.4 The Arc Consistency Algorithms

The Arc Consistency algorithms are modules respecting the signature:

```

module type Arc_consistency = sig
  exception Not_in_support of string

  module DLL = DoublyLinkedList

  type 'a data_struct
  type 'a stack_operation
  type 'a remove_in_domain = string Graph.value list

  val name : string
  val print_data_struct : string data_struct -> unit
  val initialization : ?print:bool -> string Graph.graph -> string data_struct

  val revise :
    string Graph.value ->

```

```

string data_struct ->
string stack_operation * string remove_in_domain

val back_track : string stack_operation -> unit
end

```

In fact, an AC algorithm must have an initialization function allowing to clean the graph and instantiate the internal data structure, a revise function to remove a value  $v_i$  from its domain and return the delta domains and the stack operation to be done during the backtrack to restore the internal state of the internal data structure.

### 3.4.1 AC-3

This algorithm has no data structure, therefore, its only usefull implementation is the revise function witch is the mere application of the *AC-3* definition:

```

let revise (v1 : 'a Graph.value) (graph : 'a data_struct) =
  let delta_domains : 'a Graph.value list ref = ref [] in
  DLL.iter_value
    (DLL.iter (fun v2 ->
      if DLL.not_exist (Graph.relation graph v2) v1.father then
        delta_domains := v2 :: !delta_domains))
    (Graph.get_constraint_binding graph v1.father);
  ((), !delta_domains)

```

We iterate over every node in the domains with a constraint binding with the domain of the value  $v_1 \in D_1$  passed in argument of the function. If there is a value  $v_j \in D_j$  with no support in  $D_i$  then  $v_j$  is appended to the list of delta domains.

### 3.4.2 AC-4

The internal structure of *AC-4* is complex and I have tried to make it as efficient as possible.

```

type 'a double_connection = {
  node : 'a Graph.value;
  mutable assoc : 'a double_connection DLL.node option;
}

type 'a cell_type = ('a Graph.domain, 'a double_connection DLL.t) Hashtbl.t
type 'a data_struct = ('a Graph.value, 'a cell_type) Hashtbl.t
type 'a stack_operation = 'a double_connection DLL.node list

```

The data structure is a *Map*  $\mathcal{M}_1$  associating to each value  $v_i$  of each domain a second map  $\mathcal{M}_\in$ .  $\mathcal{M}_2$  is a map containing all the domains  $D_j$  having a relation with  $v_i$ , to each domain  $D_j$  we associate a list of *double\_connection*. A *double\_connection* is a record containing the value of  $v_j \in D_j$  supporting  $v_i$  and a pointer to the reciprocal *double\_connection* going from  $v_j$  to  $v_i$ . This pointer is useful to speed the deletion of the support  $v_j$  from the support of  $v_i$  if  $v_j$  is deleted. The revise function returns the *double\_connection* nodes removed from the internal data structure of *AC-4* and the list of delta domains.

### 3.4.3 AC-6

The *AC-6* algorithm is the one that caused me a lot of problem of implementation, since it works with both *S-Lists* and the *Last* value.

```

type 'a cell = {
  s_list : ('a Graph.value * 'a cell) DLL.t;
  last : ('a Graph.domain, 'a Graph.value DLL.t) Hashtbl.t;
}

type 'a int_struct = ('a Graph.value, 'a cell) Hashtbl.t
type 'a data_struct = 'a Graph.graph * 'a int_struct
type 'a stack_operation =
  ('a Graph.value * 'a cell) DLL.node list * 'a Graph.value DLL.node list

```



The data structure of *AC-6* is made by the graph on one side and a *Hashtbl* which associates to each value a record of type *'a cell*. A *cell* contains:

- the *s\_list* that is a *dll* associating to each value of the graph a pointer to the *cell* in the internal data structure. This pointer allow to fast find which last values should be modified when a value is deleted.
- the last value that is a *Hashtbl* associating a domain to its corresponding last value.

### 3.4.4 AC-2001

```
type 'a remove_in_domain = string Graph.value list
type 'a last = (Graph.ValueDomain.t, 'a Graph.value DLL.t) Hashtbl.t
type 'a data_struct = { last : 'a last; graph : 'a Graph.graph }
type 'a stack_operation = 'a Graph.value DLL.node list
```

The *AC-2001* data structure is particularly easy to implement: *data\_struct* contains the original graph and the *field* allowing to improve the support search after the deletion of a value  $v_i \in D_i$ . Inside the *last* field, we find an *Hashtbl* associating to each couple  $v_i, D_j$  (where  $D_j$  is a domain with a constraint with  $D_i$ ) the last support of the node  $v_i$  in  $D_j$ . When a node  $v_i$  is remove in  $D_i$ , we lopp for all the domains  $D_i$  with a constraint with  $D_i$  and for all value  $v_j \in D_j$  if the last support of  $v_j$  is  $v_i$  then we look for a new value  $v'_i \in D_i$  starting from  $v_i$  supporting  $v_j$ . If this support does not exists then  $v_j$  will be returned with the delta domains, otherwise  $v'_i$  will replace the last of  $v_j$ .

## 4 Run the project

In order to simplify the interaction with the solver, the main program accepts a list of optional parameters to set the input of the problem, the verbose print mode, the arc consistency mode *etc.*

Here a list of the parameters and their behavior can be printed in the *stout* passing the *-help* option. Here the full list of commands:

```
-ac          Set the filtering algo among 3, 4, 6, 2001 - default : 3
-v          Set the verbose mode
-f          Set the input file
-first      Stop after the first valid solution
-queens [N] Set the size of the queen solver (min N: 5)
-all-int [N] Set the size of the allIntervalSeries solver (min N: 3)
-only-sol   Only print the number of fails and solutions
-only-valid Print only the valid solutions
-d          Debug mode
-print-inp  Print the input graph
-help       Display this list of options
--help      Display this list of options
```

Note that the *all-int* problem is run if you pass both the *all-int* and the *queens* parameters.

The *-f* parameter takes a file following the grammar proposed in [Section 4.1](#).

### 4.1 The parser

A simple parser have been added to the main in order to easily enter file for texting the solver. A valid file to be parsed respect the following grammar:

```
P := variables* "\n--\n" constraints*
variables := name ":" (domains " ")* "\n"
constraint := variable " " value " " variable " " value

name := [a-zA-Z]*
domains := [a-zA-Z]*
```

An example of file input can be seen in [Appendix A.1](#)

## 4.2 Example of commands

- `dune exec -- main -queens 3 -ac 3 -only-sol:` run the 3-queens problem with the *AC-3* arc consistency filter and print only the solutions on the console. Result:

```
-----
The number of fails is 0
The number of solutions is 6
Total Time: 0.000068
Time of backtracks: 0.000003
Time of revise: 0.000013
-----
```

- `dune exec -- main -f ./graphs/input_4.txt -ac 2001 -v` produces the result proposed in [Appendix A.2](#)<sup>1</sup>
- the bash executables `./allInt.sh` and `./queens.sh` run respectively the *All Interval Series* and the *N-Queens* problem for  $n$  going from 0 to respectively 12 and 11. Not that both executable accept two integers in parameter changing the default bound of computation. For example, `./allInt.sh 3 7` will launch the *All Interval Series* for  $n$  from 3 to 7.

## 5 Benchmark

In this section I will provide a brief performance comparison of the for filtering algorithm on the *All Interval Series* and the *N-Queens* problem. Of course, the performances of each *AC* algorithm will also depend on the efficiency of my implementation, since I have tried to optimize every algorithm using different programming strategy, but in all of them I use *Hash-Tables* in the data structures and therefore the performances will depend on how many time the algorithm is trying to access an element of the *Hash-Table*.

### 5.1 The *All Interval Series* problem

**The problem:** Given an integer  $n$ , the goal is to find a vector  $s = (s_1, \dots, s_n)$  such that  $s$  is a permutation of  $\mathbb{Z}_n = 0, 1, \dots, n-1$  and the interval vector  $v = (|s_2 - s_1|, |s_3 - s_2|, \dots, |s_n - s_{n-1}|)$  is a permutation of  $\mathbb{Z}_n = 1, \dots, n-1$ <sup>2</sup>

#### 5.1.1 Generation of the problem

In order to test the *All Interval Series* for a given parameter  $n$ , I have started by transforming the problem in a binary constraint problem (the absolute value is a ternary operator). As we have seen in our course, I have reasoned in term of a table indexed by an auxiliary parameter.

The generated problem is represented on the form of a text parsable by the parser detailed in [Section 4.1](#).

The variables:

- The vector  $v$  gives the variables  $v_1, \dots, v_{n-1}$  each of domain  $1, \dots, n-1$
- The vector  $s$  gives the variables  $v_1, \dots, v_n$  each of domain  $0, \dots, n-1$
- The auxiliary variables build the vector  $aux = (aux_1, \dots, aux_{n-1})$  each  $aux_i$  having the domain  $1, \dots, n^n$ .

The constraints:

- The *AllDiff* on  $v$ : each value  $val_i$  of  $v_i$  supports a variable  $val_j$  of  $v_j$  if  $val_i \neq val_j$
- The *AllDiff* on  $s$ : same reasoning for the variables of  $v$
- Each variable  $aux_i$  support a 3-tuple of variables:  $s_{i+1}, s_i$  and  $v_i$  if the absolute value of the difference between the value of  $s_{i+1}$  and the value of  $s_i$  equals the value of  $v_i$ .

An example of a generated print of this problem can be displayed through the command `dune exec -- main -all-int 3 -ac 3 -print-inp` for  $n = 3$ . The input of this problem is depicted in [Appendix B.1](#)

<sup>1</sup>The file `./graphs/input_4.txt` is the one depicted in [Appendix A.1](#)

<sup>2</sup>Description taken from <https://www.csplib.org/Problems/prob007/>

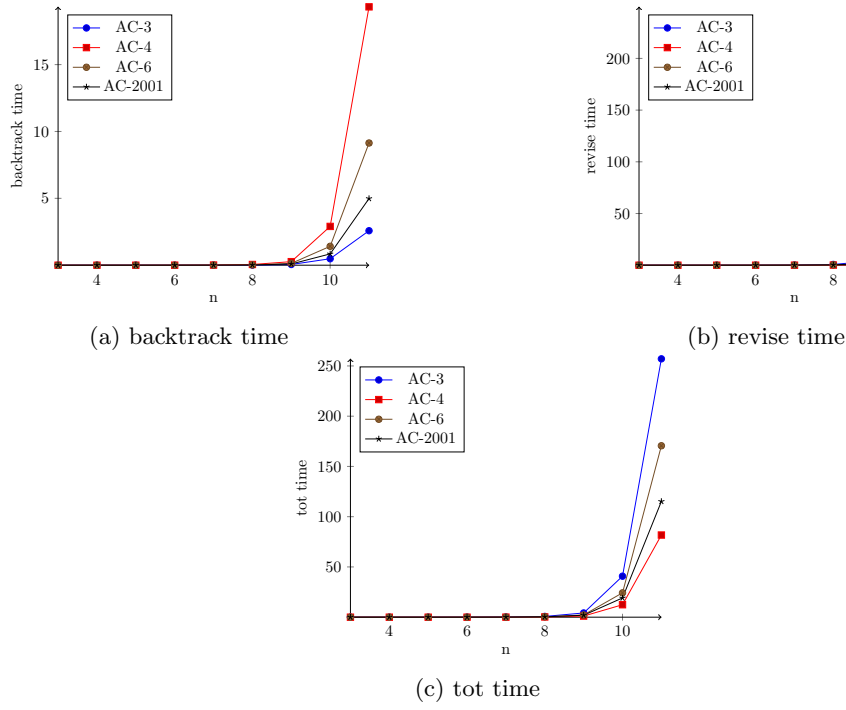


Figure 1: Time taken by AllIntervalSeries

### 5.1.2 Stats

The statistics of the run of the *All Interval Series* problem are shown in Figure 1. We see that *AC-4* passes a lot of time doing backtracks, since it is the filtering performing the most modifications on the internal data structure, but the time spent in backtracks is negligible compared to the revise time, since the *tot time* curve (Figure 1c) has the same shape of the *revise time* curve (Figure 1b).

We can finally see that *AC-4* outperforms the other filtering algorithms since its data structure allows it to efficiently know if a value is still supported. On the other hand, *AC-3* has no other way to loop over all the values of each domain, and since, for example, the domain of the *aux* variables can be potentially huge but strongly constrained, *AC-3* must loop a lot over this domain to know if a value  $v_i$  or  $s_i$  has a support in *aux*.

The other two algorithms, *AC-6* and *AC-2001* have better performances than *AC-3*, because, again, they both keep a trace of the support of each domain allowing to speed up the overall performance.

## 5.2 The *N-Queens* problem

**The problem:** Given an integer  $n$ , representing the size of a  $n \times n$  chessboard. The goal is to place a queen on each column of the chessboard such that there is no two queens on the same row and the same diagonal<sup>3</sup>

### 5.2.1 Generation of the problem

This problem is already a binary constraint satisfaction problem. The variables:

- The vector of columns  $c = (c_1, \dots, c_n)$  of domains  $1, \dots, n$

The constraints:

- Given two columns  $c_i, c_j \in c$ , the value  $val_i$  of  $c_i$  supports the value  $val_j$  of  $c_j$  if  $val_i \neq val_j$  (the *AllDiff* on the rows) and  $|val_i - val_j| \neq |i - j|$  (the *AllDiff* on the diagonals).

An input example of this problem with  $n = 3$  is provided in Appendix B.2

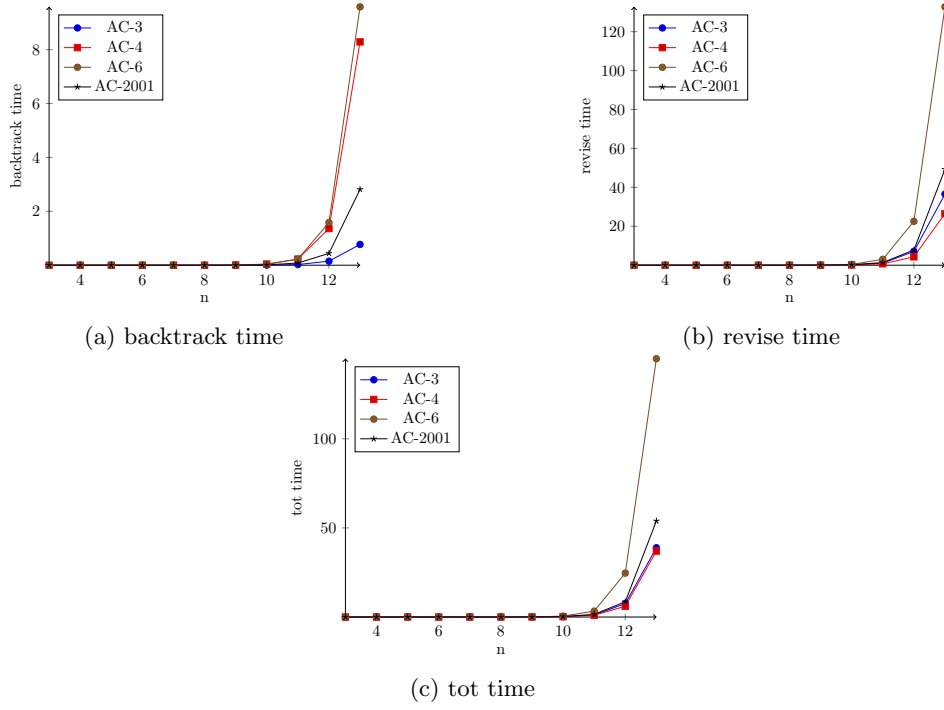


Figure 2: Time taken by Queens

### 5.2.2 Stats

The statistics of the  $N$ -Queens problem are given in Figure 2. In this implementation, we can see that *AC-4* has again good performances. A big difference wrt the *All Interval Series* results is that *AC-3* has a good behavior since it is the second fastest algorithm. This is mainly because the constraints in the  $N$ -Queens problem are homogeneously distributed for each variable: this is due to the high number of symmetries of the problem. *AC-6* has by far the slowest performances. I think this is due to its implementation that is not very efficient. I have tried to make a lot of variation of implementation for this filtering algorithm but I have not been able to improve it any more. This algorithm is implemented making a lot interaction with the *Hash-Table* which is known to be slow.

Interestingly, we can see that *AC-2001* is slower than *AC-3*, even if *AC-2001* stores the information about the last support of each value. This is due to the use of *Hash-Tables* which, again, slows down the performances.

## 6 Conclusion

In my opinion, even if it stores a lot of data *AC-4* is a good algorithm: in a first time it may be difficult to implement its data structure, in my case, I have used a lot the *dll* data structure and lots of pointer going from one side to the other of related variables, but, in a second time, we can see its good performances. This project allowed me to understand deeply the principle of backtracking a state in dynamic scenario such as the propagation of filtering values from domains. Moreover, since I wanted to improve the speed of the algorithms, I have been faced to the low performances of the standard libraries of *OCaml* and I have finally been able to use and understand the *Base* module of this programming language.

<sup>3</sup>Description taken from <https://www.csplib.org/Problems/prob054/>

## A An output example

### A.1 Example file to parse

```

d1: a b c
d2: 2 3 4 5
d3: e f g h
--
d1 b d2 2;
d1 c d2 2;
d1 c d2 3;
d2 2 d3 f;
d2 3 d3 e;
d2 3 d3 f;
d2 3 d3 g;
d2 4 d3 e;
d2 5 d3 e;

```

### A.2 Example of output

```

Initialization : removing a from d1
Initialization : removing 4 from d2
Initialization : removing 5 from d2
Initialization : removing h from d3
The data structure is:
node : (d2,2), last : (d1,b)
node : (d2,2), last : (d3,f)
node : (d3,f), last : (d2,2)
node : (d1,c), last : (d2,2)
node : (d1,b), last : (d2,2)
node : (d3,e), last : (d2,3)
node : (d3,g), last : (d2,3)
node : (d2,3), last : (d1,c)
node : (d2,3), last : (d3,e)
The domains are
-- Start Domains --
d1 : b;c;
d2 : 2;3;
d3 : e;f;g;
--- End Domains ---
-----
--> Selecting b from d1
* Removing c from d1
List of values having no more support = [(d2,3)]
* Removing 3 from d2
List of values having no more support = [(d3,g), (d3,e)]
* Removing g from d3
List of values having no more support = []
* Removing e from d3
List of values having no more support = []
--> Selecting 2 from d2
--> Selecting f from d3
A solution : [(d3,f), (d2,2), (d1,b)] !!
--> Selecting c from d1
* Removing b from d1
List of values having no more support = []
--> Selecting 2 from d2
* Removing 3 from d2
List of values having no more support = [(d3,g), (d3,e)]
* Removing g from d3

```

```
List of values having no more support = []
* Removing e from d3
List of values having no more support = []
--> Selecting f from d3
A solution : [(d3,f), (d2,2), (d1,c)] !!
--> Selecting 3 from d2
* Removing 2 from d2
List of values having no more support = []
--> Selecting e from d3
* Removing f from d3
List of values having no more support = []
* Removing g from d3
List of values having no more support = []
A solution : [(d3,e), (d2,3), (d1,c)] !!
--> Selecting f from d3
* Removing e from d3
List of values having no more support = []
* Removing g from d3
List of values having no more support = []
A solution : [(d3,f), (d2,3), (d1,c)] !!
--> Selecting g from d3
* Removing e from d3
List of values having no more support = []
* Removing f from d3
List of values having no more support = []
A solution : [(d3,g), (d2,3), (d1,c)] !!
-----
The number of fails is 0
The number of solutions is 5
Total Time: 0.000270
Time of backtracks: 0.000002
Time of revise: 0.000015
-----
```

## B Concrete problems' inputs

### B.1 A *All Interval Series* input

```
v1 : 1 2
v2 : 1 2
aux1 : 1 2 3 4 5 6 7 8 9
aux2 : 1 2 3 4 5 6 7 8 9
1: 0 1 2
2: 0 1 2
3: 0 1 2
--
# The all diff on the v vector
v1 1 v2 2
v1 2 v2 1
v2 1 v1 2
v2 2 v1 1
# the table constraint
aux1 2 1 1
aux1 2 2 0
aux1 2 v1 1
aux1 3 1 2
aux1 3 2 0
aux1 3 v1 2
aux1 4 1 0
```

```
aux1 4 2 1
aux1 4 v1 1
aux1 6 1 2
aux1 6 2 1
aux1 6 v1 1
aux1 7 1 0
aux1 7 2 2
aux1 7 v1 2
aux1 8 1 1
aux1 8 2 2
aux1 8 v1 1
aux2 2 2 1
aux2 2 3 0
aux2 2 v2 1
aux2 3 2 2
aux2 3 3 0
aux2 3 v2 2
aux2 4 2 0
aux2 4 3 1
aux2 4 v2 1
aux2 6 2 2
aux2 6 3 1
aux2 6 v2 1
aux2 7 2 0
aux2 7 3 2
aux2 7 v2 2
aux2 8 2 1
aux2 8 3 2
aux2 8 v2 1
# The all diff on the s vector
1 0 2 1
1 0 2 2
1 0 3 1
1 0 3 2
1 1 2 0
1 1 2 2
1 1 3 0
1 1 3 2
1 2 2 0
1 2 2 1
1 2 3 0
1 2 3 1
2 0 1 1
2 0 1 2
2 0 3 1
2 0 3 2
2 1 1 0
2 1 1 2
2 1 3 0
2 1 3 2
2 2 1 0
2 2 1 1
2 2 3 0
2 2 3 1
3 0 1 1
3 0 1 2
3 0 2 1
3 0 2 2
3 1 1 0
```

3 1 1 2  
3 1 2 0  
3 1 2 2  
3 2 1 0  
3 2 1 1  
3 2 2 0  
3 2 2 1

## B.2 A *N-Queens* input

1:1 2 3  
2:1 2 3  
3:1 2 3  
--  
1 1 3 2  
1 1 2 3  
2 1 1 3  
2 1 3 3  
3 1 1 2  
3 1 2 3  
1 2 3 1  
1 2 3 3  
3 2 1 1  
3 2 1 3  
1 3 2 1  
1 3 3 2  
2 3 1 1  
2 3 3 1  
3 3 2 1  
3 3 1 2