MASTER IN COMPUTER SCIENCE

COURSE : CONSTRAINT PROGRAMMING

Binary Constraint Solver in OCaml

Author:
Fissore Davide

Supervisor:
Jean-Charles Régin

Februray 2023

Contents

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 our Constraint Programming course provided by Mr. Régin, a constraint satisfaction problem is a problem made of a set of variables $V = \{V_1, ..., V_n\}$ 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 its corresponding variable.

A constraint is a relation between a set of values taken from the domains domains such that, in each constraint, each domain appears 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 represents 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 value $v_i \in D_i$ of the variable V_i is supported in the hyper-graph if for each constraint c_i involving the domains $D(C) = \{D_1, \ldots, D_n\}$ 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 having a constraint with D_i and remove all the values in D_j that are no more supported; these values belongs to the so-called *delta domains*. We repeat this operation for each value in the *delta domains* until the *delta domains* is not empty. 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 by backtracking to the state before v_i was chose and we select a new value in D_i different from v_i. If 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 not producing an empty domain, it means we have found a solution.

2 Binary constraints and arc consistency

An interesting property of the constraint satisfaction problems is that they can always be rewritten in an equivalent problem having only binary constraint. A binary constraint is a constraint relating only two variables. Thanks to this strategy the graph of the problem 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 domains and returning the *delta domains* 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 the 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 the *delta domains*. 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 variable v_j depends 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 values v_j supporting v_i, we only store the first value in each domain supporting v_i. In this way, when a value $v_i \in D_i$ is removed, we look for the values v_j supported by v_i (this is called the *s-list*). Then, we look for a new support in D_j starting from the value v_i, if this new support doesn't exists, v_j will be returned.

AC-2001 In order to use the minimum amount of space, *AC-2001* stores for each value $v_i \in D_i$ the first value v_j for each domain D_j having a constraint with D_i such that v_j and v_i are supporting each other. When v_i is removed, we look for the values v_j having has last support v_i and for them we seek a new support starting from v_i. If such support doesn't exist v_j can be removed from D_j.

3 My Implementation

I have developed my solver in OCaml (v. 4.13.1) using 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 the most important data structure I have implemented. This implementation is available at https://github.com/FissoreD/Binary-Constraint-Solver/.

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. 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 the *None* type. the *dll*s 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* mutable optional fields. The *dll*, itself, is represented by the type *t* (following the OCaml convention) containing a sentinel pointing on the first and the last element of the *dll*.

The nodes (resp. the *dll*s) are uniquely represented by their *id* field (resp. *id_dom*) in order to find them quickly when looking inside *Hash-Tables*.

The id of those records are created through the generator:

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

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

Moreover, 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 of a *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.

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

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

A domain is essentially a *dll* of values and a value is nothing but a node of a *dll*. A relation is a function returning if two values are related, that is, there exists a constraint between them.

The type graph is a record containing a *Hash_table* 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 (a relation can be seen as an edge of the graph); the *constraint_binding* is a *Hash-Table* associating to each domain D_i the set of domain with a constraint with D_i.

We can add constraints between values through the auxiliary function:

```
let add_constraint (graph : 'a graph) v1 d1 v2 d2 =
  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 as 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. If not present, d_1 and d_2 are inserted to the list of domains. Finally, the constraint between the node v_1 and v_2 is added.

3.3 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. Moreover, the AC algorithms should define the type of their internal data structure and the type of the stack operation to use when the solver backtracks.

3.3.1 *AC-3*

This algorithm has no data structure, therefore, its only useful 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.

The initialization step is made by filtering all the values having no support.

3.3.2 *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 *Hash-Table* $\mathcal{H}_1$ associating to each value v_i of each domain D_i a second *Hash-Table* $\mathcal{H}_2$. $\mathcal{H}_2$ associates to each domain D_j having a constraint with D_i a *dll* 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.

3.3.3 *AC-6*

The *AC-6* algorithm is the one that caused me the most problems 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 $v_i \in D_i$ 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 rapidly find which last values should be modified when a value is deleted.

- the last value is a *Hashtbl* associating to each domain D_j having a relation with D_i the first value in D_j supporting v_i.

3.3.4 *AC-2001*

```
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 *last* field allowing to improve the new support search after the deletion of a value $v_i \in D_i$. As we can see, the *last* field is a *Hashtbl* associating to each couple v_i, D_j (where D_j is a domain with a constraint with the domain of v_i) the first support in D_j for the node v_i. When a node v_i is remove in D_i, we loop for all the domains D_j 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.

3.4 The solver

The solver is the main function 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 ->
    ?only_stats:bool ->
    ?only_valid:bool ->
    ?verbose:bool ->
    ?one_sol:bool ->
    unit ->
    unit
end
```

These function aim to respectively initiate the problem inside the solver taking a graph $\mathcal{G}$ in input, and to find one or all the solutions obtainable from $\mathcal{G}$. All the optional argument of the *find_solution* method want to parametrize the solver; they 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 modification made inside the domains and inside the Arc-Consistency algorithm. This stack is used in order to backtrack.

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_op* deleted values. Inside this function we also update the *delta domains* returned by the filtering algorithm in order to propagate.

propagation_remove_by_node: this is a recursive function which propagates the deletion of a value v_i. The propagation is repeated by calling the *remove_by_node* function until the *delta domains* is not empty.

propagation_select_by_node: is a function taking the selected value v_i which calls the *propagation_remove_by_node* for all the values inside D_i that are different from 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 back the previous state during the backtrack phase.

4 Run the project

To run the project, you have to install all the dependencies via the command `./dependencies.sh` (this file should be executable: `chmod u+x ./dependencies.sh`).

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 print mode, the arc consistency algorithm *etc.*

Here a list of all the optional parameters and their behavior:

```
-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
-all-int [N] Set the size of the allIntervalSeries solver
-only-stats  Only print the number of fails and the number of solutions
-only-valid  Print only the valid solutions (not the fails)
-d           Debug mode
-print-inp   Print the input graph
-help        Display this list of options
--help       Display this list of options
```

The `-f` parameter takes a file path that will be used as the input graph of the problem. The file should respect the grammar proposed in Section 4.1.

4.1 The parser

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

```
start := variables* "\n--\n" constraints*
variables := v_name ":" (value " ")* "\n"
constraint := v_name " " value " " v_name " " value "\n"

v_name := [a-zA-Z0-9]*
value := [a-zA-Z0-9]*
```

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` runs *AC-2001* in verbose mode from the file `./graphs/input_4.text` and produces the result proposed in Appendix A.2 [1]

- 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. 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 four filtering algorithms on the *All Interval Series* and the *N-Queens* problems. The performances of each *AC* algorithm will depend on the efficiency of my implementation. Note that 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 $s = (s_1, \ldots, s_n)$ such that s is a permutation of $\mathbb{Z}_n = 0, 1, \ldots, n-1$ and the interval vector $v = (|s_2 - s_1|, |s_3 - s_2|, \ldots, |s_n - s_{n-1}|)$ is a permutation of $\mathbb{Z}_n = 1, \ldots, n-1$[2]

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.

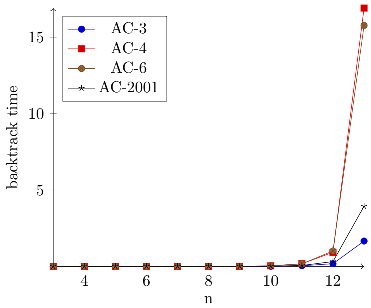
The variables:

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

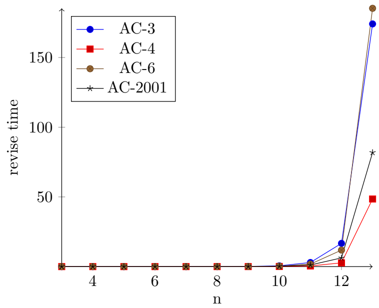
The constraints:

- 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.

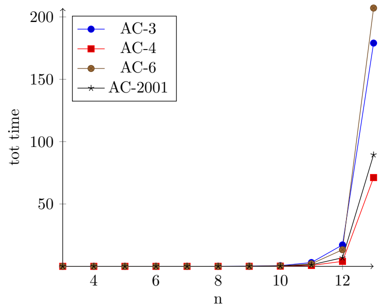
An example of a generated print of this problem can be displayed through the command `dune exec -- main -all-int 3 -ac 3 -print-inp` for $n = 3$. The input of this problem is depicted in Appendix B.1



(a) backtrack time (in sec)



(b) revise time (in sec)



(c) tot time (in sec)

Figure 1: Time taken by AllIntervalSeries

5.1.2 Stats

The statistics of the run of the *All Interval Series* problem are show in Figure 1. We see that *AC-4* spends a lot of time doing backtracks, since it is the filtering algorithm 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 to efficiently know if a value is still supported. On the other hans, *AC-3* has no other way to loop over all the values of each domains, and since, for example, the domain of the *aux* variables can be potentially huge but strongly constraint, *AC-3* must loop a lot to know if a value aux_i has a support in v_i, s_i and s_{i+1}.

AC-2001 has better performances then *AC-3*, because, they both keep a trace of the support of each domain allowing to speed up the overall performance.

I want to underline that *AC-6* has slow performances since it is an algorithm accessing a lot the *Hash-Table*s of the internal data structure. I have tried to make a lot of variation of this filtering algorithm but I have not been able to improve it any more.

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[3]

5.2.1 Generation of the problem

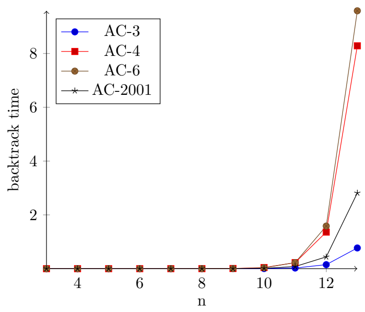
This problem is already a binary constraint satisfaction problem. The variables:

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

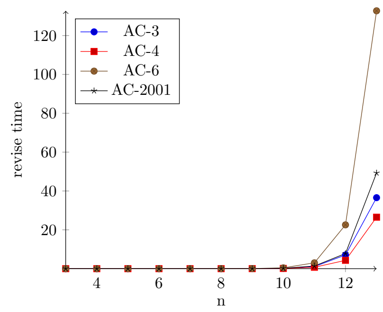
The constraints:

- 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).

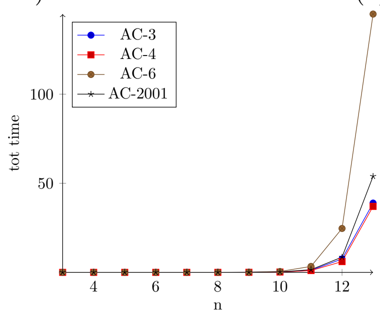
[1] The file `./graphs/input_4.text` is the one depicted in Appendix A.1
[2] Description taken from https://www.csplib.org/Problems/prob007/
[3] Description taken from https://www.csplib.org/Problems/prob054/



(a) backtrack time (in sec)



(b) revise time (in sec)



(c) tot time (in sec)

Figure 2: Time taken by Queens

An input example of this problem with $n = 4$ is provided in Appendix B.2

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.

Interestingly, we can see that *AC-2001* is slower that *AC-3*, even if *AC-2001* stores the information about the last support of each value. I think that, again, the use of *Hash-Tables* in *AC-2001* to store the last values slows down its performances.

As for the previous section, *AC-6* has by far the slowest 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 conceive and implement its data structure. In my case, I have used a lot the *dll* data structure and pointers 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 a dynamic scenario such as the propagation of the filtering operation. 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.

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 An *All Interval Series* input

```
# The s variables
s1: 0 1 2
s2: 0 1 2
s3: 0 1 2
# The v variables
v1: 1 2
v2: 1 2
# The aux variables
aux1: 1 2 3 4 5 6 7 8 9
aux2: 1 2 3 4 5 6 7 8 9
--
# The allDiff on the s vector
s1 0 s2 1;
s1 0 s2 2;
s1 0 s3 1;
s1 0 s3 2;
s1 1 s2 0;
s1 1 s2 2;
s1 1 s3 0;
s1 1 s3 2;
s1 2 s2 0;
```

```
s1 2 s2 1;
s1 2 s3 0;
s1 2 s3 1;
s2 0 s1 1;
s2 0 s1 2;
s2 0 s3 1;
s2 0 s3 2;
s2 1 s1 0;
s2 1 s1 2;
s2 1 s3 0;
s2 1 s3 2;
s2 2 s1 0;
s2 2 s1 1;
s2 2 s3 0;
s2 2 s3 1;
s3 0 s1 1;
s3 0 s1 2;
s3 0 s2 1;
s3 0 s2 2;
s3 1 s1 0;
s3 1 s1 2;
s3 1 s2 0;
s3 1 s2 2;
s3 2 s1 0;
s3 2 s1 1;
s3 2 s2 0;
s3 2 s2 1;
# The allDiff 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 s1 1;
aux1 2 s2 0;
aux1 2 v1 1;
aux1 3 s1 2;
aux1 3 s2 0;
aux1 3 v1 2;
aux1 4 s1 0;
aux1 4 s2 1;
aux1 4 v1 1;
aux1 6 s1 2;
aux1 6 s2 1;
aux1 6 v1 1;
aux1 7 s1 0;
aux1 7 s2 2;
aux1 7 v1 2;
aux1 8 s1 1;
aux1 8 s2 2;
aux1 8 v1 1;
aux2 2 s2 1;
aux2 2 s3 0;
aux2 2 v2 1;
aux2 3 s2 2;
aux2 3 s3 0;
aux2 3 v2 2;
aux2 4 s2 0;
aux2 4 s3 1;
```

```
aux2 4 v2 1;
aux2 6 s2 2;
aux2 6 s3 1;
aux2 6 v2 1;
aux2 7 s2 0;
aux2 7 s3 2;
aux2 7 v2 2;
aux2 8 s2 1;
aux2 8 s3 2;
aux2 8 v2 1;
```

B.2 A *N-Queens* input

```
# Columns
col1: 1 2 3 4
col2: 1 2 3 4
col3: 1 2 3 4
col4: 1 2 3 4
--
# Constraints
col1 1 col3 2;
col1 1 col4 2;
col1 1 col2 3;
col1 1 col4 3;
col1 1 col2 4;
col1 1 col3 4;
col2 1 col4 2;
col2 1 col1 3;
col2 1 col3 3;
col2 1 col1 4;
col2 1 col3 4;
col2 1 col4 4;
col3 1 col1 2;
col3 1 col2 3;
col3 1 col4 3;
col3 1 col1 4;
col3 1 col2 4;
col3 1 col4 4;
col4 1 col1 2;
col4 1 col2 2;
col4 1 col1 3;
col4 1 col3 3;
col4 1 col2 4;
col4 1 col3 4;
col1 2 col3 1;
col1 2 col4 1;
col1 2 col3 3;
col1 2 col4 3;
col1 2 col2 4;
col1 2 col4 4;
col2 2 col4 1;
col2 2 col4 3;
col2 2 col1 4;
col2 2 col3 4;
col3 2 col1 1;
col3 2 col1 3;
col3 2 col2 4;
col3 2 col4 4;
col4 2 col1 1;
```

```
col4 2 col2 1;
col4 2 col1 3;
col4 2 col2 3;
col4 2 col1 4;
col4 2 col3 4;
col1 3 col2 1;
col1 3 col4 1;
col1 3 col3 2;
col1 3 col4 2;
col1 3 col3 4;
col1 3 col4 4;
col2 3 col1 1;
col2 3 col3 1;
col2 3 col4 2;
col2 3 col4 4;
col3 3 col2 1;
col3 3 col4 1;
col3 3 col1 2;
col3 3 col1 4;
col4 3 col1 1;
col4 3 col3 1;
col4 3 col1 2;
col4 3 col2 2;
col4 3 col1 4;
col4 3 col2 4;
col1 4 col2 1;
col1 4 col3 1;
col1 4 col2 2;
col1 4 col4 2;
col1 4 col3 3;
col1 4 col4 3;
col2 4 col1 1;
col2 4 col3 1;
col2 4 col4 1;
col2 4 col1 2;
col2 4 col3 2;
col2 4 col4 3;
col3 4 col1 1;
col3 4 col2 1;
col3 4 col4 1;
col3 4 col2 2;
col3 4 col4 2;
col3 4 col1 3;
col4 4 col2 1;
col4 4 col3 1;
col4 4 col1 2;
col4 4 col3 2;
col4 4 col1 3;
col4 4 col2 3;
```