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 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 the 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 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 $D(C) = \{d1, \ldots, d_n\}$ 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 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_1. 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 5.1.2

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.

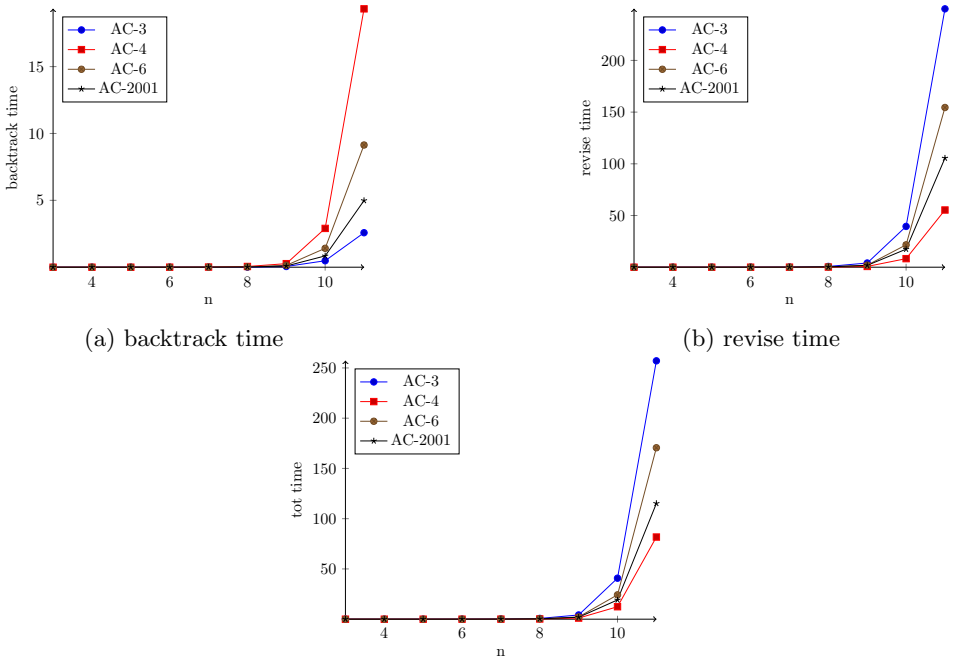
4.1 The parser

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



(a) backtrack time

(b) revise time

(c) tot time

Figure 1: Time taken by AllIntervalSeries

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[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. 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.

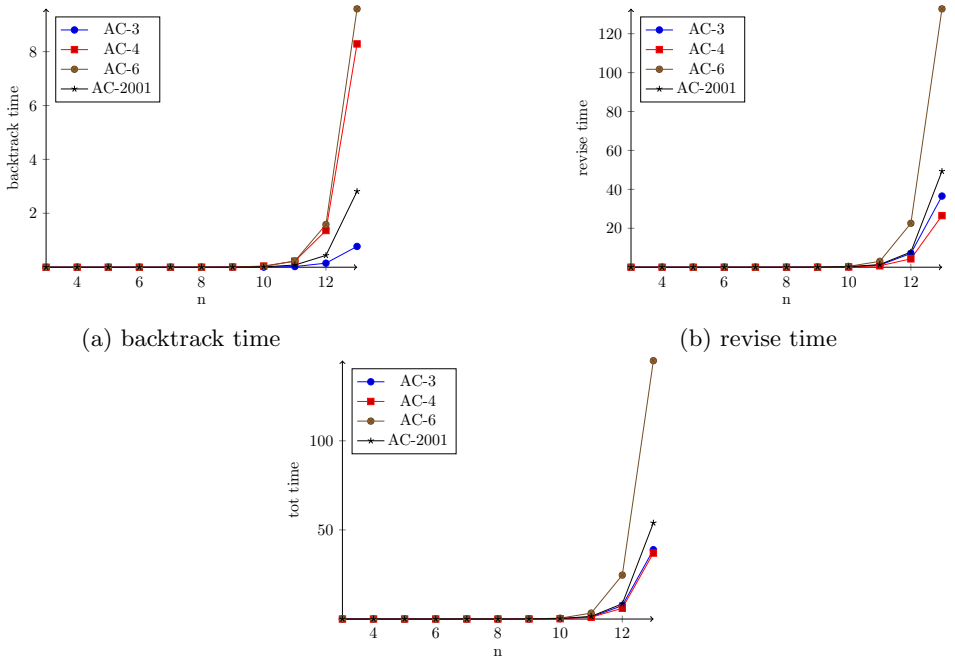
5 Benchmark

5.1 AllIntervalSeries problem

5.1.1 Stats

5.1.2 Generation of the problem

[1]The file `./graphs/input_4.txt` is the one depicted in Appendix A.1



(a) backtrack time

(b) revise time

(c) tot time

Figure 2: Time taken by Queens

5.2 Queens problem

5.2.1 Stats

5.2.2 Generation of the problem

6 Conclusion

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
------------------------------
```