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 $D(C) = \{d1, \dots, d_n\}$ 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_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 **??**

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

type 'a graph = {
  tbl : 'a table_type;
  relation : 'a relation;
  constraint_binding : (string, 'a domain DLL.t) Hashtbl.t;
  domains : (string, 'a DLL.t) Hashtbl.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.id_dom
        ~default:(fun _ ->
          Hashtbl.add_exn graph.domains ~key:d1.name ~data: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 **??**.

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 DLL.node) option Stack.t
  val backtrack_mem : 'a stack_type
  val stack_op : 'a stack_type
  val remove_by_node : ?verbose:bool -> string DLL.node -> unit
  val propagation_remove_by_node : ?verbose:bool -> string DLL.node -> unit
  val propagation_select_by_node : ?verbose:bool -> string DLL.node -> 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 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