C.Dürr — USACH — Nov 2012 — version 1

1 [Wed7] Constraint based programming

Constraint satisfaction problems (CSP) is a very general class of problems. People hoped that one could simply state a problem in a general language and a universal solver would just solve the instances. Of course this did not turned out exactly as simple. So today research on CSP is focused on exploiting particular structures of problems to speed-up search for solutions.

Still it remains a quite useful method to solve problems exactly, often in reasonable time. The general idea is to explore a search tree, and to use rules that permit to reduce the search space and to speed up the overal execution time.

1.1 The model

An instance of a constraint satisfaction problem consists typically of

- a finite set of variables, each of a finite domain[1],
- a finite set of constraints on these variables.
- the goal is to assign variables to values from their respective domain so to satisfy all constraints.

A constraint of arity k consists in a sequence of k variables and a relation on the cartesian product of their domains.

1.2 Example : Sudoku

Later we will see a better modelization.

Here we have 81 variables, each of domain $\{1,\ldots,9\}$. Every variable belongs to a column, a row and a block.

There are $81(8+6+6)/2$ binary constraints of the form $x \neq y$ for all pairs of variables x, y that belong to a same row, column or block.

Now there is a unary constraint of the form $x = v$ for every entry v in cell x of the initial grid.

1.3 Example : crossword

We are given a grid with blacked out cells, a dictionnary. Every set of adjacent and not blacked out cells in a same row or column must be assigned to a word of the dictionnary. Give two different modelizations for this problem.

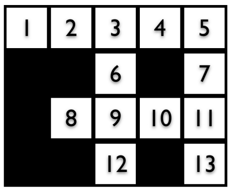

Figure 1. Example of a crossword grid

1. there exist infinite domain CSP also, which are very interesting, but not covered by this course

1.4 Example : coloring maps

We are given a polical map and need to assign to each country a color, such that adjancent countries have different colors.

1.5 Vocabulary

We denote the set of variables by $X = \{x_1, ..., x_n\}$ and their domains by $D = \{D_1, ..., D_n\}$. A constraint C_i is a relation R_i on a sequence of variables $S_i \subseteq X$ (order is important). Here S_i is the span of R_i. If $S_i = \{x_{i1}, ..., x_{\text{ir}}\}$ then $R_i \subseteq D_{i1} \times ... \times D_{\text{ir}}$. We write $C_i = \langle S_i, R_i \rangle$.

When searching for a feasible assignement, we gradually extend a partial assignement, backtracking if it is not possible. So we explore in DFS manner a search tree consisting of partial assignments. Of course we want to maintaint satisfaction of constraints which act on variables that have all been assigned.

1.6 Exercice: show that we can always assume constraints to be binary

Hint: Given an instance of a CSP, define a dual CSP, where primal variables become dual constraints and primal constraints dual variables.

1.7 Example : the n queen problem

A classical problem. We are given an $n \times n$ checker board, and need to place n queens on it, such that no two queens are in a same row, column, diagonal or anti-diagonal. This problem is well understood and there is always a explicit solution for all $n \geqslant 4$. (not sure about this constant). But it permits us to visualize the impact of improvements on a solver, since we have a simple parameter for the instance difficulty.

1.8 General structure of a solver

```
#solve: search exporation tree
def backtrack():
   if solved():
      return True
   x = selectVar()
   dom = var[i]                #save domain....
   for v in dom:
      if not consistant(x,v):
         continue
      var[x] = [v]
      if backtrack():
         return True
   var[x] = dom                #...restore domain
   return False
```

Several choices we need to make, and which could influence running time:

1. The invariant of this algorithm is consistancy of the assigned variables. How to check effiently if assigning x to value v is consistant?
2. What is the best choice of an unassigned variable. Smallest domain?
3. In what order order to loop over the domain?
4. How to restore domain at small cost?

1.9 Forward check

In order to avoid the time spend on the test consistant(x,v), we would like to maintain the invariant that for any non-assigned variable x, every value v in its domain, is consistant with the already assigned variables. So when assigning some variable y to some value u, then to maintain this invariant we need to remove from the domain of all non-assigned variables x every value v that would not be consistant with $y{:}=u$.

1. Give an example where the use of forward check would give a dramatic improvement on the running time.
2. One difficulty now is that when undoing the assignment $y{:}=u$ on needs to restore the domains of the unassigned variables. Propose an efficient way to do this.

1.10 Arc consistancy

From now, to simplify we assume that all variables have a domain of size k, and all constraints are binary. So the CSP can be viewed a a graph, where vertices are variables, and edges correspond to constraints. In fact we consider a directed graph, where there is an arc (x, y) if the variables x, y are tied by a constraint.

Now we say that for a value v in the domain of variable x, its support for variable y is the set of all values u such that the assignment $x{:}=v, y{:}=u$ would be consistant. Forward checking is nothing else than restricting the domains of unassigned variables to values that support assigned variables.

We would like to go one step further, and maintain as an invariant that all values in the domains of non-assigned variables have a support in *all* related variables, not only assigned variables. This property is called arc-consistancy.

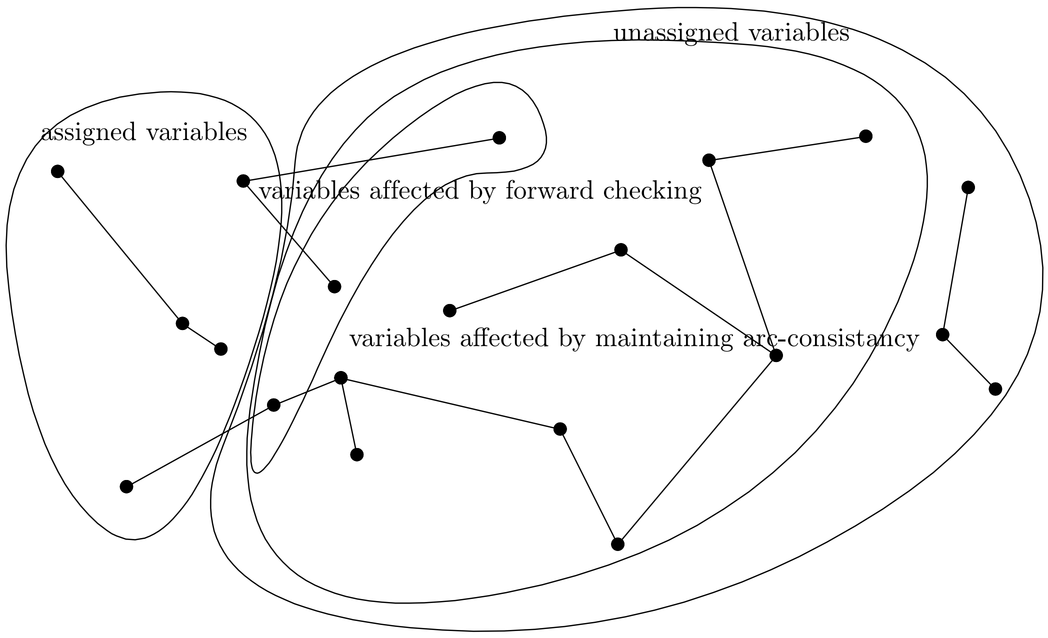
unassigned variables
assigned variables
variables affected by forward checking
variables affected by maintaining arc-consistancy

Figure 2. Variables whos domain is affected by forward-checking $\subseteq$ variables whos domain is affected by maintaining arc-consistancy $\subseteq$ unassigned variables

1.11 Revise

The main function for maintaining arc-constancy is a function called revise(x,y), which is called whenever the domain of y was restricted. It will remove from the domain of x all values that don't have a support in the domain of y. And it will return true if the domain of x has changed.

Forward checking will simply call revise for all non-assigned variables x that are related to the variable y which just has been assigned.

1.12 AC3

We maintaint a set Q of variables whos domain has changed. Initially it contains only the variable y which has been assigned. Then while Q is non empty, we remove a variable x from Q, loop over all non-assigned variables z related to x, and add z to Q whenever revise(z,x) returns true.

1. Give an example where the usage of AC3 will have a dramatic improvement of the running time over the usage only of forward check.
2. Something has to be changed in the search procedure when using arc-consistancy rather than simple forward-check. Do you see what?

1.13 AC4

Testing whether some value in the domain of a variable has a support in the domain of another variable is costly because we have to loop over all values of its domain. So a better way to do this is to maintain a set of values support(x,v,y) which is the support of $x := v$ in the domain of y.

So when a value u is removed from the domain of y, then we need to remove it from support(x,v,y) for all related variables x and values x in its domain. This time the set Q is not a set of variables that is maintained but a set of variable/values pairs. It contains all pairs (x, v) where value v has to be removed from the domain of x. Clearly whenever the support support(x,v,y) becomes empty, the pair (x, v) has to be added to Q.

1. Give an example where the usage of AC4 will have a dramatic improvement over AC3.
2. Suppose we have n variables, each has a domain of size m, and we have t constraints. What is the worst case complexity of the procedures AC3 and AC4?

2 [Thu8] Dancing Links

2.1 A motivating example : 16x16 Soduku

We consider the problem of solving 16×16 Sudoku instances. There are more difficult to solve then regular 9×9 instances. A classical approach to solve them is to use backtracking on cells: We choose a cell c whith a minimal of allowed values. Then we loop over all allowed values v and try to solve the subinstance when the value of c is set to v. For this we loop through all cells d on which c is in conflict, and remove v from the allowed values. Now it happens that this approach is just to slow. So we need to use the Rolce-Royce of backtracking algorithms, the Dancing Links algorithm. To be precise it is a clever implementation of the classical backtracking algorithm.

2.2 Exact Cover Problem

We consider a general cover problem. We are given a universe U, and some sets $S_1, ..., S_n \subseteq U$. The goal is to pick a collection of these sets, such that every element of the universe is contained in exactly of the selected sets. The sets form a partition of U.

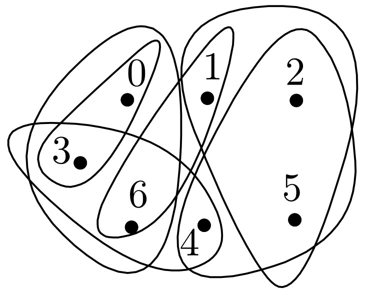

Figure 3. An instance of the exact set cover problem

2.3 Example: Sudoku

Every cell must contain exactly one value. In every row every value must be present exactly once. The same for the columns and the blocks.

So there are four kind of elements in the universe: rowcol(r,c), rowval(r,v), colval(c,v), blkval(b,c). And every assignement corresponds to a set. Assigning value v to the cell in row r, column c contains the elements {rowcol(r, c), rowval(r, v), colval(c, v), blkval(b, c)}, where b is the block containing the cell. If we choose this set to be part of our solution, then we make sure that there will be no other value assigned to the cell and that the value won't appear twice in the same row, column, block.

2.4 Backtracking

The backtracking search works like this. We choose an element of the universe, preferably one that is contained in a minimal number of sets. So for each set S, we make a recursive search on the sub-instance obtained by restricting the universe to $\overline{S}$ and to sets that are disjoint with S. If one of the calls succeed, and returned a solution A, then we can return the solution $A \cup \{S\}$ to the original instance.

2.5 Removing an element

The operation described above, of restricting the universe to $\overline{S}$ will be carried out on all elements $j \in S$. When removing j from the universe, we need to remove at the same time all sets that contain j. We call this operation *covering column j*.

2.6 Matrix representation

We represent instances by the binary membership matrix M. The columns are the elements of the universe, while the rows are the sets, and a 1 indicates a membership of an element to a set.

Now covering a column j translates as deleting from M the column j and all rows i with $M_{ij} = 1$.

If we translate the previous algorithm to this setting, then we obtain following. If the matrix has no column, the answer is the empty collection of course. Otherwise we choose a column j with a minimal number of 1's in it. Then we loop over all rows i with $M_{ij} = 1$. For each row i, the sub-instance consists of the matrix obtained from M by covering all columns k where $M_{ki} = 1$.

So the algorithm will spend all its time deleting rows and columns and restoring them when backtracking. Also it will spend quite a time looping over 1-entries in a row or in a column. If the matrix is parse this might take a while.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Figure 4. Matrix describing the instance of figure 3

2.7 Double chained lists

So the idea is to represent 1-entries as cells of a double chained list. Actually there will be a horizontal list and a vertical list. In addition there will a cell for each column, to have a starting point, and a single root cell to have a starting point for looping on the column header cells.

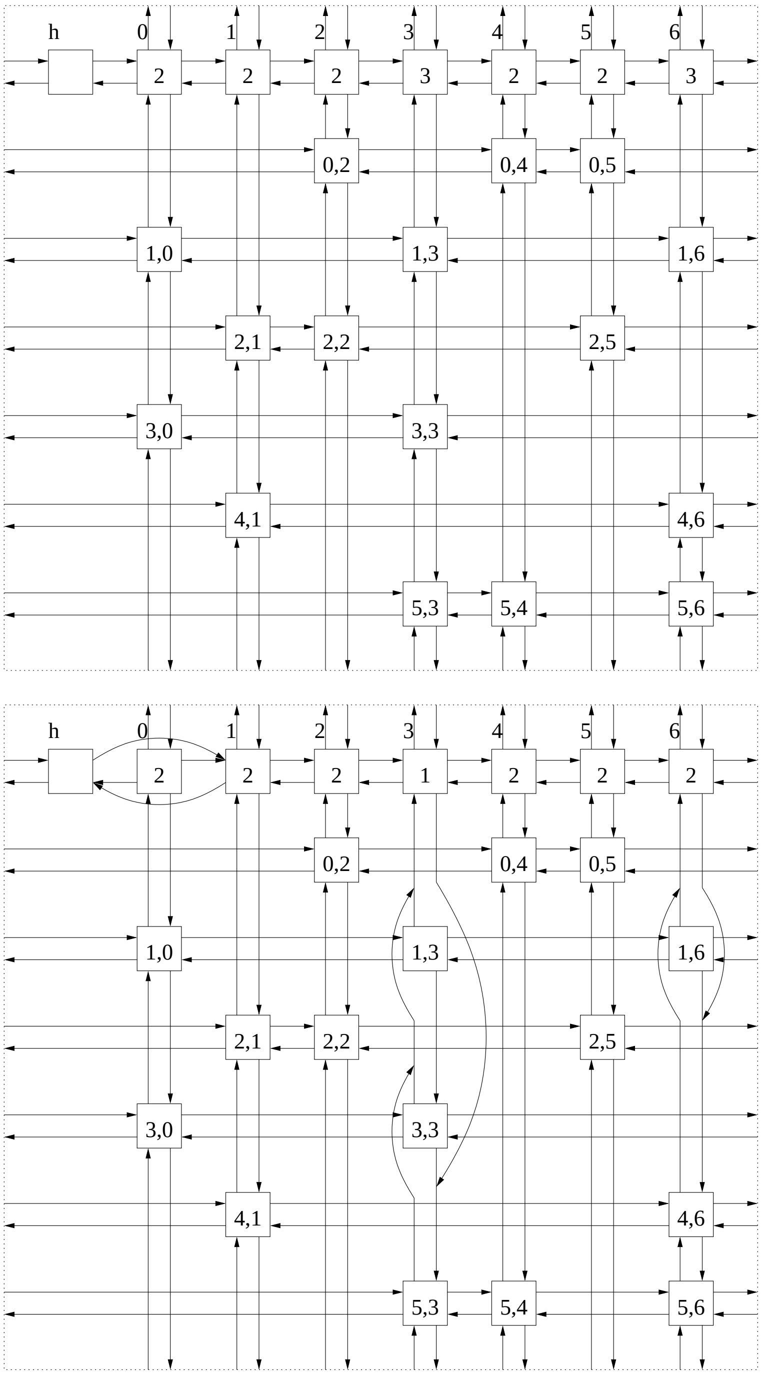

Figure 5. Linked list representation corresponding to the universe $\{0, 1, 2, 3, 4, 5, 6\}$ and the sets $\{2, 4, 5\}$, $\{0, 3, 6\}$, $\{1, 2, 5\}$, $\{0, 3\}$, $\{1, 6\}$, $\{3, 4, 6\}$. The image below shows the result of covering column 0.

Every cell will have four pointers U, D, L, R to point to the next cell in all four directions up, down, left and right, plus two integer attributes S, C that have different meanings for the 3 cell types. Removing a cell from the vertical list is done by these two instructions.

```
void hideVert() {
  U.D = D;
  D.U = U;
}
```

Now the key ingredient to this implementation is that the pointers D, U still point to their former neighbors in that list. So it is rather easy to add the cell again, see:

```
void unhideVert() {
  D.U = this;
  U.D = this;
}
```

So covering a column j is easy. We starting hiding horizontally the column header for column i, then for all entries in this column, we loop though the horizontal list and hide vertically every element in there. Note that we do not hide vertically the entries in column i itself. To uncover column j we just need to undo all these operations in the same manner. See figure 5 for an example.

2.8 Exercice: Implement Dancing Links

To implement this algorithm, we would need the follwing datastructures and methods.

Cell. Define a class Cell, with references to four other Cells U,D,L,R and two integers S,C. For the main header cells these integers will be ignored. For the column header cells, C will contain the column index and S the number of entries in this column. For regular cells, S will be row index and C the column index.

Constructor for Cell. Its parameters consists of two cells *left,below*, and two integers *_S,_C*. The two integers are used as initial values for S, C. The given cells are used to chain correctly the created cell. For example if *left* is null, then the cell will be a the unique cell of its horizontal list. Otherwise it will insert in the horizontal list, such that *left* becomes its leftneighbor.

hideVert, unhideVert, hideHoriz, unhideHoriz. As described above.

ExactCover. Define a class exact Cover. It should have an integer stack called *solution*, a cell h for the main header, and an array of column header cells *col*.

Constructor for ExactCover. The constructor should take an integer *universe* as argument, describing the size of the universe. It will initialize correctly h and *col*.

add. A method to add regular cells to the structure. It has two integer parameters r, c indicating row and column of the cell to be created. You have the promize that successive calls to add will be done in lexicographical order of (r, c). So you will need two variables, to remember the last creation of a cell, *last_row* will be the row where the last insertion has been done and *last_cell*, will be the last cell to be created. You can use this *last_cell* and *col[c]* to create correctly the cell.

cover, uncover. These methods take a column j as input, and do the operations described above.

solve. This method returns a boolean indicating if the instances could have been solved.

- If the instance is empty, return true
- Select column c will smallest number of entries.
- cover column c
- For every row r with an entry (r, c) in column c
  - Push r on the solution stack.
  - cover all columns $d \neq c$ with an entry (r, d) in row r
  - Make a recursive call to solve. If succes, return true.
  - uncover all columns d that were covered before.
  - Pop r from the solution stack.
- uncover column c
- return false.

2.9 Test your code

For a small test, we provide you with a main method to your class which will run your code on the running example of this course.

In order to make a test on larger instances, we provide also you with a class SolveSoduku that will test your implementation, toegether with 2 test files. You can call it from command line as `java Main < input.txt | diff - output.txt`. If there is no output, then your code is correct. If the command diff found differences then it could still be that your code is correct, as the solutions are not unique. In that case you can submit your code to this website: http://www.spoj.pl/problems/SUDOKU/. For this you would need to remove the *main* method from the ExactCover class (or rename it).