Module 4

Constraint satisfaction problems

Lesson 10

Constraint satisfaction problems - II

Version 1 CSE IIT, Kharagpur

4.5 Variable and Value Ordering

A search algorithm for constraint satisfaction requires the order in which variables are to be considered to be specified as well as the order in which the values are assigned to the variable on backtracking. Choosing the right order of variables (and values) can noticeably improve the efficiency of constraint satisfaction.

4.5.1 Variable Ordering

Experiments and analysis of several researchers have shown that the ordering in which variables are chosen for instantiation can have substantial impact on the complexity of backtrack search. The ordering may be either

- a *static ordering*, in which the order of the variables is specified before the search begins, and it is not changed thereafter, or
- a *dynamic ordering*, in which the choice of next variable to be considered at any point depends on the current state of the search.

Dynamic ordering is not feasible for all search algorithms, e.g., with simple backtracking there is no extra information available during the search that could be used to make a different choice of ordering from the initial ordering. However, with forward checking, the current state includes the domains of the variables as they have been pruned by the current set of instantiations, and so it is possible to base the choice of next variable on this information.

Several heuristics have been developed and analyzed for selecting variable ordering. The most common one is based on the "**first-fail**" principle, which can be explained as

"*To succeed, try first where you are most likely to fail.*"

In this method, the variable with the fewest possible remaining alternatives is selected for instantiation. Thus the order of variable instantiations is, in general, different in different branches of the tree, and is determined dynamically. This method is based on assumption that any value is equally likely to participate in a solution, so that the more values there are, the more likely it is that one of them will be a successful one.

The first-fail principle may seem slightly misleading, after all, we do not want to fail. The reason is that if the current partial solution does not lead to a complete solution, then the sooner we discover this the better. Hence encouraging early failure, if failure is inevitable, is beneficial in the long term. On the other end, if the current partial solution can be extended to a complete solution, then every remaining variable must be instantiated and the one with smallest domain is likely to be the most difficult to find a value for (instantiating other variables first may further reduce its domain and lead to a failure). Hence the principle could equally well be stated as:

"*Deal with hard cases first: they can only get more difficult if you put them off.*"

This heuristic should reduce the average depth of branches in the search tree by triggering early failure.

Another heuristic, that is applied when all variables have the same number of values, is to choose the variable which participates in most constraints (in the absence of more specific information on which constraints are likely to be difficult to satisfy, for instance). This heuristic follows also the principle of dealing with hard cases first.

There is also a heuristic for static ordering of variables that is suitable for simple backtracking. This heuristic says: choose the variable which has the largest number of constraints with the past variables. For instance, during solving graph coloring problem, it is reasonable to assign color to the vertex which has common arcs with already colored vertices so the conflict is detected as soon as possible.

4.5.2 Value Ordering

Once the decision is made to instantiate a variable, it may have several values available. Again, the order in which these values are considered can have substantial impact on the time to find the first solution. However, if all solutions are required or there are no solutions, then the value ordering is indifferent.

A different value ordering will rearrange the branches emanating from each node of the search tree. This is an advantage if it ensures that a branch which leads to a solution is searched earlier than branches which lead to death ends. For example, if the CSP has a solution, and if a correct value is chosen for each variable, then a solution can be found without any backtracking.

Suppose we have selected a variable to instantiate: how should we choose which value to try first? It may be that none of the values will succeed, in that case, every value for the current variable will eventually have to be considered, and the order does not matter. On the other hand, if we can find a complete solution based on the past instantiations, we want to choose a value which is likely to succeed, and unlikely to lead to a conflict. So, we apply the **"succeed first"** principle.

One possible heuristic is to prefer those values that maximize the number of options available. Visibly, the algorithm AC-4 is good for using this heuristic as it counts the supporting values. It is possible to count "promise" of each value, that is the product of the domain sizes of the future variables after choosing this value (this is an upper bound on the number of possible solutions resulting from the assignment). The value with highest promise should be chosen. Is also possible to calculate the percentage of values in future domains which will no longer be usable. The best choice would be the value with lowest cost.

Another heuristic is to prefer the value (from those available) that leads to an easiest to solve CSP. This requires to estimate the difficulty of solving a CSP. One method propose to convert a CSP into a tree-structured CSP by deleting a minimum number of arcs and then to find all solutions of the resulting CSP (higher the solution count, easier the CSP).

For randomly generated problems, and probably in general, the work involved in assessing each value is not worth the benefit of choosing a value which will on average be more likely to lead to a solution than the default choice. In particular problems, on the other hand, there may be information available which allows the values to be ordered according to the principle of choosing first those most likely to succeed.

4.6 Heuristic Search in CSP

In the last few years, greedy local search strategies became popular, again. These algorithms incrementally alter inconsistent value assignments to all the variables. They use a "repair" or "hill climbing" metaphor to move towards more and more complete solutions. To avoid getting stuck at "local optima" they are equipped with various heuristics for randomizing the search. Their stochastic nature generally voids the guarantee of "completeness" provided by the systematic search methods.

The local search methodology uses the following terms:

- **state (configuration):** one possible assignment of all variables; the number of states is equal to the product of domains' sizes
- **evaluation value:** the number of constraint violations of the state (sometimes weighted)
- **neighbor:** the state which is obtained from the current state by changing one variable value
- **local-minimum:** the state that is not a solution and the evaluation values of all of its neighbors are larger than or equal to the evaluation value of this state
- **strict local-minimum:** the state that is not a solution and the evaluation values of all of its neighbors are larger than the evaluation value of this state
- **non-strict local-minimum:** the state that is a local-minimum but not a strict local-minimum.

4.6.1 Hill-Climbing

Hill-climbing is probably the most known algorithm of local search. The idea of hill-climbing is:

1. start at randomly generated state
2. move to the neighbor with the best evaluation value
3. if a strict local-minimum is reached then restart at other randomly generated state.

This procedure repeats till the solution is found. In the algorithm, that we present here, the parameter Max_Flips is used to limit the maximal number of moves between restarts which helps to leave non-strict local-minimum.

Algorithm Hill-Climbing

```
procedure hill-climbing(Max Flips)
  restart: s <- random valuation of variables;
  for j:=1 to Max_Flips do
    if eval(s)=0 then return s endif;
    if s is a strict local minimum then
        goto restart
    else
      s <- neighborhood with smallest evaluation value
    endif
  endfor
  goto restart
end hill-climbing
```

Note, that the hill-climbing algorithm has to explore all neighbors of the current state before choosing the move. This can take a lot of time.

4.6.2 Min-Conflicts

To avoid exploring all neighbors of the current state some heuristics were proposed to find a next move. *Min-conflicts* heuristics chooses randomly any conflicting variable, i.e., the variable that is involved in any unsatisfied constraint, and then picks a value which minimizes the number of violated constraints (break ties randomly). If no such value exists, it picks randomly one value that does not increase the number of violated constraints (the current value of the variable is picked only if all the other values increase the number of violated constraints).

Algorithm Min-Conflicts

```
procedure MC(Max Moves)
  s <- random valuation of variables;
  nb_moves <- 0;
  while eval(s)>0 & nb moves<Max Moves do
    choose randomly a variable V in conflict;
    choose a value v' that minimizes the number of conflicts for V;
    if v' # current value of V then
      assign v' to V;
      nb_moves <- nb_moves+1;
    endif
  endwhile
  return s
end MC
```

Note, that the pure min-conflicts algorithm presented above is not able to leave local-minimum. In addition, if the algorithm achieves a strict local-minimum it does not perform any move at all and, consequently, it does not terminate.

4.6.3 GSAT

GSAT is a greedy local search procedure for satisfying logic formulas in a conjunctive normal form (CNF). Such problems are called SAT or k-SAT (*k* is a number of literals in each clause of the formula) and are known to be NP-c (each NP-hard problem can be transformed to NP-complex problem).

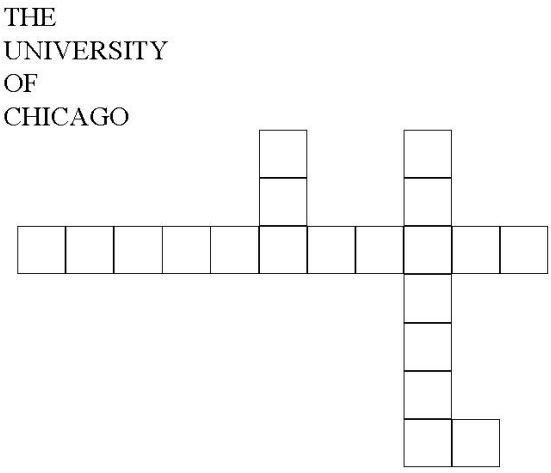
The procedure starts with an arbitrary instantiation of the problem variables and offers to reach the highest satisfaction degree by succession of small transformations called repairs or flips (flipping a variable is a changing its value).

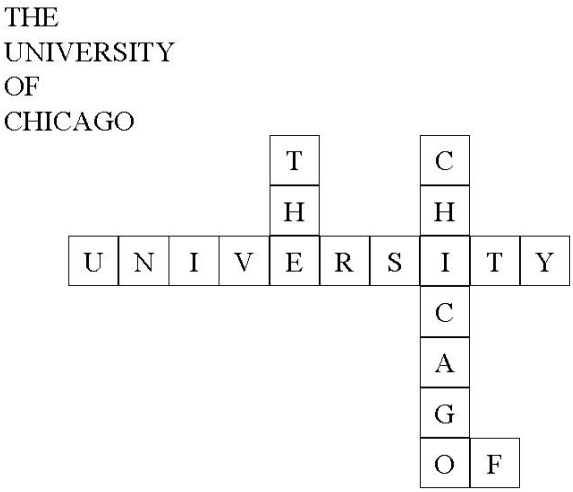
Algorithm GSAT

```
procedure GSAT(A,Max Tries,Max Flips)
  A: is a CNF formula
  for i:=1 to Max Tries do
    S <- instantiation of variables
    for j:=1 to Max_Iter do
      if A satisfiable by S then
        return S
      endif
      V <- the variable whose flip yield the most important raise in the
number of satisfied clauses;
      S <- S with V flipped;
    endfor
  endfor
  return the best instantiation found
end GSAT
```

Questions

1. Consider a variant of the crossword puzzle problem. In this variant, we assume that we have a set of words W1, W2,...Wn and a crossword puzzle grid. Our goal is to fill the crossword grid with the words such that letters of intersecting words match. An example of an uncompleted puzzle and a completed puzzle appear below.





Provide a constraint satisfaction problem formulation for this variant of the crossword puzzle problem.

a. Specify the variables to which values are to be assigned.
b. Specify the domains from which the variables take their values.
c. Define the constraints that must hold between variables. Please provide pseudo-code defining the constraints explicitly.
d. Give a simple example of a "fill-in" (crossword) puzzle of the type above that demonstrates the limitations of arc-consistency type constraint propagation for solving constraint satisfaction problems.

e. Explain why constraint satisfaction procedures based on backtracking search are not subject to this problem.
f. Briefly describe the use of the iterative refinement, min-conflict strategy to solve the crossword puzzle problem.
g. Demonstrate the application of your procedure on a simple 4 word example puzzle.

2. Consider the following classroom scheduling problem: There are 4 classes, C1, C2, C3, and C4, and 3 class rooms, R1, R2, and R3. The following table shows the class schedule:

Class	Time
C1	8am-10:30am
C2	9am-11:30pm
C3	10am-12:30pm
C4	11am-1:30pm

In addition, there are the following restrictions:

• Each class must use one of the 3 rooms, R1, R2, R3.

• R3 is too small for C3.

• R2 and R3 are too small for C4.

One way of formulating this problem as a constraint satisfaction problem is to let each class, C1, …, C4, be a variable, and each room, R1, R2, R3, be the possible values for these variables.

(a) Show the initial possible values for each variable, C1, …, C4, given the restrictions above.

(b) Express formally all the constraints in this problem.

(c) Consider each pair of variables appearing in the same constraint in (b), please point out which pairs are arc-consistent for the initial values provided in (a). For those pairs that are not arc-consistent, please provide the necessary operations so that they become arc-consistent.

Solution

1. A. Variables.
We use rows or columns of boxes as variables. In this case we have four variables H1,H2,V1,V2 (we use H for horizontal and V for vertical)

1. B. Domains
All the variables can take values out of the same domain D. In our case we define D = {THE, UNIVERSITY, OF, CHICAGO}

1. C. Constraints

We define two kinds of constraints.

Length constraints:

length(H1) == 10
length(H2) == 2
length(V1) == 3
length(V2) == 7

Cross constraints:

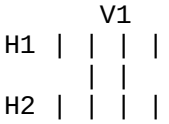
H1(5) == V1(3)
H1(8) == V2(3)
H2(1) == V2(7)

1.D. Arc consistency problems

There are two kinds of problems, when there are no legal assignments or there are more than one legal assignment.

Example of more than one legal assignment.

Assume three variables V1,H1,H2 taking values out of the domain D = {bit, its, hit, sit, ion, one}



after applying the arc consistency algorithm (page 146) the domains for each variable are equal to

D(H1) = {bit, hit, sit}
D(V1) = {ion}
D(H2) = {one}

There is more than one legal assignment for the variable H1

Example of no legal assignment.

In the previous example change the domain from D to E = {bit, its, hit, sit, ion }

1.E. Procedures based on backtracking do not have problems with multiple legal assignments because they pick the first one that satisfies the constraints without looking for more options. When there are no legal assignments, they search the whole space, then return a failure value

1. F.

(0). Start a counter of steps count := 0
(1). Assign to each variable a random word taken from its respective domain.
(2). Count the number of conflicts each assignment produced (number of constraints unsatisfied)
(3). Pick the variable with the highest number of conflicts and change its value until its number of conflicts is reduced
(4). Increase count by one

(5). If count is less than a predetermined maximum number of steps, repeat from step (2)

Note: Explain why it is necessary to have a maximum number of steps. Is this an optimal method?

1. G. Now that you have the algorithm, try it at home!

2.a. C1: { R1, R2, R3 } C2: { R1, R2, R3 } C3: { R1, R2 } C4: { R1 }

2.b C1 != C2, C1 != C3, C2 != C3, C2 != C4, C3 != C4
We may add C3 != R3, C4 != R2, C4 != R3 even though they are contained in (a).

2.c. All the five pairs of variables in the five binary constraints in (b) are not arc consistent. To make them consistent, we need to remove R1 from the domain of C3, R1, R2 from the domain of C2 and R2, R3 from the domain of C1.