Constraint Satisfaction Problems – Review

CS 575

September 13, 2005

1 Constraint Satisfaction Problems (CSP)

A *CSP* problem is defined by

1. a set of variables $X_1, \ldots, X_n$, each has a non-empty domain D_i, and
2. a set of *constraints* $C_1, \ldots, C_m$.

A constraint involves some (or all) variables and specifies the allowable combinations of values for that subset.

A *state* of the problem is defined by an *assignment* of values to some or all of the variables. A state is consistent if its corresponding assignment does not violate any of the constraints.

A complete assignment satisfiying all the constraints is a *solution* of the problem.

Note: Some CSP involves also the finding of a solution that maximizes (resp. minimizes) an object function.

Example 1.1 *The graph 3-coloring problem can be represented by a CSP with*

- *the set of nodes* $\{n_1, \ldots, n_m\}$ *as the set of variables;*
- *the domain of each variable* n_i *is the set* $\{red, blue, green\}$
- *the set of constraints* $\{n_i \neq n_j \mid (n_i, n_j)$ is an edge of the graph$\}$.

2 Classification of CSPs

Depending on the domains and the constraints, CSPs are said to be

1. discrete (or continuous): the domains of variables are disrete (or continuous)
2. finite (or infinite): the domains of variables are finite (or infinite)
3. linear (or nonlinear): the constraints are linear (or nonlinear)

Note: We will mostly working with **discrete and finite** CSPs. We will also discuss only **binary** CSPs. A CSP is binary if its constraints are either unary or binary constraints. Unary constraint restricts the value of a variable, for example, in the map-coloring problem, the constraint $n \neq red$ (or $n = red$) is a unary constraint; a constraint $n_i \neq n_j$ is a binary constraint.

A binary CSP can be represented by a constraint graph, where each node represents a variable, each link between two different nodes represents a binary constraint, and each link originating and terminating at the same node represents a unary constraint. (See csp.doc for example)

3 Solving CSPs by Search

A CSP problem can be viewed as a search problem as follows:

1. The empty assignment represents the initial state (initial node).
2. Any consistent assignment is a node in the search graph.
3. An edge connected two nodes N_1 and N_2 if $N_1 \subseteq N_2$.
4. Each solution is a final state of the search.

With this view, CSPs can be solved using standard searching algorithms.

4 Solving CSPs by Generate-And-Test

A CSP problem can also be solved by using the generate-and-test approach, i.e., repeatedly executing the following two steps

- **Step 1:** Generate a variable assigment
- **Step 2**: Test for solution; if found, return the solution; otherwise, go back to step 1.

The main disadvantage of this method is the huge number of possible states (might be infinite).

5 Backtracking Search for CSPs

The key idea of this method is to sequentially construct the solution. Instead of guessing the values for *all* variables at each search step, the algorithm guesses *one* variable at each step and *backtracks* when a variable has no legal values left to assign.

Backtracking search is depth-first search based on this idea, The algorithm is given next:

function BACKTRACKING-SEARCH(`csp`) % **return** a solution or failure
    return RECURSIVE-BACKTRACKING({}, `csp`)

function RECURSIVE-BACKTRACKING(`assignment`, `csp`) % **return** a solution or failure
    if `assignment` is complete **then return** assignment
    select a variable X in `csp` that has not been assigned a value
    for each *value* in the domain of X **do**
        if *value* is consistent with `assignment` **then**
            add $X = value$ to `assignemt`
            result = RECURSIVE-BACKTRACKING(`assignment`, `csp`)
                if result $\neq$ failure **then return** result
            remove $X = value$ from `assignemt`
    return failure

Note. The above algorithm resembles uninformed search. (Problem?)

CSP can be solved more efficient by addressing the following questions:

1. Which variable should be assigned next, and in what order should its values be tried?
2. What are the implication of the current variable assignment for the other unassigned variables?
3. When a path fails (backtracking point reached) can the search avoid repeating this failure in subsequent paths?

Variable and value ordering. This is a source for heuristic. One can select the variable with the fewest legal values (called **MRV**, limits the branching factor). This heuristic is useful when we have a non-empty assignment. For empty assigment, **degree heuristic** is useful: selecting the variable involving in the largest number of constraints. Once a variable is selected, the **least-constraining-value** is the heuristic for selecting the value: value that rules out the fewest choices for the neightboring variables in the constraining graph.

Propagating information through constraints. The idea is to consider *relevant constraints* as soon as possible.

Forward cheking: once the value of a variable X is selected, recompute the domains of variables that have not been assigned and are related to X.

Constraint propagation: propagating the implication of a constraint on one variable onto other variables; with the intention of identifying inconsitency as early as possible.

Arc consistency: an arc between two variables X and Y (in the constraint graph) is consistent if for every value x of X (in the current domain of X) there exists at least one value y of Y (in the current domain of Y) that is consistent with x.

A graph is arc consistent if all of its arcs are consistent. When some arc is inconsistent, either the problem has no solution or the domains of the variables can be reduced so that it becomes consistent. **AC-3** is an algorithm for maintaining arc consistency.

function AC-3(`csp`) % **return** the `csp`, possibly with reduced domains
  input `csp`, a binary CSP with variables $X_1, \ldots, X_n$
  local variables: *queue*: a queue of arcs, initially all the arcs in `csp`
    while *queue* is not empty
      get the first arc (X_i, X_j) in *queue*, remove it from *queue*
      if REMOVE-INCONSISTENCY-VALUES(X_i, X_j)) **then**
        for each $X_k \in neightbor(X_i)$ **do**
        add (X_k, X_i) to *queue*
  return `csp`

function REMOVE-INCONSISTENCY-VALUES(X_i, X_j) % **return** trye iff we remove a value
  removed = false
  for each x in domain of X_i **do**
    if no value y in domain of X_j allows (x, y) to satisfy the constraint between (X_i, X_j) **then**
      remove x from the domain of X_i
      removed = true
  return removed

It should be noted that if a CSP has a solution then AC-3 should not result in an arc-inconsistent graph. (Why?)

Generalized form of arc consistency includes *k-consistency* and *strong k-consistency*.

Special constraints: *all difference* is a constraint that requires that values of variables are pairwise different; this constraint can be used in forward checking: once the domain of a variable is a singleton, the value has to be removed from the domain of other variables.