CSCE475/875 Multiagent Systems

Handout 2. Distributed Constraint Satisfaction

August 22, 2017

(Based on Russell, S. and P. Norvig (2010) (3rd. Edition) *Artificial Intelligence: A Modern Approach*, Upper Saddle River, NJ: Pearson Education.)

1. Introduction

A Constraint Satisfaction Problem (CSP) is defined by a set of variables, domains for each of the variables, and the constraints on the values that the variables might take on simultaneously.

The role of the CS algorithms is to **assign values to the variables in a way that is consistent with all the constraints**, or **to determine that no such assignment exists**.

Formally speaking, a CSP consists of a finite set of variables $X = \{x_1, \dots, x_n\}$, a domain D_i for each variable x_i, and a set of constraints $\{C_1, \dots, C_m\}$. Each constraint is a predicate on some subset of the variables and the predicate defines a relation that is a subset of the Cartesian product $D_{i,1} \times \dots \times D_{i,n}$.

Example: In the US state four-coloring problem, there are fifty variables, each variable representing a state. Each variable has four possible values in its domain: $\{red, green, blue, yellow\}$. A constraint could be *Nogood*{*Nebraska* = *red*, *Kansas* = *red*} or *Nogood*{*Nebraska* = *blue, Kansas* = *blue*}. Or, *Not-equal*(*Nebraska*, *Kansas*). The CSP then finds a solution such that all variables are assigned each a value and all constraints are satisfied.

In *distributed* CSP, each variable is owned by a different agent. There are 2 types of algorithms:

- **Filtering**: Embody a least-commitment approach and attempt to rule out impossible variable values without losing any possible solutions
- **Heuristic Search**: Embody a more adventurous spirit and select tentative variable values, backtracking when those choices prove unsuccessful

2. Domain-pruning algorithms

Each node—or each agent—communicates with its neighbors—i.e., message passing—in order to eliminate values from their domains.

Filtering Algorithms. Each node communicates its domain to its neighbors, eliminates from its domain the values that are not consistent with the values received from the neighbors, and the process repeats. Specifically, each node x_i with domain D_i repeatedly executes the procedure **Revise**($x_i . x_j$) for each neighbor x_j. For example, first write the constraints as forbidden value combinations, called **nogoods**. *Nogood*$\{x_1, x_2\}$ means that x_1, x_2 cannot take the same value. So, if agent X_1 announces that $x_1 = red$ then, agent X_2 updates its domain based on that and *Nogood*$\{x_1 = red, x_2 = red\}$ and have to conclude that $\sim(x_2 = red)$ and thus removes it from its domain accordingly.

```
Procedure Revise(x_i. x_j)
Forall v_i ∈ D_i do
  If there is no value v_j ∈ D_j such that v_i is consistent with v_j then
      Delete v_i from D_i
```

- Known also "arc consistency", terminates when no further elimination takes place, or when one of the domains becomes EMPTY (in which case the problem has no solution)
- May not terminate in some problems (e.g., 3-state 2-coloring problem)
- If the process terminates with one value in each domain, that set of values constitutes a solution
- In general, filtering is a very weak method, and at best, is used as a preprocessing step for more sophisticated methods

A More Powerful Algorithm. Hyper-resolution is both sound and complete. Each agent repeatedly generates new constraints for its neighbors, notifies them of these new constraints, and prunes its own domain based on new constraints passed to it by its neighbors. NG_i = the set of all Nogoods of which agent *i* is aware and NG_j^* = set of new Nogoods communicated from agent *j* to agent *i*. The number of Nogoods can grow unmanageably large.

```
Procedure ReviseHR(NG_i, NG_j*)
Repeat
   NG_i ← NG_i ∪ NG_j*
   Let NG_i* denote the set of new Nogoods that i can derive from NG_i and its domain using
      hyper resolution
   if NG_i* is nonempty then
      NG_i ← NG_i ∪ NG_i*
      Send NG_i* to all neighbors of i
      If ∅ ∈ NG_i*  then
         stop
Until there is no change in agent i's set of Nogoods NG_i
```

3. The Basic Backtracking Search for CSP

The term **backtracking search** (A*!) is used for a DFS that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

```
Function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  Return RECURSIVE-BACKTRACKING({},csp)
End function
```

```
Function RECURSIVE-BACKTRACKING(assignment,csp) returns a solution, or failure
   If assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],assignment,csp)
   For each value in ORDER-DOMAIN-VALUES(var,assignment,csp) do
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment,csp)
      if result <> failure then return result
      remove {var = value} from assignment
   End for loop
   Return failure
End Function
```

4. But, Here Are the Questions …

We need to find general-purpose methods that address the following questions:

1. Which variable should be assigned next, and in what order should its values be tried? (ORDER-DOMAIN-VALUES and SELECT-UNASSIGNED-VARIABLES)
2. What are the implications of the current variable assignments for the other unassigned variables?
3. When a path fails—that is, a state is reached in which a variable has no legal values—can the search avoid repeating this failure in subsequent paths?

4.1. Variable and Value Ordering

By default, SELECT-UNASSIGNED-VARIABLE simply selects the next unassigned variable in the order given by the list VARIABLES[*csp*]. However, this static variable ordering seldom results in the most efficient search.

The intuitive idea—choosing the variable with the fewest "legal" values—is called the **minimum remaining values** (MRV) heuristic, aka "most constrained variable" or "fail-first" heuristic. If there is a variable X with zero legal values remaining, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables which always will fail when X is finally selected.

The MRV heuristic doesn't help if every variable has the same number of values. In this case the **degree** heuristic comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. MRV is powerful, degree as a tie-breaker.

Once a variable has been selected, choose the value—**least-constraining-value** heuristic. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.

4.2. Propagating Information through Constraints

So far, our search algorithm considers the constraints on a variable only at the time that the variable is chosen by SELECT-UNASSIGNED-VARIABLE. But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

- **Forward Checking.** Whenever a variable *X* is assigned, the forward checking process looks at each unassigned variable *Y* that is connected to *X* by a constraint and deletes from *Y*'s domain any value that is inconsistent with the value chosen for *X*. The MRV is a natural partner for forward checking. Forward checking can detect partial assignments that are inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.
- **Constraint Propagation.** Although forward checking detects many inconsistencies, it does not detect all of them because it does not look far enough. One option is to utilize **arc-consistency**. An arc is a directed arc in the constraint graph. The arc between *X* and *Y* is consistent if, for every value *x* of *X*, there is some value *y* of *Y* that is consistent with *x*.

Function AC-3(*csp*) **returns** the CSP, possibly with reduced domains
  Inputs: *csp*, a binary CSP with variables $\{X_1, X_2,\ldots, X_n\}$
  Local variables: *queue*, a queue of arcs, initially all the arcs in *csp*
  While *queue* is not empty **do**
    $\{X_i, X_j\}$←REMOVE-FIRST(*queue*)
    If REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      For each X_k **in** NEIGHBORS[X_i] - $\{X_j\}$**do**
        Add (X_k, X_i) to *queue*
      End for
  End while
End function

Function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff remove a value
  removed ← *false*
  For each *x* **in** DOMAIN[X_i] **do**
    If no value *y* in DOMAIN[X_j] allows (*x,y*) to satisfy the constraint btw. X_i & X_j
      Then delete *x* from DOMAIN[X_j]; *removed* ← *true*
    End for
  Return *removed*
End function

After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be made arc-consistent (and thus the CSP cannot be solved).

5. Local Search for Constraint Satisfaction Problems

The **min conflicts algorithm** is a search algorithm to solve constraint satisfaction problems (CSP problems).

It assigns random values to all the variables of a CSP. Then it selects randomly a variable, whose value conflicts with any constraint of the CSP. Then it assigns to this variable the value with the minimum conflicts. If there are more than one minimum, it chooses one among them randomly. After that, a new iteration starts again until a solution is found or a pre-selected maximum number of iterations is reached.

Because a CSP can be interpreted as a local search problem when all the variables have assigned a value (complete states), the min conflicts algorithm can be seen as a heuristic that chooses the state with the minimum number of conflicts.

```
Function MIN-CONFLICTS(csp,max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up
  current ← an initial assignment for csp
  For i = 1 to max_steps do
    If current is a solution of csp then return current
    var ← a randomly chosen, conflicted variable from VARIABLES[csp]
    value ← the value v for var that minimizes CONFLICTS(var,v,current,csp)
    set var = value in current
  Return failure
End Function
```