Artificial Intelligence

CS4365 --- Fall 2022

Constraint Satisfication Problem

Instructor: Yunhui Guo

Standard Search Problem

- Standard Search Problems:
  - State is a “black box”: arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything

Constraint Satisfication Problems (CSP)

- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variable X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

- Why CSPs?
  - Allows useful general-purpose algorithms with more power than standard search algorithms

Example: Map-Coloring Problem

- Color each region either red, green or blue in such a way that no neighboring regions have the same color

  - Variables: different regions
  - Domains: colors

Constraint Satisfication Problems (CSP)

- A powerful representation for (discrete) search problems.

- A Constraint Satisfication Problem (CSP) is defined by:
  - X is a set of n variables X_1, X_2, ..., X_n, each defined by a finite domain D_1, D_2, ..., D_n of possble values
  - C is a set of constraints C_1, C_2, ..., C_m. Each C_i involves some subset of the variables; specifices the allowable combinations of values for that subset.

A solution is an assigment of values to the variables that satisfies all constraints.

Cyptarithmetic as a CSP

- Variables:

  F O U R T W C_1 C_2 C_3

- Domains:

  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

- Constraints:

  $O + O = R + 10 * C_1$

  $C_1 + W + W = U + 10 * C_2$

  $C_2 + T + T = O + 10 * C_3$

  $C_3 = F$

  alldiff(F, O, U, R, T, W)

```
    TWO
+   TWO
-------
   FOUR
```

Cyptarithmetic Constraint Graph

- Variables:

  F O U R T W C_1 C_2 C_3

- Domains:

  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

- Constraints:

  $O + O = R + 10 * C_1$

  $C_1 + W + W = U + 10 * C_2$

  $C_2 + T + T = O + 10 * C_3$

  $C_3 = F$

  alldiff(F, O, U, R, T, W)

Map-Coloring Problem

- Variables:

  WA, NT, Q, NSW, V, SA, T

- Domains:

  {red, green, blue}

- Constraints:

  Adjacent regions must have different colors

  Implicit: WA ≠ NT

  Explicit: (WA, NT) ∈ {(red, green), {red, blue}, ...}

Constraint Graph

- Variables:

  WA, NT, Q, NSW, V, SA, T

- Domains:

  {red, green, blue}

- Constraints:

  Adjacent regions must have diffe-

  rent colors

  Implicit: WA ≠ NT

  Explicit: (WA, NT) ∈ {(red, green), {red, blue}, ...}

Map-Coloring Problem

- Variables:

  WA, NT, Q, NSW, V, SA, T

- Domains:

  {red, green, blue}

- Constraints:

  Adjacent regions must have different colors

  Implicit: WA ≠ NT

  Explicit: (WA, NT) ∈ {(red, green), {red, blue}, ...}

Varieties of CSPs

- Discrete variables
  - Finite domains: each variable can be only assigned to finite number of values.
  - Infinite domains: each variable can be assigned to infinite number of values, e.g., set of integers
- Continuous variables
  - Linear programming

$$
\begin{aligned}
\text{Minimize} \quad & z = 0.6x_1 + 0.35x_2 \\
\text{subject to:} \quad & \\
& 5x_1 + 7x_2 \geq 8 \\
& 4x_1 + 2x_2 \geq 15 \\
& 2x_1 + x_2 \geq 3 \\
& x_1 \geq 0, x_2 \geq 0.
\end{aligned}
$$

Varieties of Constraints

- Unary constraints involve a single variable (equivalent to reducing domains), e.g.:
  - SA ≠ Green
- Binary constraints involve pairs of variables, e.g.:
  - SA ≠ WA
- Higher-order constraints involve 3 or more variables
- Preferences (soft constraints):
  - E.g., red is better than green

Real-World CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling

Constraint Satisfication Problems (CSP)

- For a given CSP the problem is one of the following:
  - Find all solutions
  - Find one solution
  - The optimal solution given an objective function
  - Determine if a solution exists

How to View a CSP as a Search Problem?

- Initial State -- state in which all the variables are unassigned.
- Successor function -- assign a value to a variable from a set
- Goal test -- check if all the variables are assigned and all the constraints are satsified. We call the assignment is complete.
- Path cost -- assumes constant cost for each step

Backtracking Search

- Depth-first search for CPSs with single-variable assigments is called backtracking search

  - Backtracks when a variable has no legal values left to assign

Backtracking search is the basic uninformed algorithm for CSPs.

The Backtracking Search Algorithm

```
Backtrack(assignment, csp):
    If assignment is complete then return assignment
    var ← select one unassigned variable
    for each value in the domain of var do
        if value is consistent with assignment then
            add {var = value} to assignment
            result ← Backtrack(assignment, csp)
            if result ≠ failure then
                return result
            remove {var = value} from assignment
    return failure
```

Backtracking Example

The Backtracking Search Algorithm

```
Backtrack(assignment, csp):
    If assignment is complete then return assignment
    var ← select one unassigned variable  ← which one to select?
    for each value in the domain of var do ← which one to use?
        if value is consistent with assignment then
            add {var = value} to assignment     ← detect failure early
            result ← Backtrack(assignment, csp)
            if result ≠ failure then
                return result
            remove {var = value} from assignment
    return failure
```

Improving Backtracking Efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
   - Choose the most constrained variable, and break ties by choosing the most constraining variable

2. In what order should its values be tried?
   - Choose the least constraining value

3. Can we detect inevitable failure early?
   - Forward checking, constraint propagation

Improving Backtracking Efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
   - Choose the most constrained variable, and break ties by choosing the most constraining variable

2. In what order should its values be tried?
   - Choose the least constraining value

3. Can we detect inevitable failure early?
   - Forward checking, constraint propagation

Variable Selection Heuristic 1: Most Constrained Variable

Variable Selection Heuristic 1: Most Constrained Variable

Most constrained variable:

Choose the variable with the fewest legal values

a.k.a. mininum remaining values (MRV) heuristic

- Why min rather than max?
- “Fail-fast” ordering
- 1000$\times$ faster or more

Variable Selection Heuristic 2: Most Constraining Variable

- Tie-breaker among most constrained variables

Variable Selection Heuristic 2: Most Constraining Variable

- Tie-breaker among most constrained variables

Most constraining variable:

choose the variable with the most constraints on remaining variables

Improving Backtracking Efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
   - Choose the most constrained variable, and break ties by choosing the most constraining variable

2. In what order should its values be tried?
   - Choose the least constraining value

3. Can we detect inevitable failure early?
   - Forward checking, constraint propagation

Value Select Heuristic: Least Constraining Value

Value Select Heuristic: Least Constraining Value

- Given a variable, choose the least constraining value:

  the one that rules out the fewest values in the remaining variables

- Why least rather than most?

Variable and Value Ordering Heuristics

- The heuristics should be applied whenever a variable/value needs be chosen during the search process

- They should not be applied once and for all at the begining of the search process

Improving Backtracking Efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
   - Choose the most constrained variable, and break ties by choosing the most constraining variable

2. In what order should its values be tried?
   - Choose the least constraining value

3. Can we detect inevitable failure early?
   - Forward checking, constraint propagation

Consistency-Enforcing Procedure 1: Forward Checking

- Idea:

  Keep track of remaining legal values for unsigned variables

  Terminate search when any variable has no legal values

Consistency-Enforcing Procedure 2: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constrains locally.

Consistency-Enforcing Procedure 2: Constraint Propagation

- Using the constraints to reduce the number of legal values for a variable
- Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts
- Sometimes this preprocessing can solve the whole problem, so no search is required at all!
- Local consistency

Arc Consistency

- An arc X → Y is consistent iff for every x in the tail there is some y in the head which could be assigned without violating a constraint

- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency

- An arc X → Y is consistent iff for every x in the tail there is some y in the head which could be assigned without violating a constraint

- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Arc Consistency

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi];  removed ← true
   return removed
```

Runtime: $O(n^2d^3)$ n: # of variables d: domain size

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

K-Consistency

- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the kth node.

Consistency-Enforcing Procedures

- Forward checking
- Constraint propagation
- Variable/value ordering heuristics

- They can used in combination to improve search process.

General Purpose CSP Algorithm

1. If all values have been successfully assigned then stop, else go on to 2.
2. Apply the consistency enforcing procedure (e.g. forward checking if feeling computationally mean, or constraint propagation if extravagant.)
3. If a deadend is detected then backtrack (e.g., DFS-type backtrack.). Else go on to step 4.
4. Select the next variable to be assigned (using your variable ordering heuristic)
5. Select a promising value (using your value ordering heuristic)
6. Create a new search state. Cache the info you need for backtracking. And go back to 1.

Local Search for CSPs

- Hill-climbing, simulated annealing typically work with “complete” states, i.e., all variables assigned
- To apply to CSPs:

  allow states with unsatisfied constraints operators reassign variable values
- Variable selection: randomly select any conflicted variable

- Value selection by min-conflicts heuristic:

  Choose value that violates the fewest constraints

  i.e., hill-climb with h(n) = total number of violated constraints

Example: 4-Queens

- States: 4 queens in 4 columns (256 states)
- Actions: move queen in column
- Goal test: no attacks
- Evaluation: h(n) = number of attacks

- Give random inital state, can solve n-queens in almost constant time for arbirary n with high probability (e.g., n = 10,000,000)