14.

Constraint Satisfaction Problems

CS227
Spring 2011
Outline

• Example of a Constraint Satisfaction Problem (CSP)
• Representing a CSP
• Solving a CSP
  – Backtracking search
  – Problem structure and decomposition
• Constraint logic programming
• Summary
Example: Map-Coloring

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

Domains $D_i = \{\text{red, green, blue}\}$

Constraints: adjacent regions must have different colors

- e.g., $WA \neq NT$ (if the language allows this), or
- $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), \ldots\}$
Example: Map-Coloring contd.

Solutions are assignments satisfying all constraints, e.g.,
\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}
Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints
Varieties of CSPs

Discrete variables
- finite domains; size $d \Rightarrow O(d^m)$ complete assignments
  - e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
- infinite domains (integers, strings, etc.)
  - e.g., job scheduling, variables are start/end days for each job
  - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
  - linear constraints solvable, nonlinear undecidable

Continuous variables
- e.g., start/end times for Hubble Telescope observations
- linear constraints solvable in poly time by LP methods
Varieties of constraints

Unary constraints involve a single variable,
   e.g., $SA \neq green$

Binary constraints involve pairs of variables,
   e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,
   e.g., cryptarithmetic column constraints

Preferences (soft constraints), e.g., red is better than green
   often representable by a cost for each variable assignment
   → constrained optimization problems
Example: Cryptarithmetic

\[
\begin{array}{c}
\text{TWO} \\
+ \text{TWO} \\
\hline
\text{FOUR}
\end{array}
\]

Variables: \( F, T, U, W, R, O, X_1, X_2, X_3 \)
Domains: \( \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \)
Constraints
- \text{alldiff}(F, T, U, W, R, O)
- \( O + O = R + 10 \cdot X_1 \), etc.
Real-world CSPs

Assignment problems
    e.g., who teaches what class

Timetabling problems
    e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

Floorplanning

Notice that many real-world problems involve real-valued variables
Outline

• Example of a Constraint Satisfaction Problem (CSP)
• Representing a CSP
• Solving a CSP
  – Backtracking search
  – Problem structure and decomposition
• Constraint logic programming
• Summary
Standard search formulation (incremental)

Let’s start with the straightforward, dumb approach, then fix it

States are defined by the values assigned so far

◊ Initial state: the empty assignment, \{\}

◊ Successor function: assign a value to an unassigned variable
  that does not conflict with current assignment.
  \(\Rightarrow\) fail if no legal assignments (not fixable!)

◊ Goal test: the current assignment is complete

1) This is the same for all CSPs! 😃
2) Every solution appears at depth \(n\) with \(n\) variables

3) Path is irrelevant, so can also use complete-state formulation
4) \(b = (n - \ell)d\) at depth \(\ell\), hence \(n!d^n\) leaves!!!! 😜
Backtracking search

Variable assignments are commutative, i.e.,

\[ [WA = \text{red} \text{ then } NT = \text{green}] \text{ same as } [NT = \text{green} \text{ then } WA = \text{red}] \]

Only need to consider assignments to a single variable at each node

\[ \Rightarrow \quad b = d \quad \text{and there are } d^n \text{ leaves} \]

Depth-first search for CSPs with single-variable assignments
is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs
Backtracking search

function **Backtracking-Search**(*csp*) returns *solution/failure*
    return **Recursive-Backtracking**({ }, *csp*)

function **Recursive-Backtracking**(*assignment*, *csp*) returns *solution/failure*
    if *assignment* is complete then return *assignment*
    \( var \leftarrow \text{Select-Unassigned-Variable} (\text{Variables}[csp], \text{assignment}, csp) \)
    for each *value* in \( \text{Order-Domain-Values}(\text{var}, \text{assignment}, csp) \) do
        if *value* is consistent with *assignment* given \( \text{Constraints}[csp] \) then
            add \( \{\text{var} = \text{value}\} \) to *assignment*
            \( \text{result} \leftarrow \text{Recursive-Backtracking}(\text{assignment}, csp) \)
            if *result* ≠ *failure* then return *result*
            remove \( \{\text{var} = \text{value}\} \) from *assignment*
        
    return *failure*
Backtracking example
Backtracking example
Backtracking example
Backtracking example
Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?
Minimum remaining values (MRV): choose the variable with the fewest legal values
Degree heuristic

Tie-breaker among MRV variables

Degree heuristic:
choose the variable with the most constraints on remaining variables

[Diagram of Australia with states colored and labeled with arrows]
Least constraining value

Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables
Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

WA  NT  Q  NSW  V  SA  T

[Diagram of Australia with states WA, NT, Q, NSW, V, SA, T represented by different colors]
Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values
Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

<table>
<thead>
<tr>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Forward checking**

**Idea:** Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values
Constraint propagation

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

\[ \text{WA} \quad \text{NT} \quad \text{Q} \quad \text{NSW} \quad \text{V} \quad \text{SA} \quad \text{T} \]

\[ \begin{array}{cccccccc}
\text{WA} & \text{NT} & \text{Q} & \text{NSW} & \text{V} & \text{SA} & \text{T} \\
\text{\textcolor{red}{80\%}} & \text{\textcolor{blue}{20\%}} & \text{\textcolor{blue}{0\%}} & \text{\textcolor{blue}{0\%}} & \text{\textcolor{blue}{0\%}} & \text{\textcolor{red}{80\%}} & \text{\textcolor{blue}{0\%}} \\
\text{\textcolor{red}{80\%}} & \text{\textcolor{red}{80\%}} & \text{\textcolor{red}{80\%}} & \text{\textcolor{blue}{0\%}} & \text{\textcolor{blue}{0\%}} & \text{\textcolor{blue}{0\%}} & \text{\textcolor{red}{80\%}} \\
\text{\textcolor{red}{80\%}} & \text{\textcolor{red}{80\%}} & \text{\textcolor{red}{80\%}} & \text{\textcolor{blue}{0\%}} & \text{\textcolor{blue}{0\%}} & \text{\textcolor{blue}{0\%}} & \text{\textcolor{red}{80\%}} \\
\end{array} \]

\textit{NT} and \textit{SA} cannot both be blue!

Constraint propagation repeatedly enforces constraints locally
Arc consistency

Simplest form of propagation makes each arc consistent

\[ X \rightarrow Y \] is consistent iff

for every value \( x \) of \( X \) there is some allowed \( y \)
Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for every value $x$ of $X$ there is some allowed $y$
Arc consistency

Simplest form of propagation makes each arc consistent

\[ X \rightarrow Y \] is consistent iff
for every value \( x \) of \( X \) there is some allowed \( y \)

If \( X \) loses a value, neighbors of \( X \) need to be rechecked
Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff
for every value $x$ of $X$ there is some allowed $y$

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 algorithm

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
  \((X_i, X_j) \leftarrow Remove-First(queue)\)
  if Remove-Inconsistent-Values\((X_i, X_j)\) then
    for each \(X_k\) in Neighbors[\(X_i\)] do
      add \((X_k, X_i)\) to \(queue\)

function Remove-Inconsistent-Values\((X_i, X_j)\) returns true iff succeeds
removed \(\leftarrow false\)
for each \(x\) in Domain[\(X_i\)] do
  if no value \(y\) in Domain[\(X_j\)] allows \((x, y)\) to satisfy the constraint \(X_i \leftrightarrow X_j\) then delete \(x\) from Domain[\(X_i\)]; removed \(\leftarrow true\)
return removed

\(O(n^2d^3)\), can be reduced to \(O(n^2d^2)\) (but detecting all is NP-hard)
Outline

- Example of a Constraint Satisfaction Problem (CSP)
- Representing a CSP
- Solving a CSP
  - Backtracking search
  - Problem structure and decomposition
- Constraint logic programming
- Summary
Problem structure

Tasmania and mainland are independent subproblems
Identifiable as connected components of constraint graph
Problem structure contd.

Suppose each subproblem has $c$ variables out of $n$ total

Worst-case solution cost is $n/c \cdot d^c$, linear in $n$

E.g., $n=80$, $d=2$, $c=20$

$2^{80} = 4$ billion years at 10 million nodes/sec

$4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n d^2)$ time

Compare to general CSPs, where worst-case time is $O(d^n)$
Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering

   ![Diagram of tree structure]

2. For $j$ from $n$ down to 2, apply $\text{RemoveInconsistent}(\text{Parent}(X_j), X_j)$

3. For $j$ from 1 to $n$, assign $X_j$ consistently with $\text{Parent}(X_j)$
Nearly tree-structured CSPs

**Conditioning:** instantiate a variable, prune its neighbors’ domains

**Cutset conditioning:** instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size $c \Rightarrow$ runtime $O(d^c \cdot (n - c)d^2)$, very fast for small $c$
Outline

- Example of a Constraint Satisfaction Problem (CSP)
- Representing a CSP
- Solving a CSP
  - Backtracking search
  - Problem structure and decomposition
- Constraint logic programming
- Summary
A constraint logic program is a logic program that contains constraints in the body of clauses.

\[
A(X,Y) :- \\
X+Y>0, \\
B(X), \\
C(Y)
\]

Constraints are stored in a constraint store and evaluated using a CSP technique.
Example Application

- Meeting scheduling video
Meeting Scheduling Constraints

- The meeting room needs to be able to hold at least n people
- The meeting room needs to have a projector (or sound equipment or similar)
- The appointment may be recurring and need to be at the same time/location each week
- I want at least 1 hour between appointments
- If we are teleconferencing with our European office, meetings need to be scheduled at an appropriate time
- Bob will only attend appointments if Gary is not present
- I will only attend a maximum of 3 appointments in a given day
- I need to meet before a deadline
- I prefer meetings near my residence/office
Summary

CSPs are a special kind of problem:
states defined by values of a fixed set of variables
goal test defined by constraints on variable values

Backtracking = depth-first search with one variable assigned per node

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice
Reading

• Chapter on Constraint Satisfaction Problems in Russell and Norvig
  – Chapter 5 in 2nd edition
  – Chapter 6 in 3rd edition