PROBLEM SOLVING AND SEARCH IN ARTIFICIAL INTELLIGENCE

Lecture 8 Constraint Satisfaction Problems

Sarah Gaggl

Dresden, 10th December 2019
Agenda

1. Introduction
2. Uninformed Search versus Informed Search (Best First Search, A*-Search, Heuristics)
3. Local Search, Stochastic Hill Climbing, Simulated Annealing
4. Tabu Search
5. Answer-set Programming (ASP)
6. Constraint Satisfaction Problems (CSP)
7. Evolutionary Algorithms/ Genetic Algorithms
8. Structural Decomposition Techniques (Tree/Hypertree Decompositions)
Constraint satisfaction problems (CSPs)

- Standard search problem:
  - state is a “black box”—any old data structure that supports goal test, eval, successor
- CSP:
  - state is defined by variables $X_i$ with values from domain $D_i$
  - goal test is a set of constraints specifying allowable combinations of values for subsets of variables
- Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms
- Main idea: eliminate large portions of search space all at once by identifying variable/value combinations that violate constraints
Defining CSPs

Constraint Satisfaction Problem (CSP)
A CSP is defined as a tuple $C = \langle X, D, C \rangle$, with

- $X$ a set of variables, $\{X_1, \ldots, X_n\}$.
- $D$ a set of domains, $\{D_1, \ldots, D_n\}$, for each variable.
- $C$ a set of constraints that specify allowable combinations of values.

- Each domain $D_i$ consists of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable $X_i$.
- Each constraint $C_i$ consists of a pair $\langle \text{scope}, \text{rel} \rangle$, where $\text{scope}$ is a tuple of variables in the constraint, and $\text{rel}$ defines the possible values.
- A relation can be
  - an explicit list of all tuples of values satisfying the constraint, or
  - an abstract relation.
If $X_1$ and $X_2$ both have domain $\{A, B\}$, the constraint saying they have different values can be written as:

- $\langle (X_1, X_2), [(A, B), (B, A)] \rangle$, or
- $\langle (X_1, X_2), X_1 \neq X_2 \rangle$.

To solve a CSP, we define a state space and the notion of a solution.

- Each state in a CSP is defined by an assignment of values to some (or all variables), $\{X_i = v_i, X_j = v_j, \ldots\}$.
- An assignment is consistent if it does not violate any constraints.
- A complete assignment has a value assigned to each variable.
- A solution is a consistent, complete assignment.
- A partial assignment is one that assigns values to only some of the variables.
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 ctd.

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

General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent sub-problem!
Varieties of CSPs

Discrete variables
- finite domains; size $d \Rightarrow O(d^n)$ 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 $\rightarrow$ constrained optimization problems
Example: Cryptarithmetic

\[
\begin{array}{c}
T W O \\
+ T W O \\
\hline
F O U R
\end{array}
\]

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.
Constraint Propagation: Inference in CSPs

In regular state-space search, an algorithm can only perform search. In CSPs there is a choice

- an algorithm can search (choose a new variable assignment form several possibilities), or
- do a specific type of inference called constraint propagation:
  - using the constraints to reduce the number of legal values for a variable
  - this can reduce the legal values for another variable,
  - and so on.

Constraint propagation may be
- intertwined with search, or
- done as a pre-processing step (could solve the whole problem; no search is required).
Constraint Propagation

The key idea is **local consistency**.

- Treat each variable as a node in a graph.
- Each binary constraint represents an arc.
- Enforcing local consistency in each part of the graph eliminates inconsistent values throughout the graph.

Different types of local consistency:

- Node consistency
- Arc consistency
- Path consistency
Node Consistency

Node consistency

A variable $X$ is **node-consistent** if all values in the domain of $X$ satisfy the **unary** constraints of $X$. A CSP is node-consistent if every variable is node consistent.

Example

South Australia dislikes green.

- Variable $SA$ starts with $\{\text{red, green, blue}\}$,
- make it node consistent by eliminating $\text{green}$,
- reduced domain of $SA$ is $\{\text{red, blue}\}$.
Arc Consistency

A variable is **arc-consistent** if every value in its domain satisfies the variable's binary constraints. \( X_i \) is arc-consistent wrt. \( X_j \) if for every value in \( D_i \) there is some value in \( D_j \) that satisfies the binary constraint on the arc \((X_i, X_j)\). A CSP is arc-consistent if every variable is arc-consistent with every other variable.

**Example**

Consider the constraint \( Y = X^2 \), where the domain of both \( X \) and \( Y \) is the set of digits. We can write the constraint explicitly as

\[
\langle (X, Y), \{(0, 0), (1, 1), (2, 4), (3, 9)\} \rangle.
\]

To make \( X \) arc-consistent wrt. \( Y \), we reduce \( X \)'s domain to \( \{0, 1, 2, 3\} \). We also reduce \( Y \)'s domain to \( \{0, 1, 4, 9\} \) and the CSP is arc-consistent.
Path Consistency

- Arc consistency can reduce domains of variables and sometimes find a solution (or failure).
- But for other networks, arc consistency fails to make enough inferences.
- Example of map coloring of Australia with two colors.

Path consistency

A two-variable set \( \{X_i, X_j\} \) is **path-consistent** wrt. a third variable \( X_m \) if, for every assignment \( \{X_i = a, X_j = b\} \) consistent with constraints on \( \{X_i, X_j\} \), there is an assignment to \( X_m \) that satisfies the constraints on \( \{X_i, X_m\} \) and \( \{X_m, X_j\} \).
Consider two-coloring of Australia. We make \{WA, SA\} path consistent wrt. NT.

- Start by enumerating the consistent assignments to the set.
  - \{WA = red, SA = blue\}
  - \{WA = blue, SA = red\}
- With both assignments NT can be neither red nor blue.
- Eliminate both assignments.
- Thus, there is no solution to the problem.
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.

\[ \implies \text{fail if no legal assignments (not fixable!)} \]

**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

\[ \implies \text{use depth-first search} \]

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.,
  \[
  \text{[WA = red then NT = green]} \quad \text{same as} \quad \text{[NT = green then WA = red]}
  \]
- Only need to consider assignments to a single variable at each node
  \[\implies b = d \quad \text{and there are } d^n \text{ leaves}\]
- Depth-first search for CSPs with single-variable assignments is called **backtracking search**
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve \(n\)-queens for \(n \approx 25\)
Backtracking search

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

function Recursive-Backtracking(assignment, csp) returns soln/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
        if value is consistent with assignment given Constraints[csp] then
            add {var = value} to assignment
            result ← Recursive-Backtracking(assignment, csp)
            if result ≠ failure then return result
            remove {var = 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
Least constraining value

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

Combining these heuristics makes 1000 queens feasible
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
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
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 constraints locally
Arc consistency

Simplest form of propagation makes each arc consistent

\[ X \rightarrow Y \text{ is consistent iff} \]

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

\[ \text{If } X \to Y \text{ is consistent iff} \]

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 pre-processor 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
Problem structure

- Tasmania and mainland are independent sub-problems
- Identifiable as connected components of constraint graph
Problem structure ctd.

- Suppose each subproblem has $c$ variables out of $n$ total.
- Worst-case solution cost is $\frac{n}{c} \cdot d^c$, linear in $n$.
- E.g., $n = 80$, $d = 2$, $c = 20$
  - $2^{80} = 4 \text{ billion years at 10 million nodes/sec}$
  - $4 \cdot 2^{20} = 0.4 \text{ seconds at 10 million nodes/sec}$
Tree-structured CSPs

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 $O(d^n)$
- This property also applies to logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.
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

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** $c \implies$ runtime $O(d^c \cdot (n - c)d^2)$, very fast for small $c$
Iterative algorithms 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., hillclimb with $h(n) = \text{total number of violated constraints}$
Example: 4-Queens

**States:** 4 queens in 4 columns \( (4^4 = 256 \text{ states}) \)

**Operators:** move queen in column

**Goal test:** no attacks

**Evaluation:** \( h(n) = \text{number of attacks} \)

\[ h = 5 \quad h = 2 \quad h = 0 \]
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
References