Constraint Satisfaction Problems

Professor Chris Callison-Burch
What is Search For?

- **Assumptions about the world:** a single agent, deterministic actions, fully observed state, discrete state space

- **Planning:** sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance

- **Identification:** assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are specialized for identification problems
Big idea

- Represent the *constraints* that solutions must satisfy in a uniform *declarative* language.
- Find solutions by *GENERAL PURPOSE* search algorithms with no changes from problem to problem:
  - No hand-built transition functions
  - No hand-built heuristics

- Just specify the problem in a formal declarative language, and a general-purpose algorithm does everything else!
Constraint Satisfaction Problems

A CSP consists of:

- **Finite set of variables** $X_1, X_2, ..., X_n$
- **Nonempty domain** of possible values for each variable $D_1, D_2, ..., D_n$ where $D_i = \{v_1, ..., v_{k_i}\}$
- **Finite set of constraints** $C_1, C_2, ..., C_m$
  - Each constraint $C_i$ limits the values that variables can take, e.g., $X_1 \neq X_2$
  - A state is defined as an **assignment** of values to some or all variables.

  - A **consistent** assignment does not violate the constraints.
  - Example problem: Sudoku
Constraints in Sudoku

All different
### Constraints in Sudoku

All different

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
Constraints in Sudoku

All different
Constraint satisfaction problems

- An assignment is complete when every variable is assigned a value.
- A solution to a CSP is a complete, consistent assignment.
- Solutions to CSPs can be found by a completely general purpose algorithm, given only the formal specification of the CSP.

- Beyond our scope: CSPs that require a solution that maximizes an objective function.
Applications

- Map coloring
- Scheduling problems
  - Job shop scheduling
  - Scheduling the Webb Space Telescope
- Floor planning for VLSI
- Sudoku
- ...

CMOS VLSI Design

Neil H. E. Weste, David M. Harris

CMOS VLSI Design: A Circuits and Systems Perspective
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., \( WA \neq NT \)
  - So \((WA,NT)\) must be in \{\(\text{(red,green)},(\text{red,blue}),(\text{green,red}),\ldots\)\}
Example: Map-coloring

Solutions: complete and consistent assignments

- e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green
Benefits of CSP

- Clean specification of many problems, generic goal, successor function & heuristics
  - Just represent problem as a CSP & solve with general package

- CSP “knows” which variables violate a constraint
  - And hence where to focus the search

- CSPs: Automatically prune off all branches that violate constraints
  - (State space search could do this only by hand-building constraints into the successor function)
CSP Representations

- **Constraint graph:**
  - *nodes* are variables
  - *arcs* are (binary) constraints

- **Standard representation pattern:**
  - variables with values

- **Constraint graph** simplifies search.
  - e.g. Tasmania is an independent subproblem.

- **This problem: A binary CSP:**
  - each constraint relates two variables
Varieties of CSPs

- **Discrete variables**
  - finite domains:
    - $n$ variables, domain size $d \rightarrow O(d^n)$ complete assignments
    - e.g., Boolean CSPs, includes 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., $StartJob_1 + 5 \leq StartJob_3$

- **Continuous variables**
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming
Varieties of constraints

- **Unary** constraints involve a single variable,
  - e.g., SA ≠ green

- **Binary** constraints involve pairs of variables,
  - e.g., SA ≠ WA

- **Higher-order** constraints involve 3 or more variables
  - e.g., crypt-arithmetic column constraints

- **Preference** (soft constraints) e.g. *red is better than green* can be represented by a cost for each variable assignment
  - Constrained optimization problems.
Idea 1: CSP as a search problem

- A CSP can easily be expressed as a search problem
  - \textit{Initial State}: the empty assignment \{\}.  
  - \textit{Successor function}: Assign value to any unassigned variable \textit{provided that there is not a constraint conflict}.  
  - \textit{Goal test}: the current assignment is complete.  
  - \textit{Path cost}: a constant cost for every step.

- Solution is always found at depth $n$, for $n$ variables
  - Hence Depth First Search can be used
Search and branching factor

- $n$ variables of domain size $d$
- Branching factor at the root is $n*d$
- Branching factor at next level is $(n-1)*d$
- Tree has $n!*d^n$ leaves
Search and branching factor

- The variable assignments are **commutative**
  - Eg [step 1: WA = red; step 2: NT = green]
  - equivalent to [step 1: NT = green; step 2: WA = red]
  - Therefore, a tree search, not a graph search

- Only need to consider assignments to a single variable at each node
  - \( b = d \) and there are \( d^n \) leaves (\( n \) variables, \( d \) domain size)
Search and *Backtracking*

- Depth-first search for CSPs with single-variable assignments is called *backtracking* search.
- The term backtracking search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
- Backtracking search is the basic *uninformed* algorithm for CSPs.
Backtracking example
Backtracking example
Idea 2: Improving backtracking efficiency

- **General-purpose** methods & **general-purpose** heuristics can give huge gains in speed, *on average*

- **Heuristics:**
  - Q: Which variable should be assigned next?
    1. Most constrained variable
    2. (if ties:) Most constraining variable
  - Q: In what order should that variable’s values be tried?
    3. Least constraining value
  - Q: Can we detect inevitable failure early?
    4. Forward checking
Heuristic 1: Most constrained variable

- Choose a variable with the **fewest legal values**

- a.k.a. **minimum remaining values (MRV)** heuristic
Heuristic 2: Most constraining variable

- Tie-breaker among most constrained variables
- Choose the variable with the most constraints on remaining variables

These two heuristics together lead to immediate solution of our example problem.
Heuristic 3: Least constraining value

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

Note: demonstrated here independent of the other heuristics
Heuristic 4: Forward checking

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

(A first step towards Arc Consistency & AC-3)
Forward checking

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

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

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

Terminate! No possible value for SA
Example: 4-Queens Problem

Assign value to unassigned variable
Example: 4-Queens Problem

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
Example: 4-Queens Problem

Forward check!

Backtrack!!!
Example: 4-Queens Problem

Picking up a little later after two steps of backtracking....

Assign value to unassigned variable
Example: 4-Queens Problem

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
Example: 4-Queens Problem

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
Example: 4-Queens Problem

Forward check!
Example: 4-Queens Problem

Assign value to unassigned variable
Towards Constraint propagation

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

- **Constraint propagation** goes beyond forward checking & repeatedly enforces constraints locally

- NT and SA cannot both be blue!
CIS 4210/5210: ARTIFICIAL INTELLIGENCE

Constraint Satisfaction Problems

Professor Chris Callison-Burch

Reading: AIMA Sections 6.1-6.5.

Optional Course Content Recitation Tonight from 8-9pm on Games and Adversarial Search

The OHQs have gotten long! The TAs and I are brainstorming ways of improving this.
Review: CSPs and Search

- **Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space**

- **Planning: sequences of actions**
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance

- **Identification: assignments to variables**
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are specialized for identification problems
Review: Search and *Backtracking*

- Depth-first search for CSPs with single-variable assignments is called *backtracking* search.
- The term *backtracking* search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
- Backtracking search is the basic *uninformed* algorithm for CSPs.
Review: Improving backtracking efficiency

- General-purpose methods & general-purpose heuristics can give huge gains in speed, on average

- Heuristics:
  - Q: Which variable should be assigned next?
    1. Most constrained variable
    2. (if ties:) Most constraining variable
  
  - Q: In what order should that variable’s values be tried?
    3. Least constraining value
  
  - Q: Can we detect inevitable failure early?
    4. Forward checking
Review: Heuristic 1: Most constrained variable

- Choose a variable with the fewest legal values
  - a.k.a. minimum remaining values (MRV) heuristic
Review: Heuristic 2: Most constraining variable

- Tie-breaker among most constrained variables
- Choose the variable with the most constraints on remaining variables

These two heuristics together lead to immediate solution of our example problem.
Review: Heuristic 3: Least constraining value

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

Note: demonstrated here independent of the other heuristics
Review: Heuristic 4: Forward checking

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

![New data structure](image-url)
Review: Forward checking

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

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

- **Idea:**
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any unassigned variable has no remaining legal values
Towards 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** goes beyond forward checking & repeatedly enforces constraints locally
Arc Consistency, Constraint Propagation & AC-3

Professor Chris Callison-Burch
Idea 3 (big idea): **Inference** in CSPs

- CSP solvers combine search *and inference*
  - **Search**
  - **Constraint propagation (inference)**
    - Eliminates possible values for a variable if the value would violate **local consistency**
    - *Can do inference first, or intertwine it with search*
      - You’ll investigate this in the Sudoku homework

**Search** = assign a value to a variable

**Inference** = use constraints to reduce number of legal values for a variable
Local Consistency

- **Node consistency**: satisfies unary constraints
  - This is trivial!

- **Arc consistency**: satisfies binary constraints
  - $X_i$ is arc-consistent with respect to $X_j$
  - If for every value $v$ in $D_i$
  - There is some value $w$ in $D_j$ that satisfies the binary constraint on the arc between $X_i$ and $X_j$
CSP Representations

- **Constraint graph:**
  - *nodes* are variables
  - *edges are constraints*
Edges to Arcs: From Constraint Graph to Directed Graph

- Given a pair of nodes $X_i$ and $X_j$ connected by a constraint edge, we represent this not by a single undirected edge, but a pair of directed arcs.
  - For a connected pair of nodes $X_i$ and $X_j$, there are two arcs that connect them: $(i,j)$ and $(j,i)$.
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, recheck neighbors of $X$
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,
- Detects failure earlier than forward checking.
- Can be run as a preprocessor or after each assignment.
Arc consistency

An arc \((i,j)\) is arc consistent if and only if every value \(v\) on \(X_i\) is consistent with some label on \(Y_j\).

To make an arc \((i,j)\) arc consistent,

for each value \(v\) on \(X_i\),

if there is no label on \(Y_j\) consistent with \(v\)

then remove \(v\) from \(X_i\)

Given \(d\) values, checking arc \((i,j)\) takes \(O(d^2)\) time worst case.

\(d\) is the size of the domain - the number of values
Example: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects
- An early example of an AI computation posed as a CSP

- Approach:
  - Each intersection is a variable
  - Adjacent intersections impose constraints on each other
  - Solutions are physically realizable 3D interpretations

Slide credit: Dan Klein and Pieter Abbeel
http://ai.berkeley.edu
Replacing Search: Constraint Propagation Invented...

Dave Waltz’s insight:

- By *iterating* over the graph, the arc-consistency *constraints* can be *propagated* along arcs of the graph.

- *Search*: Use constraints to *add* labels to find *one solution*

- *Constraint Propagation*: Use constraints to *eliminate* labels to simultaneously find *all solutions*
The Waltz/Mackworth Constraint Propagation Algorithm

1. Assign *every* node in the constraint graph a set of *all* possible values
2. Repeat until there is no change in the set of values associated with any node:
   3. For each node $i$:
      4. For each neighboring node $j$ in the picture:
         5. Remove any value from $i$ which is not arc consistent with $j$. 
Inefficiencies: Towards AC-3

1. At each iteration, we only need to examine those $X_i$ where at least one neighbor of $X_i$ has lost a value in the previous iteration.

2. If $X_i$ loses a value only because of arc inconsistencies with $Y_j$, we don’t need to check $Y_j$ on the next iteration.

3. Removing a value on $X_i$ can only make $Y_j$ arc-inconsistent with respect to $X_i$ itself. Thus, we only need to check that $(j,i)$ is still arc-consistent.

These insights lead a much better algorithm...
AC-3

function AC-3(csp) return the CSP, possibly with reduced domains
inputs: csp, a binary csp with variables \{X_1, X_2, ..., X_n\}
local variables: queue, a queue of arcs initially the arcs in csp
while queue is not empty do
  \((X_i, X_j) \leftarrow \text{queue.pop}()\)
  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

function REMOVE-INCONSISTENT-VALUES\((X_i, X_j)\) return \text{true} iff we remove a value
\(\text{removed} \leftarrow \text{false}\)
for each \(x\) in DOMAIN[\(X_i\)] do
  if no value \(y\) in DOMAIN[\(X_j\)] allows \((x,y)\) to satisfy the constraints between \(X_i\) and \(X_j\)
    then delete \(x\) from DOMAIN[\(X_i\)]; \(\text{removed} \leftarrow \text{true}\)
return \(\text{removed}\)

Keep track of what arcs we need to process
Add back arcs to neighbors whenever a node had values removed
AC-3: Worst Case Complexity Analysis

- All nodes can be connected to every other node,
  - so each of $n$ nodes must be compared against $n-1$ other nodes,
  - so total # of arcs is $2*n*(n-1)$, i.e. $O(n^2)$
- If there are $d$ values, checking arc $(i,j)$ takes $O(d^2)$ time
- Each arc $(i,j)$ can only be inserted into the queue $d$ times
- Worst case complexity: $O(n^2d^3)$

(For planar constraint graphs, the number of arcs can only be linear in $N$ and the time complexity is only $O(nd^3)$)
When to Iterate, When to Stop?

The crucial principle:

*If a value is removed from a node $X_i$, then the values on all of $X_i$’s neighbors must be reexamined.*

Why? *Removing* a value from a node may result in one of the neighbors becoming arc *inconsistent*, so we need to check...

(but each neighbor $X_j$ can only become inconsistent with respect to the removed values on $X_i$)
Other techniques for speeding up finding solutions for CSPs

Professor Chris Callison-Burch
Chronological backtracking

- DFS does Chronological backtracking
  - If a branch of a search fails, backtrack to the most recent variable assignment and try something different
  - But this variable may not be related to the failure

- Example: Map coloring of Australia
  - Variable order
    - Q, NSW, V, T, SA, WA, NT.
  - Current assignment:
    - Q=red, NSW=green, V=blue, T= red
  - SA cannot be assigned anything
  - But reassigning T does not help!
Backjumping: Improved backtracking

- Find “the conflict set”
  - Those variable assignments that are in conflict
  - Conflict set for SA: \{Q=red, NSW=green, V=blue\}
- Jump back to reassign one of those conflicting variables
- Forward checking can build the conflict set
  - See textbook for details
Simple CSPs can be solved quickly

1. Completely independent subproblems
e.g. Australia & Tasmania
   ▪ Easiest

2. Constraint graph is a tree
   ▪ Any two variables are connected by only a single path
   ▪ Permits solution in time linear in number of variables
   ▪ Do a topological sort and just march down the list
Cutset conditioning
Local search for CSPs

- Local search like hill-climbing search for nearby solutions that improve an objective function.

- 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 $h(n) = \text{total number of violated constraints}$
Min-Conflicts algorithm

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 complete assignment for csp
for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen conflicted variable from csp. VARIABLES
    value ← the value v for var that minimizes CONFLICTS(csp, var, v, current)
    set var = value in current
return failure
Min-Conflicts Example: n-queens

- **States**: 4 queens in 4 columns ($4^4 = 256$ states)
- **Actions**: move queen in column
- **Goal test**: no attacks
- **Evaluation**: $h(n) = \text{number of attacks}$

Given random initial state, local min-conflicts can solve n-queens in almost constant time for arbitrary n with high probability (e.g., $n = 1,000,000$)

Min-conflicts reduced time to scheduled Hubble Space Telescope from 3 weeks to 10 minutes
CIS 4210/5210: ARTIFICIAL INTELLIGENCE

Next week: Logical Agents