Today’s Class

- Constraint Satisfaction Problems
  - A.K.A., Constraint Processing / CSP paradigm
- Algorithms for CSPs
- Search Terminology

Constraint (n): A relation ... between the values of one or more mathematical variables (e.g., \(x > 3\) is a constraint on \(x\)).

Constraint satisfaction assigns values to variables so that all constraints are true.

- http://foldoc.org/constraint

Overview

- **Constraint satisfaction**: a problem-solving paradigm
- Constraint programming, constraint satisfaction problems (CSPs), constraint logic programming...
- Algorithms for CSPs
  - Backtracking (systematic search)
  - Constraint agathulization (k-consistency)
  - Variable and value ordering heuristics
  - Backjumping and dependency-directed backtracking

Search Vocabulary

- We’ve talked about caring about goals (end states) vs. paths
- These correspond to...
  - **Planning**: finding sequences of actions
    - The path to the goal is the important thing
    - Paths have various costs, depths
    - Heuristics to guide, fringe to keep backups
  - **Identification**: assignments to variables representing unknowns
    - The goal itself is important, not the path
- CSPs are specialized for identification problems

Constraint Satisfaction

- Constraint /kan'strænt/ (noun):
  - Something that limits or restricts someone or something
  - Control that limits or restricts someone’s actions or behavior
  - A relation ... between the values of one or more mathematical variables (e.g., \(x > 3\) is a constraint on \(x\)).
  - Assigns values to variables so that all constraints are true.

Bookkeeping

- HW 2 questions?
- Please note: it must be due COB Blackboard time
  - 11:59 XX is late!
  - We will not let this slide again.
- Need to upload .py files – see Nikhil if it’s not working
Slightly Less Informal

Definition of CSP

- **CSP = Constraint Satisfaction Problem**
- **Given:**
  1. A finite set of *variables*
  2. Each with a *domain* of possible values they can take (often finite)
  3. A set of *constraints* that limit the values the variables can take on
- **Solution:** an assignment of a value to each variable such that the constraints are all satisfied.

CSP Applications

- Decide if a solution exists
- Find some solution
- Find all solutions
- Find the “best solution”
  - According to some metric (objective function)
  - Does that mean “optimal”?

Informal Example: Map Coloring

- Color a map, such that:
  - Using three colors (red, green, blue)
  - No two adjacent regions have the same color

Map Coloring II

- **Variables:** A, B, C, D, E
- **Domains:** RGB = {red, green, blue}
- **Constraints:**
  - $A \neq B$, $A \neq C$, $A \neq E$, $A \neq D$, $B \neq C$, $C \neq D$, $D \neq E$
- **One solution:** $A=$red, $B=$green, $C=$blue, $D=$green, $E=$blue

Example: N-Queens (1)

- **Formulation 1:**
  - **Variables:** $X_{ij}$
  - **Domains:** $\{0, 1\}$
  - **Constraints:**
    
    $\forall i, j, k$ ($X_{ij}, X_{ik}$) $\in \{(0, 0), (0, 1), (1, 0)\}$
    $\forall i, j, k$ ($X_{ij}, X_{kj}$) $\in \{(0, 0), (0, 1), (1, 0)\}$
    $\forall i, j, k$ ($X_{ij}, X_{i+k,j+k}$) $\in \{(0, 0), (0, 1), (1, 0)\}$
    $\forall i, j, k$ ($X_{ij}, X_{i+k,j-k}$) $\in \{(0, 0), (0, 1), (1, 0)\}$
    $\sum_{i,j} X_{ij} = N$
Example: N-Queens (2)

- Formulation 2:
  - Variables: \( Q_k \)
  - Domains: \( \{1, 2, 3, \ldots, N\} \)
  - Constraints:
    - Implicit: \( \forall i, j \) non-threatening\( (Q_i, Q_j) \)
    - Explicit: \( (Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\} \)

Example: SATisfiability

- Given a set of propositions containing variables, find an assignment of the variables to \{false, true\} that satisfies them.
- For example, the clauses:
  - \( (A \lor B \lor \neg C) \land (\neg A \lor D) \)
    - (equivalent to \( (C \rightarrow A) \lor (B \land D \rightarrow A) \))
  - are satisfied by
  - \( A = \text{false} \)
  - \( B = \text{true} \)
  - \( C = \text{false} \)
  - \( D = \text{false} \)

Real-World Problems

- Scheduling
- Temporal reasoning
- Building design
- Planning
- Optimization/satisfaction
- Vision

- Graph layout
- Network management
- Natural language processing
- Molecular biology/genomics
- VLSI design

Constraint Restrictions

- **Unary** constraint: only involves one variable
  - e.g.: \( C \) can't be green.
- **Binary** constraint: only involves two variables
  - e.g.: \( E \neq D \)

Formal Definition of a CN (cont.)

- **An instantiation** is an assignment of a value \( d_x \in D \) to some subset of variables \( S \).
  - Ex: \( Q_2 = \{2, 3\} \land Q_3 = \{1, 1\} \) **instantiates** \( Q_2 \) and \( Q_3 \)
- **An instantiation is legal** iff it does not violate any constraints
- **A solution** is an instantiation of all variables
  - A **correct solution** is a legal instantiation of all variables

Formal Definition: Constraint Network (CN)

- A **constraint network** (CN) consists of
  - A set of variables \( X = \{x_1, x_2, \ldots, x_n\} \)
    - Each with an associated domain of values \( \{d_1, d_2, \ldots, d_i\} \).
    - The domains are typically finite
  - A set of constraints \( \{c_1, c_2, \ldots, c_m\} \) where
    - Each constraint defines a **predicate which is a relation** over some subset of \( X \).
    - E.g., \( c_i \) involves variables \( \{X_{i_1}, X_{i_2}, \ldots, X_{i_k}\} \) and defines the relation \( R_i \subseteq D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k} \)
Typical Tasks for CSP

- **Solutions:**
  - Does a solution exist?
  - Find one solution
  - Find all solutions
  - Given a partial instantiation, do any of the above
- Transform the CN into an equivalent CN that is easier to solve

Binary CSP

- **Binary CSP:** all constraints are binary or unary
- Can convert a non-binary CSP → binary CSP by:
  - Introducing additional variables
  - Dual graph construction: one variable for each constraint; one binary constraint for each pair of original constraints that share variables
- Can represent a binary CSP as a constraint graph with:
  - A node for each variable
  - An arc between two nodes iff there is a constraint on the two variables
  - Unary constraint appears as a self-referential arc

Example: Sudoku

- **Variables**
  - $v_{ij}$ is the value in the $j^{th}$ cell of the $i^{th}$ row
- **Domains**
  - $D_{ij} = D = \{1, 2, 3, 4\}$
- **Blocks:**
  - $B_1 = \{11, 12, 21, 22\}$, ..., $B_4 = \{33, 34, 43, 44\}$

Running Example: Sudoku

- **Constraints (implicit/intensional)**
  - $C^R$: $\forall i, \cup j v_{ij} = D$ (every value appears in every row)
  - $C^C$: $\forall j, \cup i v_{ij} = D$ (every value appears in every column)
  - $C^B$: $\forall k, \cup (v_{ij} | ij \in B_k) = D$ (every value appears in every block)

- **Alternative representation: pairwise inequality constraints**
  - $P_R$: $\forall i, j | i \neq j': v_{ij} \neq v_{ij'}$ (no value appears twice in any row)
  - $P_C$: $\forall j, i | i \neq i': v_{ij} \neq v_{ij'}$ (no value appears twice in any column)
  - $P_B$: $\forall k, ij \in B_k, i'j' \in B_k, i \neq i', j \neq j' \neq v_{ij} = v_{ij'}$ (no value appears twice in any block)

Sudoku Constraint Network

Solving Constraint Problems

1. Systematic search
   - Generate and test
   - Backtracking
2. Constraint propagation (consistency)
3. Variable ordering heuristics
4. Value ordering heuristics
5. Backjumping and dependency-directed backtracking
Generate and Test: Sudoku

- Try every possible assignment of domain elements to variables until you find one that works:

```
3 1 1
1 3 1
3 1 4
1 3 1
```

- Doesn't check constraints until all variables have been instantiated
- Very inefficient way to explore the space of possibilities ($4^7$ for this trivial Sudoku puzzle, most illegal)

Systematic Search: Backtracking

(a.k.a. depth-first search!)

- Consider the variables in some order
- Pick an unassigned variable and give it a provisional value such that it is consistent with all of the constraints
- If no such assignment can be made, we’ve reached a dead end and need to backtrack to the previous variable
- Continue this process until:
  - A solution is found, or
  - We backtrack to the initial variable and have exhausted all possible values

Problems with Backtracking

- Thrashing: keep repeating same failed variable assignments
  - Consistency checking can help
  - Intelligent backtracking schemes can also help
- Inefficiency: can spend time exploring areas of search space that aren’t likely to succeed
  - Variable ordering can help
  - IF there’s a meaningful way to order them

Consistency

- An assignment of values to variables is said to be consistent if no constraints are violated
- There are multiple kinds of consistency
- Once the whole graph is consistent, we have a solution

Node and Arc Consistency

- **Node consistency**: every value in node X’s domain is consistent with X’s unary constraints
  - A graph is node-consistent if all nodes are node-consistent
  - SA = {red, green, blue}
- **Arc consistency**: for every value x of X in Arc(X,Y), \( \exists y \) for Y that satisfies the constraint represented by the arc
  - A graph is arc-consistent if all arcs are arc-consistent

Constraint Propagation

- How do we find a set of consistent assignments?
- We perform constraint propagation
  - That is, we repeatedly reduce the domain of each variable to be consistent with its arcs
- Constraints reduce \# of legal values for a variable
  - Which may then reduce legal values of another variable
  - Then another, then another…
- Key idea: local consistency
  - Enforce nearby constraints
  - Propagate
**Constraint Propagation: Sudoku**

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Example: Map-Coloring**

- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domain:** \( D = \{ \text{red, green, blue} \} \)
- **Constraints:** adjacent regions must have different colors
  - Ex: WA \& NT \( \not\in \{\text{red, green, red, blue}, \text{green, blue, green, red, (blue, red)}\} \)
- **Solutions** are assignments satisfying all constraints, e.g.:
  - \{(WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{green}\}\}

**Constraint Graphs**

- Binary CSP: each constraint relates (at most) two variables
- Binary 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 subproblem!

**Standard Search Formulation**

- Standard search formulation of CSPs (incremental)
- Let’s start with a 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
  - Goal test: the current assignment is complete and satisfies all constraints

**Search Methods**

- What does BFS do?

- What does DFS do?

**DFS & BFS: not good!**
Backtracking Search

- Idea 1: Only consider a single variable at each point
  - Variable assignments are commutative, so fix ordering
    - I.e., \[W = \text{red} \text{ then } NT = \text{green}\] same as \[NT = \text{green} \text{ then } WA = \text{red}\]
    - Only need to consider assignments to a single variable at each step
    - How many leaves are there now?
- Idea 2: Only allow legal assignments at each point
  - I.e. consider only values which do not conflict previous assignments
    - Might have to do some computation to figure out whether a value is ok
  - "Incremental goal test"

DFS for CSPs with these two improvements is called **backtracking search**

- We backtrack when there's no legal assignment for the next variable
- Backtracking search is the basic uninformed algorithm for CSPs
- Can solve n-queens for \(n = 25\)

---

**Backtracking Example**

```
function BACKTRACKING-SEARCH(cp) returns solution?failure
  return RECURSIVE-BACKTRACKING({}, cp)

function RECURSIVE-BACKTRACKING{assignment, cp} returns solution?failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE{variables}[cp], assignment, cp
  for each value in ORDERS-DOMAIN-VALUES{var, assignment, cp} do
    if value is consistent with assignment given CONSTRAINTS{cp} then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING{assignment, cp}
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```
Improving Backtracking

- General-purpose ideas give huge gains in speed
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Filtering: Can we detect inevitable failure early?
- Structure: Can we exploit the problem structure?

Forward Checking

- Idea: Keep track of remaining legal values for unassigned variables (using immediate constraints); terminate when any variable has no legal values

Forward Checking

- Propagates information from assigned to adjacent unassigned variables
- But doesn't detect more distant failures
- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- Constraint propagation repeatedly enforces constraints locally – this is a local maximum!

Arc Consistency

- Simplest form of propagation makes each arc consistent
  - \( X \rightarrow Y \) is consistent iff for every value \( x \) 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
- What's the downside of arc consistency?
- Can be run as a preprocessor or after each assignment
K-consistency

- K-consistency generalizes the notion of arc consistency to sets of more than two variables
- A graph is K-consistent if, for legal values of any K-1 variables in the graph, and for any Kth variable \( V_k \), there is a legal value for \( V_k \)
- Strong K-consistency = J-consistency for all \( J \leq K \)
- Node consistency = strong 1-consistency
- Arc consistency = strong 2-consistency
- Path consistency = strong 3-consistency

Why Do We Care?

1. A strongly N-consistent CSP with N variables can be solved without backtracking
2. For any CSP that is strongly K-consistent:
   - If we find an appropriate variable ordering (one with "small enough" branching factor)
   - We can solve the CSP without backtracking

Ordered Constraint Graphs

- Select a variable ordering, \( V_1, \ldots, V_n \)
- Width of a node in this OCG is the number of arcs leading to earlier variables:
  \( w(V_i) = \text{Count } (V_i, V_k) \mid k < i \)
- Width of the OCG is the maximum width of any node:
  \( w(G) = \text{Max } (w(V_i)), 1 \leq i \leq N \)
- Width of an unordered CG is the minimum width of all orderings of that graph ("best you can do")

Tree-Structured Constraint Graph

- A constraint tree rooted at \( V_1 \) satisfies:
  - There exists an ordering \( V_1, \ldots, V_n \) such that every node has zero or one parents (i.e., each node only has constraints with at most one "earlier" node in the ordering)
  - Also known as an ordered constraint graph with width 1
- If this constraint tree is also node- and arc-consistent (a.k.a. strongly 2-consistent), it can be solved without backtracking
  - (More generally, if the ordered graph is strongly k-consistent, and has width \( w < k \), then it can be solved without backtracking.)

So What If We Don’t Have a Tree?

- Answer #1: Try interleaving constraint propagation and backtracking
- Answer #2: Try using variable-ordering heuristics to improve search
- Answer #3: Try using value-ordering heuristics during variable instantiation
- Answer #4: See if iterative repair works better
- Answer #5: Try using intelligent backtracking methods

Variations on Interleaving Constraint Propagation and Search

<table>
<thead>
<tr>
<th>Generate and Test</th>
<th>Simple Backtracking</th>
<th>Forward Checking</th>
<th>Partial Lookahead</th>
<th>Full Lookahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>No constraint propagation: assign all variable values, then test constraints.</td>
<td>Check constraints only for variables &quot;up the tree&quot;</td>
<td>Check constraints for immediate neighbors &quot;down the tree&quot;</td>
<td>Propagate constraints forward &quot;down the tree&quot;</td>
<td>Ensure complete arc consistency after each instantiation (AC-3)</td>
</tr>
</tbody>
</table>
### Possible Variable Orderings

**Intuition:** choose variables that are highly constrained early in the search process; leave easy ones for later.

Some possibilities:

- **Minimum width ordering (MWO):** identify OCG with minimum width
- **Maximum cardinality ordering:** approximation of MWO that's cheaper to compute: order variables by decreasing cardinality (a.k.a. **degree heuristic**)

**Fail first principle (FFP):** choose variable with the fewest values (a.k.a. **minimum remaining values** (MRV))

- **Static FFP:** use domain size of variables
- **Dynamic FFP (search rearrangement method):** At each point in the search, select the variable with the fewest remaining values

### Minimum Width

- Or "minimum remaining values" (MRV):
  - Choose the variable with the fewest remaining legal values

Why min rather than max?

Also called “most constrained variable”

“Fail-fast” ordering

### Value Ordering

**Intuition:** Choose values that are the least constrained early on, leaving the most legal values in later variables

1. **Maximal options method** (a.k.a. **least-constraining-value heuristic**): Choose the value that leaves the most legal values for not-yet-instantiated variables
2. **Min-conflicts:** For iterative repair search (Coming up)
3. Symmetry: Introduce **symmetry-breaking constraints** to constrain search space to ‘useful’ solutions (don’t examine more than one symmetric/isomorphic solution)

### Variable Orderings II

- **Maximal stable set:** find largest set of variables with no constraints between them, save these for last
- **Cycle-cutset tree creation:** Find a set of variables that, once instantiated, leave a tree of uninstantiated variables; solve these, then solve the tree without backtracking
- **Tree decomposition:** Construct a tree-structured set of connected subproblems

### Iterative Repair

- Start with an initial complete (but invalid) assignment
- Hill climbing, simulated annealing
- **Min-conflicts:** Select new values that minimally conflict with the other variables
  - Use in conjunction with hill climbing or simulated annealing or…
- **Local maxima strategies**
  - Random restart
  - Random walk
  - Tabu search: don’t try recently attempted values
Min-Conflicts Heuristic

- Iterative repair method
  1. Find some "reasonably good" initial solution
     - E.g., in N-queens problem, use greedy search through rows, putting each queen where it conflicts with the smallest number of previously placed queens, breaking ties randomly
  2. Find a variable in conflict (randomly)
  3. Select a new value that minimizes the number of constraint violations
     - O(N) time and space
  4. Repeat steps 2 and 3 until done

Performance depends on quality and informativeness of initial assignment; inversely related to distance to solution

Challenges

- What if not all constraints can be satisfied?
  - Hard vs. soft constraints
  - Degree of constraint satisfaction
  - Cost of violating constraints
- What if constraints are of different forms?
  - Symbolic constraints
  - Numerical constraints [constraint solving]
  - Temporal constraints
  - Mixed constraints

More Challenges

- What if constraints are represented intensionally?
  - Cost of evaluating constraints (time, memory, resources)
- What if constraints/variables/values change over time?
  - Dynamic constraint networks
  - Temporal constraint networks
  - Constraint repair
- What if you have multiple agents or systems involved?
  - Distributed CSPs
  - Localization techniques

Questions?

Thanks!