COMP 424 - Artificial Intelligence
Lecture 5: Constraint satisfaction problems

Instructor: Joelle Pineau (jpineau@cs.mcgill.ca)

Class web page: www.cs.mcgill.ca/~jpineau/comp424

Unless otherwise noted, all material posted for this course are copyright of the instructor, and cannot be reused or reposted without the instructor's written permission.

Dartmouth Conference (1956)

- Some of the attendees:
  - John McCarthy: LISP, time-sharing, application of logic to reasoning
  - Marvin Minsky: popularized neural networks and showed their limits, introduced slots and frames
  - Claude Shannon: information theory, juggling machine
  - Allen Newell and Herb Simon: bounded rationality, general problem solver, SOAR

- The meeting coined the term "artificial intelligence"
Marvin Minsky, Pioneer in Artificial Intelligence

By GLENN RUPPEN
JUN. 20, 2016

Marvin Minsky, who combined a scientist’s thirst for knowledge with a philosopher’s quest for truth as a pioneering explorer of artificial intelligence, work that helped inspire the creation of the personal computer and the Internet, died on Sunday night in Boston. He was 88.

His family said the cause was a cerebral hemorrhage.

Well before the advent of the microprocessor and the supercomputer, Professor Minsky, a revered computer science educator at MIT, laid the foundation for the field of artificial intelligence by demonstrating the possibilities of imparting common-sense reasoning to computers.

“Marvin was one of the very few people in computing whose visions and perspectives liberated the computer from being a glorified adding machine to start to realize its destiny as one of the most powerful amplifiers for human endeavors in history,” said Alan Kay, a computer scientist and a friend and colleague of Professor Minsky’s.


Joelle Pineau © 2013

Quiz 1

Question 1 Difficulty: 1

Hill-climbing does not retain the paths followed by search.

<table>
<thead>
<tr>
<th>True</th>
<th>116 (97.48%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>3 (2.52%)</td>
</tr>
</tbody>
</table>

Question 2 Difficulty: 1

Simulated annealing uses more memory than depth-first search.

<table>
<thead>
<tr>
<th>True</th>
<th>13 (10.02%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>105 (89.98%)</td>
</tr>
</tbody>
</table>

Question 3 Difficulty: 1

The temperature parameter in simulated annealing should be set to as low a value as possible.

<table>
<thead>
<tr>
<th>True</th>
<th>11 (9.24%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>108 (86.76%)</td>
</tr>
</tbody>
</table>

Question 4 Difficulty: 1

Under what conditions would you prefer simulated annealing over hill-climbing? (1 sentence)

<table>
<thead>
<tr>
<th>Answers</th>
<th>118 (98.16%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other</td>
<td>1 (0.84%)</td>
</tr>
</tbody>
</table>

Question 5 Difficulty: 1

Would you ever prefer hill-climbing? If so, when? (1 sentence)

<table>
<thead>
<tr>
<th>Answers</th>
<th>118 (99.16%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other</td>
<td>1 (0.84%)</td>
</tr>
</tbody>
</table>
Why do quizzes?

1. Testing aids later retention.
2. Testing identifies gaps in knowledge.
3. Testing causes students to learn more from the next learning episode.
4. Testing produces better organization of knowledge.
5. Testing improves transfer of knowledge to new concepts.
6. Testing can facilitate retrieval of information that was not tested.
7. Testing improves metacognitive monitoring.
8. Testing prevents interference from prior material when learning new material.
9. Testing provides feedback to instructors.
10. Frequent testing encourages students to study.

From H. L. Roediger, Washington U., as presented in http://www.theatlantic.com/education/archive/2014/01/students-should-be-tested-more-not-less/283195/

Quick recap from last week

- **Constructive methods**: Start from scratch and build up a solution.
  - Informed / uninformed methods.

- **Iterative improvement/repair methods**: Start with a solution (which may be broken / suboptimal) and improve it.
  - Hill-climbing, simulated annealing.

- **(Optional) Global search**: Start from multiple states that are far apart, and go all around the state space.
  - Read about genetic algorithms (R&N, Sec.4.1)
Searching with constraints

- Many interesting problems have strict constraints:
  E.g. Must visit city A (to re-supply) before visiting city B (to sell).

- How can we incorporate this information in the search process?
  - At a minimum: Ensure the search will be limited to solutions that respect the constraints. Sometimes very few “legal solutions”.
  - Ideally: Use the constraints to narrow the search space.

Example

- Color a map so that no adjacent territories have the same color.
Constraint satisfaction problems (CSPs)

- A CSP is defined by:
  - Set of variables $V_i$ that can take values from domain $D_i$
  - Set of constraints specifying what combinations of values are allowed (for subsets of variables, eg. pairs of variables)
  - Constraints can be represented:
    - Implicitly, as a function, testing for the satisfaction of the constraint. E.g. $C_1 \neq C_2$
    - Explicitly, as a list of allowable values. E.g. $(C_1=\text{R}, C_2=\text{G}), (C_1=\text{G}, C_2=\text{R}), (C_1=\text{B}, C_2=\text{R}), …$

- A CSP solution is an assignment of values to variables such that all the constraints are true.

- We typically want to find any solution or find that there is no solution.

Example

- Variables $WA, NT, Q, NSW, V, SA, T$
- Domains $D_i = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors

E.g., $WA \neq NT$
Example

- Solutions are complete and consistent assignments.

E.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

Varieties of variables

- Boolean variables (e.g. satisfiability)
- Finite domain, discrete variables (e.g. colouring)
- Infinite domain, discrete variables (e.g. start/end time of operation in scheduling)
- Continuous variables.

Problem complexity? Ranges from solvable in poly-time (e.g. linear programming) to NP-complete to undecidable.
Varieties of constraints

- **Unary**: involve one variable and one constraint.
- **Binary**.
- **Higher-order** (involve 3 or more variables).
- **Relations**:
  - $T_1 + d_1 \leq T_2$ (Task$_2$ has to come after Task$_1$)  
    Scheduling
  - `Alldiff` (variables in a row), `Alldiff` (variables in a column),
    `Alldiff` (variables in a 3x3 square).  
    Sudoku
- **Preferences** (soft constraints): can be represented using costs
  and lead to constrained optimization problems.

Real-world CSPs

Often involves allocating *limited* resources:

- **Timetable problems** (e.g. which class is offered when, where)
- **Hardware configuration**.
- **Transportation scheduling**, Factory scheduling, **Floor planning**.
- **Puzzle solving** (crosswords, Sudoku)
Overview of approaches for solving CSPs

- **Constructive approach**: Start with this.
  - State is defined by the set of values assigned so far.
  - Apply forward search to fill the solution.
  - This is a general purpose algorithm which works for all CSPs.

- **Random approach**: Last part of the lecture.
  - Start with a broken but complete assignment of values to variables.
  - Gradually fix broken constraints by re-assigning variables.
  - Essentially use optimization approaches (hill-climbing, simulated annealing).

Constructive search for CSPs

- **Problem definition**:
  - **State**: defined by set of values assigned so far, could be *partial* and/or *inconsistent* assignment.
  - **Initial state**: all variables are unassigned.
  - **Operators**: assign a value to an unassigned variable.
  - **Goal test**: all variables assigned, no constraint violated. i.e. *complete and consistent* assignment.

- **Problem has deterministic action, fully observable state.**

  **Important observation**: Depth is limited to the number of variables, \( n \).
  So we can apply DFS (or depth-limited search).
Example

- Color abstract map so that adjacent countries don’t same color.
  - Variables: Countries $C_i$
  - Domains: {Red, Blue, Green}
  - Constraints: \{$C_1 \neq C_2$, $C_1 \neq C_5$, \ldots\}

Standard uninformed search for map coloring

- Is this complete? Optimal?
  Yes: known solution depth. Yes: If we check the constraints.
- Is this a practical approach? What is the complexity?
  \[(n \times d) \times [(n-1) \times d] \times [(n-2) \times d] \times \ldots \times [2 \times d] \times d = n! \times d^n\]
Analysis of the simple approach

Branching factor is very high: \[ \sum_i d \quad (i \text{ sums over unassigned variables}). \]

BUT: There can be only \(d^n\) unique complete assignments.

More important observations:
- Order in which variables are assigned is irrelevant \(\rightarrow\) Many paths are equivalent!
- Adding assignments cannot correct a violated constraint!

Constraint graph

- We want to reason about constraints, i.e. perform inference.
- Constraint graph: nodes are variables, arcs show constraints.
- Graph structure can be exploited to accelerate solution search.

E.g. Map colouring:
Example: 4 queens problem

- Put 4 queens on 4x4 board so that none attack each other.
- Put one queen in each column. Which row does each one go in?
- Variables: \{Q_1, Q_2, Q_3, Q_4\}
- Domain: \{1, 2, 3, 4\}
- Constraints:
  - \( Q_i \neq Q_j \) (cannot be in same row)
  - \(|Q_i - Q_j| \neq |i - j|\) (cannot be in same diagonal)
- Translate each constraint into set of allowable values for its variables.
  - e.g. values for \( (Q_1, Q_2) \): \((Q_1 = 1, Q_2 = 3), (Q_1 = 1, Q_2 = 4), (Q_1 = 2, Q_2 = 4) \) etc.

Constraint graph?
Adding consistency constraints

Idea: Pre-process the graph to remove obvious inconsistencies.

- A **variable** is **arc-consistent** if every value in its domain satisfies that variable’s binary constraints.
- A **network** is **generalized arc-consistent** if every value in the domain of every variable are simultaneously arc-consistent.

E.g. A CSP with variables A, B, C, each with domain {1, 2, 3, 4}

Arc-consistent variables:
- A={1, 2, 3}; B={2, 3}; C={2, 3, 4}

Generalized arc-consistent variables:
- A={1, 2}; B={2, 3}; C={3, 4}

How does this help us? Reduced domain = reduced search.

Backtracking search

Like Depth-First Search, but **fix order of variable assignment** (so \(|D|\)).

**Algorithm:**
- Select an unassigned variable, \(X\).
- For each \(value=x_1, \ldots, x_n\) in the domain of that variable
  - If the value satisfies the constraints, let \(X=x_i\) and exit the loop.
  - If an assignment was found, move to the next variable.
  - If no assignment, go back to preceding variable and try different value.

- This is the basic uninformed algorithm for CSPs.
  - Can solve n-queens for \(n=25\).
Forward checking

Idea: Keep track of legal values for unassigned variables.

- When you assign a variable X
  - look at each unassigned variable Y connected to X (by a constraint)
  - delete from Y's domain any value that is inconsistent the value of X
- Can solve n-queens for $n \approx 30$.

E.g. Map coloring.

<table>
<thead>
<tr>
<th></th>
<th>Nothing assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>RGB</td>
</tr>
<tr>
<td>C2</td>
<td>RGB</td>
</tr>
<tr>
<td>C3</td>
<td>RGB</td>
</tr>
<tr>
<td>C4</td>
<td>RGB</td>
</tr>
<tr>
<td>C5</td>
<td>RGB</td>
</tr>
<tr>
<td>C6</td>
<td>RGB</td>
</tr>
</tbody>
</table>
Forward checking

Idea: Keep track of legal values for unassigned variables.
- When you assign a variable X
  - look at each unassigned variable Y connected to X (by a constraint)
  - delete from Y’s domain any value that is inconsistent the value of X

E.g. Map coloring.

<table>
<thead>
<tr>
<th></th>
<th>Nothing assigned</th>
<th>Assign C1 = Red</th>
<th>Assign C2 = G</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>RGB</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>RGB</td>
<td>GB</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>RGB</td>
<td>GB</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>RGB</td>
<td>RGB</td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td>RGB</td>
<td>GB</td>
<td>B by forward checking!</td>
</tr>
<tr>
<td>C6</td>
<td>RGB</td>
<td>RGB</td>
<td>RB</td>
</tr>
</tbody>
</table>
Forward checking

**Idea**: Keep track of legal values for unassigned variables.

- When you assign a variable $X$
  - look at each unassigned variable $Y$ connected to $X$ (by a constraint)
  - delete from $Y$'s domain any value that is inconsistent the value of $X$

**E.g. Map coloring.**

<table>
<thead>
<tr>
<th></th>
<th>Nothing assigned</th>
<th>Assign $C_1$ = Red</th>
<th>Assign $C_2$ = G</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>RGB</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>$C_2$</td>
<td>RGB</td>
<td>GB</td>
<td>G</td>
</tr>
<tr>
<td>$C_3$</td>
<td>RGB</td>
<td>GB</td>
<td>GB</td>
</tr>
<tr>
<td>$C_4$</td>
<td>RGB</td>
<td>RGB</td>
<td>RGB</td>
</tr>
<tr>
<td>$C_5$</td>
<td>RGB</td>
<td>GB</td>
<td>B by forward checking!</td>
</tr>
<tr>
<td>$C_6$</td>
<td>RGB</td>
<td>RGB</td>
<td>RB</td>
</tr>
</tbody>
</table>

Can also apply **generalized arc-consistency**!
Forward checking

**Idea:** Keep track of legal values for unassigned variables.

- When you assign a variable X
  - look at each unassigned variable Y connected to X (by a constraint)
  - delete from Y’s domain any value that is inconsistent the value of X

E.g. Map coloring.

<table>
<thead>
<tr>
<th></th>
<th>Nothing assigned</th>
<th>Assign C1 = Red</th>
<th>Assign C2 = G</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>RGB</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>C2</td>
<td>RGB</td>
<td>GB</td>
<td>G</td>
</tr>
<tr>
<td>C3</td>
<td>RGB</td>
<td>GB</td>
<td>GB</td>
</tr>
<tr>
<td>C4</td>
<td>RGB</td>
<td>RGB</td>
<td>RGB, GB</td>
</tr>
<tr>
<td>C5</td>
<td>RGB</td>
<td>GB</td>
<td>B</td>
</tr>
<tr>
<td>C6</td>
<td>RGB</td>
<td>RGB</td>
<td>RB, R</td>
</tr>
</tbody>
</table>

Can also apply generalized arc-consistency!

Heuristics for CSP search

- More intelligent decisions on:
  - Which variable to assign next?
  - Which value to choose for the variable?

E.g. Sudoku:
Common heuristics (for faster search)

• To select a variable:
  1. Minimum-remaining values: Choose the variable that is the most constrained (i.e. fewest legal values).
  2. Degree heuristic: Choose the variable that imposes the most constraints on the remaining variables.
     • Use this to break ties from Minimum-remaining value heuristic

• To select a value:
  – Least-constraining value: Assign the value that rules out the fewest values for other variables.

Taking advantage of problem structure

• Worst-case complexity is $d^n$ (where $d$ is the number of possible values and $n$ is the number of variables.
• But a lot of problems are much easier!
• Disjoint components can be solved independently.
• Tree-structured constraint graph: complexity is $O(nd^2)$
Taking advantage of problem structure

- **Nearly-tree structured graph**: complexity is $O(d^c(n-c)d^2)$ using cutset conditioning:
  - Find a set of variables which, when removed, turn graph into tree.
  - Instantiate them all possible ways. Good if $c$ (size of cutset) is small.

Key insight of CSP: Leverage structure to accelerate solving!

Local search for CSPs

**General idea**: Iterative improvement algorithm

- Start with a broken but complete assignment of values to variables.
- Allow variable assignments that don’t satisfy some constraints.
- Randomly select any conflicted variables.
- Operators reassign variable values.
  - **Min-conflicts heuristic** chooses value that violates the fewest number of constraints.

a.k.a. Hill-climbing optimization! (Could also use simulated annealing.)
Example: 4-Queens

- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation function: number of attacks

Performance of min-conflicts heuristic

- Given random initial state, can solve $n$-queens in almost constant time for arbitrary $n$ with high probability (e.g. $n=10^7$). Why?
- The same appears to be true for many randomly-generated CSPs, except in a narrow range of the ratio:

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$
Summary

- CSPs are everywhere. Be able to recognize them!
- Know how to cast CSP solving as a search problems.
- Understand basic concepts: constraint graph, arc consistency.
- Understand both constructive and iterative improvement methods to solve CSPs.
- Know how to apply the various heuristics: minimum-remaining-values, least-constraining value, degree
- Iterative improvement methods using min-conflict heuristic are very general and often work better.