CSE 473: Artificial Intelligence

Constraint Satisfaction Examples

Cynthia Matuszek

Multiple slides adapted from Dan Klein, Stuart Russell, Andrew Moore, Paula Matuszek
Why do we care about CSPs?

- **Standard search problems:**
  - State is a “black box”
  - **Any function** can be goal, successor function can be anything

- **Constraint satisfaction problems (CSPs):**
  - Search problems that vary in the goal test.
  - State is defined by variables $X_i$ with values from a domain $D$.
  - Goal test is a **set of constraints**

- **Why do we care?**
  - Allows for **informed search**
  - Using structure of problems to search **wisely**
Revisiting and Reviewing

- Uninformed Search for Constraint Satisfaction Problems
- Backtracking Search
- Forward Checking
- $k$-Consistency
- Ordering Heuristics
  - Minimum Remaining Values Ordering
  - Least Constraining Values
- Tree- and almost-tree CSPs
Bread-first search & CSPs

$X = \{A, B\}$
$D = \{\text{red}, \text{green}\}$
Goal: $A = B$
X = \{A, B, C\}
D = \{red, green\}
Goal: A = B = C

...this isn't so good:

1. Lots of duplication
2. BFS always fills out the top of the search tree, when the solutions are at the bottom
Can We Do Better?

- It’s actually hard to understand why uninformed search does so badly. Why?
- Because you would never implement these problems that way.
  - Better successor functions, internal checks, …
- Hence, “uninformed”
Improvement 1: Commutativity

- Idea 1: Only consider a single variable at each point
  - Variable assignments are commutative, so fix ordering
  - I.e., [A = red then B = green] same as [B = green then A = red]
  - Only need to consider assignments to a single variable at each step
Improvement 1: Commutativity

- Idea 1: Only consider a single variable at each point
  - Variable assignments are commutative, so fix ordering
  - I.e., [A = red then B = green] same as [B = green then A = red]
  - Only need to consider assignments to a single variable at each step

![Diagram showing commutativity of variable assignments]
Improvement 2: Legal Assignments

- Idea 2: Only allow legal assignments at each point
  - Only assign values which don’t *eventually* doom the search
  - Might have to do some extra computation
  - “Incremental goal test”
**Improvement 2: Legal Assignments**

- **Idea 2:** Only allow legal assignments at each point
  - Only assign values which do not conflict with existing assignments
  - Might have to do some extra computation
  - “Incremental goal test”

![Diagram showing legal assignments and conflicts](image-url)
Idea 1 + Idea 2 = Backtracking

- Depth-first search for CSPs with these fixes is **backtracking search**
  - Backtrack when there’s no legal assignment for the next variable
- Backtracking search is the basic uninformed algorithm for CSPs

```python
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
```
Idea 1

\[ X = \{A, B, C\} \]
\[ D = \{\text{red, green}\} \]
Goal: \( A = B = C \)
Backtracking

- **Plus Idea 2**

X = \{A, B, C\}
D = \{red, green\}
Goal: A = B = C
Backtracking Example

Interim goal check

Can we do better?
What about never getting here?
Forward Checking

- Idea: Keep track of remaining legal values for unassigned variables (using immediate constraints)
- Terminate/prune when any *as-yet-unassigned* variable has no legal values
Forward Checking

- Idea: Keep track of remaining legal values for unassigned variables (using immediate constraints)
- Terminate/prune when any *as-yet-unassigned* variable has no legal values
Improving Forward Checking

- Why does forward checking allow this?
- Forward checking *propagates* information from assigned to adjacent unassigned variables, but doesn't detect more distant failures
Why does forward checking allow this?

Neither SA nor NT have *no* possible assignments.

How do we fix it?
Arc Consistency

- Simplest form of propagation makes each arc consistent
- Every pair of variables that affect each other share an arc
  - $X \rightarrow Y$ is consistent iff for every value $x$ there is some allowed
- If $X$ loses a value, neighbors of $X$ need to be rechecked!
Revisiting and Reviewing

- Uninformed Search for Constraint Satisfaction Problems
- Backtracking Search
- Forward Checking
- $k$-Consistency
- Ordering Heuristics
  - Minimum Remaining Values
  - Least Constraining Values
- Tree- and almost-tree CSPs
Limitations of Arc Consistency

- After running arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

- How can we fix it?

What went wrong here?
Variable Choice: Minimum Remaining Values

- Minimum remaining values (MRV):
  - Choose the variable with the fewest legal values
Ordering: Degree Heuristic

- Tie-breaker among MRV variables
- Degree heuristic:
  - Choose the variable participating in the most constraints on remaining variables (has the most arcs)

- Why most rather than fewest constraints?
Ordering: Least Constraining Value

- Given a choice of variable:
  - Choose the one that rules out the fewest values in the remaining variables

- Why?

- Computationally expensive (sometimes)
Tree-Structured CSPs

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

  For \( i = n : 2 \), apply RemoveInconsistent(Parent(\( X_i \)), \( X_i \))

  For \( i = 1 : n \), assign \( X_i \) consistently with Parent(\( X_i \))

- Runtime: \( O(n \cdot d^2) \)

- Takeaway: tree-structured CSPs can be solved very efficiently
Nearly Tree-Structured CSPs

- Cutset conditioning:
  - Choose variable to instantiate that makes everything *left* into a tree
- Instantiate a variable every possible way
  - Here, you now have 3 tree-search problems
- Takeaway: you can turn some CSPs into trees (which can still be solved very efficiently)