CSE 40171: Artificial Intelligence

Constraint Satisfaction Problems: Inference and Backtracking Search
Homework #3 has been released
It is due at 11:59PM on 10/9
In the regular state-space search algorithms, we could only do one thing:

Search
With CSPs, we can do two things:

1. **Search**
2. **Use constraints to reduce the number of legal values for a variable**

And this reduction can propagate to neighboring variables
Local Consistency

Enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph.

So where do we enforce it?
In South Australia, the color green is disliked.
Vertex Consistency

A single variable is **vertex-consistent** if all the values in its domain stratify its unary constraints.

Starting Domain: \(\{\text{red, green, blue}\}\)
Reduced Domain: \(\{\text{red, blue}\}\)

Image credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188
An edge $X \rightarrow Y$ is **consistent** iff for every $x$ in the tail there is some $y$ in the head which could be assigned without violating a constraint.

**Forward Checking**: Enforcing consistency of edges pointing to each new assignment.
Edge Consistency of an Entire CSP

A simple form of propagation makes sure **all** edges are consistent:

- Important: If $X$ loses a value, neighbors of $X$ need to be rechecked
- Edge consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What’s the downside of enforcing edge consistency?

---

Image credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188
Enforcing Edge Consistency in a CSP

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- ...but detecting all possible future problems is NP-hard – why?
Limitations of Edge Consistency

After enforcing arc consistency:

- Can have one solution left
- Can have multiple solutions left
- Can have no solutions left (and not know it)

What went wrong here?

Image credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188
Path Consistency

Tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables

Let’s color the map with two colors:

Make the set \{WA, SA\} path consistent with respect to \(NT\)

Two options: \(WA = red, SA = blue\) and \(WA = blue, SA = red\)

What do we assign to \(NT\)?
Let’s generalize the notion of consistency:

A CSP is \( K \)-consistent if, for any set of \( k - 1 \) variables, and for any consistent assignment to those variables, a consistent value can always be assigned to any \( k \)th variable.

1-consistency? vertex consistency
2-consistency? edge consistency
3-consistency? path consistency
$K$-consistency

A CSP is strongly $K$-consistent if it is also $(k - 1)$-consistent, $(k - 2)$-consistent, all the way down to 1-consistent.

Assume we have a CSP with $n$ nodes and want to make it strongly $n$-consistent:

For each variable $X_i$, we only need to search through the $d$ values in the domain to find a value consistent with $X_1, \ldots, X_{i-1}$.

Guaranteed solution in time $O(n^2d)$

- time is exponential in $n$ in the worst case!
- space is also exponential in $n$!
Global Constraints

The *Alldiff* constraint states that all variables involved must have distinct values.

Inconsistency detector: if $m$ variables are involved in the constraint, and if they have $n$ possible district values, and $m > n$, then the constraint cannot be satisfied.

Can we detect the inconsistency in the assignment $\{WA = red, NSW = red\}$?
Resource Constraints

We can also have an *Atmost* constraint, meaning no more than $n$ resources can be assigned.

Scheduling Example:

# of personnel assigned to 4 tasks: $P_1, \ldots, P_4$

Constraint that no more than 10 people are assigned:

*Atmost* $(10, P_1, P_2, P_3, P_4)$

If the domain of each variable is \{3, 4, 5, 6\}, is *Atmost* satisfied?
Bounds Propagation

Capacity: 165
$D_1 = [0, 165]$

Capacity: 385
$D_2 = [0, 385]$

Additional Constraint: the two flights must carry 420 people

$D_1 = [35, 165]$

$D_2 = [255, 385]$
Backtracking Search
Backtracking Search

Some problems, like Sudoku, can be solved by inference over constraints

- But this is not true for all problems

**Backtracking search** is the basic uninformed algorithm for solving CSPs
Idea 1: One variable at a time

- Variable assignments are commutative, so fix ordering

- For example, \([WA = red \text{ then } NT = green]\) is the same as \([NT = green \text{ then } WA = red]\)

- Only need to consider assignments to a single variable at each step

Slide credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188
Idea 2: Check constraints as you go

• For example, consider only values which do not conflict with previous assignments

• Might have to do some computation to check the constraints

• “Incremental goal test”

Slide credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188
Backtracking Example

Image credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188
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 = DFS + variable-ordering + fail-on-violation
Improving Backtracking Search

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

Filtering: Keep track of domains for unassigned variables and cross off bad options

Forward checking: Cross off values that violate a constraint when added to the existing assignment
Ordering
Ordering: Minimum Remaining Values

**Variable Ordering:** Minimum remaining values (MRV):

Choose the variable with the fewest legal values left in its domain.

Slide credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188
Ordering: Minimum Remaining Values

Why min rather than max?

Also called “most constrained variable”

“Fail-fast” ordering

Slide credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188
Ordering: Least Constraining Value

Value Ordering: Least Constraining Value

• Given a choice of variable, choose the least constraining value

• For example, the one that rules out the fewest values in the remaining variables

• Note that it may take some computation to determine this (e.g., rerunning filtering)
Ordering: Least Constraining Value

Why least rather than most?

- Combining these ordering ideas makes problems like 1000 queens feasible

Image credit: Dan Klein and Pieter Abbeel, UC Berkeley CS 188
Structure

- These solutions we’ve seen before:
  - Check the consistency of a single edge
  - Check the edge consistency of an entire CSP
    - The AC-3 algorithm!