Constraint Satisfaction Problems

Robert Platt
Northeastern University

Some images and slides are used from:
1. CS188 UC Berkeley
2. RN, AIMA

Image: Berkeley CS188 course notes (downloaded Summer 2015)
What is a CSP?

A CSP is defined by:

1. a set of variables and their associated domains
2. a set of constraints that must be satisfied.
CSP example: map coloring

Problem: assign each territory a color such that no two adjacent territories have the same color

Variables: \( X = \{WA, NT, Q, NSW, V, SA, T\} \)

Domain of variables: \( D = \{r, g, b\} \)

Constraints: \( C = \{SA \neq WA, SA \neq NT, SA \neq Q, \ldots \} \)
CSP example: n-queens

Problem: place n queens on an nxn chessboard such that no two queens threaten each other

Variables: $X = ?$

Domain of variables: $D = ?$

Constraints: $C = ?$
CSP example: n-queens

Problem: place n queens on an nxn chessboard such that no two queens threaten each other

Variables: $X = \text{One variable for every square}$

Domain of variables: $D = \text{Binary}$

Constraints: $C = \text{Enumeration of each possible disallowed configuration}$

- why is this a bad way to encode the problem?
CSP example: n-queens

Problem: place n queens on an nxn chessboard such that no two queens threaten each other.

Variables:

Domain of variables:

Constraints:

1. One variable for every square
2. Binary
3. Enumeration of each possible disallowed configuration

Is there a better way?

– why is this a bad way to encode the problem?
CSP example: n-queens

Problem: place n queens on an nxn chessboard such that no two queens threaten each other

Variables: $X$ = One variable for each row

Domain of variables: $D$ = A number between 1 and 8

Constraints: $C$ = Enumeration of disallowed configurations

– why is this representation better?
The constraint graph

Variables represented as nodes (i.e. as circles)

Constraint relations represented as edges

– map coloring is a binary CSP, so it's easier to represent...
A harder CSP to represent: Cryptarithmetic

- **Variables:**
  \[
  F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3
  \]

- **Domains:**
  \[\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\]

- **Constraints:**
  \[
  \text{alldiff}(F, T, U, W, R, O) \\
  O + O = R + 10 \cdot X_1 \\
  \ldots
  \]
Another example: sudoku

- **Variables:**
  - Each (open) square

- **Domains:**
  - \{1,2,...,9\}

- **Constraints:**
  - 9-way alldiff for each column
  - 9-way alldiff for each row
  - 9-way alldiff for each region
  (or can have a bunch of pairwise inequality constraints)

Slide: Berkeley CS188 course notes (downloaded Summer 2015)
Naive solution: apply BFS, DFS, A*, ...

Which would be better: BFS, DFS, A*?

– remember: it doesn't know if it reached a goal until all variables are assigned ...
Naive solution: apply BFS, DFS, A*, ...

How many leaf nodes are expanded in the worst case?
Naive solution: apply BFS, DFS, A*, ...

How many leaf nodes are expanded in the worst case? \(3^7 = 2187\)
Naive solution: apply BFS, DFS, A*, ...

This sucks.
How can we improve it?

How many leaf nodes are expanded in the worst case? \(3^7 = 2187\)
Backtracking search

When a node is expanded, check that each successor state is consistent before adding it to the queue.
Backtracking search

When a node is expanded, check that each successor state is consistent before adding it to the queue. Does this state have any valid successors?
Backtracking search

function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({}, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment then
            add \{var = value\} to assignment
            inferences ← INERENCE(csp, var, value)
            if inferences ≠ failure then
                add inferences to assignment
                result ← BACKTRACK(assignment, csp)
                if result ≠ failure then
                    return result
                remove \{var = value\} and inferences from assignment
            return failure

– backtracking enables us the ability to solve a problem as big as 25-queens
Forward checking

Sometimes, failure is inevitable:

Can we detect this situation in advance?
Forward checking

Sometimes, failure is inevitable:

Can we detect this situation in advance?

Yes: keep track of viable variable assignments as you go
Forward checking

Track domain for each unassigned variable
  – initialize with domains from problem statement
  – each time you expand a node, update domains of all unassigned variables
Forward checking

Track domain for each unassigned variable
  – initialize w/ domains from problem statement
  – each time you expand a node, update domains of all unassigned variables
Forward checking

Track domain for each unassigned variable
- initialize w/ domains from problem statement
- each time you expand a node, update domains of all unassigned variables
Forward checking

Track domain for each unassigned variable
- initialize w/ domains from problem statement
- each time you expand a node, update domains of all unassigned variables
But, failure was inevitable here!
– what did we miss?
Arc consistency

- An arc $X \rightarrow Y$ 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 arcs pointing to each new assignment.

Slide: Berkeley CS188 course notes (downloaded Summer 2015)
Arc consistency

- An arc $X \rightarrow Y$ 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 arcs pointing to each new assignment.

---

Slide: Berkeley CS188 course notes (downloaded Summer 2015)
Forward checking

But, failure was inevitable here!
– what did we miss?
Arc consistency

- A simple form of propagation makes sure all arcs are consistent:

Delete values from tail in order to make each arc consistent.

**Consistent:** for every value in the tail, there is some value in the head that could be assigned w/o violating a constraint.
Arc consistency

- A simple form of propagation makes sure all arcs are consistent:

Delete values from tail in order to make each arc consistent

**Consistent**: for every value in the tail, there is some value in the head that could be assigned w/o violating a constraint.
Arc consistency

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

Delete values from tail in order to make each arc consistent

**Consistent**: for every value in the tail, there is some value in the head that could be assigned w/o violating a constraint.
Arc consistency

- A simple form of propagation makes sure all arcs are consistent:

Delete values from tail in order to make each arc consistent

Consistent: for every value in the tail, there is some value in the head that could be assigned w/o violating a constraint.
Arc consistency

- A simple form of propagation makes sure all arcs are consistent:

Consistent: for every value in the tail, there is some value in the head that could be assigned w/o violating a constraint.
Arc consistency

- A simple form of propagation makes sure all arcs are consistent:

Delete values from tail in order to make each arc consistent

**Consistent**: for every value in the tail, there is some value in the head that could be assigned w/o violating a constraint.
Arc consistency

- A simple form of propagation makes sure all arcs are consistent:

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

function $\text{AC-3}(csp)$ returns false if an inconsistency is found and true otherwise
inputs: $csp$, a binary CSP with components $(X, D, C)$
local variables: $queue$, a queue of arcs, initially all the arcs in $csp$

while $queue$ is not empty do
  $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(queue)$
  if $\text{REVISE}(csp, X_i, X_j)$ then
    if size of $D_i = 0$ then return false
    for each $X_k$ in $X_i$.NEIGHBORS - $\{X_j\}$ do
      add $(X_k, X_i)$ to $queue$
  return true

function $\text{REVISE}(csp, X_i, X_j)$ returns true iff we revise the domain of $X_i$
revised $\leftarrow$ false
for each $x$ in $D_i$ do
  if no value $y$ in $D_j$ allows $(x,y)$ to satisfy the constraint between $X_i$ and $X_j$ then
    delete $x$ from $D_i$
    revised $\leftarrow$ true
return revised

Why does this algorithm converge?
Arc consistency does not detect all inconsistencies...

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

- Arc consistency still runs inside a backtracking search!

Slide: Berkeley CS188 course notes (downloaded Summer 2015)
Heuristics for improving CSP performance

Minimum remaining values (MRV) heuristic:

– expand variables w/ minimum size domain first
Heuristics for improving CSP performance

Minimum remaining values (MRV) heuristic:

– expand variables w/ minimum size domain first
Heuristics for improving CSP performance

Minimum remaining values (MRV) heuristic:

– expand variables w/ minimum size domain first
Least constraining value (LCV) heuristic:

– consider how domains of neighbors would change under A.C.

– choose value that constrains neighboring domains the least
Least constraining value (LCV) heuristic:

- consider how domains of neighbors would change under A.C.
- choose value that constrains neighboring domains the least

The combination of MRV and LCV w/ backtracking can solve the 1000-queens problem.
In general, what is the complexity of solving a CSP using backtracking?

(in terms of # variables, n, and max domain size, d)

But, sometimes CSPs have special structure that makes them simpler!
When the constraint graph is a tree

This CSP is easier to solve than the general case...
When the constraint graph is a tree

1. Do a *topological sort* 
   – a partial ordering over variables
   
i. choose any node as the root
   ii. list children after their parents

Image: Berkeley CS188 course notes (downloaded Summer 2015)
When the constraint graph is a tree

2. make the graph *directed arc consistent* – start w/ the tail and make each variable arc consistent wrt its parents

Image: Berkeley CS188 course notes (downloaded Summer 2015)
When the constraint graph is a tree

2. make the graph *directed arc consistent* – start w/ the tail and make each variable arc consistent wrt its parents
When the constraint graph is a tree

2. make the graph *directed arc consistent* – start w/ the tail and make each variable arc consistent wrt its parents

Image: Berkeley CS188 course notes (downloaded Summer 2015)
When the constraint graph is a tree

2. make the graph *directed arc consistent* – start w/ the tail and make each variable arc consistent wrt its parents
When the constraint graph is a tree

2. make the graph *directed arc consistent* – start w/ the tail and make each variable arc consistent wrt its parents
When the constraint graph is a tree

2. make the graph *directed arc consistent*
   – start w/ the tail and make each variable arc consistent wrt its parents

![Diagram of directed graph](Image: Berkeley CS188 course notes (downloaded Summer 2015))
When the constraint graph is a tree

2. make the graph directed arc consistent
   – start w/ the tail and make each variable arc consistent wrt its parents

Image: Berkeley CS188 course notes (downloaded Summer 2015)
When the constraint graph is a tree

3. Now, start at the root and do backtracking – will backtracking ever actually backtrack?

So, what's the time complexity of this algorithm?
Using structure to reduce problem complexity

But, what if the constraint graph is not a tree? – is there anything we can do?

But, sometimes CSPs have special structure that makes them simpler!
Using structure to reduce problem complexity

But, what if the constraint graph is not a tree? – is there anything we can do?

This is not a tree...
Cutset conditioning

1. Turn the graph into a tree by assigning values to a subset of variables
2. For each assignment to the subset, prune domains of the rest of the variables and solve the sub-problem CSP.
   – what does efficiency of this approach depend on?
Cutset conditioning

How many variables need to be assigned to turn this graph into a tree?