Constraint Satisfaction Problems

- **Standard search problems:**
  - State is a “black box”: arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything

- **Constraint satisfaction problems (CSPs):**
  - A special subset of search problems
  - State is defined by variables $X_i$ with values from a domain $D$ (sometimes $D$ depends on $i$)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

- Simple example of a *formal representation language*

- Allows useful general-purpose algorithms with more power than standard search algorithms
Example: Map Coloring

- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** $D = \{\text{red, green, blue}\}$
- **Constraints:** adjacent regions must have different colors
  - Implicit: WA $\neq$ NT
  - Explicit: $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \ldots\}$
- **Solutions** are assignments satisfying all constraints, e.g.:
  $$\{\text{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}\}$$
Example: N-Queens

- **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

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

- Assume all objects:
  - Have no shadows or cracks
  - Three-faced vertices
  - “General position”: no junctions change with small movements of the eye.

- Then each line on image is one of the following:
  - Boundary line (edge of an object) (→) with right hand of arrow denoting “solid” and left hand denoting “space”
  - Interior convex edge (+)
  - Interior concave edge (-)
Only certain junctions are physically possible
How can we formulate a CSP to label an image?
Variables: vertices
Domains: junction labels
Constraints: both ends of a line should have the same label

Legal Junctions
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!

[Demo: CSP applet (made available by aispace.org) -- n-queens]
Example: 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:
  \[\text{alldiff}(F,T,U,W,R,O)\]
  \[O + O = R + 10 \cdot X_1\]
  \[\ldots\]
Real-World CSPs

- Scheduling problems: e.g., when can we all meet?
- Timetabling problems: e.g., which class is offered when and where?
- Assignment problems: e.g., who teaches what class
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- ... lots more!

- Many real-world problems involve real-valued variables...
Standard Search Formulation

- Standard search formulation of CSPs

- States defined by the values assigned so far (partial assignments)
  - 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

- We’ll start with the straightforward, naïve approach, then improve it
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
  - Idea 1: One variable at a time
    - Variable assignments are commutative, so fix ordering
    - I.e., \([WA = \text{red then NT = green}]\) same as \([NT = \text{green then WA = red}]\)
    - Only need to consider assignments to a single variable at each step
  - Idea 2: Check constraints as you go
    - I.e. consider only values which do not conflict with previous assignments
    - Might have to do some computation to check the constraints
    - “Incremental goal test”
  - Depth-first search with these two improvements is called *backtracking search* (not the best name)
  - Can solve \(n\)-queens for \(n \approx 25\)
Backtracking Search

function \textsc{Backtracking-Search}(csp) returns solution/failure
return \textsc{Recursive-Backtracking}({}, csp)

function \textsc{Recursive-Backtracking}(assignment, csp) returns soln/failure
if assignment is complete then return assignment
var ← \textsc{Select-Unassigned-Variable}(\textsc{Variables}[csp], assignment, csp)
for each value in \textsc{Order-Domain-Values}(var, assignment, csp) do
if value is consistent with assignment given \textsc{Constraints}[csp] then
add \{\texttt{var} = \texttt{value}\} to assignment
result ← \textsc{Recursive-Backtracking}(assignment, csp)
if result \neq failure then return result
remove \{\texttt{var} = \texttt{value}\} from assignment
return failure

- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

[Demo: coloring -- backtracking]
Improving Backtracking

- 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?
Improving Backtracking

From https://kti.mff.cuni.cz/~bartak/constraints/propagation.html
Filtering
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
Video of Demo Coloring – Backtracking with Forward Checking
Filtering: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:
  - NT and SA cannot both be blue!
  - Why didn’t we detect this yet?
  - *Constraint propagation*: reason from constraint to constraint
Consistency of A Single Arc

- 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.
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?

Remember: Delete from the tail!
Enforcing Arc Consistency in a CSP

function AC-3(csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables \{X_1, X_2, \ldots, X_n\}
   local variables: queue, a queue of arcs, initially all the arcs in csp
   while queue is not empty do
      (X_i, X_j) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
         for each X_k in Neighbors[X_i] do
            add (X_k, X_i) to queue
   return csp

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[X_i] do
      if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ← X_j
         then delete x from DOMAIN[X_i]; removed ← true
   return removed

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

[Demo: CSP applet (made available by aispace.org) -- n-queens]
Limitations of Arc Consistency

- 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!

What went wrong here?

[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]
Ordering
Variable Ordering: Minimum remaining values (MRV):
- Choose the variable with the fewest legal left values in its domain

Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering
Value Ordering: Least Constraining Value

- Given a choice of variable, choose the least constraining value
- I.e., 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)

Why least rather than most?

Combining these ordering ideas makes 1000 queens feasible

[Demo: coloring – backtracking + AC + ordering]