CS W4701
Artificial Intelligence

Fall 2013
Chapter 6:
Constraint Satisfaction Problems

Jonathan Voris
(based on slides by Sal Stolfo)
Assignment 3

• Go
  – “Encircling Game”

• Ancient Chinese game
  – Dates back
    • At least to the 4\textsuperscript{th} century B.C.
    • Probably to 2300 B.C.
  – Abstraction of war, or princely distraction?

• Spread to Japan by 1000 A.D.

• Immigrants brought to America in the 1800s

• German mathematician Otto Korschelt began analyzing it in the early 20\textsuperscript{th} century
Assignment 3

• Go gameplay
  – 19 x 19 board
  – Players take turns placing black and white stones
  – Stones are removed if surrounded by the other player’s stones

• No set end condition
  – Game ends when both players pass
  – Winner has the most stones and controlled territory
Assignment 3

• Your objective is to develop a Go player agent
Assignment 3

• Your objective is to develop a Go player agent
• Any questions?
Assignment 3

- Your objective is to develop a Go player agent
- I’M KIDDING
Assignment 3

• Go is a rare example of a game that is harder for computers than humans
  – Only recently a computer beat a human with a 9 move handicap!
• Tons of possible moves
• Extremely sequential
  – Impact of moves on future states potentially limitless
• Tricky to evaluate
• For more see
Assignment 3

• Instead, you’ll be working with Gomoku
  – Aka Gobang
  – Aka Five in a Row
• Same board and stones as Go
• Win condition:
  Precisely 5 in a row
Assignment 3

• Assignment overview:
  – Implement Gomoku playing agent using Minimax & Alpha-Beta Pruning
  – Input:
    • Board size
    • Winning chain length
    • Move time limit

• 3 game modes:
  – Play human
  – Play random
  – Play self
Assignment 3

• Due in 2.5 weeks
  – Tuesday November 19\textsuperscript{th} @ 11:59:59 PM EST

• Please follow submission instructions

• Submit:
  – Code File
  – Documentation File

• Submissions should run on GNU/Linux CLIC machines
Outline

• Constraint Satisfaction Problems (CSP)
• Backtracking search for CSPs
• Local search for CSPs
Constraint Satisfaction Problems (CSPs)

- **Standard search problem:**
  - **Atomic** state representation
  - **state** is a "black box" – any data structure that supports successor function, heuristic function, and goal test

- **CSP:**
  - **Factored** state representation
  - **state** is defined by variables $X_i$ with values from domain $D_i$
  - **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
Constraint Satisfaction Problems (CSPs)

• Why CSPs?
• Natural way to formulate many problems
• Easier to apply existing CSP solver
• More efficient
  – Can greatly reduce size of search space
Example: Map Coloring

• **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 ≠ NT, or (WA,NT) in \{ (red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green) \}
Example: Map Coloring

- **Solutions** are *complete* and *consistent* assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green
Constraint Graph

- **Binary CSP**: each constraint relates two variables
- **Constraint graph**: nodes are variables, arcs are constraints
Varieties of CSPs

- **Discrete variables**
  - finite domains:
    - $n$ variables, domain size $d \rightarrow O(d^n)$ complete assignments
    - e.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $\text{StartJob}_1 + 5 \leq \text{StartJob}_3$

- **Continuous variables**
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming
Varieties of Constraints

- **Unary** constraints involve a single variable
  - e.g., SA ≠ green
- **Binary** constraints involve pairs of variables
  - e.g., SA ≠ WA
- **Higher-order** constraints involve 3 or more variables
  - Aka global constraints
  - Alldiff
  - Sudoku
  - Cryptarithmetic column constraints
Example: Cryptarithmetic

- **Variables:** \( F, T, U, W \)  
- **Domains:** \( \{0,1,2,3,4,5,6,7,8,9\} \)  
- **Constraints:** \( \text{Alldiff} \ (F,T,U,W,R,O) \)  
  - \( O + O = R + 10 \cdot X_1 \)  
  - \( X_1 + W + W = U + 10 \cdot X_2 \)  
  - \( X_2 + T + T = O + 10 \cdot X_3 \)  
  - \( X_3 = F, \ T \neq 0, \ F \neq 0 \)
Real World CSPs

• Assignment problems
  – e.g., Who teaches what class?
• Timetabling problems
  – e.g., Which class is offered when and where?
• Transportation scheduling
• Factory scheduling
• Notice that many real-world problems involve real-valued variables
Standard Search Formulation (Incremental)

Let's start with the straightforward approach, then fix it.

States are defined by the values assigned so far.

- **Initial state**: the empty assignment \{ \}
- **Successor function**: assign a value to an unassigned variable that does not conflict with current assignment
  \(\rightarrow\) fail if no legal assignments
- **Goal test**: the current assignment is complete

1. This is the same for all CSPs
2. Every solution appears at depth \(n\) with \(n\) variables
  \(\rightarrow\) use *depth-limited search*
3. Path is irrelevant, so can also use complete-state formulation
4. \(b = (n - \ell)d\) at depth \(\ell\), hence \(n! \cdot d^n\) leaves
Standard Search Formulation (Incremental)

• n variables
• domain size d
Standard Search Formulation (Incremental)

- n variables
- domain size d
- Branching factor:
Standard Search Formulation (Incremental)

• $n$ variables
• domain size $d$
• Branching factor:
  – $(n-1)d$
Standard Search Formulation (Incremental)

- n variables
- domain size d
- Branching factor:
  - \((n-1)d\)
- So number of leaves:
Standard Search Formulation (Incremental)

- $n$ variables
- domain size $d$
- Branching factor:
  - $(n-1)d$
- So number of leaves:
  - $n!d^n$
Standard Search Formulation (Incremental)

- $n$ variables
- domain size $d$
- Branching factor:
  - $(n-1)d$
- So number of leaves:
  - $n! \times d^n$
- But there are only $d^n$ complete assignments!
  - What went wrong?
Backtracking Search

- Variable assignments are *commutative*, i.e.,
  \[ \text{[ WA = red then NT = green ] same as [ NT = green then WA = red ]} \]
  Assignment order is irrelevant

- Only need to consider assignments to a single variable at each node
  \[ \rightarrow b = d \text{ and there are } d^n \text{ leaves} \]

- Depth-first search for CSPs with single-variable assignments is called *backtracking* search

- Backtracking search is the basic uninformed algorithm for CSPs

- Can solve *n*-queens for \( n \approx 25 \)
Backtracking Search

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

function RECURSIVE-BACKTRACKING( assignment, csp) returns a solution, or 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 according to 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 Example
Backtracking Example
Backtracking Example
Backtracking Example
Improving Backtracking Efficiency

- **General-purpose** heuristic methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can inferences be made along the way?
  - Can we detect inevitable failure early?
Most Constrained Variable

• Most constrained variable: choose the variable with the fewest legal values

• a.k.a. minimum remaining values (MRV) heuristic

• “Fail first”
  – Picking the variable most likely to cause a conflict
Most Constraining Variable

• Tie-breaker among most constrained variables

• Most constraining variable:
  – choose the variable with the most constraints on remaining variables

• aka degree heuristic
Least Constraining Value

• Given a variable, choose the least constraining value:
  – the one that rules out the fewest values in the remaining variables

• “Fail last”
  – Selecting value least likely to cause future conflicts
Improving Backtracking Efficiency

• Why fail first when selecting variables?
  – Prunes large portions of tree early on

• Why fail last when selecting values?
  – Only need one solution, so examine probable values first
Forward Checking

• Idea:
  – Keep track of remaining legal values for unassigned variables
  – Terminate search when any variable has no legal values
Forward Checking

• Idea:
  – Keep track of remaining legal values for unassigned variables
  – Terminate search when any variable has no legal values
Forward Checking

• **Idea:**
  – Keep track of remaining legal values for unassigned variables
  – Terminate search when any variable has no legal values
Forward Checking

• **Idea:**
  – Keep track of remaining legal values for unassigned variables
  – Terminate search when any variable has no legal values
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!

- **Constraint propagation** repeatedly enforces constraints locally
Arc Consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
  for every value $x$ of $X$ there is some allowed $y$
Arc Consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
  for every value $x$ of $X$ there is some allowed $y$
Arc Consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff
  for every value $x$ of $X$ there is some allowed $y$

- If $X$ loses a value, neighbors of $X$ need to be rechecked
Arc Consistency

• Simplest form of propagation makes each arc consistent
• $X \rightarrow Y$ is consistent iff
  
  for every value $x$ of $X$ there is some allowed $y$

• If $X$ loses a value, neighbors of $X$ need to be rechecked
• Arc consistency detects failure earlier than forward checking
  – Like forward checking, but recursively applies constraints
• Can be run as a preprocessor or after each assignment
Arc Consistency Algorithm AC-3

```plaintext
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 RM-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue

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

- Time complexity: $O(n^2d^3)$
Local Search for CSPs

- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned.
- To apply to CSPs:
  - Allow states with unsatisfied constraints
  - Operators reassign variable values
- Variable selection: randomly select any conflicted variable
- Value selection by **min-conflicts** heuristic:
  - Choose value that violates the fewest constraints
  - i.e., hill-climb with $h(n) = \text{total number of violated constraints}$
Example: 4-Queens

- **States**: 4 queens in 4 columns \((4^4 = 256 \text{ states})\)
- **Actions**: move queen in column
- **Goal test**: no attacks
- **Evaluation**: \(h(n) = \text{number of attacks}\)

![Diagram showing the transition from an initial state with 5 attacks to a final state with 0 attacks](image)

- Given random initial state, can solve \(n\)-queens in almost **constant** time for arbitrary \(n\) with high probability (e.g., \(n = 10,000,000\))
  - 50 steps on average
Example: 8-queens
Summary

• CSPs are a special kind of problem:
  – States are factored; defined by values of a fixed set of variables
  – Goal test defined by constraints on variable values

• Backtracking
  – Depth-first search with one variable assigned per node

• Variable ordering and value selection heuristics help significantly

• Forward checking prevents assignments that guarantee later failure

• Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies

• Iterative min-conflicts is usually effective in practice