Constraint Satisfaction Problems (CSPs)
Constraint Propagation and Local Search

This lecture topic (two lectures)
Chapter 6.1 – 6.4, except 6.3.3

Next lecture topic (two lectures)
Chapter 7.1 – 7.5

(Please read lecture topic material before and after each lecture on that topic)
You Will Be Expected to Know

• Node consistency, arc consistency, path consistency, K-consistency (6.2)

• Forward checking (6.3.2)

• Local search for CSPs
  – Min-Conflict Heuristic (6.4)

• The structure of problems (6.5)
Improving CSP efficiency

• Previous improvements on uninformed search
  → introduce heuristics

• For CSPS, general-purpose methods can give large gains in speed, e.g.,
  – Which variable should be assigned next?
  – In what order should its values be tried?
  – Can we detect inevitable failure early?
  – Can we take advantage of problem structure?

Note: CSPs are somewhat generic in their formulation, and so the heuristics are more general compared to methods in Chapter 4
Backtracking search (Figure 6.5)

function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING(∅, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return 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

Two heuristic opportunities:
(1) Which unassigned variable to explore next?
(2) In which order to explore its domain values?
Backtracking search

function BACKTRACKING-SEARCH(csp) return a solution or failure
return RECURSIVE-BACKTRACKING(Ø, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return 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 (DH)

Reduce the search branching factor!!
(1) Now --- Minimum Remaining Values (MRV) heuristic.
(2) In the future --- Degree Heuristic.
Minimum Remaining Values (MRV)  
For variable selection

• Also known as: Most Constrained Variable heuristic

• **Heuristic Rule**: choose variable with fewest legal moves
  – I.e., with the fewest number of legal values in its domain
  – E.g., will immediately detect failure if X has no legal values

Reduce the search branching factor now!!
The variable with the minimum remaining values in its domain has the minimum search branching factor now.
Detailed Example of MRV
Detailed Example of MRV

• Initially, all regions have MRV = 3
• Choose one randomly (e.g., WA=red)
• Do Forward Checking (next topic)
Detailed Example of MRV
Detailed Example of MRV

- Next, NT and SA both have MRV = 2
- Choose one randomly (e.g., NT=green)
- Do Forward Checking (next topic)

AC-3 could solve the CSP now, but Forward Checking is too simple to see it. The point here is only to illustrate MRV.
Detailed Example of MRV
Detailed Example of MRV

- Next, SA has MRV = 1 (must be blue)
- Choose it and assign it (i.e., SA=blue)
- Do Forward Checking (next topic)
Detailed Example of MRV
Degree Heuristic (DH)

(Caveat for tests, quizzes, and exams)

• This example works DH as a tie-breaker for MRV, which is its usual role in practice.

• On tests, you may be asked to work DH ignoring MRV, to prove that you know it.

• DH seeks the maximum degree in the constraint graph to unassigned nodes.
  – Whether or not it is a tie-breaker for MRV.
Degree heuristic (DH)
For variable selection (often a tie-breaker after MRV)

• Heuristic Rule: select variable that is involved in the largest number of constraints with other unassigned variables.

• Degree heuristic very often used as a tie-breaker among variables that have equal Minimum Remaining Values.

Reduce the search branching factor in the future!!
The variable with the largest number of constraints to other unassigned variables is most likely to knock out the most values from domains of other variables, thereby most reducing their search branching factor in the future.
Detailed Example of DH

(As a tie-breaker after MRV)
Detailed Example of DH (As a tie-breaker after MRV)

- Initially, all regions have MRV = 3
- SA has DH=5 (e.g., SA=blue)
- Do Forward Checking (next topic)
Detailed Example of DH
(As a tie-breaker after MRV)
Detailed Example of DH
(As a tie-breaker after MRV)

- NT, Q, and NSW have MRV = 2, DH = 2
- Choose one **randomly** (e.g., NT=green)
- Do Forward Checking (next topic)

AC-3 could solve the CSP now, but Forward Checking is too simple to see it. The point here is only to illustrate DH.
Detailed Example of DH
(As a tie-breaker after MRV)
Detailed Example of DH
(As a tie-breaker after MRV)

- WA and Q have MRV = 1, Q has DH = 1
- Choose Q and assign it (i.e., Q=red)
- Do Forward Checking (next topic)
Detailed Example of DH

(As a tie-breaker after MRV)
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return 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

In which order to explore its domain values? => Aim for early success!!
* If we are going to fail at this node, domain value ordering doesn’t matter.
  --- We have to grind out all possible domain values anyway.
* If we are going to succeed at this node, we want to succeed EARLY!
  => Put first the domain values most likely to succeed.
  --- The least-constraining domain values give the most hope of success.
Least constraining value
For domain value ordering

- Least constraining value heuristic

- Heuristic Rule: given a variable choose the least constraining value
  - leaves the maximum flexibility for subsequent variable assignments

The least-constraining domain values give the most hope of early success.
CONSTRAINT PROPAGATION

• OK, we’ve selected a variable to explore
  – We’re enumerating its domain values.
  – Backtracking search
    • Assign a value, and see what happens

• Other current domain values are not consistent with the value just assigned

• How can we reduce our search space??
Forward checking
Check only neighbors; delete any inconsistent values

- Can we detect inevitable failure early?
  - And avoid it later?

- **Forward checking idea:**
  - Keep track of legal values for neighbor’s unassigned variables.

- Terminate search when any variable has no legal values.
Forward checking

Check only neighbors; delete any inconsistent values

- Assign \{WA=red\}

- Effects on other variables connected by constraints to WA
  - \textit{NT can no longer be red}
  - \textit{SA can no longer be red}
Forward checking
Check only neighbors; delete any inconsistent values

- Assign \( Q = \text{green} \)

- Effects on other variables connected by constraints with WA
  - \( NT \) can no longer be green
  - \( NSW \) can no longer be green
  - \( SA \) can no longer be green

- \( MRV \) heuristic would automatically select NT or SA next

We already have failure, but Forward Checking is too simple to detect it now.
Forward checking

Check only neighbors; delete any inconsistent values

- If \( V \) is assigned \textit{blue}

- Effects on other variables connected by constraints with WA
  - NSW can no longer be blue
  - SA is empty

- FC has detected that partial assignment is \textit{inconsistent} with the constraints and backtracking can occur.
Example: 4-Queens Problem

Backtracking Search with Forward Checking

Book-keeping is tricky and complicated
Example: 4-Queens Problem

Red = value is assigned to variable
Example: 4-Queens Problem

Red = value is assigned to variable
Example: 4-Queens Problem

• X1 Level:
  – Deleted:
    • \{(X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4)\}

• (Please note: As always in computer science, there are many different ways to implement anything. The book-keeping method shown here was chosen because it is easy to present and understand visually. It is not necessarily the most efficient way to implement the book-keeping in a computer. Your job as an algorithm designer is to think long and hard about your problem, then devise an efficient implementation.)

• One possibly more efficient equivalent alternative (of many):
  – Deleted:
    • \{(X2:1,2) (X3:1,3) (X4:1,4)\}
Example: 4-Queens Problem

X1
{1,2,3,4}

X2
{ , ,3,4}

X3
{ ,2, ,4}

X4
{ ,2,3, }

Red = value is assigned to variable
Example: 4-Queens Problem

Red = value is assigned to variable
Example: 4-Queens Problem

Red = value is assigned to variable
Example: 4-Queens Problem

• X1 Level:
  - Deleted:
    • { (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) }

• X2 Level:
  - Deleted:
    • { (X3,2) (X3,4) (X4,3) }

(Please note: Of course, we could have failed as soon as we deleted { (X3,2) (X3,4) }. There was no need to continue to delete (X4,3), because we already had established that the domain of X3 was null, and so we already knew that this branch was futile and we were going to fail anyway. The book-keeping method shown here was chosen because it is easy to present and understand visually. It is not necessarily the most efficient way to implement the book-keeping in a computer. Your job as an algorithm designer is to think long and hard about your problem, then devise an efficient implementation.)
Example: 4-Queens Problem

Red = value is assigned to variable
Example: 4-Queens Problem

• X1 Level:
  – Deleted:
    • \{ (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) \}

• X2 Level:
  – FAIL at X2=3.
  – Restore:
    • \{ (X3,2) (X3,4) (X4,3) \}
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

• X1 Level:
  – Deleted:
    • \{ (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) \}

• X2 Level:
  – Deleted:
    • \{ (X3,4) (X4,2) \}
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

• X1 Level:
  – Deleted:
    • \{ (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) \}

• X2 Level:
  – Deleted:
    • \{ (X3,4) (X4,2) \}

• X3 Level:
  – Deleted:
    • \{ (X4,3) \}
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

• X1 Level:
  – Deleted:
    • \{ (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) \}

• X2 Level:
  – Deleted:
    • \{ (X3,4) (X4,2) \}

• X3 Level:
  – Fail at X3=2.
  – Restore:
    • \{ (X4,3) \}
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

• X1 Level:
  – Deleted:
    • \{ (X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) \}

• X2 Level:
  – Fail at X2=4.
  – Restore:
    • \{ (X3,4) (X4,2) \}
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

• X1 Level:
  – Fail at X1=1.
  – Restore:
    • \{(X2,1) (X2,2) (X3,1) (X3,3) (X4,1) (X4,4) \}
Example: 4-Queens Problem

<table>
<thead>
<tr>
<th></th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>X4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

• X1 Level:
  – Deleted:
    • \{ (X2,1) (X2,2) (X2,3) (X3,2) (X3,4) (X4,2) \}
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

• X1 Level:
  – Deleted:
    • \{ (X2,1) (X2,2) (X2,3) (X3,2) (X3,4) (X4,2) \}

• X2 Level:
  – Deleted:
    • \{ (X3,3) (X4,4) \}
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

• X1 Level:
  – Deleted:
    • \{ (X2,1) (X2,2) (X2,3) (X3,2) (X3,4) (X4,2) \}

• X2 Level:
  – Deleted:
    • \{ (X3,3) (X4,4) \}

• X3 Level:
  – Deleted:
    • \{ (X4,1) \}
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Example: 4-Queens Problem

Red = value is assigned to variable
X = value led to failure
Solving CSPs with combination of heuristics plus forward checking is more efficient than either approach alone.

Forward Checking does not detect all failures when they become obvious.
- E.g., NT and SA cannot both be blue in the example above, so failure.

Higher-order Constraint Propagation can detect early failure.
- However, to do so takes more computing time --- is it worth the extra effort??
Constraint propagation

• Techniques like Constraint Propagation and Forward Checking, in effect, work by eliminating parts of the search space
  – Somewhat complementary to search

• Constraint Propagation goes further than Forward Checking by repeatedly enforcing constraints locally
  – Needs to be faster than actually searching to be worthwhile and effective
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

- An Arc $X \rightarrow Y$ is consistent if for every value $x$ of $X$ there is some value $y$ of $Y$ consistent with $x$ *(note that this is a directed property)*

- Consider state of search after WA and Q are assigned & FC is done:

  $SA \rightarrow NSW$ is consistent if $SA=blue$ and $NSW=red$
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

• $X \rightarrow Y$ is consistent if
  for every value $x$ of $X$ there is some value $y$ of $Y$ consistent with $x$

• $NSW \rightarrow SA$ is consistent if
  $NSW=red$ and $SA=blue$
  $NSW=blue$ and $SA=???

• **NSW=blue can be pruned:** No current domain value of $SA$ is consistent
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

- Enforce arc-consistency:
  Arc can be made consistent by removing *blue* from *NSW*

- Continue to propagate constraints….
  - Check $V \rightarrow NSW$
  - Not consistent for $V = \text{red}$
  - Remove red from $V$
Arc consistency (AC-3)
Like Forward Checking, but exhaustive until quiescence

Continue to propagate constraints….

- **SA → NT** is not consistent
  - and cannot be made consistent (Failure!)

- Arc consistency detects failure earlier than FC
  - Requires more computation: Is it worth the effort??
Arc consistency checking

• Can be run as a preprocessor, or after each assignment
  – As preprocessor before search: Removes obvious inconsistencies
  – After each assignment: Reduces search cost but increases step cost

• AC must be run repeatedly until no inconsistency remains
  – Like Forward Checking, but exhaustive until quiescence

• Trade-off
  – Requires overhead to do; but usually better than direct search
  – In effect, it can successfully eliminate large (and inconsistent) parts of the state space more effectively than can direct search alone

• Need a systematic method for arc-checking
  – If X loses a value, neighbors of X need to be rechecked:
    I.e., incoming arcs can become inconsistent again (outgoing arcs will stay consistent).
Arc consistency algorithm (AC-3)

**function** AC-3(csp) **returns** false if inconsistency found, else true, may reduce csp domains

**inputs:** csp, a binary CSP with variables \{X_1, X_2, ..., X_n\}

**local variables:** queue, a queue of arcs, initially all the arcs in csp

/* initial queue must contain both \((X_i, X_j)\) and \((X_j, X_i)\) */

**while** queue is not empty **do**

\((X_i, X_j) \leftarrow \text{REMOVE-FIRST}(queue)\)

**if** REMOVE-INCONSISTENT-VALUES\((X_i, X_j)\) **then**

**if** size of \(D_i = 0\) **then** return false

**for each** \(X_k\) in NEIGHBORS\([X_i] - \{X_j\}\) **do**

add \((X_k, X_i)\) to queue if not already there

return true

**function** REMOVE-INCONSISTENT-VALUES\((X_i, X_j)\) **returns** true iff we delete a value from the domain of \(X_i\)

\(removed \leftarrow false\)

**for each** \(x\) in \(\text{DOMAIN}[X_i]\) **do**

**if** no value \(y\) in \(\text{DOMAIN}[X_j]\) allows \((x, y)\) to satisfy the constraints between \(X_i\) and \(X_j\)

then delete \(x\) from \(\text{DOMAIN}[X_i]\); \(removed \leftarrow true\)

return \(removed\)

(from Mackworth, 1977)
Complexity of AC-3

- A binary CSP has at most $n^2$ arcs

- Each arc can be inserted in the queue $d$ times (worst case)
  - $(X, Y)$: only $d$ values of $X$ to delete

- Consistency of an arc can be checked in $O(d^2)$ time

- Complexity is $O(n^2 \cdot d^3)$

- Although substantially more expensive than Forward Checking, Arc Consistency is usually worthwhile.
K-consistency

• Arc consistency does not detect all inconsistencies:
  – Partial assignment \{WA=red, NSW=red\} is inconsistent.

• Stronger forms of propagation can be defined using the notion of k-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 kth variable.
  – E.g. 1-consistency = node-consistency
  – E.g. 2-consistency = arc-consistency
  – E.g. 3-consistency = path-consistency

• Strongly k-consistent:
  – k-consistent for all values \{k, k-1, …2, 1\}
Trade-offs

• Running stronger consistency checks…
  – Takes more time
  – But will reduce branching factor and detect more inconsistent partial assignments
  – No “free lunch”
    • In worst case n-consistency takes exponential time

• Generally helpful to enforce 2-Consistency (Arc Consistency)

• Sometimes helpful to enforce 3-Consistency

• Higher levels may take more time to enforce than they save.
Further improvements

• Checking special constraints
  – Checking Alldif(…) constraint
    • *E.g. \{WA=red, NSW=red\}*
  – Checking Atmost(…) constraint
    • *Bounds propagation for larger value domains*

• Intelligent backtracking
  – Standard form is chronological backtracking i.e. try different value for preceding variable.
  – More intelligent, backtrack to conflict set.
    • Set of variables that caused the failure or set of previously assigned variables that are connected to X by constraints.
    • Backjumping moves back to most recent element of the conflict set.
    • Forward checking can be used to determine conflict set.
Local search for CSPs

- Use complete-state representation
  - Initial state = all variables assigned values
  - Successor states = change 1 (or more) values

- For CSPs
  - allow states with unsatisfied constraints (unlike backtracking)
  - operators **reassign** variable values
  - hill-climbing with n-queens is an example

- Variable selection: randomly select any conflicted variable

- Value selection: **min-conflicts heuristic**
  - Select new value that results in a minimum number of conflicts with the other variables
Local search for CSP

function MIN-CONFLICTS(csp, max_steps) return solution or failure

inputs: csp, a constraint satisfaction problem
        max_steps, the number of steps allowed before giving up

current ← an initial complete assignment for csp
for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen, conflicted variable from VARIABLES[csp]
    value ← the value v for var that minimize CONFLICTS(var,v,current,csp)
    set var = value in current
return failure
Min-conflicts example 1

Use of min-conflicts heuristic in hill-climbing.
Min-conflicts example 2

- A two-step solution for an 8-queens problem using min-conflicts heuristic
- At each stage a queen is chosen for reassignment in its column
- The algorithm moves the queen to the min-conflict square breaking ties randomly.
Comparison of CSP algorithms on different problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Backtracking</th>
<th>BT+MRV</th>
<th>Forward Checking</th>
<th>FC+MRV</th>
<th>Min-Conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>(&gt; 1,000K)</td>
<td>(&gt; 1,000K)</td>
<td>2K</td>
<td>60</td>
<td>64</td>
</tr>
<tr>
<td>n-Queens</td>
<td>(&gt; 40,000K)</td>
<td>13,500K</td>
<td>(&gt; 40,000K)</td>
<td>817K</td>
<td>4K</td>
</tr>
<tr>
<td>Zebra</td>
<td>3,859K</td>
<td>1K</td>
<td>35K</td>
<td>0.5K</td>
<td>2K</td>
</tr>
<tr>
<td>Random 1</td>
<td>415K</td>
<td>3K</td>
<td>26K</td>
<td>2K</td>
<td></td>
</tr>
<tr>
<td>Random 2</td>
<td>942K</td>
<td>27K</td>
<td>77K</td>
<td>15K</td>
<td></td>
</tr>
</tbody>
</table>

Median number of consistency checks over 5 runs to solve problem

Parentheses -> no solution found

USA: 4 coloring
n-queens: n = 2 to 50
Zebra: see exercise 6.7 (3rd ed.); exercise 5.13 (2nd ed.)
Advantages of local search

• Local search can be particularly useful in an online setting
  – Airline schedule example
    • E.g., mechanical problems require than 1 plane is taken out of service
    • Can locally search for another “close” solution in state-space
    • Much better (and faster) in practice than finding an entirely new schedule

• The runtime of min-conflicts is roughly independent of problem size.
  – Can solve the millions-queen problem in roughly 50 steps.

  – Why?
    • n-queens is easy for local search because of the relatively high density of solutions in state-space
Performance of min-conflicts

Given random initial state, can solve \( n \)-queens in almost constant time for arbitrary \( n \) with high probability (e.g., \( n = 10,000,000 \))

The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio

\[
R = \frac{\text{number of constraints}}{\text{number of variables}}
\]

CPU time

\( R \)

critical ratio
Hard satisfiability problems
Hard satisfiability problems

- Median runtime for 100 satisfiable random 3-CNF sentences, $n = 50$
Sudoku
Backtracking Search + Forward Checking

Avg Time vs. R

Success Rate vs. R

- $R = \frac{\text{number of cells}}{\text{number of filled cells}}$
- Success Rate $= P(\text{random puzzle is solvable})$
- [number of cells] = 81
- [number of filled cells] = variable
Graph structure and problem complexity

- Solving disconnected subproblems
  - Suppose each subproblem has $c$ variables out of a total of $n$.
    - Worst case solution cost is $O(n/c d^c)$, i.e. linear in $n$
      - Instead of $O(d^n)$, exponential in $n$

- E.g. $n= 80$, $c= 20$, $d=2$
  - $2^{80} = 4$ billion years at 1 million nodes/sec.
  - $4 \times 2^{20}= .4$ second at 1 million nodes/sec

- See R&N, section 6.5, for more in-depth discussion on this topic
Tree-structured CSPs

• Theorem:
  – if a constraint graph has no loops then:
    • the CSP can be solved in $O(nd^2)$ time
  – linear in the number of variables!

• Compare difference with general CSP, where worst case is $O(d^n)$
Summary

• CSPs
  – special kind of problem: states 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

• Heuristics
  – Variable ordering and value selection heuristics help significantly

• Constraint propagation does additional work to constrain values and detect inconsistencies
  – Works effectively when combined with heuristics

• Iterative min-conflicts is often effective in practice.

• Graph structure of CSPs determines problem complexity
  – e.g., tree structured CSPs can be solved in linear time.