CMSC 471
Fall 2012

Class #7

Thu 9/20/12
Constraint Satisfaction

Kevin Winner, winnerk1@umbc.edu

Constraint Satisfaction

Chapter 6

Today's Class

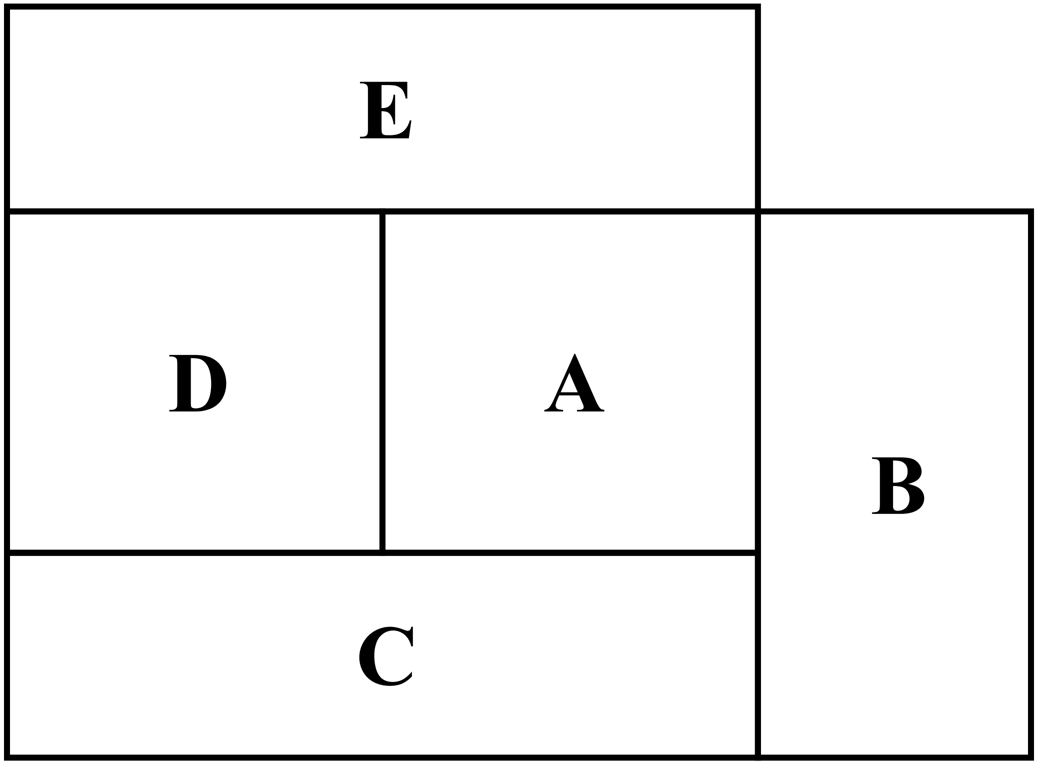
- Constraint Processing / Constraint Satisfaction Problem (CSP) paradigm
- Consistency
- Algorithms for CSPs
  - Backtracking (systematic search)
  - Constraint propagation (k-consistency)
  - Variable and value ordering heuristics
- HW2 rush-hour_basics.lisp

Overview

- Constraint satisfaction offers a powerful problem-solving paradigm
  - View a problem as a **set of variables** to which we have to assign **values** that satisfy a number of **problem-specific constraints**.
  - A **solution** to a CSP is then to find a set of **assignments** of a value to each variable which satisfies all of the constraints

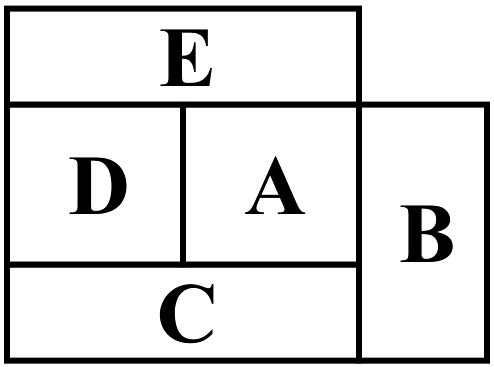
Informal Example: Map Coloring

- Color the following map using three colors (red, green, blue) such that no two adjacent regions have the same color.



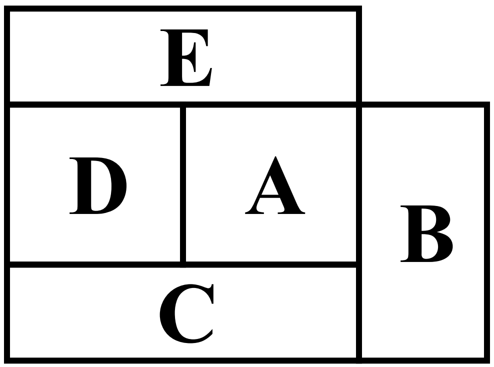
Map Coloring II

- Variables: A, B, C, D, E all of domain RGB
- Domains: RGB = {red, green, blue}
- Constraints: $A \neq B$, $A \neq C$, $A \neq E$, $A \neq D$, $B \neq C$, $C \neq D$, $D \neq E$
- One solution: A=red, B=green, C=blue, D=green, E=blue

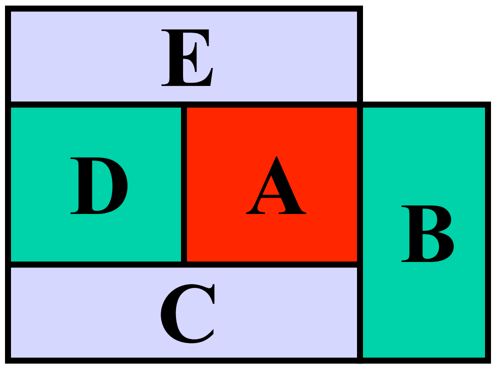


Map Coloring II

- Variables: A, B, C, D, E all of domain RGB
- Domains: RGB = {red, green, blue}
- Constraints: $A \neq B$, $A \neq C$, $A \neq E$, $A \neq D$, $B \neq C$, $C \neq D$, $D \neq E$
- One solution: A=red, B=green, C=blue, D=green, E=blue



=>



Formal Definition of a Constraint Network (CN)

A constraint network (CN) consists of

- a set of variables $X = \{x_1, x_2, \dots x_n\}$
  – each with an associated domain of values $\{d_1, d_2, \dots d_n\}$.
  – the domains are typically finite
- a set of constraints $\{c_1, c_2 \dots c_m\}$ where
  – each constraint defines a predicate which is a relation over a particular subset of X.
  – e.g., C_i involves variables $\{X_{i1}, X_{i2}, \dots X_{ik}\}$ and defines the relation $R_i \subseteq D_{i1} \times D_{i2} \times \dots D_{ik}$
- **Unary** constraint: only involves one variable
- **Binary** constraint: only involves two variables

Formal Definition of a CN (cont.)

- Instantiations
  - An **instantiation** of a subset of variables S is an assignment of a value in its domain to each variable in S
  - An instantiation is **legal** iff it does not violate any constraints.
- A **solution** is a legal instantiation of all of the variables in the network.

Typical Tasks for CSP

- Solutions:
  - Does a solution exist?
  - Find one solution
  - Find all solutions

Binary CSP

- A **binary CSP** is a CSP in which all of the constraints are binary or unary.
- A binary CSP can be represented as a **constraint graph**, which has a node for each variable and an arc between two nodes if and only there is a constraint involving the two variables.
  - Unary constraint appears as a self-referential arc

Example: Sudoku

	3		1
	1		4
3	4	1	2
		4	

Running Example: Sudoku

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}

- Variables and their domains
  - v_{ij} is the value in the jth cell of the ith row
  - $D_{ij} = D = \{1, 2, 3, 4\}$
- Blocks:
  - $B_1 = \{11, 12, 21, 22\}$
  - ...
  - $B_4 = \{33, 34, 43, 44\}$
- Constraints (implicit/intensional)
  - $C^R : \forall i, \ \cup_j v_{ij} = D$ (every value appears in every row)
  - $C^C : \forall j, \ \cup_j v_{ij} = D$ (every value appears in every column)
  - $C^B : \forall k, \ \cup (v_{ij} \mid ij \in B_k) = D$ (every value appears in every block)
  - Alternative representation: pairwise inequality constraints:
    - $I^R : \forall i, j \neq j' : v_{ij} \neq v_{ij'}$ (no value appears twice in any row)
    - $I^C : \forall j, i \neq i' : v_{ij} \neq v_{i'j}$ (no value appears twice in any column)
    - $I^B : \forall k, ij \in B_k, i'j' \in B_k, ij \neq i'j' : v_{ij} \neq v_{i'j'}$ (no value appears twice in any block)
  - Advantage of the second representation: all binary constraints!

Sudoku Constraint Network

Sudoku Constraint Network

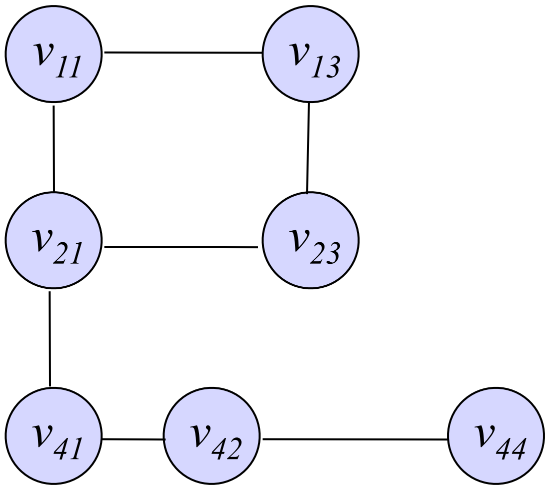
	3		1
	1		4
3	4	1	2
		4	

Sudoku Constraint Network

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}

Sudoku Constraint Network

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}



Consistency

- Node consistency
  - A node X is **node-consistent** if every value in the domain of X is consistent with X's unary constraints
  - A graph is node-consistent if all nodes are node-consistent
- Arc consistency
  - An arc (X, Y) is **arc-consistent** if, for every value x of X, there is a value y for Y that satisfies the constraint represented by the arc.
  - A graph is arc-consistent if all arcs are arc-consistent.
- To create arc consistency, we perform **constraint propagation**: that is, we repeatedly reduce the domain of each variable to be consistent with its arcs

K-consistency

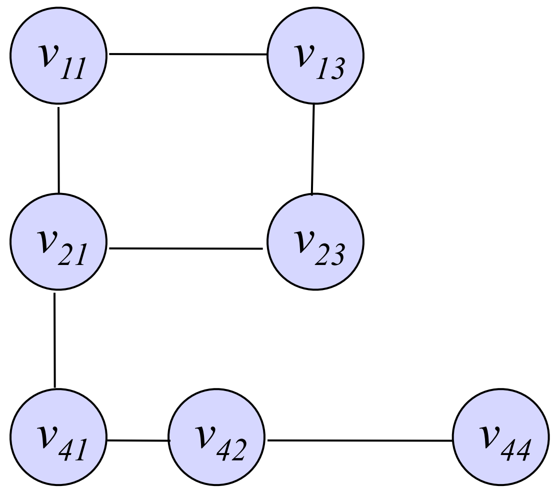
- K- consistency generalizes the notion of arc consistency to sets of more than two variables.
  - A graph is K-consistent if, for legal values of any K-1 variables in the graph, and for any Kth variable V_k, there is a legal value for V_k
- Strong K-consistency = J-consistency for all J<=K
- **Node consistency = strong 1-consistency**
- **Arc consistency = strong 2-consistency**
- Path consistency = strong 3-consistency

Constraint Propagation: Sudoku

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}

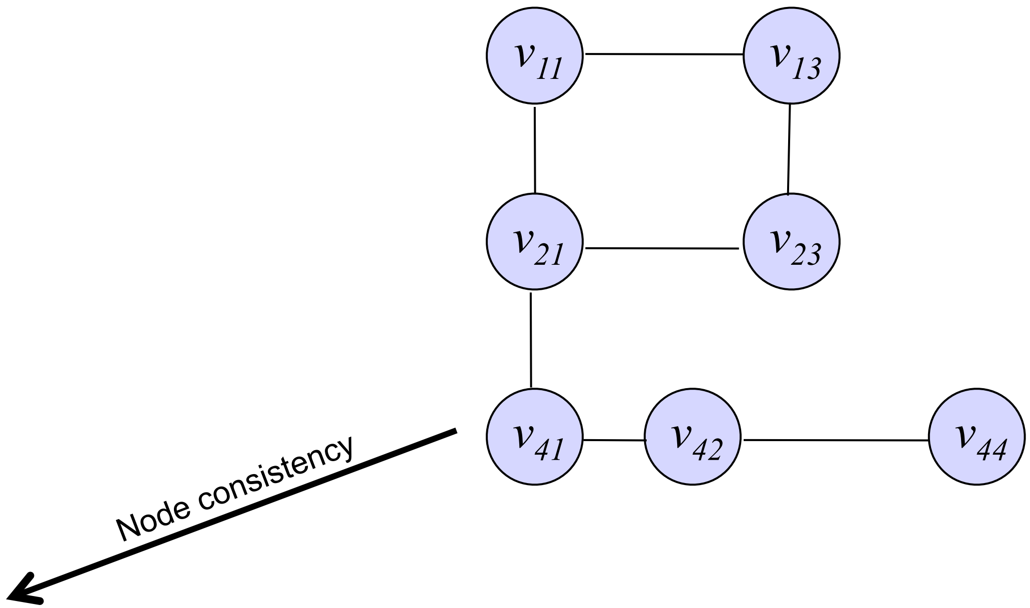
Constraint Propagation: Sudoku

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}



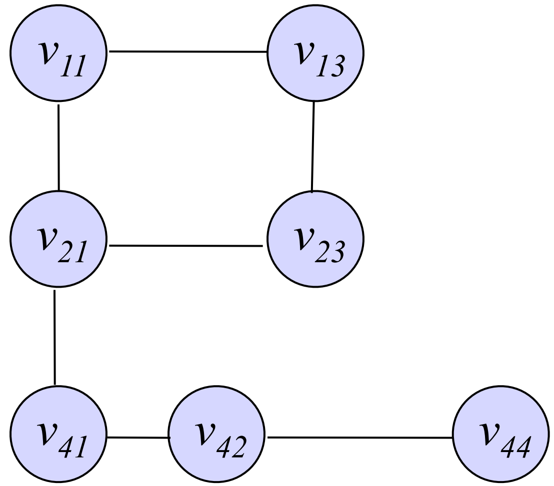
Constraint Propagation: Sudoku

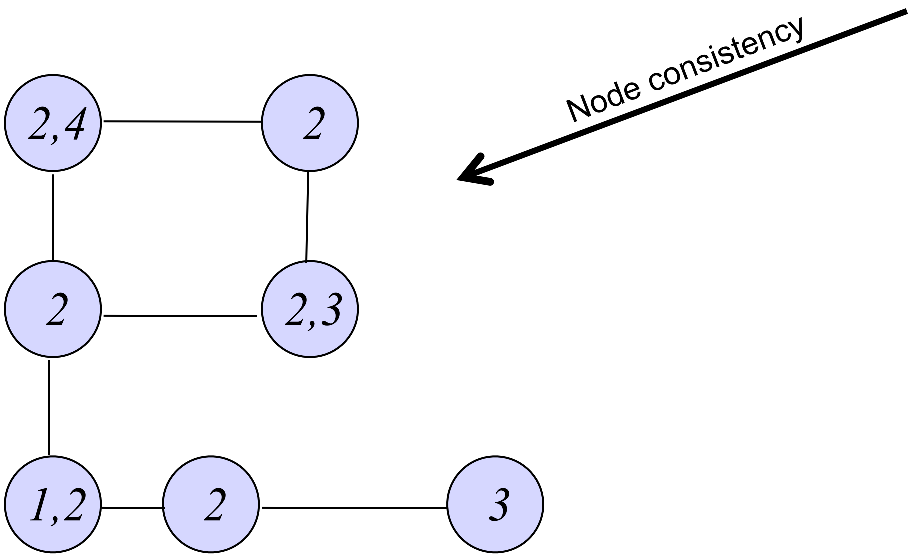
v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}



Constraint Propagation: Sudoku

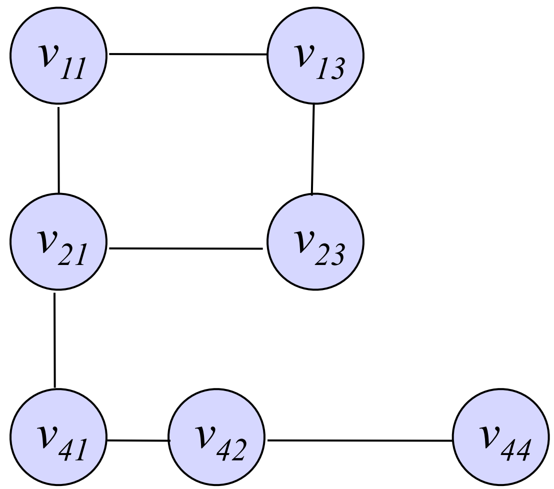
v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}

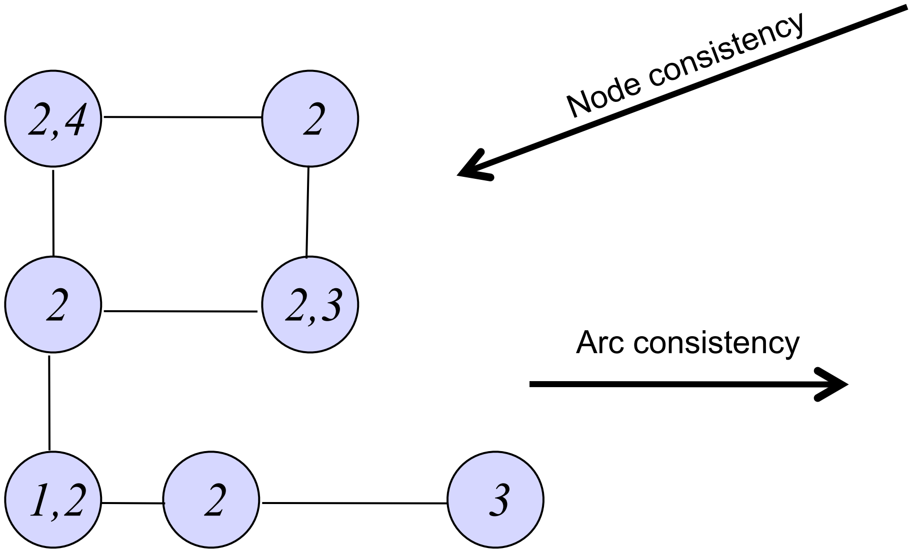




Constraint Propagation: Sudoku

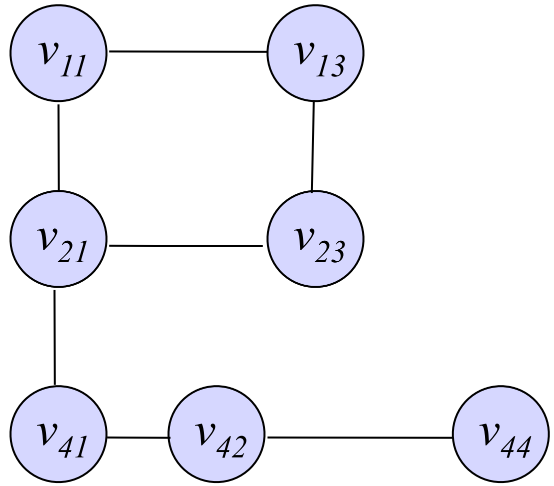
v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}

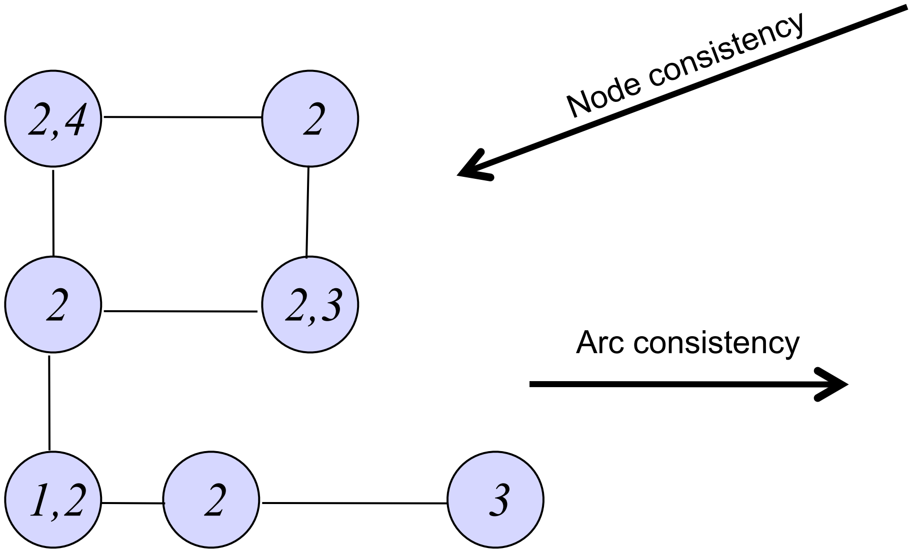


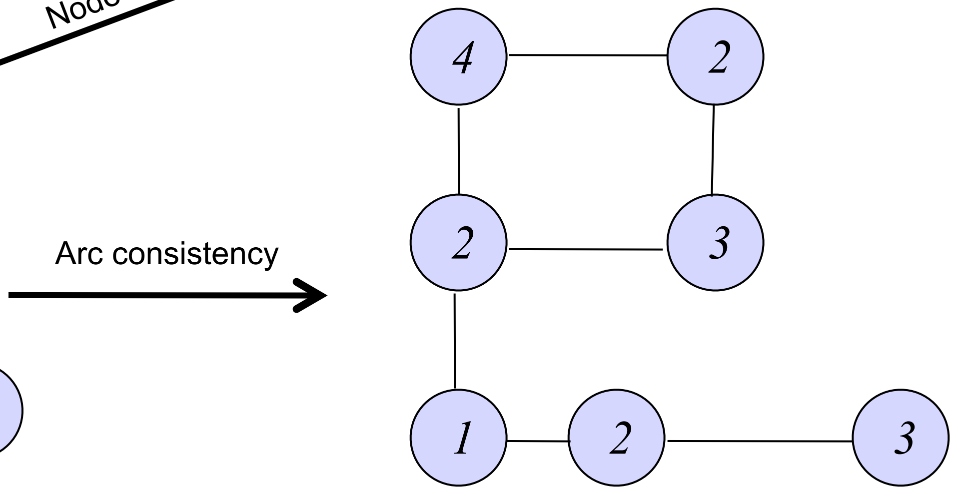


Constraint Propagation: Sudoku

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}

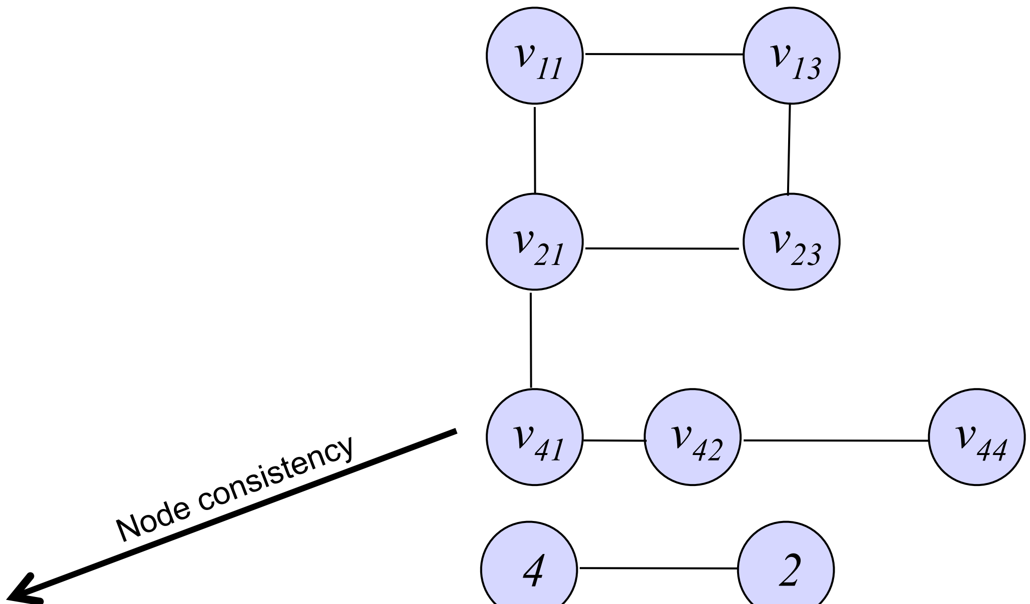


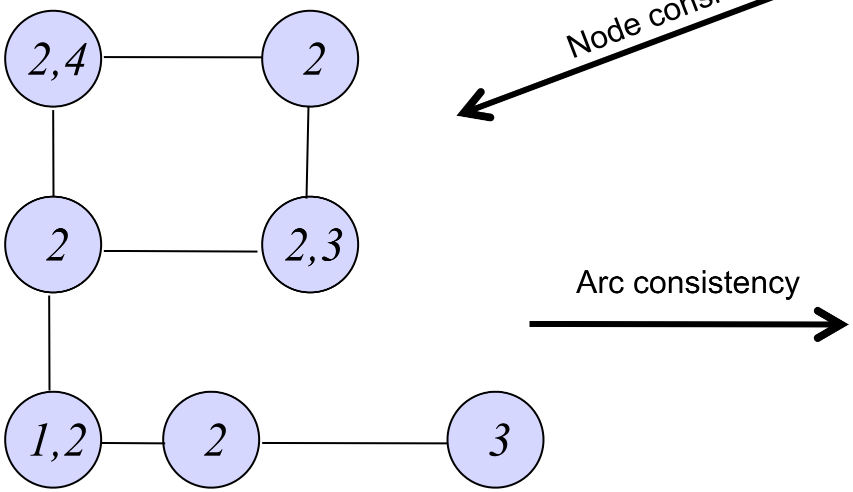


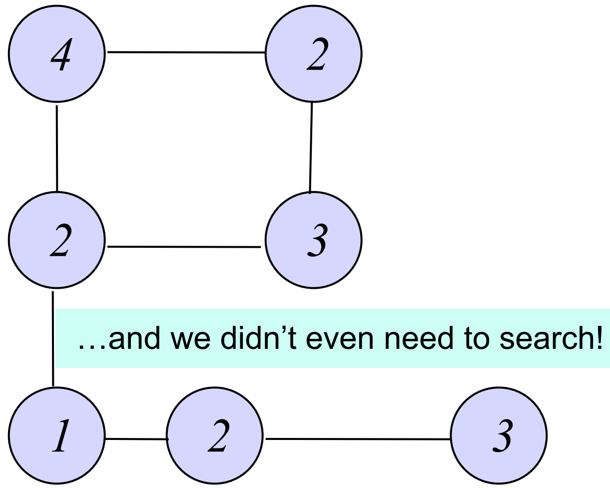


Constraint Propagation: Sudoku

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}







Solving Constraint Problems

- Systematic search
  - Generate and test
  - Backtracking
- Constraint propagation (consistency)
- Variable ordering heuristics
- Value ordering heuristics

Generate and Test: Sudoku

- Try each possible combination until you find one that works:

1	3	*1*	1
1	1	*1*	4
3	4	1	2
1	*1*	4	*1*

1	3	*1*	1
1	1	*1*	4
3	4	1	2
1	*1*	4	*2*

1	3	*1*	1
1	1	*1*	4
3	4	1	2
1	*1*	4	*3*

……

- Doesn’t check constraints until all variables have been instantiated
- Very inefficient way to explore the space of possibilities (4^7 for this trivial Sudoku puzzle, most illegal)

Systematic Search: Backtracking
(a.k.a. depth-first search!)

- Consider the variables in some order
- Pick an unassigned variable and give it a provisional value such that it is consistent with all of the constraints
- If no such assignment can be made, we've reached a dead end and need to backtrack to the previous variable
- Continue this process until a solution is found or we backtrack to the initial variable and have exhausted all possible values

Backtracking: Sudoku

Let's try it…

Backtracking: Sudoku

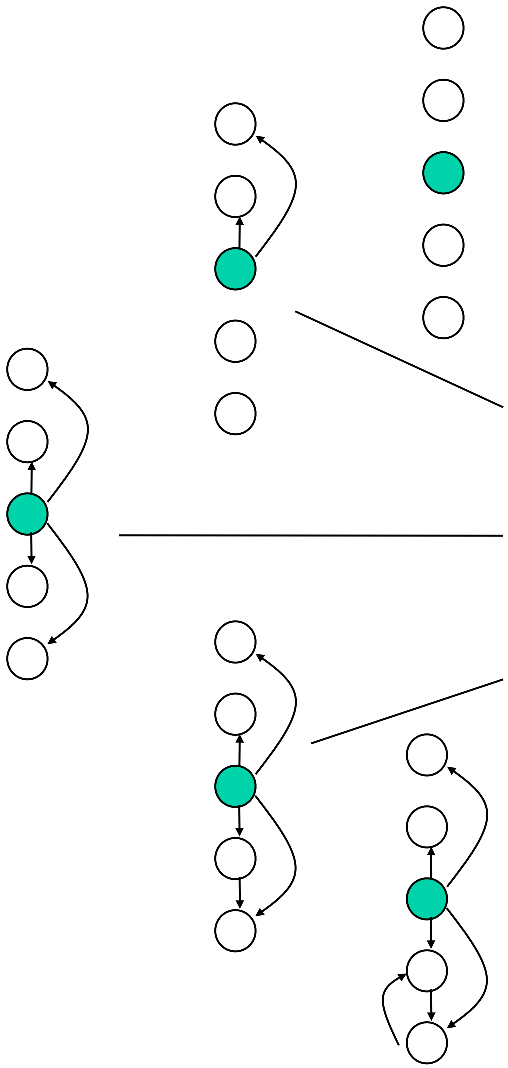
Let's try it…

v_{11}	3	v_{13}	1
v_{21}	1	v_{23}	4
3	4	1	2
v_{41}	v_{42}	4	v_{44}

Problems with Backtracking

- Thrashing: keep repeating the same failed variable assignments
  - Consistency checking can help
  - Intelligent backtracking schemes can also help
- Inefficiency: can explore areas of the search space that aren't likely to succeed
  - Variable ordering can help

Interleaving Constraint Propagation and Search



Generate and Test	No constraint propagation: assign all variable values, **then** test constraints
Simple Backtracking	Check constraints only for variables “up the tree”
Forward Checking	Check constraints for immediate neighbors “down the tree”
Partial Lookahead	Propagate constraints forward “down the tree”
Full Lookahead	Ensure complete arc consistency after each instantiation (AC-3)

Variable Ordering

- As defined, Backtracking Search selects variables to instantiate *randomly*
- **Intuition**: choose variables that are highly constrained early in the search process; leave easy ones for later

Variable Ordering

- **Fail first principle** (FFP): choose variable with the fewest values (a.k.a. **minimum remaining values** (MRV))
  - **Static** FFP: use domain size of variables
  - **Dynamic** FFP (**search rearrangement method**): At each point in the search, select the variable with the fewest remaining values
- **Maximum cardinality ordering**: order variables by decreasing cardinality

Value Ordering

- **Intuition**: Choose values that are the least constrained early on, leaving the most legal values in later variables

Value Ordering

- **Maximal options method** (a.k.a. **least-constraining-value** heuristic): Choose the value that leaves the most legal values in uninstantiated variables
- **Min-conflicts**: Used in iterative repair search (see below)

Iterative Repair

- Start with an initial complete (but invalid) assignment
- Hill climbing, simulated annealing
- Min-conflicts: Select new values that minimally conflict with the other variables
  – Use in conjunction with hill climbing or simulated annealing or…
- Local maxima strategies
  – Random restart
  – Random walk
  – Tabu search: don't try recently attempted values

Min-Conflicts Heuristic

- Iterative repair method
  1. Find some “reasonably good” initial solution
     - E.g., in N-queens problem, use greedy search through rows, putting each queen where it conflicts with the smallest number of previously placed queens, breaking ties *randomly*
  2. Find a variable in conflict (randomly)
  3. Select a new value that minimizes the number of constraint violations
     - O(N) time and space
  4. Repeat steps 2 and 3 until done
- Performance depends on quality and informativeness of initial assignment; inversely related to distance to solution