Title: Constraint Satisfaction Problems

Required reading: AIMA: Chapter 6

Recommended reading:

— Introduction to CSPs (Bartak's on-line guide)

Introduction to Artificial Intelligence

CSCE 476-876, Fall 2020

URL: `cse.unl.edu/~cse476` **URL:** `cse.unl.edu/~choueiry/F20-476-876`

Berthe Y. Choueiry (Shu-we-ri)

Constraint Processing

- Constraint Satisfaction:
  - Modeling and problem definition (Constraint Satisfaction Problem, CSP)
  - Algorithms for constraint propagation
  - Algorithms for search
- Constraint Programming: Languages and tools
  - logic-based
  - object-oriented
  - functional

Courses on Constraint Processing

http://cse.unl.edu/~choueiry/Constraint-Courses.html

- CSCE 421/821 Foundations of Constraint Processing
- CSCE 921 Advanced Constraint Processing

Outline

- Problem definition and examples
- Solution techniques: search and constraint propagation
- Exploiting the structure
- Research directions

What is this about?

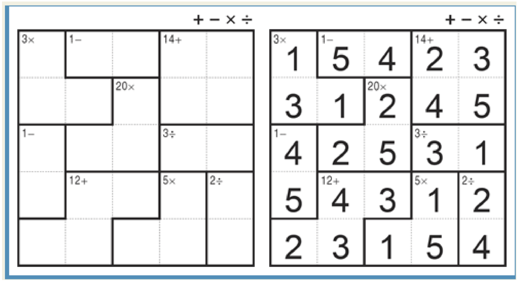
Context: Solving a Kendoku Puzzle

Problem: You need to assign numbers to unmarked cells

Possibilities: You can choose any number between 1 and 5

Constraints: restrict the choices you can make

> *Unary:* You have to respect predefined cells
>
> *Binary:* No two cells in same row or column have the same value
>
> *Global:* All the cells in each area must summ up to a given value.



You have choices, but are restricted by constraints

$\longrightarrow$ Make the right decisions

Constraint Satisfaction

Given

- A set of variables: 25 cells
- For each variable, a set of choices {1,2,3,4,5}
- A set of constraints that restrict the combinations of values the variables can take at the same time

Questions

- Does a solution exist? *classical decision problem*
- How two or more solutions differ? How to change specific choices without perturbing the solution?
- If there is no solution, what are the sources of conflicts? Which constraints should be retracted?
- *etc.*

Constraint Processing is about

- solving a decision problem
- while allowing the user to state <u>arbitrary</u> constraints in an expressive way and
- providing concise and high-level feedback about alternatives and conflicts

Power of Constraints Processing

- flexibility & expressiveness of representations
- interactivity, users can $\left\{ \begin{array}{c} \text{relax} \\ \text{reinforce} \end{array} \right\}$ constraints

Related areas: AI, OR, Algorithmic, DB, Prog. Languages, *etc.*

Definition

Given $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$:

- $\mathcal{V}$ a set of variables
  $\mathcal{V} = \{V_1, V_2, \ldots, V_n\}$
- $\mathcal{D}$ a set of variable domains (domain values)
  $\mathcal{D} = \{D_{V_1}, D_{V_2}, \ldots, D_{V_n}\}$
- $\mathcal{C}$ a set of constraints

$$C_{V_a,V_b,\ldots,V_i} = \{\,(x, y, \ldots, z)\} \subseteq D_{V_a} \times D_{V_b} \times \ldots \times D_{V_i}$$

Query: can we find one value for each variable
such that all constraints are satisfied?

In general, **NP-complete**

Terminology

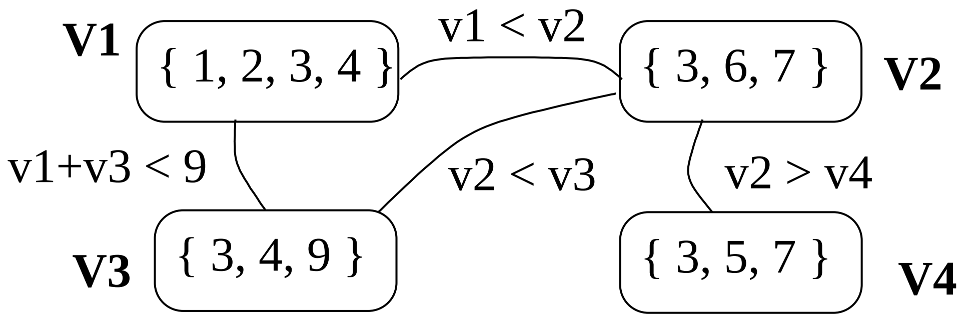
- Instantiating a variable: $V_i \leftarrow a$ where $a \in D_{V_i}$
- Variable-value pair (vvp)
- Partial assignment
- No good
- Constraint checking
- Consistent assignment
- Constrained optimization problem: Objective function

Representation: Constraint graph

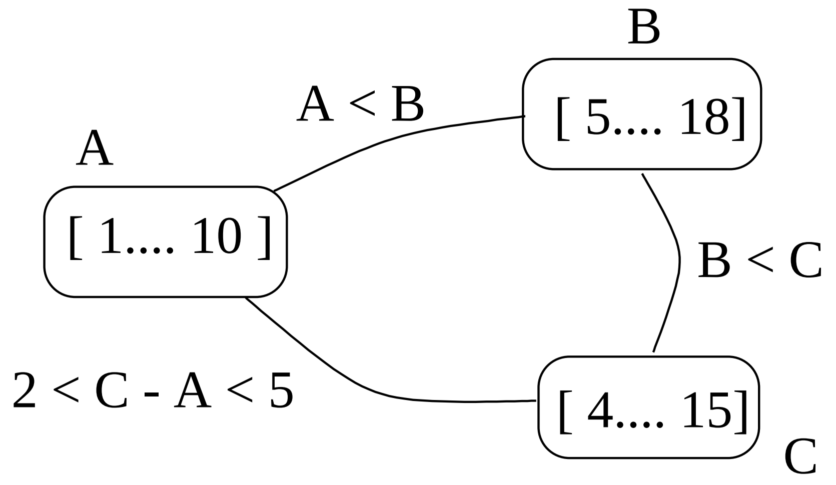
Given $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}) \begin{cases} \mathcal{V} = \{V_1, V_2, \ldots, V_n\} \\ \mathcal{D} = \{D_{V_1}, D_{V_2}, \ldots, D_{V_n}\} \\ \mathcal{C} \text{ set of constraints} \end{cases}$

$$C_{V_i,V_j} = \{\,(x,y)\} \subseteq D_{V_i} \times D_{V_j}$$

Constraint **graph**



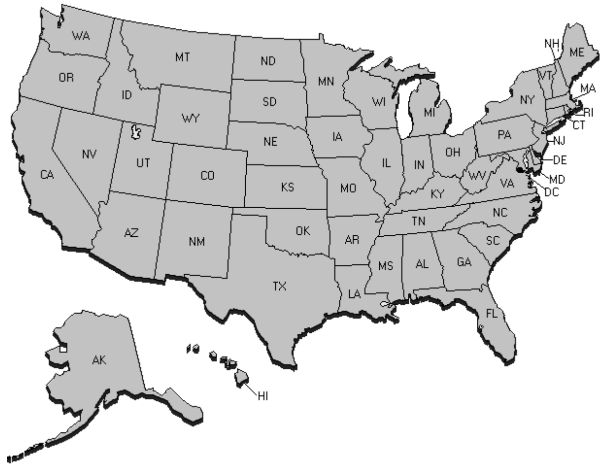
Example I: Temporal reasoning

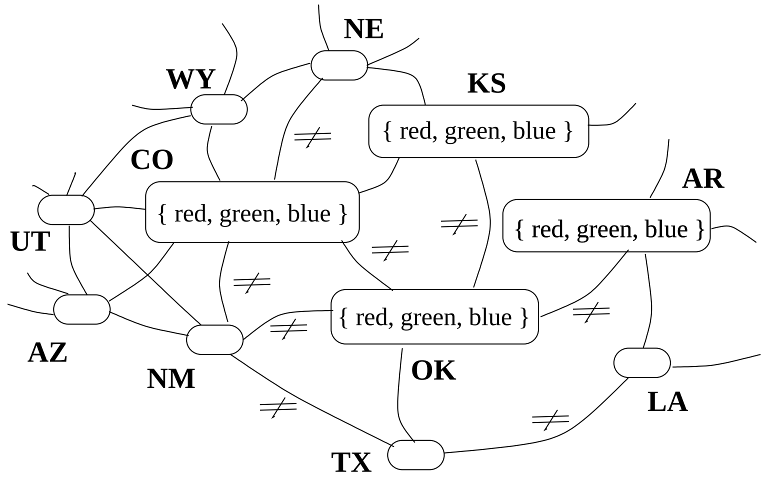


$\longrightarrow$ C-A $\in [2, 5]$ is a constraint of bounded differences

Example II: Map coloring

Using 3 colors (R, G, & B), color the US map such that no two adjacent states do have the same color





Variables? Domains? Constraints?

Domain types

Given $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C}) \begin{cases} \mathcal{V} = \{V_1, V_2, \ldots, V_n\} \\ \mathcal{D} = \{D_{V_1}, D_{V_2}, \ldots, D_{V_n}\} \\ \mathcal{C} \text{ set of constraints} \end{cases}$

$$C_{V_i,V_j} = \{\,(x, y)\} \subseteq D_{V_i} \times D_{V_j}$$

Domains:

$\longrightarrow$ restricted to $\{0, 1\}$: Boolean CSPs

$\longrightarrow$ Finite (discrete): enumeration techniques works

$\longrightarrow$ Continuous: sophisticated algebraic techniques are needed
consistency techniques on domain bounds

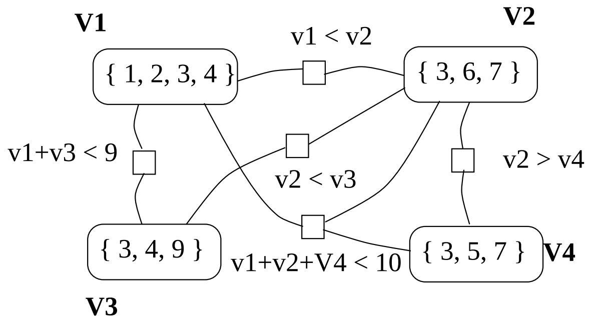
Constraint arity

Given $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ $\begin{cases} \mathcal{V} = \{V_1, V_2, \ldots, V_n\} \\ \mathcal{D} = \{D_{V_1}, D_{V_2}, \ldots, D_{V_n}\} \\ \mathcal{C} \text{ set of constraints} \end{cases}$

$$C_{V_k,V_l,V_m} = \{\,(x, y, z)\} \subseteq D_{V_k} \times D_{V_l} \times D_{V_m}$$

Constraints: universal, unary, binary, ternary, …, global

Representation: Constraint network



Constraint definition

Constraints can be defined

- Extensionally: all allowed tuples are listed
  practical for defining arbitrary constraints
  $C_{V_1,V_2} = \{(r,g),(r,b),(g,r),(g,b),(b,r),(b,g)\}$
- Intensionally: when it is not practical (or even possible) to list all tuples, define allowed tuples in intension.
  $C_{V_1,V_2} = \{(x,y) \mid x \in D_{V_1}, y \in D_{V_2}, x \neq y\}$
  $\rightarrow$ Define types of common constraints, to be used repeatedly
  Examples: Alldiff (a.k.a. mutex), Atmost, Cumulative, Balance, etc.

Other types of constraints: linear constraints, nonlinear constraints, constraints of bounded differences (e.g., in temporal reasoning), etc.

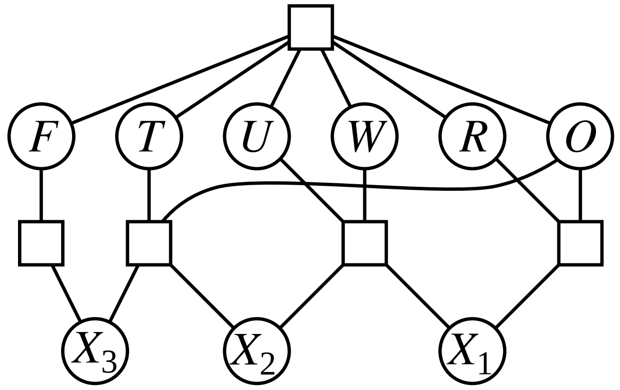
Example III: Cryptarithmetic puzzles

$$D_{X1} = D_{X2} = D_{X3} = \{0, 1\}$$

$$D_F = D_T = D_U = D_V = D_R = D_O = [0, 9]$$

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

(a)



(b)

O + O = R + 10 X1

X1 + W + W = U + 10 X2

X2 + T +T = O + 10 X3

X3 = F

Alldiff({F, D, U, V, R, O})

How to solve a CSP?

Search!

1. Constructive, systematic search
2. Local search

Incremental formulation: as a search problem

Initial state: empty assignment, all variables are unassigned

Successor function: a value is assigned to any unassigned variable, provided that it does not conflict with previously assigned variables (back-checking)

Goal test: The current assignment is complete (and consistent)

Path cost: a constant cost (e.g., 1) for every step, can be zero

— A solution is a complete, consistent assignment.

— Search tree has constant depth n (# of variables) $\rightarrow$ DFS!!

— However, path for reaching a solution is irrelevant

- Complete-state formulation is OK
- Solved with local search (ref. SAT)

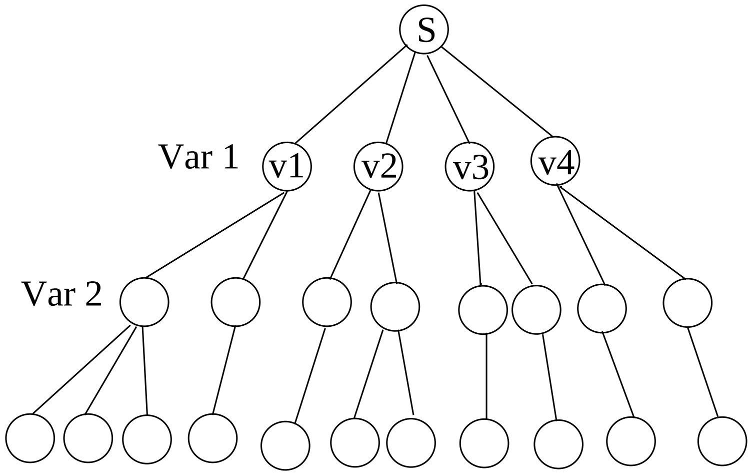
Systematic search

$\rightarrow$ Starting from a root node

$\rightarrow$ Consider all values for a variable V_1

$\rightarrow$ For every value for V_1, consider all values for V_2

$\rightarrow$ etc..



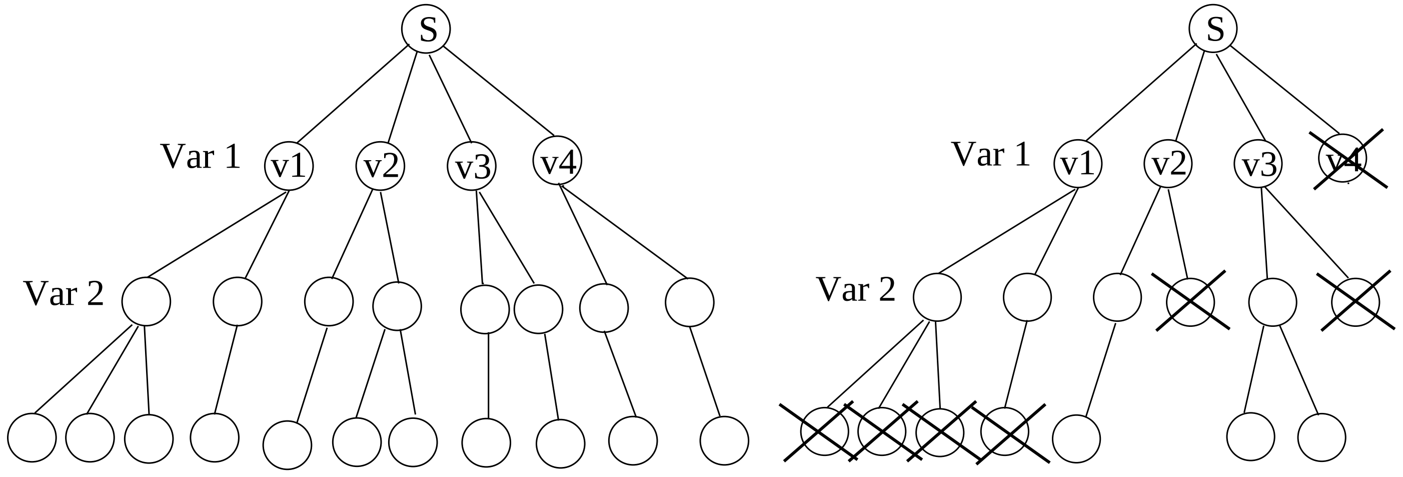
For n variables, each of domain size d:

- Maximum depth? *fixed!*
- Maximum number of paths? *size of search space, size of CSP*

Back-checking

Systematic search generates d^n possibilities

Are all possible combinations acceptable?



$\rightarrow$ Expand a partial solution only when consistent

$\longrightarrow$ **early pruning**

Before looking at search..

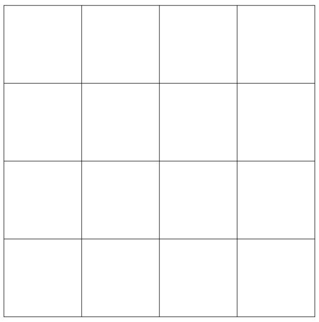
Consider

1. Importance of modeling/formulating
   to control the size of the search space

2. Preprocessing: consistency filtering
   to reduce size of search space

Importance of modeling

N-**queens**: formulation 1

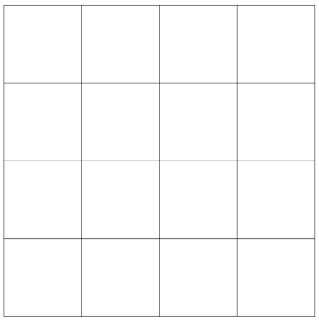


Variables?

Domains?

Size of CSP?

N-**queens**: formulation 2



variables?

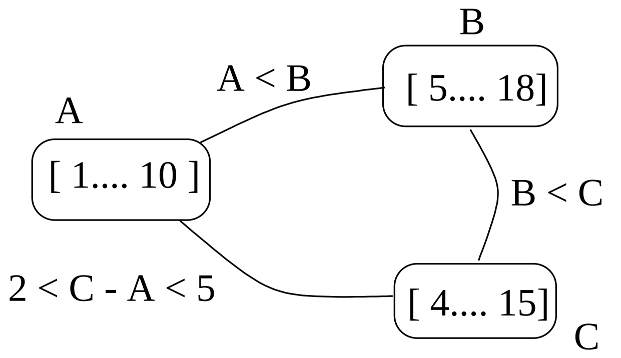
domains?

size of csp?

Constraint checking

$\longrightarrow$ Constraint filtering, constraint checking, etc..

eliminate non-acceptable tuples prior to search



1- B: [5 .. 14]
C: [6 .. 15]

2- A: [2 .. 10]
C: [6 .. 14]

3- B: [5 .. 13]

Revise(V_i,V_j)

For every value $x \in D_{V_i}$

If no $y \in D_{V_j}$ is consistent with x Then $D_{V_i} \leftarrow D_{V_i} \setminus \{x\}$

In AIMA: REMOVE-INCONSISTENT-VALUES(V_i,V_j)

REVISE (V_i,V_j)

```
1: $revised \leftarrow nil$
2: for all $x \in D_{v_i}$ do
3:     for all $y \in D_{v_j}$ do
4:         if CHECK($(V_i, x), (V_j, y)$) then
5:             RETURN($nil$)
6:         end if
7:     end for
8:     $D_{V_i} \leftarrow D_{V_i} \setminus \{x\}$
9:     $revised \leftarrow t$
10: end for
11: RETURN($revised$)
```

Arc Consistency

$\longrightarrow$ AC(C_{V_1,V_2}) = Revise(V_1,V_2) and Revise(V_2,V_1)

$\longrightarrow$ CSP is AC when all constraints are AC.

$\longrightarrow$ Algorithms: AC-1, AC-2, **AC-3**, …, **AC-7** and back to **AC-3**

$\longrightarrow$ AC-3: $O(n^2d^3)$

AC-3 (csp)

```
1: $Q \leftarrow \{(V_i, V_j) \mid C_{V_i,V_j} \text{ exists}\}$
2: **while** $Q \neq \emptyset$ **do**
3:     $(V_i, V_j) \leftarrow$ Pop($Q$)
4:     **if** Revise($V_i, V_j$) **then**
5:         **if** Domain($V_i$) $= \emptyset$ **then**
6:             Return($nil$)
7:         **else**
8:             **for all** $V_k \mid V_k \neq V_j$ and $C_{V_i,V_k}$ exists **do**
9:                 Push(($V_k$,$V_i$),$Q$)
10:            **end for**
11:        **end if**
12:    **end if**
13: **end while**
14: Return(csp)
```

Warning: arc-consistency does not solve the problem

Example: 3-coloring K_4

- In general, constraint propagation helps, but does not solve the problem
- As long as constraint checking is affordable (i.e., cost remains negligible vis-a-vis cost of search), it is advantageous to apply AC-3 before search

Levels of consistency

Node consistency: every value in the domain of a variable is consistent with the unary constraints defined on the variable

Arc-consistency: For any value in the domain of any variable, there is at least one value in the domain of any other variable with which it is consistent.

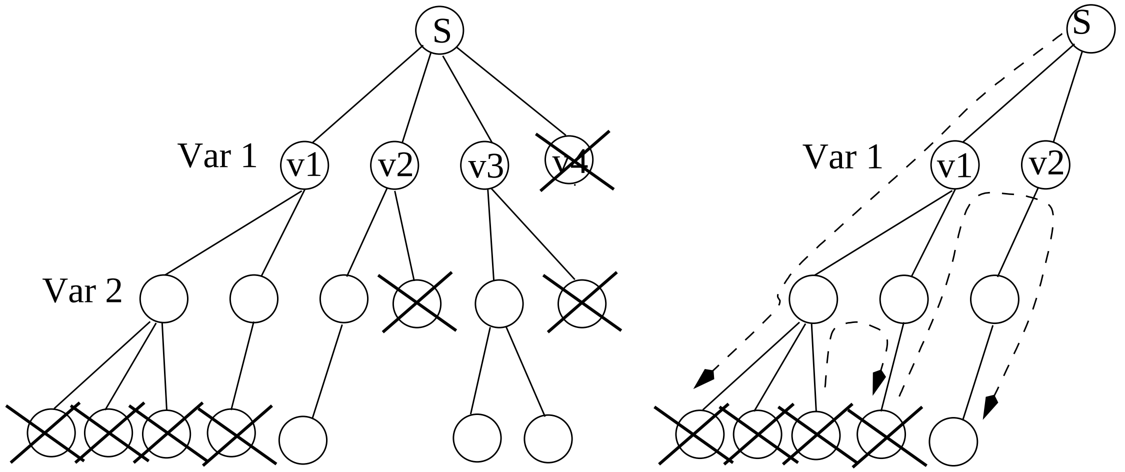
3-consistency: For any two consistent values in the domains of any two variables, there is at least one value in the domain of any third variable with which they are consistent.

k-consistency: $(k \leq n)$
For any (k-1) consistent values in the domains of any (k-1) variables, there is at least one value in the domain of any k^{th} variable with which they are consistent.

Strong k-consistency: k-consistency $\forall i \leq k$

Chronological backtracking

What if only one solution is needed?

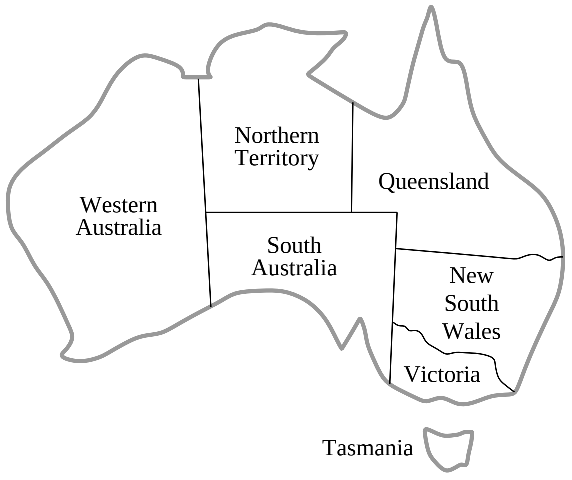


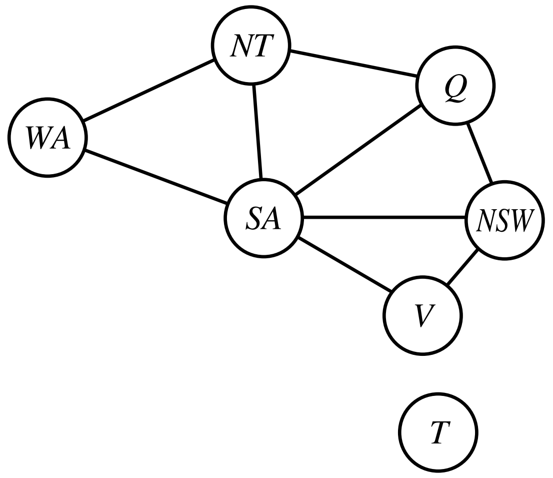
$\longrightarrow$ **Depth-first search & chronological backtracking**

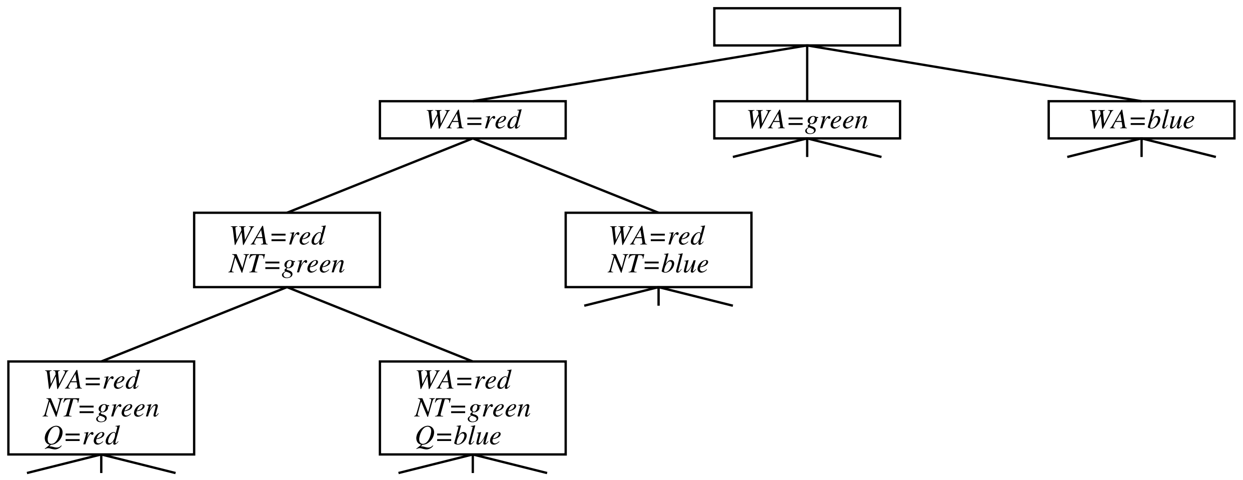
$\longrightarrow$ Terms: current variable V_c, past variables $\mathcal{V}_p$, future variables $\mathcal{V}_f$, current path

$\rightarrow$ DFS: soundness? completeness?

Example of BT







Backtrack(ing) search (BT)

Refer to algorithm BACKTRACKING-SEARCH

- Implementation: BACKTRACKING-SEARCH
  Careful, recursive, do not implement!!
  Use [Prosser 93] for iterative versions
- Variable ordering heuristic: SELECT-UNASSIGNED-VARIABLE
- Value ordering heuristic: ORDER-DOMAIN-VALUES

Improving BT

General purpose methods for:

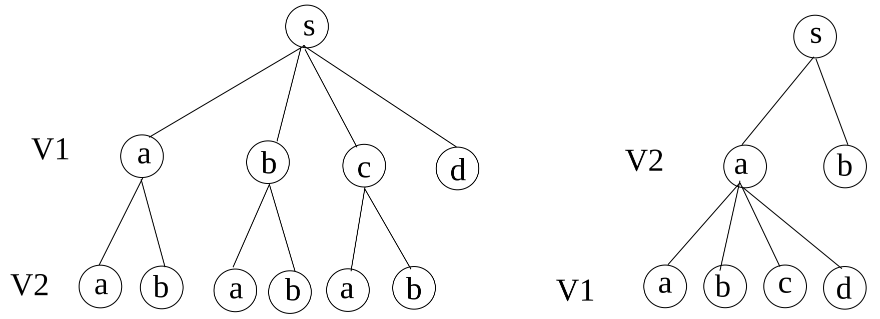
1. Variable, value ordering
2. Improving backtracking: intelligent backtracking avoids repeating failure
3. Look-ahead techniques: constraint propagation as instantiations are made

Ordering heuristics

Which variable to expand first?

Exp: $V_1, V_2,\ D_{V_1} = \{a, b, c, d\}, D_{V_2} = \{a, b\}$

Sol: $\{(V_1 = c), (V_2 = a)\}$ and $\{(V_1 = c), (V_2 = b)\}$



Heuristics: $\begin{cases} \text{most } \underline{\text{constrained}} \text{ variable first (reduce branching factor)} \\ \text{most } \underline{\text{promising}} \text{ value first (find quickly first solution)} \end{cases}$

Examples of ordering heuristics

For variables:

- least domain (LD), aka minimum remaining values (MRV
- degree
- ratio of domain size to degree (DD)
- width, promise, etc. [Tsang, Chapter 6]

For values:

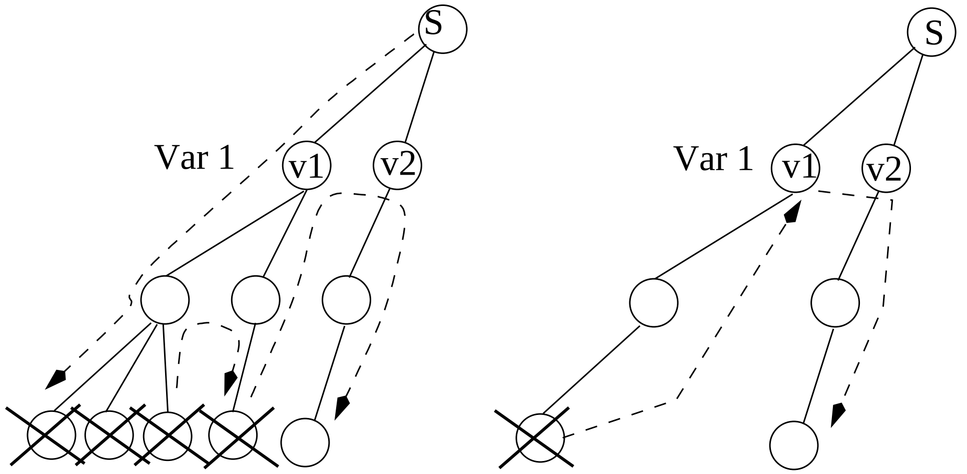
- min-conflict [Minton, 92]
- promise [Geelen, 94], etc.

$$\text{Strategies for} \left\{ \begin{array}{l} \text{variable ordering} \\ \text{value ordering} \end{array} \right\} \text{could be} \left\{ \begin{array}{l} \text{static} \\ \text{dynamic} \end{array} \right.$$

Intelligent backtracking

What if the reason for failure was higher up in the tree?

Backtrack to source of conflict!!



→ Backjumping, conflict-directed backjumping, etc.

→ Additional data structures that keep track of failure encountered during back-checking [Prosser, 93]

Look-ahead strategies: partial or full

As instantiations are made, remove the values from the domain of future variables that are not consistent with the current path

Terminology

- V_c is the current variable
- $\mathcal{V}_f$ is the set of future variables, V_f is a future variable
- Instantiate V_c, update the domains of (some) future variables

Strategies

- Forward checking (FC): partial look-ahead
- Directional arc-consistency checking (DAC): partial look-ahead
- Maintaining Arc-Consistency (MAC): full look-ahead

$\rightarrow$ Special data structures can be used to refresh filtered domains upon backtracking [Prosser, 93]

Forward checking (FC)

$\rightarrow$ Apply Revise(V_f, V_c) to the each variable V_f connected to V_c
$\rightarrow$ In AIMA, it is Remove-Inconsistent-Values(V_f, V_c)

Procedure:

- Instantiate V_c
- Apply Revise(V_f, V_c) to the each variable V_f

Directional Arc-Consistency (DAC)

$\rightarrow$ Repeat forward checking on all $V_f \in \mathcal{V}_f$ while respecting order
$\rightarrow$ Applicable under static ordering

Procedure:

- Choose a variable ordering
- Instantiate V_c
- Apply FC to V_c
- Move to next variable V_f in ordering, and apply FC to V_f.
  Repeat for all variables in $\mathcal{V}_f$ in the specified order.

Maintaining Arc-Consistency (MAC)

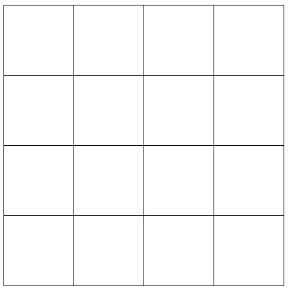
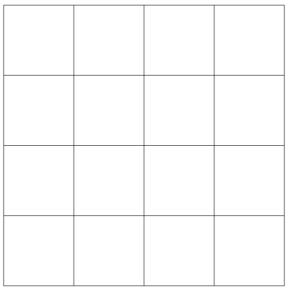
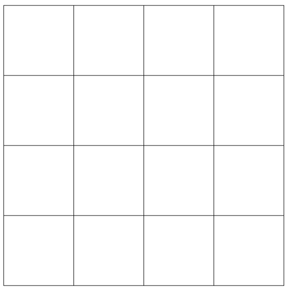
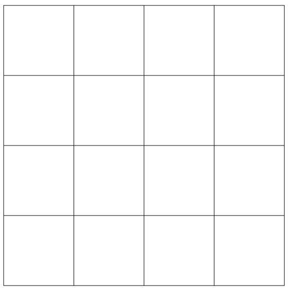
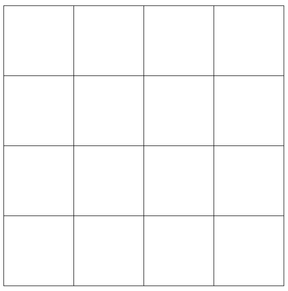
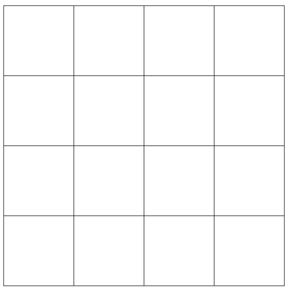
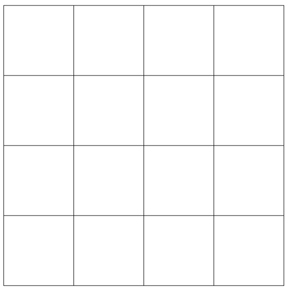
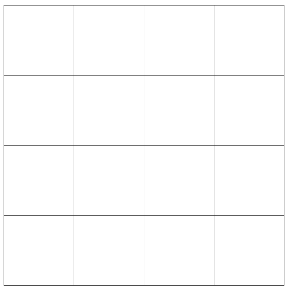
$\rightarrow$ Maintain AC in the subproblem induced by $\mathcal{V}_f \cup \{V_c\}$
$\rightarrow$ In practice, useful when problem has few, tight constraints

Procedure:

- Instantiate V_c
- Apply AC-3($\mathcal{V}_f \cup \{V_c\}$)
  Every constraint revision uses two operations: Revise(V_a, V_b) and Revise(V_b, V_a)
  Updates domains of all variables in subproblems

Search (V) ***Forward checking***

Why not filter right away effects of an action?

CSP: a decision problem (NP-complete)

1- Modeling:

— abstraction and reformulation

2- Preprocessing techniques:

— eliminate non-acceptable tuples prior to search

3- Search:

— potentially d^n paths of fixed length

— chronological backtracking

— variable/value ordering heuristics

— intelligent backtracking

4- Search 'hybrids':

— Mixing constraint propagation with search: FC, DAC, MAC

Non-systematic search (i.e., local search)

- **Methodology:** Iterative repair, local search: modifies a global but **inconsistent** solution to decrease the number of violated constraints
- **Example:** MIN-CONFLICTS algorithm in Fig 5.8, page 151. Choose (randomly) a variable in a broken constraint, and change its value using the min-conflict heuristic (which is a value ordering heuristic)
- **Other examples:** Hill climbing, taboo search, simulated annealing, etc.

$\longrightarrow$ Anytime algorithm
$\longrightarrow$ Strategies to avoid getting trapped: RandomWalk
$\longrightarrow$ Strategies to recover: Break-Out, Random restart, etc.
$\longrightarrow$ Incomplete & not sound

Exploiting structure: example of deep analysis

- Tree-structured CSP
- Cycle-cutset method

Tree-structured CSP

Any tree-structured CSP can be solved in time linear in the number of variables.

- Apply arc-consistency
  Directional arc-consistency is enough: starting from the leaves, revise a parent given the domain of a child; keep going up to the root
- Proceed, instantiating the variables from the root to the leaves
- The assignment can be done in a backtrack-free manner
- Runs in $O(nd^2)$, n is #variables and d domain size.

Cycle-cutset method

1. Identify a cycle cutset S in the CSP (nodes that when removed yield a tree), the remaining variables form the set T
2. Find a solution to the variables in S (S is smaller than initial problem)
3. For every consistent solution for variables in S:
   - Apply DAC from S to T
   - If no domain is wiped out, solve T (quick) and you have a solution to the CSP

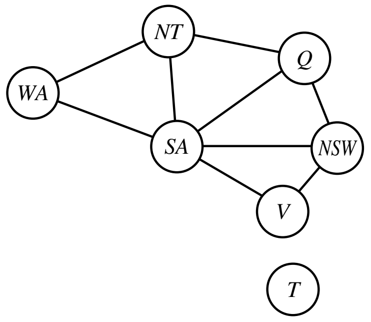
Note:

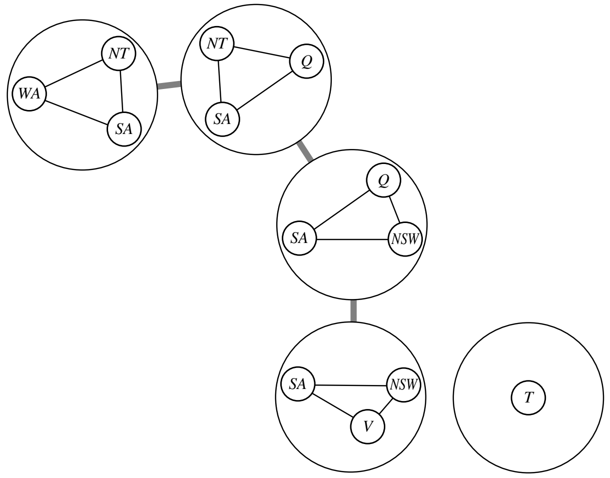
- For a cycle cutset $|S| = c$, time is $O(d^c.(n-c)d^2)$. If graph is nearly a tree, c is small, and savings are large. In the worst-case, $c = n - 2$:–(.
- Finding the smallest cutset is NP-hard :–(

Tree decomposition (tree-clustering)

Cluster the nodes of the CSP into subproblems, which are organized in a tree structure:

- Every variable appears in at least one subproblem
- If 2 variables are connected by a constraint, they must appear together (along with the constraint) in at least one subproblem
- If a variable appears in 2 subproblems, it must appear in every suproblem along the path between the 2 subproblems.





Solving the tree decomposition (tree-clustering)

- Each subproblem is a meta-variable, whose domain is the set of all solutions to the subproblem.
- Choose a subproblem, find all its solutions.
- Solve the constraints connecting the subproblem and its neighbors (common variables must agree).
- Repeat the process from a node to its descendant.
- Complexity depends on w, the tree width of the decomposition $=$ number of nodes in largest subproblem - 1. It is $O(nd^{w+1})$.
- Thus, CSPs with a constraint graph of bounded w can be solved in polynomial time.
- Finding the decomposition with minimal tree width in NP-hard..

Research directions

Preceding (*i.e.*, search, backtrack, iterative repair, V/V/ordering, consistency checking, decomposition, symmetries & interchangeability, deep analysis) + ...

Evaluation of algorithms:

worst-case analysis vs. empirical studies
random problems?

Cross-fertilization:

SAT, DB, mathematical programming,
interval mathematics, planning, etc.

Modeling & Reformulation

Multi agents:

Distribution and negotiation
$\rightarrow$ decomposition & alliance formation

CSP in a nutshell (I)

$$\textbf{Solution technique: } \text{Search}\begin{cases}\text{constructive}\\ \text{iterative repair}\end{cases}$$

$$\textbf{Enhancing search: }\begin{cases}\text{intelligent backtrack}\\ \text{variable/value ordering}\\ \text{consistency checking}\\ \text{hybrid search}\\ \heartsuit\ \text{symmetries}\\ \heartsuit\ \text{decomposition}\end{cases}$$

CSP in a nutshell (II)

Deep analysis: exploit problem structure
- ♡ graph topology
- ♡ constraint semantics
- phase transition

Research:
- k-ary constraints, soft constraints
- continuous vs. finite domains
- evaluation of algorithms (empirical)
- cross-fertilization (mathematical program.)
- ♡ reformulation and approximation
- ♡ architectures (multi-agent, negotiation)

Constraint Logic Programming (CLP)

A merger of

$\surd$ Constraint solving

$\longrightarrow$ Logic Programming, mostly Horn clauses (*e.g.*, Prolog)

Building blocks

- Constraint: primitives *but also user-defined*
  - cumulative/capacity (linear ineq), MUTEX, cycle, *etc.*
  - domain: Booleans, natural/rational/real numbers, finite
- Rules (declarative): a statement is a conjunction of constraints and is tested for satisfiability before execution proceeds further
- Mechanisms: satisfiability, entailment, delaying constraints

Constraint Processing Techniques are the basis of new languages:

Were you to ask me which programming paradigm is likely to gain most in commercial significance over the next 5 years I'd have to pick Constraint Logic Programming (CLP), even though it's perhaps currently one of the least known and understood. That's because CLP has the power to tackle those difficult combinatorial problems encountered for instance in job scheduling, timetabling, and routing which stretch conventional programming techniques beyond their breaking point.
<u>Though CLP is still the subject of intensive research, it's already being used by large corporations</u> such as manufacturers Michelin and Dassault, the French railway authority SNCF, airlines Swissair, SAS and Cathay Pacific, and Hong Kong International Terminals, the world's largest privately-owned container terminal.

Byte, Dick Pountain