

Fundamentals of Artificial Intelligence

Chapter 06: **Constraint Satisfaction Problems**

Roberto Sebastiani

DISI, Università di Trento, Italy – roberto.sebastiani@unitn.it

https://disi.unitn.it/rseba/DIDATTICA/fai_2023/

Teaching assistants:

Mauro Dragoni, dragoni@fbk.eu, <https://www.maurodragoni.com/teaching/fai/>

Paolo Morettin, paolo.morettin@unitn.it, <https://paolomorettin.github.io/>

M.S. Course “Artificial Intelligence Systems”, academic year 2024-2025

Last update: Thursday 5th September, 2024, 18:57

Copyright notice: Most examples and images displayed in the slides of this course are taken from [Russell & Norvig, “Artificial Intelligence, a Modern Approach”, 3rd ed., Pearson], including explicitly figures from the above-mentioned book, so that their copyright is detained by the authors. A few other material (text, figures, examples) is authored by (in alphabetical order): Pieter Abbeel, Bonnie J. Dorr, Anca Dragan, Dan Klein, Nikita Kitaev, Tom Lenaerts, Michela Milano, Dana Nau, Maria Simi, who detain its copyright.

These slides cannot be displayed in public without the permission of the author.

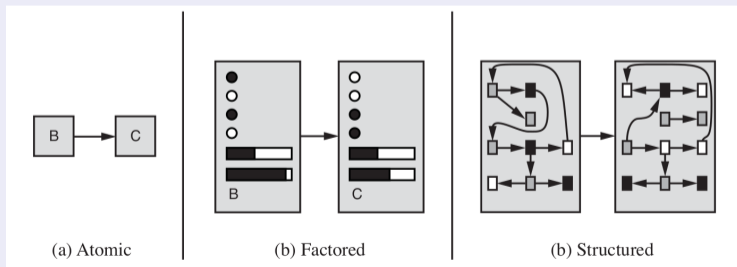
- 1 Constraint Satisfaction Problems (CSPs)
- 2 Search with CSPs
 - Inference: Constraint Propagation
 - Backtracking Search
 - Interleaving Search and Inference
 - Chronological vs. Conflict-Driven Backtracking
- 3 Local Search with CSPs
- 4 Exploiting Structure of CSPs

- 1 Constraint Satisfaction Problems (CSPs)
- 2 Search with CSPs
 - Inference: Constraint Propagation
 - Backtracking Search
 - Interleaving Search and Inference
 - Chronological vs. Conflict-Driven Backtracking
- 3 Local Search with CSPs
- 4 Exploiting Structure of CSPs

Recall: State Representations [Ch. 02]

Representations of states and transitions

- Three ways to represent states and transitions between them:
 - **atomic**: a state is a **black box with no internal structure**
 - **factored**: a state consists of a **vector of attribute values**
 - **structured**: a state **includes objects**, each of which may have **attributes** of its own as well as **relationships** to other objects
- increasing **expressive power** and **computational complexity**
- reality represented at **different levels of abstraction**



Constraint Satisfaction Problems (CSPs): Generalities

Constraint Satisfaction Problems, CSPs (aka Constraint Satisfiability Problems)

- Search problem so far: **Atomic representation of states**
 - black box with no internal structure
 - goal test as set inclusion
- Henceforth: use a **Factored representation of states**
 - **state** is defined by a **set of variables values** from some domains
 - **goal test** is a **set of constraints** specifying allowable combinations of values for subsets of variables
 - a set of variable values is a goal iff the values verify all constraints
- **CSP Search Algorithms**
 - take advantage of the **structure of states**
 - **use general-purpose heuristics rather than problem-specific ones**
 - main idea: **eliminate large portions of the search space all at once**
 - identify variable/value combinations that violate the constraints

Constraint Satisfaction Problems (CSPs): Generalities

Constraint Satisfaction Problems, CSPs (aka Constraint Satisfiability Problems)

- Search problem so far: **Atomic representation of states**
 - black box with no internal structure
 - goal test as set inclusion
- Henceforth: use a **Factored representation of states**
 - **state** is defined by **a set of variables values** from some domains
 - **goal test** is a **set of constraints** specifying allowable combinations of values for subsets of variables
 - a set of variable values is a goal iff the values verify all constraints
- **CSP Search Algorithms**
 - take advantage of the **structure of states**
 - **use general-purpose heuristics rather than problem-specific ones**
 - main idea: **eliminate large portions of the search space all at once**
 - identify variable/value combinations that violate the constraints

Constraint Satisfaction Problems (CSPs): Generalities

Constraint Satisfaction Problems, CSPs (aka Constraint Satisfiability Problems)

- Search problem so far: **Atomic representation of states**
 - black box with no internal structure
 - goal test as set inclusion
- Henceforth: use a **Factored representation of states**
 - **state** is defined by **a set of variables values** from some domains
 - **goal test** is a **set of constraints** specifying allowable combinations of values for subsets of variables
 - a set of variable values is a goal iff the values verify all constraints
- **CSP Search Algorithms**
 - take advantage of the **structure of states**
 - **use general-purpose heuristics rather than problem-specific ones**
 - main idea: **eliminate large portions of the search space all at once**
 - identify variable/value combinations that violate the constraints

CSPs: Definitions

CSPs

- A **Constraint Satisfaction Problem** is a tuple $\langle X, D, C \rangle$:
 - a set of variables $X \stackrel{\text{def}}{=} \{X_1, \dots, X_n\}$
 - a set of (non-empty) domains $D \stackrel{\text{def}}{=} \{D_1, \dots, D_n\}$, one for each X_i
 - a set of constraints $C \stackrel{\text{def}}{=} \{C_1, \dots, C_m\}$
 - specify allowable combinations of values for the variables in X
- Each D_i is a set of allowable values $\{v_j, \dots, v_k\}$ for variable X_i
- Each C_i is a pair $\langle \text{scope}, \text{rel} \rangle$
 - **scope** is a tuple of variables that participate in the constraint
 - **rel** is a **relation** defining the values that such variables can take
- A **relation** is
 - an explicit list of all tuples of values that satisfy the constraint (most often inconvenient), or
 - an abstract relation supporting two operations:
 - test if a tuple is a member of the relation
 - enumerate the members of the relation
- We need a language to express constraint relations!

CSPs: Definitions

CSPs

- A **Constraint Satisfaction Problem** is a tuple $\langle X, D, C \rangle$:
 - a set of variables $X \stackrel{\text{def}}{=} \{X_1, \dots, X_n\}$
 - a set of (non-empty) domains $D \stackrel{\text{def}}{=} \{D_1, \dots, D_n\}$, one for each X_i
 - a set of constraints $C \stackrel{\text{def}}{=} \{C_1, \dots, C_m\}$
 - specify allowable combinations of values for the variables in X
- Each D_i is a set of allowable values $\{v_i, \dots, v_k\}$ for variable X_i
- Each C_i is a pair $\langle \text{scope}, \text{rel} \rangle$
 - **scope** is a tuple of variables that participate in the constraint
 - **rel** is a relation defining the values that such variables can take
- A relation is
 - an explicit list of all tuples of values that satisfy the constraint (most often inconvenient), or
 - an abstract relation supporting two operations:
 - test if a tuple is a member of the relation
 - enumerate the members of the relation
- We need a language to express constraint relations!

CSPs: Definitions

CSPs

- A **Constraint Satisfaction Problem** is a tuple $\langle X, D, C \rangle$:
 - a set of variables $X \stackrel{\text{def}}{=} \{X_1, \dots, X_n\}$
 - a set of (non-empty) domains $D \stackrel{\text{def}}{=} \{D_1, \dots, D_n\}$, one for each X_i
 - a set of constraints $C \stackrel{\text{def}}{=} \{C_1, \dots, C_m\}$
 - specify allowable combinations of values for the variables in X
- Each D_i is a set of allowable values $\{v_i, \dots, v_k\}$ for variable X_i
- Each C_i is a pair $\langle \text{scope}, \text{rel} \rangle$
 - **scope** is a tuple of variables that participate in the constraint
 - **rel** is a relation defining the values that such variables can take
- A relation is
 - an explicit list of all tuples of values that satisfy the constraint (most often inconvenient), or
 - an abstract relation supporting two operations:
 - test if a tuple is a member of the relation
 - enumerate the members of the relation
- We need a language to express constraint relations!

CSPs: Definitions

CSPs

- A **Constraint Satisfaction Problem** is a tuple $\langle X, D, C \rangle$:
 - a set of variables $X \stackrel{\text{def}}{=} \{X_1, \dots, X_n\}$
 - a set of (non-empty) domains $D \stackrel{\text{def}}{=} \{D_1, \dots, D_n\}$, one for each X_i
 - a set of constraints $C \stackrel{\text{def}}{=} \{C_1, \dots, C_m\}$
 - specify allowable combinations of values for the variables in X
- Each D_i is a set of allowable values $\{v_i, \dots, v_k\}$ for variable X_i
- Each C_i is a pair $\langle \text{scope}, \text{rel} \rangle$
 - **scope** is a tuple of variables that participate in the constraint
 - **rel** is a relation defining the values that such variables can take
- A **relation** is
 - an explicit list of all tuples of values that satisfy the constraint (most often inconvenient), or
 - an abstract relation supporting two operations:
 - test if a tuple is a member of the relation
 - enumerate the members of the relation
- We need a language to express constraint relations!

CSPs: Definitions

CSPs

- A **Constraint Satisfaction Problem** is a tuple $\langle X, D, C \rangle$:
 - a set of variables $X \stackrel{\text{def}}{=} \{X_1, \dots, X_n\}$
 - a set of (non-empty) domains $D \stackrel{\text{def}}{=} \{D_1, \dots, D_n\}$, one for each X_i
 - a set of constraints $C \stackrel{\text{def}}{=} \{C_1, \dots, C_m\}$
 - specify allowable combinations of values for the variables in X
- Each D_i is a set of allowable values $\{v_i, \dots, v_k\}$ for variable X_i
- Each C_i is a pair $\langle \text{scope}, \text{rel} \rangle$
 - **scope** is a tuple of variables that participate in the constraint
 - **rel** is a relation defining the values that such variables can take
- A **relation** is
 - an explicit list of all tuples of values that satisfy the constraint (most often inconvenient), or
 - an abstract relation supporting two operations:
 - test if a tuple is a member of the relation
 - enumerate the members of the relation
- We need a language to express constraint relations!

CSPs: Definitions [cont.]

States, Assignments and Solutions

- A **state** in a CSP is **an assignment of values to some or all of the variables** $\{X_i = v_{x_i}\}_i$ s.t. $X_i \in X$ and $v_{x_i} \in D_i$
- An assignment is
 - **complete** (aka **total**) if every variable is assigned a value
 - **incomplete** (aka **partial**) if some variable is assigned a value
- An assignment that does not violate any constraints in the CSP is called a **consistent** or **legal assignment**
- A **solution** to a CSP is a **consistent and complete assignment**
- A CSP consists in finding one solution (or state there is none)
- **Constraint Optimization Problems (COPs):**
CSPs requiring solutions that maximize/minimize an objective function

CSPs: Definitions [cont.]

States, Assignments and Solutions

- A **state** in a CSP is **an assignment of values to some or all of the variables** $\{X_i = v_{x_i}\}_i$ s.t. $X_i \in X$ and $v_{x_i} \in D_i$
- An assignment is
 - **complete** (aka **total**) if every variable is assigned a value
 - **incomplete** (aka **partial**) if some variable is assigned a value
- An assignment that does not violate any constraints in the CSP is called a **consistent** or **legal assignment**
- A **solution** to a CSP is a **consistent and complete assignment**
- A CSP consists in finding one solution (or state there is none)
- **Constraint Optimization Problems (COPs):**
CSPs requiring solutions that maximize/minimize an objective function

CSPs: Definitions [cont.]

States, Assignments and Solutions

- A **state** in a CSP is **an assignment of values to some or all of the variables** $\{X_i = v_{x_i}\}_i$ s.t. $X_i \in X$ and $v_{x_i} \in D_i$
- An assignment is
 - **complete** (aka **total**) if every variable is assigned a value
 - **incomplete** (aka **partial**) if some variable is assigned a value
- An assignment that does not violate any constraints in the CSP is called a **consistent** or **legal assignment**
- A **solution** to a CSP is a **consistent and complete assignment**
- A CSP consists in finding one solution (or state there is none)
- **Constraint Optimization Problems (COPs):**
CSPs requiring solutions that maximize/minimize an objective function

CSPs: Definitions [cont.]

States, Assignments and Solutions

- A **state** in a CSP is **an assignment of values to some or all of the variables** $\{X_i = v_{x_i}\}_i$ s.t $X_i \in X$ and $v_{x_i} \in D_i$
- An assignment is
 - **complete** (aka **total**) if every variable is assigned a value
 - **incomplete** (aka **partial**) if some variable is assigned a value
- An assignment that does not violate any constraints in the CSP is called a **consistent** or **legal assignment**
- A **solution** to a CSP is a **consistent and complete assignment**
- A CSP consists in finding one solution (or state there is none)
- Constraint Optimization Problems (COPs):
CSPs requiring solutions that maximize/minimize an objective function

CSPs: Definitions [cont.]

States, Assignments and Solutions

- A **state** in a CSP is **an assignment of values to some or all of the variables** $\{X_i = v_{x_i}\}_i$ s.t. $X_i \in X$ and $v_{x_i} \in D_i$
- An assignment is
 - **complete** (aka **total**) if every variable is assigned a value
 - **incomplete** (aka **partial**) if some variable is assigned a value
- An assignment that does not violate any constraints in the CSP is called a **consistent** or **legal assignment**
- A **solution** to a CSP is a **consistent and complete assignment**
- **A CSP consists in finding one solution (or state there is none)**
- **Constraint Optimization Problems (COPs):**
CSPs requiring solutions that maximize/minimize an objective function

CSPs: Definitions [cont.]

States, Assignments and Solutions

- A **state** in a CSP is **an assignment of values to some or all of the variables** $\{X_i = v_{x_i}\}_i$ s.t. $X_i \in X$ and $v_{x_i} \in D_i$
- An assignment is
 - **complete** (aka **total**) if every variable is assigned a value
 - **incomplete** (aka **partial**) if some variable is assigned a value
- An assignment that does not violate any constraints in the CSP is called a **consistent** or **legal assignment**
- A **solution** to a CSP is a **consistent and complete assignment**
- **A CSP consists in finding one solution (or state there is none)**
- **Constraint Optimization Problems (COPs):**
CSPs requiring solutions that maximize/minimize an objective function

Example: Sudoku

- 81 Variables: (each square) X_{ij} ,
 $i = A, \dots, I$; $j = 1 \dots 9$
- Domain: $\{1, 2, \dots, 8, 9\}$
- Constraints:
 - $AllDiff(X_{i1}, \dots, X_{i9})$ for each row i
 - $AllDiff(X_{A1}, \dots, X_{I1})$ for each column j
 - $AllDiff(X_{A1}, \dots, X_{A3}, X_{B1}, \dots, X_{C3})$ for each 3×3 square region(alternatively, a long list of pairwise inequality constraints: $X_{A1} \neq X_{A2}, X_{A1} \neq X_{A3}, \dots$)
- Solution: total value assignment satisfying all the constraints: $X_{A1} = 4, X_{A2} = 8, X_{A3} = 3, \dots$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(© S. Russell & P. Norwig, AIMA)

Example: Sudoku

- 81 Variables: (each square) X_{ij} ,
 $i = A, \dots, I$; $j = 1 \dots 9$
- Domain: $\{1, 2, \dots, 8, 9\}$
- Constraints:
 - $AllDiff(X_{i1}, \dots, X_{i9})$ for each row i
 - $AllDiff(X_{A1}, \dots, X_{I1})$ for each column j
 - $AllDiff(X_{A1}, \dots, X_{A3}, X_{B1}, \dots, X_{C3})$ for each 3×3 square region(alternatively, a long list of pairwise inequality constraints: $X_{A1} \neq X_{A2}, X_{A1} \neq X_{A3}, \dots$)
- Solution: total value assignment satisfying all the constraints: $X_{A1} = 4, X_{A2} = 8, X_{A3} = 3, \dots$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(© S. Russell & P. Norwig, AIMA)

Example: Sudoku

- 81 Variables: (each square) X_{ij} ,
 $i = A, \dots, I$; $j = 1 \dots 9$
 - Domain: $\{1, 2, \dots, 8, 9\}$
 - Constraints:
 - $AllDiff(X_{i1}, \dots, X_{i9})$ for each row i
 - $AllDiff(X_{A_j}, \dots, X_{I_j})$ for each column j
 - $AllDiff(X_{A_1}, \dots, X_{A_3}, X_{B_1}, \dots, X_{C_3})$ for each 3×3 square region
- (alternatively, a long list of pairwise inequality constraints: $X_{A1} \neq X_{A2}, X_{A1} \neq X_{A3}, \dots$)
- Solution: total value assignment satisfying all the constraints: $X_{A1} = 4, X_{A2} = 8, X_{A3} = 3, \dots$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(© S. Russell & P. Norwig, AIMA)

Example: Sudoku

- 81 Variables: (each square) X_{ij} ,
 $i = A, \dots, I$; $j = 1 \dots 9$
- Domain: $\{1, 2, \dots, 8, 9\}$
- Constraints:
 - $AllDiff(X_{i1}, \dots, X_{i9})$ for each row i
 - $AllDiff(X_{A_j}, \dots, X_{I_j})$ for each column j
 - $AllDiff(X_{A_1}, \dots, X_{A_3}, X_{B_1}, \dots, X_{C_3})$ for each 3×3 square region(alternatively, a long list of pairwise inequality constraints: $X_{A_1} \neq X_{A_2}, X_{A_1} \neq X_{A_3}, \dots$)
- Solution: total value assignment satisfying all the constraints: $X_{A_1} = 4, X_{A_2} = 8, X_{A_3} = 3, \dots$

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(© S. Russell & P. Norwig, AIMA)

Example: Map-Coloring

- Variables WA, NT, Q, NSW, V, SA, T
- Domain $D_i = \{\text{red}, \text{green}, \text{blue}\}, \forall i$
- Constraints: adjacent regions must have different colours
 - e.g. (explicit enumeration): $\langle WA, NT \rangle \in \{\langle \text{red}, \text{green} \rangle, \langle \text{red}, \text{blue} \rangle, \}$
or (implicit, if language allows it): $WA \neq NT$
- A solution: $\{\text{WA}=\text{red}, \text{NT}=\text{green}, \text{Q}=\text{red}, \text{NSW}=\text{green}, \text{V}=\text{red}, \text{SA}=\text{blue}, \text{T}=\text{green}\}$



Example: Map-Coloring

- Variables WA, NT, Q, NSW, V, SA, T
- Domain $D_i = \{\text{red}, \text{green}, \text{blue}\}, \forall i$
- Constraints: adjacent regions must have different colours
 - e.g. (explicit enumeration): $\langle WA, NT \rangle \in \{\langle \text{red}, \text{green} \rangle, \langle \text{red}, \text{blue} \rangle, \}$
or (implicit, if language allows it): $WA \neq NT$
- A solution: $\{\text{WA}=\text{red}, \text{NT}=\text{green}, \text{Q}=\text{red}, \text{NSW}=\text{green}, \text{V}=\text{red}, \text{SA}=\text{blue}, \text{T}=\text{green}\}$



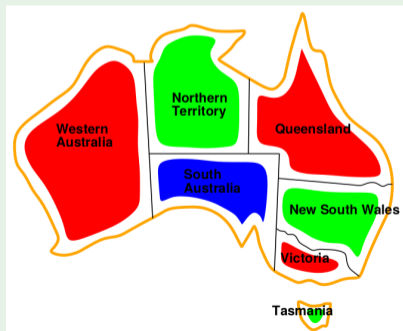
Example: Map-Coloring

- Variables WA, NT, Q, NSW, V, SA, T
- Domain $D_i = \{\text{red}, \text{green}, \text{blue}\}, \forall i$
- Constraints: adjacent regions must have different colours
 - e.g. (explicit enumeration): $\langle \text{WA}, \text{NT} \rangle \in \{\langle \text{red}, \text{green} \rangle, \langle \text{red}, \text{blue} \rangle, \}$
or (implicit, if language allows it): $\text{WA} \neq \text{NT}$
- A solution: $\{\text{WA}=\text{red}, \text{NT}=\text{green}, \text{Q}=\text{red}, \text{NSW}=\text{green}, \text{V}=\text{red}, \text{SA}=\text{blue}, \text{T}=\text{green}\}$



Example: Map-Coloring

- Variables WA, NT, Q, NSW, V, SA, T
- Domain $D_i = \{\text{red}, \text{green}, \text{blue}\}, \forall i$
- Constraints: adjacent regions must have different colours
 - e.g. (explicit enumeration): $\langle \text{WA}, \text{NT} \rangle \in \{\langle \text{red}, \text{green} \rangle, \langle \text{red}, \text{blue} \rangle, \}$
or (implicit, if language allows it): $\text{WA} \neq \text{NT}$
- A solution: $\{\text{WA}=\text{red}, \text{NT}=\text{green}, \text{Q}=\text{red}, \text{NSW}=\text{green}, \text{V}=\text{red}, \text{SA}=\text{blue}, \text{T}=\text{green}\}$



Constraint Graphs

- Useful to visualize a CSP as a **constraint graph** (aka **network**)
 - the **nodes of the graph** correspond to variables of the problem
 - an **edge** connects any two variables that participate in a constraint
- CSP algorithms use the graph structure to speed up search
 - Ex: Tasmania is an independent subproblem!

Constraint Graphs

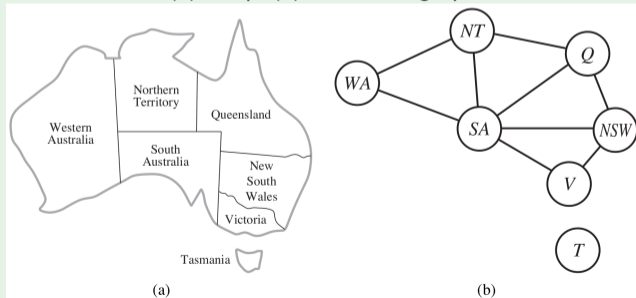
- Useful to visualize a CSP as a **constraint graph** (aka **network**)
 - the **nodes of the graph** correspond to variables of the problem
 - an **edge** connects any two variables that participate in a constraint
- CSP algorithms use the graph structure to speed up search
 - Ex: Tasmania is an independent subproblem!

Constraint Graphs

- Useful to visualize a CSP as a **constraint graph** (aka **network**)
 - the **nodes of the graph** correspond to **variables of the problem**
 - an **edge** connects **any two variables that participate in a constraint**
- CSP algorithms use the graph structure to speed up search
 - Ex: **Tasmania is an independent subproblem!**

Example: Map Coloring

(a): map; (b) constraint graph



Varieties of CSPs

- Discrete variables
 - Finite domains (ex: Booleans, bounded integers, lists of values)
 - domain size $d \implies d^n$ complete assignments (candidate solutions)
 - e.g. Boolean CSPs, incl. Boolean satisfiability (NP-complete)
 - possible to define constraints by enumerating all combinations (although unpractical)
 - Infinite domains (ex: unbounded integers)
 - infinite domain size \implies infinite # of complete assignments
 - e.g. job scheduling: variables are start/end days for each job
 - need a constraint language (ex: $StartJob_1 + 5 \leq StartJob_3$)
 - linear constraints \implies solvable (but NP-Hard)
 - non-linear constraints \implies undecidable (ex: $x^n + y^n = z^n, n > 2$)
- Continuous variables (ex: reals, rationals)
 - linear constraints solvable in poly time by LP methods
 - non-linear constraints solvable (e.g. by Cylindrical Algebraic Decomposition) but dramatically hard

The same problem may have distinct formulations as CSP!

Varieties of CSPs

- Discrete variables
 - Finite domains (ex: Booleans, bounded integers, lists of values)
 - domain size $d \implies d^n$ complete assignments (candidate solutions)
 - e.g. Boolean CSPs, incl. Boolean satisfiability (NP-complete)
 - possible to define constraints by enumerating all combinations (although unpractical)
 - Infinite domains (ex: unbounded integers)
 - infinite domain size \implies infinite # of complete assignments
 - e.g. job scheduling: variables are start/end days for each job
 - need a constraint language (ex: $StartJob_1 + 5 \leq StartJob_3$)
 - linear constraints \implies solvable (but NP-Hard)
 - non-linear constraints \implies undecidable (ex: $x^n + y^n = z^n, n > 2$)
- Continuous variables (ex: reals, rationals)
 - linear constraints solvable in poly time by LP methods
 - non-linear constraints solvable (e.g. by Cylindrical Algebraic Decomposition) but dramatically hard

The same problem may have distinct formulations as CSP!

Varieties of CSPs

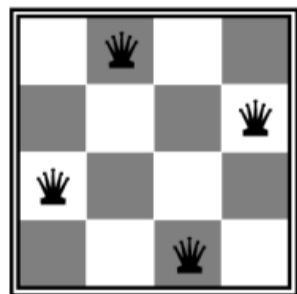
- Discrete variables
 - Finite domains (ex: **Booleans**, **bounded integers**, **lists of values**)
 - domain size $d \implies d^n$ **complete assignments** (candidate solutions)
 - e.g. Boolean CSPs, incl. **Boolean satisfiability** (NP-complete)
 - **possible to define constraints by enumerating all combinations** (although unpractical)
 - Infinite domains (ex: **unbounded integers**)
 - infinite domain size \implies **infinite # of complete assignments**
 - e.g. **job scheduling**: variables are start/end days for each job
 - need a **constraint language** (ex: $StartJob_1 + 5 \leq StartJob_3$)
 - **linear constraints** \implies solvable (but NP-Hard)
 - **non-linear constraints** \implies undecidable (ex: $x^n + y^n = z^n, n > 2$)
- Continuous variables (ex: **reals**, **rationals**)
 - **linear constraints** solvable in poly time by LP methods
 - **non-linear constraints** solvable (e.g. by **Cylindrical Algebraic Decomposition**) but dramatically hard

The same problem may have distinct formulations as CSP!

Example: N-Queens

Formulation #1

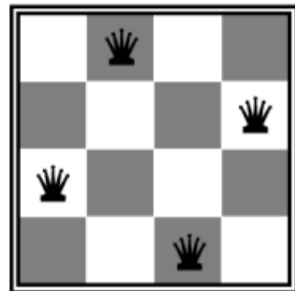
- variables X_{ij} , $i, j = 1..N$ (there is a queen i position i, j)
- domains: $\{0, 1\}$ (false,true)
- constraints (explicit):
 - $\forall i, j, k \langle X_{ij}, X_{ik} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (row)
 - $\forall i, j, k \langle X_{ij}, X_{kj} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (column)
 - $\forall i, j, k \langle X_{ij}, X_{i+k, j+k} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (upward diagonal)
 - $\forall i, j, k \langle X_{ij}, X_{i+k, j-k} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (downward diagonal)
- explicit representation
- very inefficient



Example: N-Queens

Formulation #1

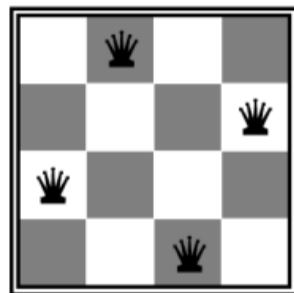
- variables X_{ij} , $i, j = 1..N$ (there is a queen i position i, j)
- domains: $\{0, 1\}$ (false,true)
- constraints (explicit):
 - $\forall i, j, k \langle X_{ij}, X_{ik} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (row)
 - $\forall i, j, k \langle X_{ij}, X_{kj} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (column)
 - $\forall i, j, k \langle X_{ij}, X_{i+k, j+k} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (upward diagonal)
 - $\forall i, j, k \langle X_{ij}, X_{i+k, j-k} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (downward diagonal)
- explicit representation
- very inefficient



Example: N-Queens

Formulation #1

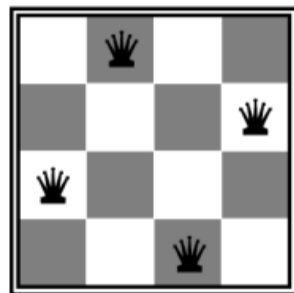
- variables X_{ij} , $i, j = 1..N$ (there is a queen i position i, j)
- domains: $\{0, 1\}$ (false,true)
- constraints (explicit):
 - $\forall i, j, k \langle X_{ij}, X_{ik} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (row)
 - $\forall i, j, k \langle X_{ij}, X_{kj} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (column)
 - $\forall i, j, k \langle X_{ij}, X_{i+k, j+k} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (upward diagonal)
 - $\forall i, j, k \langle X_{ij}, X_{i+k, j-k} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (downward diagonal)
- **explicit representation**
- very inefficient



Example: N-Queens

Formulation #1

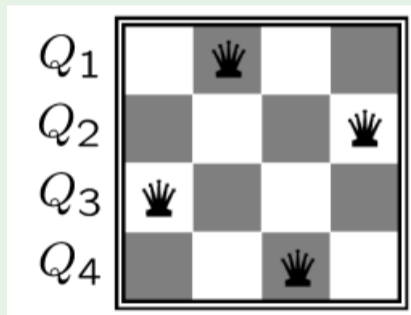
- variables X_{ij} , $i, j = 1..N$ (there is a queen i position i, j)
- domains: $\{0, 1\}$ (false,true)
- constraints (explicit):
 - $\forall i, j, k \langle X_{ij}, X_{ik} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (row)
 - $\forall i, j, k \langle X_{ij}, X_{kj} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (column)
 - $\forall i, j, k \langle X_{ij}, X_{i+k, j+k} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (upward diagonal)
 - $\forall i, j, k \langle X_{ij}, X_{i+k, j-k} \rangle \in \{\langle 0, 0 \rangle, \langle 1, 0 \rangle, \langle 0, 1 \rangle\}$ (downward diagonal)
- explicit representation
- very inefficient



Example: N-Queens [cont.]

Formulation #2

- variables Q_k , $k = 1..N$ (row)
- domains: $\{1..N\}$ (column position)
- constraints (implicit): *Nonthreatening*($Q_k, Q_{k'}$):
 - none (row)
 - $Q_i \neq Q_j$ (column)
 - $Q_i \neq Q_{j+k} + k$ (downward diagonal)
 - $Q_i \neq Q_{j+k} - k$ (upward diagonal)
- implicit representation
- much more efficient

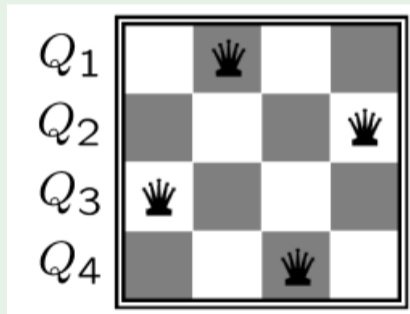


(© S. Russell & P. Norwig, AIMA)

Example: N-Queens [cont.]

Formulation #2

- variables Q_k , $k = 1..N$ (row)
- domains: $\{1..N\}$ (column position)
- constraints (implicit): *Nonthreatening*($Q_k, Q_{k'}$):
 - none (row)
 - $Q_i \neq Q_j$ (column)
 - $Q_i \neq Q_{j+k} + k$ (downward diagonal)
 - $Q_i \neq Q_{j+k} - k$ (upward diagonal)
- implicit representation
- much more efficient

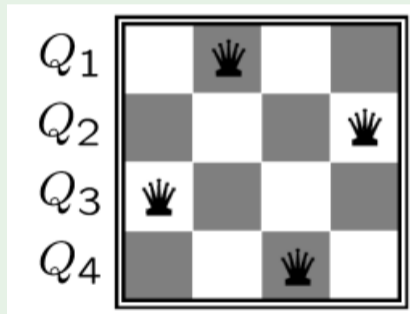


(© S. Russell & P. Norwig, AIMA)

Example: N-Queens [cont.]

Formulation #2

- variables Q_k , $k = 1..N$ (row)
- domains: $\{1..N\}$ (column position)
- constraints (implicit): *Nonthreatening*($Q_k, Q_{k'}$):
 - none (row)
 - $Q_i \neq Q_j$ (column)
 - $Q_i \neq Q_{j+k} + k$ (downward diagonal)
 - $Q_i \neq Q_{j+k} - k$ (upward diagonal)
- **implicit representation**
- much more efficient

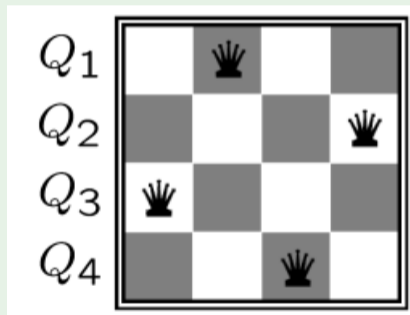


(© S. Russell & P. Norwig, AIMA)

Example: N-Queens [cont.]

Formulation #2

- variables Q_k , $k = 1..N$ (row)
- domains: $\{1..N\}$ (column position)
- constraints (implicit): *Nonthreatening*($Q_k, Q_{k'}$):
 - none (row)
 - $Q_i \neq Q_j$ (column)
 - $Q_i \neq Q_{j+k} + k$ (downward diagonal)
 - $Q_i \neq Q_{j+k} - k$ (upward diagonal)
- implicit representation
- much more efficient



(© S. Russell & P. Norwig, AIMA)

Varieties of Constraints

- **Unary constraints**: involve one single variable
 - ex: ($SA \neq green$)
- **Binary constraints**: involve pairs of variables
 - ex: ($SA \neq WA$)
- **Higher-order constraints**: involve ≥ 3 variables
 - ex: cryptarithmic column constraints
 - can be represented by **constraint hypergraphs** (hypernodes represent n-ary constraints, squares in cryptarithmic example)
- **Global constraints**: involve an **arbitrary number of variables**
 - ex: $AllDiff(X_1, \dots, X_k)$
 - note: maximum domain size $\geq k$, otherwise $AllDiff()$ unsatisfiable
 - compact, specialized routines for handling them
- **Preference constraints** (aka **soft constraints**): describe preferences between/among solutions
 - ex: “I’d rather WA in red than in blue or green”
 - can often be encoded as **costs/rewards** for variables/constraints:
⇒ **solved by cost-optimization search techniques** (Constraint Optimization Problems (COPs))

Varieties of Constraints

- **Unary constraints**: involve one single variable
 - ex: ($SA \neq green$)
- **Binary constraints**: involve pairs of variables
 - ex: ($SA \neq WA$)
- **Higher-order constraints**: involve ≥ 3 variables
 - ex: cryptarithmic column constraints
 - can be represented by **constraint hypergraphs** (hypernodes represent n-ary constraints, squares in cryptarithmic example)
- **Global constraints**: involve an **arbitrary number of variables**
 - ex: $AllDiff(X_1, \dots, X_k)$
 - note: maximum domain size $\geq k$, otherwise $AllDiff()$ unsatisfiable
 - compact, specialized routines for handling them
- **Preference constraints** (aka **soft constraints**): describe preferences between/among solutions
 - ex: “I’d rather WA in red than in blue or green”
 - can often be encoded as **costs/rewards** for variables/constraints:
⇒ **solved by cost-optimization search techniques** (Constraint Optimization Problems (COPs))

Varieties of Constraints

- **Unary constraints**: involve one single variable
 - ex: ($SA \neq green$)
- **Binary constraints**: involve pairs of variables
 - ex: ($SA \neq WA$)
- **Higher-order constraints**: involve ≥ 3 variables
 - ex: **cryptarithmic column constraints**
 - can be represented by **constraint hypergraphs** (hypernodes represent n-ary constraints, squares in cryptarithmic example)
- **Global constraints**: involve an **arbitrary number of variables**
 - ex: $AllDiff(X_1, \dots, X_k)$
 - note: maximum domain size $\geq k$, otherwise $AllDiff()$ unsatisfiable
 - compact, specialized routines for handling them
- **Preference constraints** (aka **soft constraints**): describe preferences between/among solutions
 - ex: “I’d rather WA in red than in blue or green”
 - can often be encoded as **costs/rewards** for variables/constraints:
⇒ **solved by cost-optimization search techniques** (Constraint Optimization Problems (COPs))

Varieties of Constraints

- **Unary constraints:** involve one single variable
 - ex: ($SA \neq green$)
- **Binary constraints:** involve pairs of variables
 - ex: ($SA \neq WA$)
- **Higher-order constraints:** involve ≥ 3 variables
 - ex: cryptarithmic column constraints
 - can be represented by **constraint hypergraphs** (hypernodes represent n-ary constraints, squares in cryptarithmic example)
- **Global constraints:** involve an **arbitrary number of variables**
 - ex: $AllDiff(X_1, \dots, X_k)$
 - note: maximum domain size $\geq k$, otherwise $AllDiff()$ unsatisfiable
 - compact, specialized routines for handling them
- **Preference constraints** (aka **soft constraints**): describe preferences between/among solutions
 - ex: “I’d rather WA in red than in blue or green”
 - can often be encoded as **costs/rewards** for variables/constraints:
⇒ **solved by cost-optimization search techniques** (Constraint Optimization Problems (COPs))

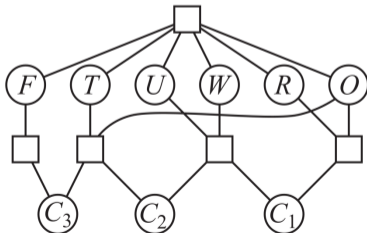
Varieties of Constraints

- **Unary constraints**: involve one single variable
 - ex: ($SA \neq green$)
- **Binary constraints**: involve pairs of variables
 - ex: ($SA \neq WA$)
- **Higher-order constraints**: involve ≥ 3 variables
 - ex: **cryptarithmic column constraints**
 - can be represented by **constraint hypergraphs** (hypernodes represent n-ary constraints, squares in cryptarithmic example)
- **Global constraints**: involve an **arbitrary number of variables**
 - ex: $AllDiff(X_1, \dots, X_k)$
 - note: maximum domain size $\geq k$, otherwise $AllDiff()$ unsatisfiable
 - compact, specialized routines for handling them
- **Preference constraints** (aka **soft constraints**): describe preferences between/among solutions
 - ex: “I’d rather WA in red than in blue or green”
 - can often be encoded as **costs/rewards** for variables/constraints:
⇒ **solved by cost-optimization search techniques** (**Constraint Optimization Problems (COPs)**)

Example: Cryptarithmic Puzzle

- Variables: F, T, U, W, R, O , plus C_1, C_2, C_3 (carry)
- Domains: $F, T, U, W, R, O \in \{0, 1, \dots, 9\}$; $C_1, C_2, C_3 \in \{0, 1\}$
- Constraints:
$$\left\{ \begin{array}{l} O + O = R + 10 \cdot C_1 \\ W + W + C_1 = U + 10 \cdot C_2 \\ T + T + C_2 = 10 \cdot C_3 + O \\ F = C_3, F \neq 0, T \neq 0 \end{array} \right\}$$
- (one) solution: $\{F=1, T=7, U=2, W=1, R=8, O=4\}$ (714+714=1428)

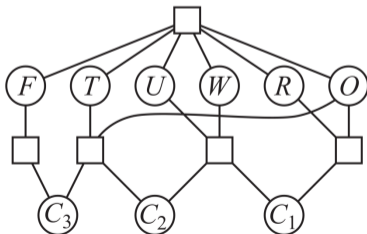
$$\begin{array}{r} T W O \\ + T W O \\ \hline F O U R \end{array}$$



Example: Cryptarithmic Puzzle

- Variables: F, T, U, W, R, O , plus C_1, C_2, C_3 (carry)
- Domains: $F, T, U, W, R, O \in \{0, 1, \dots, 9\}$; $C_1, C_2, C_3 \in \{0, 1\}$
- Constraints:
$$\left\{ \begin{array}{l} O + O = R + 10 \cdot C_1 \\ W + W + C_1 = U + 10 \cdot C_2 \\ T + T + C_2 = 10 \cdot C_3 + O \\ F = C_3, F \neq 0, T \neq 0 \end{array} \right\}$$
- (one) solution: $\{F=1, T=7, U=2, W=1, R=8, O=4\}$ (714+714=1428)

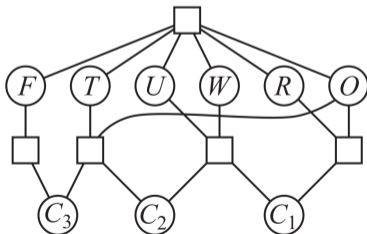
$$\begin{array}{r} T W O \\ + T W O \\ \hline F O U R \end{array}$$



Example: Cryptarithmic Puzzle

- Variables: F, T, U, W, R, O , plus C_1, C_2, C_3 (carry)
- Domains: $F, T, U, W, R, O \in \{0, 1, \dots, 9\}$; $C_1, C_2, C_3 \in \{0, 1\}$
- Constraints:
$$\left\{ \begin{array}{l} O + O = R + 10 \cdot C_1 \\ W + W + C_1 = U + 10 \cdot C_2 \\ T + T + C_2 = 10 \cdot C_3 + O \\ F = C_3, F \neq 0, T \neq 0 \end{array} \right.$$
- (one) solution: $\{F=1, T=7, U=2, W=1, R=8, O=4\}$ (714+714=1428)

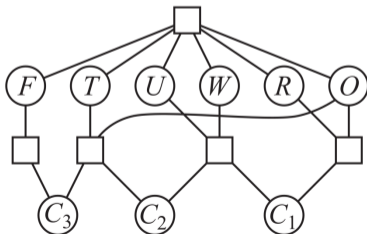
$$\begin{array}{r} T W O \\ + T W O \\ \hline F O U R \end{array}$$



Example: Cryptarithmic Puzzle

- Variables: F, T, U, W, R, O , plus C_1, C_2, C_3 (carry)
- Domains: $F, T, U, W, R, O \in \{0, 1, \dots, 9\}$; $C_1, C_2, C_3 \in \{0, 1\}$
- Constraints:
$$\left\{ \begin{array}{l} O + O = R + 10 \cdot C_1 \\ W + W + C_1 = U + 10 \cdot C_2 \\ T + T + C_2 = 10 \cdot C_3 + O \\ F = C_3, F \neq 0, T \neq 0 \end{array} \right\}$$
- (one) solution: $\{F=1, T=7, U=2, W=1, R=8, O=4\}$ (714+714=1428)

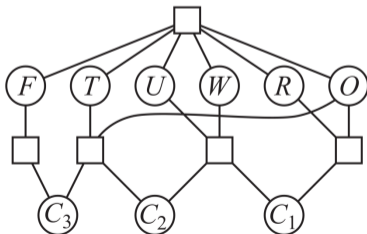
$$\begin{array}{r} T W O \\ + T W O \\ \hline F O U R \end{array}$$



Example: Cryptarithmic Puzzle

- Variables: F, T, U, W, R, O , plus C_1, C_2, C_3 (carry)
- Domains: $F, T, U, W, R, O \in \{0, 1, \dots, 9\}$; $C_1, C_2, C_3 \in \{0, 1\}$
- Constraints:
$$\left\{ \begin{array}{l} O + O = R + 10 \cdot C_1 \\ W + W + C_1 = U + 10 \cdot C_2 \\ T + T + C_2 = 10 \cdot C_3 + O \\ F = C_3, F \neq 0, T \neq 0 \end{array} \right\}$$
- (one) solution: $\{F=1, T=7, U=2, W=1, R=8, O=4\}$ (714+714=1428)

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



Example: Job-Shop Scheduling

- Scheduling the assembling of a car requires several tasks
 - ex: installing axles, installing wheels, tightening nuts, put on hubcap, inspect
- Variables X_t (for each task t): starting times of the tasks
- Domain: (bounded) integers (time units)
- Constraints:
 - Precedence: $(X_T + duration_T \leq X_{T'})$ (task T precedes task T')
 - $duration_T$ constant value (ex: $(X_{axleA} + 10 \leq X_{axleB})$)
 - Alternative precedence (combine arithmetic and logic):
 $(X_T + duration_T \leq X_{T'})$ or $(X_{T'} + duration_{T'} \leq X_T)$

Example: Job-Shop Scheduling

- Scheduling the assembling of a car requires several tasks
 - ex: installing axles, installing wheels, tightening nuts, put on hubcap, inspect
- Variables X_t (for each task t): **starting times of the tasks**
- Domain: (bounded) integers (time units)
- Constraints:
 - Precedence: $(X_T + duration_T \leq X_{T'})$ (task T precedes task T')
 - $duration_T$ constant value (ex: $(X_{axleA} + 10 \leq X_{axleB})$)
 - Alternative precedence (combine arithmetic and logic):
 $(X_T + duration_T \leq X_{T'})$ or $(X_{T'} + duration_{T'} \leq X_T)$

Example: Job-Shop Scheduling

- Scheduling the assembling of a car requires several tasks
 - ex: installing axles, installing wheels, tightening nuts, put on hubcap, inspect
- Variables X_t (for each task t): **starting times of the tasks**
- Domain: **(bounded) integers** (time units)
- Constraints:
 - Precedence: $(X_T + duration_T \leq X_{T'})$ (task T precedes task T')
 - $duration_T$ constant value (ex: $(X_{axleA} + 10 \leq X_{axleB})$)
 - Alternative precedence (combine arithmetic and logic):
 $(X_T + duration_T \leq X_{T'})$ or $(X_{T'} + duration_{T'} \leq X_T)$

Example: Job-Shop Scheduling

- Scheduling the assembling of a car requires several tasks
 - ex: installing axles, installing wheels, tightening nuts, put on hubcap, inspect
- Variables X_t (for each task t): **starting times of the tasks**
- Domain: **(bounded) integers** (time units)
- Constraints:
 - Precedence: $(X_T + duration_T \leq X_{T'})$ (task T precedes task T')
 - $duration_T$ constant value (ex: $(X_{axleA} + 10 \leq X_{axleB})$)
 - Alternative precedence (combine arithmetic and logic):
 $(X_T + duration_T \leq X_{T'})$ or $(X_{T'} + duration_{T'} \leq X_T)$

Remark

- **k-ary constraints can be transformed into sets of binary constraints**
 - hint: add enough auxiliary variables (see ex. 6.6 in AIMA book)

⇒ often CSP solvers work with binary constraints only

- In the rest of this chapter (unless specified otherwise) we assume we have only binary constraints in the CSP
- We call **neighbours** two variables sharing a binary constraint

Remark

- **k-ary constraints can be transformed into sets of binary constraints**
 - hint: add enough auxiliary variables (see ex. 6.6 in AIMA book)

⇒ often CSP solvers work with binary constraints only

- In the rest of this chapter (unless specified otherwise) we assume we have only binary constraints in the CSP
- We call **neighbours** two variables sharing a binary constraint

Remark

- **k-ary constraints can be transformed into sets of binary constraints**
 - hint: add enough auxiliary variables (see ex. 6.6 in AIMA book)

⇒ often CSP solvers work with binary constraints only

- In the rest of this chapter (unless specified otherwise) we assume we have only binary constraints in the CSP
- We call **neighbours** two variables sharing a binary constraint

Real-World CSPs

- Task-Assignment problems
 - Ex: who teaches which class?
- Timetabling problems
 - Ex: which class is offered when and where?
- Hardware configuration
 - Ex: which component is placed where? with which connections?
- Transportation scheduling
 - Ex: which van goes where?
- Factory scheduling
 - Ex: which machine/worker takes which task? in which order?
- ...

Remarks

- many real-world problems involve real/rational-valued variables
- many real-world problems involve combinatorics and logic
- many real-world problems require optimization

Real-World CSPs

- Task-Assignment problems
 - Ex: who teaches which class?
- Timetabling problems
 - Ex: which class is offered when and where?
- Hardware configuration
 - Ex: which component is placed where? with which connections?
- Transportation scheduling
 - Ex: which van goes where?
- Factory scheduling
 - Ex: which machine/worker takes which task? in which order?
- ...

Remarks

- many real-world problems involve **real/rational-valued variables**
- many real-world problems involve **combinatorics and logic**
- many real-world problems require **optimization**

- 1 Constraint Satisfaction Problems (CSPs)
- 2 Search with CSPs**
 - Inference: Constraint Propagation
 - Backtracking Search
 - Interleaving Search and Inference
 - Chronological vs. Conflict-Driven Backtracking
- 3 Local Search with CSPs
- 4 Exploiting Structure of CSPs

Search & Constraint Propagation with CSPs

- In state-space search, an algorithm can only **search**
 - **move from complete state to complete state**
- A CSPs interleaves **search** with **constraint propagation**:
 - **search**: pick a new variable assignment (and backtrack when needed)
 - does not move from complete state to complete state,
 - rather, builds a complete state by progressively extending partial ones
 - **constraint propagation** (aka inference):
 - use the constraints to reduce the set of legal candidate values for a variable
 - forces next variable assignment when candidate values are reduced to one
 - forces backtracking when candidate values are reduced to zero
- Constraint propagation can either:
 - be interleaved with search
 - be performed as a preprocessing step

Search & Constraint Propagation with CSPs

- In state-space search, an algorithm can only **search**
 - **move from complete state to complete state**
- A CSPs interleaves **search** with **constraint propagation**:
 - **search**: **pick a new variable assignment** (and backtrack when needed)
 - does not move from complete state to complete state,
 - rather, builds a complete state by progressively extending partial ones
 - **constraint propagation** (aka **inference**):
 - **use the constraints to reduce the set of legal candidate values for a variable**
 - forces next variable assignment when candidate values are reduced to one
 - forces backtracking when candidate values are reduced to zero
- Constraint propagation can either:
 - be interleaved with search
 - be performed as a preprocessing step

Search & Constraint Propagation with CSPs

- In state-space search, an algorithm can only **search**
 - **move from complete state to complete state**
- A CSPs interleaves **search** with **constraint propagation**:
 - **search**: **pick a new variable assignment** (and backtrack when needed)
 - does not move from complete state to complete state,
 - rather, builds a complete state by progressively extending partial ones
 - **constraint propagation** (aka **inference**):
 - **use the constraints to reduce the set of legal candidate values for a variable**
 - forces next variable assignment when candidate values are reduced to one
 - forces backtracking when candidate values are reduced to zero
- Constraint propagation can either:
 - be interleaved with search
 - be performed as a preprocessing step

Search & Constraint Propagation with CSPs

- In state-space search, an algorithm can only **search**
 - **move from complete state to complete state**
- A CSPs interleaves **search** with **constraint propagation**:
 - **search**: **pick a new variable assignment** (and backtrack when needed)
 - does not move from complete state to complete state,
 - rather, builds a complete state by progressively extending partial ones
 - **constraint propagation** (aka **inference**):
 - **use the constraints to reduce the set of legal candidate values for a variable**
 - forces next variable assignment when candidate values are reduced to one
 - forces backtracking when candidate values are reduced to zero
- Constraint propagation can either:
 - be interleaved with search
 - be performed as a preprocessing step

Search & Constraint Propagation with CSPs

- In state-space search, an algorithm can only **search**
 - **move from complete state to complete state**
- A CSPs interleaves **search** with **constraint propagation**:
 - **search**: **pick a new variable assignment** (and backtrack when needed)
 - does not move from complete state to complete state,
 - rather, builds a complete state by progressively extending partial ones
 - **constraint propagation** (aka **inference**):
 - **use the constraints to reduce the set of legal candidate values for a variable**
 - forces next variable assignment when candidate values are reduced to one
 - forces backtracking when candidate values are reduced to zero
- Constraint propagation can either:
 - be interleaved with search
 - be performed as a preprocessing step

- 1 Constraint Satisfaction Problems (CSPs)
- 2 **Search with CSPs**
 - **Inference: Constraint Propagation**
 - Backtracking Search
 - Interleaving Search and Inference
 - Chronological vs. Conflict-Driven Backtracking
- 3 Local Search with CSPs
- 4 Exploiting Structure of CSPs

Constraint Propagation

- Use the constraints to reduce the set of legal candidate values for variables
- Intuition: **preserve and propagate local consistency**
 - enforcing local consistency in each part of the constraint graph

⇒ inconsistent values eliminated throughout the graph
- Different types of local consistency:
 - node consistency (aka 1-consistency)
 - arc consistency (aka 2-consistency)
 - path consistency (aka 3-consistency)
 - k-consistency $k \geq 1$

Constraint Propagation

- Use the constraints to reduce the set of legal candidate values for variables
- Intuition: **preserve and propagate local consistency**
 - enforcing local consistency in each part of the constraint graph

⇒ inconsistent values eliminated throughout the graph
- Different types of local consistency:
 - node consistency (aka 1-consistency)
 - arc consistency (aka 2-consistency)
 - path consistency (aka 3-consistency)
 - k-consistency $k \geq 1$

Constraint Propagation

- Use the constraints to reduce the set of legal candidate values for variables
- Intuition: **preserve and propagate local consistency**
 - enforcing local consistency in each part of the constraint graph

⇒ inconsistent values eliminated throughout the graph
- Different types of local consistency:
 - **node consistency** (aka **1-consistency**)
 - **arc consistency** (aka **2-consistency**)
 - **path consistency** (aka **3-consistency**)
 - **k-consistency** $k \geq 1$

Node Consistency (aka 1-Consistency)

- X_i is node-consistent if all the values in the variable's domain satisfy its unary constraints
- A CSP is node-consistent if every variable is node-consistent
- Node-consistency propagation:
remove all values from the domain D_i of X_i which violate unary constraints on X_i
 - ex: if the constraint $WA \neq green$ is added to map-coloring problem then WA domain $\{red, green, blue\}$ is reduced to $\{red, blue\}$
 - ex: if the constraint $WA = green$ is added to map-coloring problem then WA domain $\{red, green, blue\}$ is reduced to $\{green\}$
- Unary constraints can be removed a priori by node consistency propagation

Node Consistency (aka 1-Consistency)

- X_i is node-consistent if all the values in the variable's domain satisfy its unary constraints
- A CSP is node-consistent if every variable is node-consistent
- Node-consistency propagation:
remove all values from the domain D_i of X_i which violate unary constraints on X_i
 - ex: if the constraint $WA \neq green$ is added to map-coloring problem then WA domain $\{red, green, blue\}$ is reduced to $\{red, blue\}$
 - ex: if the constraint $WA = green$ is added to map-coloring problem then WA domain $\{red, green, blue\}$ is reduced to $\{green\}$
- Unary constraints can be removed a priori by node consistency propagation

Node Consistency (aka 1-Consistency)

- X_i is node-consistent if all the values in the variable's domain satisfy its unary constraints
- A CSP is node-consistent if every variable is node-consistent
- Node-consistency propagation:
remove all values from the domain D_i of X_i which violate unary constraints on X_i
 - ex: if the constraint $WA \neq green$ is added to map-coloring problem then WA domain $\{red, green, blue\}$ is reduced to $\{red, blue\}$
 - ex: if the constraint $WA = green$ is added to map-coloring problem then WA domain $\{red, green, blue\}$ is reduced to $\{green\}$
- Unary constraints can be removed a priori by node consistency propagation

Node Consistency (aka 1-Consistency)

- X_i is node-consistent if all the values in the variable's domain satisfy its unary constraints
- A CSP is node-consistent if every variable is node-consistent
- Node-consistency propagation:
remove all values from the domain D_i of X_i which violate unary constraints on X_i
 - ex: if the constraint $WA \neq \text{green}$ is added to map-coloring problem then WA domain $\{\text{red}, \text{green}, \text{blue}\}$ is reduced to $\{\text{red}, \text{blue}\}$
 - ex: if the constraint $WA = \text{green}$ is added to map-coloring problem then WA domain $\{\text{red}, \text{green}, \text{blue}\}$ is reduced to $\{\text{green}\}$
- Unary constraints can be removed a priori by node consistency propagation

Arc Consistency (aka 2-Consistency)

- X_i is arc-consistent wrt. X_j iff for every value d_i of X_i in D_i exists a value d_j for X_j in D_j which satisfy all binary constraints on $\langle X_i, X_j \rangle$
 - A CSP is arc-consistent if every variable is arc consistent with every other variable
 - Forward Checking: remove values from unassigned variables which are not arc consistent with assigned variables
 - i.e., remove values which are non consistent with the assigned values of neighbour variables
 - ⇒ ensure arcs from assigned to unassigned variables are arc consistent
 - Limitation: If X loses a value, neighbors of X are not rechecked
 - Arc-consistency propagation: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
 - Idea: If X loses a value, neighbors of X are rechecked
 - ⇒ ensure all arcs are arc consistent!
 - A well-known algorithm: AC-3
 - ⇒ every arc is arc-consistent, or some variable domain is empty
 - complexity: $O(|C| \cdot |D|^3)$ worst-case
 - AC-4 is $O(|C| \cdot |D|^2)$ worst-case, but worse than AC-3 on average
- ⇒ Can be interleaved with search or used as a preprocessing step

Arc Consistency (aka 2-Consistency)

- X_i is arc-consistent wrt. X_j iff for every value d_i of X_i in D_i exists a value d_j for X_j in D_j which satisfy all binary constraints on $\langle X_i, X_j \rangle$
 - A CSP is arc-consistent if every variable is arc consistent with every other variable
 - Forward Checking: remove values from unassigned variables which are not arc consistent with assigned variables
 - i.e., remove values which are non consistent with the assigned values of neighbour variables
 - ⇒ ensure arcs from assigned to unassigned variables are arc consistent
 - Limitation: If X loses a value, neighbors of X are not rechecked
 - Arc-consistency propagation: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
 - Idea: If X loses a value, neighbors of X are rechecked
 - ⇒ ensure all arcs are arc consistent!
 - A well-known algorithm: AC-3
 - ⇒ every arc is arc-consistent, or some variable domain is empty
 - complexity: $O(|C| \cdot |D|^3)$ worst-case
 - AC-4 is $O(|C| \cdot |D|^2)$ worst-case, but worse than AC-3 on average
- ⇒ Can be interleaved with search or used as a preprocessing step

Arc Consistency (aka 2-Consistency)

- X_i is arc-consistent wrt. X_j iff for every value d_i of X_i in D_i exists a value d_j for X_j in D_j which satisfy all binary constraints on $\langle X_i, X_j \rangle$
 - A CSP is arc-consistent if every variable is arc consistent with every other variable
 - **Forward Checking**: remove values from unassigned variables which are not arc consistent with assigned variables
 - i.e., remove values which are non consistent with the assigned values of neighbour variables \Rightarrow ensure arcs from assigned to unassigned variables are arc consistent
 - Limitation: If X loses a value, neighbors of X are not rechecked
 - **Arc-consistency propagation**: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
 - Idea: If X loses a value, neighbors of X are rechecked \Rightarrow ensure all arcs are arc consistent!
 - A well-known algorithm: **AC-3**
 - \Rightarrow every arc is arc-consistent, or some variable domain is empty
 - complexity: $O(|C| \cdot |D|^3)$ worst-case
 - AC-4 is $O(|C| \cdot |D|^2)$ worst-case, but worse than AC-3 on average
- \Rightarrow
- Can be interleaved with search or used as a preprocessing step

Arc Consistency (aka 2-Consistency)

- X_i is arc-consistent wrt. X_j iff for every value d_i of X_i in D_i exists a value d_j for X_j in D_j which satisfy all binary constraints on $\langle X_i, X_j \rangle$
 - A CSP is arc-consistent if every variable is arc consistent with every other variable
 - **Forward Checking**: remove values from unassigned variables which are not arc consistent with assigned variables
 - i.e., remove values which are non consistent with the assigned values of neighbour variables
 - ⇒ ensure arcs from assigned to unassigned variables are arc consistent
 - Limitation: If X loses a value, neighbors of X are not rechecked
 - **Arc-consistency propagation**: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
 - Idea: If X loses a value, neighbors of X are rechecked
 - ⇒ ensure all arcs are arc consistent!
 - A well-known algorithm: **AC-3**
 - ⇒ every arc is arc-consistent, or some variable domain is empty
 - complexity: $O(|C| \cdot |D|^3)$ worst-case
 - AC-4 is $O(|C| \cdot |D|^2)$ worst-case, but worse than AC-3 on average
- ⇒ Can be interleaved with search or used as a preprocessing step

Arc Consistency (aka 2-Consistency)

- X_i is arc-consistent wrt. X_j iff for every value d_i of X_i in D_i exists a value d_j for X_j in D_j which satisfy all binary constraints on $\langle X_i, X_j \rangle$
- A CSP is arc-consistent if every variable is arc consistent with every other variable
- **Forward Checking**: remove values from unassigned variables which are not arc consistent with assigned variables
 - i.e., remove values which are non consistent with the assigned values of neighbour variables
 - ⇒ ensure arcs from assigned to unassigned variables are arc consistent
 - Limitation: If X loses a value, neighbors of X are not rechecked
- **Arc-consistency propagation**: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
 - Idea: If X loses a value, neighbors of X are rechecked
 - ⇒ ensure all arcs are arc consistent!
- A well-known algorithm: **AC-3**
 - ⇒ every arc is arc-consistent, or some variable domain is empty
 - complexity: $O(|C| \cdot |D|^3)$ worst-case
 - **AC-4** is $O(|C| \cdot |D|^2)$ worst-case, but worse than AC-3 on average

⇒ Can be interleaved with search or used as a preprocessing step

Arc Consistency (aka 2-Consistency)

- X_i is arc-consistent wrt. X_j iff for every value d_i of X_i in D_i exists a value d_j for X_j in D_j which satisfy all binary constraints on $\langle X_i, X_j \rangle$
 - A CSP is arc-consistent if every variable is arc consistent with every other variable
 - **Forward Checking**: remove values from unassigned variables which are not arc consistent with assigned variables
 - i.e., remove values which are non consistent with the assigned values of neighbour variables
 - ⇒ ensure arcs from assigned to unassigned variables are arc consistent
 - Limitation: If X loses a value, neighbors of X are not rechecked
 - **Arc-consistency propagation**: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
 - Idea: If X loses a value, neighbors of X are rechecked
 - ⇒ ensure all arcs are arc consistent!
 - A well-known algorithm: **AC-3**
 - ⇒ every arc is arc-consistent, or some variable domain is empty
 - complexity: $O(|C| \cdot |D|^3)$ worst-case
 - **AC-4** is $O(|C| \cdot |D|^2)$ worst-case, but worse than AC-3 on average
- ⇒ Can be interleaved with search or used as a preprocessing step

Forward Checking

- Simplest form of propagation
- Idea: **propagate information from assigned to unassigned variables**
 - pick (novel) variable assignment
 - update remaining legal values for unassigned variables
- Does not provide early detection for all failures
- Limitation: If X loses a value, neighbors of X are not rechecked!
 - ex: SA single value is incompatible with NT single value

- Can we conclude anything?
 - NT and SA cannot both be blue!
- Why didn't we detect this inconsistency yet?



Forward Checking

- Simplest form of propagation
- Idea: **propagate information from assigned to unassigned variables**
 - pick (novel) variable assignment
 - update remaining legal values for unassigned variables
- **Does not provide early detection for all failures**
- Limitation: If X loses a value, neighbors of X are not rechecked!
 - ex: SA single value is incompatible with NT single value

- Can we conclude anything?
 - NT and SA cannot both be blue!
- Why didn't we detect this inconsistency yet?



Forward Checking

- Simplest form of propagation
- Idea: **propagate information from assigned to unassigned variables**
 - pick (novel) variable assignment
 - update remaining legal values for unassigned variables
- **Does not provide early detection for all failures**
- Limitation: If X loses a value, neighbors of X are not rechecked!
 - ex: SA single value is incompatible with NT single value

- Can we conclude anything?
 - NT and SA cannot both be blue!
- Why didn't we detect this inconsistency yet?



Forward Checking

- Simplest form of propagation
- Idea: **propagate information from assigned to unassigned variables**
 - pick (novel) variable assignment
 - update remaining legal values for unassigned variables
- **Does not provide early detection for all failures**
- Limitation: If X loses a value, neighbors of X are not rechecked!
 - ex: SA single value is incompatible with NT single value

- Can we conclude anything?
 - NT and SA cannot both be blue!
- Why didn't we detect this inconsistency yet?



Forward Checking

- Simplest form of propagation
- Idea: **propagate information from assigned to unassigned variables**
 - pick (novel) variable assignment
 - update remaining legal values for unassigned variables
- **Does not provide early detection for all failures**
- Limitation: If X loses a value, neighbors of X are not rechecked!
 - ex: SA single value is incompatible with NT single value

- Can we conclude anything?
 - **NT and SA cannot both be blue!**
- Why didn't we detect this inconsistency yet?



Forward Checking

- Simplest form of propagation
- Idea: **propagate information from assigned to unassigned variables**
 - pick (novel) variable assignment
 - update remaining legal values for unassigned variables
- **Does not provide early detection for all failures**
- Limitation: If X loses a value, neighbors of X are not rechecked!
 - ex: SA single value is incompatible with NT single value

- Can we conclude anything?
 - **NT and SA cannot both be blue!**
- Why didn't we detect this inconsistency yet?



Forward Checking

- Simplest form of propagation
- Idea: **propagate information from assigned to unassigned variables**
 - pick (novel) variable assignment
 - update remaining legal values for unassigned variables
- Does not provide early detection for all failures
- Limitation: **If X loses a value, neighbors of X are not rechecked!**
 - ex: **SA single value is incompatible with NT single value**
- Can we conclude anything?
 - **NT and SA cannot both be blue!**
- Why didn't we detect this inconsistency yet?



Forward Checking Example: Sudoku

(consider *AllDiff()* as a set of binary constraints)

Apply forward checking:

- What about E6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 \implies Domain(E6)={1, 4, 7}
 - forward checking on square:
drop 1,7 \implies Domain(E6)={4}
(will be assigned to 4 at next search step,
but does not trigger other propagations)
- What about I6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 \implies Domain(I6)={1, 4, 7}
 - forward checking on square:
drop 1 \implies Domain(I6)={4, 7}
- What about A6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 \implies Domain(A6)={1, 4, 7}
- Next decisions: assign $E6 = 4$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Forward Checking Example: Sudoku

(consider $AllDiff()$ as a set of binary constraints)

Apply forward checking:

- What about E6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 \implies $Domain(E6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1,7 \implies $Domain(E6)=\{4\}$
(will be assigned to 4 at next search step,
but does not trigger other propagations)
- What about I6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 \implies $Domain(I6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1 \implies $Domain(I6)=\{4, 7\}$
- What about A6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 \implies $Domain(A6)=\{1, 4, 7\}$
- Next decisions: assign $E6 = 4$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Forward Checking Example: Sudoku

(consider $AllDiff()$ as a set of binary constraints)

Apply forward checking:

- What about E6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(E6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1,7 $\implies \text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step,
but does not trigger other propagations)
- What about I6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(I6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1 $\implies \text{Domain}(I6)=\{4, 7\}$
- What about A6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(A6)=\{1, 4, 7\}$
- Next decisions: assign $E6 = 4$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Forward Checking Example: Sudoku

(consider *AllDiff()* as a set of binary constraints)

Apply forward checking:

- What about E6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(E6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1,7 \implies $\text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step,
but does not trigger other propagations)
- What about I6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(I6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1 \implies $\text{Domain}(I6)=\{4, 7\}$
- What about A6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(A6)=\{1, 4, 7\}$
- Next decisions: assign $E6 = 4$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Forward Checking Example: Sudoku

(consider $AllDiff()$ as a set of binary constraints)

Apply forward checking:

- What about E6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(E6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1,7 $\implies \text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step,
but does not trigger other propagations)
- What about I6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(I6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1 $\implies \text{Domain}(I6)=\{4, 7\}$
- What about A6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(A6)=\{1, 4, 7\}$
- Next decisions: assign $E6 = 4$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Forward Checking Example: Sudoku

(consider $AllDiff()$ as a set of binary constraints)

Apply forward checking:

- What about E6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(E6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1,7 $\implies \text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step,
but does not trigger other propagations)
- What about I6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(I6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1 $\implies \text{Domain}(I6)=\{4, 7\}$
- What about A6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(A6)=\{1, 4, 7\}$
- Next decisions: assign $E6 = 4$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Forward Checking Example: Sudoku

(consider *AllDiff()* as a set of binary constraints)

Apply forward checking:

- What about E6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(E6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1,7 $\implies \text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step,
but does not trigger other propagations)
- What about I6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(I6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1 $\implies \text{Domain}(I6)=\{4, 7\}$
- What about A6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(A6)=\{1, 4, 7\}$
 - Next decisions: assign $E6 = 4$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Forward Checking Example: Sudoku

(consider $AllDiff()$ as a set of binary constraints)

Apply forward checking:

- What about E6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(E6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1,7 $\implies \text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step,
but does not trigger other propagations)
- What about I6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(I6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1 $\implies \text{Domain}(I6)=\{4, 7\}$
- What about A6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(A6)=\{1, 4, 7\}$
- Next decisions: assign $E6 = 4$

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Forward Checking Example: Sudoku

(consider $AllDiff()$ as a set of binary constraints)

Apply forward checking:

- What about E6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(E6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1,7 $\implies \text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step,
but does not trigger other propagations)
- What about I6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(I6)=\{1, 4, 7\}$
 - forward checking on square:
drop 1 $\implies \text{Domain}(I6)=\{4, 7\}$
- What about A6?
 - forward checking on column 6:
drop 2,3,5,6,8,9 $\implies \text{Domain}(A6)=\{1, 4, 7\}$
- **Next decisions: assign $E6 = 4$**

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7					4			8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

The Arc-Consistency Propagation Algorithm AC-3

function AC-3(*msp*) **returns** false if an inconsistency is found and true otherwise

inputs: *msp*, a binary CSP with components (X , D , C)

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

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

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

if REVISE(*msp*, X_i , X_j) **then** *// makes X_i arc-consistent wrt. X_j*

if size of $D_i = 0$ **then return** false

for each X_k **in** $X_i.\text{NEIGHBORS} - \{X_j\}$ **do**

 add (X_k, X_i) to *queue*

return true

function REVISE(*msp*, X_i , X_j) **returns** true iff we revise the domain of X_i

revised \leftarrow false

for each x **in** D_i **do**

if no value y in D_j allows (x, y) to satisfy the constraint between X_i and X_j **then**

 delete x from D_i

revised \leftarrow true

return *revised*

(© S. Russell & P. Norwig, AIMA)

note: “queue” is LIFO \implies revises first the neighbours of revised vars

Arc-Consistency Propagation AC-3: Example

- Idea: If X loses a value, neighbors of X need to be rechecked

- Ex:

- $\text{Revise}(\text{SA}, \text{NSW}) \implies D_{\text{SA}}$ unchanged

- ...

- $\text{Revise}(\text{NSW}, \text{SA}) \implies D_{\text{NSW}}$ revised

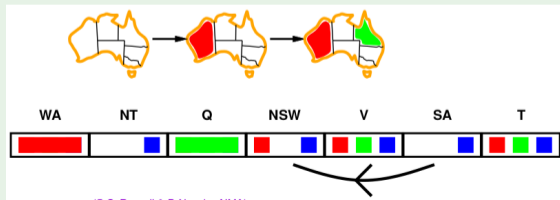
- $\text{Revise}(\text{V}, \text{NSW}) \implies D_{\text{V}}$ revised

- ...

- $\text{Revise}(\text{SA}, \text{NT}) \implies D_{\text{SA}}$ revised

- Empty domain!

\implies Arc-consistency propagation detects failure earlier than forward checking



Arc-Consistency Propagation AC-3: Example

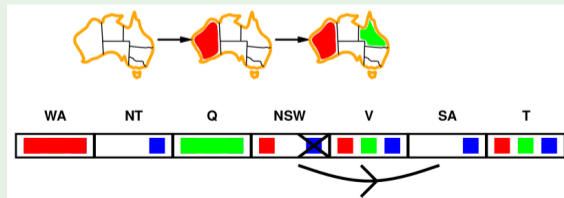
- Idea: If X loses a value, neighbors of X need to be rechecked

Ex:

- Revise(SA,NSW) $\implies D_{SA}$ unchanged
- ...
- Revise(NSW,SA) $\implies D_{NSW}$ revised
- Revise(V,NSW) $\implies D_V$ revised
- ...
- Revise(SA,NT) $\implies D_{SA}$ revised

Empty domain!

\implies Arc-consistency propagation detects failure earlier than forward checking



Arc-Consistency Propagation AC-3: Example

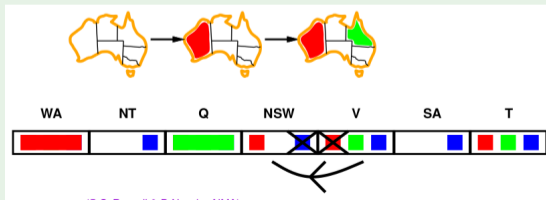
- Idea: If X loses a value, neighbors of X need to be rechecked

- Ex:

- Revise(SA,NSW) $\implies D_{SA}$ unchanged
- ...
- Revise(NSW,SA) $\implies D_{NSW}$ revised
- Revise(V,NSW) $\implies D_V$ revised
- ...
- Revise(SA,NT) $\implies D_{SA}$ revised

- Empty domain!

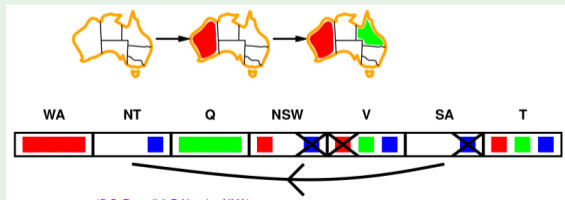
\implies Arc-consistency propagation detects failure earlier than forward checking



Arc-Consistency Propagation AC-3: Example

- Idea: If X loses a value, neighbors of X need to be rechecked
- Ex:
 - $\text{Revise}(\text{SA}, \text{NSW}) \implies D_{\text{SA}}$ unchanged
 - ...
 - $\text{Revise}(\text{NSW}, \text{SA}) \implies D_{\text{NSW}}$ revised
 - $\text{Revise}(\text{V}, \text{NSW}) \implies D_{\text{V}}$ revised
 - ...
 - $\text{Revise}(\text{SA}, \text{NT}) \implies D_{\text{SA}}$ revised
- Empty domain!

\implies Arc-consistency propagation detects failure earlier than forward checking



Arc-Consistency Propagation AC-3: Example

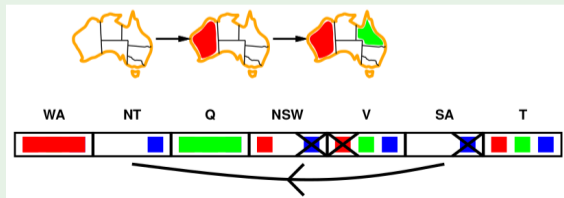
- Idea: If X loses a value, neighbors of X need to be rechecked

● Ex:

- $\text{Revise}(\text{SA}, \text{NSW}) \implies D_{\text{SA}}$ unchanged
- ...
- $\text{Revise}(\text{NSW}, \text{SA}) \implies D_{\text{NSW}}$ revised
- $\text{Revise}(\text{V}, \text{NSW}) \implies D_{\text{V}}$ revised
- ...
- $\text{Revise}(\text{SA}, \text{NT}) \implies D_{\text{SA}}$ revised

● Empty domain!

\implies Arc-consistency propagation detects failure earlier than forward checking



Remark

Notice the differences between:

(a) an assigned variable X_i , with value v_j , and

(b) an unassigned variable X_i whose domain is reduced to a singleton $\{v_j\}$:

- With (b) X_i is not (yet) assigned the value v_j
(although it will be likely assigned soon the value v_j by next search steps)
- With **Forward Checking**, (a) forces checking the domain of X_i 's unassigned neighbours wrt. X_i , whereas (b) does not
- With **ARC-Consistency Propagation**, both (a) and (b) force checking the domain of X_i 's unassigned neighbours wrt. X_i

Remark

Notice the differences between:

(a) an assigned variable X_i , with value v_j , and

(b) an unassigned variable X_i whose domain is reduced to a singleton $\{v_j\}$:

- With (b) X_i is not (yet) assigned the value v_j
(although it will be likely assigned soon the value v_j by next search steps)
- With **Forward Checking**, (a) forces checking the domain of X_i 's unassigned neighbours wrt. X_i , whereas (b) does not
- With **ARC-Consistency Propagation**, both (a) and (b) force checking the domain of X_i 's unassigned neighbours wrt. X_i

Remark

Notice the differences between:

(a) an assigned variable X_i , with value v_j , and

(b) an unassigned variable X_i whose domain is reduced to a singleton $\{v_j\}$:

- With (b) X_i is not (yet) assigned the value v_j
(although it will be likely assigned soon the value v_j by next search steps)
- With **Forward Checking**, (a) forces checking the domain of X_i 's unassigned neighbours wrt. X_i , whereas (b) does not
- With **ARC-Consistency Propagation**, both (a) and (b) force checking the domain of X_i 's unassigned neighbours wrt. X_i

Arc-consistency Propagation AC-3 Example: Sudoku [cont.]

Apply arc-consistency propagation:

- What about E6?

- arc-consistency propagation on column 6:
drop 2,3,5,6,8,9 \implies Domain(E6)={1, 4, 7}
- arc-consistency propagation on square:
drop 1,7 \implies Domain(E6)={4}
(will be assigned to 4 at next search step, but triggers next propagations)

- What about I6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,8,9 \implies Domain(I6)={1, 7}
- arc-consistency propagation on square:
drop 1 \implies Domain(I6)={7}

- What about A6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,7,8,9 \implies Domain(A6)={1}

- Next decisions: assign E6=4, I6=7, A6=1,...

- Exercise: show that AC-3 solves the whole puzzle

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Arc-consistency Propagation AC-3 Example: Sudoku [cont.]

Apply arc-consistency propagation:

- What about E6?

- arc-consistency propagation on column 6:
drop 2,3,5,6,8,9 \implies **Domain(E6)={1, 4, 7}**
- arc-consistency propagation on square:
drop 1,7 \implies **Domain(E6)={4}**
(will be assigned to 4 at next search step, but triggers next propagations)

- What about I6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,8,9 \implies **Domain(I6)={1, 7}**
- arc-consistency propagation on square:
drop 1 \implies **Domain(I6)={7}**

- What about A6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,7,8,9 \implies **Domain(A6)={1}**

- **Next decisions: assign E6=4, I6=7, A6=1,...**

- Exercise: show that AC-3 solves the whole puzzle

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Arc-consistency Propagation AC-3 Example: Sudoku [cont.]

Apply arc-consistency propagation:

- What about E6?

- arc-consistency propagation on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(E6)=\{1, 4, 7\}$
- arc-consistency propagation on square:
drop 1,7 \implies $\text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step, but triggers next propagations)

- What about I6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,8,9 \implies $\text{Domain}(I6)=\{1, 7\}$
- arc-consistency propagation on square:
drop 1 \implies $\text{Domain}(I6)=\{7\}$

- What about A6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,7,8,9 \implies $\text{Domain}(A6)=\{1\}$

- Next decisions: assign $E6=4$, $I6=7$, $A6=1$,...

- Exercise: show that AC-3 solves the whole puzzle

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Arc-consistency Propagation AC-3 Example: Sudoku [cont.]

Apply arc-consistency propagation:

- What about E6?

- arc-consistency propagation on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(E6)=\{1, 4, 7\}$
- arc-consistency propagation on square:
drop 1,7 \implies $\text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step, but triggers next propagations)

- What about I6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,8,9 \implies $\text{Domain}(I6)=\{1, 7\}$
- arc-consistency propagation on square:
drop 1 \implies $\text{Domain}(I6)=\{7\}$

- What about A6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,7,8,9 \implies $\text{Domain}(A6)=\{1\}$

- Next decisions: assign $E6=4$, $I6=7$, $A6=1$,...

- Exercise: show that AC-3 solves the whole puzzle

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Arc-consistency Propagation AC-3 Example: Sudoku [cont.]

Apply arc-consistency propagation:

- What about E6?

- arc-consistency propagation on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(E6)=\{1, 4, 7\}$
- arc-consistency propagation on square:
drop 1,7 \implies $\text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step, but triggers next propagations)

- What about I6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,8,9 \implies $\text{Domain}(I6)=\{1, 7\}$
- arc-consistency propagation on square:
drop 1 \implies $\text{Domain}(I6)=\{7\}$

- What about A6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,7,8,9 \implies $\text{Domain}(A6)=\{1\}$

- Next decisions: assign $E6=4$, $I6=7$, $A6=1$,...

- Exercise: show that AC-3 solves the whole puzzle

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Arc-consistency Propagation AC-3 Example: Sudoku [cont.]

Apply arc-consistency propagation:

- What about E6?

- arc-consistency propagation on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(E6)=\{1, 4, 7\}$
- arc-consistency propagation on square:
drop 1,7 \implies $\text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step, but triggers next propagations)

- What about I6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,8,9 \implies $\text{Domain}(I6)=\{1, 7\}$
- arc-consistency propagation on square:
drop 1 \implies $\text{Domain}(I6)=\{7\}$

- What about A6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,7,8,9 \implies $\text{Domain}(A6)=\{1\}$

- Next decisions: assign $E6=4$, $I6=7$, $A6=1$,...

- Exercise: show that AC-3 solves the whole puzzle

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Arc-consistency Propagation AC-3 Example: Sudoku [cont.]

Apply arc-consistency propagation:

- What about E6?

- arc-consistency propagation on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(E6)=\{1, 4, 7\}$
- arc-consistency propagation on square:
drop 1,7 \implies $\text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step, but triggers next propagations)

- What about I6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,8,9 \implies $\text{Domain}(I6)=\{1, 7\}$
- arc-consistency propagation on square:
drop 1 \implies $\text{Domain}(I6)=\{7\}$

- What about A6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,7,8,9 \implies $\text{Domain}(A6)=\{1\}$

- Next decisions: assign $E6=4$, $I6=7$, $A6=1$,...

- Exercise: show that AC-3 solves the whole puzzle

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Arc-consistency Propagation AC-3 Example: Sudoku [cont.]

Apply arc-consistency propagation:

- What about E6?

- arc-consistency propagation on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(E6)=\{1, 4, 7\}$
- arc-consistency propagation on square:
drop 1,7 \implies $\text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step, but triggers next propagations)

- What about I6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,8,9 \implies $\text{Domain}(I6)=\{1, 7\}$
- arc-consistency propagation on square:
drop 1 \implies $\text{Domain}(I6)=\{7\}$

- What about A6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,7,8,9 \implies $\text{Domain}(A6)=\{1\}$

- Next decisions: assign $E6=4$, $I6=7$, $A6=1$,...

- Exercise: show that AC-3 solves the whole puzzle

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

Arc-consistency Propagation AC-3 Example: Sudoku [cont.]

Apply arc-consistency propagation:

- What about E6?

- arc-consistency propagation on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(E6)=\{1, 4, 7\}$
- arc-consistency propagation on square:
drop 1,7 \implies $\text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step, but triggers next propagations)

- What about I6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,8,9 \implies $\text{Domain}(I6)=\{1, 7\}$
- arc-consistency propagation on square:
drop 1 \implies $\text{Domain}(I6)=\{7\}$

- What about A6?

- arc-consistency propagation on column 6:
drop 2,3,4,5,6,7,8,9 \implies $\text{Domain}(A6)=\{1\}$
- Next decisions: assign E6=4, I6=7, A6=1,...

- Exercise: show that AC-3 solves the whole puzzle

	1	2	3	4	5	6	7	8	9
A			3		2	1	6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7					4			8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1	7	3		

Arc-consistency Propagation AC-3 Example: Sudoku [cont.]

Apply arc-consistency propagation:

- What about E6?
 - arc-consistency propagation on column 6:
drop 2,3,5,6,8,9 \implies $\text{Domain}(E6)=\{1, 4, 7\}$
 - arc-consistency propagation on square:
drop 1,7 \implies $\text{Domain}(E6)=\{4\}$
(will be assigned to 4 at next search step, but triggers next propagations)
- What about I6?
 - arc-consistency propagation on column 6:
drop 2,3,4,5,6,8,9 \implies $\text{Domain}(I6)=\{1, 7\}$
 - arc-consistency propagation on square:
drop 1 \implies $\text{Domain}(I6)=\{7\}$
- What about A6?
 - arc-consistency propagation on column 6:
drop 2,3,4,5,6,7,8,9 \implies $\text{Domain}(A6)=\{1\}$
- Next decisions: assign E6=4, I6=7, A6=1,...
- Exercise: show that AC-3 solves the whole puzzle

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Path Consistency & K-Consistency

Path Consistency

A two-variable set $\{X_i, X_j\}$ is **path-consistent** wrt. a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$.

K-Consistency

- A CSP is k -consistent iff for any set of $k - 1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any other k -th variable
 - 1-consistency is node consistency
 - 2-consistency is arc consistency
 - 3-consistency is path consistency
- Algorithm for 3-consistency available: PC-2
 - generalization of AC-3
- Time and space complexity grow exponentially with k

Path Consistency & K-Consistency

Path Consistency

A two-variable set $\{X_i, X_j\}$ is **path-consistent** wrt. a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$.

K-Consistency

- A CSP is **k-consistent** iff for any set of $k - 1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any other k-th variable
 - 1-consistency is **node consistency**
 - 2-consistency is **arc consistency**
 - 3-consistency is **path consistency**
- Algorithm for 3-consistency available: PC-2
 - generalization of AC-3
- Time and space complexity grow exponentially with k

Path Consistency & K-Consistency

Path Consistency

A two-variable set $\{X_i, X_j\}$ is **path-consistent** wrt. a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$.

K-Consistency

- A CSP is **k-consistent** iff for any set of $k - 1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any other k-th variable
 - 1-consistency is **node consistency**
 - 2-consistency is **arc consistency**
 - 3-consistency is **path consistency**
- Algorithm for 3-consistency available: PC-2
 - generalization of AC-3
- Time and space complexity grow exponentially with k

Path Consistency & K-Consistency

Path Consistency

A two-variable set $\{X_i, X_j\}$ is **path-consistent** wrt. a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$.

K-Consistency

- A CSP is **k-consistent** iff for any set of $k - 1$ variables and for any consistent assignment to those variables, a consistent value can always be assigned to any other k-th variable
 - 1-consistency is **node consistency**
 - 2-consistency is **arc consistency**
 - 3-consistency is **path consistency**
- Algorithm for 3-consistency available: PC-2
 - generalization of AC-3
- Time and space complexity grow exponentially with k

Arc vs. Path Consistency

- Can we say anything about X1?

We can drop red & blue from D1

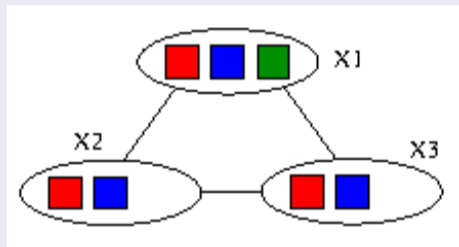
⇒ Infers the assignment $C1 = \text{green}$

- Can arc-consistency propagation reveal it?

NO!

- Can path-consistency propagation reveal it?

YES!



Arc vs. Path Consistency

- Can we say anything about X1?

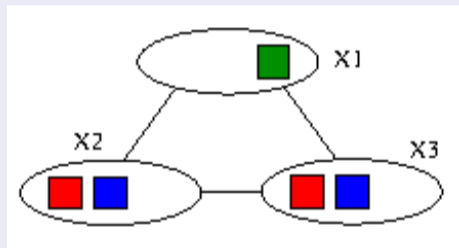
We can drop red & blue from D1

⇒ Infers the assignment $C1 = \text{green}$

- Can arc-consistency propagation reveal it?
- Can path-consistency propagation reveal it?

NO!

YES!



Arc vs. Path Consistency

- Can we say anything about X1?

We can drop red & blue from D1

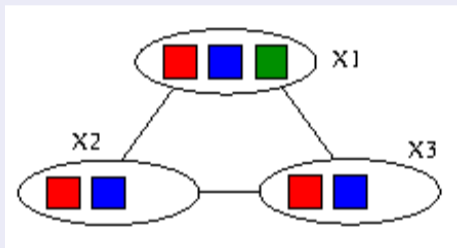
⇒ Infers the assignment $C1 = \text{green}$

- Can arc-consistency propagation reveal it?

NO!

- Can path-consistency propagation reveal it?

YES!



Arc vs. Path Consistency

- Can we say anything about X1?

We can drop red & blue from D1

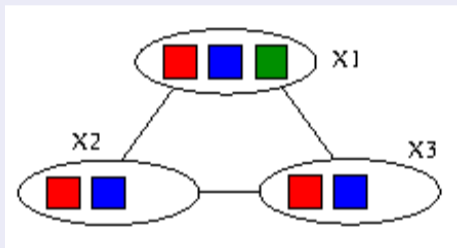
⇒ Infers the assignment $C1 = \text{green}$

- Can arc-consistency propagation reveal it?

NO!

- Can path-consistency propagation reveal it?

YES!



Arc vs. Path Consistency

- Can we say anything about X1?

We can drop red & blue from D1

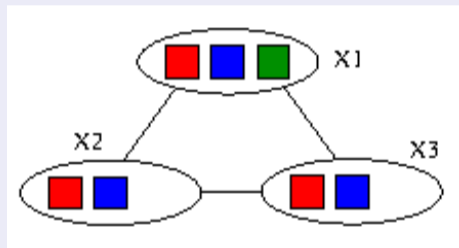
⇒ Infers the assignment $C1 = \text{green}$

- Can arc-consistency propagation reveal it?

NO!

- Can path-consistency propagation reveal it?

YES!



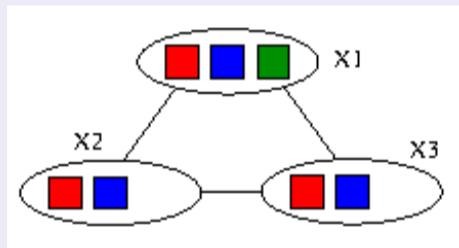
Arc vs. Path Consistency

- Can we say anything about X1?

We can drop red & blue from D1

⇒ Infers the assignment $C1 = \text{green}$

- Can arc-consistency propagation reveal it?
NO!
- Can path-consistency propagation reveal it?
YES!



Arc vs. Path Consistency [cont.]

- Can we say anything?

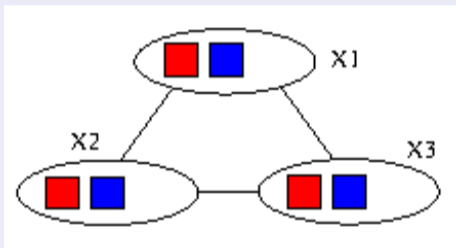
The triplet is inconsistent

- Can arc-consistency propagation reveal it?

NO!

- Can path-consistency propagation reveal it?

YES!



Arc vs. Path Consistency [cont.]

- Can we say anything?

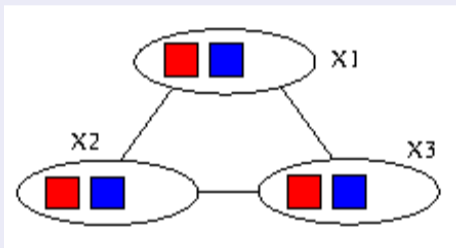
The triplet is inconsistent

- Can arc-consistency propagation reveal it?

NO!

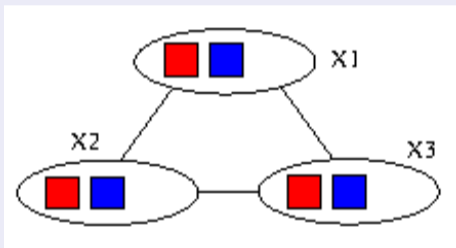
- Can path-consistency propagation reveal it?

YES!



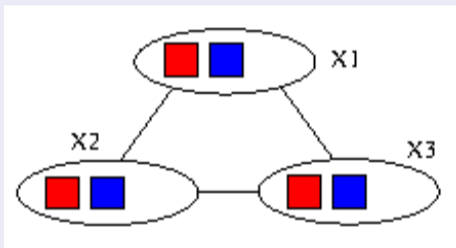
Arc vs. Path Consistency [cont.]

- Can we say anything?
The triplet is inconsistent
- Can arc-consistency propagation reveal it?
NO!
- Can path-consistency propagation reveal it?
YES!



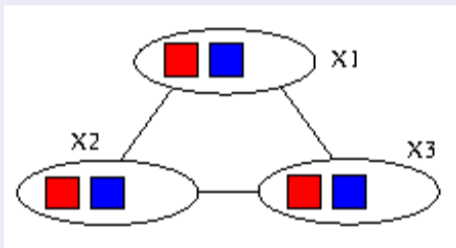
Arc vs. Path Consistency [cont.]

- Can we say anything?
The triplet is inconsistent
- Can arc-consistency propagation reveal it?
NO!
- Can path-consistency propagation reveal it?
YES!



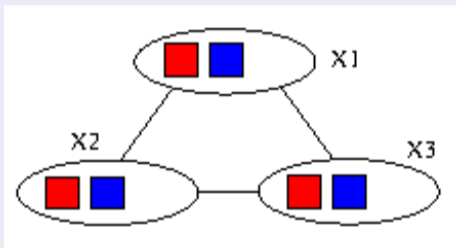
Arc vs. Path Consistency [cont.]

- Can we say anything?
The triplet is inconsistent
- Can arc-consistency propagation reveal it?
NO!
- Can path-consistency propagation reveal it?
YES!



Arc vs. Path Consistency [cont.]

- Can we say anything?
The triplet is inconsistent
- Can arc-consistency propagation reveal it?
NO!
- Can path-consistency propagation reveal it?
YES!



- 1 Constraint Satisfaction Problems (CSPs)
- 2 **Search with CSPs**
 - Inference: Constraint Propagation
 - **Backtracking Search**
 - Interleaving Search and Inference
 - Chronological vs. Conflict-Driven Backtracking
- 3 Local Search with CSPs
- 4 Exploiting Structure of CSPs

Backtracking Search: Generalities

Backtracking Search

- Basic uninformed algorithm for solving CSPs
- Idea 1: **Pick one variable at a time**
 - variable assignments are commutative \implies fix an ordering
 - ex: $\{WA = red, NT = green\}$ same as $\{NT = green, WA = red\}$ \implies can consider assignments to a single variable at each step
 - reasons on **partial assignments**
- Idea 2: **Check constraints as long as you proceed**
 - pick only values which do not conflict with previous assignments
 - requires some computation to check the constraints \implies “incremental goal test”
 - can detect if a partial assignments violate a goal
 - \implies early detection of inconsistencies \implies **pruning**
- **Backtracking search**: DFS with the two above improvements

Backtracking Search: Generalities

Backtracking Search

- Basic uninformed algorithm for solving CSPs
- Idea 1: **Pick one variable at a time**
 - variable assignments are commutative \implies fix an ordering
 - ex: $\{WA = red, NT = green\}$ same as $\{NT = green, WA = red\}$
 - \implies can consider assignments to a single variable at each step
 - reasons on **partial assignments**
- Idea 2: **Check constraints as long as you proceed**
 - pick only **values which do not conflict with previous assignments**
 - requires some computation to check the constraints
 - \implies “incremental goal test”
 - can detect if a partial assignments violate a goal
 - \implies early detection of inconsistencies \implies **pruning**
- Backtracking search: DFS with the two above improvements

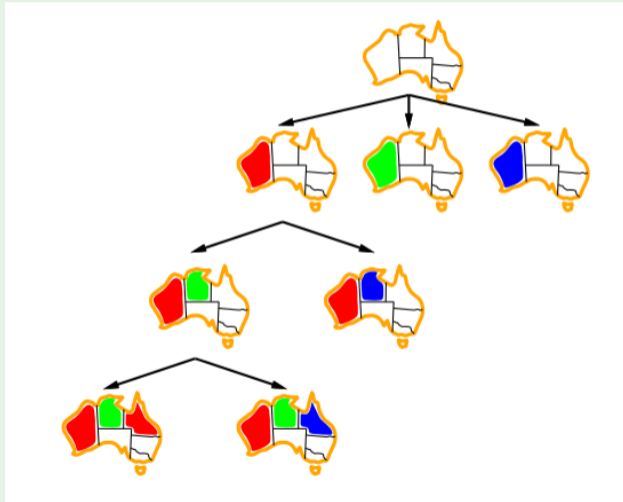
Backtracking Search: Generalities

Backtracking Search

- Basic uninformed algorithm for solving CSPs
- Idea 1: **Pick one variable at a time**
 - variable assignments are commutative \implies fix an ordering
 - ex: $\{WA = red, NT = green\}$ same as $\{NT = green, WA = red\}$
 - \implies can consider assignments to a single variable at each step
 - reasons on **partial assignments**
- Idea 2: **Check constraints as long as you proceed**
 - pick only **values which do not conflict with previous assignments**
 - requires some computation to check the constraints
 - \implies “incremental goal test”
 - can detect if a partial assignments violate a goal
 - \implies early detection of inconsistencies \implies **pruning**
- **Backtracking search**: DFS with the two above improvements

Backtracking Search: Example

(Part of) Search Tree for Map-Coloring



Backtracking Search Algorithm

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

function BACKTRACK(*csp*, *assignment*) **returns** a solution or *failure*
if *assignment* is complete **then return** *assignment*
var ← SELECT-UNASSIGNED-VARIABLE(*csp*, *assignment*)
for each *value* **in** ORDER-DOMAIN-VALUES(*csp*, *var*, *assignment*) **do**
 if *value* is consistent with *assignment* **then**
 add {*var* = *value*} to *assignment*
 inferences ← INFERENCE(*csp*, *var*, *assignment*)
 if *inferences* ≠ *failure* **then**
 add *inferences* to *csp*
 result ← BACKTRACK(*csp*, *assignment*)
 if *result* ≠ *failure* **then return** *result*
 remove *inferences* from *csp*
 remove {*var* = *value*} from *assignment*
return *failure*

Backtracking Search Algorithm [cont.]

- General-purpose algorithm for generic CSPs
- The representation of CSPs is standardized
 - ⇒ no need to provide a domain-specific initial state, action function, transition model, or goal test
- BACKTRACKING-SEARCH() keeps a single representation of a state
 - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
 - SELECT-UNASSIGNED-VARIABLE(...): which variable should be assigned next?
 - ORDER-DOMAIN-VALUES(...): in which order should its values be tried?
 - INFERENCE(...): what inferences should be performed at each step?
- We can also wonder: when an assignment violates a constraint:
 - where should we backtrack s.t. to avoid useless search?
 - how can we avoid repeating the same failure in the future?

Backtracking Search Algorithm [cont.]

- General-purpose algorithm for generic CSPs
- The representation of CSPs is standardized
 - ⇒ no need to provide a domain-specific initial state, action function, transition model, or goal test
- BACKTRACKING-SEARCH() keeps a single representation of a state
 - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
 - SELECT-UNASSIGNED-VARIABLE(...): which variable should be assigned next?
 - ORDER-DOMAIN-VALUES(...): in which order should its values be tried?
 - INFERENCE(...): what inferences should be performed at each step?
- We can also wonder: when an assignment violates a constraint:
 - where should we backtrack s.t. to avoid useless search?
 - how can we avoid repeating the same failure in the future?

Backtracking Search Algorithm [cont.]

- General-purpose algorithm for generic CSPs
- The representation of CSPs is standardized
 - ⇒ no need to provide a domain-specific initial state, action function, transition model, or goal test
- **BACKTRACKING-SEARCH()** keeps a single representation of a state
 - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
 - **SELECT-UNASSIGNED-VARIABLE(...)**: which variable should be assigned next?
 - **ORDER-DOMAIN-VALUES(...)**: in which order should its values be tried?
 - **INFERENCE(...)**: what inferences should be performed at each step?
- We can also wonder: when an assignment violates a constraint:
 - where should we backtrack s.t. to avoid useless search?
 - how can we avoid repeating the same failure in the future?

Backtracking Search Algorithm [cont.]

- General-purpose algorithm for generic CSPs
- The representation of CSPs is standardized
 - ⇒ no need to provide a domain-specific initial state, action function, transition model, or goal test
- **BACKTRACKING-SEARCH()** keeps a single representation of a state
 - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
 - **SELECT-UNASSIGNED-VARIABLE(...)**: which variable should be assigned next?
 - **ORDER-DOMAIN-VALUES(...)**: in which order should its values be tried?
 - **INFERENCE(...)**: what inferences should be performed at each step?
- We can also wonder: when an assignment violates a constraint:
 - where should we backtrack s.t. to avoid useless search?
 - how can we avoid repeating the same failure in the future?

Backtracking Search Algorithm [cont.]

- General-purpose algorithm for generic CSPs
- The representation of CSPs is standardized
 - ⇒ no need to provide a domain-specific initial state, action function, transition model, or goal test
- **BACKTRACKING-SEARCH()** keeps a single representation of a state
 - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
 - **SELECT-UNASSIGNED-VARIABLE(...)**: which variable should be assigned next?
 - **ORDER-DOMAIN-VALUES(...)**: in which order should its values be tried?
 - **INFERENCE(...)**: what inferences should be performed at each step?
- We can also wonder: when an assignment violates a constraint:
 - where should we backtrack s.t. to avoid useless search?
 - how can we avoid repeating the same failure in the future?

Backtracking Search Algorithm [cont.]

- General-purpose algorithm for generic CSPs
- The representation of CSPs is standardized
 - ⇒ no need to provide a domain-specific initial state, action function, transition model, or goal test
- **BACKTRACKING-SEARCH()** keeps a single representation of a state
 - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
 - **SELECT-UNASSIGNED-VARIABLE(...)**: which variable should be assigned next?
 - **ORDER-DOMAIN-VALUES(...)**: in which order should its values be tried?
 - **INFERENCE(...)**: what inferences should be performed at each step?
- We can also wonder: when an assignment violates a constraint:
 - where should we backtrack s.t. to avoid useless search?
 - how can we avoid repeating the same failure in the future?

Backtracking Search Algorithm [cont.]

- General-purpose algorithm for generic CSPs
- The representation of CSPs is standardized
 - ⇒ no need to provide a domain-specific initial state, action function, transition model, or goal test
- **BACKTRACKING-SEARCH()** keeps a single representation of a state
 - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
 - **SELECT-UNASSIGNED-VARIABLE(...)**: which variable should be assigned next?
 - **ORDER-DOMAIN-VALUES(...)**: in which order should its values be tried?
 - **INFERENCE(...)**: what inferences should be performed at each step?
- We can also wonder: when an assignment violates a constraint:
 - where should we backtrack s.t. to avoid useless search?
 - how can we avoid repeating the same failure in the future?

Backtracking Search Algorithm [cont.]

- General-purpose algorithm for generic CSPs
- The representation of CSPs is standardized
 - ⇒ no need to provide a domain-specific initial state, action function, transition model, or goal test
- **BACKTRACKING-SEARCH()** keeps a single representation of a state
 - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
 - **SELECT-UNASSIGNED-VARIABLE(...)**: which variable should be assigned next?
 - **ORDER-DOMAIN-VALUES(...)**: in which order should its values be tried?
 - **INFERENCE(...)**: what inferences should be performed at each step?
- We can also wonder: when an assignment violates a constraint:
 - where should we backtrack s.t. to avoid useless search?
 - how can we avoid repeating the same failure in the future?

Variable-Selection Heuristics

Minimum Remaining Values (MRV) heuristic

- Aka **most constrained variable** or **fail-first** heuristic
- MRV: **Choose the variable with the fewest legal values**
 - ⇒ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X
 - ⇒ **failure detected immediately**
 - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X
 - ⇒ **performs deterministic choices first!**
 - postpones nondeterministic steps as much as possible

Variable-Selection Heuristics

Minimum Remaining Values (MRV) heuristic

- Aka **most constrained variable** or **fail-first** heuristic
- MRV: **Choose the variable with the fewest legal values**
 - ⇒ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X
 - ⇒ **failure detected immediately**
 - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X
 - ⇒ **performs deterministic choices first!**
 - postpones nondeterministic steps as much as possible

Variable-Selection Heuristics

Minimum Remaining Values (MRV) heuristic

- Aka **most constrained variable** or **fail-first** heuristic
- MRV: **Choose the variable with the fewest legal values**
 - ⇒ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X
 - ⇒ **failure detected immediately**
 - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X
 - ⇒ **performs deterministic choices first!**
 - postpones nondeterministic steps as much as possible

Variable-Selection Heuristics

Minimum Remaining Values (MRV) heuristic

- Aka **most constrained variable** or **fail-first** heuristic
- MRV: **Choose the variable with the fewest legal values**
 - ⇒ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X
 - ⇒ **failure detected immediately**
 - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X
 - ⇒ **performs deterministic choices first!**
 - postpones nondeterministic steps as much as possible

Variable-Selection Heuristics

Minimum Remaining Values (MRV) heuristic

- Aka **most constrained variable** or **fail-first** heuristic
- MRV: **Choose the variable with the fewest legal values**
 - ⇒ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X
 - ⇒ **failure detected immediately**
 - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X
 - ⇒ **performs deterministic choices first!**
 - postpones nondeterministic steps as much as possible

• Pick (*WA = red*), (*NT = green*) ⇒ (*SA = blue*) (deterministic)

• Next? (*Q = red*)

• ...



Variable-Selection Heuristics

Minimum Remaining Values (MRV) heuristic

- Aka **most constrained variable** or **fail-first** heuristic
- MRV: **Choose the variable with the fewest legal values**
 - ⇒ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X
 - ⇒ **failure detected immediately**
 - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X
 - ⇒ **performs deterministic choices first!**
 - postpones nondeterministic steps as much as possible

- Pick (*WA = red*), (*NT = green*) ⇒ (*SA = blue*) (deterministic)
- Next? (*Q = red*)
- ...

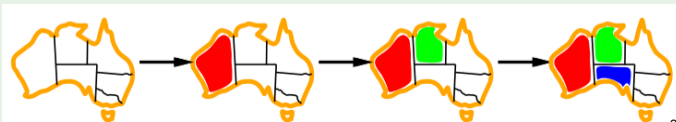


Variable-Selection Heuristics

Minimum Remaining Values (MRV) heuristic

- Aka **most constrained variable** or **fail-first** heuristic
- MRV: **Choose the variable with the fewest legal values**
 - ⇒ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X
 - ⇒ **failure detected immediately**
 - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X
 - ⇒ **performs deterministic choices first!**
 - postpones nondeterministic steps as much as possible

- Pick (*WA = red*), (*NT = green*) ⇒ (*SA = blue*) (deterministic)
- Next? (*Q = red*)
- ...



Variable-Selection Heuristics [cont.]

Degree heuristic

- Pick the variable that is involved in the largest number of constraints on other unassigned variables
 - ⇒ attempts to reduce the branching factor on future choices
 - ⇒ favours future deterministic choices
- Used as tie-breaker in combination with MRV
 - apply MRV; if ties, apply DH to these variables

Variable-Selection Heuristics [cont.]

Degree heuristic

- Pick the variable that is involved in the largest number of constraints on other unassigned variables
 - ⇒ attempts to reduce the branching factor on future choices
 - ⇒ favours future deterministic choices
- Used as tie-breaker in combination with MRV
 - apply MRV; if ties, apply DH to these variables

Variable-Selection Heuristics [cont.]

Degree heuristic

- Pick the variable that is involved in the largest number of constraints on other unassigned variables
 - ⇒ attempts to reduce the branching factor on future choices
 - ⇒ favours future deterministic choices
- Used as tie-breaker in combination with MRV
 - apply MRV; if ties, apply DH to these variables

Variable-Selection Heuristics [cont.]

Degree heuristic

- Pick the variable that is involved in the largest number of constraints on other unassigned variables
 - ⇒ attempts to reduce the branching factor on future choices
 - ⇒ favours future deterministic choices
- Used as tie-breaker in combination with MRV
 - apply MRV; if ties, apply DH to these variables

Example: MRV+DH

- Pick ($SA = blue$), ($NT = green$) \implies ($Q = red$) (deterministic)
- Next? ($NSW = green$)... (deterministic MRV+DH),



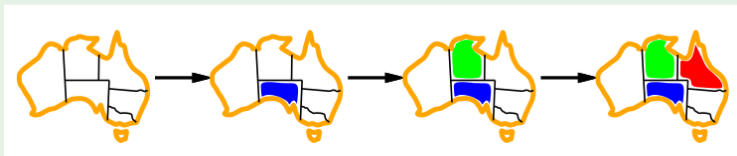
Variable-Selection Heuristics [cont.]

Degree heuristic

- Pick the variable that is involved in the largest number of constraints on other unassigned variables
 - ⇒ attempts to reduce the branching factor on future choices
 - ⇒ favours future deterministic choices
- Used as tie-breaker in combination with MRV
 - apply MRV; if ties, apply DH to these variables

Example: MRV+DH

- Pick ($SA = blue$), ($NT = green$) \implies ($Q = red$) (deterministic)
- Next? ($NSW = green$)... (deterministic MRV+DH),



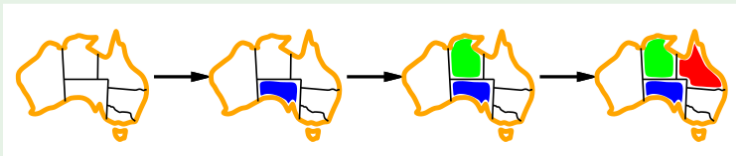
Variable-Selection Heuristics [cont.]

Degree heuristic

- Pick the variable that is involved in the largest number of constraints on other unassigned variables
 - ⇒ attempts to reduce the branching factor on future choices
 - ⇒ favours future deterministic choices
- Used as tie-breaker in combination with MRV
 - apply MRV; if ties, apply DH to these variables

Example: MRV+DH

- Pick ($SA = blue$), ($NT = green$) \implies ($Q = red$) (deterministic)
- Next? ($NSW = green$)... (deterministic MRV+DH),



Value Selection Heuristics

Least Constraining Value (LCS) heuristic

- Pick the value that rules out the fewest choices for the neighboring variables
 - ⇒ tries maximum flexibility for subsequent variable assignments
- Look for the most likely values first
 - ⇒ improve chances of finding solutions earlier
- Ex: MRV+DH+LCS allow for solving 1000-queens

Value Selection Heuristics

Least Constraining Value (LCS) heuristic

- Pick the value that rules out the fewest choices for the neighboring variables
 - ⇒ tries maximum flexibility for subsequent variable assignments
- Look for the most likely values first
 - ⇒ improve chances of finding solutions earlier
- Ex: MRV+DH+LCS allow for solving 1000-queens

Value Selection Heuristics

Least Constraining Value (LCS) heuristic

- Pick the value that rules out the fewest choices for the neighboring variables
 - ⇒ tries maximum flexibility for subsequent variable assignments
- Look for the most likely values first
 - ⇒ improve chances of finding solutions earlier
- Ex: MRV+DH+LCS allow for solving 1000-queens

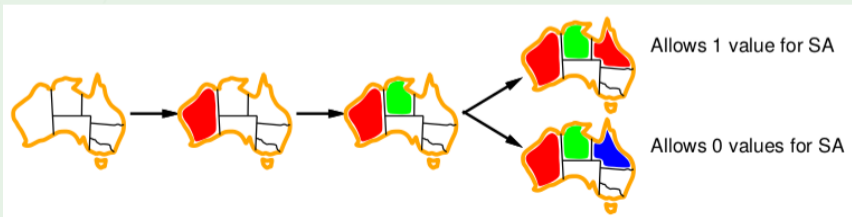
Value Selection Heuristics

Least Constraining Value (LCS) heuristic

- Pick the value that rules out the fewest choices for the neighboring variables
⇒ tries maximum flexibility for subsequent variable assignments
- Look for the most likely values first
⇒ improve chances of finding solutions earlier
- Ex: MRV+DH+LCS allow for solving 1000-queens

LCS

- Pick ($SA = red$), ($NT = green$) \implies ($Q = red$) (preferred)
- Next? ($SA=blue$)



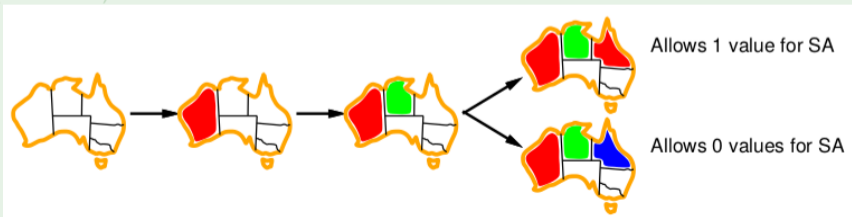
Value Selection Heuristics

Least Constraining Value (LCS) heuristic

- Pick the value that rules out the fewest choices for the neighboring variables
⇒ tries maximum flexibility for subsequent variable assignments
- Look for the most likely values first
⇒ improve chances of finding solutions earlier
- Ex: MRV+DH+LCS allow for solving 1000-queens

LCS

- Pick ($SA = red$), ($NT = green$) \implies ($Q = red$) (preferred)
- Next? ($SA=blue$)



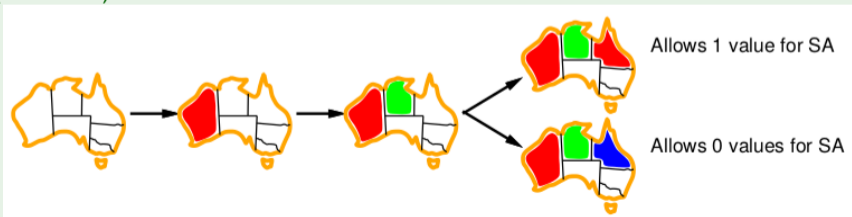
Value Selection Heuristics

Least Constraining Value (LCS) heuristic

- Pick the value that rules out the fewest choices for the neighboring variables
⇒ tries maximum flexibility for subsequent variable assignments
- Look for the most likely values first
⇒ improve chances of finding solutions earlier
- Ex: MRV+DH+LCS allow for solving 1000-queens

LCS

- Pick ($SA = red$), ($NT = green$) \implies ($Q = red$) (preferred)
- Next? ($SA=blue$)



- 1 Constraint Satisfaction Problems (CSPs)
- 2 **Search with CSPs**
 - Inference: Constraint Propagation
 - Backtracking Search
 - **Interleaving Search and Inference**
 - Chronological vs. Conflict-Driven Backtracking
- 3 Local Search with CSPs
- 4 Exploiting Structure of CSPs

Interleaving search and inference

Interleaving search and inference:

- After each choice, **infer new domain reductions on other variables**
 - detect inconsistencies earlier
 - reduce search spaces
 - may produce unary domains (deterministic steps)
⇒ returned as assignments (“inferences”)
- Tradeoff between effectiveness and efficiency
- Forward checking
 - cheap
 - ensures arc consistency of $\langle assigned, unassigned \rangle$ variable pairs only
- AC-3
 - more expensive
 - ensure arc consistency of all variable pairs
 - strategy (MAC):
 - after X_i is assigned, start AC-3 with only the arcs $\langle X_j, X_i \rangle$ s.t. X_j unassigned neighbour variables of X_i
⇒ much more effective than forward checking, more expensive

Interleaving search and inference

Interleaving search and inference:

- After each choice, **infer new domain reductions on other variables**
 - detect inconsistencies earlier
 - reduce search spaces
 - may produce unary domains (deterministic steps)
⇒ returned as assignments (“inferences”)
- Tradeoff between effectiveness and efficiency
- Forward checking
 - cheap
 - ensures arc consistency of $\langle assigned, unassigned \rangle$ variable pairs only
- AC-3
 - more expensive
 - ensure arc consistency of all variable pairs
 - strategy (MAC):
 - after X_i is assigned, start AC-3 with only the arcs $\langle X_j, X_i \rangle$ s.t. X_j unassigned neighbour variables of X_i
⇒ much more effective than forward checking, more expensive

Interleaving search and inference

Interleaving search and inference:

- After each choice, **infer new domain reductions on other variables**
 - detect inconsistencies earlier
 - reduce search spaces
 - may produce unary domains (deterministic steps)
⇒ returned as assignments (“inferences”)
- Tradeoff between effectiveness and efficiency
- **Forward checking**
 - cheap
 - ensures arc consistency of $\langle \text{assigned}, \text{unassigned} \rangle$ variable pairs only
- **AC-3**
 - more expensive
 - ensure arc consistency of all variable pairs
 - strategy (MAC):
 - after X_i is assigned, start AC-3 with only the arcs $\langle X_j, X_i \rangle$ s.t. X_j unassigned neighbour variables of X_i
⇒ much more effective than forward checking, more expensive

Interleaving search and inference

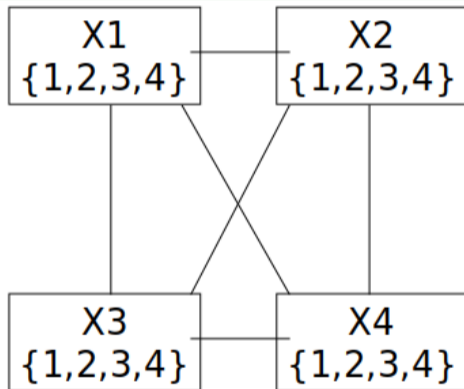
Interleaving search and inference:

- After each choice, **infer new domain reductions on other variables**
 - detect inconsistencies earlier
 - reduce search spaces
 - may produce unary domains (deterministic steps)
⇒ returned as assignments (“inferences”)
- Tradeoff between effectiveness and efficiency
- **Forward checking**
 - cheap
 - ensures arc consistency of $\langle \text{assigned}, \text{unassigned} \rangle$ variable pairs only
- **AC-3**
 - more expensive
 - ensure arc consistency of all variable pairs
 - strategy (MAC):
 - after X_i is assigned, start AC-3 with only the arcs $\langle X_j, X_i \rangle$ s.t. X_j unassigned neighbour variables of X_i
⇒ much more effective than forward checking, more expensive

Backtracking with Forward Checking: Example

4-Queens (columns)

	1	2	3	4
1		■		■
2	■		■	
3		■		■
4	■		■	

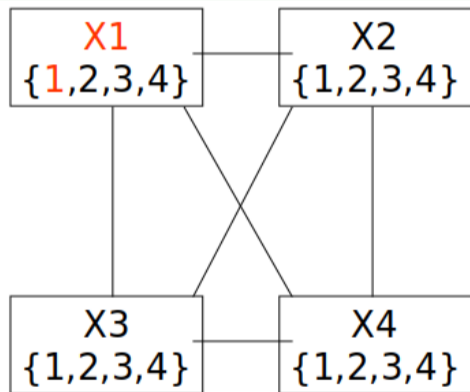
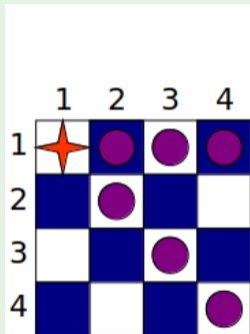


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)

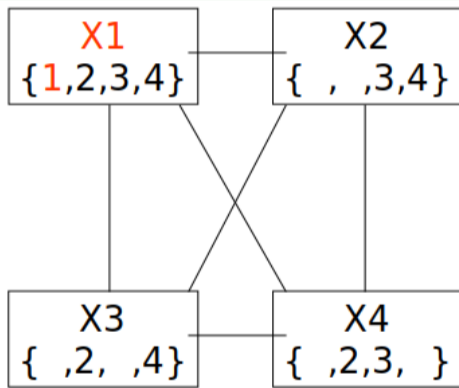
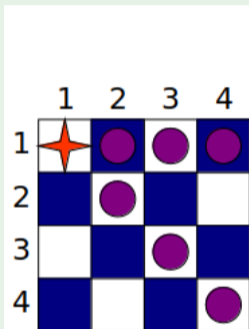


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)

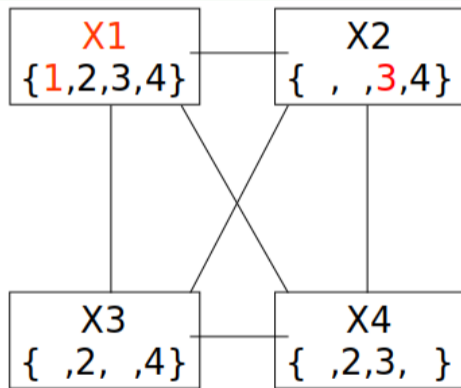
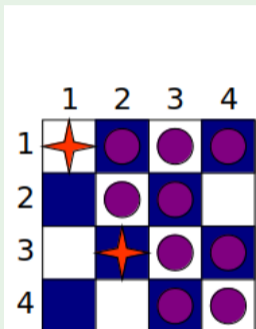


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)

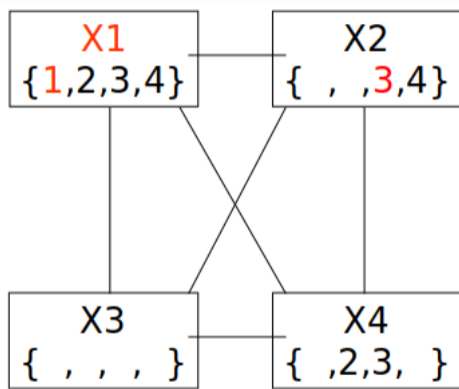
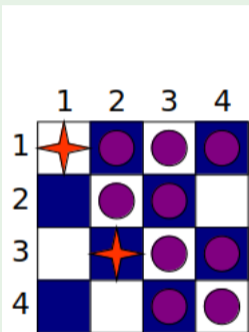


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)

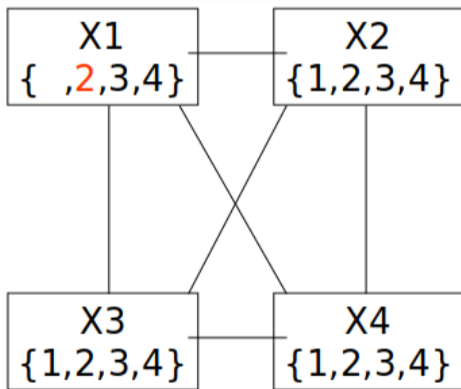
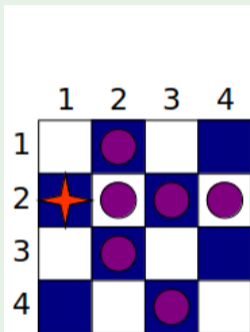


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)

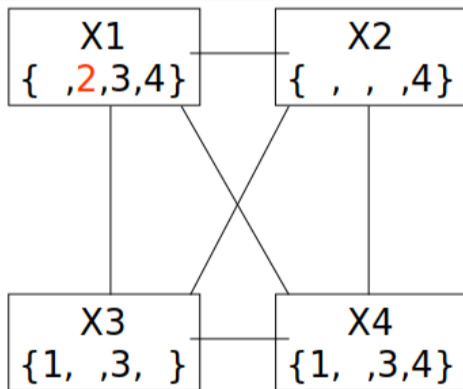
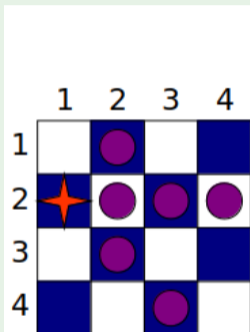


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)

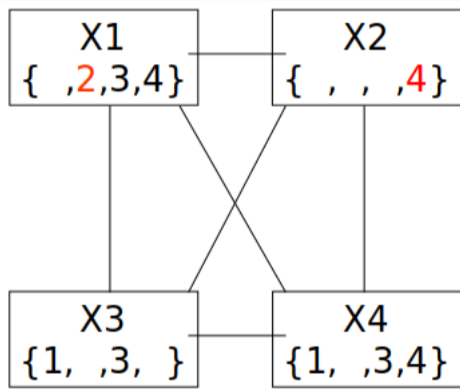
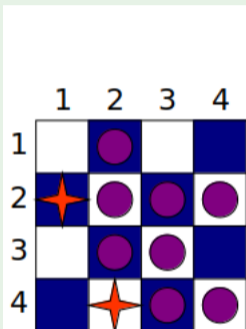


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)

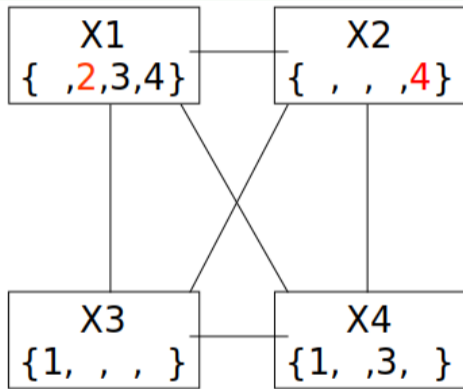
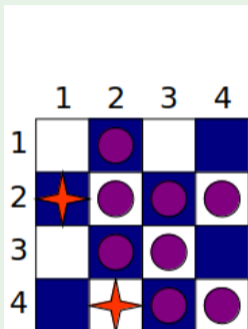


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)

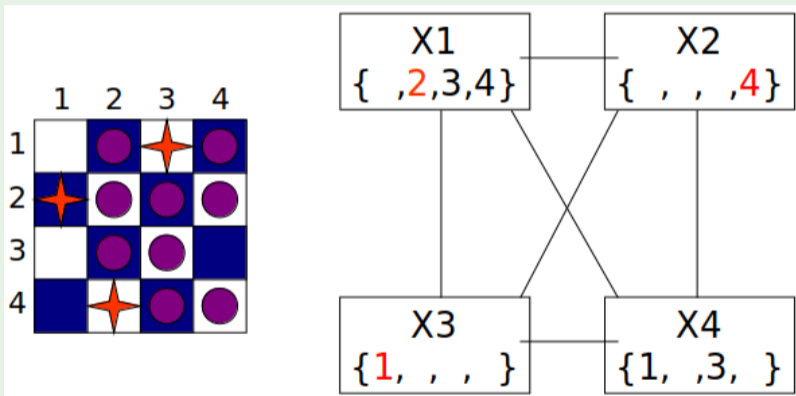


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)

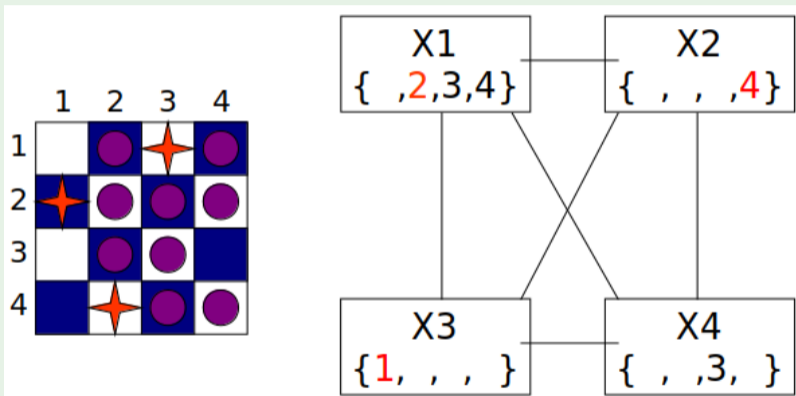


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)

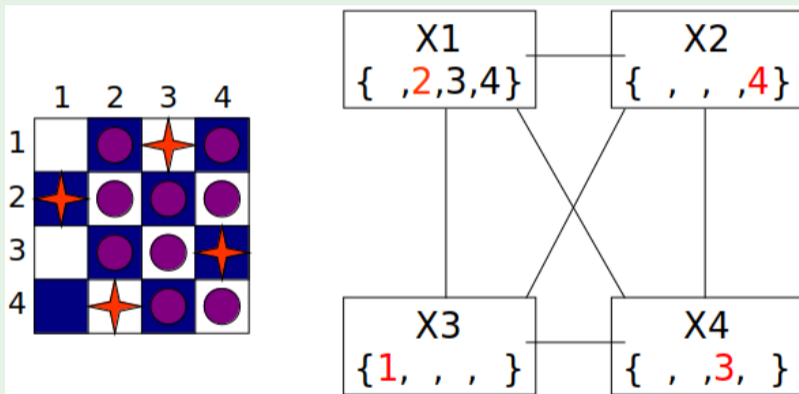


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

Backtracking with Forward Checking: Example

4-Queens (columns)



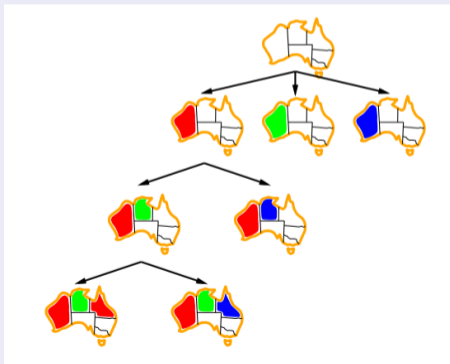
(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X_2 = 4$, $X_3 = 2$, failing and backtracking) assign $X_1 = 2$...

- 1 Constraint Satisfaction Problems (CSPs)
- 2 **Search with CSPs**
 - Inference: Constraint Propagation
 - Backtracking Search
 - Interleaving Search and Inference
 - **Chronological vs. Conflict-Driven Backtracking**
- 3 Local Search with CSPs
- 4 Exploiting Structure of CSPs

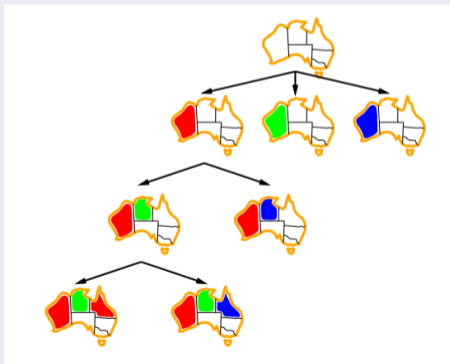
Standard Chronological Backtracking

- When a branch fails (empty domain for variable X_i):
 - 1 back up to the preceding variable which still has some untried value
 - forward-propagated assignments and rightmost choices are skipped
 - 2 try a different value for it
- Problem: lots of search wasted!



Standard Chronological Backtracking

- When a branch fails (empty domain for variable X_i):
 - 1 back up to the preceding variable which still has some untried value
 - forward-propagated assignments and rightmost choices are skipped
 - 2 try a different value for it
- Problem: lots of search wasted!



Standard Chronological Backtracking: Example

Assume variable selection order: WA,NSW,T,NT,Q,V,SA

	<i>step</i>	<i>assignment</i>	<i>[domain]</i>
	(1)	<i>pick</i> WA = <i>r</i>	[<i>rbg</i>]
	(2)	<i>pick</i> NSW = <i>r</i>	[<i>rbg</i>]
● failed branch:	(3)	<i>pick</i> T = <i>r</i>	[<i>rbg</i>]
	(4)	<i>pick</i> NT = <i>g</i>	[<i>bg</i>]
	(5)	\xrightarrow{fc} Q = <i>b</i>	[<i>b</i>]
	(6)	<i>pick</i> V = <i>b</i>	[<i>b, g</i>]
	(7)	\xrightarrow{fc} SA = {}	[]

● backtrack to (5), pick V = *g* \implies (7) again

● backtrack to (3), pick NT = *b* \xrightarrow{fc} Q = *g* \implies same subtree (6), with values switched

● backtrack to (2), pick T = *b* \implies same subtree (4)...

● backtrack to (2), pick T = *g* \implies same subtree (4)...

\implies backtrack to (1), then assign NSW another value

\implies lots of useless search on T and V values

● source of inconsistency not identified: {WA = *r*, NSW = *r*}



Standard Chronological Backtracking: Example

Assume variable selection order: WA,NSW,T,NT,Q,V,SA

	<i>step</i>	<i>assignment</i>	<i>[domain]</i>
	(1) <i>pick</i>	WA = <i>r</i>	[<i>rbg</i>]
	(2) <i>pick</i>	NSW = <i>r</i>	[<i>rbg</i>]
	(3) <i>pick</i>	T = <i>r</i>	[<i>rbg</i>]
● failed branch:	(4) <i>pick</i>	NT = <i>g</i>	[<i>bg</i>]
	(5) \xrightarrow{fc}	Q = <i>b</i>	[<i>b</i>]
	(6) <i>pick</i>	V = <i>b</i>	[<i>b, g</i>]
	(7) \xrightarrow{fc}	SA = {}	[]

● backtrack to (5), pick V = *g* \implies (7) again

● backtrack to (3), pick NT = *b* \xrightarrow{fc} Q = *g* \implies same subtree (6), with values switched

● backtrack to (2), pick T = *b* \implies same subtree (4)...

● backtrack to (2), pick T = *g* \implies same subtree (4)...

\implies backtrack to (1), then assign NSW another value

\implies lots of useless search on T and V values

● source of inconsistency not identified: {WA = *r*, NSW = *r*}



Standard Chronological Backtracking: Example

Assume variable selection order: WA,NSW,T,NT,Q,V,SA

	<i>step</i>	<i>assignment</i>	<i>[domain]</i>
	(1)	<i>pick</i> WA = <i>r</i>	[<i>rbg</i>]
	(2)	<i>pick</i> NSW = <i>r</i>	[<i>rbg</i>]
● failed branch:	(3)	<i>pick</i> T = <i>r</i>	[<i>rbg</i>]
	(4)	<i>pick</i> NT = <i>g</i>	[<i>bg</i>]
	(5)	\xrightarrow{fc} Q = <i>b</i>	[<i>b</i>]
	(6)	<i>pick</i> V = <i>b</i>	[<i>b, g</i>]
	(7)	\xrightarrow{fc} SA = {}	[]

● backtrack to (5), pick V = *g* \implies (7) again

● backtrack to (3), pick NT = *b* \xrightarrow{fc} Q = *g* \implies same subtree (6), with values switched

● backtrack to (2), pick T = *b* \implies same subtree (4)...

● backtrack to (2), pick T = *g* \implies same subtree (4)...

\implies backtrack to (1), then assign NSW another value

\implies lots of useless search on T and V values

● source of inconsistency not identified: {WA = *r*, NSW = *r*}



Standard Chronological Backtracking: Example

Assume variable selection order: WA,NSW,T,NT,Q,V,SA

	<i>step</i>	<i>assignment</i>	<i>[domain]</i>
	(1)	<i>pick</i> WA = <i>r</i>	[<i>rbg</i>]
	(2)	<i>pick</i> NSW = <i>r</i>	[<i>rbg</i>]
● failed branch:	(3)	<i>pick</i> T = <i>r</i>	[<i>rbg</i>]
	(4)	<i>pick</i> NT = <i>g</i>	[<i>bg</i>]
	(5)	\xrightarrow{fc} Q = <i>b</i>	[<i>b</i>]
	(6)	<i>pick</i> V = <i>b</i>	[<i>b, g</i>]
	(7)	\xrightarrow{fc} SA = {}	[]

● backtrack to (5), pick V = *g* \implies (7) again

● backtrack to (3), pick NT = *b* \xrightarrow{fc} Q = *g* \implies same subtree (6), with values switched

● backtrack to (2), pick T = *b* \implies same subtree (4)...

● backtrack to (2), pick T = *g* \implies same subtree (4)...

\implies backtrack to (1), then assign NSW another value

\implies lots of useless search on T and V values

● source of inconsistency not identified: {WA = *r*, NSW = *r*}



Standard Chronological Backtracking: Example

Assume variable selection order: WA,NSW,T,NT,Q,V,SA

	<i>step</i>	<i>assignment</i>	<i>[domain]</i>
	(1)	<i>pick</i> WA = <i>r</i>	[<i>rbg</i>]
	(2)	<i>pick</i> NSW = <i>r</i>	[<i>rbg</i>]
	(3)	<i>pick</i> T = <i>r</i>	[<i>rbg</i>]
● failed branch:	(4)	<i>pick</i> NT = <i>g</i>	[<i>bg</i>]
	(5)	\xrightarrow{fc} Q = <i>b</i>	[<i>b</i>]
	(6)	<i>pick</i> V = <i>b</i>	[<i>b, g</i>]
	(7)	\xrightarrow{fc} SA = {}	[]

● backtrack to (5), pick V = *g* \implies (7) again

● backtrack to (3), pick NT = *b* \xrightarrow{fc} Q = *g* \implies same subtree (6), with values switched

● backtrack to (2), pick T = *b* \implies same subtree (4)...

● backtrack to (2), pick T = *g* \implies same subtree (4)...

\implies backtrack to (1), then assign NSW another value

\implies lots of useless search on T and V values

● source of inconsistency not identified: {WA = *r*, NSW = *r*}



Standard Chronological Backtracking: Example

Assume variable selection order: WA,NSW,T,NT,Q,V,SA

	<i>step</i>	<i>assignment</i>	<i>[domain]</i>
	(1)	<i>pick</i> WA = <i>r</i>	[<i>rbg</i>]
	(2)	<i>pick</i> NSW = <i>r</i>	[<i>rbg</i>]
● failed branch:	(3)	<i>pick</i> T = <i>r</i>	[<i>rbg</i>]
	(4)	<i>pick</i> NT = <i>g</i>	[<i>bg</i>]
	(5)	\xrightarrow{fc} Q = <i>b</i>	[<i>b</i>]
	(6)	<i>pick</i> V = <i>b</i>	[<i>b, g</i>]
	(7)	\xrightarrow{fc} SA = {}	[]

● backtrack to (5), pick V = *g* \implies (7) again

● backtrack to (3), pick NT = *b* \xrightarrow{fc} Q = *g* \implies same subtree (6), with values switched

● backtrack to (2), pick T = *b* \implies same subtree (4)...

● backtrack to (2), pick T = *g* \implies same subtree (4)...

\implies backtrack to (1), then assign NSW another value

\implies lots of useless search on T and V values

● source of inconsistency not identified: {WA = *r*, NSW = *r*}



Standard Chronological Backtracking: Example

Assume variable selection order: WA,NSW,T,NT,Q,V,SA

	<i>step</i>	<i>assignment</i>	<i>[domain]</i>
	(1)	<i>pick</i> WA = <i>r</i>	[<i>rbg</i>]
	(2)	<i>pick</i> NSW = <i>r</i>	[<i>rbg</i>]
● failed branch:	(3)	<i>pick</i> T = <i>r</i>	[<i>rbg</i>]
	(4)	<i>pick</i> NT = <i>g</i>	[<i>bg</i>]
	(5)	\xrightarrow{fc} Q = <i>b</i>	[<i>b</i>]
	(6)	<i>pick</i> V = <i>b</i>	[<i>b, g</i>]
	(7)	\xrightarrow{fc} SA = {}	[]

● backtrack to (5), pick V = *g* \implies (7) again

● backtrack to (3), pick NT = *b* \xrightarrow{fc} Q = *g* \implies same subtree (6), with values switched

● backtrack to (2), pick T = *b* \implies same subtree (4)...

● backtrack to (2), pick T = *g* \implies same subtree (4)...

\implies backtrack to (1), then assign NSW another value

\implies lots of useless search on T and V values

● source of inconsistency not identified: {WA = *r*, NSW = *r*}



Standard Chronological Backtracking: Example

Assume variable selection order: WA,NSW,T,NT,Q,V,SA

	<i>step</i>	<i>assignment</i>	<i>[domain]</i>
	(1)	<i>pick</i> WA = <i>r</i>	[<i>rbg</i>]
	(2)	<i>pick</i> NSW = <i>r</i>	[<i>rbg</i>]
● failed branch:	(3)	<i>pick</i> T = <i>r</i>	[<i>rbg</i>]
	(4)	<i>pick</i> NT = <i>g</i>	[<i>bg</i>]
	(5)	\xrightarrow{fc} Q = <i>b</i>	[<i>b</i>]
	(6)	<i>pick</i> V = <i>b</i>	[<i>b, g</i>]
	(7)	\xrightarrow{fc} SA = {}	[]



● backtrack to (5), pick V = *g* \implies (7) again

● backtrack to (3), pick NT = *b* \xrightarrow{fc} Q = *g* \implies same subtree (6), with values switched

● backtrack to (2), pick T = *b* \implies same subtree (4)...

● backtrack to (2), pick T = *g* \implies same subtree (4)...

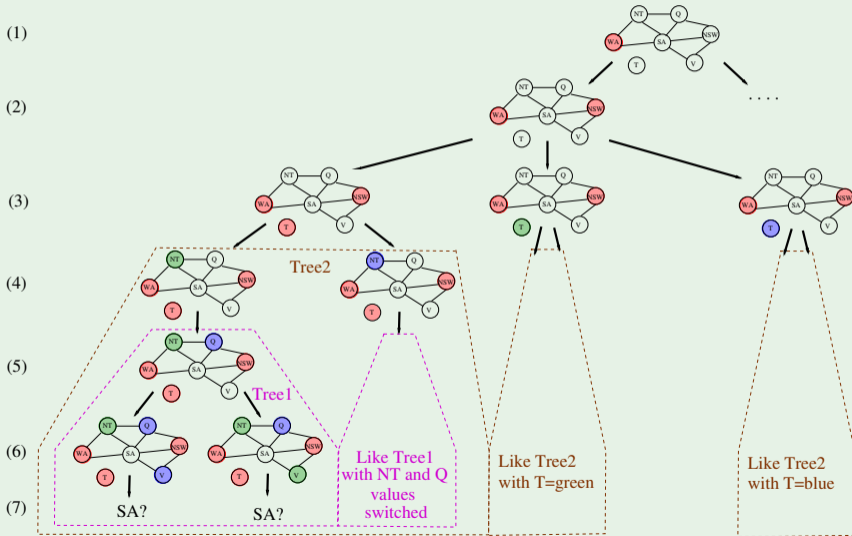
\implies backtrack to (1), then assign NSW another value

\implies lots of useless search on T and V values

● **source of inconsistency not identified:** {WA = *r*, NSW = *r*}

Standard Chronological Backtracking: Example [cont.]

Search Tree



Nogoods & Conflict Sets

- **Nogood**: subassignment which cannot be part of any solution
 - ex: $\{WA = r, NSW = r\}$ (see previous example)
- **Conflict set for X_j** (aka explanations):
(minimal) set of value assignments which caused the reduction of D_j via forward checking (i.e., in direct conflict with some values of X_j)
 - ex: $NSW=r, NT=g$ in conflict with r and g values for Q resp.
 \implies domain of Q reduced to $\{b\}$ via forward checking
 - a conflict set of an empty-domain variable is a nogood

Nogoods & Conflict Sets

- **Nogood**: subassignment which cannot be part of any solution
 - ex: $\{WA = r, NSW = r\}$ (see previous example)
- **Conflict set for X_j** (aka **explanations**):
(minimal) set of value assignments which caused the reduction of D_j via forward checking
(i.e., in direct conflict with some values of X_j)
 - ex: $NSW=r, NT=g$ in conflict with r and g values for Q resp.
 \implies domain of Q reduced to $\{b\}$ via forward checking
 - a conflict set of an empty-domain variable is a nogood

Conflict-Driven Backjumping

- Idea: When a branch fails (empty domain for variable X_i):
 - 1 identify nogood which caused the failure deterministically via forward checking
 - 2 backtrack s.t. to pop the most-recently assigned element in nogood,
 - 3 change its value

⇒ May jump much higher, lots of search saved

- Identify nogood:

- 1 take the conflict set C_i of empty-domain X_i (initial nogood)
- 2 progressively backward-substitute inside C_i every deterministic assignments $X_j = v$ with its respective conflict set C_j :

$$C_i := C_i \cup C_j \setminus \{X_j = v\}$$

until none is left

⇒ Identify the most recent decision which caused the failure due to FC by “undoing” FC steps

- Many different strategies & variants available

Conflict-Driven Backjumping

- Idea: When a branch fails (empty domain for variable X_i):
 - 1 identify nogood which caused the failure deterministically via forward checking
 - 2 backtrack s.t. to pop the most-recently assigned element in nogood,
 - 3 change its value

⇒ May jump much higher, lots of search saved

- Identify nogood:

- 1 take the conflict set C_i of empty-domain X_i (initial nogood)
- 2 progressively backward-substitute inside C_i every deterministic assignments $X_j = v$ with its respective conflict set C_j :

$$C_i := C_i \cup C_j \setminus \{X_j = v\}$$

until none is left

⇒ Identify the most recent decision which caused the failure due to FC by “undoing” FC steps

- Many different strategies & variants available

Conflict-Driven Backjumping

- Idea: When a branch fails (empty domain for variable X_i):
 - 1 identify nogood which caused the failure deterministically via forward checking
 - 2 backtrack s.t. to pop the most-recently assigned element in nogood,
 - 3 change its value

⇒ May jump much higher, lots of search saved

- Identify nogood:
 - 1 take the conflict set C_i of empty-domain X_i (initial nogood)
 - 2 progressively backward-substitute inside C_i every deterministic assignments $X_j = v$ with its respective conflict set C_j :

$$C_i := C_i \cup C_j \setminus \{X_j = v\}$$

until none is left

⇒ Identify the most recent decision which caused the failure due to FC by “undoing” FC steps

- Many different strategies & variants available

Conflict-Driven Backjumping

- Idea: When a branch fails (empty domain for variable X_i):
 - 1 identify nogood which caused the failure deterministically via forward checking
 - 2 backtrack s.t. to pop the most-recently assigned element in nogood,
 - 3 change its value

⇒ May jump much higher, lots of search saved

- Identify nogood:
 - 1 take the conflict set C_i of empty-domain X_i (initial nogood)
 - 2 progressively backward-substitute inside C_i every deterministic assignments $X_j = v$ with its respective conflict set C_j :

$$C_i := C_i \cup C_j \setminus \{X_j = v\}$$

until none is left

⇒ Identify the most recent decision which caused the failure due to FC by “undoing” FC steps

- Many different strategies & variants available

Conflict-Driven Backjumping

- Idea: When a branch fails (empty domain for variable X_i):
 - 1 identify nogood which caused the failure deterministically via forward checking
 - 2 backtrack s.t. to pop the most-recently assigned element in nogood,
 - 3 change its value

⇒ May jump much higher, lots of search saved

- Identify nogood:

- 1 take the conflict set C_i of empty-domain X_i (initial nogood)
- 2 progressively backward-substitute inside C_i every deterministic assignments $X_j = v$ with its respective conflict set C_j :

$$C_i := C_i \cup C_j \setminus \{X_j = v\}$$

until none is left

⇒ Identify the most recent decision which caused the failure due to FC by “undoing” FC steps

- Many different strategies & variants available

Conflict-Driven Backjumping: Example

- failed branch:

step	assign.	[domain]	← {conflict set}
(1) pick	WA = <i>r</i>	[<i>rbg</i>]	← {}
(2) pick	NSW = <i>r</i>	[<i>rbg</i>]	← {}
(3) pick	T = <i>r</i>	[<i>rbg</i>]	← {}
(4) pick	NT = <i>g</i>	[<i>bg</i>]	← {WA = <i>r</i> }
(5) \xrightarrow{fc}	Q = <i>b</i>	[<i>b</i>]	← {NSW = <i>r</i> , NT = <i>g</i> }
(6) pick	V = <i>b</i>	[<i>b, g</i>]	← {NSW = <i>r</i> }
(7) \xrightarrow{fc}	SA = \emptyset	[]	← {WA = <i>r</i> , NT = <i>g</i> , Q = <i>b</i> }

- backward-substitute assignments

$$\frac{\emptyset \quad (7)}{\frac{\{WA = r, NT = g, Q = b\} \quad (5)}{\{WA = r, NT = g, NSW = r\}}}$$

⇒ backtrack till (3) s.t. to pop (4), then assign $NT = b$

⇒ saves useless search on V values



Conflict-Driven Backjumping: Example

- failed branch:

<i>step</i>	<i>assign.</i>	<i>[domain]</i>	\leftarrow { <i>conflict set</i> }
(1) <i>pick</i>	WA = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(2) <i>pick</i>	NSW = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(3) <i>pick</i>	T = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(4) <i>pick</i>	NT = <i>g</i>	[<i>bg</i>]	\leftarrow {WA = <i>r</i> }
(5) \xrightarrow{fc}	Q = <i>b</i>	[<i>b</i>]	\leftarrow {NSW = <i>r</i> , NT = <i>g</i> }
(6) <i>pick</i>	V = <i>b</i>	[<i>b, g</i>]	\leftarrow {NSW = <i>r</i> }
(7) \xrightarrow{fc}	SA = \emptyset	[]	\leftarrow {WA = <i>r</i> , NT = <i>g</i> , Q = <i>b</i> }

- backward-substitute assignments

$$\frac{\emptyset \quad (7)}{\frac{\{WA = r, NT = g, Q = b\} \quad (5)}{\{WA = r, NT = g, NSW = r\}}}$$

\Rightarrow backtrack till (3) s.t. to pop (4), then assign $NT = b$

\Rightarrow saves useless search on V values



Conflict-Driven Backjumping: Example

- failed branch:

<i>step</i>	<i>assign.</i>	<i>[domain]</i>	\leftarrow { <i>conflict set</i> }
(1) <i>pick</i>	WA = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(2) <i>pick</i>	NSW = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(3) <i>pick</i>	T = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(4) <i>pick</i>	NT = <i>g</i>	[<i>bg</i>]	\leftarrow {WA = <i>r</i> }
(5) \xrightarrow{fc}	Q = <i>b</i>	[<i>b</i>]	\leftarrow {NSW = <i>r</i> , NT = <i>g</i> }
(6) <i>pick</i>	V = <i>b</i>	[<i>b, g</i>]	\leftarrow {NSW = <i>r</i> }
(7) \xrightarrow{fc}	SA = \emptyset	[]	\leftarrow {WA = <i>r</i> , NT = <i>g</i> , Q = <i>b</i> }

- backward-substitute assignments

$$\frac{\emptyset \quad (7)}{\frac{\{WA = r, NT = g, Q = b\} \quad (5)}{\{WA = r, NT = g, NSW = r\}}}$$

⇒ backtrack till (3) s.t. to pop (4), then assign *NT = b*

⇒ saves useless search on *V* values



Conflict-Driven Backjumping: Example

- failed branch:

<i>step</i>	<i>assign.</i>	<i>[domain]</i>	\leftarrow { <i>conflict set</i> }
(1) <i>pick</i>	WA = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(2) <i>pick</i>	NSW = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(3) <i>pick</i>	T = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(4) <i>pick</i>	NT = <i>g</i>	[<i>bg</i>]	\leftarrow {WA = <i>r</i> }
(5) \xrightarrow{fc}	Q = <i>b</i>	[<i>b</i>]	\leftarrow {NSW = <i>r</i> , NT = <i>g</i> }
(6) <i>pick</i>	V = <i>b</i>	[<i>b, g</i>]	\leftarrow {NSW = <i>r</i> }
(7) \xrightarrow{fc}	SA = \emptyset	[]	\leftarrow {WA = <i>r</i> , NT = <i>g</i> , Q = <i>b</i> }

- backward-substitute assignments

$$\frac{\emptyset \quad (7)}{\frac{\{WA = r, NT = g, Q = b\} \quad (5)}{\{WA = r, NT = g, NSW = r\}}}$$

⇒ backtrack till (3) s.t. to pop (4), then assign $NT = b$

⇒ saves useless search on V values



Conflict-Driven Backjumping: Example [cont.]

- new failed branch:

<i>step</i>	<i>assign.</i>	<i>[domain]</i>	\leftarrow { <i>conflict set</i> }
(1) <i>pick</i>	WA = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(2) <i>pick</i>	NSW = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(3) <i>pick</i>	T = <i>r</i>	[<i>rbg</i>]	\leftarrow {}
(4) <i>pick</i>	NT = <i>b</i>	[<i>b</i>]	\leftarrow {WA = <i>r</i> }
(5) \xrightarrow{fc}	Q = <i>g</i>	[<i>g</i>]	\leftarrow {NSW = <i>r</i> , NT = <i>b</i> }
(6) <i>pick</i>	V = <i>b</i>	[<i>b, g</i>]	\leftarrow {NSW = <i>r</i> }
(7) \xrightarrow{fc}	SA = \emptyset	[]	\leftarrow {WA = <i>r</i> , NT = <i>b</i> , Q = <i>g</i> }

- backward-substitute assignments

$$\frac{\emptyset \quad (7)}{\frac{\{WA=r, NT=b, Q=g\} \quad (5)}{\{WA=r, NT=b, NSW=r\} \quad (4)}} \quad \{WA=r, NSW=r\}$$

\Rightarrow backtrack till (1), then assign NSW another value

\Rightarrow saves useless search on T values

\Rightarrow overall, saves lots of search wrt. chronological backtracking



Conflict-Driven Backjumping: Example [cont.]

- new failed branch:

step	assign.	[domain]	← {conflict set}
(1) pick	WA = <i>r</i>	[<i>rbg</i>]	← {}
(2) pick	NSW = <i>r</i>	[<i>rbg</i>]	← {}
(3) pick	T = <i>r</i>	[<i>rbg</i>]	← {}
(4) pick	NT = <i>b</i>	[<i>b</i>]	← {WA = <i>r</i> }
(5) \xrightarrow{fc}	Q = <i>g</i>	[<i>g</i>]	← {NSW = <i>r</i> , NT = <i>b</i> }
(6) pick	V = <i>b</i>	[<i>b, g</i>]	← {NSW = <i>r</i> }
(7) \xrightarrow{fc}	SA = \emptyset	[]	← {WA = <i>r</i> , NT = <i>b</i> , Q = <i>g</i> }

- backward-substitute assignments

$$\frac{\frac{\emptyset \quad (7)}{\{WA = r, NT = b, Q = g\} \quad (5)}}{\{WA = r, NT = b, NSW = r\} \quad (4)}}{\{WA = r, NSW = r\}}$$

⇒ backtrack till (1), then assign NSW another value

⇒ saves useless search on T values

⇒ overall, saves lots of search wrt. chronological backtracking



Conflict-Driven Backjumping: Example [cont.]

- new failed branch:

step	assign.	[domain]	← {conflict set}
(1) pick	WA = <i>r</i>	[<i>rbg</i>]	← {}
(2) pick	NSW = <i>r</i>	[<i>rbg</i>]	← {}
(3) pick	T = <i>r</i>	[<i>rbg</i>]	← {}
(4) pick	NT = <i>b</i>	[<i>b</i>]	← {WA = <i>r</i> }
(5) \xrightarrow{fc}	Q = <i>g</i>	[<i>g</i>]	← {NSW = <i>r</i> , NT = <i>b</i> }
(6) pick	V = <i>b</i>	[<i>b, g</i>]	← {NSW = <i>r</i> }
(7) \xrightarrow{fc}	SA = \emptyset	[]	← {WA = <i>r</i> , NT = <i>b</i> , Q = <i>g</i> }

- backward-substitute assignments

$$\frac{\frac{\emptyset \quad (7)}{\{WA = r, NT = b, Q = g\} \quad (5)}}{\{WA = r, NT = b, NSW = r\} \quad (4)}}{\{WA = r, NSW = r\}}$$

⇒ backtrack till (1), then assign NSW another value

⇒ saves useless search on T values

⇒ overall, saves lots of search wrt. chronological backtracking



Conflict-Driven Backjumping: Example [cont.]

- new failed branch:

step	assign.	[domain]	← {conflict set}
(1) pick	WA = <i>r</i>	[<i>rbg</i>]	← {}
(2) pick	NSW = <i>r</i>	[<i>rbg</i>]	← {}
(3) pick	T = <i>r</i>	[<i>rbg</i>]	← {}
(4) pick	NT = <i>b</i>	[<i>b</i>]	← {WA = <i>r</i> }
(5) \xrightarrow{fc}	Q = <i>g</i>	[<i>g</i>]	← {NSW = <i>r</i> , NT = <i>b</i> }
(6) pick	V = <i>b</i>	[<i>b, g</i>]	← {NSW = <i>r</i> }
(7) \xrightarrow{fc}	SA = \emptyset	[]	← {WA = <i>r</i> , NT = <i>b</i> , Q = <i>g</i> }

- backward-substitute assignments

$$\frac{\frac{\emptyset \quad (7)}{\{WA=r, NT=b, Q=g\} \quad (5)}}{\{WA=r, NT=b, NSW=r\} \quad (4)}}{\{WA=r, NSW=r\}}$$

⇒ backtrack till (1), then assign NSW another value

⇒ **saves useless search on T values**

⇒ overall, saves lots of search wrt. chronological backtracking



Conflict-Driven Backjumping: Example [cont.]

- new failed branch:

step	assign.	[domain]	← {conflict set}
(1) pick	WA = <i>r</i>	[<i>rbg</i>]	← {}
(2) pick	NSW = <i>r</i>	[<i>rbg</i>]	← {}
(3) pick	T = <i>r</i>	[<i>rbg</i>]	← {}
(4) pick	NT = <i>b</i>	[<i>b</i>]	← {WA = <i>r</i> }
(5) \xrightarrow{fc}	Q = <i>g</i>	[<i>g</i>]	← {NSW = <i>r</i> , NT = <i>b</i> }
(6) pick	V = <i>b</i>	[<i>b, g</i>]	← {NSW = <i>r</i> }
(7) \xrightarrow{fc}	SA = \emptyset	[]	← {WA = <i>r</i> , NT = <i>b</i> , Q = <i>g</i> }

- backward-substitute assignments

$$\frac{\emptyset \quad (7)}{\frac{\{WA = r, NT = b, Q = g\} \quad (5)}{\{WA = r, NT = b, NSW = r\} \quad (4)} \quad (4)}{\{WA = r, NSW = r\}}$$

⇒ backtrack till (1), then assign NSW another value

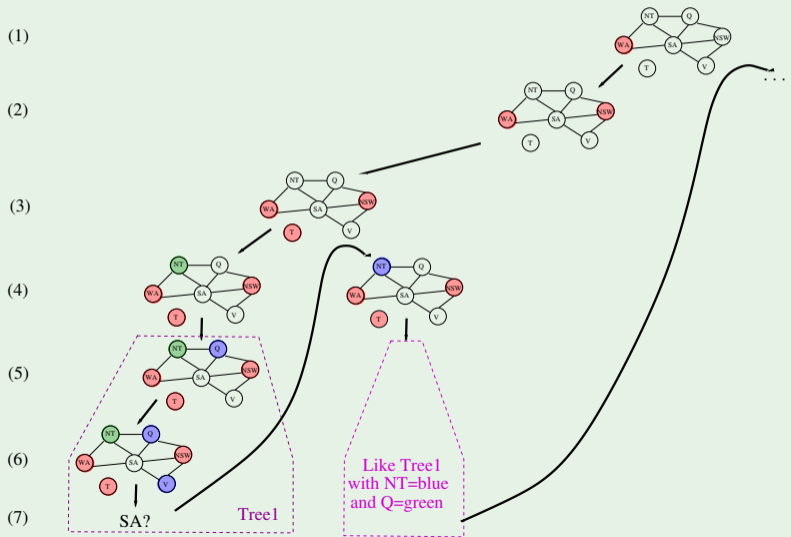
⇒ saves useless search on T values

⇒ overall, saves lots of search wrt. chronological backtracking



Conflict-Driven Backjumping: Example [cont.]

Search Tree



Learning Nogoods

- Nogood can be *learned* (stored) for future search pruning:
 - added to constraints (e.g. “ $(WA \neq r) \text{ or } (NSW \neq r)$ ”)
 - added to explicit nogood list
- As soon as assignment contains all but one element of a nogood, *drop the value of the remaining element from variable's domain*
- Example:
 - given nogood: $\{WA = r, NSW = r\}$
 - as soon as $\{NSW = r\}$ is added to assignment
 r is dropped from WA domain
- Allows for
 - early-reveal inconsistencies
 - cause further constraint propagation
- Nogoods can be learned either temporarily or permanently
 - pruning effectiveness vs. memory consumption & overhead
- Many different strategies & variants available

Learning Nogoods

- Nogood can be *learned* (stored) for future search pruning:
 - added to constraints (e.g. “(WA \neq r) or (NSW \neq r)”)
 - added to explicit nogood list
- As soon as assignment contains all but one element of a nogood, **drop the value of the remaining element from variable's domain**
- Example:
 - given nogood: {WA = r, NSW = r}
 - as soon as {NSW = r} is added to assignment
r is dropped from WA domain
- Allows for
 - early-reveal inconsistencies
 - cause further constraint propagation
- Nogoods can be learned either temporarily or permanently
 - pruning effectiveness vs. memory consumption & overhead
- Many different strategies & variants available

Learning Nogoods

- Nogood can be *learned* (stored) for future search pruning:
 - added to constraints (e.g. “ $(WA \neq r) \text{ or } (NSW \neq r)$ ”)
 - added to explicit nogood list
- As soon as assignment contains all but one element of a nogood, **drop the value of the remaining element from variable's domain**
- Example:
 - given nogood: $\{WA=r, NSW=r\}$
 - as soon as $\{NSW=r\}$ is added to assignment
 r is dropped from WA domain
- Allows for
 - early-reveal inconsistencies
 - cause further constraint propagation
- Nogoods can be learned either temporarily or permanently
 - pruning effectiveness vs. memory consumption & overhead
- Many different strategies & variants available

Learning Nogoods

- Nogood can be *learned* (stored) for future search pruning:
 - added to constraints (e.g. “ $(WA \neq r) \text{ or } (NSW \neq r)$ ”)
 - added to explicit nogood list
- As soon as assignment contains all but one element of a nogood, **drop the value of the remaining element from variable's domain**
- Example:
 - given nogood: $\{WA=r, NSW=r\}$
 - as soon as $\{NSW=r\}$ is added to assignment
 r is dropped from WA domain
- Allows for
 - early-reveal inconsistencies
 - cause further constraint propagation
- Nogoods can be learned either temporarily or permanently
 - pruning effectiveness vs. memory consumption & overhead
- Many different strategies & variants available

Learning Nogoods

- Nogood can be *learned* (stored) for future search pruning:
 - added to constraints (e.g. “ $(WA \neq r) \text{ or } (NSW \neq r)$ ”)
 - added to explicit nogood list
- As soon as assignment contains all but one element of a nogood, **drop the value of the remaining element from variable's domain**
- Example:
 - given **nogood**: $\{WA=r, NSW=r\}$
 - as soon as $\{NSW=r\}$ is added to assignment
 r is dropped from WA domain
- Allows for
 - early-reveal inconsistencies
 - cause further constraint propagation
- Nogoods can be learned either temporarily or permanently
 - pruning effectiveness vs. memory consumption & overhead
- Many different strategies & variants available

Learning Nogoods

- Nogood can be *learned* (stored) for future search pruning:
 - added to constraints (e.g. “ $(WA \neq r) \text{ or } (NSW \neq r)$ ”)
 - added to explicit nogood list
- As soon as assignment contains all but one element of a nogood, **drop the value of the remaining element from variable's domain**
- Example:
 - given **nogood**: $\{WA=r, NSW=r\}$
 - as soon as $\{NSW=r\}$ is added to assignment
 r is dropped from WA domain
- Allows for
 - early-reveal inconsistencies
 - cause further constraint propagation
- Nogoods can be learned either temporarily or permanently
 - pruning effectiveness vs. memory consumption & overhead
- Many different strategies & variants available

Outline

- 1 Constraint Satisfaction Problems (CSPs)
- 2 Search with CSPs
 - Inference: Constraint Propagation
 - Backtracking Search
 - Interleaving Search and Inference
 - Chronological vs. Conflict-Driven Backtracking
- 3 Local Search with CSPs**
- 4 Exploiting Structure of CSPs

Local Search with CSPs

- Extension of Local Search to CSPs straightforward
- Use complete-state representation (**complete assignments**)
 - allow states with unsatisfied constraints
 - “**neighbour states**” **differ for one variable value**
 - steps: reassign variable values
- **Min-conflicts heuristic** in hill-climbing:
 - Variable selection: **randomly select any conflicted variable**
 - Value selection: **select new value that results in a minimum number of conflicts with the other variables**
 - Improvement: adaptive strategies giving different weights to constraints according to their criticality
- SLC variants [see Ch. 4] apply to CSPs as well
 - random walk, simulated annealing, GAs, taboo search, ...
- ex: 1000-queens solved in few minutes

Local Search with CSPs

- Extension of Local Search to CSPs straightforward
- Use complete-state representation (**complete assignments**)
 - allow states with unsatisfied constraints
 - “**neighbour states**” differ for one variable value
 - steps: reassign variable values
- **Min-conflicts heuristic** in hill-climbing:
 - Variable selection: randomly select any conflicted variable
 - Value selection: select new value that results in a minimum number of conflicts with the other variables
 - Improvement: adaptive strategies giving different weights to constraints according to their criticality
- SLC variants [see Ch. 4] apply to CSPs as well
 - random walk, simulated annealing, GAs, taboo search, ...
- ex: 1000-queens solved in few minutes

Local Search with CSPs

- Extension of Local Search to CSPs straightforward
- Use complete-state representation (**complete assignments**)
 - allow states with unsatisfied constraints
 - “**neighbour states**” differ for one variable value
 - steps: reassign variable values
- **Min-conflicts heuristic** in hill-climbing:
 - Variable selection: **randomly select any conflicted variable**
 - Value selection: **select new value that results in a minimum number of conflicts with the other variables**
 - Improvement: adaptive strategies giving different weights to constraints according to their criticality
- SLC variants [see Ch. 4] apply to CSPs as well
 - random walk, simulated annealing, GAs, taboo search, ...
- ex: 1000-queens solved in few minutes

Local Search with CSPs

- Extension of Local Search to CSPs straightforward
- Use complete-state representation (**complete assignments**)
 - allow states with unsatisfied constraints
 - “**neighbour states**” differ for one variable value
 - steps: reassign variable values
- **Min-conflicts heuristic** in hill-climbing:
 - Variable selection: **randomly select any conflicted variable**
 - Value selection: **select new value that results in a minimum number of conflicts with the other variables**
 - Improvement: adaptive strategies giving different weights to constraints according to their criticality
- SLC variants [see Ch. 4] apply to CSPs as well
 - random walk, simulated annealing, GAs, taboo search, ...
- ex: 1000-queens solved in few minutes

Local Search with CSPs

- Extension of Local Search to CSPs straightforward
- Use complete-state representation (**complete assignments**)
 - allow states with unsatisfied constraints
 - “**neighbour states**” differ for one variable value
 - steps: reassign variable values
- **Min-conflicts heuristic** in hill-climbing:
 - Variable selection: **randomly select any conflicted variable**
 - Value selection: **select new value that results in a minimum number of conflicts with the other variables**
 - Improvement: adaptive strategies giving different weights to constraints according to their criticality
- SLC variants [see Ch. 4] apply to CSPs as well
 - random walk, simulated annealing, GAs, taboo search, ...
- ex: **1000-queens solved in few minutes**

The Min-Conflicts Heuristic

function MIN-CONFLICTS(*csp*, *max_steps*) **returns** a solution or failure

inputs: *csp*, a constraint satisfaction problem

max_steps, the number of steps allowed before giving up

current \leftarrow an initial complete assignment for *csp*

for *i* = 1 to *max_steps* **do**

if *current* is a solution for *csp* **then return** *current*

var \leftarrow a randomly chosen conflicted variable from *csp*.VARIABLES

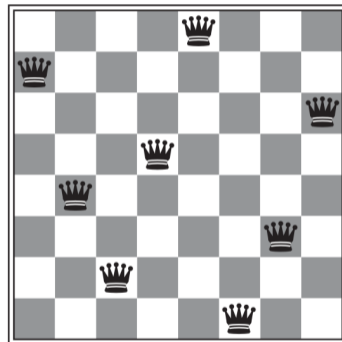
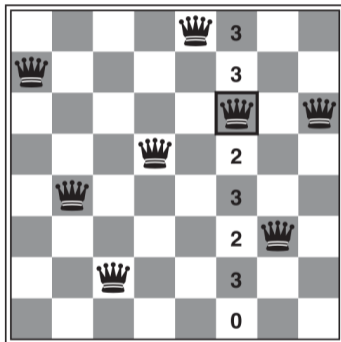
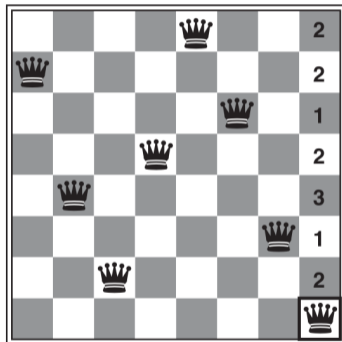
value \leftarrow the value *v* for *var* that minimizes CONFLICTS(*var*, *v*, *current*, *csp*)

 set *var* = *value* in *current*

return *failure*

The Min-Conflicts Heuristic: Example

Two steps solution of 8-Queens problem



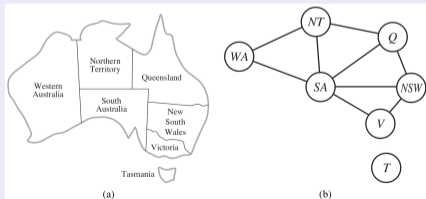
(© S. Russell & P. Norwig, AIMA)

- 1 Constraint Satisfaction Problems (CSPs)
- 2 Search with CSPs
 - Inference: Constraint Propagation
 - Backtracking Search
 - Interleaving Search and Inference
 - Chronological vs. Conflict-Driven Backtracking
- 3 Local Search with CSPs
- 4 Exploiting Structure of CSPs

Partitioning CFPs

“Divide & Conquer” CSPs

- Idea (when applicable): **Partition a CSP into independent CSPs**
 - identify **strongly-connected components** in constraint graph
 - e.g. by **Tarjan’s algorithms** (linear!)
- Ex: **Tasmania and mainland are independent subproblems**
- E.g. partition n -variable CSP into n/c CSPs with c variables each:
 - from d^n to $n/c \cdot d^c$ steps in worst-case
 - if $n = 80, d = 2, c = 20$, then from $2^{80} \approx 10^{24}$ to $4 \cdot 2^{20} \approx 4 \cdot 10^6$
⇒ from **4 billion years** to **0.4 secs** at 10million steps/sec

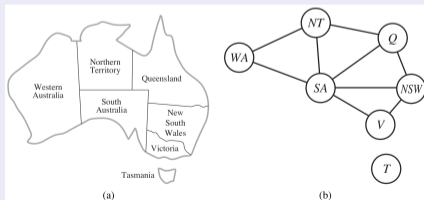


(© S. Russell & P. Norwig, AIMA)

Partitioning CFPs

“Divide & Conquer” CSPs

- Idea (when applicable): **Partition a CSP into independent CSPs**
 - identify **strongly-connected components** in constraint graph
 - e.g. by **Tarjan's algorithms** (linear!)
- Ex: **Tasmania and mainland are independent subproblems**
- E.g. partition n -variable CSP into n/c CSPs with c variables each:
 - from d^n to $n/c \cdot d^c$ steps in worst-case
 - if $n = 80, d = 2, c = 20$, then from $2^{80} \approx 10^{24}$ to $4 \cdot 2^{20} \approx 4 \cdot 10^6$
 \implies from **4 billion years** to **0.4 secs** at 10million steps/sec

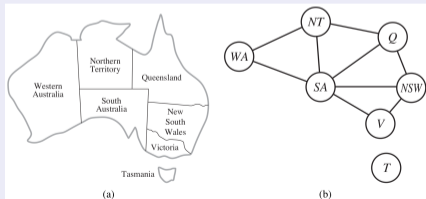


(© S. Russell & P. Norwig, AIMA)

Partitioning CFPs

“Divide & Conquer” CSPs

- Idea (when applicable): **Partition a CSP into independent CSPs**
 - identify **strongly-connected components** in constraint graph
 - e.g. by **Tarjan’s algorithms** (linear!)
- Ex: **Tasmania and mainland are independent subproblems**
- E.g. partition n -variable CSP into n/c CSPs with c variables each:
 - from d^n to $n/c \cdot d^c$ steps in worst-case
 - if $n = 80, d = 2, c = 20$, then from $2^{80} \approx 10^{24}$ to $4 \cdot 2^{20} \approx 4 \cdot 10^6$
 \implies from **4 billion years** to **0.4 secs** at 10million steps/sec



(© S. Russell & P. Norwig, AIMA)

Solving Tree-structured CSPs

Theorem:

- If the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time in worst case
 - general CSPs can be solved $O(d^n)$ time worst-case

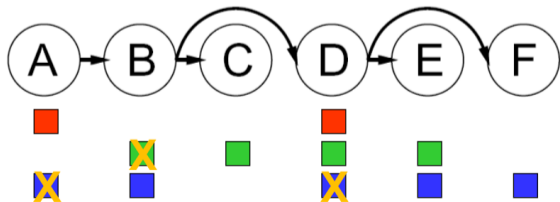
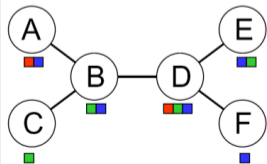
Solving Tree-structured CSPs

Theorem:

- If the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time in worst case
 - general CSPs can be solved $O(d^n)$ time worst-case

Algorithm

- 1 Choose a variable as root, order variables from root to leaves
- 2 For $j \in n..2$ apply MAKEARCCONSISTENT(PARENT(X_j), X_j)
- 3 (If no empty domain, then) For $j \in 2..n$, assign X_j consistently with PARENT(X_j)



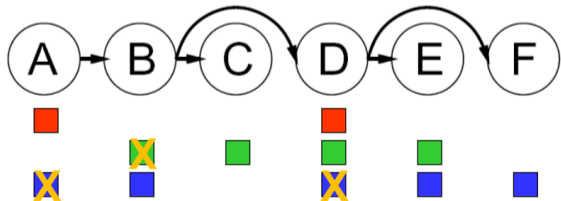
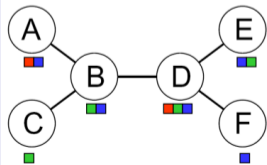
Solving Tree-structured CSPs

Theorem:

- If the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time in worst case
 - general CSPs can be solved $O(d^n)$ time worst-case

Algorithm

- 1 Choose a variable as root, order variables from root to leaves
- 2 For $j \in n..2$ apply MAKEARCCONSISTENT(PARENT(X_j), X_j)
- 3 (If no empty domain, then) For $j \in 2..n$, assign X_j consistently with PARENT(X_j)



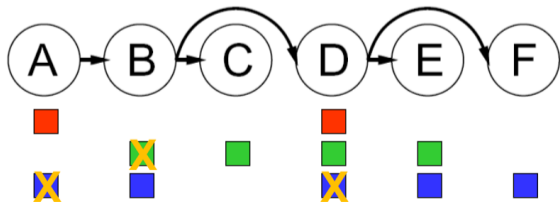
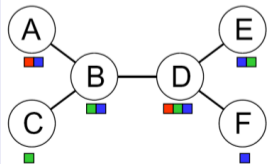
Solving Tree-structured CSPs

Theorem:

- If the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time in worst case
 - general CSPs can be solved $O(d^n)$ time worst-case

Algorithm

- 1 Choose a variable as root, order variables from root to leaves
- 2 For $j \in n..2$ apply MAKEARCCONSISTENT(PARENT(X_j), X_j)
- 3 (If no empty domain, then) For $j \in 2..n$, assign X_j consistently with PARENT(X_j)



Solving Tree-structured CSPs [cont.]

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or failure

inputs: *csp*, a CSP with components X , D , C

$n \leftarrow$ number of variables in X

assignment \leftarrow an empty assignment

root \leftarrow any variable in X

$X \leftarrow$ TOPOLOGICALSORT(X , *root*)

for $j = n$ **down to** 2 **do**

 MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)

if it cannot be made consistent **then return** *failure*

for $i = 1$ **to** n **do**

assignment[X_i] \leftarrow any consistent value from D_i

if there is no consistent value **then return** *failure*

return *assignment*

Solving Nearly Tree-Structured CSPs

Cutset Conditioning

- 1 Identify a (small) **cycle cutset** S : a set of variables s.t. the remaining constraint graph is a tree
 - finding smallest cycle cutset is NP-hard
 - fast approximated techniques known
- 2 For each possible consistent assignment to the variables in S
 - a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S
 - b) apply the tree-structured CSP algorithm
- 3 If $c \stackrel{\text{def}}{=} |S|$, then runtime is $O(d^c \cdot (n - c)d^2)$
 \implies much smaller than d^n if c small

Solving Nearly Tree-Structured CSPs

Cutset Conditioning

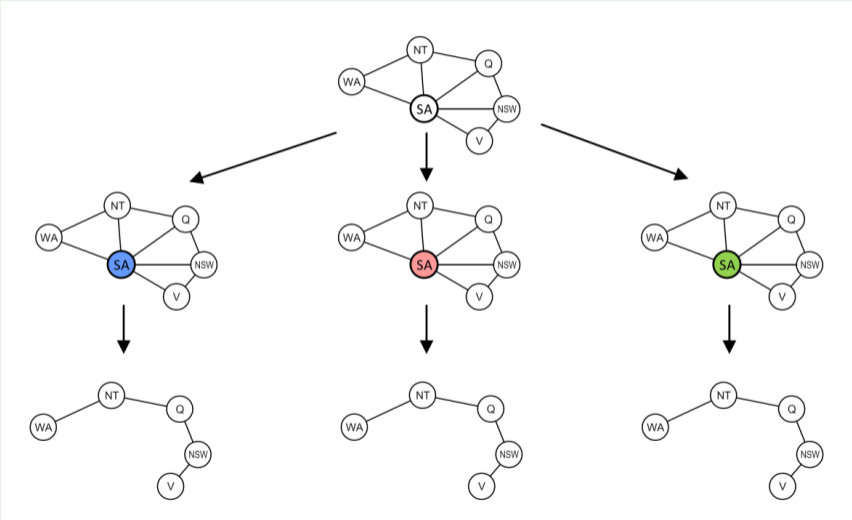
- 1 Identify a (small) **cycle cutset** S : a set of variables s.t. the remaining constraint graph is a tree
 - finding smallest cycle cutset is NP-hard
 - fast approximated techniques known
- 2 For each possible consistent assignment to the variables in S
 - a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S
 - b) apply the tree-structured CSP algorithm
- 3 If $c \stackrel{\text{def}}{=} |S|$, then runtime is $O(d^c \cdot (n - c)d^2)$
 \implies much smaller than d^n if c small

Solving Nearly Tree-Structured CSPs

Cutset Conditioning

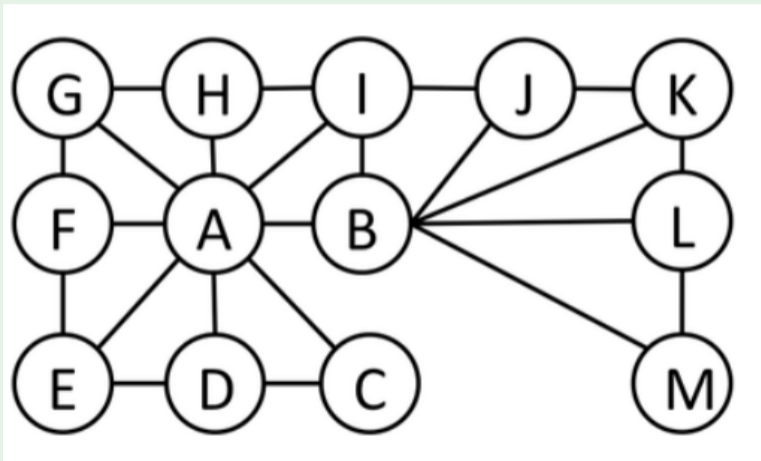
- 1 Identify a (small) **cycle cutset** S : a set of variables s.t. the remaining constraint graph is a tree
 - finding smallest cycle cutset is NP-hard
 - fast approximated techniques known
- 2 For each possible consistent assignment to the variables in S
 - a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S
 - b) apply the tree-structured CSP algorithm
- 3 If $c \stackrel{\text{def}}{=} |S|$, then runtime is $O(d^c \cdot (n - c)d^2)$
 \implies much smaller than d^n if c small

Cutset Conditioning: Example



Exercise

- Solve the following 3-coloring problem by Cutset Conditioning



Breaking Value Symmetry

- **Value symmetry**: if domain size is n and no unary constraints
 - every solution has $n!$ solutions obtained by permuting value names
 - ex: 3-coloring, $3! = 6$ permutations for every solutions
- **Symmetry Breaking**: add symmetry-breaking constraints s.t. only one of the $n!$ solution is possible
 - ⇒ reduce search space by $n!$ factor
- Add **value-ordering constraints** on n variables:
 - give an ordering of values (ex: $r < b < g$)
 - impose an ordering on the values of n variables s.t. $x_i \neq x_j$
(ex: $WA < NT < SA$)
 - ⇒ only one solution out of $n!$

Breaking Value Symmetry

- **Value symmetry**: if domain size is n and no unary constraints
 - every solution has $n!$ solutions obtained by permuting value names
 - ex: 3-coloring, $3! = 6$ permutations for every solutions
- **Symmetry Breaking**: add **symmetry-breaking constraints** s.t. only one of the $n!$ solution is possible
 - ⇒ reduce search space by $n!$ factor
- Add **value-ordering constraints** on n variables:
 - give an ordering of values (ex: $r < b < g$)
 - impose an ordering on the values of n variables s.t. $x_i \neq x_j$
(ex: $WA < NT < SA$)
 - ⇒ only one solution out of $n!$

Breaking Value Symmetry

- **Value symmetry**: if domain size is n and no unary constraints
 - every solution has $n!$ solutions obtained by permuting value names
 - ex: 3-coloring, $3! = 6$ permutations for every solutions
- **Symmetry Breaking**: add **symmetry-breaking constraints** s.t. only one of the $n!$ solution is possible
 - ⇒ reduce search space by $n!$ factor
- Add **value-ordering constraints** on n variables:
 - give an ordering of values (ex: $r < b < g$)
 - impose an ordering on the values of n variables s.t. $x_i \neq x_j$
(ex: $WA < NT < SA$)
 - ⇒ only one solution out of $n!$