Fundamentals of Artificial Intelligence
Chapter 06: **Constraint Satisfaction Problems**

Roberto Sebastiani

DISI, Università di Trento, Italy – roberto.sebastiani@unitn.it
http://disi.unitn.it/rseba/DIDATTICA/fai_2021/

Teaching assistant: Mauro Dragoni – dragoni@fbk.eu
http://www.maurodragoni.com/teaching/fai/

M.S. Course “Artificial Intelligence Systems”, academic year 2021-2022

Last update: Friday 29th October, 2021, 12:05

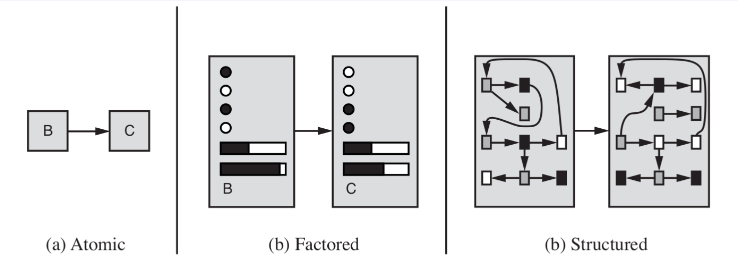
Outline

Outline

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



(© S. Russell & P. Norwig, AIMA)

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

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 $X \stackrel{\text{def}}{=} \{X_1, ..., X_n\}$
  - a set of (non-empty) domains $D \stackrel{\text{def}}{=} \{D_1, ..., D_n\}$, one for each X_i
  - a set of constraints $C \stackrel{\text{def}}{=} \{C_1, ..., C_m\}$
    - specify allowable combinations of values for the variables in X
- Each D_i is a set of allowable values $\{v_i, ..., v_k\}$ for variable X_i
- Each C_i is a pair $\langle scope, rel \rangle$
  - 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 $X \stackrel{\text{def}}{=} \{X_1, ..., X_n\}$
  - a set of (non-empty) domains $D \stackrel{\text{def}}{=} \{D_1, ..., D_n\}$, one for each X_i
  - a set of constraints $C \stackrel{\text{def}}{=} \{C_1, ..., C_m\}$
    - specify allowable combinations of values for the variables in X
- Each D_i is a set of allowable values $\{v_i, ..., v_k\}$ for variable X_i
- Each C_i is a pair $\langle scope, rel\rangle$
  - 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 $X \stackrel{\text{def}}{=} \{X_1, ..., X_n\}$
  - a set of (non-empty) domains $D \stackrel{\text{def}}{=} \{D_1, ..., D_n\}$, one for each X_i
  - a set of constraints $C \stackrel{\text{def}}{=} \{C_1, ..., C_m\}$
    - specify allowable combinations of values for the variables in X
- Each D_i is a set of allowable values $\{v_i, ..., v_k\}$ for variable X_i
- Each C_i is a pair $\langle scope, rel \rangle$
  - 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 $X \stackrel{\text{def}}{=} \{X_1, ..., X_n\}$
  - a set of (non-empty) domains $D \stackrel{\text{def}}{=} \{D_1, ..., D_n\}$, one for each X_i
  - a set of constraints $C \stackrel{\text{def}}{=} \{C_1, ..., C_m\}$
    - specify allowable combinations of values for the variables in X
- Each D_i is a set of allowable values $\{v_i, ..., v_k\}$ for variable X_i
- Each C_i is a pair $\langle scope, rel \rangle$
  - 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 $X \stackrel{\text{def}}{=} \{X_1, ..., X_n\}$
  - a set of (non-empty) domains $D \stackrel{\text{def}}{=} \{D_1, ..., D_n\}$, one for each X_i
  - a set of constraints $C \stackrel{\text{def}}{=} \{C_1, ..., C_m\}$
    - specify allowable combinations of values for the variables in X
- Each D_i is a set of allowable values $\{v_i, ..., v_k\}$ for variable X_i
- Each C_i is a pair $\langle scope, rel \rangle$
  - 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 or total, if every variable is assigned a value
  - incomplete or 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 or total, if every variable is assigned a value
  - incomplete or 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 or total, if every variable is assigned a value
  - incomplete or 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 or total, if every variable is assigned a value
  - incomplete or 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 or total, if every variable is assigned a value
  - incomplete or 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 or total, if every variable is assigned a value
  - incomplete or 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 or total, if every variable is assigned a value
  - incomplete or 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}, $i = A, ..., I;\ j = 1...9$
- Domain: $\{1, 2, ..., 8, 9\}$
- Constraints:
  - $AllDiff(X_{i1}, ..., X_{i9})$ for each row i
  - $AllDiff(X_{Aj}, ..., X_{Ij})$ for each column j
  - $AllDiff(X_{A1}, ..., X_{A3}, X_{B1} ..., X_{C3})$ for each 3×3 square region

  (alternatively, a long list of pairwise inequality constraints: $X_{A1} \neq X_{A2}, X_{A1} \neq X_{A3}, ...$)
- Solution: total value assignment satisfying all the constraints: $X_{A1} = 4, X_{A2} = 8, X_{A3} = 3, ...$

	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		

Example: Sudoku

- 81 Variables: (each square) X_{ij}, $i = A, ..., I$; $j = 1...9$
- Domain: $\{1, 2, ..., 8, 9\}$
- Constraints:
  - $AllDiff(X_{i1}, ..., X_{i9})$ for each row i
  - $AllDiff(X_{Aj}, ..., X_{Ij})$ for each column j
  - $AllDiff(X_{A1}, ..., X_{A3}, X_{B1} ..., X_{C3})$ for each 3×3 square region

  (alternatively, a long list of pairwise inequality constraints: $X_{A1} \neq X_{A2}, X_{A1} \neq X_{A3}, ...$)
- Solution: total value assignment satisfying all the constraints: $X_{A1} = 4, X_{A2} = 8, X_{A3} = 3, ...$

	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		

Example: Sudoku

- 81 Variables: (each square) X_{ij}, $i = A, ..., I;\ j = 1...9$
- Domain: $\{1, 2, ..., 8, 9\}$
- Constraints:
  - $AllDiff(X_{i1}, ..., X_{i9})$ for each row i
  - $AllDiff(X_{Aj}, ..., X_{Ij})$ for each column j
  - $AllDiff(X_{A1}, ..., X_{A3}, X_{B1} ..., X_{C3})$ for each 3×3 square region

  (alternatively, a long list of pairwise inequality constraints: $X_{A1} \neq X_{A2}, X_{A1} \neq X_{A3}, ...$)
- Solution: total value assignment satisfying all the constraints: $X_{A1} = 4, X_{A2} = 8, X_{A3} = 3, ...$

	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		

Example: Sudoku

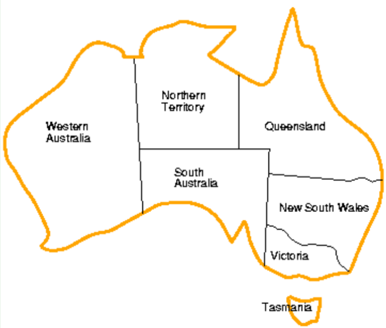
- 81 Variables: (each square) X_{ij}, $i = A, \ldots, I;\ j = 1 \ldots 9$
- Domain: $\{1, 2, \ldots, 8, 9\}$
- Constraints:
  - $AllDiff(X_{i1}, \ldots, X_{i9})$ for each row i
  - $AllDiff(X_{Aj}, \ldots, X_{Ij})$ for each column j
  - $AllDiff(X_{A1}, \ldots, X_{A3}, X_{B1} \ldots, X_{C3})$ for each 3×3 square region

  (alternatively, a long list of pairwise inequality constraints: $X_{A1} \neq X_{A2}, X_{A1} \neq X_{A3}, \ldots$)
- Solution: total value assignment satisfying all the constraints: $X_{A1} = 4, X_{A2} = 8, X_{A3} = 3, \ldots$

	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

Example: Map-Coloring

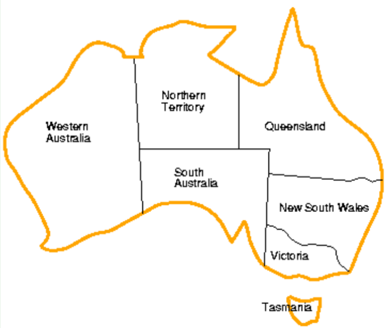
- Variables WA, NT, Q, NSW, V, SA, T
- Domain $D_i = \{red, green, blue\}, \forall i$
- Constraints: adjacent regions must have different colours
  - e.g. (explicit enumeration): $\langle WA, NT \rangle \in \{\langle red, green \rangle, \langle red, blue \rangle, \}$
    or (implicit, if language allows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



(© S. Russell & P. Norwig, AIMA)

Example: Map-Coloring

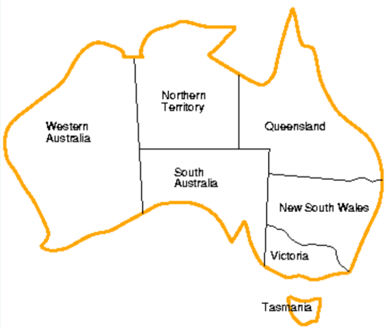
- Variables WA, NT, Q, NSW, V, SA, T
- Domain $D_i = \{red, green, blue\}, \forall i$
- Constraints: adjacent regions must have different colours
  - e.g. (explicit enumeration): $\langle WA, NT \rangle \in \{\langle red, green \rangle, \langle red, blue \rangle, \}$
    or (implicit, if language allows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



(© S. Russell & P. Norwig, AIMA)

Example: Map-Coloring

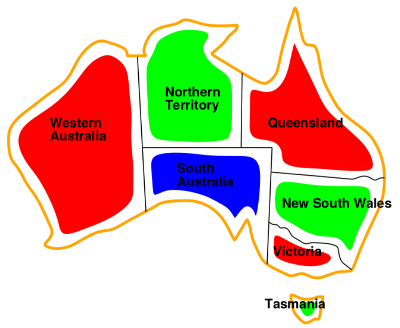
- Variables WA, NT, Q, NSW, V, SA, T
- Domain $D_i = \{red, green, blue\}, \forall i$
- Constraints: adjacent regions must have different colours
  - e.g. (explicit enumeration): $\langle WA, NT \rangle \in \{\langle red, green \rangle, \langle red, blue \rangle, \}$
    or (implicit, if language allows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



(© S. Russell & P. Norwig, AIMA)

Example: Map-Coloring

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



(© S. Russell & P. Norwig, AIMA)

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 constrain
- CSP algorithms use the graph structure to speed up search
  - Ex: Tasmania is an independent subproblem!

Constraint Graphs

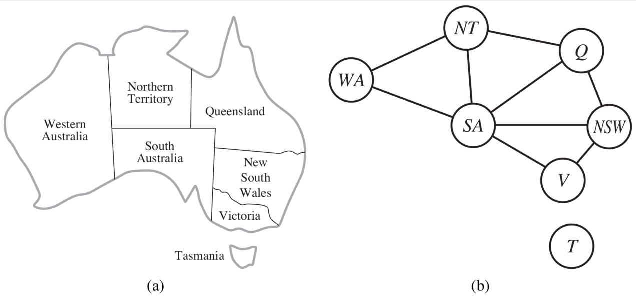
- Useful to visualize a CSP as a constraint graph (aka network)
  - the nodes of the graph correspond to variables of the problem
  - an edge connects any two variables that participate in a constrain
- CSP algorithms use the graph structure to speed up search
  - Ex: Tasmania is an independent subproblem!

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 constrain
- 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 $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ solvable (but NP-Hard)
    - non-linear constraints $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ solvable (but NP-Hard)
    - non-linear constraints $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ solvable (but NP-Hard)
    - non-linear constraints $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ solvable (but NP-Hard)
    - non-linear constraints $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ 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 $\Longrightarrow$ solvable (but NP-Hard)
    - non-linear constraints $\Longrightarrow$ 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

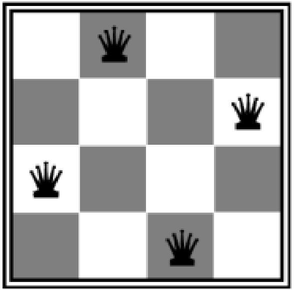
- Discrete variables
  - Finite domains (ex: Booleans, bounded integers, lists of values)
    - domain size d $\Longrightarrow$ d^n complete assignments (candidate solutions)
    - e.g. Boolean CSPs, incl. Boolean satisfiability (NP-complete)
    - possible to define constraints by enumerating all combinations (although unpractical)
  - Infinite domains (ex: unbounded integers)
    - infinite domain size $\Longrightarrow$ 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 $\Longrightarrow$ solvable (but NP-Hard)
    - non-linear constraints $\Longrightarrow$ 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

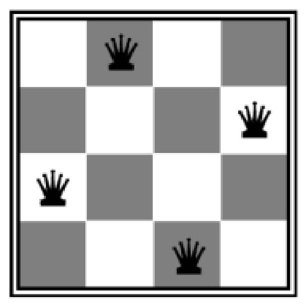
- variables X_{ij}, $i, j = 1..N$
- domains: $\{0, 1\}$
- constraints (explicit):
  - $\forall i, j, k\ \langle X_{ij}, X_{ik} \rangle \in \{\langle 0,0 \rangle, \langle 1,0 \rangle, \langle 0,1 \rangle\}$ (row)
  - $\forall i, j, k\ \langle X_{ij}, X_{kj} \rangle \in \{\langle 0,0 \rangle, \langle 1,0 \rangle, \langle 0,1 \rangle\}$ (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

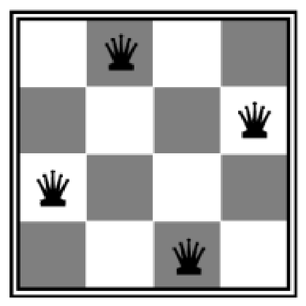
- variables X_{ij}, $i, j = 1..N$
- domains: $\{0, 1\}$
- constraints (explicit):
  - $\forall i, j, k \ \langle X_{ij}, X_{ik} \rangle \in \{\langle 0,0\rangle, \langle 1,0\rangle, \langle 0,1\rangle\}$ (row)
  - $\forall i, j, k \ \langle X_{ij}, X_{kj} \rangle \in \{\langle 0,0\rangle, \langle 1,0\rangle, \langle 0,1\rangle\}$ (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

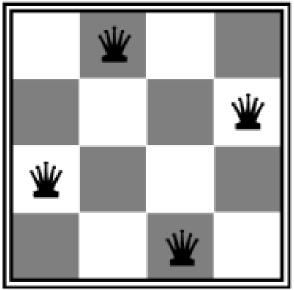
- variables X_{ij}, $i, j = 1..N$
- domains: $\{0, 1\}$
- constraints (explicit):
  - $\forall i, j, k \ \langle X_{ij}, X_{ik} \rangle \in \{\langle 0,0 \rangle, \langle 1,0 \rangle, \langle 0,1 \rangle\}$ (row)
  - $\forall i, j, k \ \langle X_{ij}, X_{kj} \rangle \in \{\langle 0,0 \rangle, \langle 1,0 \rangle, \langle 0,1 \rangle\}$ (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

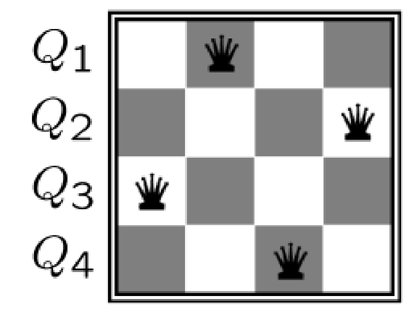
- variables X_{ij}, $i, j = 1..N$
- domains: $\{0, 1\}$
- constraints (explicit):
  - $\forall i, j, k \; \langle X_{ij}, X_{ik} \rangle \in \{\langle 0,0\rangle, \langle 1,0\rangle, \langle 0,1\rangle\}$ (row)
  - $\forall i, j, k \; \langle X_{ij}, X_{kj} \rangle \in \{\langle 0,0\rangle, \langle 1,0\rangle, \langle 0,1\rangle\}$ (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

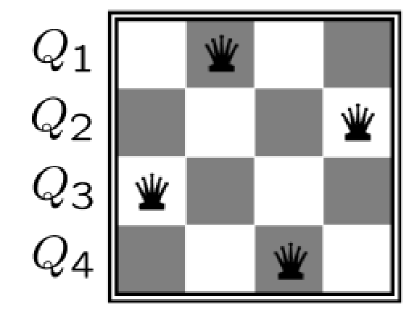


(© S. Russell & P. Norwig, AIMA)

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

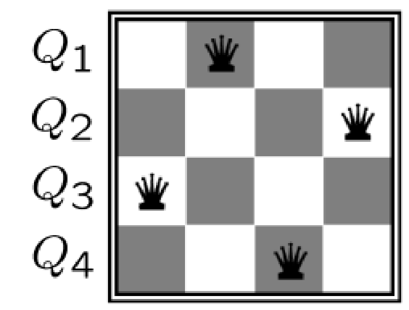


(© S. Russell & P. Norwig, AIMA)

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

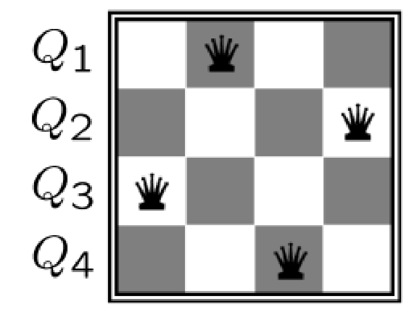


(© S. Russell & P. Norwig, AIMA)

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



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: cryptarithmetic column constraints
  - can be represented by constraint hypergraphs (hypernodes represent n-ary constraints, squares in cryptarithmetic example)
- Global constraints: involve an arbitrary number of variables
  - ex: $AllDiff(X_1, ..., X_k)$
  - 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:

$\Longrightarrow$ 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: cryptarithmetic column constraints
  - can be represented by constraint hypergraphs (hypernodes represent n-ary constraints, squares in cryptarithmetic example)
- Global constraints: involve an arbitrary number of variables
  - ex: $AllDiff(X_1, ..., X_k)$
  - 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:

$\Longrightarrow$ 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: cryptarithmetic column constraints
  - can be represented by constraint hypergraphs (hypernodes represent n-ary constraints, squares in cryptarithmetic example)
- Global constraints: involve an arbitrary number of variables
  - ex: $AllDiff(X_1, ..., X_k)$
  - 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:

$\Longrightarrow$ 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: cryptarithmetic column constraints
  - can be represented by constraint hypergraphs (hypernodes represent n-ary constraints, squares in cryptarithmetic example)
- Global constraints: involve an arbitrary number of variables
  - ex: $AllDiff(X_1, ..., X_k)$
  - 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:

  $\Longrightarrow$ 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: cryptarithmetic column constraints
  - can be represented by constraint hypergraphs (hypernodes represent n-ary constraints, squares in cryptarithmetic example)
- Global constraints: involve an arbitrary number of variables
  - ex: $AllDiff(X_1, ..., X_k)$
  - 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:

$\Longrightarrow$ solved by cost-optimization search techniques (Constraint Optimization Problems (COPs))

Varieties of Constraints

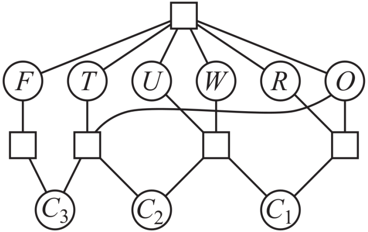
- Unary constraints: involve one single variable
  - ex: $(SA \neq green)$
- Binary constraints: involve pairs of variables
  - ex: $(SA \neq WA)$
- Higher-order constraints: involve ≥ 3 variables
  - ex: cryptarithmetic column constraints
  - can be represented by constraint hypergraphs (hypernodes represent n-ary constraints, squares in cryptarithmetic example)
- Global constraints: involve an arbitrary number of variables
  - ex: $AllDiff(X_1, ..., X_k)$
  - 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:

$\Longrightarrow$ solved by cost-optimization search techniques (Constraint Optimization Problems (COPs))

Example: Cryptarithmetic Puzzle

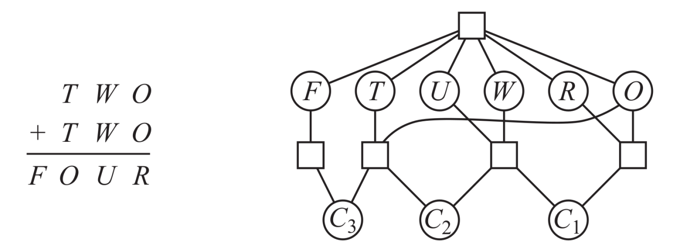
- Variables: F, T, U, W, R, O, plus C_1, C_2, C_3 (carry)
- Domains: $F, T, U, W, R, O \in \{0, 1, ..., 9\}$; $C_1, C_2, C_3 \in \{0, 1\}$
- Constraints: $\left\{ \begin{array}{l} AllDiff(F, T, U, W, R, O), \\ O + O = R + 10 \cdot C_1 \\ W + W + C_1 = U + 10 \cdot C_2 \\ T + T + C_2 = 10 \cdot C_3 + O \\ F = C_3, F \neq 0, T \neq 0 \end{array} \right\}$
- (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: Cryptarithmetic Puzzle

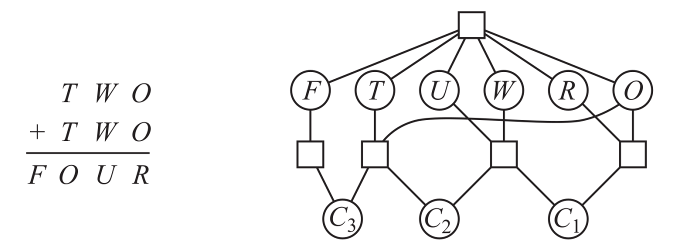
- Variables: F, T, U, W, R, O, plus C_1, C_2, C_3 (carry)
- Domains: $F, T, U, W, R, O \in \{0, 1, ..., 9\}$; $C_1, C_2, C_3 \in \{0, 1\}$
- Constraints: $\left\{ \begin{array}{l} AllDiff(F, T, U, W, R, O), \\ O + O = R + 10 \cdot C_1 \\ W + W + C_1 = U + 10 \cdot C_2 \\ T + T + C_2 = 10 \cdot C_3 +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



(© S. Russell & P. Norwig, AIMA)

Example: Cryptarithmetic Puzzle

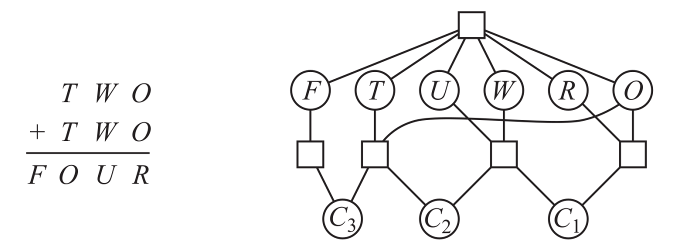
- Variables: F, T, U, W, R, O, plus C_1, C_2, C_3 (carry)
- Domains: $F, T, U, W, R, O \in \{0, 1, ..., 9\}$; $C_1, C_2, C_3 \in \{0, 1\}$
- Constraints: $\left\{ \begin{array}{l} AllDiff(F, T, U, W, R, O), \\ O + O = R + 10 \cdot C_1 \\ W + W + C_1 = U + 10 \cdot C_2 \\ T + T + C_2 = 10 \cdot C_3 +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



(© S. Russell & P. Norwig, AIMA)

Example: Cryptarithmetic Puzzle

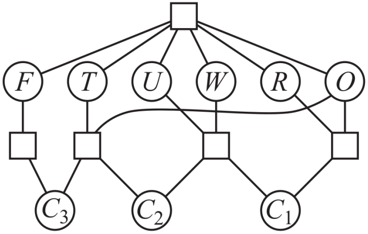
- Variables: F, T, U, W, R, O, plus C_1, C_2, C_3 (carry)
- Domains: $F, T, U, W, R, O \in \{0, 1, ..., 9\}$; $C_1, C_2, C_3 \in \{0, 1\}$
- Constraints: $\left\{ \begin{array}{l} AllDiff(F, T, U, W, R, O), \\ O + O = R + 10 \cdot C_1 \\ W + W + C_1 = U + 10 \cdot C_2 \\ T + T + C_2 = 10 \cdot C_3 +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



Example: Cryptarithmetic Puzzle

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

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



(© S. Russell & P. Norwig, AIMA)

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* $(X_{T'} + duration_{T'} \leq X_T)$

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* $(X_{T'} + duration_{T'} \leq X_T)$

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

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

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

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

Outline

1. Defining Constraint Satisfaction Problems (CSPs)
2. Inference in CSPs: Constraint Propagation
3. Backtracking Search with CSPs
4. Local Search with CSPs
5. Exploiting Structure of CSPs

Remark

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

$\Longrightarrow$ often CSP solvers work with binary constraints only

- In 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)

$\Longrightarrow$ often CSP solvers work with binary constraints only

- In 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)

$\Longrightarrow$ often CSP solvers work with binary constraints only

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

Constraint Propagation

- In state-space search, an algorithm can only search
- With CSPs, an algorithm can
  - search: pick a new variable assignment
  - infer (apply constraint propagation):
    use the constraints to reduce the set of legal candidate values for a variable
- Constraint propagation can either:
  - be interleaved with search
  - be performed as a preprocessing step
- Intuition: preserve and propagate local consistency
  - enforcing local consistency in each part of the constraint graph

  $\Longrightarrow$ 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 and strong k-consistency, $k \geq 1$

Constraint Propagation

- In state-space search, an algorithm can only search
- With CSPs, an algorithm can
  - search: pick a new variable assignment
  - infer (apply constraint propagation): use the constraints to reduce the set of legal candidate values for a variable
- Constraint propagation can either:
  - be interleaved with search
  - be performed as a preprocessing step
- Intuition: preserve and propagate local consistency
  - enforcing local consistency in each part of the constraint graph

  $\Longrightarrow$ 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 and strong k-consistency, $k \geq 1$

Constraint Propagation

- In state-space search, an algorithm can only search
- With CSPs, an algorithm can
  - search: pick a new variable assignment
  - infer (apply constraint propagation):
    use the constraints to reduce the set of legal candidate values for a variable
- Constraint propagation can either:
  - be interleaved with search
  - be performed as a preprocessing step
- Intuition: preserve and propagate local consistency
  - enforcing local consistency in each part of the constraint graph

  $\Longrightarrow$ 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 and strong k-consistency, $k \geq 1$

Constraint Propagation

- In state-space search, an algorithm can only search
- With CSPs, an algorithm can
  - search: pick a new variable assignment
  - infer (apply constraint propagation):
    use the constraints to reduce the set of legal candidate values for a variable
- Constraint propagation can either:
  - be interleaved with search
  - be performed as a preprocessing step
- Intuition: preserve and propagate local consistency
  - enforcing local consistency in each part of the constraint graph

  $\Longrightarrow$ 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 and strong k-consistency, $k \geq 1$

Constraint Propagation

- In state-space search, an algorithm can only search
- With CSPs, an algorithm can
  - search: pick a new variable assignment
  - infer (apply constraint propagation):
    use the constraints to reduce the set of legal candidate values for a variable
- Constraint propagation can either:
  - be interleaved with search
  - be performed as a preprocessing step
- Intuition: preserve and propagate local consistency
  - enforcing local consistency in each part of the constraint graph

  $\Longrightarrow$ 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 and strong k-consistency, $k \geq 1$

Constraint Propagation

- In state-space search, an algorithm can only search
- With CSPs, an algorithm can
  - search: pick a new variable assignment
  - infer (apply constraint propagation):
    use the constraints to reduce the set of legal candidate values for a variable
- Constraint propagation can either:
  - be interleaved with search
  - be performed as a preprocessing step
- Intuition: preserve and propagate local consistency
  - enforcing local consistency in each part of the constraint graph

  $\Longrightarrow$ 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 and strong 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 $WA \neq green$ 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

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
  - ensure arcs from assigned to unassigned variables are arc consistent
- Arc-consistency propagation: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
  - ensure all arcs are arc consistent!
- A well-known algorithm: AC-3

  $\Longrightarrow$ 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
  - ensure arcs from assigned to unassigned variables are arc consistent
- Arc-consistency propagation: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
  - ensure all arcs are arc consistent!
- A well-known algorithm: AC-3

  $\Longrightarrow$ 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
  - ensure arcs from assigned to unassigned variables are arc consistent
- Arc-consistency propagation: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
  - ensure all arcs are arc consistent!
- A well-known algorithm: AC-3

  $\Longrightarrow$ 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
  - ensure arcs from assigned to unassigned variables are arc consistent
- Arc-consistency propagation: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
  - ensure all arcs are arc consistent!
- A well-known algorithm: AC-3

  $\Longrightarrow$ 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
  - ensure arcs from assigned to unassigned variables are arc consistent
- Arc-consistency propagation: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
  - ensure all arcs are arc consistent!
- A well-known algorithm: AC-3

  $\Longrightarrow$ 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)

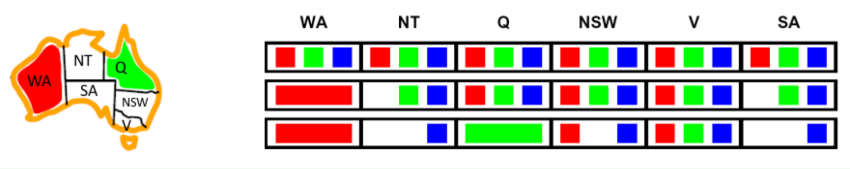
- X_i is arc-consistent wrt. X_j iff for every value d_i of X_i in D_i exists a value d_j for X_j in D_j which satisfy all binary constraints on $\langle X_i, X_j \rangle$
- A CSP is arc-consistent if every variable is arc consistent with every other variable
- Forward Checking: remove values from unassigned variables which are not arc consistent with assigned variables
  - ensure arcs from assigned to unassigned variables are arc consistent
- Arc-consistency propagation: remove all values from the domains of every variable which are not arc-consistent with these of some other variables
  - ensure all arcs are arc consistent!
- A well-known algorithm: AC-3

  $\Longrightarrow$ 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 variable assignment
  - update remaining legal values for unassigned variables
- Does not provide early detection for all failures
- If X loses a value, neighbors of X need to be 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?

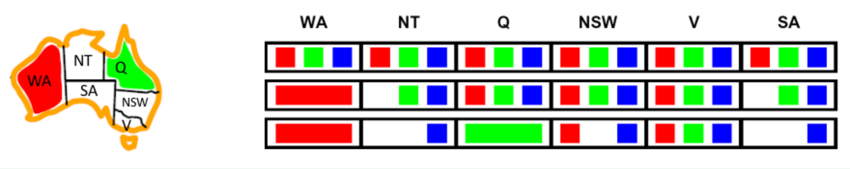


(© S. Russell & P. Norwig, AIMA)

Forward Checking

- Simplest form of propagation
- Idea: propagate information from assigned to unassigned variables
  - pick variable assignment
  - update remaining legal values for unassigned variables
- Does not provide early detection for all failures
- If X loses a value, neighbors of X need to be 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?

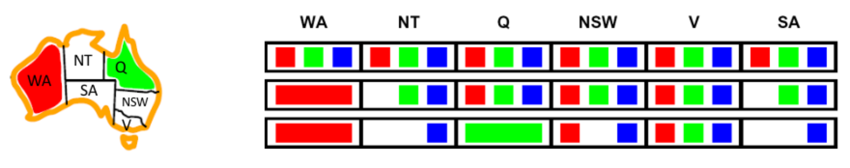


(© S. Russell & P. Norwig, AIMA)

Forward Checking

- Simplest form of propagation
- Idea: propagate information from assigned to unassigned variables
    - pick variable assignment
    - update remaining legal values for unassigned variables
- Does not provide early detection for all failures
- If X loses a value, neighbors of X need to be 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?

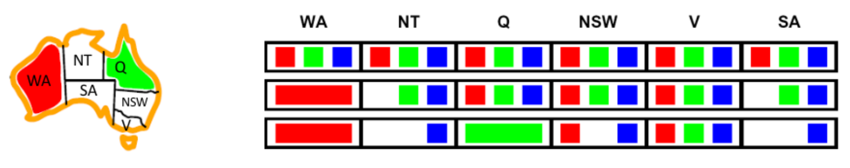


(© S. Russell & P. Norwig, AIMA)

Forward Checking

- Simplest form of propagation
- Idea: propagate information from assigned to unassigned variables
  - pick variable assignment
  - update remaining legal values for unassigned variables
- Does not provide early detection for all failures
- If X loses a value, neighbors of X need to be 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?

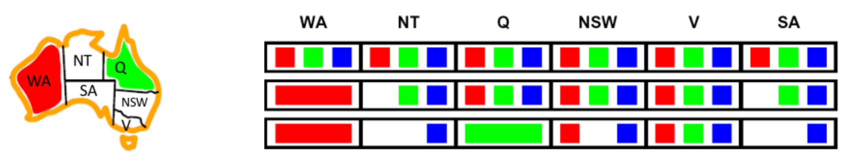


(© S. Russell & P. Norwig, AIMA)

Forward Checking

- Simplest form of propagation
- Idea: propagate information from assigned to unassigned variables
  - pick variable assignment
  - update remaining legal values for unassigned variables
- Does not provide early detection for all failures
- If X loses a value, neighbors of X need to be 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?

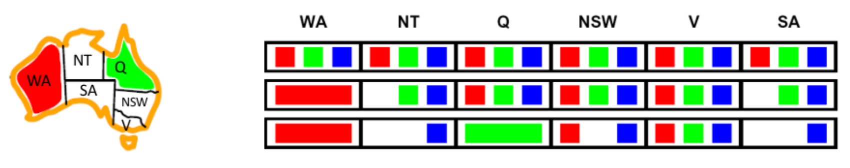


(© S. Russell & P. Norwig, AIMA)

Forward Checking

- Simplest form of propagation
- Idea: propagate information from assigned to unassigned variables
  - pick variable assignment
  - update remaining legal values for unassigned variables
- Does not provide early detection for all failures
- If X loses a value, neighbors of X need to be 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?

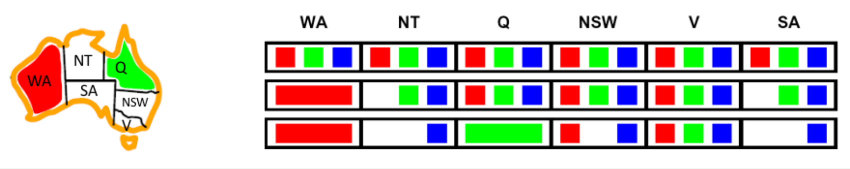


(© S. Russell & P. Norwig, AIMA)

Forward Checking

- Simplest form of propagation
- Idea: propagate information from assigned to unassigned variables
  - pick variable assignment
  - update remaining legal values for unassigned variables
- Does not provide early detection for all failures
- If X loses a value, neighbors of X need to be 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?



(© S. Russell & P. Norwig, AIMA)

The Arc-Consistency Propagation Algorithm AC-3

```
function AC-3( csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j) then     // makes Xi arc-consistent wrt. Xj
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true
```

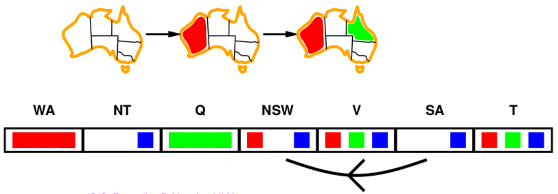
```
function REVISE( csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```


note: “queue” is LIFO $\Longrightarrow$ 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:
  - Revise(SA,NSW) $\Longrightarrow$ D_{SA} unchanged
  - ...
  - Revise(NSW,SA) $\Longrightarrow$ D_{NSW} revised
  - Revise(V,NSW) $\Longrightarrow$ D_V revised
  - ...
  - Revise(SA,NT) $\Longrightarrow$ D_{SA} revised
- Empty domain!

$\Longrightarrow$ Arc-consistency propagation detects failure earlier than forward checking

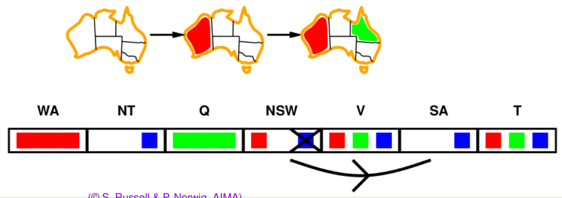


(© S. Russell & P. Norwig, AIMA)

Arc-Consistency Propagation AC-3: Example

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

$\Longrightarrow$ Arc-consistency propagation detects failure earlier than forward checking

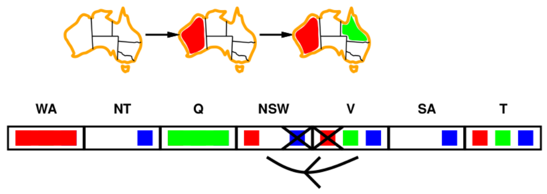


(© S. Russell & P. Norwig, AIMA)

Arc-Consistency Propagation AC-3: Example

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

$\Longrightarrow$ Arc-consistency propagation detects failure earlier than forward checking

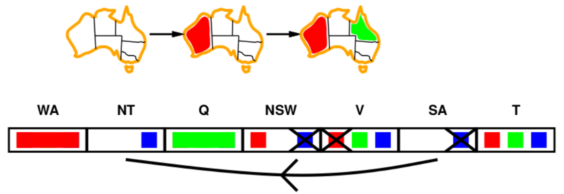


(© S. Russell & P. Norwig, AIMA)

Arc-Consistency Propagation AC-3: Example

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

$\Longrightarrow$ Arc-consistency propagation detects failure earlier than forward checking

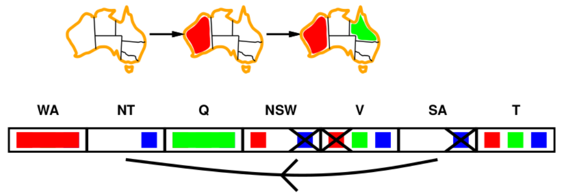


(© S. Russell & P. Norwig, AIMA)

Arc-Consistency Propagation AC-3: Example

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

$\Longrightarrow$ Arc-consistency propagation detects failure earlier than forward checking



Example: Sudoku

(consider *AllDiff*() as a set of binary constraints)
Apply arc-consistency propagation:

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 $\Longrightarrow$ E6=4
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 $\Longrightarrow$ I6=7
- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7,8,9 $\Longrightarrow$ A6=1
- ...

	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)

Exercise: Show that AC-3 solves the whole puzzle

Example: Sudoku

(consider *AllDiff*() as a set of binary constraints)
Apply arc-consistency propagation:

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 $\Longrightarrow$ E6=4
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 $\Longrightarrow$ I6=7
- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7,8,9 $\Longrightarrow$ A6=1
- ...

	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		

Exercise: Show that AC-3 solves the whole puzzle

Example: Sudoku

(consider *AllDiff*() as a set of binary constraints)
Apply arc-consistency propagation:

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 $\Longrightarrow$ E6=4
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 $\Longrightarrow$ I6=7
- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7,8,9 $\Longrightarrow$ A6=1
- ...

	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		

Exercise: Show that AC-3 solves the whole puzzle

Example: Sudoku

(consider *AllDiff*() as a set of binary constraints)
Apply arc-consistency propagation:

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 $\Longrightarrow$ E6=4
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 $\Longrightarrow$ I6=7
- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7,8,9 $\Longrightarrow$ A6=1
- ...

	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		

Exercise: Show that AC-3 solves the whole puzzle

Example: Sudoku

(consider *AllDiff*() as a set of binary constraints)
Apply arc-consistency propagation:

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 $\Longrightarrow$ E6=4
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 $\Longrightarrow$ I6=7
- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7,8,9 $\Longrightarrow$ A6=1
- ...

	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		

Exercise: Show that AC-3 solves the whole puzzle

Example: Sudoku

(consider *AllDiff*() as a set of binary constraints)
Apply arc-consistency propagation:

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 $\Longrightarrow$ E6=4
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 $\Longrightarrow$ I6=7
- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7,8,9 $\Longrightarrow$ A6=1
- ...

	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	**7**	3		

Exercise: Show that AC-3 solves the whole puzzle

Example: Sudoku

(consider *AllDiff*() as a set of binary constraints)
Apply arc-consistency propagation:

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 $\Longrightarrow$ E6=4
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 $\Longrightarrow$ I6=7
- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7.8,9 $\Longrightarrow$ A6=1
- ...

	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	**7**	3		

Exercise: Show that AC-3 solves the whole puzzle

Example: Sudoku

(consider *AllDiff*() as a set of binary constraints)
Apply arc-consistency propagation:

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 $\Longrightarrow$ E6=4
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 $\Longrightarrow$ I6=7
- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7.8,9 $\Longrightarrow$ A6=1
- ...

	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		

Exercise: Show that AC-3 solves the whole puzzle

Example: Sudoku

(consider *AllDiff*() as a set of binary constraints)
Apply arc-consistency propagation:

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 $\Longrightarrow$ E6=4
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 $\Longrightarrow$ I6=7
- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7.8,9 $\Longrightarrow$ A6=1
- ...

	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		

Exercise: Show that AC-3 solves the whole puzzle

Example: Sudoku

(consider *AllDiff*() as a set of binary constraints)
Apply arc-consistency propagation:

- What about E6?
  - arc-consistency propagation on column 6: drop 2,3,5,6,8,9
  - arc-consistency propagation on square: drop 1,7 $\Longrightarrow$ E6=4
- What about I6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,8,9
  - arc-consistency propagation on square: drop 1 $\Longrightarrow$ I6=7
- What about A6?
  - arc-consistency propagation on column 6: drop 2,3,4,5,6,7.8,9 $\Longrightarrow$ A6=1
- ...

	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

Exercise: Show that AC-3 solves the whole puzzle

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

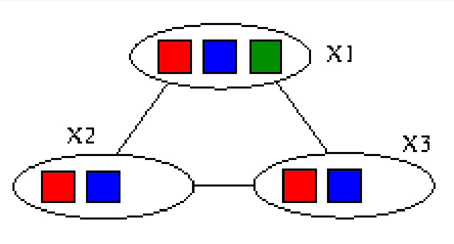
$\Longrightarrow$ Infers the assignment $C1 = green$

- 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

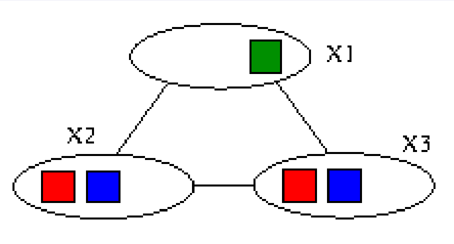
$\Longrightarrow$ Infers the assignment $C1 = green$

- 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

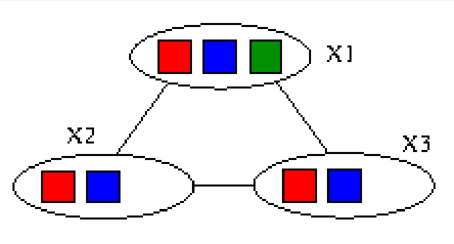
$\Longrightarrow$ Infers the assignment $C1 = \textit{green}$

- Can arc-consistency propagation reveal it?

  NO!

- Can path-consistency propagation reveal it?

  YES!



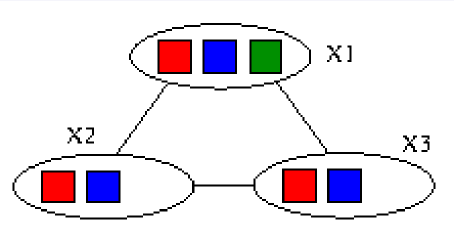
Arc vs. Path Consistency

- Can we say anything about X1?

  We can drop red & blue from D1

$\Longrightarrow$ Infers the assignment $C1 = green$

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



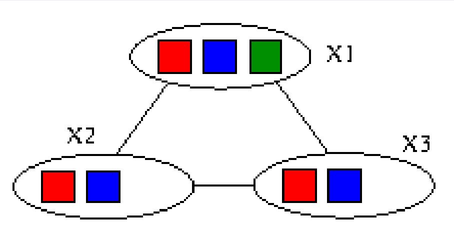
Arc vs. Path Consistency

- Can we say anything about X1?

  We can drop red & blue from D1

$\Longrightarrow$ Infers the assignment $C1 = green$

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



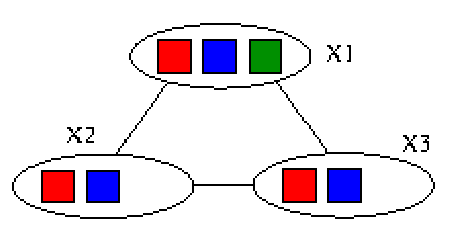
Arc vs. Path Consistency

- Can we say anything about X1?

  We can drop red & blue from D1

$\Longrightarrow$ Infers the assignment $C1 = green$

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



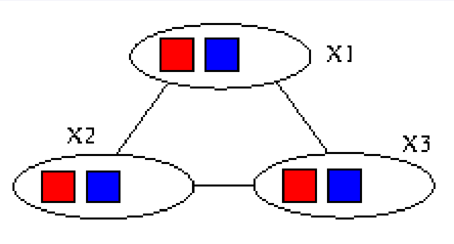
Arc vs. Path Consistency [cont.]

- Can we say anything?

  The triplet is inconsistent
- Can arc-consistency propagation reveal it?

  NO!
- Can path-consistency propagation reveal it?

  YES!



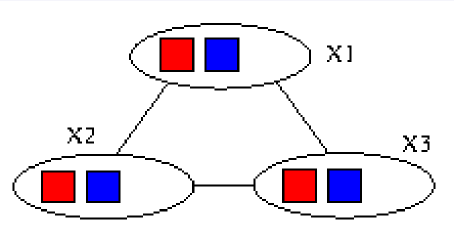
Arc vs. Path Consistency [cont.]

- Can we say anything?

  The triplet is inconsistent
- Can arc-consistency propagation reveal it?

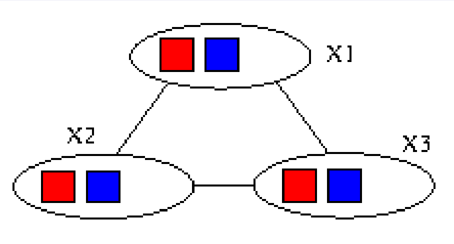
  NO!
- Can path-consistency propagation reveal it?

  YES!



Arc vs. Path Consistency [cont.]

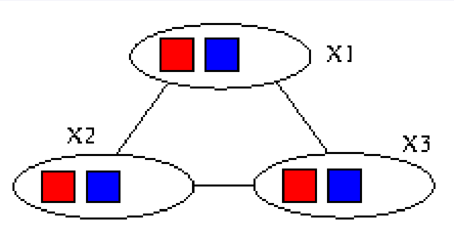
- Can we say anything?
  The triplet is inconsistent
- Can arc-consistency propagation reveal it?
  NO!
- Can path-consistency propagation reveal it?
  YES!



Arc vs. Path Consistency [cont.]

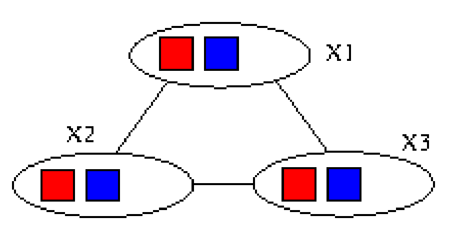
- Can we say anything?

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



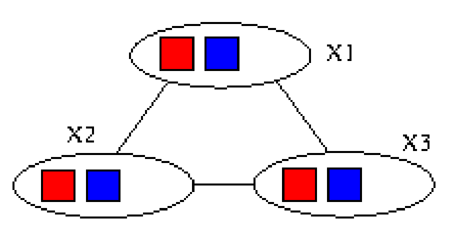
Arc vs. Path Consistency [cont.]

- 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!



Outline

Backtracking Search: Generalities

Backtracking Search

- Basic uninformed algorithm for solving CSPs
- Idea 1: Pick one variable at a time
  - variable assignments are commutative $\Longrightarrow$ fix an ordering
  - ex: $\{WA = red, NT = green\}$ same as $\{NT = green, WA = red\}$

  $\Longrightarrow$ 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

  $\Longrightarrow$ "incremental goal test"
  - can detect if a partial assignments violate a goal

    $\Longrightarrow$ early detection of inconsistencies
- 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 $\Longrightarrow$ fix an ordering
  - ex: $\{WA = red, NT = green\}$ same as $\{NT = green, WA = red\}$

  $\Longrightarrow$ 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

  $\Longrightarrow$ "incremental goal test"
  - can detect if a partial assignments violate a goal
    $\Longrightarrow$ early detection of inconsistencies
- 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 $\Longrightarrow$ fix an ordering
  - ex: $\{WA = red, NT = green\}$ same as $\{NT = green, WA = red\}$

  $\Longrightarrow$ 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

  $\Longrightarrow$ "incremental goal test"
  - can detect if a partial assignments violate a goal

    $\Longrightarrow$ early detection of inconsistencies
- 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 $\Longrightarrow$ fix an ordering
  - ex: $\{WA = red, NT = green\}$ same as $\{NT = green, WA = red\}$

  $\Longrightarrow$ 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

  $\Longrightarrow$ "incremental goal test"
  - can detect if a partial assignments violate a goal

    $\Longrightarrow$ early detection of inconsistencies
- Backtracking search: DFS with the two above improvements

Backtracking Search: Generalities

Backtracking Search

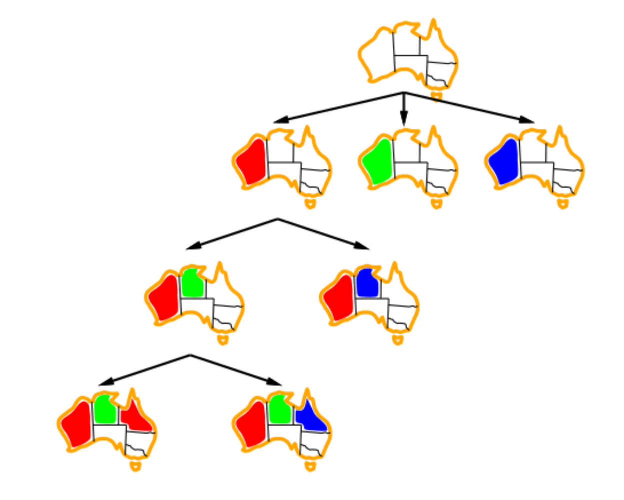
- Basic uninformed algorithm for solving CSPs
- Idea 1: Pick one variable at a time
  - variable assignments are commutative $\Longrightarrow$ fix an ordering
  - ex: $\{WA = red, NT = green\}$ same as $\{NT = green, WA = red\}$

  $\Longrightarrow$ 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

  $\Longrightarrow$ "incremental goal test"
  - can detect if a partial assignments violate a goal
    $\Longrightarrow$ early detection of inconsistencies
- Backtracking search: DFS with the two above improvements

Backtracking Search: Example

(Part of) Search Tree for Map-Coloring



(© S. Russell & P. Norwig, AIMA)

Backtracking Search Algorithm

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

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

inside first "if"

(© S. Russell & P. Norwig, AIMA)

Backtracking Search Algorithm [cont.]

- General-purpose algorithm for generic CSPs
- The representation of CSPs is standardized

  $\Longrightarrow$ no need to provide a domain-specific initial state, action function, transition model, or goal test
- *BacktrackingSearch*() keeps a single representation of a state
  - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
  - *SelectUnassignedVariable*(): which variable should be assigned next?
  - *OrderDomainValues*(): in what 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 usuless 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
  $\Longrightarrow$ no need to provide a domain-specific initial state, action function, transition model, or goal test
- *BacktrackingSearch*() keeps a single representation of a state
  - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
  - *SelectUnassignedVariable*(): which variable should be assigned next?
  - *OrderDomainValues*(): in what 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 usuless 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

  $\Longrightarrow$ no need to provide a domain-specific initial state, action function, transition model, or goal test
- *BacktrackingSearch*() keeps a single representation of a state
  - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
  - *SelectUnassignedVariable*(): which variable should be assigned next?
  - *OrderDomainValues*(): in what 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 usuless 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

  $\Longrightarrow$ no need to provide a domain-specific initial state, action function, transition model, or goal test
- *BacktrackingSearch*() keeps a single representation of a state
  - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
  - *SelectUnassignedVariable*(): which variable should be assigned next?
  - *OrderDomainValues*(): in what 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 usuless 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

  $\Longrightarrow$ no need to provide a domain-specific initial state, action function, transition model, or goal test
- *BacktrackingSearch*() keeps a single representation of a state
  - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
  - *SelectUnassignedVariable*(): which variable should be assigned next?
  - *OrderDomainValues*(): in what 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 usuless 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

  $\Longrightarrow$ no need to provide a domain-specific initial state, action function, transition model, or goal test
- *BacktrackingSearch*() keeps a single representation of a state
  - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
  - *SelectUnassignedVariable*(): which variable should be assigned next?
  - *OrderDomainValues*(): in what 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 usuless 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

  $\Longrightarrow$ no need to provide a domain-specific initial state, action function, transition model, or goal test
- *BacktrackingSearch*() keeps a single representation of a state
  - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
  - *SelectUnassignedVariable*(): which variable should be assigned next?
  - *OrderDomainValues*(): in what 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 usuless 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

  $\Longrightarrow$ no need to provide a domain-specific initial state, action function, transition model, or goal test
- *BacktrackingSearch*() keeps a single representation of a state
  - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
  - *SelectUnassignedVariable*(): which variable should be assigned next?
  - *OrderDomainValues*(): in what 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 usuless 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

  $\Longrightarrow$ no need to provide a domain-specific initial state, action function, transition model, or goal test
- *BacktrackingSearch*() keeps a single representation of a state
  - alters such representation rather than creating new ones
- We can add some sophistication to the unspecified functions:
  - *SelectUnassignedVariable*(): which variable should be assigned next?
  - *OrderDomainValues*(): in what 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 usuless 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

  $\Longrightarrow$ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X

  $\Longrightarrow$ failure detected immediately
  - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X

  $\Longrightarrow$ 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

  $\Longrightarrow$ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X

  $\Longrightarrow$ failure detected immediately
  - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X

  $\Longrightarrow$ 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

  $\Longrightarrow$ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X

  $\Longrightarrow$ failure detected immediately
  - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X

  $\Longrightarrow$ 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
  $\Longrightarrow$ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X
  $\Longrightarrow$ failure detected immediately
  - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X
  $\Longrightarrow$ performs deterministic choices first!
  - postpones nondeterministic steps as much as possible

Variable Selection Heuristics

Minimum Remaining Values (MRV) heuristic

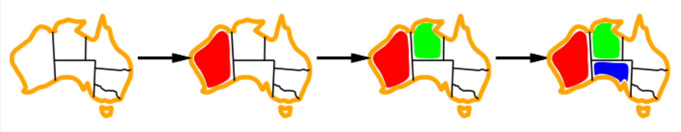
- Aka most constrained variable or fail-first heuristic
- MRV: Choose the variable with the fewest legal values

  $\Longrightarrow$ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X

  $\Longrightarrow$ failure detected immediately
  - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X

  $\Longrightarrow$ performs deterministic choices first!
  - postpones nondeterministic steps as much as possible

- Pick $(WA = red), (NT = green) \Longrightarrow (SA = blue)$ (deterministic)
- Next? $(Q = red)$



(© S. Russell & P. Norwig, AIMA)

Variable Selection Heuristics

Minimum Remaining Values (MRV) heuristic

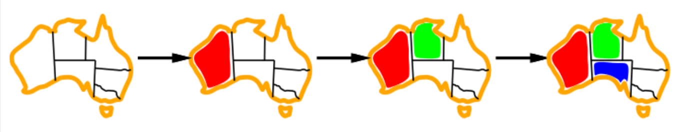
- Aka most constrained variable or fail-first heuristic
- MRV: Choose the variable with the fewest legal values

  $\Longrightarrow$ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X

  $\Longrightarrow$ failure detected immediately
  - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X

  $\Longrightarrow$ performs deterministic choices first!
  - postpones nondeterministic steps as much as possible

- Pick $(WA = red), (NT = green) \Longrightarrow (SA = blue)$ (deterministic)
- Next? $(Q = red)$



(© S. Russell & P. Norwig, AIMA)

Variable Selection Heuristics

Minimum Remaining Values (MRV) heuristic

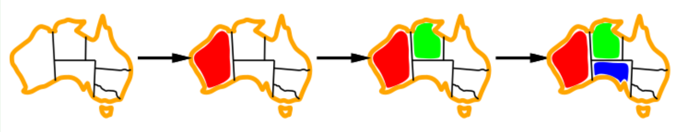
- Aka most constrained variable or fail-first heuristic
- MRV: Choose the variable with the fewest legal values

  $\Longrightarrow$ pick a variable that is most likely to cause a failure soon
- If X has no legal values left, MRV heuristic selects X

  $\Longrightarrow$ failure detected immediately
  - avoid pointless search through other variables
- (Otherwise) If X has one legal value left, MRV selects X

  $\Longrightarrow$ performs deterministic choices first!
  - postpones nondeterministic steps as much as possible

- Pick $(WA = red), (NT = green) \Longrightarrow (SA = blue)$ (deterministic)
- Next? $(Q = red)$



(© S. Russell & P. Norwig, AIMA)

Variable Selection Heuristics [cont.]

Degree heuristic

- Used as tie-breaker in combination with MRV
  - apply MRV; if ties, apply DH to these variables
- Pick the variable with most constraints on remaining variables

$\Longrightarrow$ attempts to reduce the branching factor on future choices

Variable Selection Heuristics [cont.]

Degree heuristic

- Used as tie-breaker in combination with MRV
  - apply MRV; if ties, apply DH to these variables
- Pick the variable with most constraints on remaining variables

  $\Longrightarrow$ attempts to reduce the branching factor on future choices

Variable Selection Heuristics [cont.]

Degree heuristic

- Used as tie-breaker in combination with MRV
  - apply MRV; if ties, apply DH to these variables
- Pick the variable with most constraints on remaining variables

$\Longrightarrow$ attempts to reduce the branching factor on future choices

Variable Selection Heuristics [cont.]

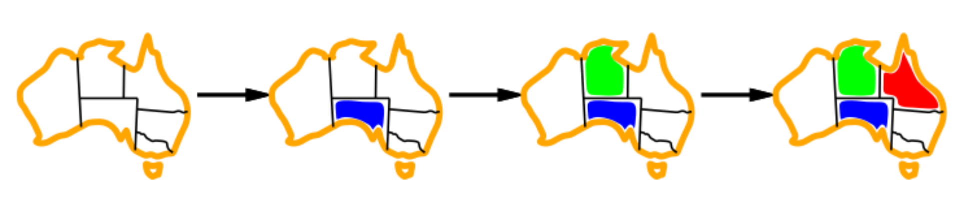
Degree heuristic

- Used as tie-breaker in combination with MRV
  - apply MRV; if ties, apply DH to these variables
- Pick the variable with most constraints on remaining variables

$\Longrightarrow$ attempts to reduce the branching factor on future choices

Example: MRV+DH

- Pick $(SA = blue), (NT = green) \Longrightarrow (Q = red)$ (deterministic)
- Next? (NSW=green)



(© S. Russell & P. Norwig, AIMA)

Variable Selection Heuristics [cont.]

Degree heuristic

- Used as tie-breaker in combination with MRV
  - apply MRV; if ties, apply DH to these variables
- Pick the variable with most constraints on remaining variables

$\Longrightarrow$ attempts to reduce the branching factor on future choices

Example: MRV+DH

- Pick $(SA = blue), (NT = green) \Longrightarrow (Q = red)$ (deterministic)
- Next? (NSW=green)

(© S. Russell & P. Norwig, AIMA)

Variable Selection Heuristics [cont.]

Degree heuristic

- Used as tie-breaker in combination with MRV
  - apply MRV; if ties, apply DH to these variables
- Pick the variable with most constraints on remaining variables

$\Longrightarrow$ attempts to reduce the branching factor on future choices

Example: MRV+DH

- Pick $(SA = blue), (NT = green) \Longrightarrow (Q = red)$ (deterministic)
- Next? (NSW=green)

(© S. Russell & P. Norwig, AIMA)

Value Selection Heuristics

Least Constraining Value (LCS) heuristic

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

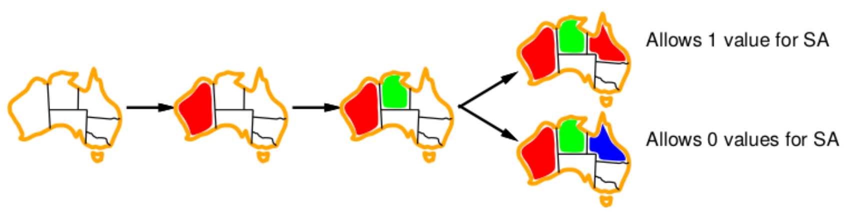
- Pick $(SA = red), (NT = green) \Longrightarrow (Q = red)$ (preferred)
- Next? (SA=blue)

Value Selection Heuristics

Least Constraining Value (LCS) heuristic

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

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

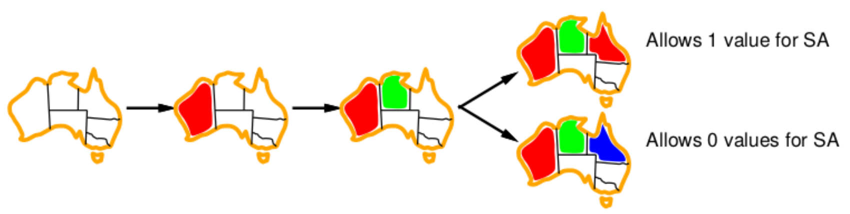


Value Selection Heuristics

Least Constraining Value (LCS) heuristic

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

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



(© S. Russell & P. Norwig, AIMA)

Inference

Interleaving search and inference

- After a choice, infer new domain reductions on other variables
  - detect inconsistencies earlier
  - reduce search spaces
  - may produce unary domains (deterministic steps)
    $\Longrightarrow$ returned as assignments ("inferences")
- Tradeoff between effectiveness and efficiency
- Forward checking
  - cheap
  - ensures arc consistency of $\langle$*assigned*, *unassigned*$\rangle$ variable pairs
- 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

    $\Longrightarrow$ much more effective than forward checking, more expensive

Inference

Interleaving search and inference

- After a choice, infer new domain reductions on other variables
  - detect inconsistencies earlier
  - reduce search spaces
  - may produce unary domains (deterministic steps)
    $\Longrightarrow$ returned as assignments ("inferences")
- Tradeoff between effectiveness and efficiency
- Forward checking
  - cheap
  - ensures arc consistency of $\langle assigned, unassigned \rangle$ variable pairs
- 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

    $\Longrightarrow$ much more effective than forward checking, more expensive

Inference

Interleaving search and inference

- After a choice, infer new domain reductions on other variables
  - detect inconsistencies earlier
  - reduce search spaces
  - may produce unary domains (deterministic steps)
    $\Longrightarrow$ returned as assignments ("inferences")
- Tradeoff between effectiveness and efficiency
- Forward checking
  - cheap
  - ensures arc consistency of $\langle assigned, unassigned \rangle$ variable pairs
- 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

    $\Longrightarrow$ much more effective than forward checking, more expensive

Inference

Interleaving search and inference

- After a choice, infer new domain reductions on other variables
  - detect inconsistencies earlier
  - reduce search spaces
  - may produce unary domains (deterministic steps)
    $\Longrightarrow$ returned as assignments ("inferences")
- Tradeoff between effectiveness and efficiency
- Forward checking
  - cheap
  - ensures arc consistency of $\langle$*assigned*, *unassigned*$\rangle$ variable pairs
- 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

    $\Longrightarrow$ much more effective than forward checking, more expensive

Inference

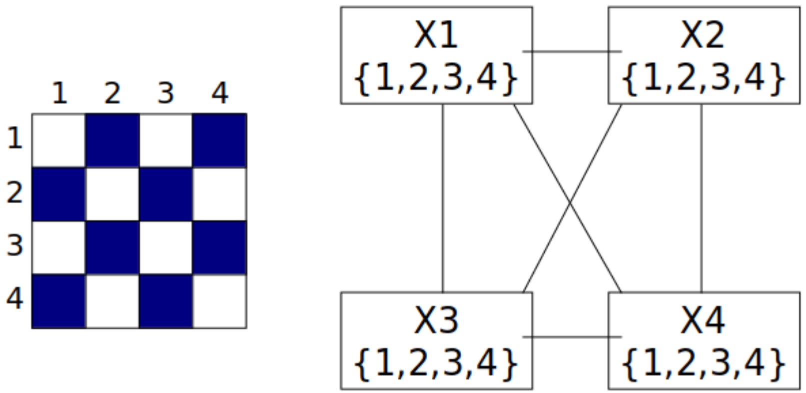
Interleaving search and inference

- After a choice, infer new domain reductions on other variables
  - detect inconsistencies earlier
  - reduce search spaces
  - may produce unary domains (deterministic steps)
    $\Longrightarrow$ returned as assignments ("inferences")
- Tradeoff between effectiveness and efficiency
- Forward checking
  - cheap
  - ensures arc consistency of $\langle \textit{assigned}, \textit{unassigned} \rangle$ variable pairs
- 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

    $\Longrightarrow$ much more effective than forward checking, more expensive

Backtracking with Forward Checking: Example

4-Queens

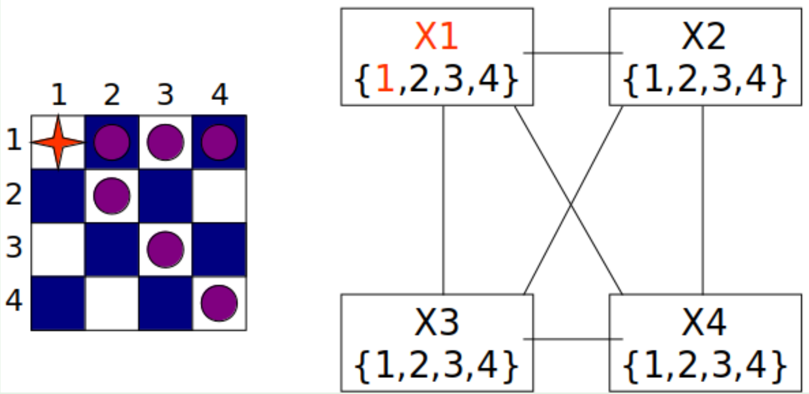


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

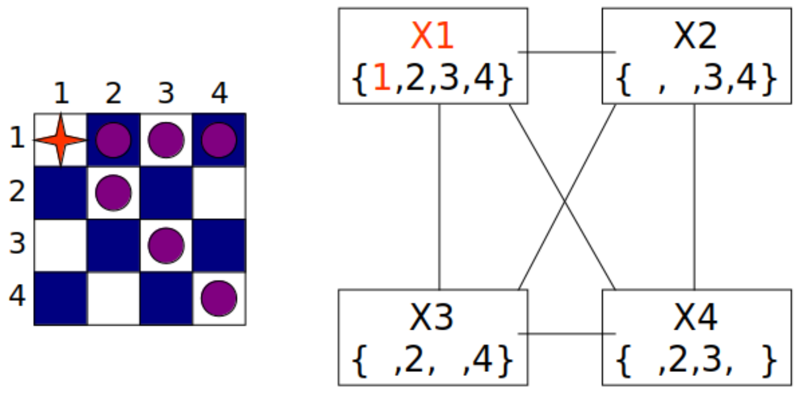


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

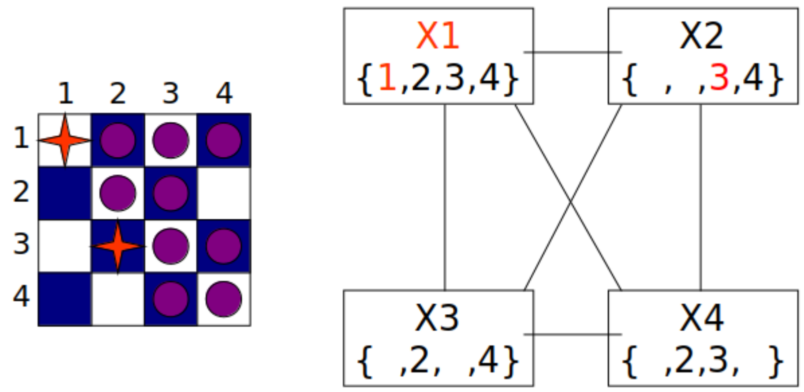


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

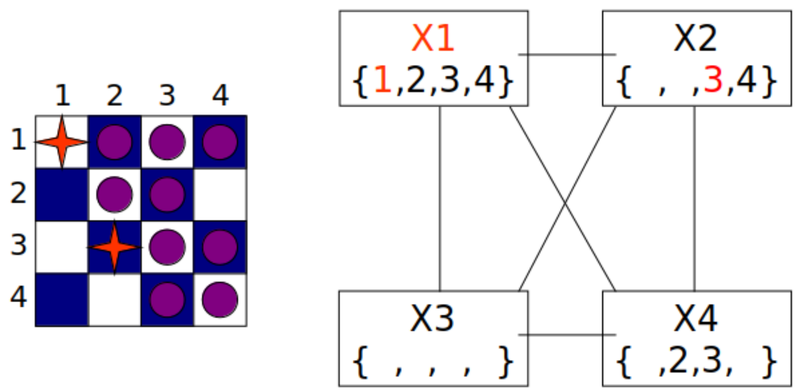


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

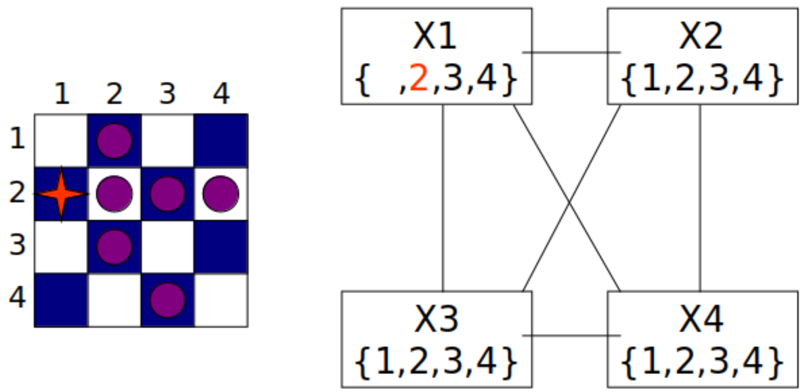


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

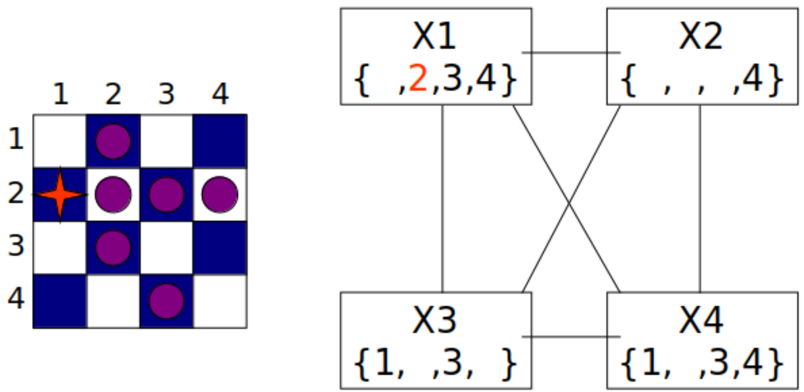


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

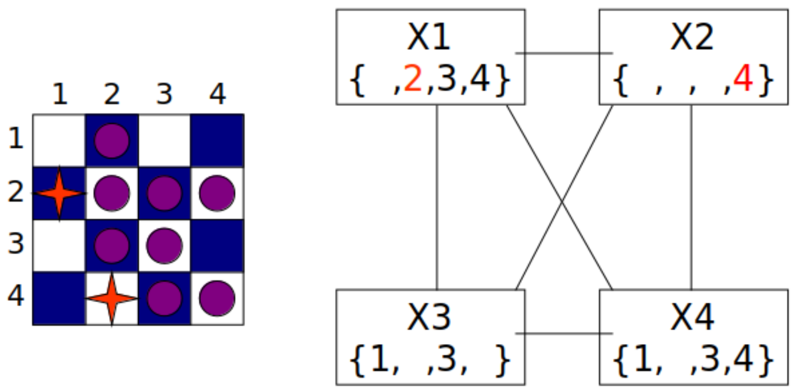


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

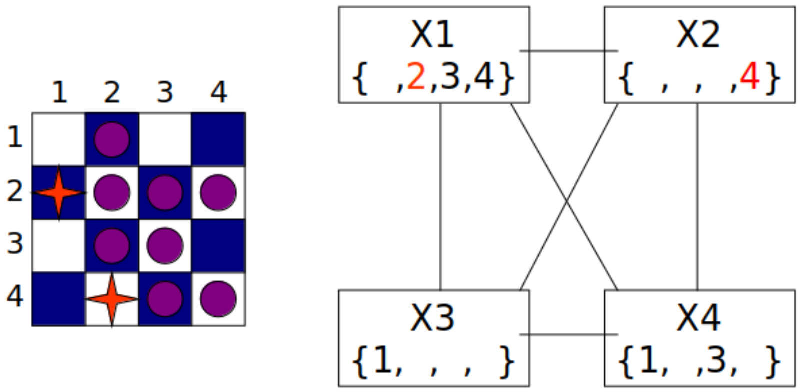


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

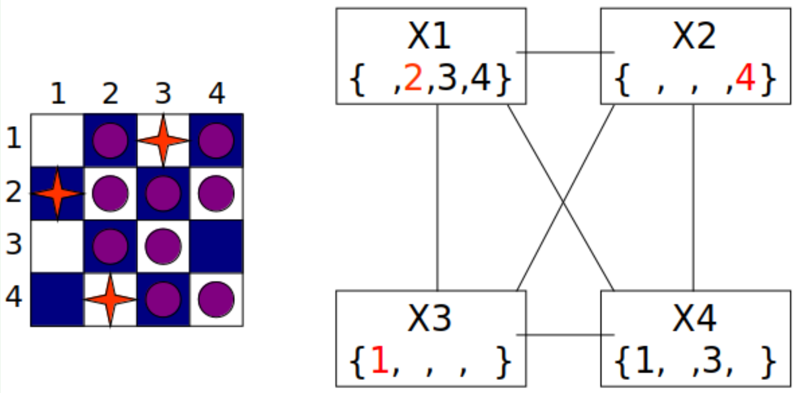


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

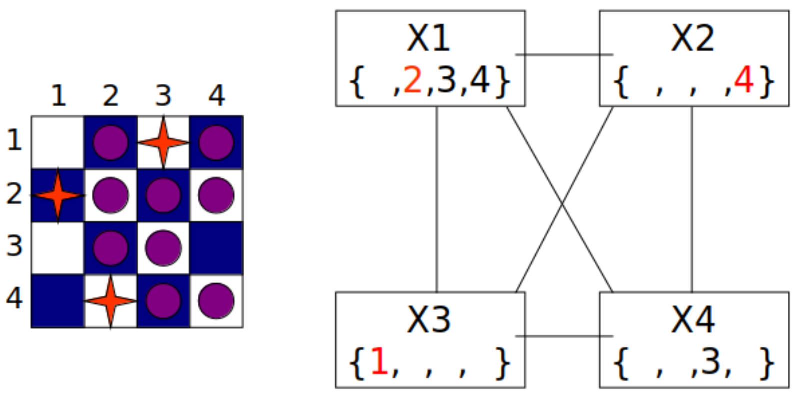


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

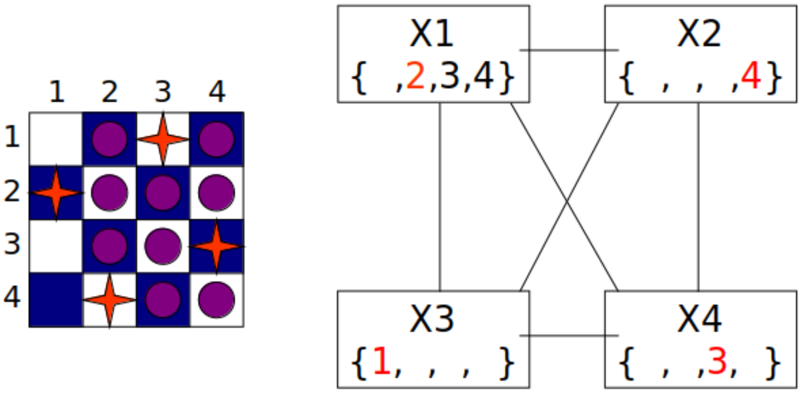


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Backtracking with Forward Checking: Example

4-Queens

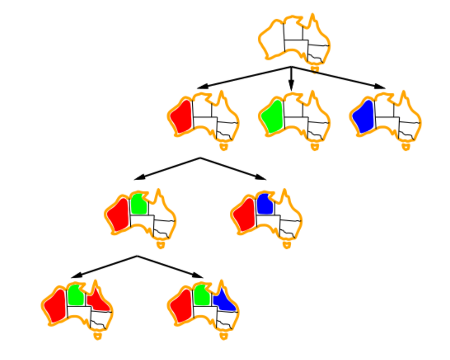


(© B.J.Dorr U.Md & Tom Lenaerts, IRIDIA)

...(after trying $X2 = 4$, failing and backtracking)...

Standard Chronological Backtracking

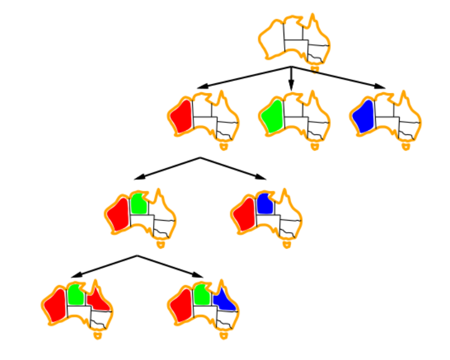
- When a branch fails (empty domain for variable X_i):
  1. back up to the preceding variable (who still has an untried value)
     - forward-propagated assignments and rightmost choices are skipped
  2. try a different value for it
- Problem: lots of search wasted!



(© S. Russell & P. Norwig, AIMA)

Standard Chronological Backtracking

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



(© S. Russell & P. Norwig, AIMA)

Standard Chronological Backtracking: Example

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

- failed branch:

	step	assignment [domain]
(1)	*pick*	$WA = r$ [rbg]
(2)	*pick*	$NSW = r$ [rbg]
(3)	*pick*	$T = r$ [rbg]
(4)	*pick*	$NT = g$ [bg]
(5)	$\stackrel{fc}{\Longrightarrow}$	$Q = b$ [b]
(6)	*pick*	$V = b$ [b, g]
(7)	$\stackrel{fc}{\Longrightarrow}$	$SA = \{\}$ []

- backtrack to (5), pick $V = g \Longrightarrow$ (7) again
- backtrack to (3), pick $NT = b \stackrel{fc}{\Longrightarrow} Q = g \Longrightarrow$ same subtree (6)...
- backtrack to (2), pick $T = b \Longrightarrow$ same subtree (4)...
- backtrack to (2), pick $T = g \Longrightarrow$ same subtree (4)...

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

$\Longrightarrow$ 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

- failed branch:

	step	assignment [domain]
(1)	*pick*	$WA = r$ $[rbg]$
(2)	*pick*	$NSW = r$ $[rbg]$
(3)	*pick*	$T = r$ $[rbg]$
(4)	*pick*	$NT = g$ $[bg]$
(5)	$\stackrel{fc}{\Longrightarrow}$	$Q = b$ $[b]$
(6)	*pick*	$V = b$ $[b, g]$
(7)	$\stackrel{fc}{\Longrightarrow}$	$SA = \{\}$ $[]$

- backtrack to (5), pick $V = g \Longrightarrow$ (7) again
- backtrack to (3), pick $NT = b \stackrel{fc}{\Longrightarrow} Q = g \Longrightarrow$ same subtree (6)...
- backtrack to (2), pick $T = b \Longrightarrow$ same subtree (4)...
- backtrack to (2), pick $T = g \Longrightarrow$ same subtree (4)...

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

$\Longrightarrow$ 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

- failed branch:

	step	*assignment* [*domain*]
(1)	*pick*	$WA = r$ [rbg]
(2)	*pick*	$NSW = r$ [rbg]
(3)	*pick*	$T = r$ [rbg]
(4)	*pick*	$NT = g$ [bg]
(5)	$\stackrel{fc}{\Longrightarrow}$	$Q = b$ [b]
(6)	*pick*	$V = b$ [b, g]
(7)	$\stackrel{fc}{\Longrightarrow}$	$SA = \{\}$ []

- backtrack to (5), pick $V = g \Longrightarrow$ (7) again
- backtrack to (3), pick $NT = b \stackrel{fc}{\Longrightarrow} Q = g \Longrightarrow$ same subtree (6)...
- backtrack to (2), pick $T = b \Longrightarrow$ same subtree (4)...
- backtrack to (2), pick $T = g \Longrightarrow$ same subtree (4)...

$\Longrightarrow$ backtrack to (1), then assign *NSW* another value

$\Longrightarrow$ 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

- failed branch:

	step	*assignment* [*domain*]
(1)	*pick*	$WA = r$ $[rbg]$
(2)	*pick*	$NSW = r$ $[rbg]$
(3)	*pick*	$T = r$ $[rbg]$
(4)	*pick*	$NT = g$ $[bg]$
(5)	$\stackrel{fc}{\Longrightarrow}$	$Q = b$ $[b]$
(6)	*pick*	$V = b$ $[b, g]$
(7)	$\stackrel{fc}{\Longrightarrow}$	$SA = \{\}$ $[]$

- backtrack to (5), pick $V = g \Longrightarrow$ (7) again
- backtrack to (3), pick $NT = b \stackrel{fc}{\Longrightarrow} Q = g \Longrightarrow$ same subtree (6)...
- backtrack to (2), pick $T = b \Longrightarrow$ same subtree (4)...
- backtrack to (2), pick $T = g \Longrightarrow$ same subtree (4)...

$\Longrightarrow$ backtrack to (1), then assign *NSW* another value

$\Longrightarrow$ 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

- failed branch:

	step	assignment [domain]
(1)	*pick*	$WA = r\ [rbg]$
(2)	*pick*	$NSW = r\ [rbg]$
(3)	*pick*	$T = r\ [rbg]$
(4)	*pick*	$NT = g\ [bg]$
(5)	$\stackrel{fc}{\Longrightarrow}$	$Q = b\ [b]$
(6)	*pick*	$V = b\ [b, g]$
(7)	$\stackrel{fc}{\Longrightarrow}$	$SA = \{\}\ []$

- backtrack to (5), pick $V = g \Longrightarrow$ (7) again
- backtrack to (3), pick $NT = b \stackrel{fc}{\Longrightarrow} Q = g \Longrightarrow$ same subtree (6)...
- backtrack to (2), pick $T = b \Longrightarrow$ same subtree (4)...
- backtrack to (2), pick $T = g \Longrightarrow$ same subtree (4)...

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

$\Longrightarrow$ 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

- failed branch:

	step	assignment [domain]
(1)	*pick*	$WA = r\ [rbg]$
(2)	*pick*	$NSW = r\ [rbg]$
(3)	*pick*	$T = r\ [rbg]$
(4)	*pick*	$NT = g\ [bg]$
(5)	$\stackrel{fc}{\Longrightarrow}$	$Q = b\ [b]$
(6)	*pick*	$V = b\ [b, g]$
(7)	$\stackrel{fc}{\Longrightarrow}$	$SA = \{\}\ []$

- backtrack to (5), pick $V = g \Longrightarrow$ (7) again
- backtrack to (3), pick $NT = b \stackrel{fc}{\Longrightarrow} Q = g \Longrightarrow$ same subtree (6)...
- backtrack to (2), pick $T = b \Longrightarrow$ same subtree (4)...
- backtrack to (2), pick $T = g \Longrightarrow$ same subtree (4)...

$\Longrightarrow$ backtrack to (1), then assign *NSW* another value

$\Longrightarrow$ 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

- failed branch:

	step	*assignment* [*domain*]
(1)	*pick*	$WA = r\ [rbg]$
(2)	*pick*	$NSW = r\ [rbg]$
(3)	*pick*	$T = r\ [rbg]$
(4)	*pick*	$NT = g\ [bg]$
(5)	$\stackrel{fc}{\Longrightarrow}$	$Q = b\ [b]$
(6)	*pick*	$V = b\ [b, g]$
(7)	$\stackrel{fc}{\Longrightarrow}$	$SA = \{\}\ []$

- backtrack to (5), pick $V = g \Longrightarrow$ (7) again
- backtrack to (3), pick $NT = b \stackrel{fc}{\Longrightarrow} Q = g \Longrightarrow$ same subtree (6)...
- backtrack to (2), pick $T = b \Longrightarrow$ same subtree (4)...
- backtrack to (2), pick $T = g \Longrightarrow$ same subtree (4)...

$\Longrightarrow$ backtrack to (1), then assign *NSW* another value

$\Longrightarrow$ 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

- failed branch:

	step	assignment [domain]
(1)	*pick*	$WA = r$ [rbg]
(2)	*pick*	$NSW = r$ [rbg]
(3)	*pick*	$T = r$ [rbg]
(4)	*pick*	$NT = g$ [bg]
(5)	$\stackrel{fc}{\Longrightarrow}$	$Q = b$ [b]
(6)	*pick*	$V = b$ [b, g]
(7)	$\stackrel{fc}{\Longrightarrow}$	$SA = \{\}$ []

- backtrack to (5), pick $V = g \Longrightarrow$ (7) again
- backtrack to (3), pick $NT = b \stackrel{fc}{\Longrightarrow} Q = g \Longrightarrow$ same subtree (6)...
- backtrack to (2), pick $T = b \Longrightarrow$ same subtree (4)...
- backtrack to (2), pick $T = g \Longrightarrow$ same subtree (4)...

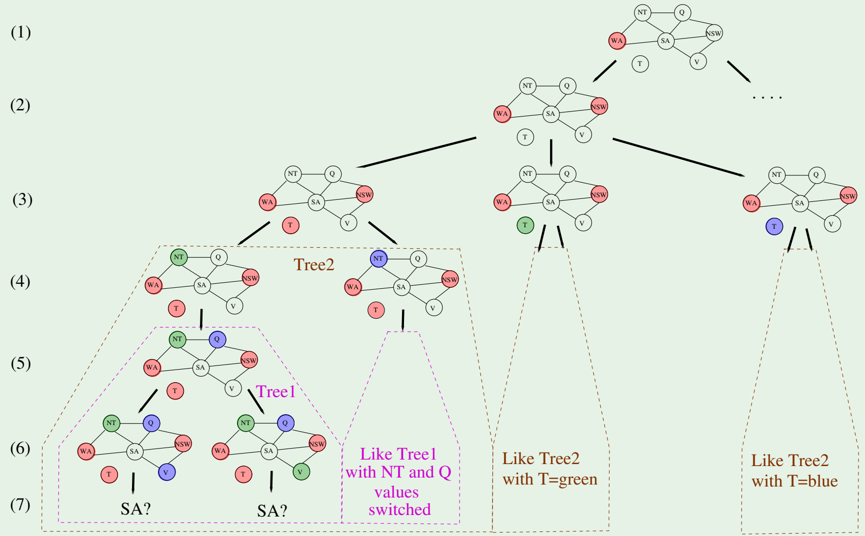
$\Longrightarrow$ backtrack to (1), then assign *NSW* another value

$\Longrightarrow$ 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.
    $\Longrightarrow$ domain of Q reduced to $\{b\}$ via f.c.
  - 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.
    $\Longrightarrow$ domain of Q reduced to $\{b\}$ via f.c.
  - 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 to the most-recently assigned element in nogood,
  3. change its value

$\Longrightarrow$ 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. backward-substitute deterministic unit assignments with their respective conflict set (until none is left)

$\Longrightarrow$ 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 to 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. backward-substitute deterministic unit assignments with their respective conflict set (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 to the most-recently assigned element in nogood,
  3. change its value

$\Longrightarrow$ 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. backward-substitute deterministic unit assignments with their respective conflict set (until none is left)

$\Longrightarrow$ 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 to the most-recently assigned element in nogood,
  3. change its value

$\Longrightarrow$ 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. backward-substitute deterministic unit assignments with their respective conflict set (until none is left)

$\Longrightarrow$ 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 to the most-recently assigned element in nogood,
  3. change its value

$\Longrightarrow$ 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. backward-substitute deterministic unit assignments with their respective conflict set (until none is left)

$\Longrightarrow$ 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 to the most-recently assigned element in nogood,
  3. change its value

$\Longrightarrow$ 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. backward-substitute deterministic unit assignments with their respective conflict set (until none is left)

$\Longrightarrow$ 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*	$WA = r$ [rbg]	← {}
(2) *pick*	$NSW = r$ [rbg]	← {}
(3) *pick*	$T = r$ [rbg]	← {}
(4) *pick*	$NT = g$ [bg]	← {$WA = r$}
(5) $\overset{fc}{\Longrightarrow}$	$Q = b$ [b]	← {$NSW = r, NT = g$}
(6) *pick*	$V = b$ [b, g]	← {$WA = r$}
(7) $\overset{fc}{\Longrightarrow}$	$SA = \emptyset$ []	← {$WA = r, NT = g, Q = b$}

- backward-substitute assignments

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

$\Longrightarrow$ backtrack till (3), then assign $NT = b$

$\Longrightarrow$ saves useless search on V values

Conflict-Driven Backjumping: Example

- failed branch:

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

- backward-substitute assignments

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

$\Longrightarrow$ backtrack till (3), then assign $NT = b$

$\Longrightarrow$ saves useless search on V values

Conflict-Driven Backjumping: Example

- failed branch:

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

- backward-substitute assignments

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

$\Longrightarrow$ backtrack till (3), then assign $NT = b$

$\Longrightarrow$ saves useless search on V values

Conflict-Driven Backjumping: Example

- failed branch:

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

- backward-substitute assignments

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

$\Longrightarrow$ backtrack till (3), then assign $NT = b$

$\Longrightarrow$ saves useless search on V values

Conflict-Driven Backjumping: Example [cont.]

- new failed branch:

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

- backward-substitute assignments

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

$\Longrightarrow$ backtrack till (1), then assign *NSW* another value

$\Longrightarrow$ saves useless search on *T* values

$\Longrightarrow$ overall, saves lots of search wrt. chronological backtracking

Conflict-Driven Backjumping: Example [cont.]

- new failed branch:

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

- backward-substitute assignments

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

$\Longrightarrow$ backtrack till (1), then assign *NSW* another value

$\Longrightarrow$ saves useless search on *T* values

$\Longrightarrow$ overall, saves lots of search wrt. chronological backtracking

Conflict-Driven Backjumping: Example [cont.]

- new failed branch:

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

- backward-substitute assignments

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

$\Longrightarrow$ backtrack till (1), then assign *NSW* another value

$\Longrightarrow$ saves useless search on T values

$\Longrightarrow$ overall, saves lots of search wrt. chronological backtracking

Conflict-Driven Backjumping: Example [cont.]

- new failed branch:

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

- backward-substitute assignments

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

$\Longrightarrow$ backtrack till (1), then assign *NSW* another value

$\Longrightarrow$ saves useless search on *T* values

$\Longrightarrow$ overall, saves lots of search wrt. chronological backtracking

Conflict-Driven Backjumping: Example [cont.]

- new failed branch:

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

- backward-substitute assignments

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

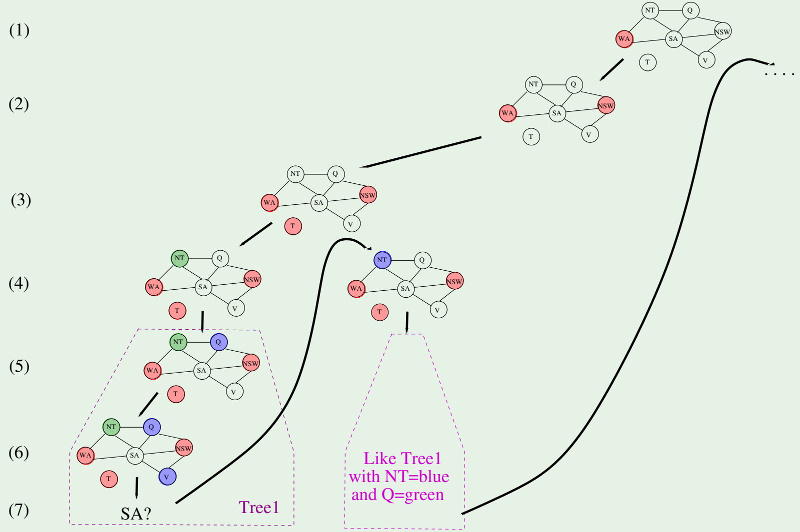
$\Longrightarrow$ backtrack till (1), then assign *NSW* another value

$\Longrightarrow$ saves useless search on *T* values

$\Longrightarrow$ 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. "$(WA \neq r)$ *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. "$(WA \neq r)$ *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. "$(WA \neq r)$ *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. "$(WA \neq r)$ *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

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

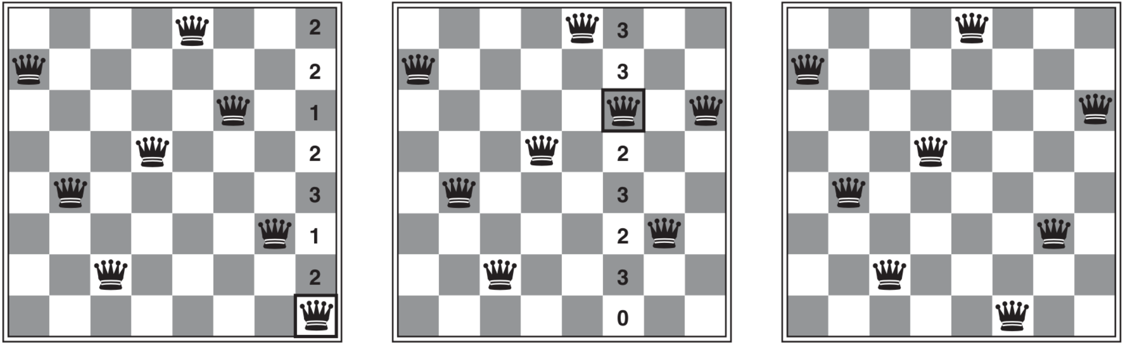
The Min-Conflicts Heuristic

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



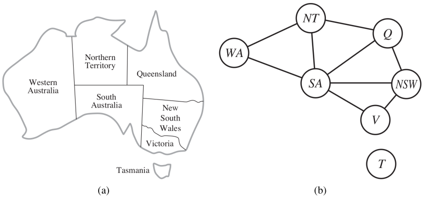
(© S. Russell & P. Norwig, AIMA)

Outline

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 w. 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 $4 \cdot 2^{20} \approx 4 \cdot 10^6$
    $\Longrightarrow$ from 4 billion years to 0.4 secs at 10million steps/sec

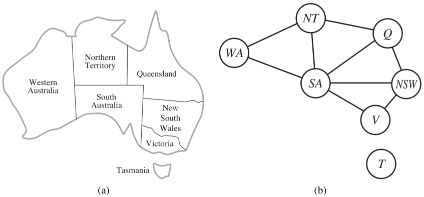


(© S. Russell & P. Norwig, AIMA)

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 w. 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 $4 \cdot 2^{20} \approx 4 \cdot 10^6$
    $\Longrightarrow$ from 4 billion years to 0.4 secs at 10million steps/sec

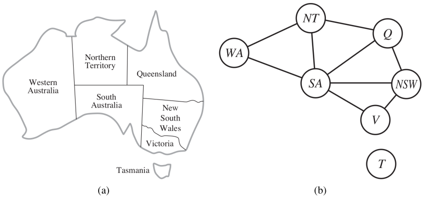


(© S. Russell & P. Norwig, AIMA)

Partitioning CFPs

"Divide & Conquer" CSPs

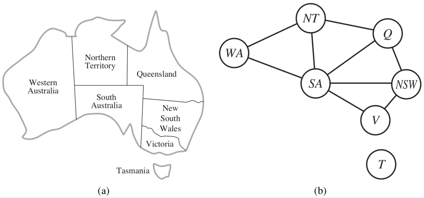
- Idea (when applicable): Partition a CSP into independent CSPs
  - identify strongly-connected components in constraint graph
  - e.g. by Tarjan's algorithms (linear!)
- Ex: Tasmania and mainland are independent subproblems
- E.g. partition n-variable CSP into n/c CSPs w. c variables each:
  - from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



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 w. 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 $4 \cdot 2^{20} \approx 4 \cdot 10^6$
    $\Longrightarrow$ 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

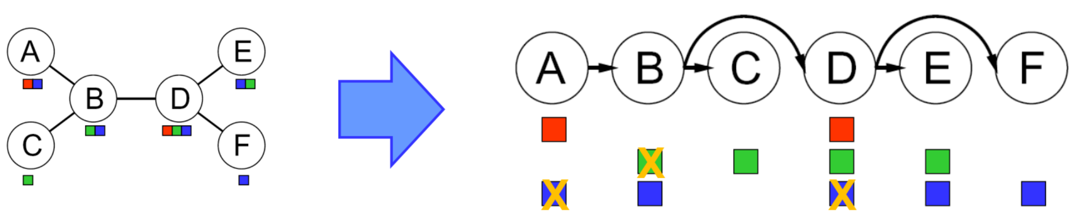
Solving Tree-structured CSPs

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. For $j \in 2..n$, assign X_j consistently with PARENT(X_j)



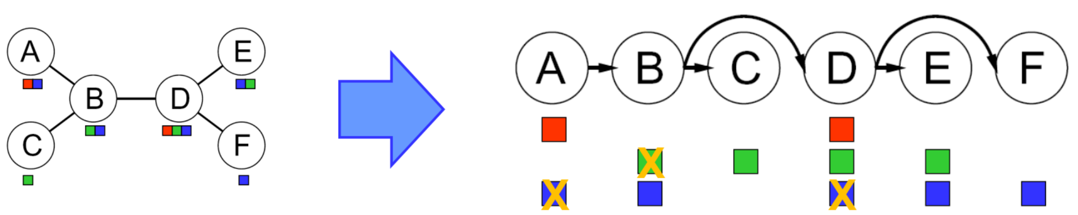
Solving Tree-structured CSPs

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. For $j \in 2..n$, assign X_j consistently with PARENT(X_j)



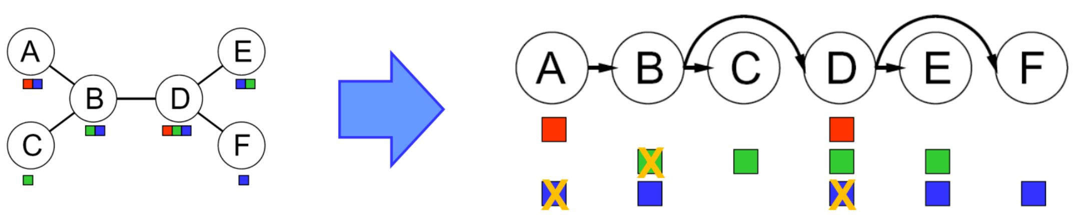
Solving Tree-structured CSPs

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. 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 ← number of variables in X
   assignment ← an empty assignment
   root ← any variable in X
   X ← 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] ← 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 $O(d^c \cdot (n-c)d^2)$

   $\Longrightarrow$ 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 $O(d^c \cdot (n-c)d^2)$

   $\Longrightarrow$ much smaller than d^n if c small

Solving Nearly Tree-Structured CSPs

Cutset Conditioning

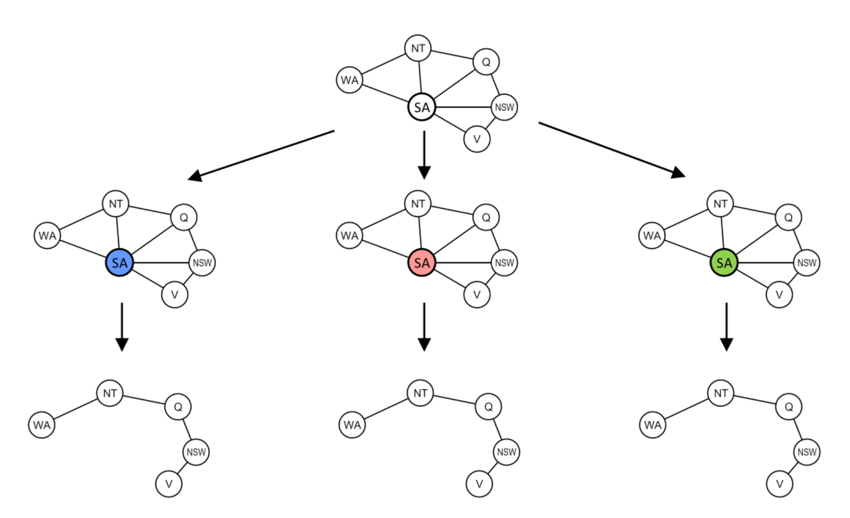
1. Identify a (small) cycle cutset S: a set of variables s.t. the remaining constraint graph is a tree
   - finding smallest cycle cutset is NP-hard
   - fast approximated techniques known
2. For each possible consistent assignment to the variables in S
   a) remove from the domains of the remaining 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 $O(d^c \cdot (n-c)d^2)$
   $\Longrightarrow$ 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 $O(d^c \cdot (n-c)d^2)$
   $\Longrightarrow$ much smaller than d^n if c small

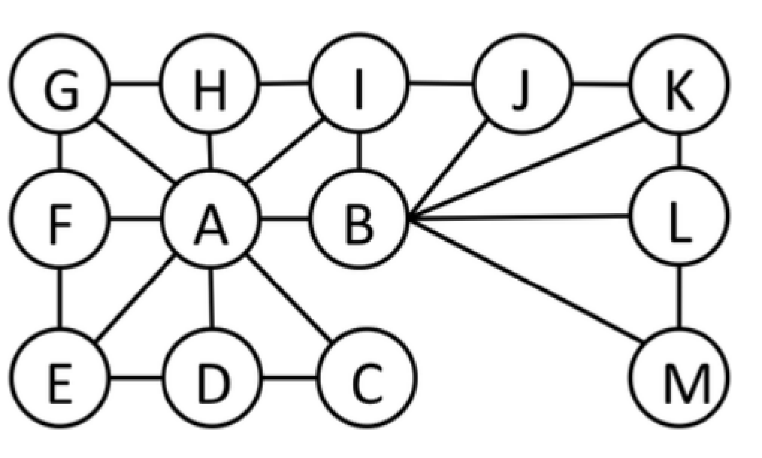
Cutset Conditioning: Example



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

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 color 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

  $\Longrightarrow$ 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: $WA < NT < SA$)

  $\Longrightarrow$ 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 color 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

  $\Longrightarrow$ 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: $WA < NT < SA$)

  $\Longrightarrow$ 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 color 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

  $\Longrightarrow$ 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: $WA < NT < SA$)

  $\Longrightarrow$ 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 color 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

  $\Longrightarrow$ 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: $WA < NT < SA$)

  $\Longrightarrow$ only one solution out of $n!$