Lecture 6: Constraint Satisfaction

Davide Grossi

university of
groningen

PART I
Constraint-Satisfaction Problems

A map-coloring problem

Can you assign a color (blue, red or green) to every state in the map so that no two adjacent states have the same color?

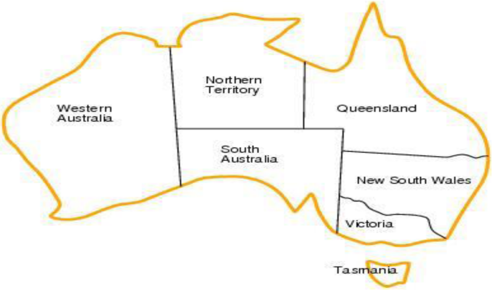
Map coloring

- Variables: {WA, NT, Q, NSW, V, SA, T}

Map coloring

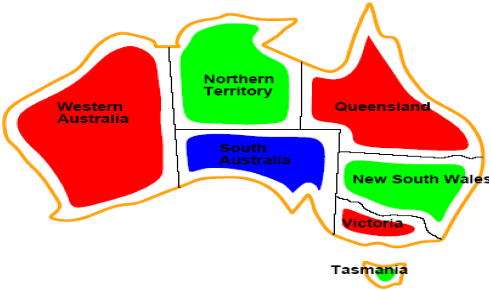
- Variables: {WA, NT, Q, NSW, V, SA, T}
- Domain: { red, green, blue } for all variables

Map coloring



- Variables: {WA, NT, Q, NSW, V, SA, T}
- Domain: { red, green, blue } for all variables
- Constraints: $\begin{cases} WA \neq NT & WA \neq SA & SA \neq NT \\ SA \neq Q & NT \neq Q & Q \neq NSW \\ NSW \neq V & & \end{cases}$

Map coloring



A solution assigns a color to each variable in a way that satisfies the constraints

- SA = blue
- WA, Q, V = red
- NT, NSW, T = green

A cryptarithmetic problem

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

- Variables: {T, W, O, F, U, R, X1, X2, X3}
- Domains: T, W, O, F, U, R $\in \{0, \ldots, 9\}$; X1, X2, X3 $\in \{0, 1\}$

Constraints[1]

- Alldiff(T,W,O,F,U,R), that is, all values should be different
- O+O = R + X1*10
- X1+W+W = U + X2*10
- X2+T+T = O + X3*10
- X3 = F

[1] A cryptoarithmetic solver

Constraint Satisfaction Problems: Definition

A **Constraint Satisfaction Problem** (CSP) consists of

- A finite set of variables $\{V_1, \ldots, V_n\}$
- A list of non-empty domains $(D_1, \ldots, D_n)$ for each variable
- A finite set of constraints $C_1, \ldots, C_m$.

Constraint Satisfaction Problems: Definition

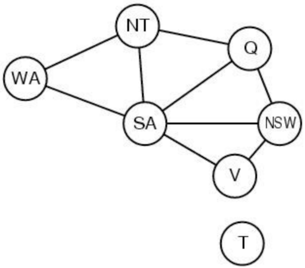
A **Constraint Satisfaction Problem** (CSP) consists of

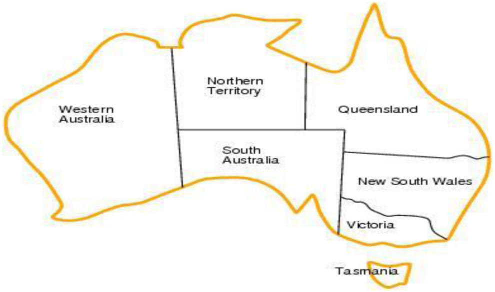
- A finite set of variables $\{V_1, \dots, V_n\}$
- A list of non-empty domains $(D_1, \dots, D_n)$ for each variable
- A finite set of constraints $C_1, \dots, C_m$.

A **state** is an assignment of values to (some or all) variables V_i

- An assignment is **complete** if $V_i \in D_i$ for all i
- It is **consistent** (or legal) if it satisfies all constraints C_j
- A **solution** is a complete and legal assignment

Constraint graph

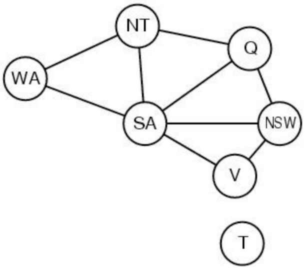


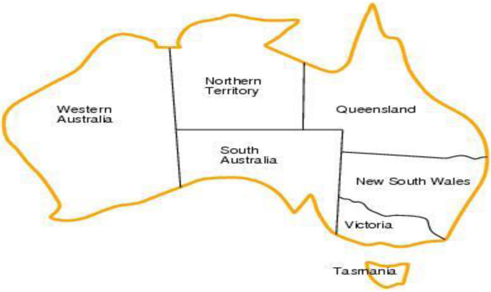


The **constraint graph** represents variables as nodes and constraints as edges

- Graphs can simplify the problem structure

Constraint graph

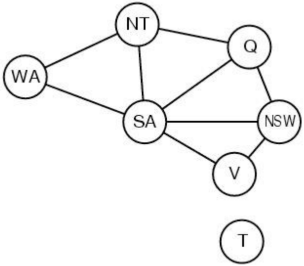


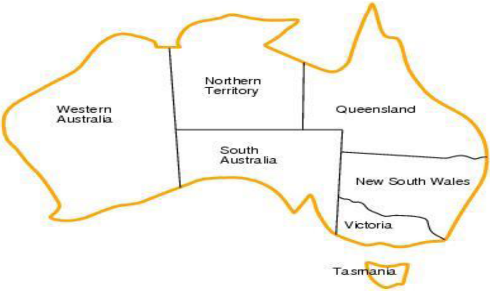


The **constraint graph** represents variables as nodes and constraints as edges

- Graphs can simplify the problem structure
- QUESTION What feature of the problem does the graph make immediately explicit?

Constraint graph



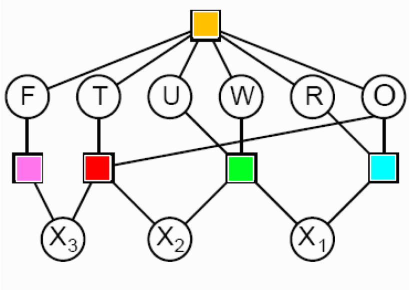


The **constraint graph** represents variables as nodes and constraints as edges

- Graphs can simplify the problem structure
- QUESTION What feature of the problem does the graph make immediately explicit?
- Tasmania is an independent subproblem

Constraint graph

```
      T W O
+     T W O
  ---------
    F O U R
```



Constraints

- Alldiff(T,W,O,F,U,R)
- O+O = R + X1*10
- X1+W+W = U + X2*10
- X2+T+T = O + X3*10
- X3 = F

Constraints

Constraints are of different types

- Unary constraints
  - e.g. $X \neq 0$
- Binary constraints
  - e.g. $X \neq Y$
- Higher-order constraints
  - e.g. $X + Y = A + B$

Constraints

Constraints are of different types

- Unary constraints
  - e.g. $X \neq 0$
- Binary constraints
  - e.g. $X \neq Y$
- Higher-order constraints
  - e.g. $X + Y = A + B$
- QUESTION What type of constraints did we handle in the map coloring problem?

Types of Constraint Satisfaction Problems

- Discrete (for *n* variables)

Types of Constraint Satisfaction Problems

- Discrete (for n variables)
  - Finite domains
    - e.g. common domain of size d, d^n possible assignments
    - $\boxed{\text{QUESTION}}$ What problem in propositional logic is of this type?

Types of Constraint Satisfaction Problems

- Discrete (for n variables)
  - Finite domains
    - e.g. common domain of size d, d^n possible assignments
    - $\boxed{\text{QUESTION}}$ What problem in propositional logic is of this type? Satisfiability!

Types of Constraint Satisfaction Problems

- Discrete (for *n* variables)
  - Finite domains
    - e.g. common domain of size d, d^n possible assignments
    - $\boxed{\text{QUESTION}}$ What problem in propositional logic is of this type? Satisfiability!
  - Infinite domains (integers)
    - e.g. job scheduling with integer job start times

Types of Constraint Satisfaction Problems

- Discrete (for *n* variables)
  - Finite domains
    - e.g. common domain of size d, d^n possible assignments
    - $\boxed{\text{QUESTION}}$ What problem in propositional logic is of this type? Satisfiability!
  - Infinite domains (integers)
    - e.g. job scheduling with integer job start times
    - $\boxed{\text{QUESTION}}$ What's the problem with infinite domains?

Types of Constraint Satisfaction Problems

- Discrete (for *n* variables)
  - Finite domains
    - e.g. common domain of size d, d^n possible assignments
    - $\boxed{\text{QUESTION}}$ What problem in propositional logic is of this type? Satisfiability!
  - Infinite domains (integers)
    - e.g. job scheduling with integer job start times
    - $\boxed{\text{QUESTION}}$ What's the problem with infinite domains?
    - We cannot possibly enumerate all solutions! We need a constraint language to describe them, e.g. $StartJob_1 + 5 \leq StartJob_2$

Types of Constraint Satisfaction Problems

- Discrete (for n variables)
  - Finite domains
    - e.g. common domain of size d, d^n possible assignments
    - $\boxed{\text{QUESTION}}$ What problem in propositional logic is of this type? Satisfiability!
  - Infinite domains (integers)
    - e.g. job scheduling with integer job start times
    - $\boxed{\text{QUESTION}}$ What's the problem with infinite domains?
    - We cannot possibly enumerate all solutions! We need a constraint language to describe them, e.g. $StartJob_1 + 5 \leq StartJob_2$
- Continuous variables

PART II

Solving CSPs (backtracking)

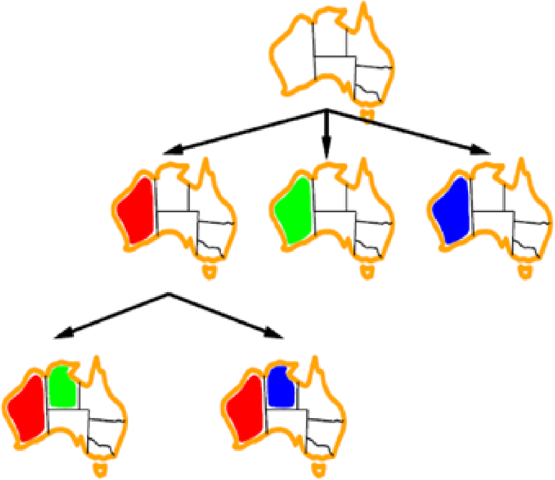
Solving CSP: Search

A CSP can be expressed as a search problem

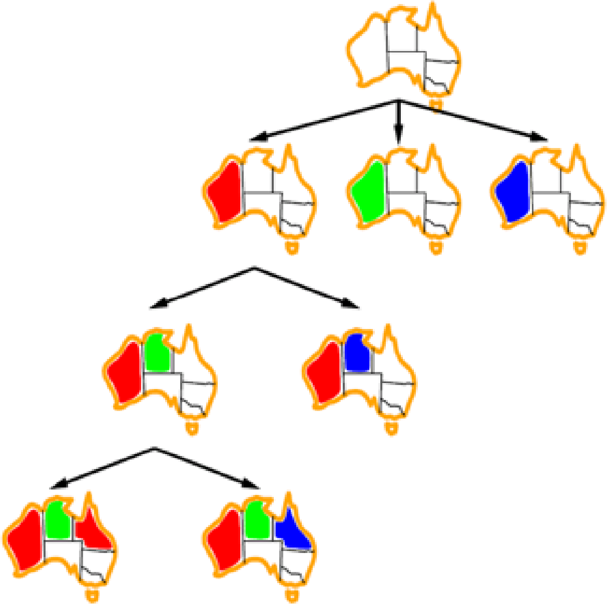
- **Initial state**: Empty assignment
- **Actions**: Assign a value to an unassigned variable so that no constraints are violated
- **Goal test**: Is the complete assignment a solution?

(Part of) the *search tree* for the map coloring problem

(Part of) the *search tree* for the map coloring problem



(Part of) the *search tree* for the map coloring problem



Search for CSP: General Features

The solution of a CSP (with **finite** domains) with n variables is found at depth n

- Depth-first search is an option
- For a finite domain size d CSP, the top level has $n \cdot d$ successors
  - At depth k, there are $(n - k) \cdot d$ successors
- In total, there are $n!d^n$ leaves

Search for CSP: General Features

The solution of a CSP (with **finite** domains) with n variables is found at depth n

- Depth-first search is an option
- For a finite domain size d CSP, the top level has $n \cdot d$ successors
  - At depth k, there are $(n-k) \cdot d$ successors
- In total, there are $n!d^n$ leaves
  - $\boxed{\text{QUESTION}}$ Is it a good idea to set up the search problem like this?

Search for CSP: General Features

The solution of a CSP (with **finite** domains) with n variables is found at depth n

- Depth-first search is an option
- For a finite domain size d CSP, the top level has $n \cdot d$ successors
  - At depth k, there are $(n - k) \cdot d$ successors
- In total, there are $n!d^n$ leaves
  - $\boxed{\text{QUESTION}}$ Is it a good idea to set up the search problem like this? There are only d^n possible assignments!
- The order of assignment does not affect the outcome!

Search for CSP: General Features

The solution of a CSP (with **finite** domains) with n variables is found at depth n

- Depth-first search is an option
- For a finite domain size d CSP, the top level has $n \cdot d$ successors
  - At depth k, there are $(n-k) \cdot d$ successors
- In total, there are $n!d^n$ leaves
  - QUESTION Is it a good idea to set up the search problem like this? There are only d^n possible assignments!
- The order of assignment does not affect the outcome!

e.g. In the map-coloring problem it does not matter whether I assign blue to WA before assigning red to SA

Search for CSP: General Features

The solution of a CSP (with **finite** domains) with n variables is found at depth n

- Depth-first search is an option
- For a finite domain size d CSP, the top level has $n \cdot d$ successors
  - At depth k, there are $(n-k) \cdot d$ successors
- In total, there are $n!d^n$ leaves
  - $\boxed{\text{QUESTION}}$ Is it a good idea to set up the search problem like this? There are only d^n possible assignments!
- The order of assignment does not affect the outcome!

e.g. In the map-coloring problem it does not matter whether I assign blue to WA before assigning red to SA

- CSP search problems are said to be **commutative**

Search for CSP: General Features

The solution of a CSP (with **finite** domains) with n variables is found at depth n

- Depth-first search is an option
- For a finite domain size d CSP, the top level has $n \cdot d$ successors
  - At depth k, there are $(n - k) \cdot d$ successors
- In total, there are $n!d^n$ leaves
  - QUESTION Is it a good idea to set up the search problem like this? There are only d^n possible assignments!
- The order of assignment does not affect the outcome!

e.g. In the map-coloring problem it does not matter whether I assign blue to WA before assigning red to SA

- CSP search problems are said to be **commutative**
  QUESTION Have we already encountered commutative search problems?

Search for CSP: General Features

The solution of a CSP (with **finite** domains) with n variables is found at depth n

- Depth-first search is an option
- For a finite domain size d CSP, the top level has $n \cdot d$ successors
  - At depth k, there are $(n - k) \cdot d$ successors
- In total, there are $n!d^n$ leaves
  - QUESTION Is it a good idea to set up the search problem like this? There are only d^n possible assignments!
- The order of assignment does not affect the outcome!

e.g. In the map-coloring problem it does not matter whether I assign blue to WA before assigning red to SA

- CSP search problems are said to be **commutative**
  QUESTION Have we already encountered commutative search problems? What kind of techniques did we use to solve them?

Backtracking search in CSP

- $\boxed{\text{IDEA}}$ Exploiting commutativity, choose values for one variable at a time, and backtrack when a variable has no legal values to assign

Backtracking search in CSP

- $\boxed{\text{IDEA}}$ Exploiting commutativity, choose values for one variable at a time, and backtrack when a variable has no legal values to assign
- = *Depth First Search (DFS) with backtracking*

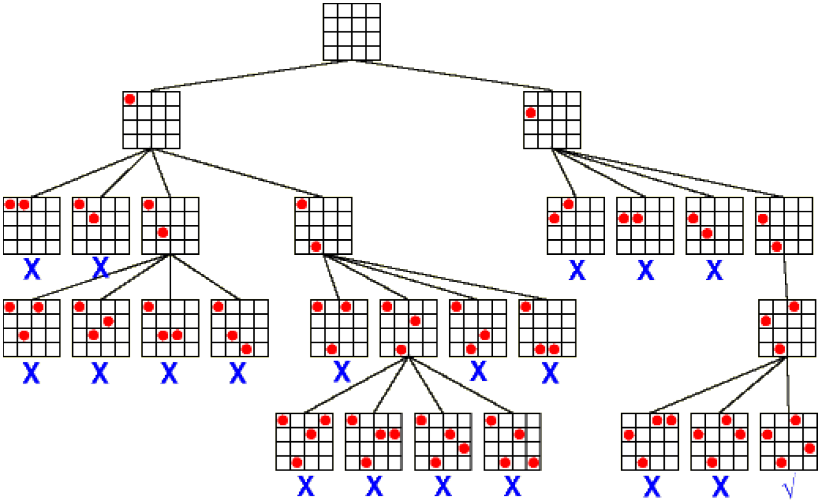
Basic (naive) Backtracking Search in CSP: Pseudocode

```
function BACKTRACK (assignment, csp) returns solution
    if assignment is complete then return assignment
    var ← SELECTUNASSIGNEDVARIABLE (csp)
    for each value in ORDERDOMAIN(var, assignment, csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
                result ← BACKTRACK(assignment, csp)
                if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

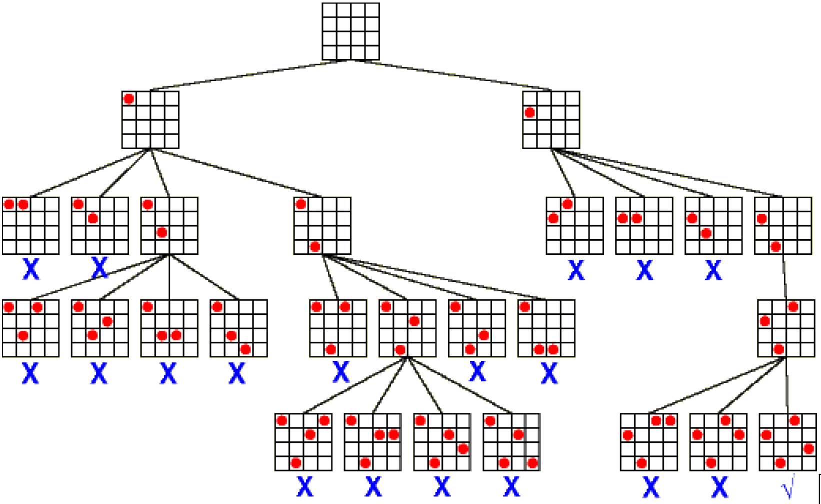
Basic (naive) Backtracking Search in CSP: Pseudocode

```
function BACKTRACK (assignment, csp) returns solution
    if assignment is complete then return assignment
    var ← SELECTUNASSIGNEDVARIABLE (csp)
    for each value in ORDERDOMAIN(var, assignment, csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
                result ← BACKTRACK(assignment, csp)
                if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

Example: 4 Queens Problem



Example: 4 Queens Problem



QUES

Is this satisfactory?

Example: Sudoku

	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		

Solving Sudoku by backtracking, sketch:

- Select an empty square
- Fill in a valid integer
- If no valid integer exists, backtrack
- Repeat until a solution is found

Example: Sudoku

	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		

Solving Sudoku by backtracking, sketch:

- Select an empty square
- Fill in a valid integer
- If no valid integer exists, backtrack
- Repeat until a solution is found

QUESTION Is this satisfactory?

PART III
Solving CSPs
(heuristics)

Backtracking efficiency

How can we increase the efficiency of backtracking?
Three approaches:

Backtracking efficiency

How can we increase the efficiency of backtracking?
Three approaches:

- What **variable** should be assigned first?
  var $\leftarrow$ SELECT-UNASSIGNED-VARIABLE (*csp*)
- What **value** should be tried first?
  for each *value* **in** ORDERDOMAIN(*var, assignment, csp*)
- Can we **detect failure** that are inevitable earlier in the search process?

Variable selection: MRV heuristic

- What **variable** should be assigned first?

Variable selection: MRV heuristic

- What **variable** should be assigned first?
- $\boxed{\text{QUESTION}}$ Any ideas?

Variable selection: MRV heuristic

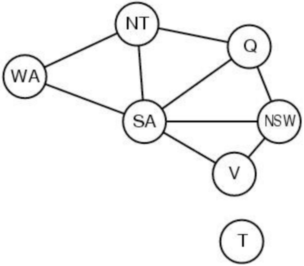
- What **variable** should be assigned first?
- $\boxed{\text{QUESTION}}$ Any ideas?
- $\boxed{\text{INTUITION 1}}$ Choose the variable that has the fewest possible options
- Also known as the *most constrained variable* (MRV) heuristic

Minimum remaining values: example

1 4 5 6 9	7	1 4 6 9	1 4 5 6	8	1 2 4 5	3	1 2 5 9	1 4 5 9
2	1 6 9	1 4 6 9	1 3 4 5 6 7	1 4 5 6 7	1 3 4 5 7	1 5 6 9	1 5 9	8
3	**1 6**	1 4 6 8	9	1 4 5 6	1 2 4 5	1 2 5 6	7	1 4 5
1 4 6 7	1 2 3 6	1 3 4 6	1 3 4 5 6 7 8	9	1 3 4 5 7 8	1 5 8	1 3 5	1 3 5
1 6 9	1 3 6 9	5	1 3 6 8	2	1 3 8	1 8 9	4	7
1 4 7 9	8	1 3 4 9	1 3 4 5 7	1 4 5 7	1 3 4 5 7	1 5 9	6	2
1 9	1 3 9	2	1 4 5 7	1 4 5 7	6	1 5 7 9	8	1 3 5 9
1 8	4	7	1 5 8	3	9	1 2 5	1 2 5	6
1 6 8 9	5	1 3 6 8 9	2	**1 7**	1 7 8	4	1 3 9	1 3 9

Variable selection: Degree heuristic

- What **variable** should be assigned first?
- QUESTION Any ideas?
- INTUITION 2 Select the variable that is involved in the largest number of constraints with unassigned variables



Value selection: LCV heuristic

- What **value** should be tried first?

Value selection: LCV heuristic

- What **value** should be tried first?
- $\boxed{\text{QUESTION}}$ Any ideas?

Value selection: LCV heuristic

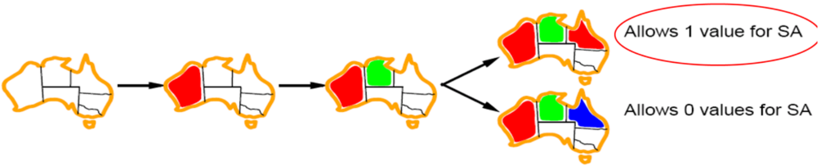
- What **value** should be tried first?
- $\boxed{\text{QUESTION}}$ Any ideas?
- $\boxed{\text{IDEA}}$ Select the value that leaves the most values open for other unassigned variables (*least constraining value*, LCV)

Value selection: LCV heuristic

- What **value** should be tried first?
- $\boxed{\text{QUESTION}}$ Any ideas?
- $\boxed{\text{IDEA}}$ Select the value that leaves the most values open for other unassigned variables (*least constraining value*, LCV) ... the one that leaves the most values open for other unassigned variables

Value selection: LCV heuristic

- What **value** should be tried first?
- QUESTION Any ideas?
- IDEA Select the value that leaves the most values open for other unassigned variables (*least constraining value*, LCV) . . . the one that leaves the most values open for other unassigned variables



Early failure detection: Forward checking

- Can we **detect failures** that are inevitable earlier in the search process?

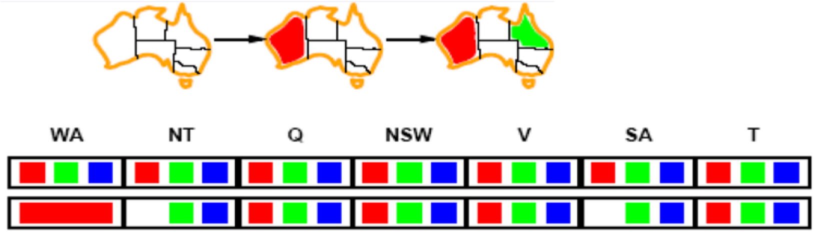
Early failure detection: Forward checking

- Can we **detect failures** that are inevitable earlier in the search process?
- **Forward checking**
  - Track remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values left

Early failure detection: Forward checking

Initially, everything is possible

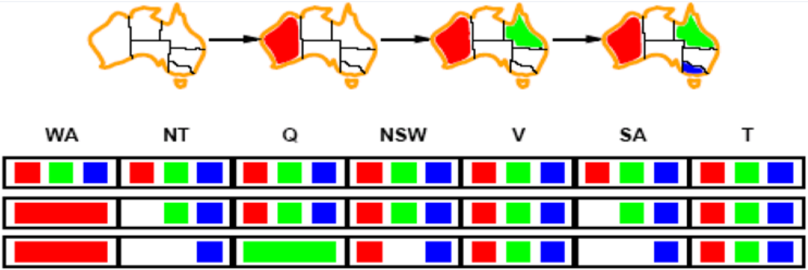
Early failure detection: Forward checking



Assign red to WA

- NT can no longer be red
- SA can no longer be red

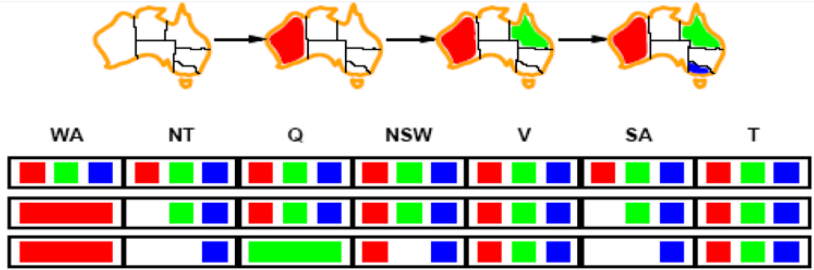
Early failure detection: Forward checking



Assign green to Q

- NT can no longer be green
- NSW can no longer be green
- SA can no longer be green

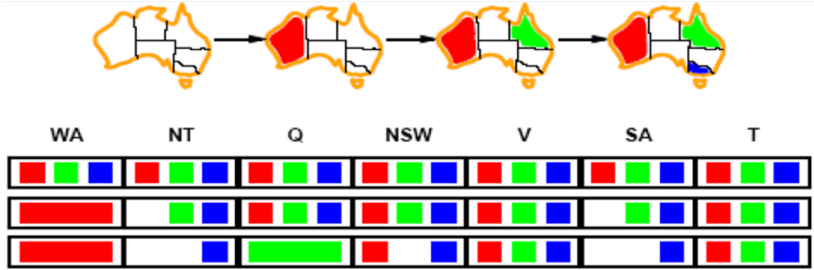
Early failure detection: Forward checking



Try now to assign blue to V

- Forward checking detects that SA would have no valid values remaining (failure).

Early failure detection: Forward checking

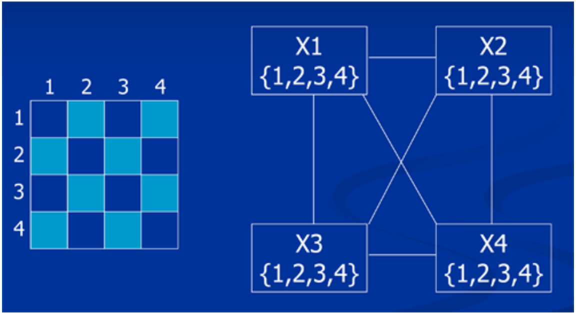


Try now to assign blue to V

- Forward checking detects that SA would have no valid values remaining (failure).

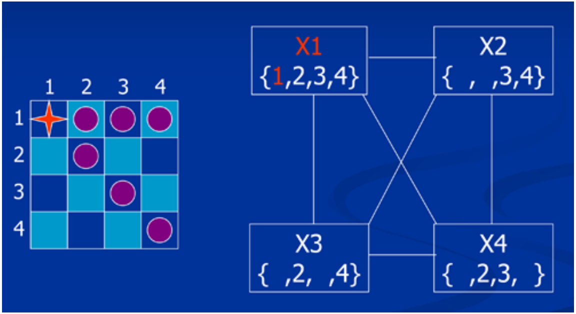
In general, combining heuristics can lead to better performance

4 Queens Problem: Retake



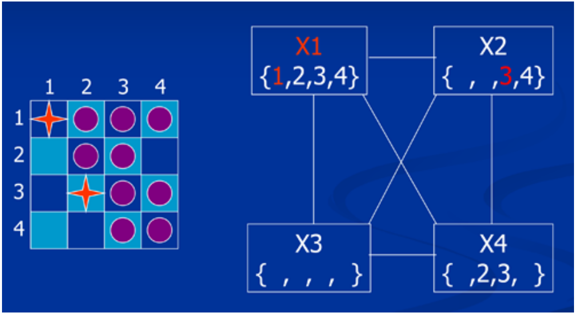
X*i* is the row number of the queen in column *i*

Four queens problem



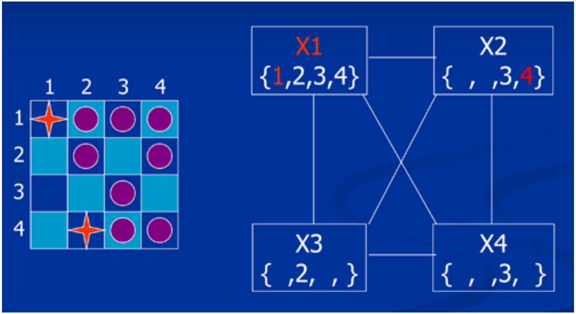
Visited states: 1

Four queens problem: forward checking



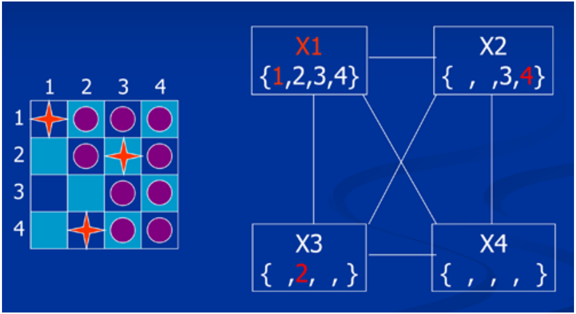
Visited states: 2

Four queens problem: forward checking



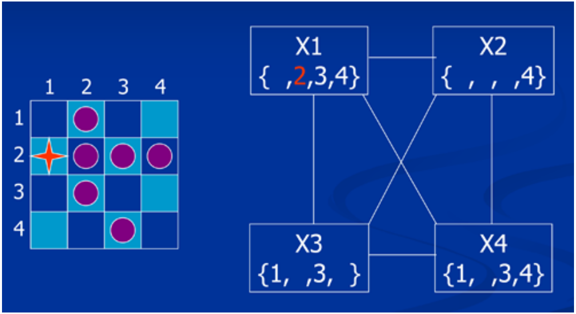
Visited states: 3

Four queens problem: forward checking



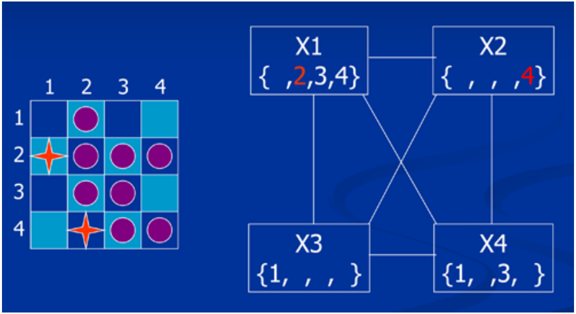
Visited states: 4

Four queens problem: forward checking



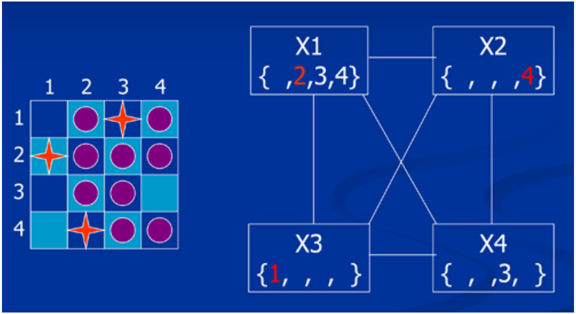
Visited states: 5

Four queens problem: forward checking



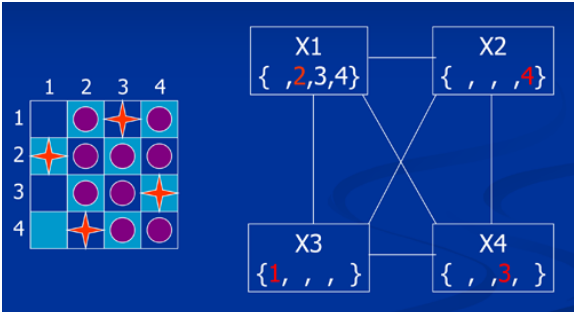
Visited states: 6

Four queens problem: forward checking



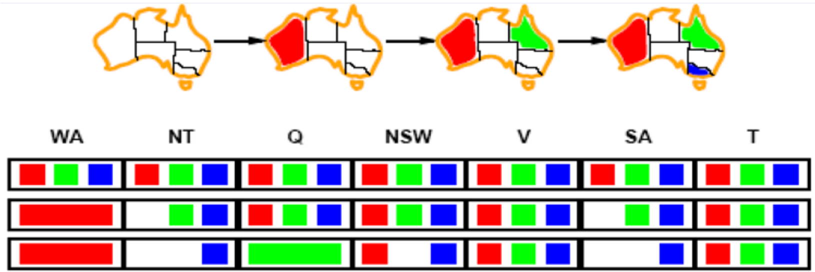
Visited states: 7

Four queens problem: forward checking



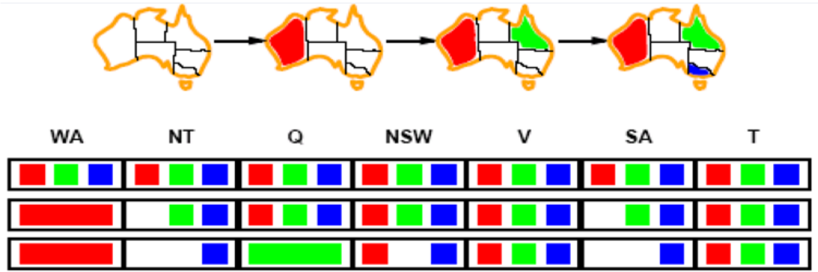
Visited states: 8

Constraint propagation



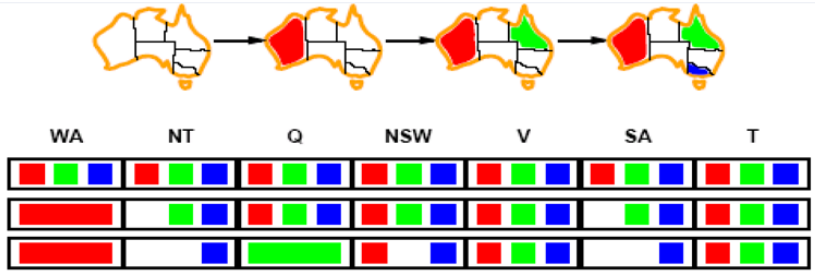
- Forward checking (FC) propagates information from assigned to unassigned variables, but does not detect all future failures
  - Example: NT and SA are adjacent and cannot both be blue

Constraint propagation



- Forward checking (FC) propagates information from assigned to unassigned variables, but does not detect all future failures
  - Example: NT and SA are adjacent and cannot both be blue
  - Forward checking checks only constraints involving the current variable

Constraint propagation



- Forward checking (FC) propagates information from assigned to unassigned variables, but does not detect all future failures
  - Example: NT and SA are adjacent and cannot both be blue
  - Forward checking checks only constraints involving the current variable
- **Constraint propagation** iteratively enforces constraints

PART IV
Solving CSPs
(inference)

Backtracking search with inference (constraint propagation)

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

function BACKTRACK (assignment, csp) returns solution
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE (csp)
    for each value in DOMAIN(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}, inferences from assignment
    return failure
```

university of groningen

Node consistency

- A variable X is **node consistent** iff each value of X satisfies all unary constraints on X
- Node consistency can be applied as a *preprocessing* step before starting search to remove all the node inconsistent values
  - Effectively changes the domain D_i of a variable X_i

Arc consistency

- **Arc consistency** (AC) deals with binary constraints
- X is arc consistent w.r.t. Y iff
  - for every value $x \in D_X$, there exists some legal $y \in D_Y$

Arc consistency

- **Arc consistency** (AC) deals with binary constraints
- X is arc consistent w.r.t. Y iff
  - <u>for every</u> value $x \in D_X$, <u>there exists</u> some legal $y \in D_Y$
- Example: two variables X and Y, both with digit domains
  - $D_X = D_Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Constraint: $Y = X^2$
  - It amounts to $(X, Y) \in \{(0, 0), (1, 1), (2, 4), (3, 9)\}$

Arc consistency

- **Arc consistency** (AC) deals with binary constraints
- X is arc consistent w.r.t. Y iff
  - for every value $x \in D_X$, there exists some legal $y \in D_Y$
- Example: two variables X and Y, both with digit domains
  - $D_X = D_Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Constraint: $Y = X^2$
  - It amounts to $(X, Y) \in \{(0, 0), (1, 1), (2, 4), (3, 9)\}$
- So to make X arc consistent w.r.t. Y we reduce D_X to $\{0, 1, 2, 3\}$

Arc consistency

- **Arc consistency** (AC) deals with binary constraints
- X is arc consistent w.r.t. Y iff
  - for every value $x \in D_X$, there exists some legal $y \in D_Y$
- Example: two variables X and Y, both with digit domains
  - $D_X = D_Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Constraint: $Y = X^2$
  - It amounts to $(X, Y) \in \{(0, 0), (1, 1), (2, 4), (3, 9)\}$
- So to make X arc consistent w.r.t. Y we reduce D_X to $\{0, 1, 2, 3\}$
- If we make X arc consistent w.r.t. Y *and* Y w.r.t. X, then $D_X = \{0, 1, 2, 3\}$ and $D_y = \{0, 1, 4, 9\}$

university of groningen

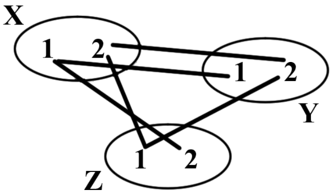
Arc consistency

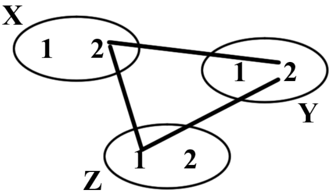
- **Arc consistency** (AC) deals with binary constraints
- X is arc consistent w.r.t. Y iff
  - for every value $x \in D_X$, there exists some legal $y \in D_Y$
- Example: two variables X and Y, both with digit domains
  - $D_X = D_Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Constraint: $Y = X^2$
  - It amounts to $(X, Y) \in \{(0, 0), (1, 1), (2, 4), (3, 9)\}$
- So to make X arc consistent w.r.t. Y we reduce D_X to $\{0, 1, 2, 3\}$
- If we make X arc consistent w.r.t. Y *and* Y w.r.t. X, then $D_X = \{0, 1, 2, 3\}$ and $D_y = \{0, 1, 4, 9\}$
- The constraint graph of a CSP is **arc consistent** if every variable is arc consistent w.r.t. any other variable

Arc consistency: example

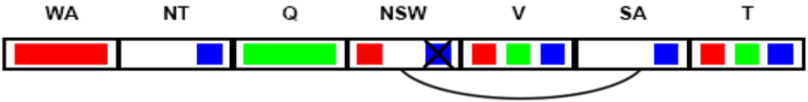
- Problem:

$$D_X = D_Y = D_Z = \{1, 2\}, X = Y, X \neq Z, Y > Z$$



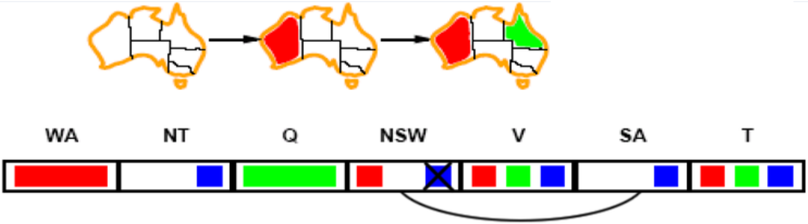


Arc consistency: example



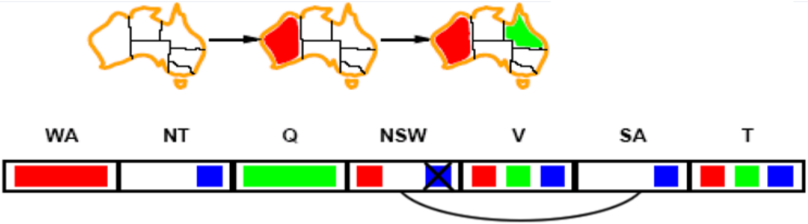
- *X* is arc consistent w.r.t. *Y* iff
  - *for every* value *x* of *X*, *there exists* some legal *y* of *Y*
- QUESTION Is *SA* arc consistent w.r.t. *NSW*?

Arc consistency: example



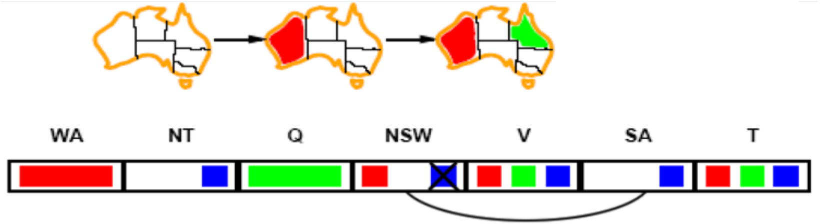
- *X* is arc consistent w.r.t. *Y* iff
  - *for every* value *x* of *X*, *there exists* some legal *y* of *Y*
- QUESTION Is *SA* arc consistent w.r.t. *NSW*?
  - if *SA* = *blue*, then *NSW* = *red* is a legal choice

Arc consistency: example



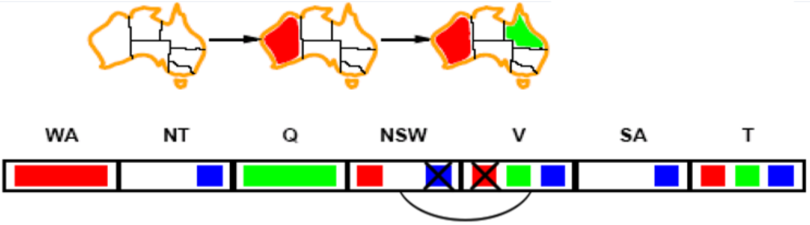
- *X* is arc consistent w.r.t. *Y* iff
  - *for every* value *x* of *X*, *there exists* some legal *y* of *Y*
- QUESTION Is *SA* arc consistent w.r.t. *NSW*?
  - if *SA* = *blue*, then *NSW* = *red* is a legal choice
- QUESTION And vice-versa?

Arc consistency: example



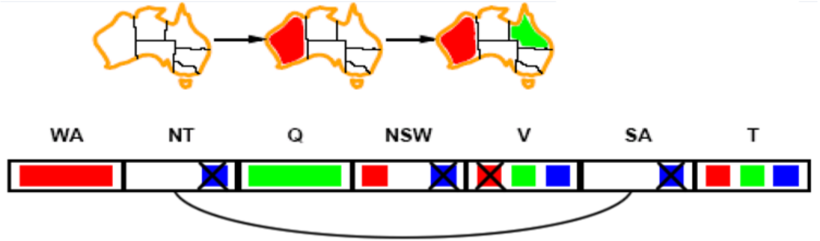
- *X* is arc consistent w.r.t. *Y* iff
  - *for every* value *x* of *X*, *there exists* some legal *y* of *Y*
- QUESTION Is *SA* arc consistent w.r.t. *NSW*?
  - if *SA* = *blue*, then *NSW* = *red* is a legal choice
- QUESTION And vice-versa?
  - if *NSW* = *red*, then *SA* = *blue* is a legal choice
  - if *NSW* = *blue*, then *SA* has no legal value

Arc consistency: example



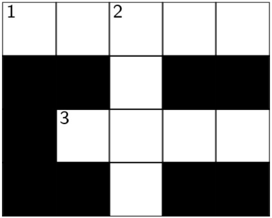
- *X* is arc consistent w.r.t. *Y* iff
  - *for every* value *x* of *X*, *there exists* some legal *y* of *Y*
- By removing *blue* from *NSW*, *V* becomes inconsistent w.r.t. *NSW*
  - if *V* = *red*, then *NSW* has no legal value

Arc consistency: example



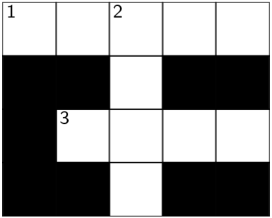
- *X* is arc consistent w.r.t. *Y* iff
  - *for every* value *x* of *X*, *there exists* some legal *y* of *Y*
- Inconsistency of *SA* w.r.t. *NT* and *NT* w.r.t. *SA* detects failure earlier than forward checking

Arc consistency: example



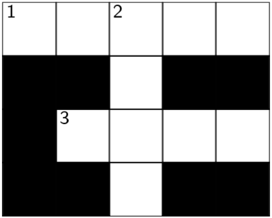
X1	X2	X3
astar	live	live
happy	load	load
hello	peal	peal
hoses	peel	peel
	save	save
	talk	talk

Arc consistency: example



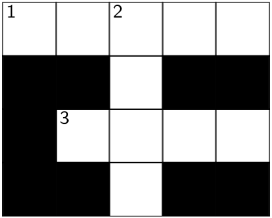
X1	X2	X3
astar	~~live~~	live
happy	load	load
hello	peal	peal
hoses	peel	peel
	~~save~~	save
	~~talk~~	talk

Arc consistency: example



X1	X2	X3
~~astar~~	~~live~~	live
happy	load	load
hello	peal	peal
~~hoses~~	peel	peel
	~~save~~	save
	~~talk~~	talk

Arc consistency: example



X1	X2	X3
~~astar~~	~~live~~	~~live~~
happy	load	~~load~~
hello	peal	peal
~~hoses~~	peel	peel
	~~save~~	save
	~~talk~~	talk

Arc consistency: algorithm

function AC-3(*csp*) **returns** *csp* with inconsistencies removed
  inputs: *csp*, a binary CSP with components (X, D, C)
  local variables: *queue*, initially all the arcs in *csp*
  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
      add(X_k, X_i) to *queue*

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise D_{X_i}
  revised $\leftarrow$ false
  for each x **in** D_i **do**
    if no value y in D_j results in legal (x, y) **then**
      delete x from D_i
      revised $\leftarrow$ true
  return revised

Arc consistency

Arc consistency is applied

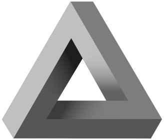
- As a preprocessing step before the search
  - To reduce size of search tree
  - To identify inconsistencies
  - May solve the problem without search!

Arc consistency

Arc consistency is applied

- As a preprocessing step before the search
  - To reduce size of search tree
  - To identify inconsistencies
  - May solve the problem without search!
- Or after an assignment, in order to do early failure detection, called **MAC** (Maintaining Arc Consistency)

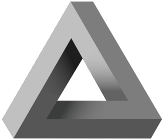
Arc consistency



Is arc consistency enough?

- If a domain is empty, there is no solution
- If every domain has at least one legal value, then there may still be a solution

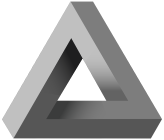
Arc consistency



Is arc consistency enough?

- If a domain is empty, there is no solution
- If every domain has at least one legal value, then there may still be a solution
- However:
  $D_X = D_Y = D_Z = \{1, 2\}, X \neq Y, Y \neq Z, X \neq Z$
  - $\boxed{\text{QUESTION}}$ Is this problem arc consistent?

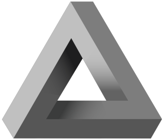
Arc consistency



Is arc consistency enough?

- If a domain is empty, there is no solution
- If every domain has at least one legal value, then there may still be a solution
- However:
  $D_X = D_Y = D_Z = \{1, 2\}, X \neq Y, Y \neq Z, X \neq Z$
  - $\boxed{\text{QUESTION}}$ Is this problem arc consistent?
  - Yes

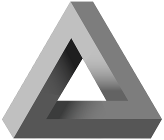
Arc consistency



Is arc consistency enough?

- If a domain is empty, there is no solution
- If every domain has at least one legal value, then there may still be a solution
- However:
  $D_X = D_Y = D_Z = \{1, 2\}, X \neq Y, Y \neq Z, X \neq Z$
  - QUESTION Is this problem arc consistent?
  - Yes
  - QUESTION Does it have a solution?

Arc consistency



Is arc consistency enough?

- If a domain is empty, there is no solution
- If every domain has at least one legal value, then there may still be a solution
- However:
  $D_X = D_Y = D_Z = \{1, 2\}, X \neq Y, Y \neq Z, X \neq Z$
  - $\boxed{\text{QUESTION}}$ Is this problem arc consistent?
  - Yes
  - $\boxed{\text{QUESTION}}$ Does it have a solution?
  - No

K-consistency

- Arc consistency does not detect all inconsistencies. The problem are inconsistencies which span more arcs
- K-consistency is a stronger form of **constraint propagation**

K-consistency

- Arc consistency does not detect all inconsistencies. The problem are inconsistencies which span more arcs
- K-consistency is a stronger form of **constraint propagation**
- A CSP is K**-consistent** if *for all* consistent assignments of any set of $K - 1$ variables, *there exists* a legal value for the K-th variable

K-consistency

- Arc consistency does not detect all inconsistencies. The problem are inconsistencies which span more arcs
- K-consistency is a stronger form of **constraint propagation**
- A CSP is K**-consistent** if *for all* consistent assignments of any set of $K - 1$ variables, *there exists* a legal value for the K-th variable
  - $\boxed{\text{QUESTION}}$ What is 1-consistency?

K-consistency

- Arc consistency does not detect all inconsistencies. The problem are inconsistencies which span more arcs
- K-consistency is a stronger form of **constraint propagation**
- A CSP is K**-consistent** if *for all* consistent assignments of any set of $K-1$ variables, *there exists* a legal value for the K-th variable
  - $\boxed{\text{QUESTION}}$ What is 1-consistency?
  - It is the same as node consistency
  - $\boxed{\text{QUESTION}}$ And 2-consistency?

K-consistency

- Arc consistency does not detect all inconsistencies. The problem are inconsistencies which span more arcs
- K-consistency is a stronger form of **constraint propagation**
- A CSP is K**-consistent** if *for all* consistent assignments of any set of $K-1$ variables, *there exists* a legal value for the K-th variable
  - $\boxed{\text{QUESTION}}$ What is 1-consistency?
  - It is the same as node consistency
  - $\boxed{\text{QUESTION}}$ And 2-consistency?
  - It is the same as arc consistency

Strong K-consistency

A graph is **strongly K-consistent** if

- the graph is K-consistent,
- the graph is $(K-1)$-consistent
- $\dots$
- the graph is 1-consistent

Strong K-consistency

A graph is **strongly K-consistent** if

- the graph is K-consistent,
- the graph is $(K-1)$-consistent
- ...
- the graph is 1-consistent

Suppose we have a problem with n variables that is strongly n-consistent

- A solution can be found in $O(n^2 d)$ time
  - Choose any value for X_1 (consistent since 1-consistent)
  - Choose a consistent value for X_2 (exists since 2-consistent)
  - etc.

Strong K-consistency

A graph is **strongly K-consistent** if

- the graph is K-consistent,
- the graph is $(K-1)$-consistent
- ...
- the graph is 1-consistent

Suppose we have a problem with n variables that is strongly n-consistent

- A solution can be found in $O(n^2d)$ time
  - Choose any value for X_1 (consistent since 1-consistent)
  - Choose a consistent value for X_2 (exists since 2-consistent)
  - etc.
- However, establishing n-consistency takes exponential time in n, in the worst case

Special constraints

- Global constraints: *AllDiff(...)*

Special constraints

- Global constraints: *AllDiff(...)*
- $\boxed{\text{QUESTION}}$ What kind of inferences can we draw from *AllDiff*?

Special constraints

- Global constraints: *AllDiff(...)*
- $\boxed{\text{QUESTION}}$ What kind of inferences can we draw from *AllDiff*?
  - If there are fewer possible values than variables, inconsistency is detected

Special constraints

- Global constraints: *AllDiff(. . .)*
- $\boxed{\text{QUESTION}}$ What kind of inferences can we draw from *AllDiff*?
  - If there are fewer possible values than variables, inconsistency is detected
  - $\boxed{\text{IDEA}}$
    - Remove variables with singleton domain *and* delete that value from the domains of the other variables
    - Repeat as long as there are variables with singleton domains. If an empty domain is produced, then there are more variables than distinct values in the domains

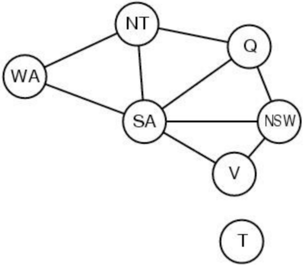
Backjumping

- Intelligent backtracking
  - Standard backtracking is **chronological backtracking**: change the latest assigned variable

Backjumping

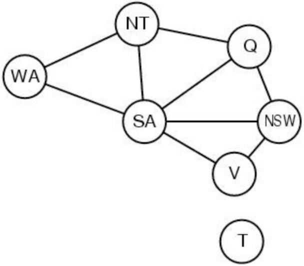
- Intelligent backtracking
  - Standard backtracking is **chronological backtracking**: change the latest assigned variable
  - More intelligent backtracking: select a variable in the **conflict set**
    - **Conflict set**: All previously assigned variables connected to the current variable by at least one constraint
    - **Backjumping**: Backtrack to the most recently assigned variable that conflicts with the current variable

Backjumping



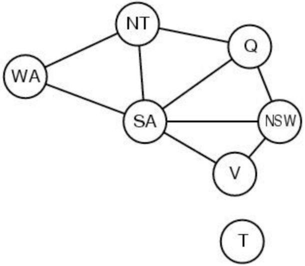
Assume assignment *{Q=red, NSW=green, V=blue, T=red }* and a fixed variable ordering (*Q, NSW, V, T, SA, WA, NT*).

Backjumping



Assume assignment *{Q=red, NSW=green, V=blue, T=red }* and a fixed variable ordering (*Q, NSW, V, T, SA, WA, NT*). And run the backtracking algorithm:

Backjumping



Assume assignment *{Q=red, NSW=green, V=blue, T=red }* and a fixed variable ordering (*Q, NSW, V, T, SA, WA, NT*). And run the backtracking algorithm:

- Failure is detected when trying to assign *SA*
- Backtracking to *T* is not useful, since changing the assignment of *T* will not solve an inconsistency
- Backjumping jumps to *V* instead

PART V
Local Search for CSP

A different approach: Local search for CSP

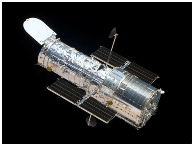
- We have seen that CSP can be expressed as a tree search problem
- Local search techniques using complete-state representations are applicable
  - states are complete (possibly illegal!) assignments
  - actions *reassign* values to variables

A different approach: Local search for CSP

- We have seen that CSP can be expressed as a tree search problem
- Local search techniques using complete-state representations are applicable
  - states are complete (possibly illegal!) assignments
  - actions *reassign* values to variables
- Variable selection: any variable involved in a violated constraint
- Value selection: **min-conflicts** heuristic
  - Select the value that minimizes the number of variables in violated constraints

A different approach: Local search for CSP

- We have seen that CSP can be expressed as a tree search problem
- Local search techniques using complete-state representations are applicable
  - states are complete (possibly illegal!) assignments
  - actions *reassign* values to variables
- Variable selection: any variable involved in a violated constraint
- Value selection: **min-conflicts** heuristic
  - Select the value that minimizes the number of variables in violated constraints

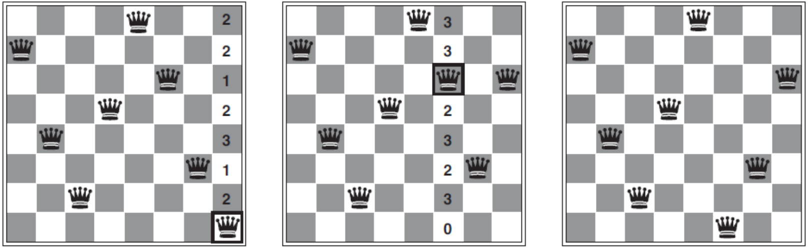


Local search for CSP: Min-Conflicts Algorithm

function MIN-CONFLICTS (*csp, maxSteps*) **returns** solution
  inputs: *csp*, a constraint satisfaction problem
  maxSteps, the number of steps before giving up

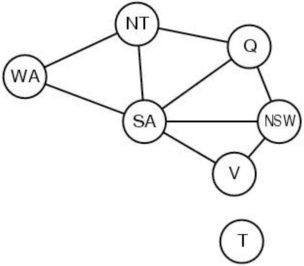
  current $\leftarrow$ an initial complete assignment for *csp*
  for $i = 1$ to *maxSteps* **do**
    if *current* solves *csp* **then return** *current*
    var $\leftarrow$ random conflicted variable from *csp*.VARIABLES
    val $\leftarrow$ minimizes CONFLICTS(*var, val, current, csp*)
    set *var* = *value* in *current*
  return *failure*

Local search for CSP: Example



- At each step, select a queen that violates a constraint
- Then move the queen to a spot that minimizes the number of conflicts

Problem structure



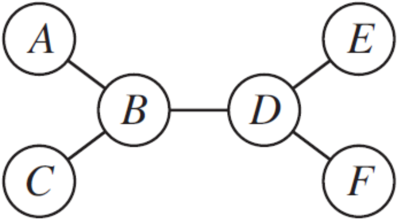
- The Australia map-coloring problem can be split into two independent subproblems
- Independent subproblems are represented as sets of constraints that use non-overlapping sets of variables
  - e.g. $X_1 + X_2 < 10, X_3 + X_4 > 20$

Problem structure

Splitting up the problem significantly improves performance

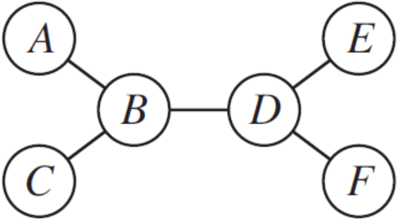
- Assume that some CSP consists of several smaller CSPs
  - Assume that each subproblem has c of the n variables (n/c subproblems)
- Worst-case performance is $O\left(\frac{n}{c} \cdot d^c\right)$
  - Suppose $n = 80, c = 20, d = 2$
  - Standard problem takes $2^{80} \approx 4$ billion years (1 million nodes/sec)
  - Split problem takes $4 \cdot 2^{20} \approx 0.4$ seconds (1 million nodes/sec)

Tree-structured CSP



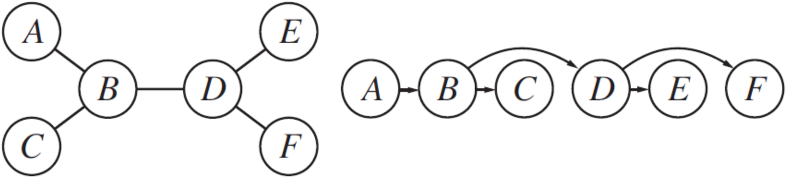
If the constraint graph is a tree (any two variables are therefore connected by at most one path), the associated CSP can be solved in $O(nd^2)$.

Tree-structured CSP



If the constraint graph is a tree (any two variables are therefore connected by at most one path), the associated CSP can be solved in $O(nd^2)$. While in the general case worst case performance is $O(d^n)$.

Tree-structured CSP: Intuitions



- Choose a variable as root, order variables from root to leaves
- Apply arc-consistency on parent-child pairs
- Assign variables of child nodes consistent with parent nodes

Summary

- Definition of CSP
- Search in CSP (Backtracking search algorithm)
- Local search in CSP (Min-Conflicts algorithm)