Constraint Satisfaction Problems

Dr. Steven Bethard

Computer and Information Sciences
University of Alabama at Birmingham

9 Feb 2016

Outline

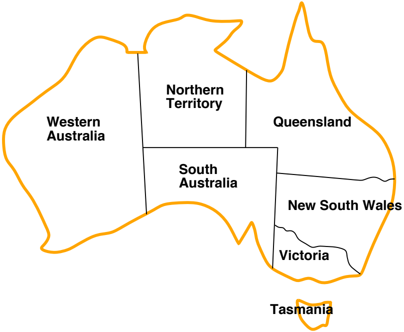
Outline

1. Constraint Satisfaction
   - Defining Problems
   - Problem Types
2. Backtracking Search
   - CSPs as Search
   - Search Heuristics
3. Search Alternatives
   - Local Search
   - Tree Search

Example: Map Coloring

Problem

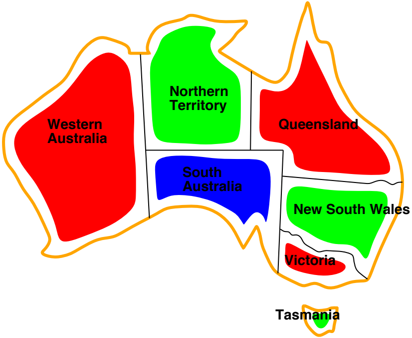
Color all countries with red, green or blue, and with no 2 adjacent countries the same color



Example: Map Coloring

Problem

Color all countries with red, green or blue, and with no 2 adjacent countries the same color



Constraint Satisfaction Problem

Terms

Variables $X_1, X_2, \ldots, X_n$

Domains Allowable values for each variable

Constraints Allowable combinations of variables

States

Assignment of values to variables, $\{X_i = v_i, X_j = v_j, \ldots\}$

Types of States

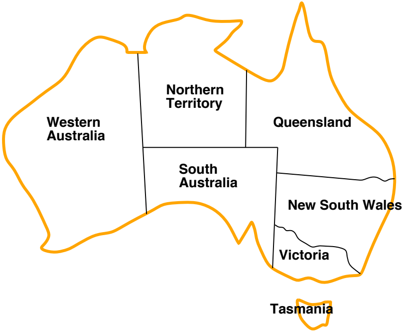
Consistent No constraints violated

Solution No constraints violated, all variables assigned

Example: Map Coloring

Problem

Color all countries with red, green or blue, and with no 2 adjacent countries the same color



Variables WA, NT, Q, NSW, V, SA, T

Domains $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$

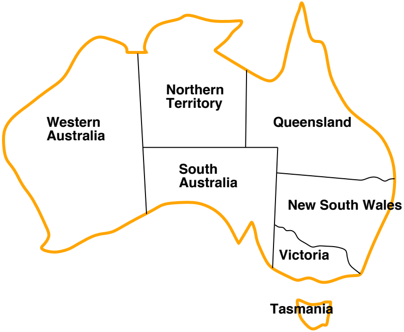
Constraints $\texttt{WA} \neq \texttt{NT}, \texttt{WA} \neq \texttt{SA}, \texttt{NT} \neq \texttt{SA}, \ldots$

Solution $\left\{ \begin{array}{l} \texttt{WA} = \texttt{red}, \texttt{NT} = \texttt{green}, \texttt{Q} = \texttt{red}, \\ \texttt{NSW} = \texttt{green}, \texttt{V} = \texttt{red}, \texttt{SA} = \texttt{blue}, \\ \texttt{T} = \texttt{green} \end{array} \right\}$

Example: Map Coloring

Problem

Color all countries with red, green or blue, and with no 2 adjacent countries the same color



Variables `WA, NT, Q, NSW, V, SA, T`

Domains $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$

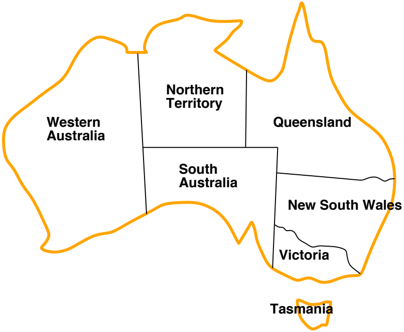
Constraints $\texttt{WA} \neq \texttt{NT}, \texttt{WA} \neq \texttt{SA}, \texttt{NT} \neq \texttt{SA}, \ldots$

Solution $\left\{ \begin{array}{l} \texttt{WA} = \texttt{red}, \texttt{NT} = \texttt{green}, \texttt{Q} = \texttt{red}, \\ \texttt{NSW} = \texttt{green}, \texttt{V} = \texttt{red}, \texttt{SA} = \texttt{blue}, \\ \texttt{T} = \texttt{green} \end{array} \right\}$

Example: Map Coloring

Problem

Color all countries with red, green or blue, and with no 2 adjacent countries the same color



Variables $\mathtt{WA}, \mathtt{NT}, \mathtt{Q}, \mathtt{NSW}, \mathtt{V}, \mathtt{SA}, \mathtt{T}$

Domains $D_i = \{\mathtt{red}, \mathtt{green}, \mathtt{blue}\}$

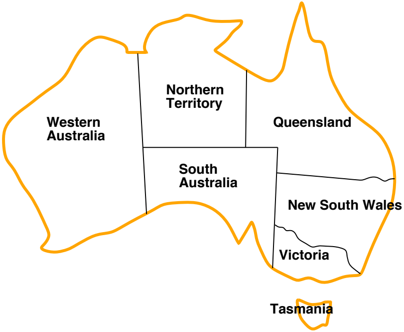
Constraints $\mathtt{WA} \neq \mathtt{NT}, \mathtt{WA} \neq \mathtt{SA}, \mathtt{NT} \neq \mathtt{SA}, \ldots$

Solution $\left\{ \begin{array}{l} \mathtt{WA} = \mathtt{red}, \mathtt{NT} = \mathtt{green}, \mathtt{Q} = \mathtt{red}, \\ \mathtt{NSW} = \mathtt{green}, \mathtt{V} = \mathtt{red}, \mathtt{SA} = \mathtt{blue}, \\ \mathtt{T} = \mathtt{green} \end{array} \right\}$

Example: Map Coloring

Problem

Color all countries with red, green or blue, and with no 2 adjacent countries the same color

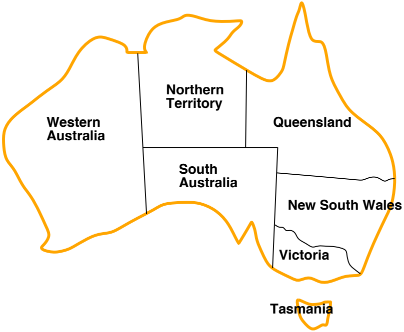


Variables	WA, NT, Q, NSW, V, SA, T
Domains	$D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$
Constraints	WA $\neq$ NT, WA $\neq$ SA, NT $\neq$ SA, ...
Solution	$\left\{ \begin{array}{l} \texttt{WA} = \texttt{red}, \texttt{NT} = \texttt{green}, \texttt{Q} = \texttt{red}, \\ \texttt{NSW} = \texttt{green}, \texttt{V} = \texttt{red}, \texttt{SA} = \texttt{blue}, \\ \texttt{T} = \texttt{green} \end{array} \right\}$

Example: Map Coloring

Problem

Color all countries with red, green or blue, and with no 2 adjacent countries the same color



Variables WA, NT, Q, NSW, V, SA, T

Domains $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$

Constraints $\texttt{WA} \neq \texttt{NT}, \texttt{WA} \neq \texttt{SA}, \texttt{NT} \neq \texttt{SA}, \ldots$

Solution
$$\left\{ \begin{array}{l} \texttt{WA} = \texttt{red}, \texttt{NT} = \texttt{green}, \texttt{Q} = \texttt{red}, \\ \texttt{NSW} = \texttt{green}, \texttt{V} = \texttt{red}, \texttt{SA} = \texttt{blue}, \\ \texttt{T} = \texttt{green} \end{array} \right\}$$

Example: Cryptarithmetic

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

Variables $T, W, O, F, U, R, C_1, C_2, C_3$

Domains $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints

$$
\begin{aligned}
O + O &= R + 10 \cdot C_1 \\
C_1 + W + W &= U + 10 \cdot C_2 \\
C_2 + T + T &= O + 10 \cdot C_3 \\
C_3 &= F
\end{aligned}
$$

Solution $\left\{ \begin{array}{l} T = 7, W = 3, O = 4, F = 1, U = 6, \\ R = 8, C_1 = 0, C_2 = 0, C_3 = 1 \end{array} \right\}$

Example: Cryptarithmetic

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

Variables $T, W, O, F, U, R, C_1, C_2, C_3$

Domains $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints

$$\begin{aligned} O + O &= R + 10 \cdot C_1 \\ C_1 + W + W &= U + 10 \cdot C_2 \\ C_2 + T + T &= O + 10 \cdot C_3 \\ C_3 &= F \end{aligned}$$

Solution $\left\{ \begin{array}{l} T = 7, W = 3, O = 4, F = 1, U = 6, \\ R = 8, C_1 = 0, C_2 = 0, C_3 = 1 \end{array} \right\}$

Example: Cryptarithmetic

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

Variables $T, W, O, F, U, R, C_1, C_2, C_3$

Domains $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints

$$\begin{aligned} O + O &= R + 10 \cdot C_1 \\ C_1 + W + W &= U + 10 \cdot C_2 \\ C_2 + T + T &= O + 10 \cdot C_3 \\ C_3 &= F \end{aligned}$$

Solution $\left\{ \begin{array}{l} T = 7, W = 3, O = 4, F = 1, U = 6, \\ R = 8, C_1 = 0, C_2 = 0, C_3 = 1 \end{array} \right\}$

Example: Cryptarithmetic

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

Variables $T, W, O, F, U, R, C_1, C_2, C_3$

Domains $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints

$$
\begin{aligned}
O + O &= R + 10 \cdot C_1 \\
C_1 + W + W &= U + 10 \cdot C_2 \\
C_2 + T + T &= O + 10 \cdot C_3 \\
C_3 &= F
\end{aligned}
$$

Solution $\left\{ \begin{array}{l} T = 7, W = 3, O = 4, F = 1, U = 6, \\ R = 8, C_1 = 0, C_2 = 0, C_3 = 1 \end{array} \right\}$

Example: Cryptarithmetic

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

Variables $T, W, O, F, U, R, C_1, C_2, C_3$

Domains $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints

$$\begin{aligned} O + O &= R + 10 \cdot C_1 \\ C_1 + W + W &= U + 10 \cdot C_2 \\ C_2 + T + T &= O + 10 \cdot C_3 \\ C_3 &= F \end{aligned}$$

Solution $\left\{ \begin{array}{l} T = 7, W = 3, O = 4, F = 1, U = 6, \\ R = 8, C_1 = 0, C_2 = 0, C_3 = 1 \end{array} \right\}$

Example: Cryptarithmetic

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

Variables $T, W, O, F, U, R, C_1, C_2, C_3$

Domains $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints

$$\begin{aligned} O + O &= R + 10 \cdot C_1 \\ C_1 + W + W &= U + 10 \cdot C_2 \\ C_2 + T + T &= O + 10 \cdot C_3 \\ C_3 &= F \end{aligned}$$

Solution $\left\{ \begin{array}{l} T = 7, W = 3, O = 4, F = 1, U = 6, \\ R = 8, C_1 = 0, C_2 = 0, C_3 = 1 \end{array} \right\}$

Example: Meeting Scheduling

Problem Description

- 2 hour meetings: Jim & Tammy
- 1 hour meetings: Jim & Martha, Martha & Tammy
- Meetings start on the hour 9:00am to 5:00pm
- Busy: Jim 12-2, Martha 11-1, Tammy 10-11 and 2-3

Example: Meeting Scheduling

Problem Description

- 2 hour meetings: Jim & Tammy
- 1 hour meetings: Jim & Martha, Martha & Tammy
- Meetings start on the hour 9:00am to 5:00pm
- Busy: Jim 12-2, Martha 11-1, Tammy 10-11 and 2-3

Variables $\mathtt{J\&T}$, $\mathtt{J\&M}$, $\mathtt{M\&T}$

Domains $D_{\mathtt{J\&T}} = \{\mathtt{9\text{-}11}, \mathtt{10\text{-}12}, \ldots\}$

$D_{\mathtt{J\&M}} = D_{\mathtt{M\&T}} = \{\mathtt{9\text{-}10}, \mathtt{10\text{-}11}, \ldots\}$

Constraints $\mathtt{J\&T} \cap \mathtt{J\&M} = \emptyset$, $\mathtt{J\&M} \cap \mathtt{M\&T} = \emptyset$,

$\mathtt{12\text{-}2} \cap \mathtt{J\&T} = \emptyset$, $\mathtt{12\text{-}2} \cap \mathtt{J\&M} = \emptyset, \ldots$

Solution $\{\mathtt{J\&T} = \mathtt{3\text{-}5}, \mathtt{M\&T} = \mathtt{1\text{-}2}, \mathtt{J\&M} = \mathtt{10\text{-}11}\}$

Domain Types

Finite Domains

- Examples: map coloring, 8-queens
- Constraints described by enumeration, e.g. $(\mathtt{WA}, \mathtt{NT}) \in \{(\mathtt{red}, \mathtt{green}), (\mathtt{red}, \mathtt{blue}), \ldots\}$

Infinite but Countable Domains

- Examples: scheduling jobs by day or hour
- Need constraint language, e.g. $\mathtt{Start}_{\mathtt{Job1}} + 5 \leq \mathtt{Start}_{\mathtt{Job3}}$

Continuous Domains

- Examples: scheduling jobs by any fraction of time
- Some can be solved by linear programming

Domain Types

Finite Domains

- Examples: map coloring, 8-queens
- Constraints described by enumeration, e.g. $(\mathtt{WA}, \mathtt{NT}) \in \{(\mathtt{red}, \mathtt{green}), (\mathtt{red}, \mathtt{blue}), \ldots\}$

Infinite but Countable Domains

- Examples: scheduling jobs by day or hour
- Need constraint language, e.g. $\mathtt{Start}_{\mathtt{Job1}} + 5 \leq \mathtt{Start}_{\mathtt{Job3}}$

Continuous Domains

- Examples: scheduling jobs by any fraction of time
- Some can be solved by linear programming

Domain Types

Finite Domains

- Examples: map coloring, 8-queens
- Constraints described by enumeration, e.g. $(\texttt{WA}, \texttt{NT}) \in \{(\texttt{red}, \texttt{green}), (\texttt{red}, \texttt{blue}), \ldots\}$

Infinite but Countable Domains

- Examples: scheduling jobs by day or hour
- Need constraint language, e.g. $\mathsf{Start}_{\mathsf{Job1}} + 5 \leq \mathsf{Start}_{\mathsf{Job3}}$

Continuous Domains

- Examples: scheduling jobs by any fraction of time
- Some can be solved by linear programming

Constraint Types

Unary Constraints

- Restrict the value of a single variable
- Can be eliminated by preprocessing domains

Binary Constraints

- Relate two variables, e.g. $\mathtt{SA} \neq \mathtt{NSW}$

- Relate more than two variables

Constraint Types

Unary Constraints

- Restrict the value of a single variable
- Can be eliminated by preprocessing domains

Binary Constraints

- Relate two variables, e.g. $\texttt{SA} \neq \texttt{NSW}$

Higher Order Constraints

- Relate more than two variables: can convert to binary

Preferences

- Cost of assignments, e.g. `red` is better than `green`

Constraint Types

Unary Constraints

- Restrict the value of a single variable
- Can be eliminated by preprocessing domains

Binary Constraints

- Relate two variables, e.g. `SA` $\neq$ `NSW`

Higher Order Constraints

- Relate more than two variables: can convert to binary

Preferences

- Cost of assignments, e.g. `red` is better than `green`

Constraint Types

Unary Constraints

- Restrict the value of a single variable
- Can be eliminated by preprocessing domains

Binary Constraints

- Relate two variables, e.g. $\mathtt{SA} \neq \mathtt{NSW}$

Higher Order Constraints

- Relate more than two variables: can convert to binary

Preferences

- Cost of assignments, e.g. `red` is better than `green`

Constraint Types

Unary Constraints

- Restrict the value of a single variable
- Can be eliminated by preprocessing domains

Binary Constraints

- Relate two variables, e.g. $\texttt{SA} \neq \texttt{NSW}$

Higher Order Constraints

- Relate more than two variables: can convert to binary

Preferences

- Cost of assignments, e.g. `red` is better than `green`

Constraint Types

Unary Constraints

- Restrict the value of a single variable
- Can be eliminated by preprocessing domains

Binary Constraints

- Relate two variables, e.g. $\texttt{SA} \neq \texttt{NSW}$

Higher Order Constraints

- Relate more than two variables: can convert to binary

Preferences

- Cost of assignments, e.g. `red` is better than `green`

Outline

1. Constraint Satisfaction
   - Defining Problems
   - Problem Types
2. Backtracking Search
   - CSPs as Search
   - Search Heuristics
3. Search Alternatives
   - Local Search
   - Tree Search

CSPs as Search

Formulation

States	Full or partial assignments
Initial	The empty assignment, {}
Actions	Assign value to variable, obeying constraints
Goal	Assignment is complete

Benefits

- Same for all CSPs
- All solutions at depth $n \Rightarrow$ depth-first search ok

CSPs as Search

Formulation

States	Full or partial assignments
Initial	The empty assignment, {}
Actions	Assign value to variable, obeying constraints
Goal	Assignment is complete

Benefits

- Same for all CSPs
- All solutions at depth n $\Rightarrow$ depth-first search ok

CSPs as Search

Formulation

States	Full or partial assignments
Initial	The empty assignment, {}
Actions	Assign value to variable, obeying constraints
Goal	Assignment is complete

Benefits

- Same for all CSPs
- All solutions at depth $n \Rightarrow$ depth-first search ok

CSPs as Search

Problem

Given n variables, d values in domains:

	Root	Level 1	Level 2	...
Branches	nd	$(n-1)d$	$(n-2)d$	...

Leaves: $n! \cdot d^n$ Total possible assignments: d^n

Note

Variable assignments are commutative, e.g.

- `WA = red` then `NT = green`
- `NT = green` then `WA = red`

Solution: Consider only one variable at each node

CSPs as Search

Problem

Given n variables, d values in domains:

	Root	Level 1	Level 2	…
Branches	nd	$(n-1)d$	$(n-2)d$	…

Leaves: $n! \cdot d^n$ Total possible assignments: d^n

Note

Variable assignments are commutative, e.g.

- `WA = red` then `NT = green`
- `NT = green` then `WA = red`

Solution: Consider only one variable at each node

CSPs as Search

Problem

Given n variables, d values in domains:

	Root	Level 1	Level 2	...
Branches	nd	$(n-1)d$	$(n-2)d$	...

Leaves: $n! \cdot d^n$ Total possible assignments: d^n

Note

Variable assignments are commutative, e.g.

- `WA = red` then `NT = green`
- `NT = green` then `WA = red`

Solution: Consider only one variable at each node

CSPs as Search

Problem

Given n variables, d values in domains:

	Root	Level 1	Level 2	...
Branches	nd	$(n-1)d$	$(n-2)d$	...

Leaves: $n! \cdot d^n$ Total possible assignments: d^n

Note

Variable assignments are commutative, e.g.

- `WA = red` then `NT = green`
- `NT = green` then `WA = red`

Solution: Consider only one variable at each node

CSPs as Search

Problem

Given n variables, d values in domains:

	Root	Level 1	Level 2	...
Branches	nd	$(n-1)d$	$(n-2)d$	...

Leaves: $n! \cdot d^n$ Total possible assignments: d^n

Note

Variable assignments are commutative, e.g.

- `WA = red` then `NT = green`
- `NT = green` then `WA = red`

Solution: Consider only one variable at each node

CSPs as Search

Problem

Given n variables, d values in domains:

	Root	Level 1	Level 2	...
Branches	nd	$(n-1)d$	$(n-2)d$	...

Leaves: $n! \cdot d^n$ Total possible assignments: d^n

Note

Variable assignments are commutative, e.g.

- `WA = red` then `NT = green`
- `NT = green` then `WA = red`

Solution: Consider only one variable at each node

CSPs as Search

Problem

Given n variables, d values in domains:

	Root	Level 1	Level 2	...
Branches	nd	$(n-1)d$	$(n-2)d$	...

Leaves: $n! \cdot d^n$ Total possible assignments: d^n

Note

Variable assignments are commutative, e.g.

- WA = red then NT = green
- NT = green then WA = red

Solution: Consider only one variable at each node

CSPs as Search

Problem

Given n variables, d values in domains:

	Root	Level 1	Level 2	...
Branches	nd	$(n-1)d$	$(n-2)d$	...

Leaves: $n! \cdot d^n$ Total possible assignments: d^n

Note

Variable assignments are commutative, e.g.

- `WA = red` then `NT = green`
- `NT = green` then `WA = red`

Solution: Consider only one variable at each node

CSPs as Search

Problem

Given n variables, d values in domains:

	Root	Level 1	Level 2	...
Branches	nd	$(n-1)d$	$(n-2)d$	...

Leaves: $n! \cdot d^n$ Total possible assignments: d^n

Note

Variable assignments are commutative, e.g.

- `WA` = `red` then `NT` = `green`
- `NT` = `green` then `WA` = `red`

Solution: Consider only one variable at each node

CSPs as Search

Problem

Given n variables, d values in domains:

	Root	Level 1	Level 2	...
Branches	nd	$(n-1)d$	$(n-2)d$	...

Leaves: $n! \cdot d^n$ Total possible assignments: d^n

Note

Variable assignments are commutative, e.g.

- `WA` = `red` then `NT` = `green`
- `NT` = `green` then `WA` = `red`

Solution: Consider only one variable at each node

Backtracking Search Code

```python
def csp_search(csp, heuristic, assignment=None):
    if assignment is None:
        assignment = {}
    # if assignment is complete, return it
    if len(assignment) == len(csp.variables):
        return assignment
    # select an unassigned variable and order the values
    variable = heuristic.select_variable(csp, assignment)
    # try assigning each value to the variable
    for value in heuristic.order_values(csp, assignment, variable):
        assignment[variable] = value
        # for consistent assignments, recursively check if
        # it is possible to assign the remaining variables
        if csp.is_consistent(assignment):
            result = csp_search(csp, heuristic, assignment)
            if result is not None:
                return result
        del assignment[variable]
    # all assignments failed
    return None
```

Backtracking Search Code

```python
def csp_search(csp, heuristic, assignment=None):
    if assignment is None:
        assignment = {}
    # if assignment is complete, return it
    if len(assignment) == len(csp.variables):
        return assignment
    # select an unassigned variable and order the values
    variable = heuristic.select_variable(csp, assignment)
    # try assigning each value to the variable
    for value in heuristic.order_values(csp, assignment, variable):
        assignment[variable] = value
        # for consistent assignments, recursively check if
        # it is possible to assign the remaining variables
        if csp.is_consistent(assignment):
            result = csp_search(csp, heuristic, assignment)
            if result is not None:
                return result
        del assignment[variable]
    # all assignments failed
    return None
```

Backtracking Search Code

```python
def csp_search(csp, heuristic, assignment=None):
    if assignment is None:
        assignment = {}
    # if assignment is complete, return it
    if len(assignment) == len(csp.variables):
        return assignment
    # select an unassigned variable and order the values
    variable = heuristic.select_variable(csp, assignment)
    # try assigning each value to the variable
    for value in heuristic.order_values(csp, assignment, variable):
        assignment[variable] = value
        # for consistent assignments, recursively check if
        # it is possible to assign the remaining variables
        if csp.is_consistent(assignment):
            result = csp_search(csp, heuristic, assignment)
            if result is not None:
                return result
        del assignment[variable]
    # all assignments failed
    return None
```

Backtracking Search Code

```
def csp_search(csp, heuristic, assignment=None):
    if assignment is None:
        assignment = {}
    # if assignment is complete, return it
    if len(assignment) == len(csp.variables):
        return assignment
    # select an unassigned variable and order the values
    variable = heuristic.select_variable(csp, assignment)
    # try assigning each value to the variable
    for value in heuristic.order_values(csp, assignment, variable):
        assignment[variable] = value
        # for consistent assignments, recursively check if
        # it is possible to assign the remaining variables
        if csp.is_consistent(assignment):
            result = csp_search(csp, heuristic, assignment)
            if result is not None:
                return result
        del assignment[variable]
    # all assignments failed
    return None
```

Backtracking Search Code

```
def csp_search(csp, heuristic, assignment=None):
    if assignment is None:
        assignment = {}
    # if assignment is complete, return it
    if len(assignment) == len(csp.variables):
        return assignment
    # select an unassigned variable and order the values
    variable = heuristic.select_variable(csp, assignment)
    # try assigning each value to the variable
    for value in heuristic.order_values(csp, assignment, variable):
        assignment[variable] = value
        # for consistent assignments, recursively check if
        # it is possible to assign the remaining variables
        if csp.is_consistent(assignment):
            result = csp_search(csp, heuristic, assignment)
            if result is not None:
                return result
        del assignment[variable]
    # all assignments failed
    return None
```

Backtracking Search Code

```
def csp_search(csp, heuristic, assignment=None):
    if assignment is None:
        assignment = {}
    # if assignment is complete, return it
    if len(assignment) == len(csp.variables):
        return assignment
    # select an unassigned variable and order the values
    variable = heuristic.select_variable(csp, assignment)
    # try assigning each value to the variable
    for value in heuristic.order_values(csp, assignment, variable):
        assignment[variable] = value
        # for consistent assignments, recursively check if
        # it is possible to assign the remaining variables
        if csp.is_consistent(assignment):
            result = csp_search(csp, heuristic, assignment)
            if result is not None:
                return result
        del assignment[variable]
    # all assignments failed
    return None
```

Backtracking Search Code

```python
def csp_search(csp, heuristic, assignment=None):
    if assignment is None:
        assignment = {}
    # if assignment is complete, return it
    if len(assignment) == len(csp.variables):
        return assignment
    # select an unassigned variable and order the values
    variable = heuristic.select_variable(csp, assignment)
    # try assigning each value to the variable
    for value in heuristic.order_values(csp, assignment, variable):
        assignment[variable] = value
        # for consistent assignments, recursively check if
        # it is possible to assign the remaining variables
        if csp.is_consistent(assignment):
            result = csp_search(csp, heuristic, assignment)
            if result is not None:
                return result
        del assignment[variable]
    # all assignments failed
    return None
```

Backtracking Example

Variables WA, NT, Q, SA, NSW, V, T

Domains $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$

Constraints $\texttt{SA} \neq \texttt{WA}$, $\texttt{SA} \neq \texttt{NT}$, $\texttt{SA} \neq \texttt{Q}$, $\texttt{SA} \neq \texttt{NSW}$, $\texttt{SA} \neq \texttt{V}$, $\texttt{WA} \neq \texttt{NT}$, $\texttt{NT} \neq \texttt{Q}$, $\texttt{Q} \neq \texttt{NSW}$, $\texttt{NSW} \neq \texttt{V}$

Backtracking Example

Variables	WA, NT, Q, SA, NSW, V, T
Domains	$D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$
Constraints	$\texttt{SA} \neq \texttt{WA}$, $\texttt{SA} \neq \texttt{NT}$, $\texttt{SA} \neq \texttt{Q}$, $\texttt{SA} \neq \texttt{NSW}$, $\texttt{SA} \neq \texttt{V}$, $\texttt{WA} \neq \texttt{NT}$, $\texttt{NT} \neq \texttt{Q}$, $\texttt{Q} \neq \texttt{NSW}$, $\texttt{NSW} \neq \texttt{V}$

$\{\}$

Backtracking Example

Variables WA, NT, Q, SA, NSW, V, T

Domains $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$

Constraints SA $\neq$ WA, SA $\neq$ NT, SA $\neq$ Q, SA $\neq$ NSW, SA $\neq$ V,
WA $\neq$ NT, NT $\neq$ Q, Q $\neq$ NSW, NSW $\neq$ V

{}

{WA=red}

Backtracking Example

Variables WA, NT, Q, SA, NSW, V, T

Domains $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$

Constraints $\texttt{SA} \neq \texttt{WA}$, $\texttt{SA} \neq \texttt{NT}$, $\texttt{SA} \neq \texttt{Q}$, $\texttt{SA} \neq \texttt{NSW}$, $\texttt{SA} \neq \texttt{V}$, $\texttt{WA} \neq \texttt{NT}$, $\texttt{NT} \neq \texttt{Q}$, $\texttt{Q} \neq \texttt{NSW}$, $\texttt{NSW} \neq \texttt{V}$

{}

{WA=red}

{ WA = red, NT = green}

Backtracking Example

Variables WA, NT, Q, SA, NSW, V, T

Domains $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$

Constraints SA $\neq$ WA, SA $\neq$ NT, SA $\neq$ Q, SA $\neq$ NSW, SA $\neq$ V,
WA $\neq$ NT, NT $\neq$ Q, Q $\neq$ NSW, NSW $\neq$ V

{}

{WA=red}

{ WA = red, NT = green}

{ WA = red, NT = green, Q = blue}

Backtracking Example

Variables WA, NT, Q, SA, NSW, V, T

Domains $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$

Constraints $\texttt{SA} \neq \texttt{WA}$, $\texttt{SA} \neq \texttt{NT}$, $\texttt{SA} \neq \texttt{Q}$, $\texttt{SA} \neq \texttt{NSW}$, $\texttt{SA} \neq \texttt{V}$, $\texttt{WA} \neq \texttt{NT}$, $\texttt{NT} \neq \texttt{Q}$, $\texttt{Q} \neq \texttt{NSW}$, $\texttt{NSW} \neq \texttt{V}$

{}

{WA=red}

{ WA = red, NT = green}

{ WA = red, NT = green, Q = blue}

{ WA = red, NT = green, Q = blue, SA = ???}

Backtracking Example

Variables WA, NT, Q, SA, NSW, V, T

Domains $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$

Constraints SA $\neq$ WA, SA $\neq$ NT, SA $\neq$ Q, SA $\neq$ NSW, SA $\neq$ V,
WA $\neq$ NT, NT $\neq$ Q, Q $\neq$ NSW, NSW $\neq$ V

{}

{WA=red}

{ WA = red, NT = green}

{ WA = red, NT = green, Q = red}

Backtracking Example

Variables WA, NT, Q, SA, NSW, V, T

Domains $D_i = \{\texttt{red}, \texttt{green}, \texttt{blue}\}$

Constraints SA ≠ WA, SA ≠ NT, SA ≠ Q, SA ≠ NSW, SA ≠ V, WA ≠ NT, NT ≠ Q, Q ≠ NSW, NSW ≠ V

{}

{WA=red}

{ WA = red, NT = green}

{ WA = red, NT = green, Q = red}

...

Backtracking Heuristics

Problem

- Basic depth first search is still too inefficient.
- E.g. Can only solve n-queens for $n \approx 25$

How can we be smarter?

- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?

Backtracking Heuristics

Problem

- Basic depth first search is still too inefficient.
- E.g. Can only solve n-queens for $n \approx 25$

How can we be smarter?

- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?

Backtracking Heuristics

Problem

- Basic depth first search is still too inefficient.
- E.g. Can only solve n-queens for $n \approx 25$

How can we be smarter?

- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?

Backtracking Heuristics

Problem

- Basic depth first search is still too inefficient.
- E.g. Can only solve n-queens for $n \approx 25$

How can we be smarter?

- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?

Minimum Remaining Values

Idea

Select the variable with the fewest legal values

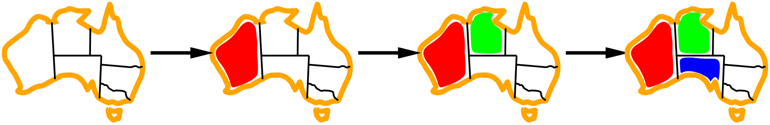
Also Known As

Most Constrained Variable

Minimum Remaining Values

Idea

Select the variable with the fewest legal values



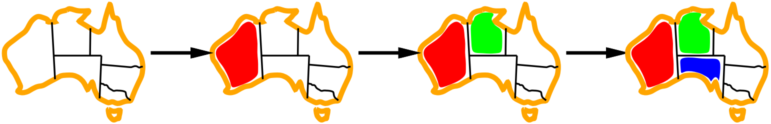
Also Known As

Most Constrained Variable

Minimum Remaining Values

Idea

Select the variable with the fewest legal values



Also Known As

Most Constrained Variable

Degree Heuristic

Idea

- But what to do when MRV produces ties?
- Select variable with most constraints on other values

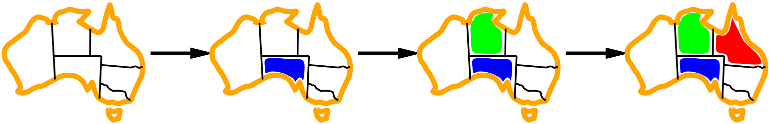
Also Known As

Most Constraining Variable

Degree Heuristic

Idea

- But what to do when MRV produces ties?
- Select variable with most constraints on other values



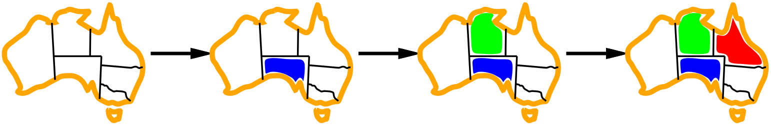
Also Known As

Most Constraining Variable

Degree Heuristic

Idea

- But what to do when MRV produces ties?
- Select variable with most constraints on other values



Also Known As

Most Constraining Variable

Least Constraining Value

Idea

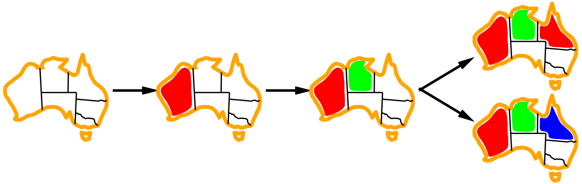
Select the variable that rules out the smallest number of values for the remaining variables

	Minimum Remaining Values
+	Degree Heuristic
+	Least Constraining Value
≈	1000 queens

Least Constraining Value

Idea

Select the variable that rules out the smallest number of values for the remaining variables



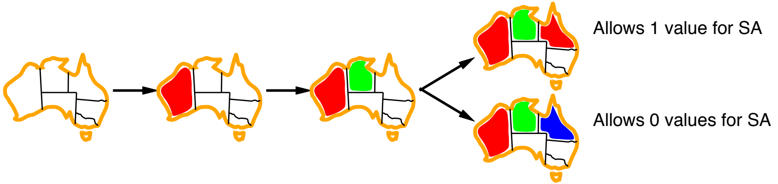
Minimum Remaining Values
\+ Degree Heuristic
\+ Least Constraining Value

≈ 1000 queens

Least Constraining Value

Idea

Select the variable that rules out the smallest number of values for the remaining variables

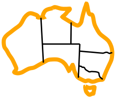


	Minimum Remaining Values
+	Degree Heuristic
+	Least Constraining Value
≈	1000 queens

Forward Checking

Idea

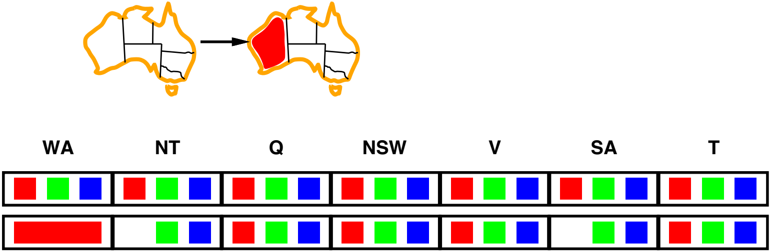
- Keep track of remaining legal values for all variables
- Stop search when any variable has no legal values



Forward Checking

Idea

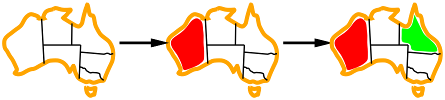
- Keep track of remaining legal values for all variables
- Stop search when any variable has no legal values



Forward Checking

Idea

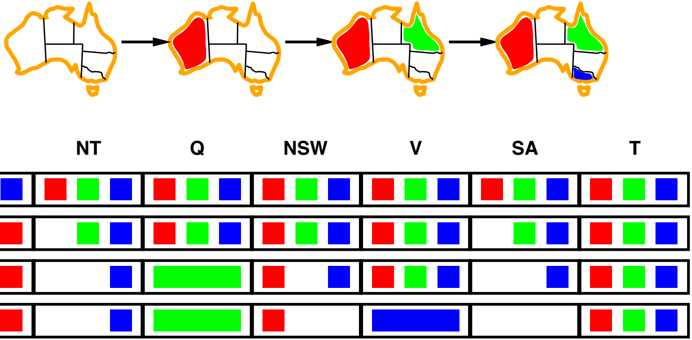
- Keep track of remaining legal values for all variables
- Stop search when any variable has no legal values



Forward Checking

Idea

- Keep track of remaining legal values for all variables
- Stop search when any variable has no legal values



Outline

1. Constraint Satisfaction
   - Defining Problems
   - Problem Types
2. Backtracking Search
   - CSPs as Search
   - Search Heuristics
3. Search Alternatives
   - Local Search
   - Tree Search

CSPs as Local Search

Formulation

States	Complete assignments
Initial	Any complete assignment
Actions	Change value of one variable
Goal	Consistent assignment

Benefits

- Minimal memory consumption
- Emprically very effective

CSPs as Local Search

Formulation

States Complete assignments

Initial Any complete assignment

Actions Change value of one variable

Goal Consistent assignment

Benefits

- Minimal memory consumption
- Emprically very effective

Min-Conflicts

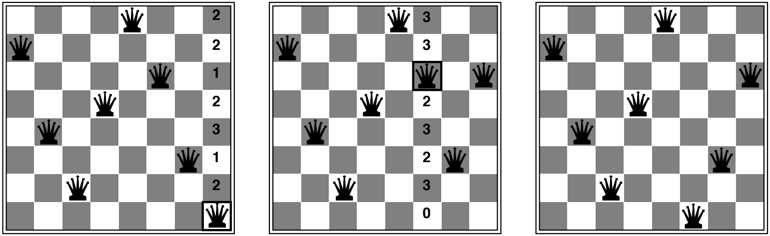
Idea

- Pick a variable with constraint violations
- Assign the value that violates the fewest constraints

Min-Conflicts

Idea

- Pick a variable with constraint violations
- Assign the value that violates the fewest constraints



Min-Conflicts Code

```python
def min_conflicts(csp, max_steps):
    # start with an initial complete assignment
    assignment = {}
    for variable in csp.variables:
        assignment[variable] = random.choice(variable.domain)
    # adjust one variable each time through the loop
    for i in range(max_steps):
        # return the assignment when it is consistent
        if csp.is_consistent(assignment):
            return assignment
        # otherwise, select a random conflicted variable
        var = random.choice(csp.get_conflicts(assignment))
        # assign the variable the value with minimal conflicts
        counts = {}
        for value in var.domain:
            assignment[var] = value
            counts[value] = len(csp.get_conflicts(assignment))
        assignment[var] = min(counts, key=counts.get)
    # all assignments failed
    return None
```

Min-Conflicts Code

```
def min_conflicts(csp, max_steps):
    # start with an initial complete assignment
    assignment = {}
    for variable in csp.variables:
        assignment[variable] = random.choice(variable.domain)
    # adjust one variable each time through the loop
    for i in range(max_steps):
        # return the assignment when it is consistent
        if csp.is_consistent(assignment):
            return assignment
        # otherwise, select a random conflicted variable
        var = random.choice(csp.get_conflicts(assignment))
        # assign the variable the value with minimal conflicts
        counts = {}
        for value in var.domain:
            assignment[var] = value
            counts[value] = len(csp.get_conflicts(assignment))
        assignment[var] = min(counts, key=counts.get)
    # all assignments failed
    return None
```

Min-Conflicts Code

```
def min_conflicts(csp, max_steps):
    # start with an initial complete assignment
    assignment = {}
    for variable in csp.variables:
        assignment[variable] = random.choice(variable.domain)
    # adjust one variable each time through the loop
    for i in range(max_steps):
        # return the assignment when it is consistent
        if csp.is_consistent(assignment):
            return assignment
        # otherwise, select a random conflicted variable
        var = random.choice(csp.get_conflicts(assignment))
        # assign the variable the value with minimal conflicts
        counts = {}
        for value in var.domain:
            assignment[var] = value
            counts[value] = len(csp.get_conflicts(assignment))
        assignment[var] = min(counts, key=counts.get)
    # all assignments failed
    return None
```

Min-Conflicts Code

```
def min_conflicts(csp, max_steps):
    # start with an initial complete assignment
    assignment = {}
    for variable in csp.variables:
        assignment[variable] = random.choice(variable.domain)
    # adjust one variable each time through the loop
    for i in range(max_steps):
        # return the assignment when it is consistent
        if csp.is_consistent(assignment):
            return assignment
        # otherwise, select a random conflicted variable
        var = random.choice(csp.get_conflicts(assignment))
        # assign the variable the value with minimal conflicts
        counts = {}
        for value in var.domain:
            assignment[var] = value
            counts[value] = len(csp.get_conflicts(assignment))
        assignment[var] = min(counts, key=counts.get)
    # all assignments failed
    return None
```

Min-Conflicts Code

```python
def min_conflicts(csp, max_steps):
    # start with an initial complete assignment
    assignment = {}
    for variable in csp.variables:
        assignment[variable] = random.choice(variable.domain)
    # adjust one variable each time through the loop
    for i in range(max_steps):
        # return the assignment when it is consistent
        if csp.is_consistent(assignment):
            return assignment
        # otherwise, select a random conflicted variable
        var = random.choice(csp.get_conflicts(assignment))
        # assign the variable the value with minimal conflicts
        counts = {}
        for value in var.domain:
            assignment[var] = value
            counts[value] = len(csp.get_conflicts(assignment))
        assignment[var] = min(counts, key=counts.get)
    # all assignments failed
    return None
```

Min-Conflicts Code

```
def min_conflicts(csp, max_steps):
    # start with an initial complete assignment
    assignment = {}
    for variable in csp.variables:
        assignment[variable] = random.choice(variable.domain)
    # adjust one variable each time through the loop
    for i in range(max_steps):
        # return the assignment when it is consistent
        if csp.is_consistent(assignment):
            return assignment
        # otherwise, select a random conflicted variable
        var = random.choice(csp.get_conflicts(assignment))
        # assign the variable the value with minimal conflicts
        counts = {}
        for value in var.domain:
            assignment[var] = value
            counts[value] = len(csp.get_conflicts(assignment))
        assignment[var] = min(counts, key=counts.get)
    # all assignments failed
    return None
```

Min-Conflicts Code

```python
def min_conflicts(csp, max_steps):
    # start with an initial complete assignment
    assignment = {}
    for variable in csp.variables:
        assignment[variable] = random.choice(variable.domain)
    # adjust one variable each time through the loop
    for i in range(max_steps):
        # return the assignment when it is consistent
        if csp.is_consistent(assignment):
            return assignment
        # otherwise, select a random conflicted variable
        var = random.choice(csp.get_conflicts(assignment))
        # assign the variable the value with minimal conflicts
        counts = {}
        for value in var.domain:
            assignment[var] = value
            counts[value] = len(csp.get_conflicts(assignment))
        assignment[var] = min(counts, key=counts.get)
    # all assignments failed
    return None
```

Min-Conflicts Properties

n-Queens

Almost constant time for arbitrary n with high probability

Other kinds of CSPs

Appears the same is true except for a narrow range of:

$$R = \frac{|\text{constraints}|}{|\text{variables}|}$$

Min-Conflicts Properties

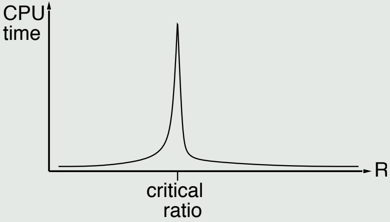
n-Queens

Almost constant time for arbitrary n with high probability

Other kinds of CSPs

Appears the same is true except for a narrow range of:

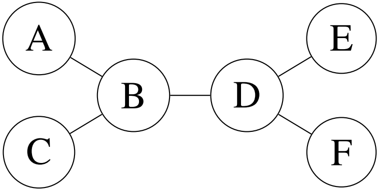
$$R = \frac{|\text{constraints}|}{|\text{variables}|}$$



Tree-Structured CSPs

CSPs as graphs:

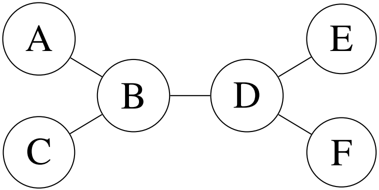
- Nodes = Variables
- Edges = Constraints



Tree-Structured CSPs

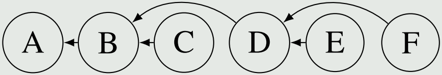
CSPs as graphs:

- Nodes = Variables
- Edges = Constraints



Solver

1. Choose root variable, list parents before children, e.g.

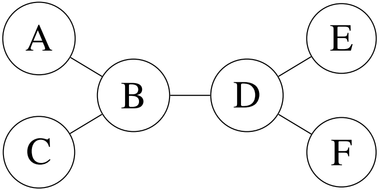


2. End to start: remove values inconsistent with parent
3. Start to end: assign any remaining consistent value

Tree-Structured CSPs

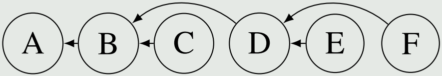
CSPs as graphs:

- Nodes = Variables
- Edges = Constraints



Solver

1. Choose root variable, list parents before children, e.g.



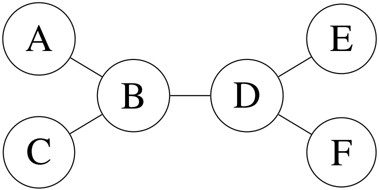
2. End to start: remove values inconsistent with parent
3. Start to end: assign any remaining consistent value

Time complexity:

Tree-Structured CSPs

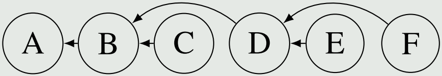
CSPs as graphs:

- Nodes = Variables
- Edges = Constraints



Solver

1. Choose root variable, list parents before children, e.g.



2. End to start: remove values inconsistent with parent
3. Start to end: assign any remaining consistent value

Time complexity: $O(nd^2)$

Tree-Structured CSP Example

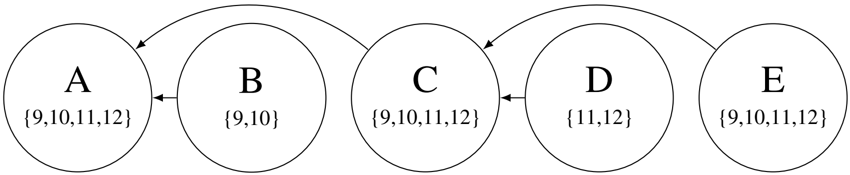
Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C

Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

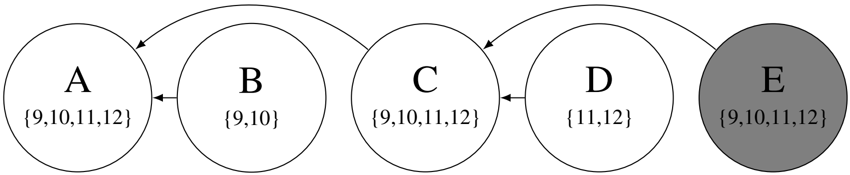
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

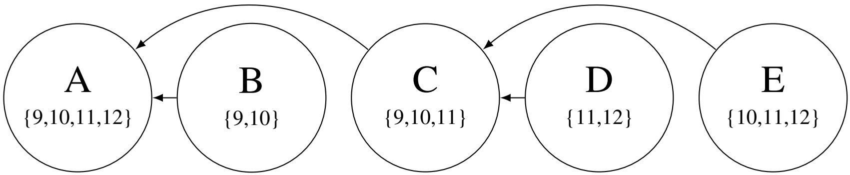
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

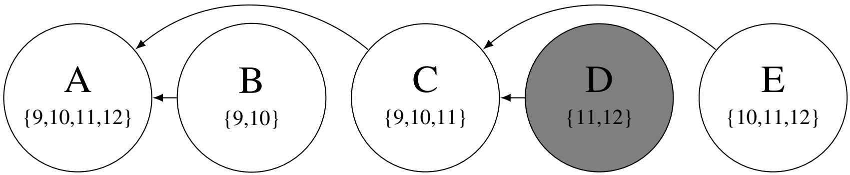
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

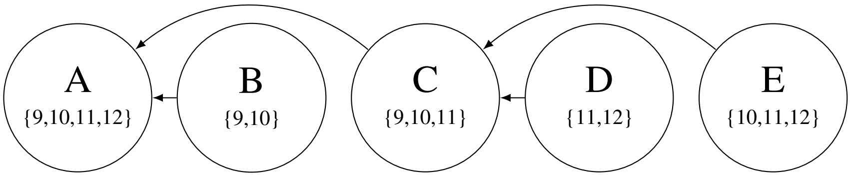
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

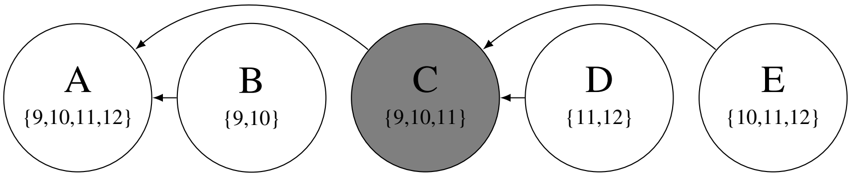
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

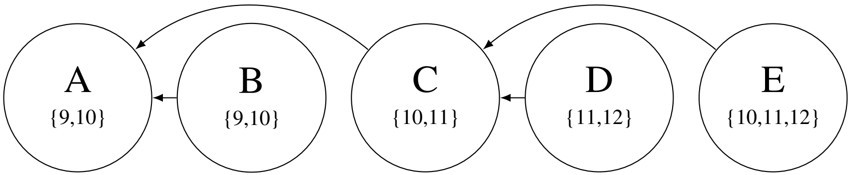
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

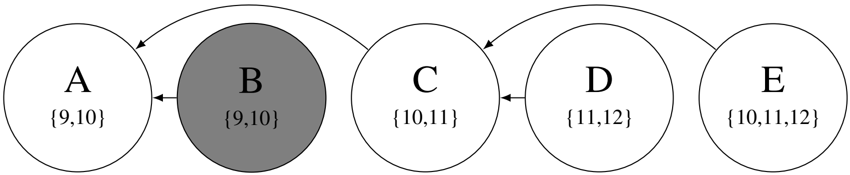
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

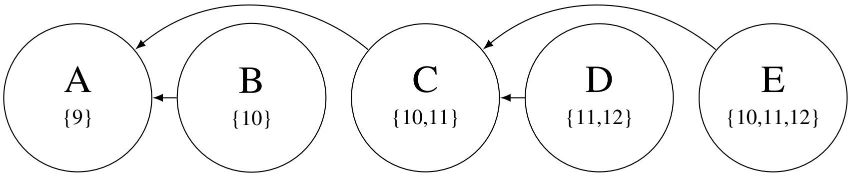
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

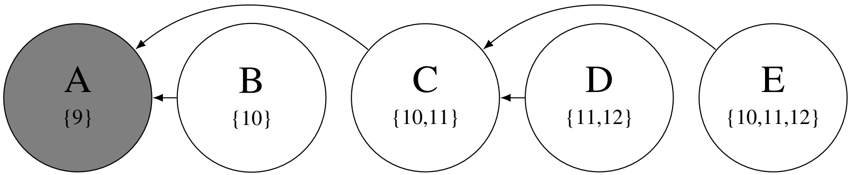
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

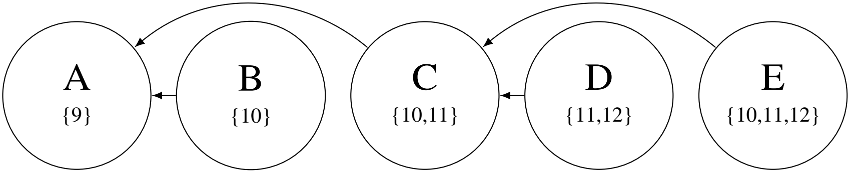
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

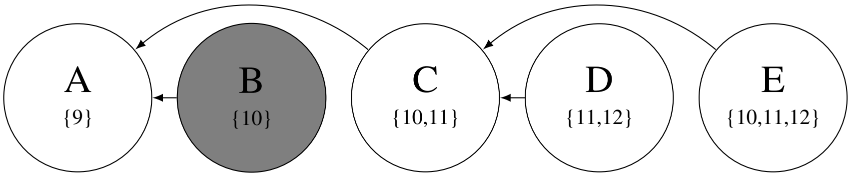
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

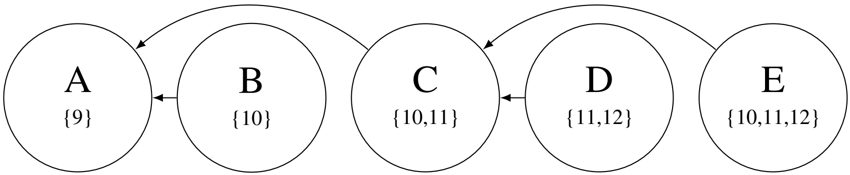
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

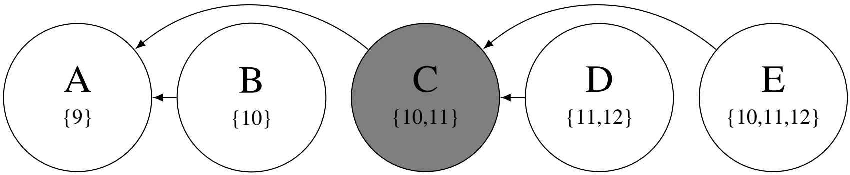
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

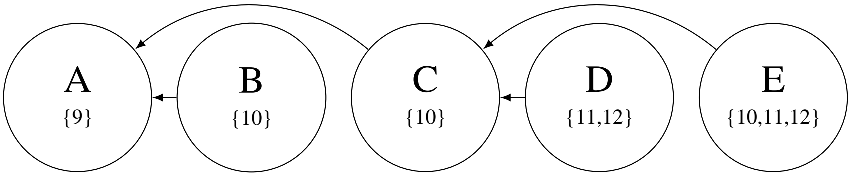
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

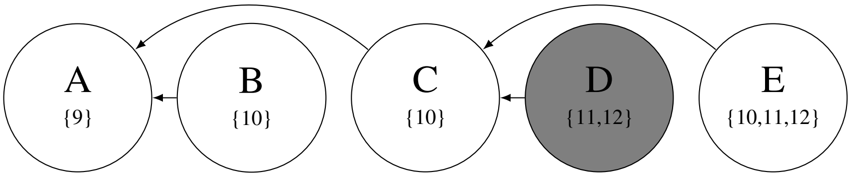
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

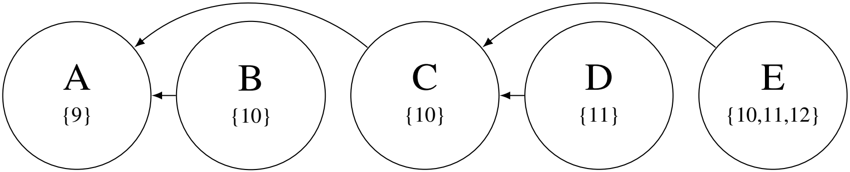
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

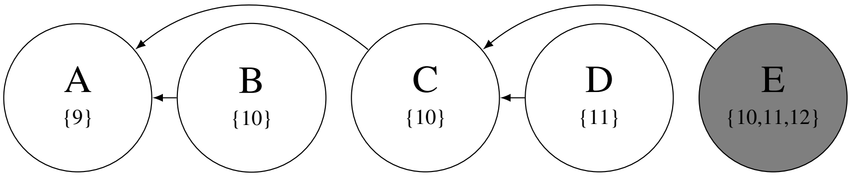
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

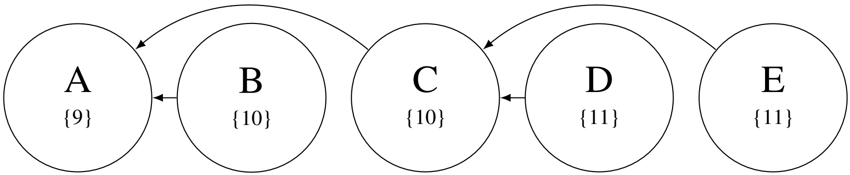
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Tree-Structured CSP Example

Schedule jobs A, B, C, D, E for 9, 10, 11 or 12

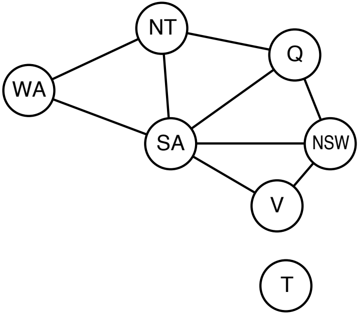
- B must be before 11:00
- D must be after 10:00
- B & C must be after A
- D & E must be after C



Subproblems and Decomposition

Example

Tasmania, mainland: separate components



If each subproblem has c of the n total variables

- Num. of subproblems: n/c
- Time per subproblem: d^c
- Total work: $O(nd^c/c)$

$n = 80, d = 2, c = 20$

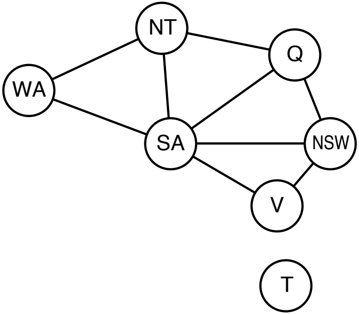
At 10 million nodes/sec:

- $2^{80} = 4$ billion years
- $4 \cdot 2^{20} = 0.4$ seconds

Subproblems and Decomposition

Example

Tasmania, mainland: separate components



If each subproblem has c of the n total variables

- Num. of subproblems: n/c
- Time per subproblem: d^c
- Total work: $O(nd^c/c)$

$n = 80, d = 2, c = 20$

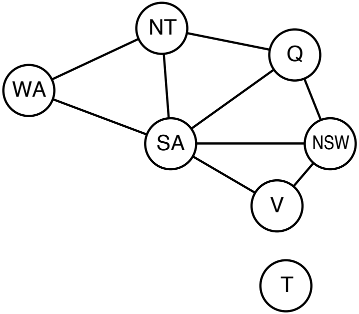
At 10 million nodes/sec:

- $2^{80} = 4$ billion years
- $4 \cdot 2^{20} = 0.4$ seconds

Subproblems and Decomposition

Example

Tasmania, mainland: separate components



If each subproblem has c of the n total variables

- Num. of subproblems: n/c
- Time per subproblem: d^c
- Total work: $O(nd^c/c)$

$n = 80, d = 2, c = 20$

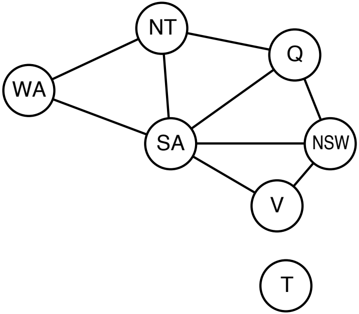
At 10 million nodes/sec:

- 2^{80} = 4 billion years
- $4 \cdot 2^{20}$ = 0.4 seconds

Subproblems and Decomposition

Example

Tasmania, mainland: separate components



If each subproblem has c of the n total variables

- Num. of subproblems: n/c
- Time per subproblem: d^c
- Total work: $O(nd^c/c)$

$n = 80, d = 2, c = 20$

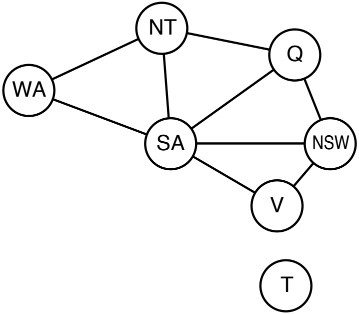
At 10 million nodes/sec:

- 2^{80} = 4 billion years
- $4 \cdot 2^{20}$ = 0.4 seconds

Subproblems and Decomposition

Example

Tasmania, mainland: separate components



If each subproblem has c of the n total variables

- Num. of subproblems: n/c
- Time per subproblem: d^c
- Total work: $O(nd^c/c)$

$n = 80, d = 2, c = 20$

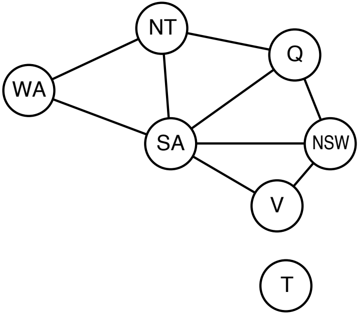
At 10 million nodes/sec:

- 2^{80} = 4 billion years
- $4 \cdot 2^{20}$ = 0.4 seconds

Subproblems and Decomposition

Example

Tasmania, mainland: separate components



If each subproblem has c of the n total variables

- Num. of subproblems: n/c
- Time per subproblem: d^c
- Total work: $O(nd^c/c)$

$n = 80, d = 2, c = 20$

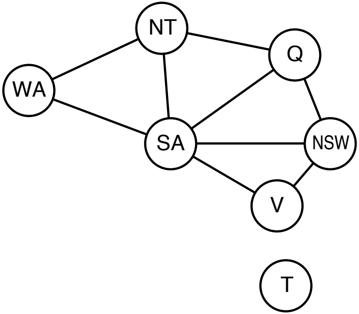
At 10 million nodes/sec:

- 2^{80} = 4 billion years
- $4 \cdot 2^{20}$ = 0.4 seconds

Subproblems and Decomposition

Example

Tasmania, mainland: separate components



If each subproblem has c of the n total variables

- Num. of subproblems: n/c
- Time per subproblem: d^c
- Total work: $O(nd^c/c)$

$n = 80, d = 2, c = 20$

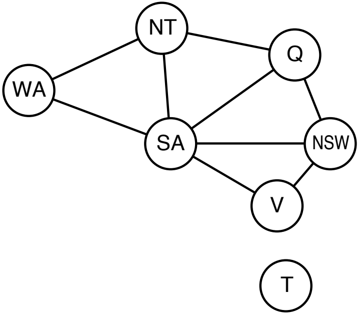
At 10 million nodes/sec:

- $2^{80} = 4$ billion years
- $4 \cdot 2^{20} = 0.4$ seconds

Subproblems and Decomposition

Example

Tasmania, mainland: separate components



If each subproblem has c of the n total variables

- Num. of subproblems: n/c
- Time per subproblem: d^c
- Total work: $O(nd^c/c)$

$n = 80, d = 2, c = 20$

At 10 million nodes/sec:

- 2^{80} = 4 billion years
- $4 \cdot 2^{20}$ = 0.4 seconds

Tree Decomposition

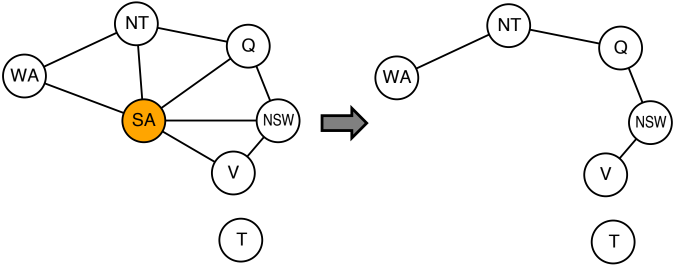
Convert constraint graphs to trees by assigning variables:

Properties

- Complexity: $O(d^c \cdot (n-c)d^2)$, given cutset size c
- Finding the smallest cycle cutset is NP-hard, but some efficient approximations exist

Tree Decomposition

Convert constraint graphs to trees by assigning variables:

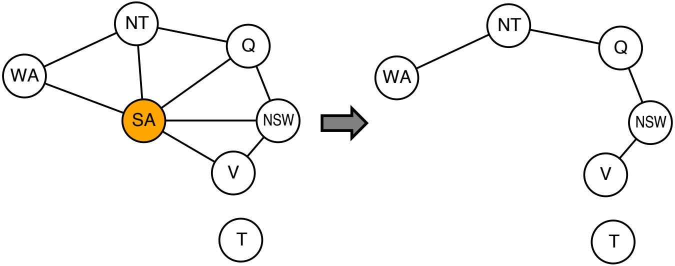


Properties

- Complexity: $O(d^c \cdot (n-c)d^2)$, given cutset size c
- Finding the smallest cycle cutset is NP-hard, but some efficient approximations exist

Tree Decomposition

Convert constraint graphs to trees by assigning variables:



Properties

- Complexity: $O(d^c \cdot (n - c)d^2)$, given cutset size c
- Finding the smallest cycle cutset is NP-hard, but some efficient approximations exist

Key Points

Constraint Satisfaction Problems

- States are assignment of values to variables
- Goals are assignments with no constraint violations

- Backtracking
  - Depth-first search, one variable assigned per node
  - Can be made effective with a number of heuristics
- Min-Conflicts
  - One value changed to reduce violations per iteration
  - Usually effective in practice
- Tree-Structured Search
  - Can be solved in linear time
  - Graphs can sometimes be decomposed into trees