Chapter 6: Constraint satisfaction problems

DIT410/TIN173 Artificial Intelligence

Peter Ljunglöf

(inspired by slides by Poole & Mackworth, Russell & Norvig, et al)

15 April, 2016

Outline

1. *Variables and constraints*
   - Russell & Norvig 6.1, Poole & Mackworth 4.1–4.2
2. *Soving CSPs using search*
   - Russell & Norvig 6.3–6.3.2, Poole & Mackworth 4.3–4.4
3. *Constraint progagation*
   - Russell & Norvig 6.2–6.2.2, Poole & Mackworth 4.5–4.6
4. *Local search for CSPs*
   - Russell & Norvig 6.4, Poole & Mackworth 4.8.1

Outline

1. *Variables and constraints*
   - Russell & Norvig 6.1, Poole & Mackworth 4.1–4.2
2. *Soving CSPs using search*
   - Russell & Norvig 6.3–6.3.2, Poole & Mackworth 4.3–4.4
3. *Constraint progagation*
   - Russell & Norvig 6.2–6.2.2, Poole & Mackworth 4.5–4.6
4. *Local search for CSPs*
   - Russell & Norvig 6.4, Poole & Mackworth 4.8.1

Constraint satisfaction problems (CSP)

Standard search problem:

- the state is a "black box" – any old data structure that supports goal test, cost evaluation, successor

CSP is a more specific search problem:

- the state is defined by *variables* X_i, taking *values* from the *domain* $\mathbf{D}_i$
- the goal test is a set of *constraints* specifying allowable combinations of values for subsets of variables

Since CSP is more specific, it allows useful algorithms with more power than standard search algorithms

States and variables

Just a few variables can describe many states:

n	binary variables can describe	2^n	states
10	binary variables can describe	2^{10}	$= 1{,}024$
20	binary variables can describe	2^{20}	$= 1{,}048{,}576$
30	binary variables can describe	2^{30}	$= 1{,}073{,}741{,}824$
100	binary variables can describe	2^{100}	$= 1{,}267{,}650{,}600{,}228{,}229,$ $401{,}496{,}703{,}205{,}376$

Hard and soft constraints

Given a set of variables, assign a value to each variable that either

- satisfies some set of constraints:
  - satisfiability problems — "hard constraints"
- minimizes some cost function, where each assignment of values to variables has some cost:
  - optimization problems — "soft constraints"

Many problems are a mix of hard and soft constraints
(called constrained optimization problems)

Relationship to search

CSP differences to general search problems:

- The path to a goal isn't important, only the solution is.
- There are no predefined starting nodes.
- Often these problems are huge, with thousands of variables, so systematically searching the space is infeasible.
- For optimization problems, there are no well-defined goal nodes.

Posing a CSP

A CSP is characterized by

- A set of variables $X_1, X_2, \ldots, X_n$.
- Each variable X_i has an associated domain $\mathbf{D}_i$ of possible values.
- There are hard constraints on various subsets of the variables which specify legal combinations of values for these variables.
- A solution to the CSP is an assignment of a value to each variable that satisfies all the constraints.

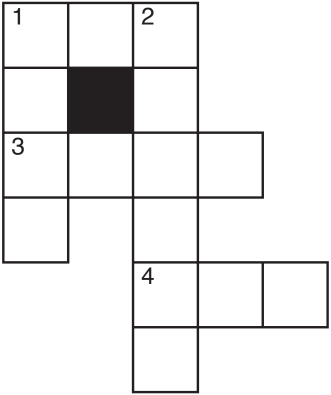
Example: Scheduling activities

Variables: A, B, C, D, E
representing the starting times of various activities.

Domains: $\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \mathbf{D}_D = \mathbf{D}_E = \{1, 2, 3, 4\}$

Constraints:
$$(B \neq 3) \land (C \neq 2) \land (A \neq B) \land (B \neq C) \land$$
$$(C < D) \land (A = D) \land (E < A) \land (E < B) \land$$
$$(E < C) \land (E < D) \land (B \neq D)$$

Example: Crossword puzzle



Words:

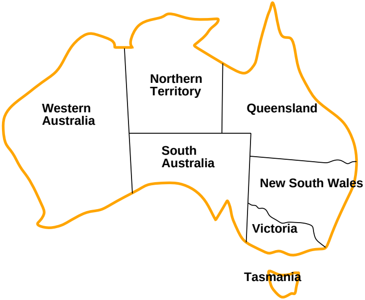
ant, big, bus, car, has
book, buys, hold,
lane, year
beast, ginger, search,
symbol, syntax

Dual representations

Many problems can be represented in different ways as a CSP, e.g., the crossword puzzle:

- First representation:
  - nodes represent word positions: 1-down. . . 6-across
  - domains are the words
  - constraints specify that the letters on the intersections must be the same
- Dual representation:
  - nodes represent the individual squares
  - domains are the letters
  - constraints specify that the words must fit

Example: Map colouring

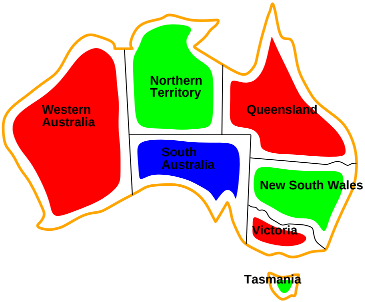


Variables: WA, NT, Q, NSW, V, SA, T

Domains: $\mathbf{D}_i = \{red, green, blue\}$

Constraints: adjacent regions must have different colors,
e.g., $WA \neq NT$, $WA \neq SA$, $NT \neq SA$, $NT \neq Q$, ...

Example: Map colouring



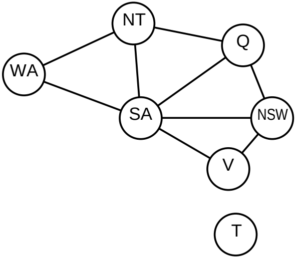
Solutions are assignments satisfying all constraints, e.g.,

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$$

Constraint graph

Binary CSP: each constraint relates at most two variables
(*note:* this does not say anything about the domains)

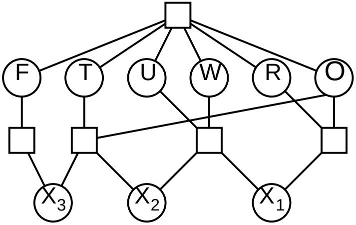
Constraint graph: nodes are variables, arcs show constraints



CSP algorithms can use the graph structure to speed up search,
e.g., Tasmania is an independent subproblem.

Example: Cryptarithmetic puzzle

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



Variables: $F, T, U, W, R, O, X_1, X_2, X_3$

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

Constraints: $Alldiff(F, T, U, W, R, O)$
$O + O = R + 10 \cdot X_1$, etc.

Note: This is not a binary CSP.

Varieties of CSPs

Discrete variables:

- Finite domains:
  - size $d \implies O(d^n)$ complete assignments
  - e.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
- Infinite domains (integers, strings, etc.)
  - e.g., job scheduling – variables are start/end times for each job
  - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
  - linear constraints are solvable – nonlinear undecidable

Continuous variables:

- e.g., scheduling for Hubble Telescope observations and manouvers
- linear constraints – solvable in polynomial time!

Varieties of constraints

Unary constraints involve a single variable:

- e.g., $SA \neq green$

Binary constraints involve pairs of variables:

- e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables:

- e.g., $Alldiff(WA, NT, SA)$

Preferences (soft constraints):

- e.g., *red* is better than *green*
- often representable by a cost for each variable assignment

Outline

1. *Variables and constraints*
   - Russell & Norvig 6.1, Poole & Mackworth 4.1–4.2
2. *Soving CSPs using search*
   - Russell & Norvig 6.3–6.3.2, Poole & Mackworth 4.3–4.4
3. *Constraint progagation*
   - Russell & Norvig 6.2–6.2.2, Poole & Mackworth 4.5–4.6
4. *Local search for CSPs*
   - Russell & Norvig 6.4, Poole & Mackworth 4.8.1

Generate-and-test algorithm

- Generate the assignment space $\mathbf{D} = \mathbf{D}_{V_1} \times \mathbf{D}_{V_2} \times \ldots \times \mathbf{D}_{V_n}$. Test each assignment with the constraints.
- *Example*:

$$
\begin{aligned}
\mathbf{D} &= \mathbf{D}_A \times \mathbf{D}_B \times \mathbf{D}_C \times \mathbf{D}_D \times \mathbf{D}_E \\
&= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \\
&\quad \times \{1,2,3,4\} \times \{1,2,3,4\} \\
&= \{\langle 1,1,1,1,1\rangle, \langle 1,1,1,1,2\rangle, \ldots, \langle 4,4,4,4,4\rangle\}.
\end{aligned}
$$

- How many assignments need to be tested for n variables each with domain size d?

CSP as a search problem

Let's start with the straightforward, dumb approach.

States are defined by the values assigned so far:

- Initial state: the empty assignment, $\{\}$
- Successor function: assign a value to an unassigned variable that does not conflict with current assignment
  $\Longrightarrow$ fail if there are no legal assignments
- Goal test: the current assignment is complete

Every solution appears at depth n (assuming n variables)
$\Longrightarrow$ we can use depth-first-search

At depth l, $b = (n - l)d$, where d is the domain size
$\Longrightarrow$ hence there are $n!d^n$ leaves!

Backtracking search

Variable assignments are *commutative*, i.e.:

- $[WA = red, NT = green]$ is the same as $[NT = green, WA = red]$

We only need to assign a single variable at each node:

- i.e., $b = d$, so there are d^n leaves (instead of $n!d^n$)

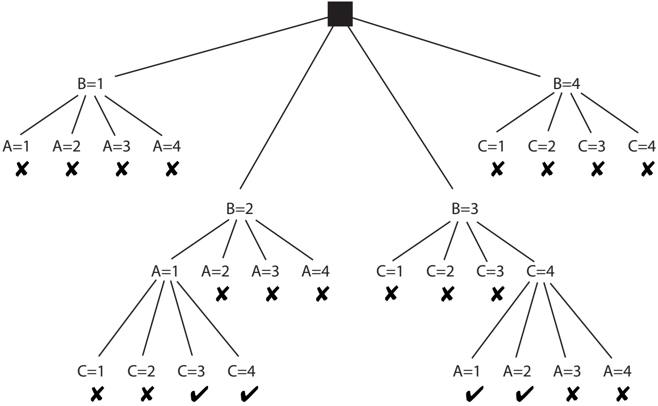
Depth-first search for CSPs with single-variable assignments is called *backtracking search*:

- backtracking search is the basic uninformed CSP algorithm
- it can solve n-queens for $n \approx 25$
- why not use breadth-first search?

Simple backtracking example

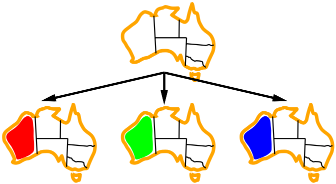
Variables: A, B, C. Domains: $\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \{1, 2, 3, 4\}$.

Constraints: $(A < B) \wedge (B < C)$.

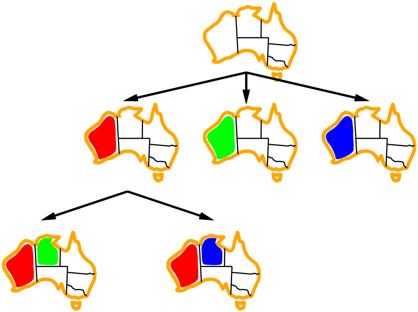


Example: Australia map colours

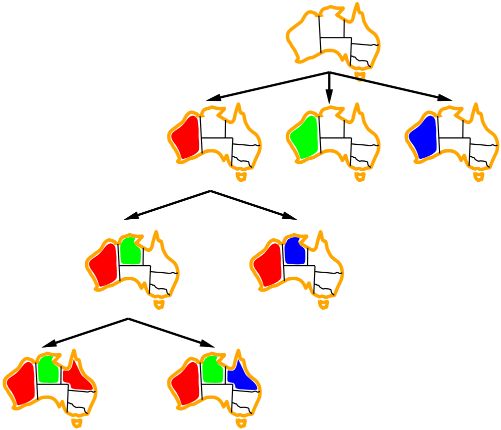
Example: Australia map colours



Example: Australia map colours



Example: Australia map colours



Algorithm for backtracking search

function BacktrackingSearch(*csp*):
    return Backtrack({}, *csp*)

function Backtrack(*assignment*, *csp*):
    if *assignment* is complete **then return** *assignment*
    var := SelectUnassignedVariable(*assignment*, *csp*)
    for each *value* **in** OrderDomainValues(*var*, *assignment*, *csp*):
        if *value* is consistent with *assignment* **then:**
            add $\{var = value\}$ **to** *assignment*
            inferences := Inference(*csp*, *var*, *value*)
            if $inferences \neq failure$ **then:**
                add *inferences* **to** *assignment*
                result := Backtrack(*assignment*, *csp*)
                if $result \neq failure$ **then return** *result*
            remove $\{var = value\}$ and *inferences* **from** *assignment*
    return *failure*

Improving backtracking efficiency

The general-purpose algorithm gives rise to several questions:

- Which variable should be assigned next?
  - SelectUnassignedVariable(*assignment*, *csp*)
- In what order should its values be tried?
  - OrderDomainValues(*var*, *assignment*, *csp*)
- What inferences should be performed at each step?
  - Inference(*csp*, *var*, *value*)
- Can the search avoid repeating failures?
  - "intelligent" backtracking (R&N 6.3.3, not covered in this course)

Selecting unassigned variables

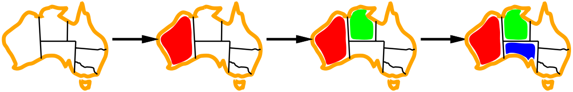
Heuristics for selecting the next unassigned variable:

- Minimum remaining values (MRV):
  - choose the variable with the fewest legal values
- Degree heuristic (if there are several MRV variables):
  - choose the variable with most constraints on remaining variables

Selecting unassigned variables

Heuristics for selecting the next unassigned variable:

- Minimum remaining values (MRV):
  - choose the variable with the fewest legal values

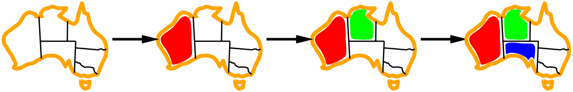


- Degree heuristic (if there are several MRV variables):
  - choose the variable with most constraints on remaining variables

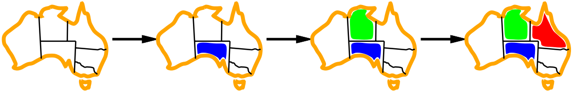
Selecting unassigned variables

Heuristics for selecting the next unassigned variable:

- Minimum remaining values (MRV):
  - choose the variable with the fewest legal values



- Degree heuristic (if there are several MRV variables):
  - choose the variable with most constraints on remaining variables



Ordering domain values

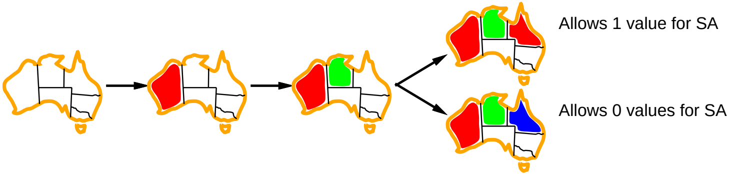
Heuristics for ordering the values of a selected variable:

- Least constraining value:
  - prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph

Ordering domain values

Heuristics for ordering the values of a selected variable:

- Least constraining value:
  - prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph



Inference: Forward checking

Forward checking is a simple form of inference.

- Keep track of remaining legal values for unassigned variables
  – terminate when any variable has no legal values left
- When a new variable is assigned,
  recalculate the legal values for its neighboring variables

Inference: Forward checking

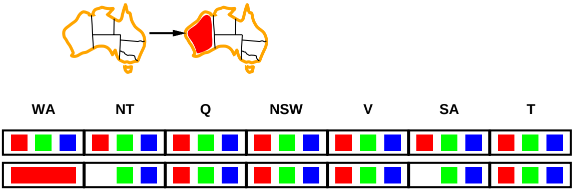
Forward checking is a simple form of inference.

- Keep track of remaining legal values for unassigned variables
  – terminate when any variable has no legal values left
- When a new variable is assigned,
  recalculate the legal values for its neighboring variables

Inference: Forward checking

Forward checking is a simple form of inference.

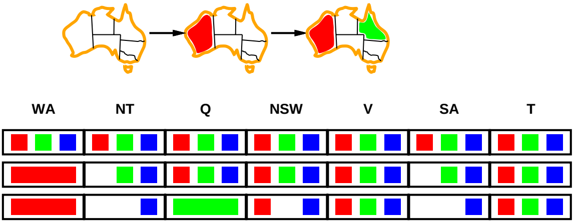
- Keep track of remaining legal values for unassigned variables
  – terminate when any variable has no legal values left
- When a new variable is assigned,
  recalculate the legal values for its neighboring variables



Inference: Forward checking

Forward checking is a simple form of inference.

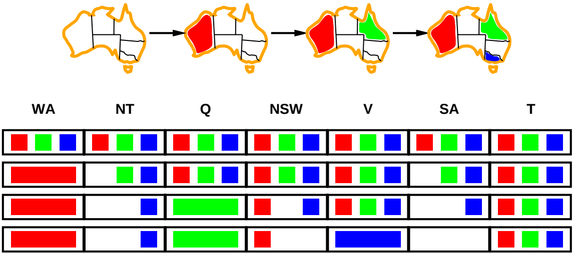
- Keep track of remaining legal values for unassigned variables
  – terminate when any variable has no legal values left
- When a new variable is assigned,
  recalculate the legal values for its neighboring variables



Inference: Forward checking

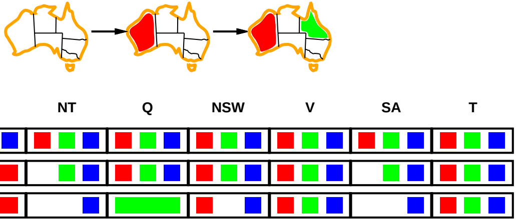
Forward checking is a simple form of inference.

- Keep track of remaining legal values for unassigned variables
  – terminate when any variable has no legal values left
- When a new variable is assigned,
  recalculate the legal values for its neighboring variables



Inference: Constraint propagation

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



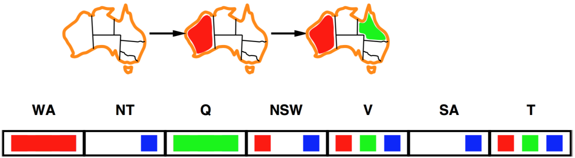
NT and SA cannot both be blue!

- *Constraint propagation* repeatedly enforces constraints locally

Inference: Arc consistency

The simplest form of propagation is to make each arc *consistent*:

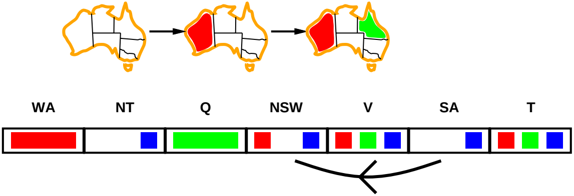
- $X \rightarrow Y$ is consistent iff
  - for *every* value x of X, there is *some* allowed value y in Y



Inference: Arc consistency

The simplest form of propagation is to make each arc *consistent*:

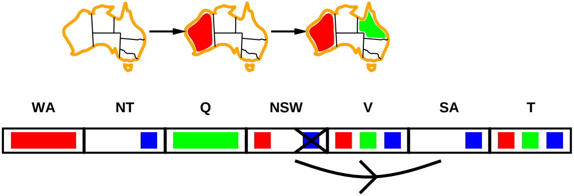
- $X \to Y$ is consistent iff
  - for *every* value x of X, there is *some* allowed value y in Y



Inference: Arc consistency

The simplest form of propagation is to make each arc *consistent*:

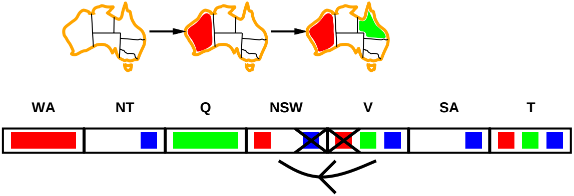
- $X \to Y$ is consistent iff
  - for *every* value x of X, there is *some* allowed value y in Y



Inference: Arc consistency

The simplest form of propagation is to make each arc *consistent*:

- $X \to Y$ is consistent iff
  - for *every* value x of X, there is *some* allowed value y in Y

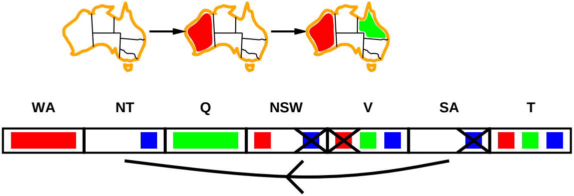


- If X loses a value, neighbors of X need to be rechecked

Inference: Arc consistency

The simplest form of propagation is to make each arc *consistent*:

- $X \to Y$ is consistent iff
  - for *every* value x of X, there is *some* allowed value y in Y



- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking

Outline

1. *Variables and constraints*
   - Russell & Norvig 6.1, Poole & Mackworth 4.1–4.2
2. *Soving CSPs using search*
   - Russell & Norvig 6.3–6.3.2, Poole & Mackworth 4.3–4.4
3. *Constraint progagation*
   - Russell & Norvig 6.2–6.2.2, Poole & Mackworth 4.5–4.6
4. *Local search for CSPs*
   - Russell & Norvig 6.4, Poole & Mackworth 4.8.1

Consistency

- A variable is *node-consistent* if all values in its domain satisfy its own unary constraints.
  - (Poole & Mackworth uses the term *domain-consistent*)
- A variable is *arc-consistent* if every value in its domain satisfies the variable's binary constraints.
  - *generalised arc-consistency* is the same, but for n-ary constraints
- There are also path consistency, k-consistency and general global constraints (R&N 6.2.3–6.2.5, not covered in this course)
- A network is X-consistent if every variable is X-consistent with every other variable.

Scheduling example (again)

Variables: A, B, C, D, E
representing the starting times of various activities.

Domains: $\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \mathbf{D}_D = \mathbf{D}_E = \{1, 2, 3, 4\}$

Constraints: $(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge$
$(C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge$
$(E < C) \wedge (E < D) \wedge (B \neq D)$

Is this example *node consistent*?

Scheduling example (again)

Variables: A, B, C, D, E
representing the starting times of various activities.

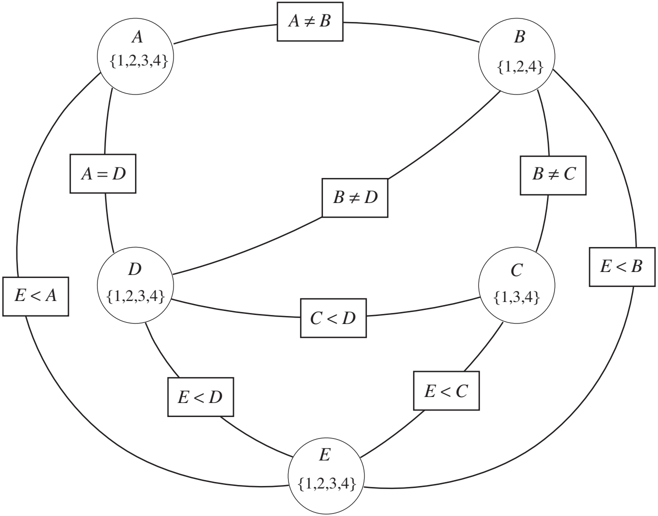
Domains: $\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \mathbf{D}_D = \mathbf{D}_E = \{1, 2, 3, 4\}$

Constraints: $(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge$
$(C < D) \wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge$
$(E < C) \wedge (E < D) \wedge (B \neq D)$

Is this example *node consistent*?

- $\mathbf{D}_B = \{1, 2, 3, 4\}$ is *not* node consistent,
  since $B = 3$ violates the constraint $B \neq 3$.
- $\mathbf{D}_C = \{1, 2, 3, 4\}$ is *not* node consistent,
  since $C = 2$ violates the constraint $C \neq 2$.

Scheduling example as a constraint graph



Arc consistency

- A binary arc (X, Y) is arc-consistent if:
  - for each value $x \in \mathbf{D}_X$, there is some $y \in \mathbf{D}_Y$ such that the constraint $r(x, y)$ is satisfied.
- More generally, an arc $(X, Y, Z, \dots)$ is arc-consistent if:
  - for each value $x \in \mathbf{D}_X$, there is some assignment $y, z, \dots \in \mathbf{D}_Y, \mathbf{D}_Z, \dots$ such that $r(x, y, z, \dots)$ is satisfied.
- What if arc (X, Y) is *not* arc consistent?
  - all values $x \in \mathbf{D}_X$ for which there is no corresponding $y \in \mathbf{D}_Y$ can be deleted from $\mathbf{D}_X$ to make the arc consistent.

Note! The arcs in a constraint graph are *directed* – (X, Y) and (Y, X) are considered as two different arcs

Arc consistency algorithm

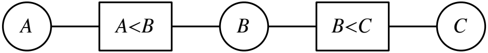
- The arcs can be considered in turn making each arc consistent.
- When an arc has been made arc consistent, does it ever need to be checked again?
  - An arc (X, Y) needs to be revisited if the domain of Y is revised.
- Three possible outcomes when all arcs are made arc consistent: (Is there a solution?)
  - One domain is empty $\Longrightarrow$
  - Each domain has a single value $\Longrightarrow$
  - Some domains have more than one value $\Longrightarrow$

Arc consistency algorithm

- The arcs can be considered in turn making each arc consistent.
- When an arc has been made arc consistent, does it ever need to be checked again?
  - An arc (X, Y) needs to be revisited if the domain of Y is revised.
- Three possible outcomes when all arcs are made arc consistent: (Is there a solution?)
  - One domain is empty $\Longrightarrow$ no solution
  - Each domain has a single value $\Longrightarrow$ unique solution
  - Some domains have more than one value $\Longrightarrow$ there may or may not be a solution

Quiz: Arc consistency

The variables and constraints are in the constraint graph:



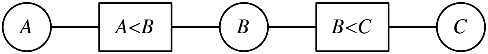
Assume the initial domains are $\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \{1, 2, 3, 4\}$.

How will the domains look like after making the graph arc consistent?

Note: AC in Russell&Norvig vs Poole&Mackworth

R&N and P&M have different formulations of the AC algorithm:

- For R&N, the arcs in the constraint graph are between variables, and they are labeled with the constraints.
  - i.e., constraints are labels, *not* nodes
  - the ABC graph below has 3 nodes and 4 labeled arcs (one arc in each direction)
- For P&M, the constraint graph has two kinds of nodes: variables and constraints
  - *Pro*: it can handle general n-ary constraints (not just binary)
  - *Con*: the graph data structure becomes more complex
  - the ABC graph below has 5 nodes and 4 unlabeled arcs



Maintaining arc-consistency

What if some domains have more than one element after AC?

- We can always resort to backtracking search
- Select a variable and a value using, e.g., MRV, degree heuristic, least constraining value
- Make the graph arc-consistent again
- Backtrack and try new values/variables, if AC fails
- Select a new variable/value, perform arc-consistency, etc.

Do we need to restart AC from scratch?

- no, only some arcs risk becoming inconsistent after the assignment of a new variable
- R&N calls this *Maintaining Arc Consistency* (MAC)

Domain splitting

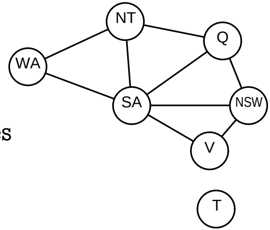
What if some domains are very big?

- Instead of trying to assign every possible value to a variable, we can split its domain
- Split one of the domains, then recursively solve each half
  - i.e., perform AC on the resulting graph, then split a domain, perform AC, split a domain, perform AC, split, etc.
- It is often best to split a domain in half
  - i.e., if $\mathbf{D}_X = \{1, \ldots, 1000\}$, we can split into $\{1, \ldots 500\}$ and $\{501, \ldots, 1000\}$

Problem structure

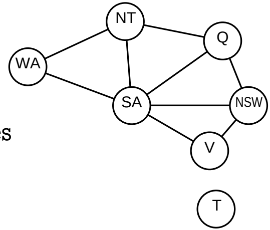
Tasmania and mainland are *independent subproblems* – identifiable as *connected components* of the constraint graph.

Suppose that each subproblem has c variables out of n total. The worst-case solution cost is $n/c \cdot d^c$, which is *linear* in n.



Problem structure

Tasmania and mainland are *independent subproblems* – identifiable as *connected components* of the constraint graph.



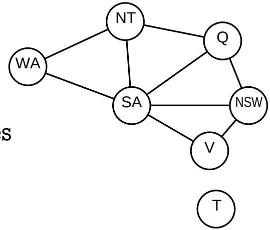
Suppose that each subproblem has c variables out of n total. The worst-case solution cost is $n/c \cdot d^c$, which is *linear* in n.

E.g., $n = 80$, $d = 2$, $c = 20$:

- $2^{80} = 4$ billion years at 10 million nodes/sec

Problem structure

Tasmania and mainland are *independent subproblems* – identifiable as *connected components* of the constraint graph.



Suppose that each subproblem has c variables out of n total. The worst-case solution cost is $n/c \cdot d^c$, which is *linear* in n.

E.g., $n = 80$, $d = 2$, $c = 20$:

- $2^{80} = 4$ billion years at 10 million nodes/sec

If we divide it into 4 equal-size subproblems:

- $4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec

Outline

1. *Variables and constraints*
   - Russell & Norvig 6.1, Poole & Mackworth 4.1–4.2
2. *Soving CSPs using search*
   - Russell & Norvig 6.3–6.3.2, Poole & Mackworth 4.3–4.4
3. *Constraint progagation*
   - Russell & Norvig 6.2–6.2.2, Poole & Mackworth 4.5–4.6
4. *Local search for CSPs*
   - Russell & Norvig 6.4, Poole & Mackworth 4.8.1

Local search for CSPs

Given an assignment of a value to each variable:

- A *conflict* is an unsatisfied constraint.
- The goal is an assignment with zero conflicts.

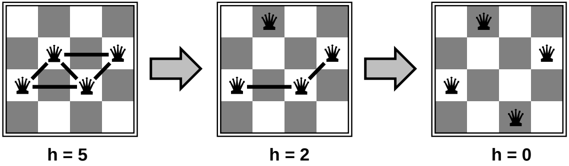
Local search / Greedy descent algorithm:

- Repeat until a satisfying assignment is found:
  - Select a variable to change
  - Select a new value for that variable
- Heuristic function to be minimized: the number of conflicts.
  - this is the *min-conflicts* heuristic in Russell & Norvig, 6.4
- *Note*: this does not always work! – *local minimum*

Example: n-queens (revisited)

Do you remember this example?

- Put n queens on an $n \times n$ board, in separate columns
- Conflicts = unsatisfied constraints = threatened queens
- Move a queen to reduce the number of conflicts; repeat until we cannot move any queen anymore
  - then we are at a local maximum, hopefully it is global too



Variants of greedy descent

To choose a variable to change and a new value for it:

- Find a variable-value pair that minimizes the number of conflicts
- Select a variable that participates in the most conflicts.
  Select a value that minimizes the number of conflicts.
- Select a variable that appears in any conflict.
  Select a value that minimizes the number of conflicts.
- Select a variable at random.
  Select a value that minimizes the number of conflicts.
- Select a variable and value at random;
  accept this change if it doesn't increase the number of conflicts.