Fundamentals of Artificial Intelligence – Constraint Satisfaction Problems

Matthias Althoff

TU München

Winter semester 2022/23

Organization

The content is covered in the AI book by the section "Constraint Satisfaction Problems".

Learning Outcomes

- You can explain the difference between constraint satisfaction problems (CSPs) and standard search problems.
- You can judge whether a problem is a CSP.
- You can create formally defined CSPs from a problem description.
- You can create constraint graphs.
- You can apply backtracking search.
- You can apply and decide when to use the following heuristics: *Minimum Remaining Values*, *Degree Heuristic*, and *Least Constraining Value*.
- You can apply techniques interleaving search and inference: *Forward Checking* and *Arc Consistency Algorithm*.
- You can exploit the structure of CSPs and reduce the complexity of solving tree-structured and nearly tree-structured CSPs.

Difference to Standard Search Problems

Standard Search Problems

Each **state** is atomic, or indivisible, and has no internal structure.

Constraint Satisfaction Problems (CSPs)

- We use a **factored representation** of each state: a set of variables, each of which has a value.
- The goal test is whether each variable has a value that satisfies all constraints of the problem.

Benefit: Allows useful general-purpose algorithms with more power than standard search algorithms by exploiting the structure of the states.

Set-Up

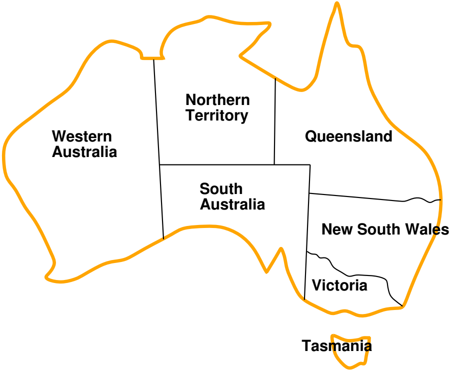
Constraint Satisfaction Problem

A constraint satisfaction problem is a tuple (X, D, C), where:

- $X = \{X_1, \dots, X_n\}$ is a set of variables,
- $D = \{D_1, \dots, D_n\}$ is a set of the respective domains of values, and
- $C = \{C_1, \dots, C_m\}$ is a set of constraints.

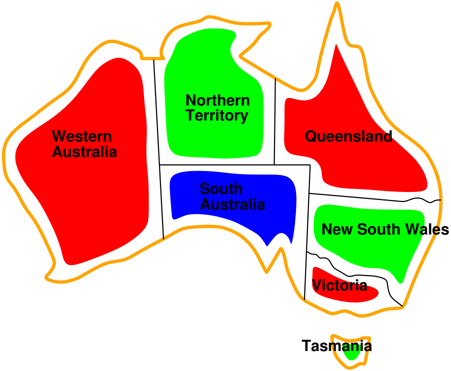
- Each domain D_i consists of a set of allowable values $\{v_1, \dots, v_k\}$ for variable X_i.
- Each constraint C_i consists of a pair $\langle \mathit{scope}, \mathit{rel} \rangle$, where *scope* is a tuple of variables that participate in the constraint and *rel* is a relation that defines the possible values.

Example Problem: Map Coloring



- **Variables**: $X = \{WA, NT, Q, NSW, V, SA, T\}$
- **Domains**: $D_i = \{red, green, blue\}$
- **Constraints**: adjacent regions must have different colors $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$; ($SA \neq WA$ is short for $\langle (SA, WA), SA \neq WA \rangle$)

Possible Solution of Map Coloring

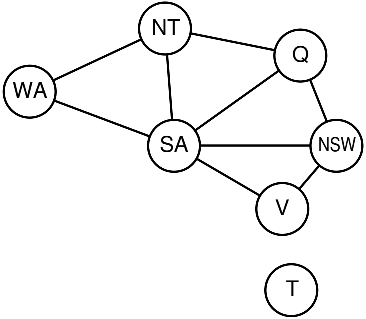


Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Constraint Graph

It can be helpful to visualize a constraint satisfaction problem as a **constraint graph**:

- Nodes correspond to variables,
- edges connect two variables that participate in a constraint.



Standard search would require searching all combinations, but

- by fixing e.g., $SA = blue$, none of the five neighbors can choose *blue*. This reduces from $3^5 = 243$ assignments in standard search to only $2^5 = 32$ (reduction by 87%).
- Tasmania is even an independent subproblem.

Varieties of Constraint Satisfaction Problems

Not relevant for the exam

Discrete domains

Finite domains; size $d \Rightarrow \mathcal{O}(d^n)$ complete assignments

- For instance, Boolean CSPs, incl. Boolean satisfiability (NP-complete).

Infinite domains (integers, strings, etc.)

- For instance, job scheduling, variables are start/end days for each job;
- requires a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$;
- **linear** constraints solvable, **nonlinear** undecidable.

Continuous domains (not part of this lecture)

- For instance, start/end times for Hubble Telescope observations.
- Linear constraints solvable in polynomial time by linear programming methods.

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., $SA \neq WA \neq NT$. Higher order constraints can be rewritten as several binary constraints. Previous example: $SA \neq WA$ and $WA \neq NT$.

  For that reason, we only consider binary constraints from now on.
- **Preferences** (soft constraints), e.g., *red* is better than *green* is often representable by a cost for each variable assignment
  $\rightarrow$ constrained optimization problems (not part of this lecture)

Real-World Constraint Satisfaction Problems

Not relevant for the exam

- **Assignment problems**, e.g., who teaches what class?
- **Timetabling problems**, e.g., which class is offered when and where?
- **Hardware configuration**, e.g., what kind of processor, memory, bus system, motherboards, etc., can be combined?
- **Spreadsheets**, e.g., check constraints to ensure correctness of data.
- **Transportation scheduling**, e.g., scheduling of trains so that changing trains is easy.
- **Factory scheduling**, e.g., determining in which order pieces have to be assembled.
- **Floorplanning**, e.g., how should the rooms in a house be arranged, when the living room should face south and the bathroom should have a window to the outside?

Many real-world problems involve real-valued variables.

Standard Search Formulation

Let's start with the naive search approach, then fix it.

States are defined by the values assigned so far:

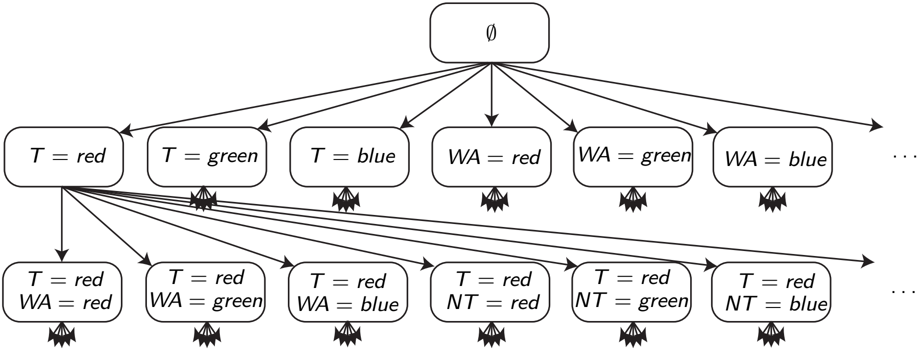
- **Initial state**: the empty assignment $\emptyset$.
- **Successor function**: assign a value to an unassigned variable not conflicting with the current assignment.
- **Goal test**: the current assignment is complete.

Comments:

1. This is the same for all CSPs! ☺
2. Every solution appears at depth n with n variables $\Rightarrow$ use depth-first search.
3. Path is irrelevant.
4. $b = (n - l)d$ at depth l, hence $n!d^n$ leaves in the worst case! ☹
   (n: nr. of variables, d: nr. of values; assumption: all variables have same nr. of values)

Example of the Naive Approach

We use the map coloring problem:



Number of nodes in the worst case:

- **First level**: $n \cdot d$ nodes (n: number of variables, d: number of values).
- **Second level**: $(n \cdot d)((n-1) \cdot d)$ nodes.
- n[th] **level**: $\prod_{l=0}^{n-1}(n-l) \cdot d = \prod_{l=0}^{n-1}(n-l) \cdot \prod_{l=0}^{n-1} d = n! d^n$ nodes.

Backtracking Search

- We can drastically improve the naive approach by considering that variable assignments are **commutative**:

  $[WA = red$ then $NT = green]$ same as $[NT = green$ then $WA = red]$

  $\Rightarrow$ Only consider assignments to a single variable at each node
  $\Rightarrow$ $b = d$ and there are only d^n leaves

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

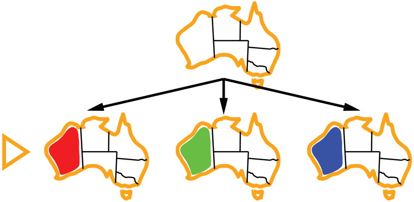
- Backtracking search is the basic uninformed algorithm for CSPs (e.g., solves the n-queens problem for $n \approx 25$).

Backtracking Search: Map-Coloring Example (1)

Part of the search tree for the map-coloring problem:

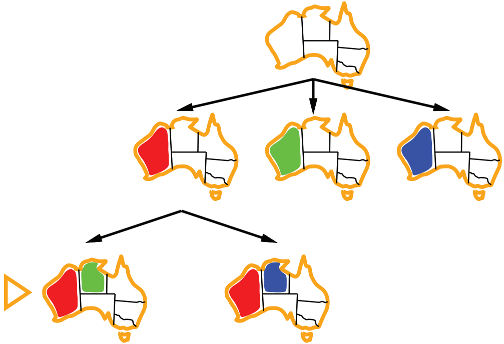
Backtracking Search: Map-Coloring Example (2)

Part of the search tree for the map-coloring problem:



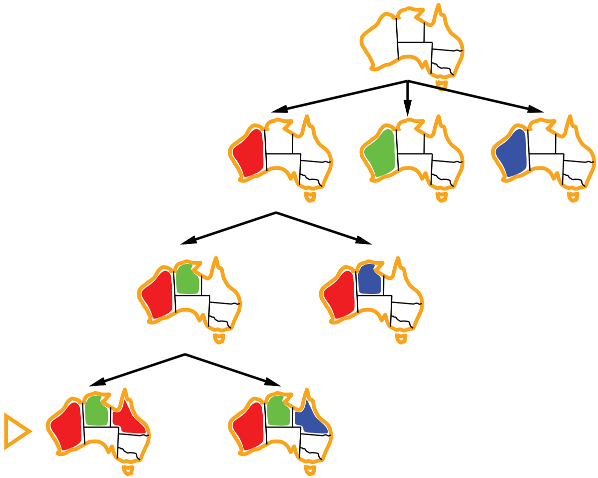
Backtracking Search: Map-Coloring Example (3)

Part of the search tree for the map-coloring problem:



Backtracking Search: Map-Coloring Example (4)

Part of the search tree for the map-coloring problem:



Backtracking Search: Algorithm

function Backtracking-Search (*csp*) **returns** solution/failure

```
return Recursive-Backtracking({ },csp)
```

function Recursive-Backtracking (*assignment*, *csp*) **returns** sol./failure

```
if assignment is complete then return assignment
var ← Select-Unassigned-Variable(csp)
for each value in Order-Domain-Values(var, assignment, csp) do
    if value is consistent with assignment given Constraints[csp] then
        add {var = value} to assignment
        inferences ← Inference(csp,var,value)
        if inferences ≠ failure then
            add inferences to assignment
            result ← Recursive-Backtracking(assignment,csp)
            if result ≠ failure then return result
            remove inferences from assignment
        remove {var = value} from assignment
return failure
```

Backtracking Search: Heuristics (1)

function Recursive-Backtracking (*assignment*, *csp*) **returns** sol./failure

if *assignment* is complete **then return** *assignment*
var ← `Select-Unassigned-Variable`(*csp*)
for each *value* **in** `Order-Domain-Values`(*var*, *assignment*, *csp*) **do**
  if *value* is consistent with *assignment* given `Constraints`[*csp*] **then**
    add $\{var = value\}$ to *assignment*
    inferences ← `Inference`(*csp*,*var*,*value*)
    if *inferences* $\neq$ *failure* **then**
      add *inferences* to *assignment*
      result ← `Recursive-Backtracking`(*assignment*,*csp*)
      if *result* $\neq$ *failure* **then return** *result*
      remove *inferences* from *assignment*
    remove $\{var = value\}$ from *assignment*
return *failure*

1. Which variable should be assigned next? (Sec. Variable Selection)

Backtracking Search: Heuristics (2)

function Recursive-Backtracking (*assignment*, *csp*) **returns** sol./failure

if *assignment* is complete **then return** *assignment*
var ← `Select-Unassigned-Variable`(*csp*)
for each *value* **in** `Order-Domain-Values`(*var*, *assignment*, *csp*) **do**
    if *value* is consistent with *assignment* given `Constraints`[*csp*] **then**
        add {*var* = *value*} to *assignment*
        inferences ← `Inference`(*csp*,*var*,*value*)
        if *inferences* ≠ *failure* **then**
            add *inferences* to *assignment*
            result ← `Recursive-Backtracking`(*assignment*,*csp*)
            if *result* ≠ *failure* **then return** *result*
            remove *inferences* from *assignment*
        remove {*var* = *value*} from *assignment*
return *failure*

1. Which variable should be assigned next? (Sec. Variable Selection)
2. In what order should its values be tried? (Sec. Value Selection)

Backtracking Search: Heuristics (3)

function Recursive-Backtracking (*assignment*, *csp*) **returns** sol./failure

```
if assignment is complete then return assignment
var ← Select-Unassigned-Variable(csp)
for each value in Order-Domain-Values(var, assignment, csp) do
    if value is consistent with assignment given Constraints[csp] then
        add {var = value} to assignment
        inferences ← Inference(csp,var,value)
        if inferences ≠ failure then
            add inferences to assignment
            result ← Recursive-Backtracking(assignment,csp)
            if result ≠ failure then return result
            remove inferences from assignment
        remove {var = value} from assignment
return failure
```

1. Which variable should be assigned next? (Sec. Variable Selection)
2. In what order should its values be tried? (Sec. Value Selection)
3. Can we detect inevitable failure early? (Sec. Interleaving Search and Inference)

Backtracking Search: Heuristics (4)

function Recursive-Backtracking (*assignment*, *csp*) **returns** sol./failure

if *assignment* is complete **then return** *assignment*
var ← `Select-Unassigned-Variable`(*csp*)
for each *value* **in** `Order-Domain-Values`(*var*, *assignment*, *csp*) **do**
    if *value* is consistent with *assignment* given `Constraints`[*csp*] **then**
        add $\{var = value\}$ to *assignment*
        inferences ← `Inference`(*csp*,*var*,*value*)
        if *inferences* $\neq$ *failure* **then**
            add *inferences* to *assignment*
            result ← `Recursive-Backtracking`(*assignment*,*csp*)
            if *result* $\neq$ *failure* **then return** *result*
            remove *inferences* from *assignment*
        remove $\{var = value\}$ from *assignment*
return *failure*

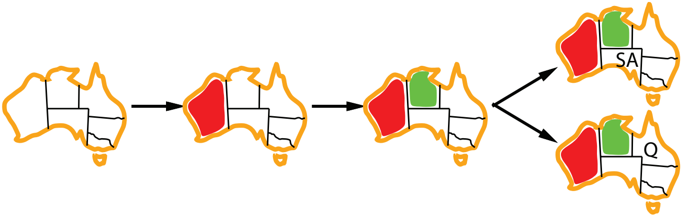
1. Which variable should be assigned next? (Sec. Variable Selection)
2. In what order should its values be tried? (Sec. Value Selection)
3. Can we detect inevitable failure early? (Sec. Interleaving Search and Inference)
4. Can we take advantage of problem structure? (Sec. The Structure of CSPs)

Variable Selection I: Minimum Remaining Values

The backtracking algorithm contains *var*← `Select-Unassigned-Variable`(*csp*).

- The simplest strategy is to choose the next unassigned variable in order $(\{X_1, X_2, \ldots\})$.
- The above strategy seldomly is the most efficient one.
  Example (see figure): for $WA = red$ and $NT = green$, there is only one possibility for SA, so it makes sense to assign $SA = blue$ rather than assigning Q. After SA is assigned, the choices for Q, NSW, and V are all forced.

The intuitive idea of choosing the variable with the fewest possible values first is called **minimum-remaining-values** (MRV) heuristic.



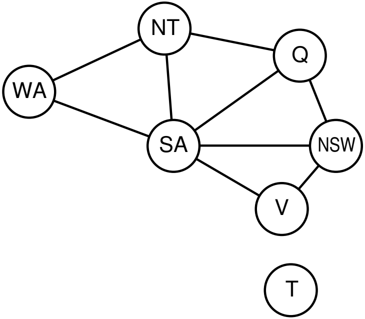
1 remaining value for SA

2 remaining values for Q

Variable Selection II: Degree Heuristic

- The MRV heuristic does not help in choosing the first region in Australia.
- A good choice is to select the variable that is involved in the largest number of constraints on other unassigned variables, called **degree heuristic**.

Example (see figure): SA is involved in 5 constraints, while other variables are only in 2 or 3 constraints, except for T. Once SA is chosen, continuing use of degree heuristic solves the problem without any false step.

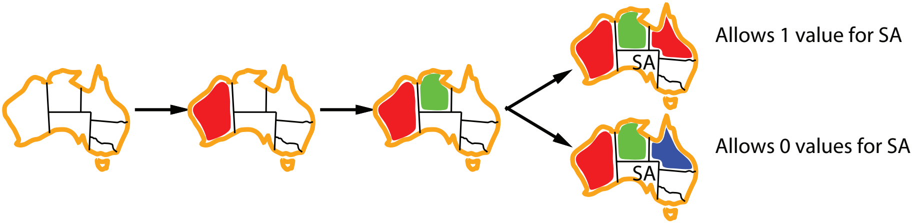


Value Selection: Least Constraining Value

The backtracking algorithm contains
for each *value* **in** `Order-Domain-Values`(*var*, *assignment*, *csp*) **do**.

- Optimal order of choosing values?
- A good choice is to select the value that rules out the fewest choices for neighboring values in the constraint graph, called **least-constraining-value** heuristic.

Example (see figure): $WA = red$ and $NT = green$ are selected and our next choice is Q. Blue would be a bad choice since it removes all options for the neighbor SA, while *red* leaves an option for SA. Thus, *red* is preferred.



Comments on Variable and Value Selection

Why should variable selection be fail-first, but value selection be fail-last?

- **Variable selection:** Choosing variables with the minimum number of remaining values helps to prune the search tree (in the end, each variable has to be selected anyways).
- **Value selection:** We only need one solution; therefore, it makes sense to look at the most likely values first.

It is not guaranteed that the proposed heuristics always provide fast solutions; however, they work well in most cases.

Inference in Constraint Satisfaction Problems

Inference

Act or process of deriving logical conclusions from known premises.

Inference can be applied at different times:

- **After each assignment:** see `Inference()` in backtracking algorithm.
- **As pre-processing:** before applying the backtracking algorithm.

Considered inference techniques (others exist)

- **Forward checking** (after each assignment): inconsistent values of neighboring variables are removed.
- **Arc consistency algorithm** (after each assignment or as pre-processing): inconsistent values of all variables are removed.

Definition of Arc Consistency

Arc consistency of a variable

Variable X_i is arc-consistent with variable X_j, if for every value in the domain D_i there exists a value in D_j satisfying the binary constraint of the arc (X_i, X_j).

Example: X is arc-consistent with Y for the constraint $Y = X^2$ if $D_X = \{0, 1, 2, 3\}$ and $D_Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, but Y is not arc-consistent with X (direction of the arc matters).

Arc consistency of a CSP

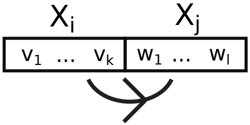
A constraint graph is arc-consistent if every variable is arc-consistent with every other variable.

Example: the constraint graph $Y = X^2$ is arc-consistent, if $D_X = \{0, 1, 2, 3\}$ and $D_Y = \{0, 1, 4, 9\}$.

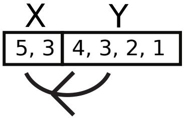
Visualization and Examples of Arc Consistency

Visualization

If variable X_i with domain $D_i = \{v_1, \ldots, v_k\}$ is arc-consistent with variable X_j with domain $D_j = \{w_1, \ldots w_l\}$, we visualize this by:

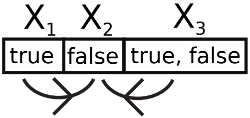


Examples: are the visualized arc-consistencies correct?



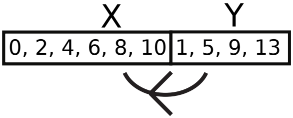
$Y < X < Y^2$

no



$X_1 \vee X_2,\ X_2 \Rightarrow X_3$

yes



$Y = 2X + 1$

yes

Inference I: Forward Checking (1)

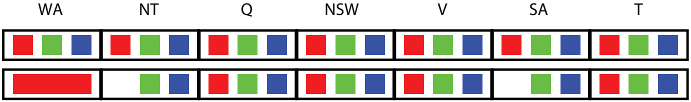
After a variable X_i is assigned during backtracking search, forward checking makes all variables X_j constrained with X_i arc-consistent with X_i.

Initial domains:

Inference I: Forward Checking (2)

After a variable X_i is assigned during backtracking search, forward checking makes all variables X_j constrained with X_i arc-consistent with X_i.

After assigning $WA = red$**:**

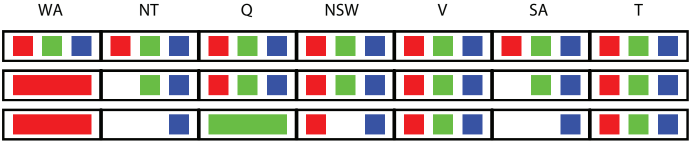


Comment: forward checking removes the value *red* from all neighbors of *WA* so that they are arc-consistent with *WA*. Thus, *NT* is arc-consistent with *WA*, and *SA* is arc-consistent with *WA*.

Inference I: Forward Checking (3)

After a variable X_i is assigned during backtracking search, forward checking makes all variables X_j constrained with X_i arc-consistent with X_i.

After assigning $Q = \mathit{green}$**:**

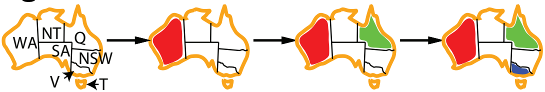


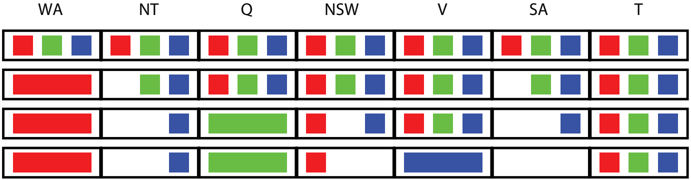
Comment: forward checking removes the value *green* from all neighbors of Q. Note that forward checking does not change the domains of other variables and does not assign variables whose domain only contains a single value.

Inference I: Forward Checking (4)

After a variable X_i is assigned during backtracking search, forward checking makes all variables X_j constrained with X_i arc-consistent with X_i.

After assigning $V = blue$**:**





Comment: forward checking results in a failure, since the domain for SA becomes empty. Thus, the backtracking search algorithm will backtrack: it revokes the last inconsistent assignment $V = blue$.

Inference II: Arc Consistency Alg. (ArcConsistencyAlgorithm.ipynb)

function AC-3 (*csp*, *queue*) **returns** *failure* or the reduced *csp* otherwise

inputs: *csp*: a binary CSP, *queue*: a queue of arcs (X_i, X_j)
while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ `Remove-First`(*queue*)
    if `Remove-Inconsistent-Values`(X_i, X_j) **then**
        if size of `Domain`$(X_i) = 0$ **then return** *failure*
        for each X_k **in** `Neighbors`$[X_i] \setminus \{X_j\}$ **do**
            add (X_k, X_i) to *queue*
return *csp*

function Remove-Inconsistent-Values (X_i, X_j) **returns** true iff succeeds

removed $\leftarrow$ *false*
for each x **in** `Domain`$[X_i]$ **do**
    if no value y in `Domain`$[X_j]$ allows (x,y) to satisfy the constraint of (X_i, X_j)
        then delete x from `Domain`$[X_i]$; *removed* $\leftarrow$ *true*
return *removed*

Time complexity is $\mathcal{O}(cd^3)$ (c: number of arcs, d: maximum domain size).

Initialization of the Arc Consistency Algorithm

The input variable *queue* is initialized depending on when AC-3 is used:

- **after each assignment:** after assigning variable X_i, add the arcs (X_j, X_i) to *queue*, where X_j are all unassigned neighbors of X_i.
- **as pre-processing:** add all arcs of the CSP to *queue*.

If the arc consistency algorithm terminates successfully, all variables of the CSP are arc-consistent with each other and possibly have reduced domains.

Arc Consistency Algorithm: Example (1)

Example for applying the arc consistency algorithm (AC-3) after an assignment.

In the initial map-coloring problem, we assigned $WA = red$, applied the arc consistency algorithm in `Inference`(*csp*,*var*,*value*), and just assigned $Q = green$.

Now, we again apply `Inference`(*csp*,*var*,*value*).

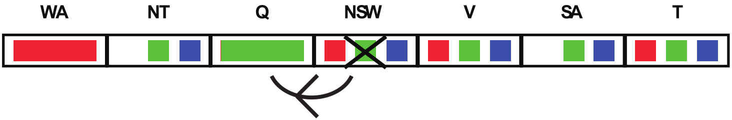
- add $(NSW, Q), (SA, Q), (NT, Q)$ to *queue*: since Q has just been assigned, the *queue* is initialized with all arcs to neighbors of Q
- call `AC-3`(*csp*,*queue*)

Arc Consistency Algorithm: Example (2)

function AC-3 (*csp*, *queue*) **returns** *failure* or the reduced *csp* otherwise

while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ `Remove-First`(*queue*)
    if `Remove-Inconsistent-Values`(X_i, X_j) **then**
        if size of `Domain`$(X_i) = 0$ **then return** *failure*
        for each X_k **in** `Neighbors`$[X_i] \setminus \{X_j\}$ **do**
            add (X_k, X_i) to *queue*

- $(NSW, Q) \leftarrow$ `Remove-First`(*queue*)
- `Remove-Inconsistent-Values`(NSW, Q) **returns** *true*: *green* removed from domain of *NSW*
- add $(V, NSW), (SA, NSW)$ to *queue*: neighbors of *NSW* (except *Q*)

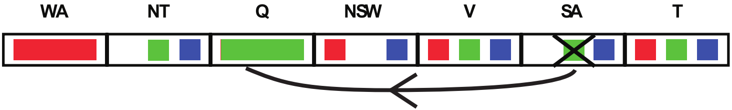


Arc Consistency Algorithm: Example (3)

function AC-3 (*csp*, *queue*) **returns** *failure* or the reduced *csp* otherwise

while *queue* is not empty **do**
  $(X_i, X_j) \leftarrow$ `Remove-First`$(queue)$
  if `Remove-Inconsistent-Values`(X_i, X_j) **then**
    if size of `Domain`$(X_i) = 0$ **then return** *failure*
    for each X_k **in** `Neighbors`$[X_i] \setminus \{X_j\}$ **do**
      add (X_k, X_i) to *queue*

- $(SA, Q) \leftarrow$ `Remove-First`$(queue)$
- `Remove-Inconsistent-Values`(SA, Q) **returns** *true*: *green* removed from domain of SA
- add $(WA, SA), (NT, SA), (NSW, SA), (V, SA)$ to *queue*: neighbors of SA (except Q)

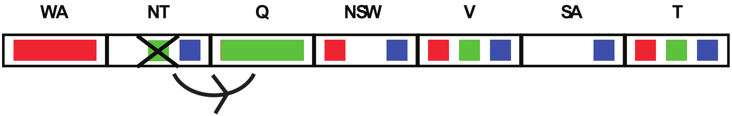


Arc Consistency Algorithm: Example (4)

function AC-3 (*csp*, *queue*) **returns** *failure* or the reduced *csp* otherwise

while *queue* is not empty **do**
  $(X_i, X_j) \leftarrow$ `Remove-First`(*queue*)
  if `Remove-Inconsistent-Values`(X_i, X_j) **then**
    if size of `Domain`$(X_i) = 0$ **then return** *failure*
    for each X_k **in** `Neighbors`$[X_i] \setminus \{X_j\}$ **do**
      add (X_k, X_i) to *queue*

- $(NT, Q) \leftarrow$ `Remove-First`(*queue*)
- `Remove-Inconsistent-Values`(NT, Q) **returns** *true*: *green* removed from domain of NT
- add $(WA, NT), (SA, NT)$ to *queue*: neighbors of NT (except Q)

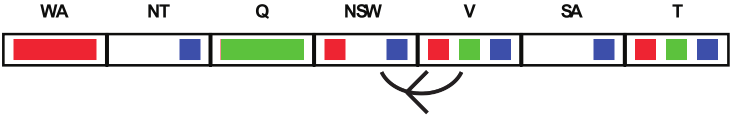


Arc Consistency Algorithm: Example (5)

function AC-3 (*csp*, *queue*) **returns** *failure* or the reduced *csp* otherwise

while *queue* is not empty **do**
  $(X_i, X_j) \leftarrow$ `Remove-First`$(queue)$
  if `Remove-Inconsistent-Values`(X_i, X_j) **then**
    if size of `Domain`$(X_i) = 0$ **then return** *failure*
    for each X_k **in** `Neighbors`$[X_i] \setminus \{X_j\}$ **do**
      add (X_k, X_i) to *queue*

- $(V, NSW) \leftarrow$ `Remove-First`$(queue)$
- `Remove-Inconsistent-Values`(V, NSW) **returns** *false*: V is already arc-consistent with NSW

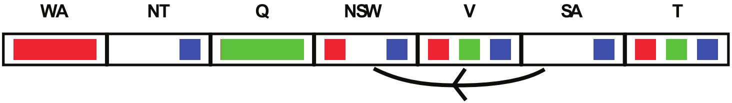


Arc Consistency Algorithm: Example (6)

function AC-3 (*csp*, *queue*) **returns** *failure* or the reduced *csp* otherwise

while *queue* is not empty **do**
  $(X_i, X_j) \leftarrow$ `Remove-First`(*queue*)
  if `Remove-Inconsistent-Values`(X_i, X_j) **then**
    if size of `Domain`$(X_i) = 0$ **then return** *failure*
    for each X_k **in** `Neighbors`$[X_i] \setminus \{X_j\}$ **do**
      add (X_k, X_i) to *queue*

- $(SA, NSW) \leftarrow$ `Remove-First`(*queue*)
- `Remove-Inconsistent-Values`(SA, NSW) **returns** *false*: SA is already arc-consistent with NSW



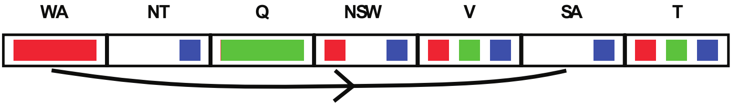
Comment: even though SA is arc-consistent with NSW, NSW is not arc-consistent with SA.

Arc Consistency Algorithm: Example (7)

function AC-3 (*csp*, *queue*) **returns** *failure* or the reduced *csp* otherwise

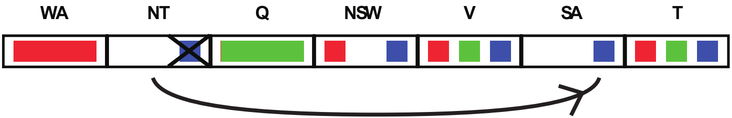
while *queue* is not empty **do**
  $(X_i, X_j) \leftarrow$ `Remove-First`$(queue)$
  if `Remove-Inconsistent-Values`(X_i, X_j) **then**
    if size of `Domain`$(X_i) = 0$ **then return** *failure*
    for each X_k **in** `Neighbors`$[X_i] \setminus \{X_j\}$ **do**
      add (X_k, X_i) to *queue*

- $(WA, SA) \leftarrow$ `Remove-First`$(queue)$
- `Remove-Inconsistent-Values`(WA, SA) **returns** *false*: *WA* is already arc-consistent with *SA*



Arc Consistency Algorithm: Example (8)

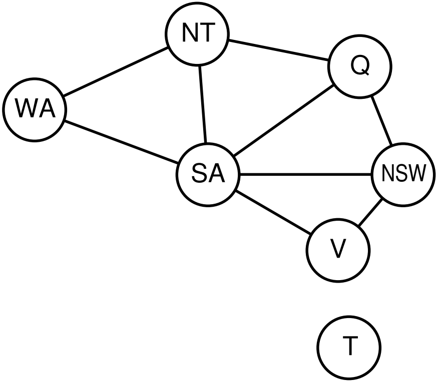
- $(NT, SA) \leftarrow$ `Remove-First`$(queue)$
- `Remove-Inconsistent-Values`(NT, SA) **returns** *true*: *blue* removed from domain of NT
- `AC-3` **returns** *failure*: size of `Domain`$(NT) = 0$



Result of applying AC-3: the backtracking search algorithm receives a *failure* and knows that the CSP cannot be solved with the last assignment $Q = green$. Thus, the search algorithm backtracks by removing the inconsistent assignment and restoring the modified domains.

Further examples of applying AC-3 are presented in the exercise.

Problem Structure



Tasmania and the mainland are **independent subproblems**.

Identifiable as **connected components** of the constraint graph.

Complexity Reduction of Independent Problems

Completely independent problems provide a fantastic simplification:

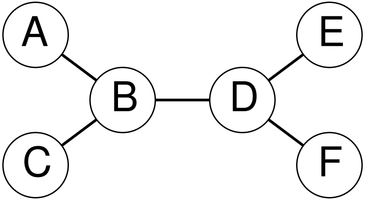
- Suppose each subproblem has c variables out of n total.
- Worst-case solution cost is $n/c \cdot d^c$, **linear** in n.
- Worst-case solution cost of the full problem is d^n, **exponential** in n.

Example

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

- full problem: $2^{80} = 4$ billion years at 10 million nodes/sec (approximate time until earth consumed by the sun)
- independent problems: $4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec

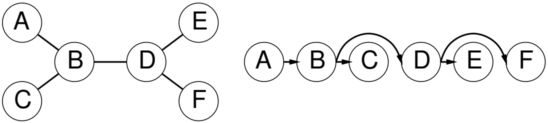
Tree-Structured CSPs



- If the constraint graph has no loops, the CSP can be solved in $\mathcal{O}(n\, d^2)$ time (n: nr. of variables, d: domain size).
- Compare to general CSPs, where worst-case time is $O(d^n)$.
- This property also applies to logical and probabilistic reasoning.

Tree-Structured CSPs: Algorithm

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering (**topological sort**).



Any tree with n nodes has only $n - 1$ arcs, so that topological sort is $\mathcal{O}(n)$.

2. The obtained variable order $X_1, X_2, \ldots, X_n$ is made directly arc-consistent.

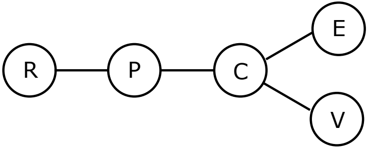
Direct arc consistency

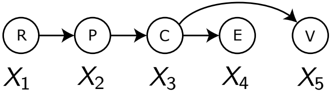
A CSP is direct arc-consistent for the ordered variables $X_1, X_2, \ldots, X_n$ if and only if every X_i is arc-consistent with each directly connected X_j for $j > i$.

The complexity is $\mathcal{O}(nd^2)$ since for each node, two variables with d possible domain values have to be compared pairwise.

3. We can march down the list of variables in the directed arc-consistent graph and choose any remaining value without backtracking.

Tree-Structured CSPs: Example (1)



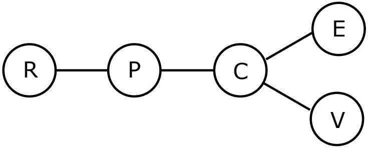


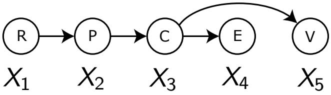
Option 1: The initial domains of all countries are $D_i = \{red, green, blue\}$

$$D_5 = \{red, green, blue\}$$
$$D_4 = \{red, green, blue\}$$
$$D_3 = \{red, green, blue\}$$
$$D_2 = \{red, green, blue\}$$
$$D_1 = \{red, green, blue\}$$

We select $X_1 = red$, $X_2 = green$, $X_3 = blue$, $X_4 = red$, $X_5 = green$.

Tree-Structured CSPs: Example (2)



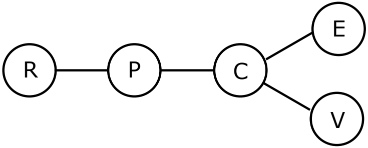


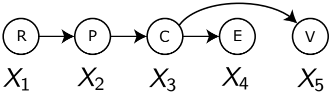
Option 2: The initial domains of all countries are $D_i = \{red, green, blue\}$, except that $D_5 = \{blue\}$

$$D_5 = \{blue\}$$
$$D_4 = \{red, green, blue\}$$
$$D_3 = \{red, green\}$$
$$D_2 = \{red, green, blue\}$$
$$D_1 = \{red, green, blue\}$$

We select $X_1 = red$, $X_2 = green$, $X_3 = red$, $X_4 = green$, $X_5 = blue$.

Tree-Structured CSPs: Example (3)





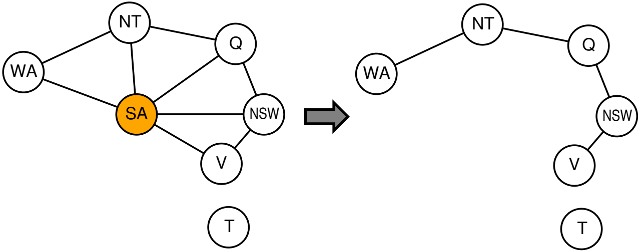
Option 3: The initial domains of all countries are $D_i = \{red, green, blue\}$, except that $D_5 = \{blue\}$ and $D_4 = \{green\}$

$$D_5 = \{blue\}$$
$$D_4 = \{green\}$$
$$D_3 = \{red\}$$
$$D_2 = \{green, blue\}$$
$$D_1 = \{red, green, blue\}$$

We select $X_1 = red$, $X_2 = green$, $X_3 = red$, $X_4 = green$, $X_5 = blue$.

Nearly tree-structured CSPs: Conditioning (1)

Conditioning: instantiate a variable, prune its neighbors' domains.



The value chosen for *SA* could be wrong, requiring us to try all values:

1. Choose a subset of variables $S \subset X$ such that the constraint graph becomes a tree after removing S.
2. For each possible constraint-satisfying assignment to variables in S,
   1. remove values from the other domains inconsistent with S, and
   2. if the remaining CSP has a solution, return it with the one of S.

Nearly tree-structured CSPs: Conditioning (2)

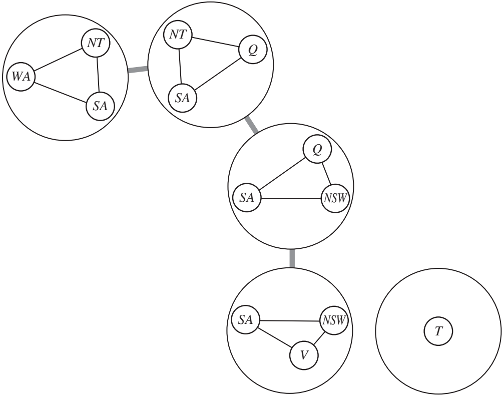
A few comments:

- Size of S is c: runtime $\mathcal{O}(d^c \cdot (n-c)d^2)$.
  Explanation: We try each of the d^c combinations in S, and for each one we solve a tree problem of size $n-c$.
- For small c, the approach is very fast. In the worst case, c can become as large as $n-2$.
- Finding the smallest S to obtain a tree-structure is NP-hard, but several efficient approximation algorithms are known.

Not for the Exam

Nearly tree-structured CSPs: Tree Decomposition (1)

Tree Decomposition: decomposing the CSP into a “super-tree” of subproblems.



Requirements:

- Every variable X_i appears in at least one subproblem.
- If variables X_i and X_j are connected in the original problem, they must be connected in at least one subproblem.
- If a variable appears in two subproblems, it must appear in every subproblem along the connecting path of the super-tree.

Nearly tree-structured CSPs: Tree Decomposition (2)

- We solve each subproblem independently.
- If any subproblem has no solution, the entire problem has no solution.
- If we can solve all subproblems, we attempt to construct a global solution as follows:
  1. We view each subproblem as a "super-variable" $\hat{X}_i$ whose domain $\hat{D}_i$ is the set of all solutions of the subproblem.
     Example (leftmost subproblem):

$$\begin{aligned}\hat{D}_1 = \Big\{ & \{WA = red, SA = blue, NT = green\}, \\ & \{WA = red, SA = green, NT = blue\}, \\ & \{WA = green, SA = blue, NT = red\}, \\ & \{WA = green, SA = red, NT = blue\}, \\ & \{WA = blue, SA = red, NT = green\}, \\ & \{WA = blue, SA = green, NT = red\}, \Big\}\end{aligned}$$

Nearly tree-structured CSPs: Tree Decomposition (3)

2. We solve the constraints connecting the subproblems, using the efficient algorithm for trees. The constraints between subproblems simply insist that the solutions of the subproblems agree on their shared variables.
   Example (leftmost solution):

$$\{WA = red, SA = blue, NT = green\},$$

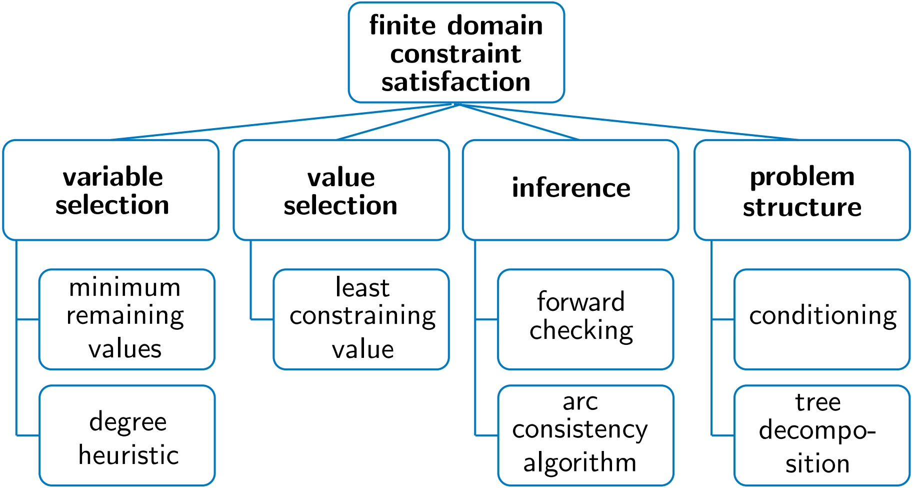
   the only consistent solution for the next subproblem is

$$\{SA = blue, NT = green, Q = red\}$$

   as can be verified using

$$\begin{aligned}\hat{D}_2 = \Big\{&\{SA = red, NT = blue, Q = green\},\\ &\{SA = red, NT = green, Q = blue\},\\ &\{SA = green, NT = blue, Q = red\},\\ &\{SA = green, NT = red, Q = blue\},\\ &\{SA = blue, NT = red, Q = green\},\\ &\{SA = blue, NT = green, Q = red\},\Big\}\end{aligned}$$

Overview of Constraint Satisfaction Methods



Summary

- **Constraint satisfaction problems** (CSPs) require finding evaluations of variables within their domains to satisfy a set of constraints.
- **Backtracking search**, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.
- The **minimum-remaining-values** and **degree** heuristics are domain-independent methods to choose the next variable in backtracking search.
- The **least-constraining-value** heuristics helps in selecting the first value for a variable.
- The complexity of solving CSPs strongly depends on the structure of their constraint graph:
  - Tree-structured problems can be solved in linear time.
  - **Conditioning** and **tree decomposition** can partially transform a problem into a tree-structured one.