CS 480

Introduction to Artificial Intelligence

September 23rd, 2021

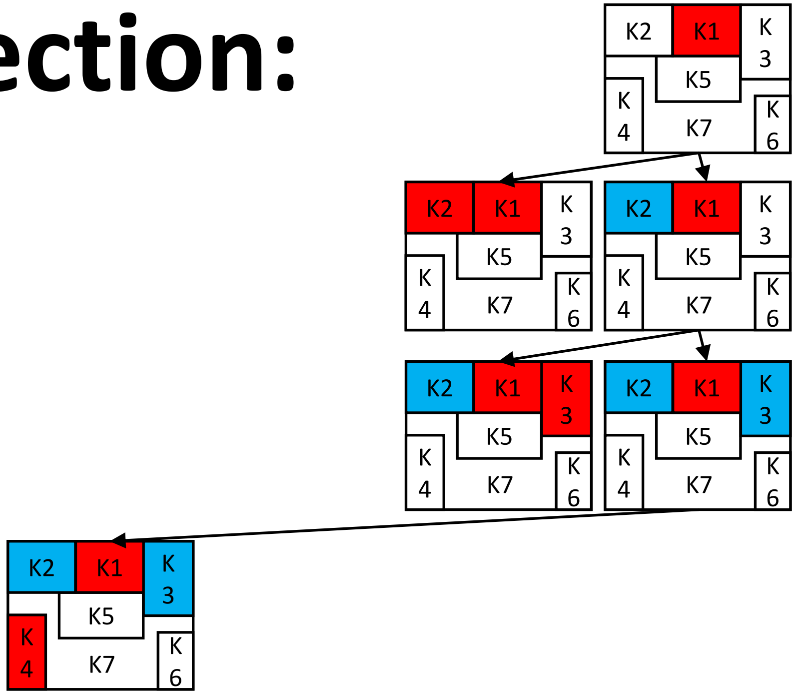
Announcements / Reminders

- **Programming Assignment #01:**
  - **will be posted this Friday and you will have three (3) weeks to complete it**
- **Contribute to the discussion on Blackboard**
- **Please follow the Week 05 To Do List instructions**
- **Fall Semester midterm course evaluation will be opened on Monday**
  - **I would love to hear your feedback. Please participate if you can. Thank you!**

Plan for Today

- **Constraint Satisfaction Problems: Continued**
- **Logical agents and reasoning: Introduction**

Correction:



K1 = ???

K2 = ???

K3 = ???

K4 = ???

Which variable to explore next (ignore the EXPECTED sequence on the right)? Available options:

K5: {GREEN}

K6: {RED, BLUE, GREEN}

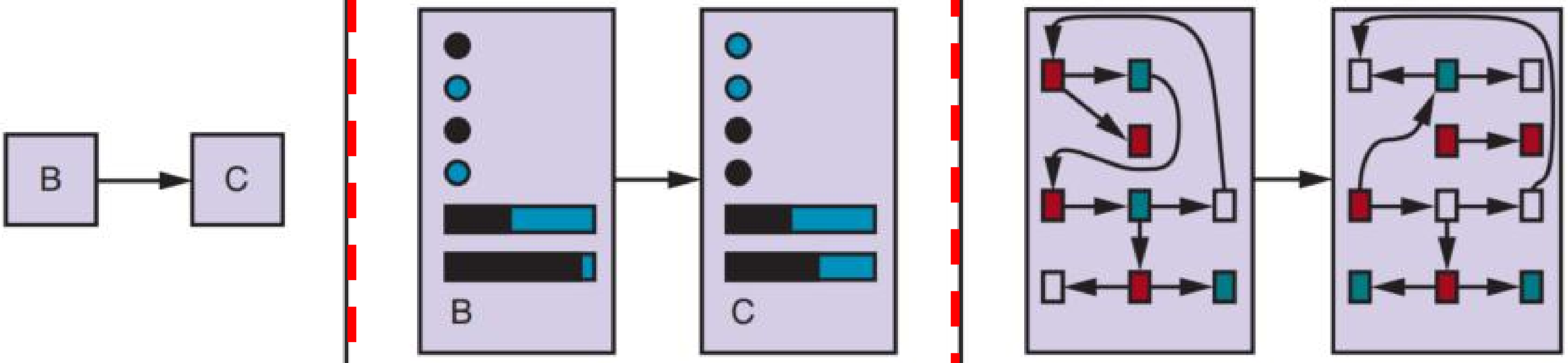
K7: {GREEN}

MRV should pick K5 or K7 ("fail first" variable). Tie needs to be resolved.

CSP Backtracking: Problems

- **Thrashing: keeps repeating the same failed variable assignments**
  - **What can help?**
    - **consistency checking**
    - **intelligent backtracking schemes**
- **Inefficiency: can explore areas of the search tree that aren't likely to succeed**
  - **What can help?**
    - **variable ordering (see last lecture)**
- **General strategies:**
  - **try to detect inevitable failure early**
  - **order variables and values in a smart way (see last lecture)**

How CSP Can Reduce Work



(a) Atomic (b) Factored (c) Structured

Next move?

- **Expand the node and visit succesors**

Next move?

- **Expand the node (assign value to a variable) and visit successors**
- **Infer where to go from current assignment and constraints (constraint propagation)**

CSP: More Pruning with Inference

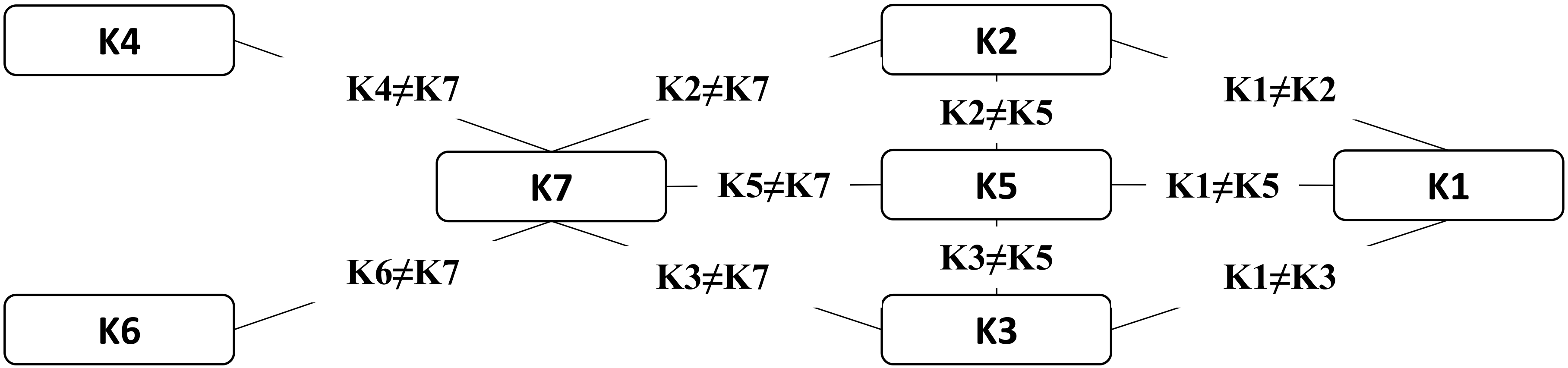
```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

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

With the information available to you, you can INFER that a particular branch is going to be INCONSISTENT

Inference in CSP

- **Simplifying the problem:**
  - **preprocessing / pre-check or part of the search**
  - **it can reduce the problem OR even solve it**
- **Inference with Constraint Propagation:**
  - **use constraint graph to enforce consistency locally**

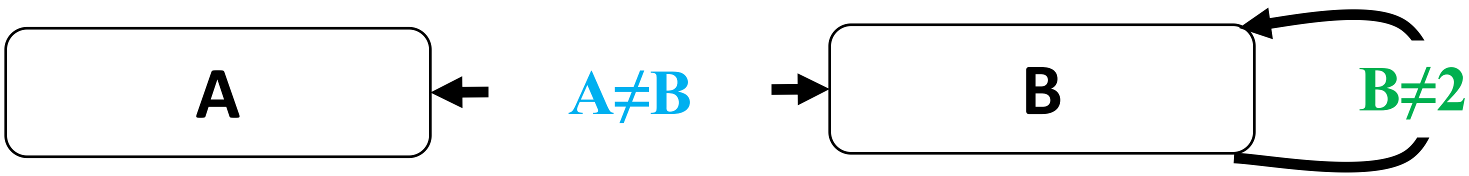


Local Consistency

- **The idea:**
  - **remove inconsistent values from variable domains as we go as they would make certain assignments inconsistent later anyway**
- **Types:**
  - **Node consistency**
  - **Arc consistency (or edge consistency)**
  - **Path consistency**

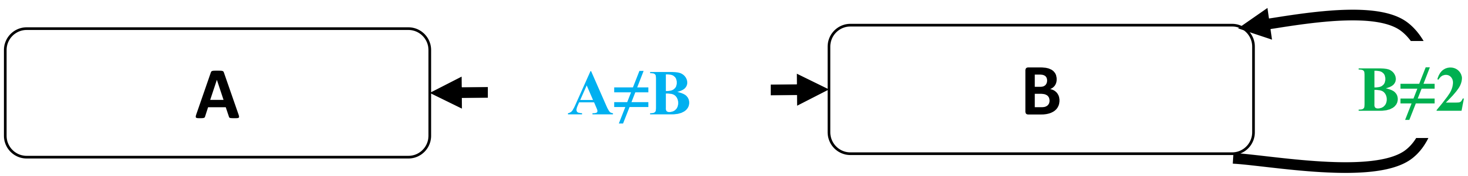
Node Consistency

- **Consider the following CSP example:**
  - **variables:** $X = \{A, B\}$
  - **domains:**
    - $D_A = \{0, 1, 3\}$
    - $D_B = \{2, 3, 4\}$
  - **constraints:** $C = \{A \neq B, B \neq 2\}$
    - **one binary and one unary constraint**
  - **constraint graph:**



Node Consistency

- **The idea:**
  - **a single variable is node-consistent (in a constraint graph) if all the values in its domain satisfy variable <u>unary</u> constraints**
- **(Constraint) graph is node-consistent if every variable in the graph is node-consistent**

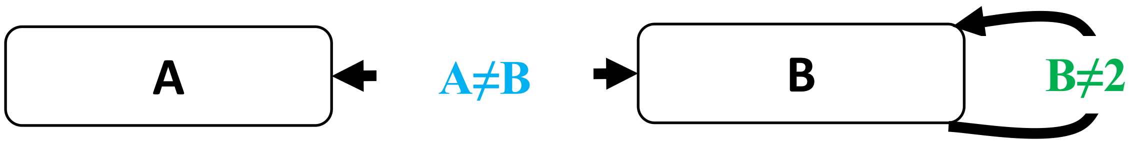


Variable B is NOT node-consistent because in $D_B = \{2,3,4\}$ value 2 does not satisfy unary B≠2

- **Approach: remove unary constraints by reducing variable domain**

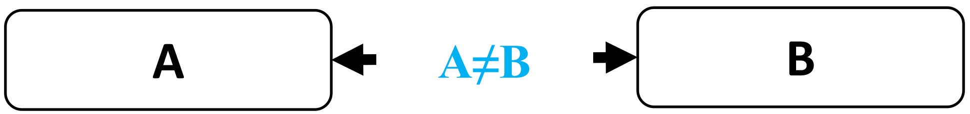
Node Consistency

- **Unary constraints can easily be removed to reduce the problem:**
  - **BEFORE (unary constraint removal) domains:**
    - $D_A = \{0, 1, 3\}$
    - $D_B = \{2, 3, 4\}$



Constraint graph is NOT node-consistent because of variable B

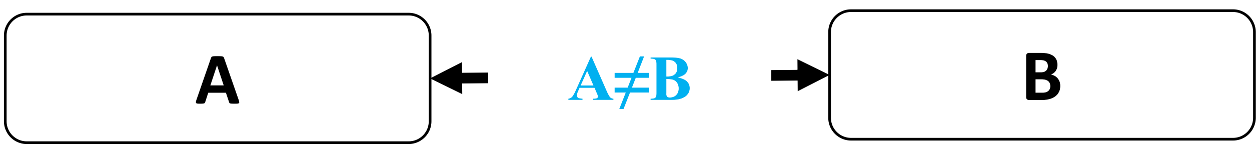
  - **AFTER (unary constraint removal) domains:**
    - $D_A = \{0, 1, 3\}$
    - $D_B = \{3, 4\}$



Constraint graph is node-consistent

Arc (Edge) Consistency

- **The idea:**
  - **a single variable is arc-consistent (in a constraint graph) if all the values in its domains satisfy ALL its binary constraints**
- **(Constraint) graph is arc-consistent if every variable in the graph is arc-consistent**



Variables A and B are NOT arc-consistent because in $D_A = \{1,2,3\}$ **and** $D_B = \{3,4\}$ **value** 3 **clashes**

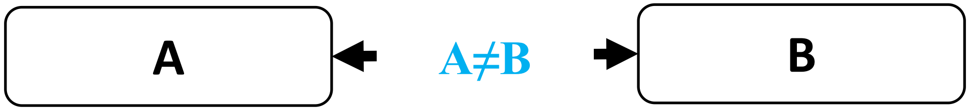
- **Approach: reducing variable domains to remove clashes**

Arc (Edge) Consistency

- **Values that clash can be removed from variable domains to reduce the problem:**
  - **BEFORE (clashing value(s) removal) domains:**
    - $D_A = \{0, 1, 3\}$
    - $D_B = \{3, 4\}$

Constraint graph is NOT arc-consistent because of value 3 clashing in both domains

  - **AFTER (clashing value(s) removal) domains:**
    - $D_A = \{0, 1, 3\}$
    - $D_B = \{4\}$ or
    - $D_A = \{0, 1\}$
    - $D_B = \{3, 4\}$ (depends on: which variable we start with)



Constraint graph is arc-consistent

AC-3 Algorithm: Pseudocode

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  $\boxed{queue \leftarrow \text{a queue of arcs, initially all the arcs in } csp}$

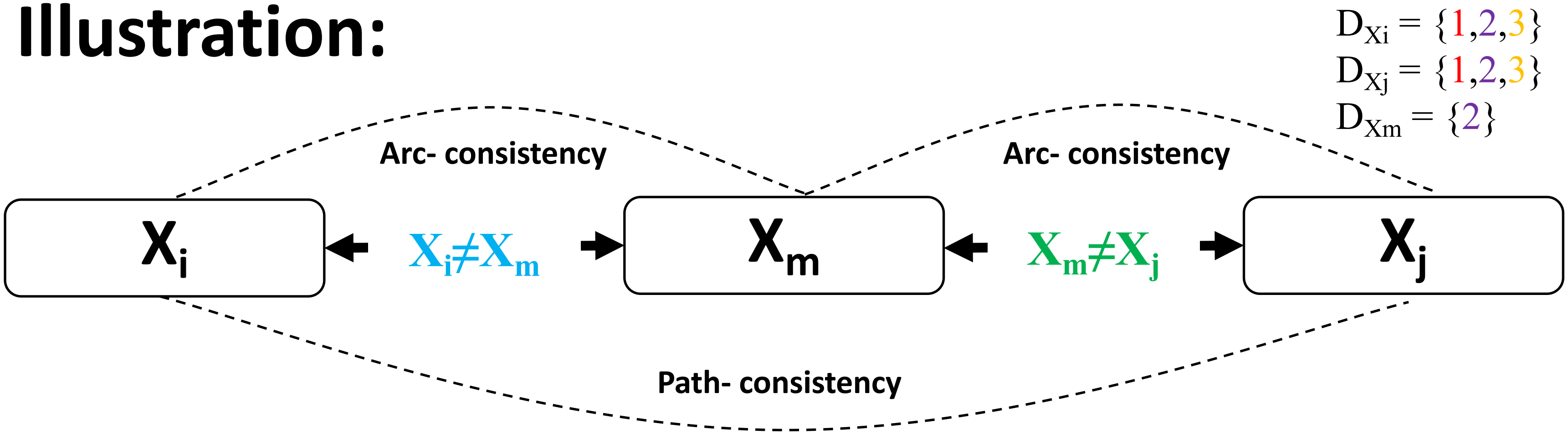
  while *queue* is not empty **do**
    $\boxed{(X_i,\ X_j) \leftarrow \text{POP}(queue)}$
    if REVISE(csp, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add $(X_k,\ X_i)$ to *queue*
  return *true*

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

Note: treat a constraint graph edge as two directional edges: constraint $X_i \neq X_j$ corresponds to edges (X_i, X_j) and (X_j, X_i)

Path Consistency

- **The idea:**
  - **two variable set $\{X_i, X_j\}$ is path-consistent (in a constraint graph) with respect to a third variable X_m if for EVERY assignment $\{X_i = a, X_j = b\}$ there is an assignment to X_m (between X_i and X_j) that satisfies constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$.**
- **Illustration:**



Path Consistency

- **NOT path-consistent assignment** $\{X_i = 1, X_j = 2\}$:

$D_{Xi} = \{1,2,3\}$
$D_{Xj} = \{1,2,3\}$
$D_{Xm} = \{2\}$

YES! Arc- consistency

NO! Arc- consistency

$X_i = 1$ ← $X_i \neq X_m$ → $X_m = 2$ ← $X_m \neq X_j$ → $X_j = 2$

NO!
Path- consistency

- **Path-consistent assignment** $\{X_i = 1, X_j = 3\}$:

$D_{Xi} = \{1,2,3\}$
$D_{Xj} = \{1,2,3\}$
$D_{Xm} = \{2\}$

YES! Arc- consistency

YES! Arc- consistency

$X_i = 1$ ← $X_i \neq X_m$ → $X_m = 2$ ← $X_m \neq X_j$ → $X_j = 3$

YES!
Path- consistency

Searching with Inference

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

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

Apply local consistency checks and report failure if you know that following given path is going to dead end

Searching with Inference

Two key ideas:

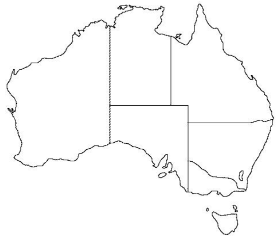
- **Forward checking**
- **Maintaining Arc Consistency**

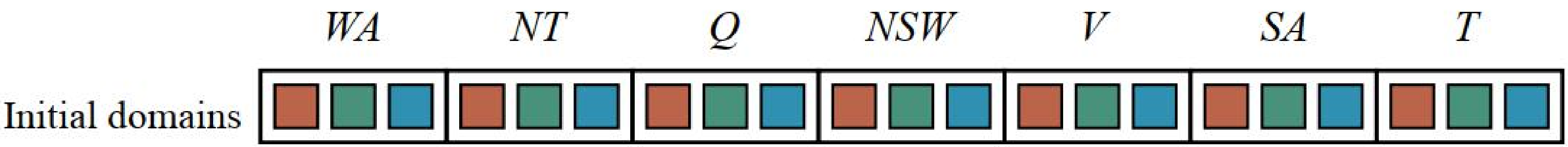
Forward Checking

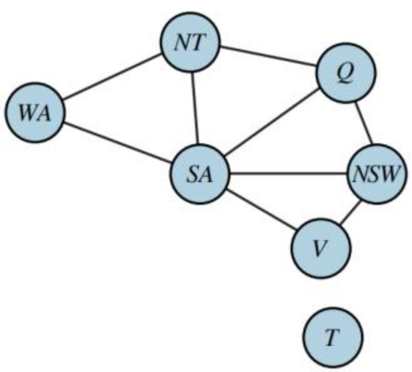
Idea:

After some value a is assigned to variable X, examine every unassigned variable Y connected to X by a constraint and delete values from Y's domain that are inconsistent with a

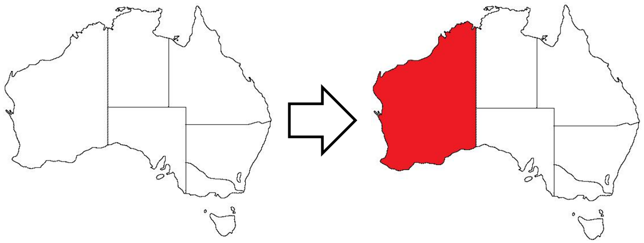
Forward Checking: Map of Australia

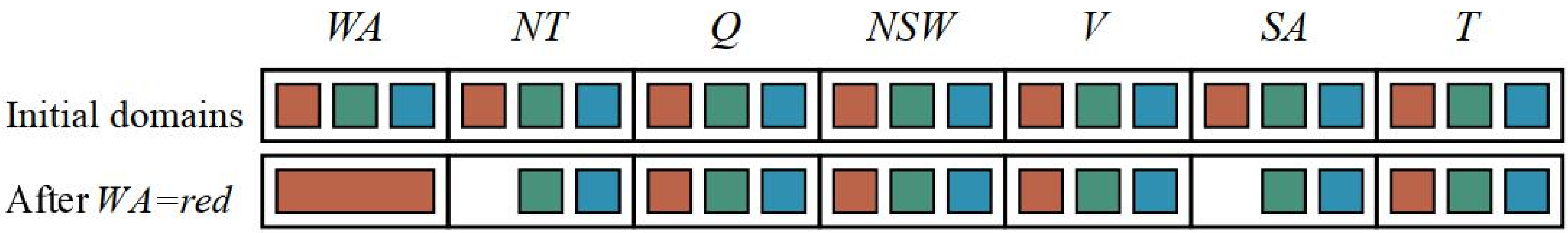


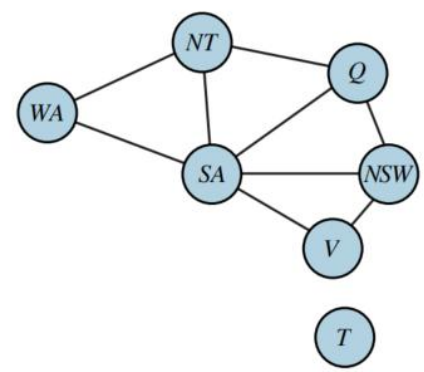




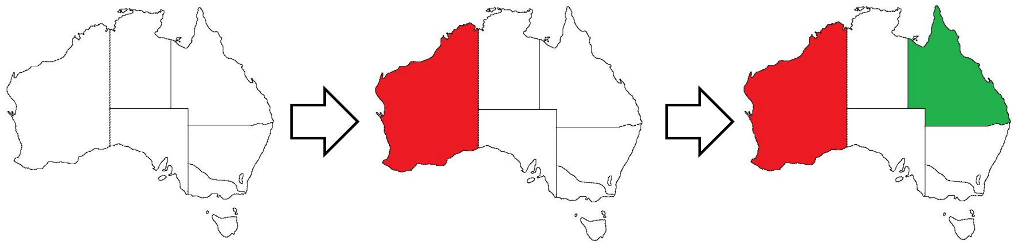
Forward Checking: Map of Australia

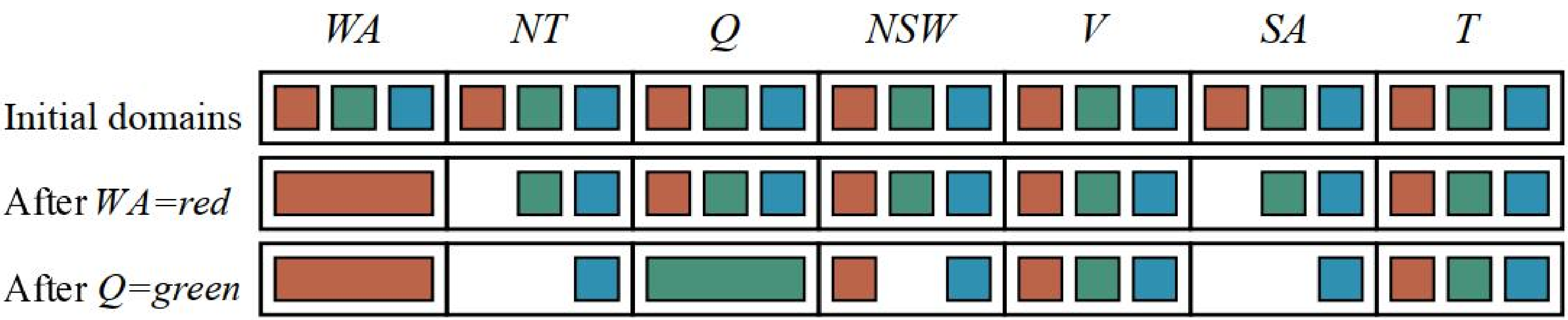


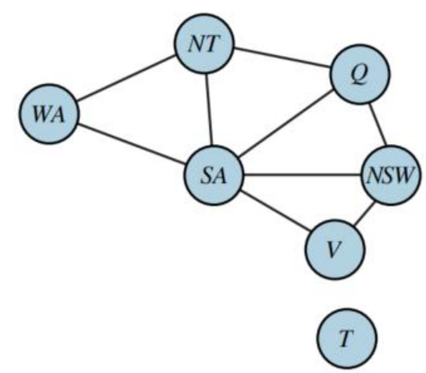




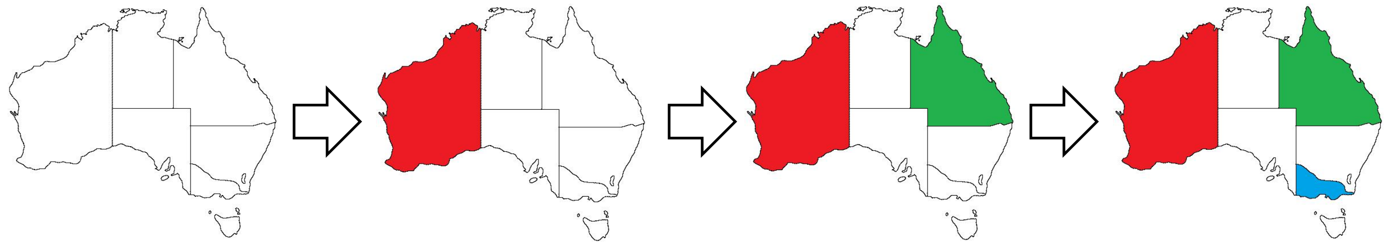
Forward Checking: Map of Australia

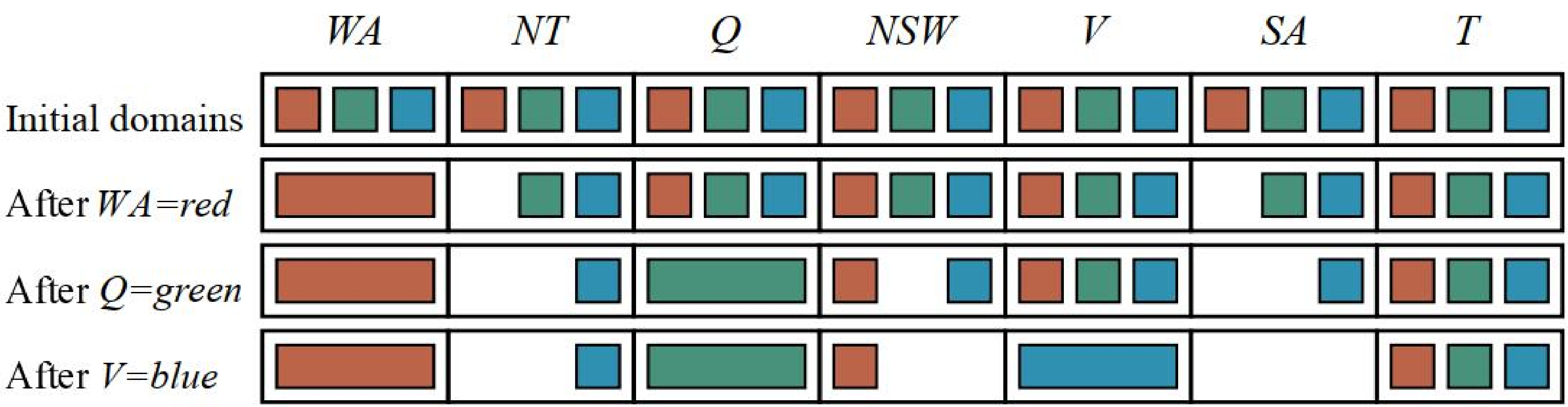


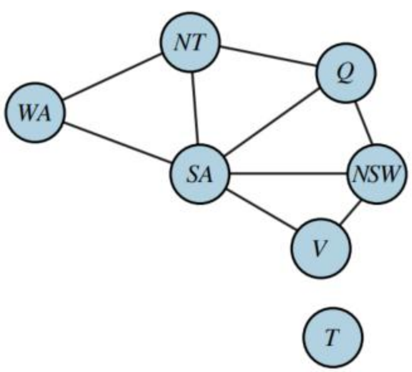




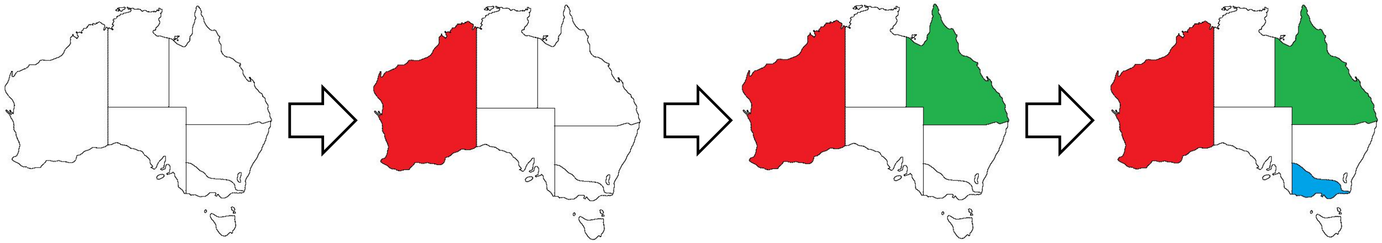
Forward Checking: Map of Australia

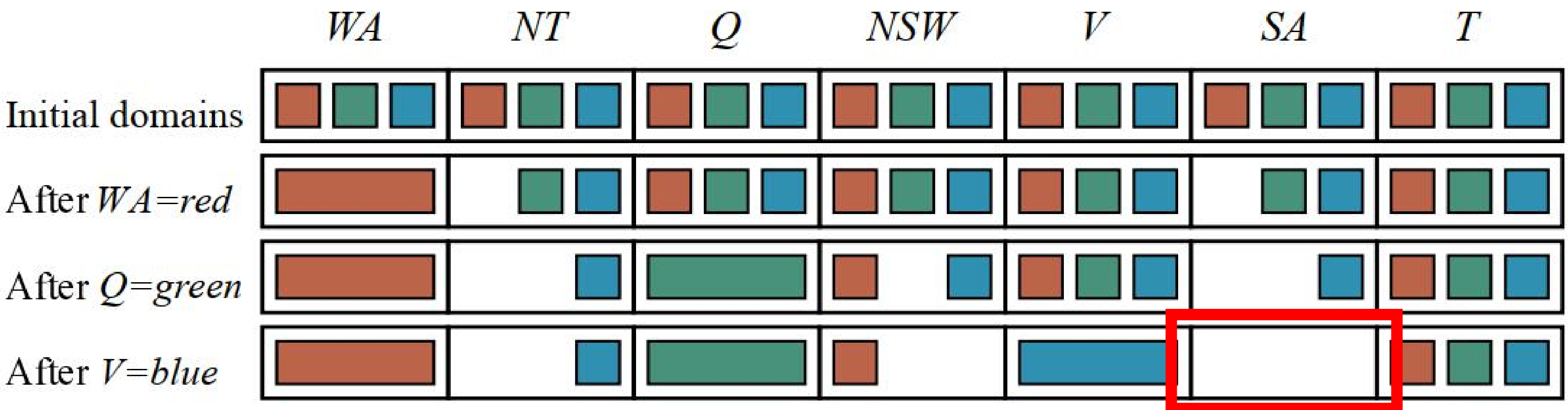


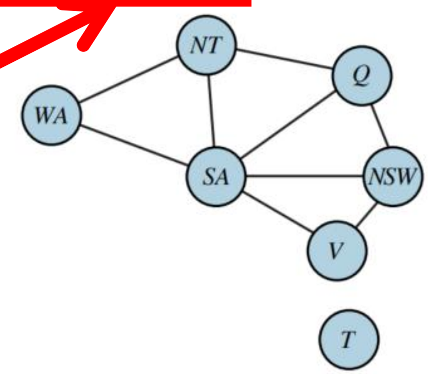




Forward Checking: Map of Australia

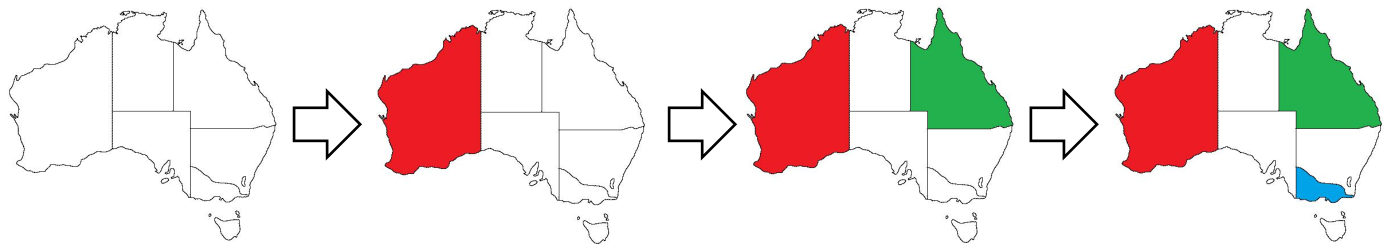


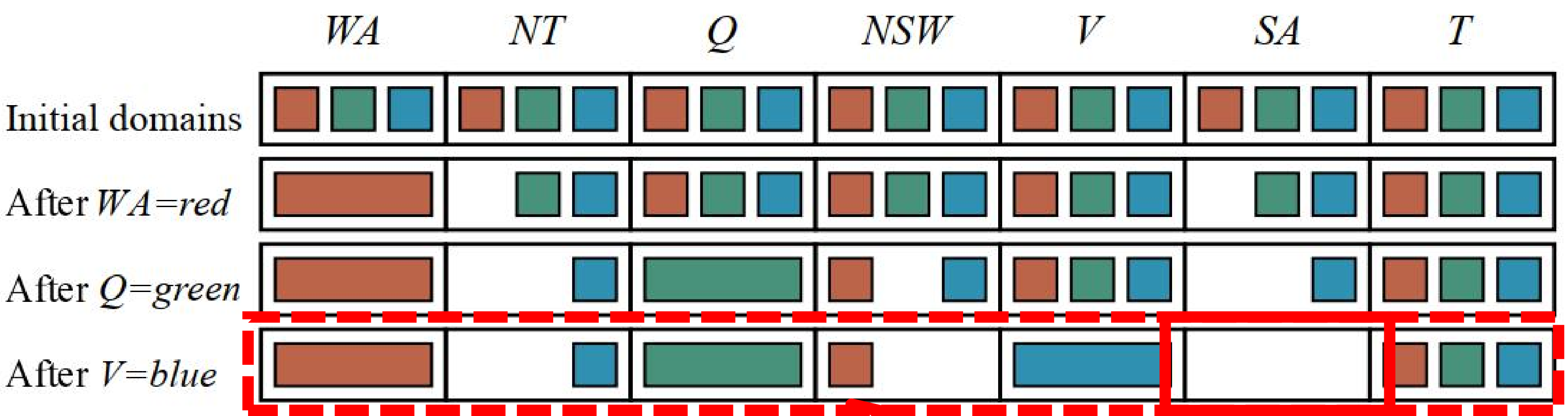


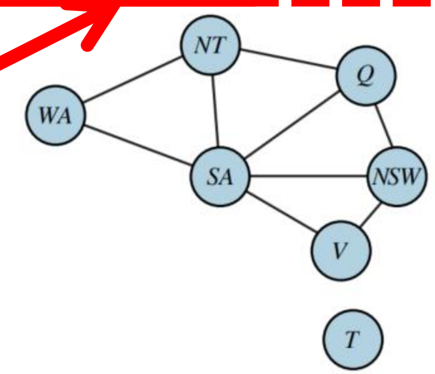


SA domain is empty!

Forward Checking: Map of Australia



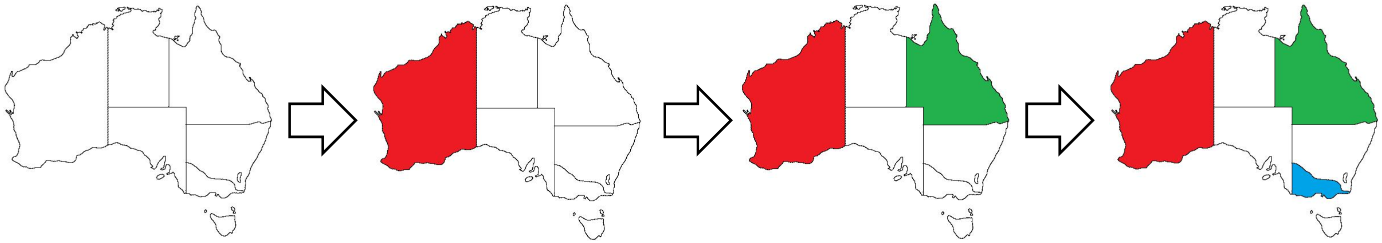


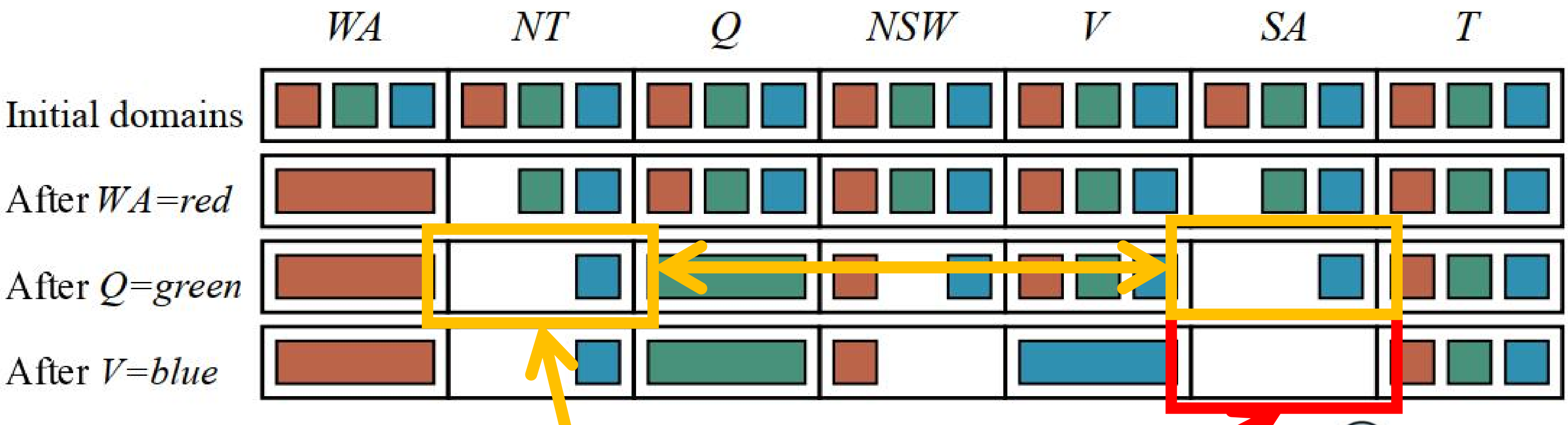


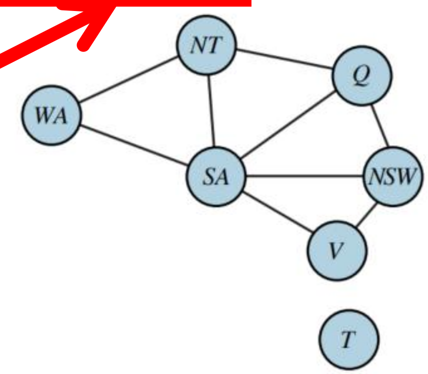
Inconsistent assignment

SA domain is empty!

Forward Checking: Map of Australia







Path inconsistency missed!

SA domain is empty!

AC-3 Algorithm: Pseudocode

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  queue $\leftarrow$ a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ POP(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

Note: treat a constraint graph edge as two directional edges: constraint $X_i \neq X_j$ corresponds to edges (X_i, X_j) and (X_j, X_i)

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

Maintaing Arc-Consistency Algorithm

Idea:

After some value is assigned to variable X_i, infer by calling AC3 algorithm, but with a reduced number of edges / arcs for its queue:

- **only (X_i, X_j) arcs for all X_j variables that:**
  - **are constrained by X_i (neighbors of X_i on the constraint graph)**
  - **have no value assigned**

MAC Algorithm Call to AC3

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise

$\quad$ *queue* ← a queue of arcs, ~~initially all the arcs in~~ *csp*

only (Xi, Xj) arcs for all Xj variables that:

- are constrained by X_i (neighbors of X_i on the constraint graph)
- have no value assigned

$\quad$ **while** *queue* is not empty **do**
$\quad\quad$ $(X_i, X_j) \leftarrow$ POP(*queue*)
$\quad\quad$ **if** REVISE(*csp*, X_i, X_j) **then**
$\quad\quad\quad$ **if** size of $D_i = 0$ **then return** *false*
$\quad\quad\quad$ **for each** X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
$\quad\quad\quad\quad$ add (X_k, X_i) to *queue*
$\quad$ **return** *true*

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

Intelligent Backtracking

- **Chronological Backtracking:**
  - **Backpropagation used it**
- **Backjumping:**
  - **maintains a conflict set for a node X: a set of assignments that are in conflict with some X domain value**
  - **backtracks to a variable assignment level where a conflict (it ruled out some potential value of X earlier)**
  - **Forward checking can help construct conflict set**

Search Problems: Summary

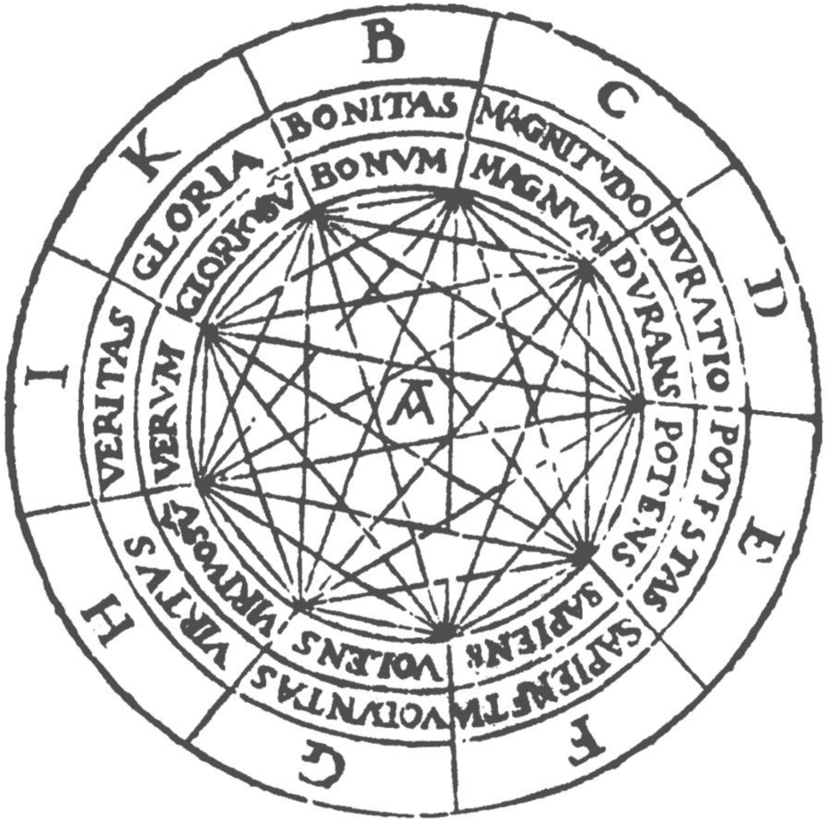
- **Initial problem analysis:**
  - **can it be represented with a state space?**
  - **what is the most useful state representation?**
  - **where, in the search tree, solution is expected? BFS or DFS?**
- **Do problem solutions need to be optimal?**
- **Do you care about time or space performance? Or both?**
- **Does your problem representation match known search algorithms?**
  - **Yes? Use it. No? See if you can make some simplifying assumptions and ask that question again**
- **Use all available knowledge about the problem to come with handy heuristics and use them to prune search tree**

Some CSP Challenges

- **What if not all constraints can be satisfied?**
  - **Hard vs. soft constraints vs. preferences**
  - **Degree of constraint satisfaction concept**
  - **Cost of violating constraints**
- **What if constraints are of different forms?**
  - **Symbolic constraints**
  - **Logical constraints**
  - **Temporal constraints**
  - **Mixed constraints**

Logical Agents and Reasoning

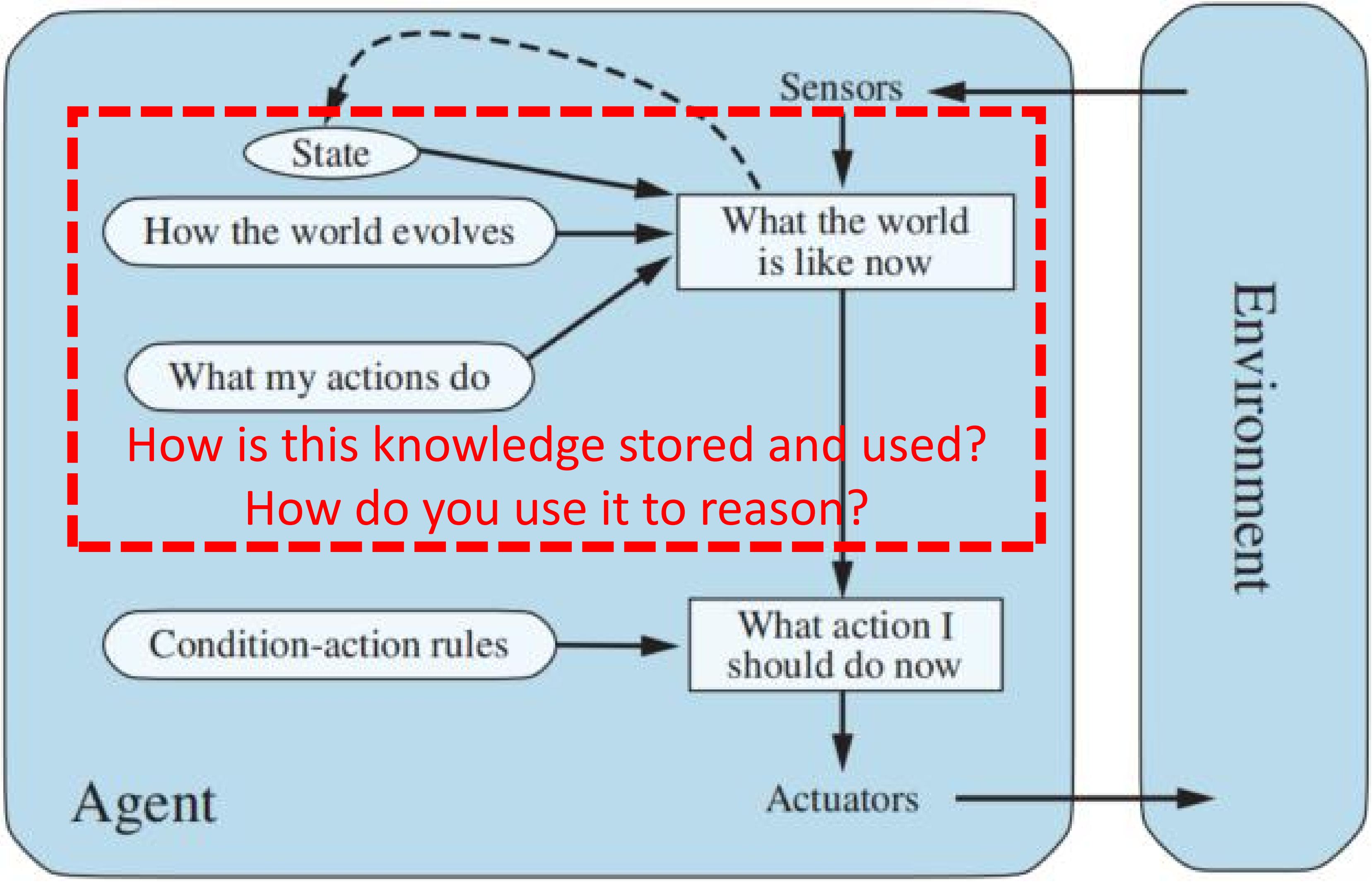
Llull’s Ars Magna (around 1305)



Catalan philosopher Ramon Llull in his book “Ars Magna”. It was an attempt to use logic to **artificially produce new knowledge by generating combinations of elemental truths** (a fixed set of preliminary ideas). Some consider it an **early step towards a “thinking machine”**.

Source: https://commons.wikimedia.org/wiki/File:Ramon_Llull_-_Ars_Magna_Fig_1.png

Knowledge-based Agent



Knowledge-based Agents

Knowledge-based agents use a process of **reasoning** over an **internal representation** of knowledge to **decide what actions** to take

Logic is one way to represent knowledge and reason:

- Propositional logic
- First-order logic

Knowledge-based Agents

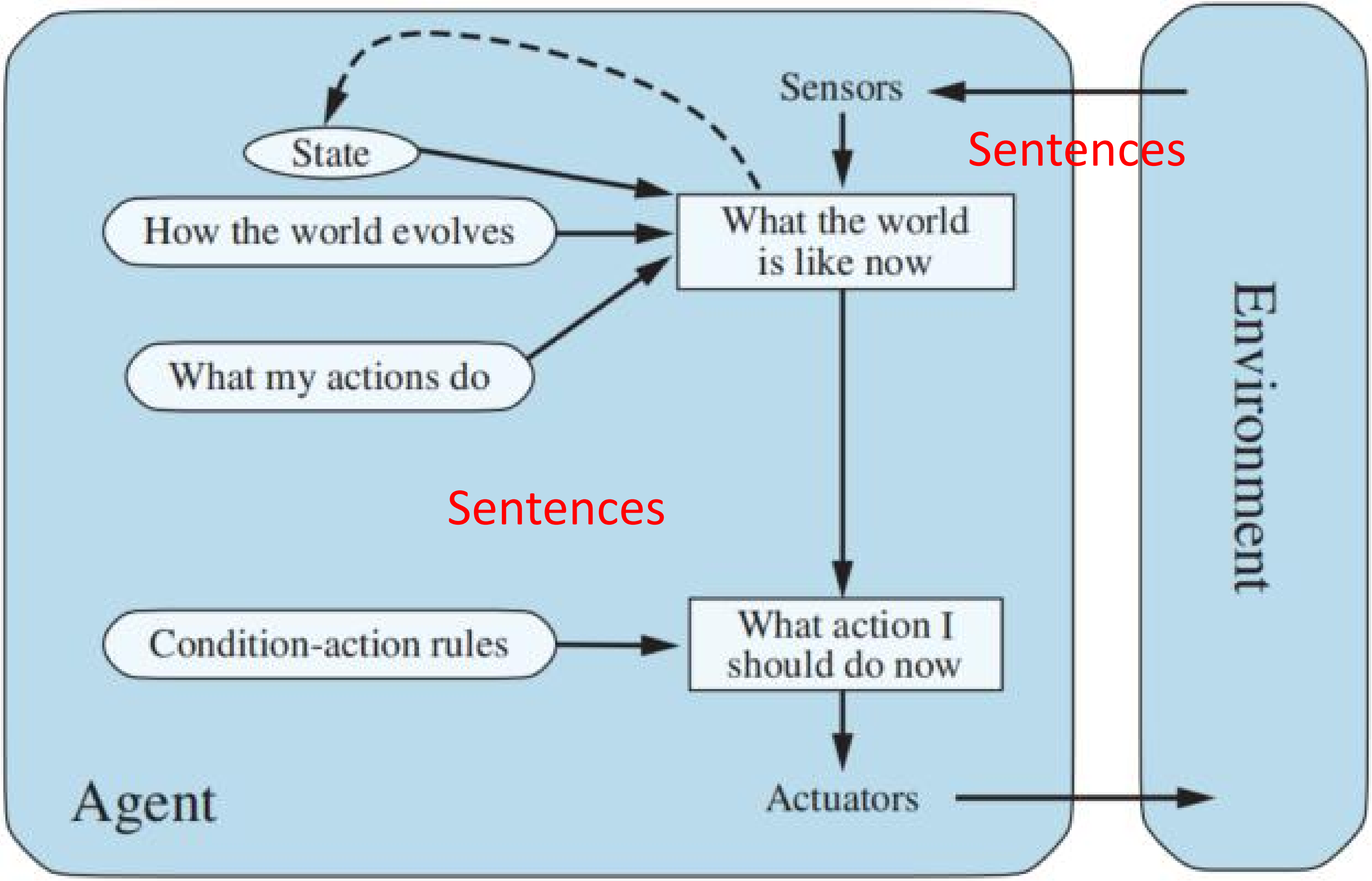
function KB-AGENT($percept$) **returns** an $action$
  persistent: KB, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE($percept$, t))
  $action \leftarrow$ ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE($action$, t))
  $t \leftarrow t + 1$
  return $action$

Knowledge-based Agents

- **Central component: Knowledge Base (KB)**
- **Knowledge Base is a set of sentences**
- **All Sentences are expressed in knowledge representation language**
- **Sentences can be:**
  - **given (axioms)**
  - **derived**
  - **used for inference**
- **KB can have background knowledge**

Knowledge-based Agent



Propositional Logic

Propositional logic, also known as sentential logic and statement logic, is the branch of logic that studies ways of joining and/or modifying entire propositions, statements or sentences to form more complicated propositions, statements or sentences, as well as the logical relationships and properties that are derived from these methods of combining or altering statements

Mathematical Symbols Refresher

Symbol	Name	Alternative symbols*	Should be read
$\neg$	Negation	$\sim$!	not
$\wedge$	Logical conjunction	$\bullet$ &	and
$\vee$	Logical discjunction	$+$ $\|\|$	or
$\Rightarrow$	Material implication	$\rightarrow$ $\supset$	implies
$\Leftrightarrow$	Material equivalence	$\leftrightarrow$ $\equiv$ iff	if and only if
$\forall$	Universal quantification		for all
$\exists$	Existential quantification		there exist
$\exists!$	Uniqueness quantification		there exist exactly one

* you can encounter it elsewhere in literature

Creating Complex Sentences

atomic sentences

p ? q

logical connective

complex sentences

r ? s

logical connective

p, q, r, s - proposition (sentence) symbols

Logical Connectives: $\neg \wedge \vee \Leftrightarrow \Rightarrow$

Negation (not)

literal (atomic sequence)

$$\neg p$$

negation

Logical conjunction (and)

conjuncts

$$p \wedge q$$

conjunction

Logical disjunction (or)

disjuncts

$$p \vee q$$

disjunction

Material implication and equivalence

premise* conclusion**

$$p \Rightarrow q$$

implication

$$p \Leftrightarrow q$$

biconditional

* also called antecedent | ** also called consequent

Logic Operator Precedence

Operator Precedence

Higher precedence

$\neg$

$\wedge$

$\vee$

$\Rightarrow \quad \Leftrightarrow$

Lower precedence

Precedence in Sentences

If in doubt: left can be rewritten as right

$\neg p \wedge q$	$((\neg p) \wedge q)$
$p \wedge \neg q$	$(p \wedge (\neg q))$
$p \wedge q \vee r$	$((p \wedge q) \vee r)$
$p \vee q \wedge r$	$(p \vee (q \wedge r)$
$p \Rightarrow q \Rightarrow r$	$(p \Rightarrow (q \Rightarrow r))$
$p \Rightarrow q \Leftrightarrow r$	$(p \Rightarrow (q \Leftrightarrow r))$

BNF (Backus-Naur Form) Grammar

$$
\begin{aligned}
Sentence &\rightarrow AtomicSentence \mid ComplexSentence \\
AtomicSentence &\rightarrow True \mid False \mid P \mid Q \mid R \mid \ldots \\
ComplexSentence &\rightarrow (\ Sentence\ ) \\
&\mid \neg\ Sentence \\
&\mid Sentence \land Sentence \\
&\mid Sentence \lor Sentence \\
&\mid Sentence \ \Rightarrow\ Sentence \\
&\mid Sentence \ \Leftrightarrow\ Sentence \\
\\
\text{OPERATOR PRECEDENCE} &: \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

Logical Connectives: Truth Table

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*