COE206 – Principles of Artificial Intelligence

Mustafa MISIR

Istinye University, Department of Computer Engineering

mustafa.misir@istinye.edu.tr

http://mustafamisir.github.io
http://memoryrlab.github.io

ARTIFICIAL INTELLIGENCE IN MEDICINE RESEARCH CENTER
ISTINYE UNIVERSITY

MEMORY

L6: Constraint Satisfaction Problems (CSPs)[1]

[1] https://en.wikipedia.org/wiki/Constraint_satisfaction_problem

Outline

- Formal Definition
- Constraint Propagation
- Backtracking Search
- Local Search
- Problem Structure

Outline

- Formal Definition
- Constraint Propagation
- Backtracking Search
- Local Search
- Problem Structure

Constraint Satisfaction Problems (CSPs)

A constraint satisfaction problem consists of three components, X, D, and C:

- X is a set of variables, $\{X_1, \ldots, X_n\}$
- D is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable
- C is a set of (hard) constraints that specify allowable combinations of values

Each domain D_i consists of a set of allowable values, $v_1, \ldots, v_k$ for variable X_i.

Each constraint C_i consists of a pair $\langle scope, rel \rangle$, where

- $scope$ is a tuple of variables that participate in the constraint and
- rel is a relation that defines the values that those variables can take on.

CSP – Variables / Domains[3]

A **discrete variable** is one whose **domain** is finite or countably infinite[2].

A **binary variable** is a discrete variable with two values in its **domain**.

- One particular case of a binary variable is a **Boolean variable**, which is a variable with **domain** $\{true, false\}$.

A **variable** whose **domain** corresponds to the real values is a **continuous variable**.

[2] https://mathworld.wolfram.com/CountablyInfinite.html

[3] https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS2.html

CSP – Constraints[4]

A **constraint** can be evaluated on any assignment that extends its **scope**.

Consider **constraint** c on S:

- **Assignment** A on S', where $S \subseteq S'$ satisfies c if A, restricted to S, is mapped to true by the **relation**.
- Otherwise, the **constraint** is violated by the **assignment**.

[4] https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS2.html

CSP – Constraints[5]

A **unary constraint** is a **constraint** on a single variable

- e.g., $B \leq 3$

A **binary constraint** is a **constraint** over a pair of variables

- e.g., $A \leq B$

In general, a k-**ary constraint** has a **scope** of size k

- e.g. A + B = C is a 3-**ary (ternary) constraint**

[5] https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS2.html – image source: https://www.hackingwithswift.com/articles/74/understanding-protocol-associated-types-and-their-constraints

CSP – Constraints[6]

Constraints are defined either by

- their **intension**, in terms of formulas
- their **extension**, listing all the assignments that are true

[6] https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS2.html

CSP – Constraints[7]

Consider a **constraint** on the possible dates for 3 activities.

- A, B, C are the **variables** that represent the date of each activity.
- The domain of each **variable** is $\{1, 2, 3, 4\}$

A **constraint** with **scope** $\{A, B, C\}$ can be described by its **intension**, using a formula of the legal assignments, e.g.

- This formula says that A is on the same date or before B, and B is before day 3, B is before C, and it cannot be that A and B are on the same date and C is on or before day 3.

$$(A \leq B) \wedge (B < 3) \wedge (B < C) \wedge \neg(A = B \wedge C \leq 3)$$

This **constraint** could instead have its relation defined its **extension**, as a table of the legal assignments:

A	*B*	*C*
2	*2*	*4*
1	*1*	*4*
1	*2*	*3*
1	*2*	*4*

[7] https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS2.html – $\wedge$ means and; $\neg$ means not

CSP – Scopes[8]

Example **constraints** and their **scopes**

- $V_2 \neq 2$ has **scope** $\{V_2\}$
- $V_1 > V_2$ has **scope** $\{V_1, V_2\}$
- $V_1 + V_2 + V_4 < 5$ has **scope** $\{V_1, V_2, V_4\}$

[8] https://www.cs.ubc.ca/~mack/CS322/lectures/3-CSP2.pdf

CSP – Relations

A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations:

- testing if a tuple is a member of the **relation**
- enumerating the members of the **relation**

e.g. if X_1 and X_2 both have the domain $\{A, B\}$, then the constraint saying the two variables must have different values can be written as

$$\langle (X1, X2), [(A, B), (B, A)] \rangle \text{ or as } \langle (X1, X2), X_1 \neq X_2] \rangle$$

CSP – Delivery Robot[9], e.g.

A delivery robot must carry out a number of delivery activities, a, b, c, d, and e.

- Each activity happens at any of times $1, 2, 3, 4$
- Let A be the **variable** representing the time that activity a will occur, and similarly for the other activities.
- The **variable domains**, which represent possible times for each of the deliveries, are $\{1, 2, 3, 4\}$

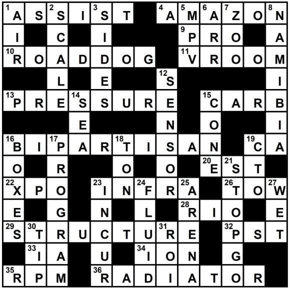
Suppose the following **constraints** must be satisfied:

$$\{(B \neq 3), (C \neq 2), (A \neq B), (B \neq C), (C < D), (A = D), (E < A), (E < B), (E < C), (E < D), (B \neq D)\}$$

[9] https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS3.html

CSP – Crossword Puzzle[10], e.g.

- X, **variables** are words that have to be filled in
- D, **domains** are English words of correct length
- C, **constraints**: words have the same letters at cells where they intersect



10 https://www.cs.ubc.ca/~mack/CS322/lectures/3-CSP2.pdf – image source: https://www.ttnews.com/articles/crossword-puzzle-solution-june-3-2019

CSP – Sudoku[11], e.g.

- X, **variables** are cells
- D, **domain** of each **variable** is 1,2,3,4,5,6,7,8,9
- C, **constraints**: rows, columns, boxes contain all different numbers

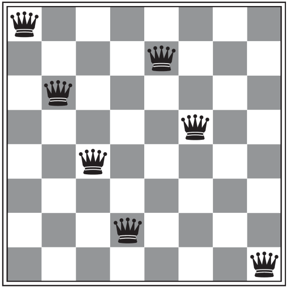
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

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

[11] https://www.cs.ubc.ca/~mack/CS322/lectures/3-CSP2.pdf

CSP – n-Queens[12], e.g.

- X, **variables** are the locations of queens on a chess board
- D, **domains** are grid coordinates
- C, **constraints**: no queen can attack another



[12] https://www.cs.ubc.ca/~mack/CS322/lectures/3-CSP2.pdf

CSP

To solve a CSP, we need to define a state space and the notion of a solution.

- Each state in a CSP is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \ldots\}$.
- An **assignment** that does not violate any constraints is called a **consistent** / **legal assignment**.
- A **complete (total) assignment** is one in which every variable is assigned.
- A **solution** to a CSP is a **consistent**, **complete assignment**.
- A **partial assignment** is one that assigns values to only some of the **variables**.
- A **possible world** is defined to be a **total assignment**; it is a function from variables into values that assigns a value to every **variable**.
  - If world w is the **assignment** $\{X_1 = v_1, X_2 = v_2, \ldots, X_k = v_k\}$, variable X_i has value v_i in world w.

CSP – Possible Worlds[13], e.g.

If there are n **variables**, each with **domain** size d, there are d^n **possible worlds**.

- e.g. for 2 **variables**, A with **domain** $\{0, 1, 2\}$ and B with **domain** $\{true, false\}$, there are 6 **possible worlds**:

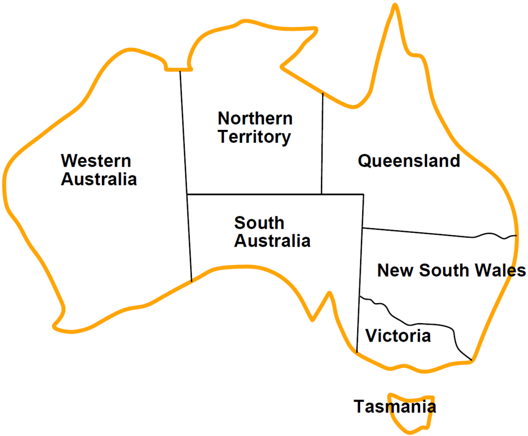
$$
\begin{aligned}
w_0 &= \{A = 0, B = true\} \\
w_1 &= \{A = 0, B = false\} \\
w_2 &= \{A = 1, B = true\} \\
w_3 &= \{A = 1, B = false\} \\
w_4 &= \{A = 2, B = true\} \\
w_5 &= \{A = 2, B = false\}
\end{aligned}
$$

A **possible world** is a model of the **constraints** – a **model** is a **possible world** that satisfies all of the **constraints**

[13] https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS1.html

CSP – Map Coloring, e.g.

Coloring each region either red, green, or blue in such a way that no neighboring regions have the same color.



CSP – Map Coloring, e.g.

Variables representing the regions:

$$X = \{WA, NT, Q, NSW, V, SA, T\}$$

The domain of each variable is the set

$$D_i = red, green, blue$$

There are 9 constraints[14]

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V,$$
$$WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$$

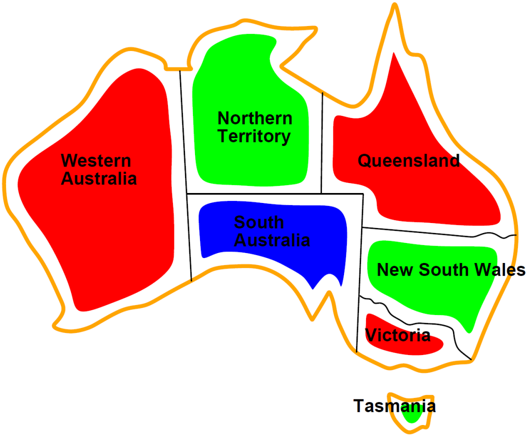
$SA \neq WA$ is a shortcut for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated as:

$$\{(red, green), (red, blue), (green, red),$$
$$(green, blue), (blue, red), (blue, green)\}$$

[14] The constraints require neighboring regions to have distinct colors and there are nine places where regions border

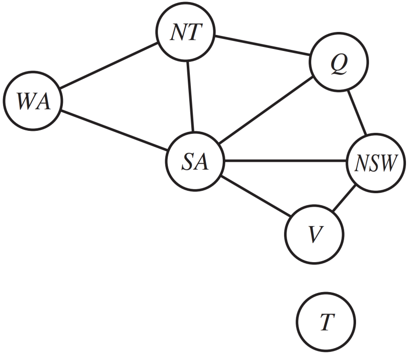
CSP – Map Coloring, e.g. Sample Solution

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



CSP – Map Coloring, e.g. Constraint Graph

Nodes are variables and links / arcs represent constraints[15]

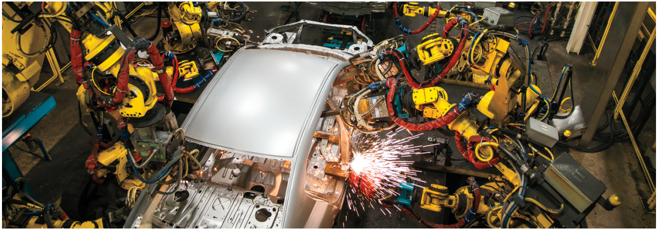


15 binary CSP as each constraint relates at most two variables

CSP – Job-Shop Scheduling, e.g. Car Assembly[16]

Problem can be defined as multiple tasks:

- Each task is a variable, where its value is the time that the task starts, expressed as an integer number of minutes
- Constraints can assert that one task must occur before another – e.g. a wheel must be installed before the wheel-cap
- Constraints can also specify that a task completion time



[16] image source: https://ferntransport.wordpress.com/about/

CSP – Job-Shop Scheduling, e.g. Car Assembly

Consisting of 15 tasks – each represented with a variable:

- install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly.

$$
\begin{aligned}
X = \{&Axle_F, Axle_B,\\
&Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB},\\
&Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB},\\
&Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB},\\
&Inspected\}
\end{aligned}
$$

The value of each variable is the start time.

CSP – Job-Shop Scheduling, e.g. Car Assembly

Precedence constraints – task T_1 must occur before task T_2, and task T_1 takes duration d_1 to complete:

$$T_1 + d_1 \leq T_2$$

The axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle:

$$Axle_F + 10 \leq Wheel_{RF} \text{ ; } Axle_F + 10 \leq Wheel_{LF}$$
$$Axle_B + 10 \leq Wheel_{RB} \text{ ; } Axle_B + 10 \leq Wheel_{LB}$$

CSP – Job-Shop Scheduling, e.g. Car Assembly

For each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$
\begin{aligned}
Wheel_{RF} + 1 \leq Nuts_{RF} \; ; \; Nuts_{RF} + 2 \leq Cap_{RF} \\
Wheel_{LF} + 1 \leq Nuts_{LF} \; ; \; Nuts_{LF} + 2 \leq Cap_{LF} \\
Wheel_{RB} + 1 \leq Nuts_{RB} \; ; \; Nuts_{RB} + 2 \leq Cap_{RB} \\
Wheel_{LB} + 1 \leq Nuts_{LB} \; ; \; Nuts_{LB} + 2 \leq Cap_{LB}
\end{aligned}
$$

With 4 workers to install wheels, but they have to share one tool that helps put the axle in place.

- **disjunctive constraint** to say that $Axle_F$ and $Axle_B$ must not overlap in time; either one comes first or the other does:

$$(Axle_F + 10 \leq Axle_B) \text{ or } (Axle_B + 10 \leq Axle_F)$$

CSP – Job-Shop Scheduling, e.g. Car Assembly

The inspection comes last and takes 3 minutes.

- For every variable except Inspect we add a constraint of the form $X + d_X \leq Inspect$.

Whole assembly should be done in 30 minutes.

- achieve that by limiting the domain of all variables:

$$Di = \{1, 2, 3, \ldots, 27\}$$

Outline

- Formal Definition
- Constraint Propagation
- Backtracking Search
- Local Search
- Problem Structure

Constraint Propagation

An algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of inference called **constraint propagation**:

- using the constraints to reduce the number of legal values for a **variable**, which in turn can reduce the legal values for another **variable**, and so on.

Constraint propagation may be interconnected with search, or it may be done as a preprocessing step, before search starts.

- Sometimes this preprocessing can solve the whole problem, so no search is required at all.

Constraint Propagation – Node Consistency

A single variable (corresponding to a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints.

- e.g. in the variant of the Australia map-coloring problem where South Australians dislike green, the variable SA starts with domain $\{red, green, blue\}$,
- can make it **node consistent** by eliminating green, leaving SA with the reduced domain $\{red, blue\}$

A network is **node-consistent** if every variable in the network is **node-consistent**.

Constraint Propagation – Arc Consistency

Simplest form of **propagation** makes each **arc consistent**.

A **variable** in a CSP is **arc-consistent** if every value in its **domain** satisfies the **variable**'s **binary constraints**.

- X_i is **arc-consistent** with respect to another **variable** X_j if for every value in the current **domain** D_i there is some value in the **domain** D_j that satisfies the **binary constraint** on the **arc** (X_i, X_j)
- A network is **arc-consistent** if every **variable** is arc consistent with every other **variable**

Pruning out possible values for the **variables** in a CSP which cannot possibly be part of a **consistent solution**

Constraint Propagation – Arc Consistency

e.g. consider the **constraint** $Y = X^2$ where the **domain** of both X and Y is the set of digits:

$$\langle (X,Y),(0,0),(1,1),(2,4),(3,9) \rangle$$

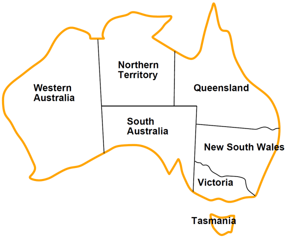
To make X **arc-consistent** with respect to Y, we reduce X's **domain** to $\{0,1,2,3\}$.

- If we also make Y **arc-consistent** with respect to X, then Y's **domain** becomes $\{0,1,4,9\}$ and the whole CSP is **arc-consistent**.

All the **variables** which cannot possibly be part of a **consistent solution** are removed!

Constraint Propagation – Arc Consistency

On the other hand, **arc consistency** can do nothing for the Australia map-coloring problem. Consider the following inequality **constraint** on (SA, WA):



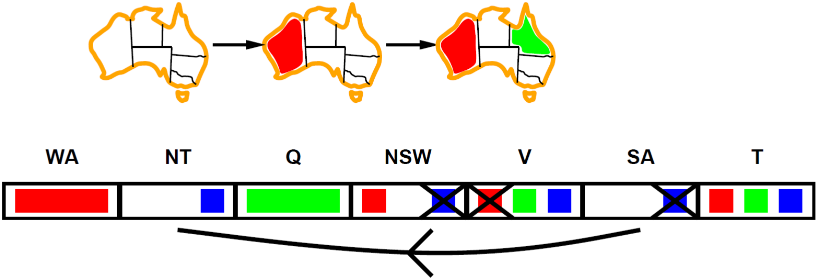
$$\{(red, green), (red, blue), (green, red),$$
$$(green, blue), (blue, red), (blue, green)\}$$

No matter what value you choose for SA (or for WA), there is a valid value for the other **variable**.

- Applying **arc consistency** has no effect on the **domains** of either **variable**.

Constraint Propagation – Arc Consistency

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



If X loses a value, neighbors of X need to be rechecked **arc consistency** which detects failure earlier than **forward checking**

- can be run as a preprocessor or after each assignment

Constraint Propagation – Arc Consistency, AC-3[17]

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  inputs: *csp*, a binary CSP with components (X, D, C)
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      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 $\leftarrow$ *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*

---

17 https://en.wikipedia.org/wiki/AC-3_algorithm

Constraint Propagation – Path Consistency

Arc consistency tightens down the **domains** (**unary constraints**) using the **arcs** (**binary constraints**).

- To make progress on problems like map coloring, we need a stronger notion of **consistency**.

Path consistency tightens the **binary constraints** by using implicit **constraints** that are inferred by looking at triples of **variables**.

Constraint Propagation – K-Consistency

Stronger forms of **propagation** can be defined with the notion of k-**consistency**.

- A CSP is k-consistent if, for any set of $k-1$ variables and for any **consistent assignment** to those **variables**, a consistent value can always be assigned to any kth variable.

$1 \rightsquigarrow 3$ **consistency**:

- 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called **node consistency**.
- 2-consistency is the same as **arc consistency**.
- For binary constraint networks, 3-consistency is the same as **path consistency**.

Constraint Propagation – Global Constraints

A **global constraint** is one involving an arbitrary number of variables (but not necessarily all variables).

- e.g. *Alldiff*: all of the variables involved in the constraint must have different values

Global constraints occur frequently in real problems and can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far.

Constraint Propagation – Global Constraints

resource (atmost) constraint in a scheduling problem,
$P_1, \ldots, P_4$ denote the numbers of personnel assigned to each task

- The **constraint** that no more than 10 personnel are assigned in total is written as $Atmost(10, P_1, P_2, P_3, P_4)$.

Domains are represented by **upper** / **lower bounds** and are managed by **bounds propagation**

- e.g. in an airline-scheduling problem, let's suppose there are two flights, F_1 and F_2, for which the planes have capacities 165 and 385, respectively.
- The initial **domains** for the numbers of passengers on each flight are then

$$D_1 = [0, 165] \text{ and } D_2 = [0, 385]$$

Constraint Propagation – Global Constraints

Now suppose we have the additional **constraint** that the two flights together must carry 420 people: $F_1 + F_2 = 420$.

- **Propagating bounds constraints**, we reduce the domains to

$$D_1 = [35, 165] \text{ and } D_2 = [255, 385]$$

A CSP is **bounds consistent** if for every variable X, and for both the **lower / upper-bound values** of X, there exists some value of Y that satisfies the **constraint** between X and Y for every **variable** Y.

Constraint Propagation, e.g. Sudoku

A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9.

- The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3 × 3 box.

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Constraint Propagation, e.g. Sudoku

A Sudoku puzzle can be considered a CSP with 81 variables, one for each square.

- The variables are $A1$ through $A9$ for the top row (left to right), down to $I1$ through $I9$ for the bottom row.
- The empty squares have the domain $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and the prefilled squares have a domain consisting of a single value.
- There are 27 different *Alldiff* constraints: one for each row, column, and box of 9 squares.

$$
\begin{gathered}
Alldiff(A1, A2, A3, A4, A5, A6, A7, A8, A9) \\
Alldiff(B1, B2, B3, B4, B5, B6, B7, B8, B9) \\
\cdots \\
Alldiff(A1, B1, C1, D1, E1, F1, G1, H1, I1) \\
Alldiff(A2, B2, C2, D2, E2, F2, G2, H2, I2) \\
\cdots \\
Alldiff(A1, A2, A3, B1, B2, B3, C1, C2, C3) \\
Alldiff(A4, A5, A6, B4, B5, B6, C4, C5, C6) \\
\cdots
\end{gathered}
$$

Outline

- Formal Definition
- Constraint Propagation
- Backtracking Search
- Local Search
- Problem Structure

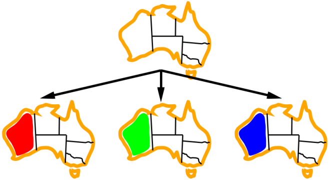
Backtracking Search

The algorithm is modeled on the recursive depth-first search – two critical elements: **variable** and **value** ordering

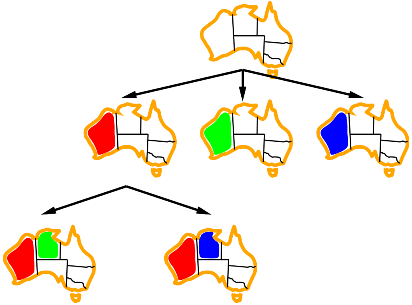
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or 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 then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
           add inferences to assignment
           result ← BACKTRACK(assignment, csp)
           if result ≠ failure then
              return result
     remove {var = value} and inferences from assignment
  return failure
```

Backtracking Search – Map Coloring, e.g.

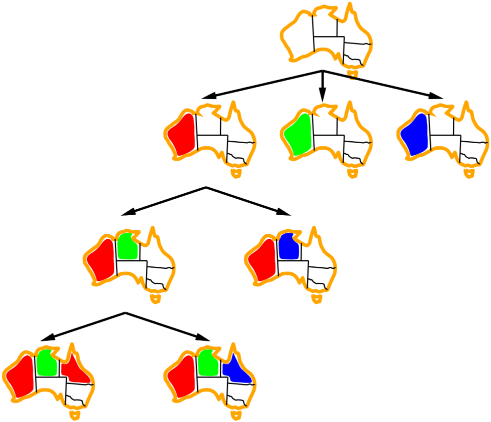
Backtracking Search – Map Coloring, e.g.



Backtracking Search – Map Coloring, e.g.



Backtracking Search – Map Coloring, e.g.



Improving Backtracking Search

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

Backtracking Search – Minimum Remaining Values (MRV)

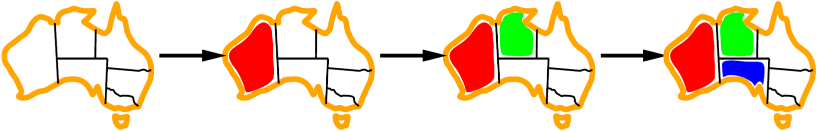
Choose the **variable** with the fewest legal values (most constrained variable) – a.k.a. fail first heuristic

- Such a **variable** is most likely to cause a failure soon
- If a variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately – avoiding pointless searches through other variables.

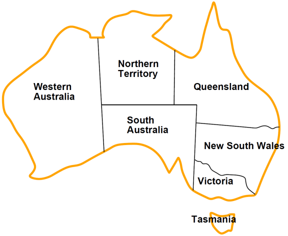
Backtracking Search – Minimum Remaining Values

Suppose we already made the assignments of red to WA and green to NT.

- There is only one possible value left for SA.



It makes sense to assign SA, rather than the one for Q (which has two possible values left)



Backtracking Search – Degree Heuristic

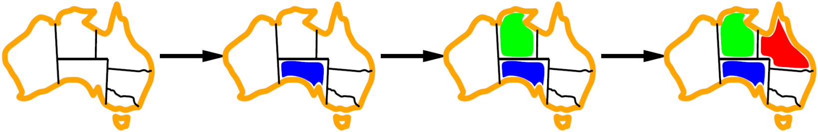
Tie-breaker among **MRV variables**

- choose the **variable** with the most **constraints** on remaining **variables**

The **degree heuristic** attempts to reduce the branching factor on future choices by selecting the **variable** that is involved in the largest number of **constraints** on other unassigned **variables**.

Backtracking Search – Degree Heuristic

The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors.



SA is the variable with highest **degree** 5 (number of neighboring cities); the other variables have **degree** 2 or 3, except for T, which has **degree** 0.

- Once SA is chosen, applying the **degree heuristic** solves the problem without any false steps–you can choose any **consistent** color at each choice point and still arrive at a solution with no **backtracking**.

Backtracking Search – Least Constraining Value

Once a **variable** has been selected, the algorithm must decide on the **order** in which to examine its **values**

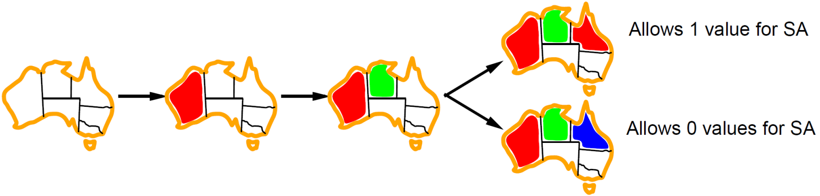
Given a **variable**, choose the least constraining value:

- the one that rules out the fewest **values** in the remaining **variables**

Backtracking Search – Least Constraining Value

Suppose that we have generated the **partial assignment** with $WA = red$ and $NT = green$ and that our next choice is for Q.

- $blue$ would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA.
- The **least constraining value** heuristic prefers red to $blue$.



In general, the heuristic is trying to leave the maximum flexibility for subsequent **variable assignments**.

Backtracking Search – Forward Checking

Inference can be powerful in the course of a search:

- every time we make a choice of a value for a **variable**, we have a brand-new opportunity to **infer** new domain reductions on the neighboring **variables**.

forward checking offers **inference**:

- Whenever a **variable** X is assigned, the **forward-checking** process establishes **arc consistency** for it: for each unassigned **variable** Y that is connected to X by a **constraint**, delete from Y's **domain** any value that is **inconsistent** with the value chosen for X.

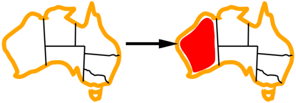
As **forward checking** only does **arc consistency inferences**, no reason to do **forward checking** if we have already done **arc consistency** as a preprocessing step.

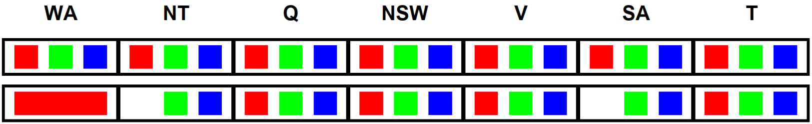
Backtracking Search – Forward Checking, e.g.

Keep track of remaining legal values for unassigned variables

- Terminate search when any variable has no legal values

Backtracking Search – Forward Checking, e.g.

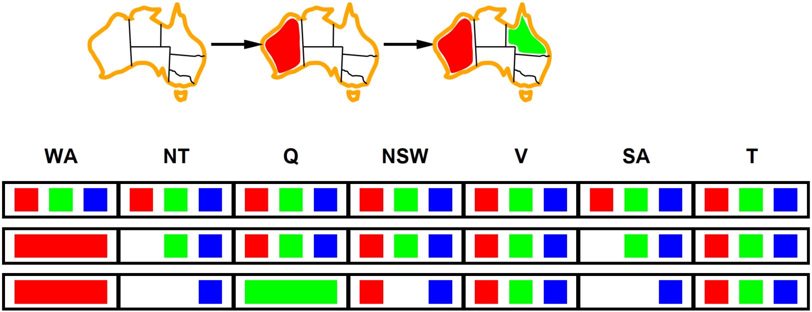




Assign $\{WA = red\}$ $\rightsquigarrow$ effects on other **variables**

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

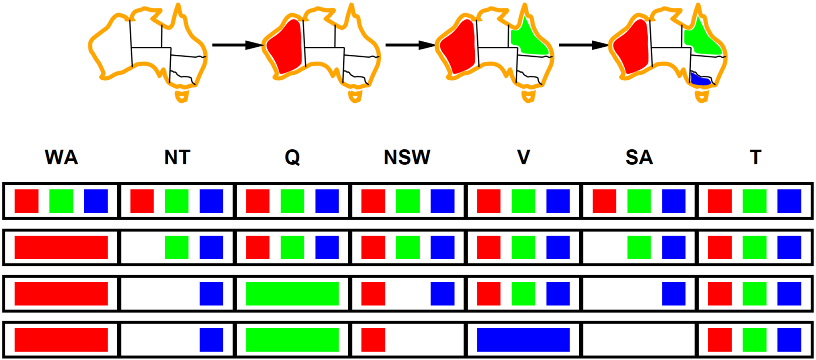
Backtracking Search – Forward Checking, e.g.



Assign $\{Q = green\}$ $\rightsquigarrow$ effects on other **variables**

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

Backtracking Search – Forward Checking, e.g.



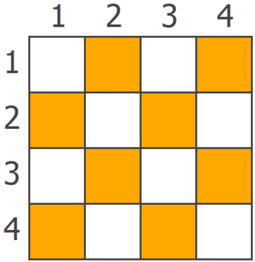
If V is assigned *blue* $\rightsquigarrow$ effects on other **variables**

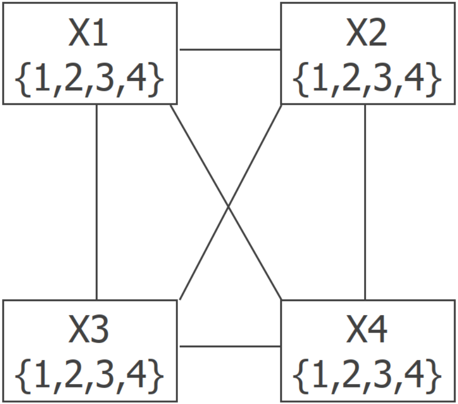
- SA is empty
- NSW can no longer be *blue*

Detected that **partial assignment** is **inconsistent** with the **constraints** and **backtracking** can occur.

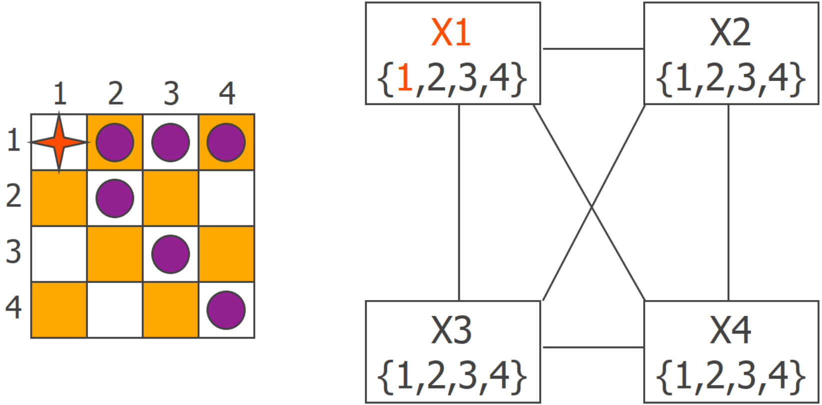
Backtracking Search – Forward Checking, e.g. 4-Queens

4 queens, $\{X_1, X_2, X_3, X_4\}$, each with the domain $\{1, 2, 3, 4\}$ referring to the column indices

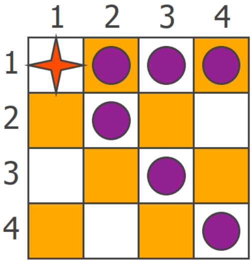


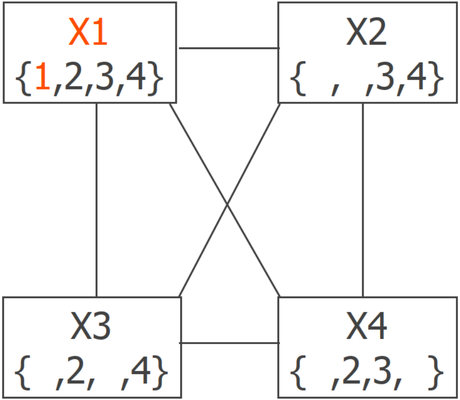


Backtracking Search – Forward Checking, e.g. 4-Queens

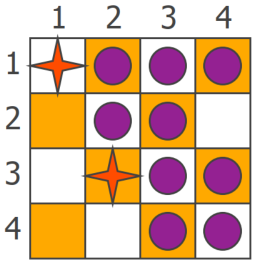


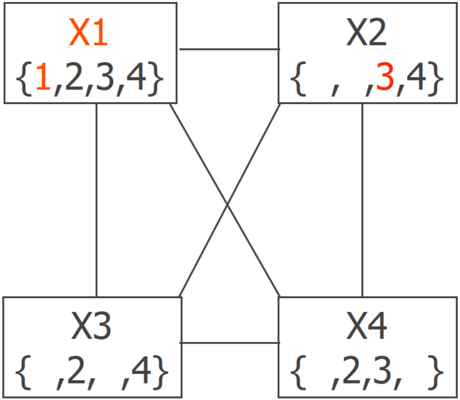
Backtracking Search – Forward Checking, e.g. 4-Queens



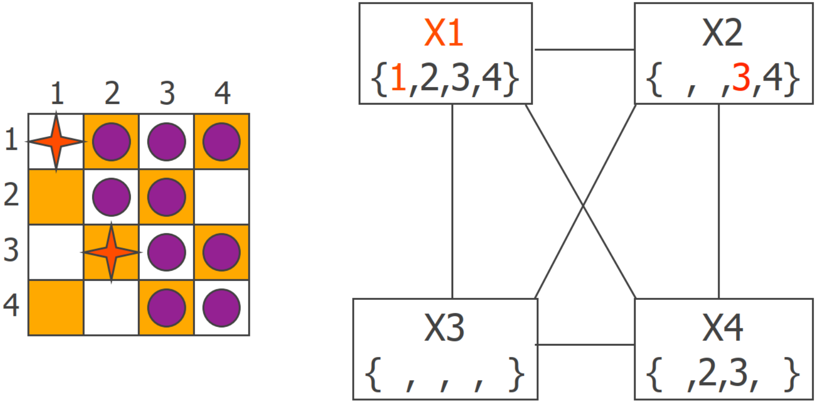


Backtracking Search – Forward Checking, e.g. 4-Queens

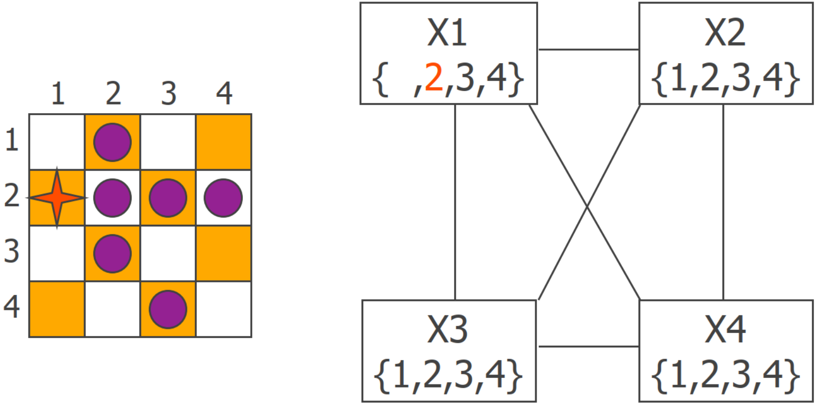




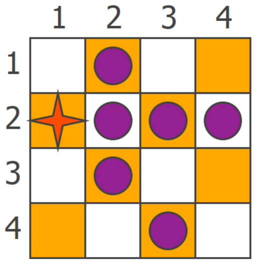
Backtracking Search – Forward Checking, e.g. 4-Queens

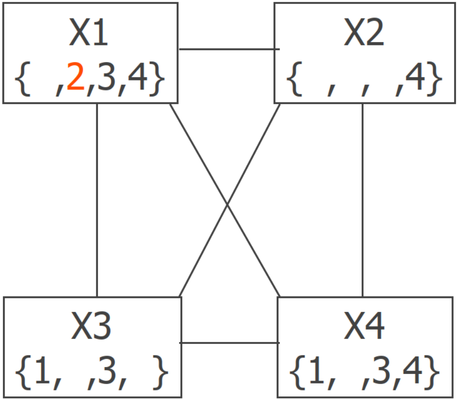


Backtracking Search – Forward Checking, e.g. 4-Queens

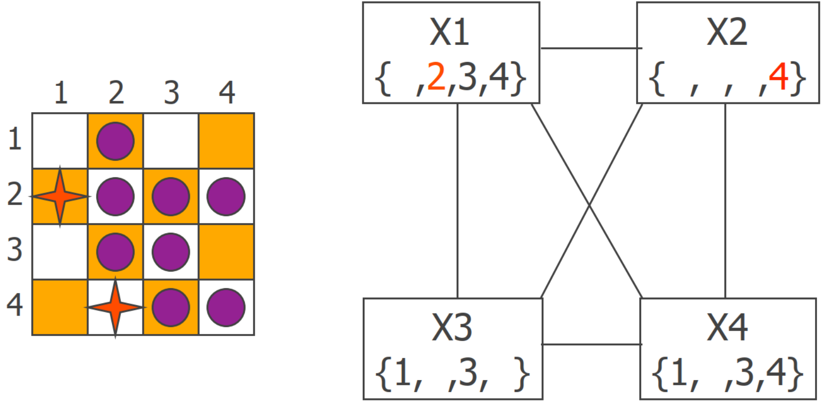


Backtracking Search – Forward Checking, e.g. 4-Queens

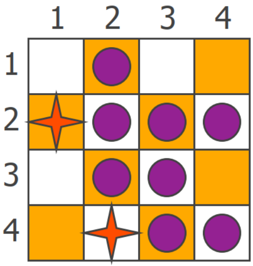


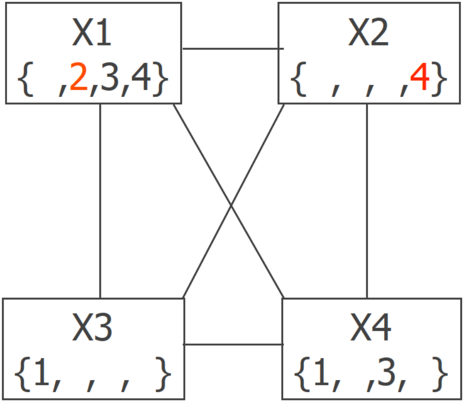


Backtracking Search – Forward Checking, e.g. 4-Queens

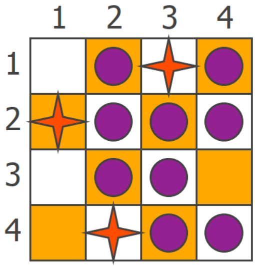


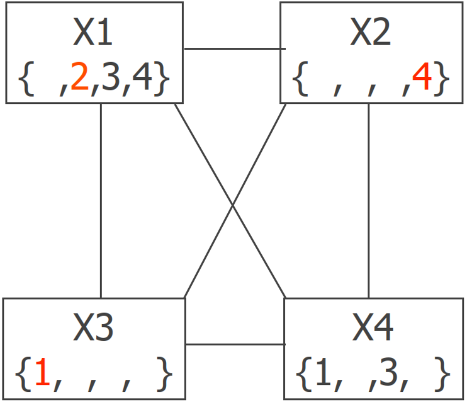
Backtracking Search – Forward Checking, e.g. 4-Queens



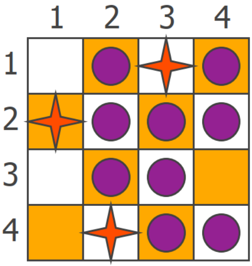


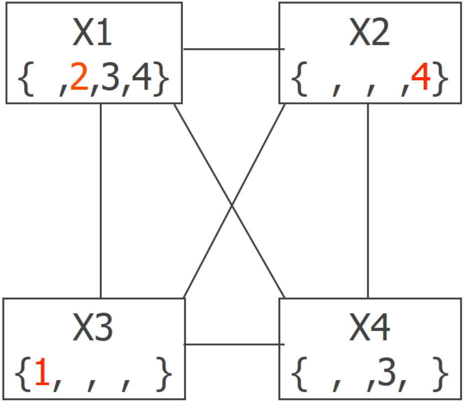
Backtracking Search – Forward Checking, e.g. 4-Queens



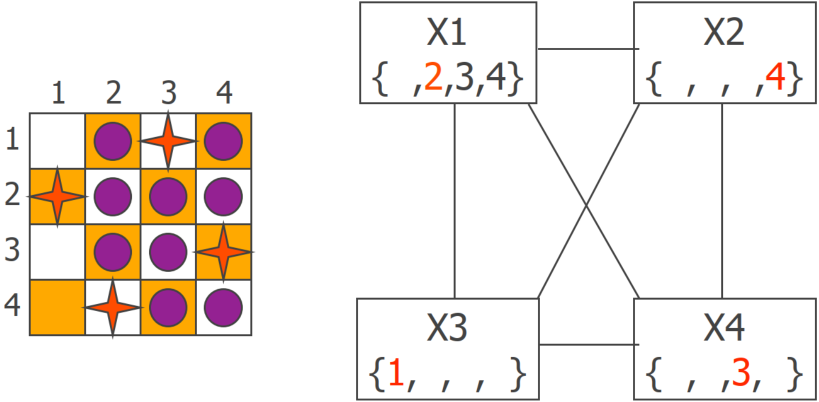


Backtracking Search – Forward Checking, e.g. 4-Queens





Backtracking Search – Forward Checking, e.g. 4-Queens



Outline

- Formal Definition
- Constraint Propagation
- Backtracking Search
- Local Search
- Problem Structure

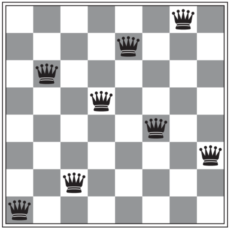
Local Search

Use a complete-state formulation:

- the initial state assigns a value to every variable, and the search changes the value of one variable at a time

e.g. in 8-queens, the initial state is a random configuration of 8 queens in 8 columns, and each step moves a single queen to a new position in its column

- Typically, the initial guess violates several constraints.



Local Search – Min-Conflicts[18], e.g. 8-Queens

In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variable – **the min-conflicts** heuristic

The function counts the number of constraints violated by a particular value, given the rest of the current assignment.

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

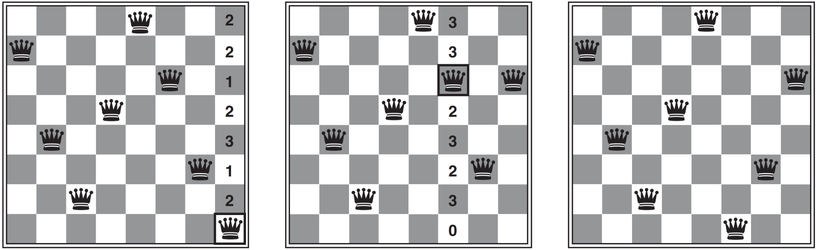
  $current \leftarrow$ an initial complete assignment for csp
  for i = 1 to max_steps **do**
    if $current$ is a solution for csp **then return** $current$
    $var \leftarrow$ a randomly chosen conflicted variable from csp.VARIABLES
    $value \leftarrow$ the value v for var that minimizes CONFLICTS(var, v, $current$, csp)
    set $var = value$ in $current$
  return $failure$

[18] the initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn.

Local Search – Min-Conflicts, e.g. 8-Queens

A two-step solution using **min-conflicts**:

- At each stage, a queen is chosen for reassignment in its column.
- The number of **conflicts** (in this case, the number of attacking queens) is shown in each square
- The algorithm moves the queen to the **min-conflicts** square, breaking ties randomly



Outline

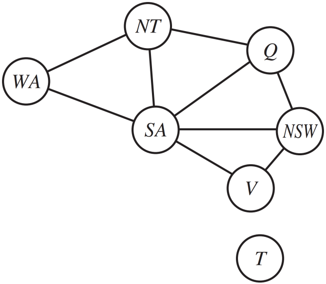
- Formal Definition
- Constraint Propagation
- Backtracking Search
- Local Search
- Problem Structure

Problem Structure, e.g.

The constraint graph for Australia indicates that *Tasmania is not connected to the mainland*.

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

- any **solution** for the mainland combined with any **solution** for Tasmania yields a **solution** for the whole map



Problem Structure[19]

Independence can be ascertained simply by finding **connected components** of the constraint graph.

- Each component corresponds to a subproblem CSP_i
- If assignment S_i is a solution of CSP_i, $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$

Consider the following:

- suppose each CSP_i has c variables from the total of n variables, where c is a constant
- there are n/c subproblems, each of which takes at most d^c work to solve, where d is the size of the domain
- the total work is $O(d^c n/c)$, which is linear in n; without the decomposition, the total work is $O(d^n)$ – exponential in n

[19] dividing a **Boolean CSP** with 80 **variables** into 4 **subproblems** reduces the worst-case solution time from the lifetime of the universe down to less than a second.

Problem Structure

Completely **independent subproblems** are practical, but rare.
Fortunately, some other graph structures are also easy to solve.

- e.g. a **constraint graph** is a **tree** when any two variables are connected by only one path

The key is a new notion of **consistency**, called **directed arc consistency** (DAC).

- A CSP is defined to be **directed arc-consistent** under an ordering of **variables** $X_1, X_2, \ldots, X_n$ if and only if every X_i is **arc-consistent** with each X_j for $j > i$

Problem Structure — DAC

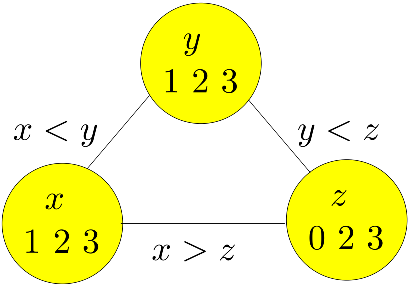
procedure $\texttt{DAC}(X, D, C)$
  for each $i := n - 1$ **downto** 1 **do**
    for each c_{ij} s.t. $x_i \prec x_j$ **do** $\texttt{Revise}(i, j)$
endprocedure

- Only one pass is required
- Once x_i is made **arc-consistent** with respect to $x_i \prec x_j$, removing values from x_i such that the **arc-consistency** of x_i wrt. x_j is not destroyed

Problem Structure — DAC, e.g.[20]

Consider a CSP with 3 **variables** in this order: $x \prec y \prec z$

- **domains** $D_x = D_y = \{1, 2, 3\}$ and $D_z = \{0, 2, 3\}$
- **constraints** C: $x < y$, $y < z$, $x > z$

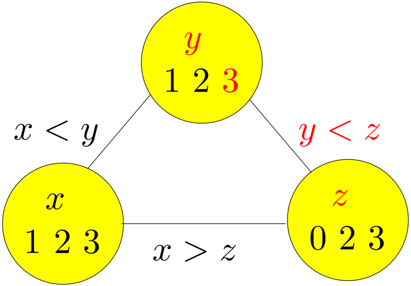


[20] https://www.cs.upc.edu/~erodri/webpage/cps/theory/cp/local-consistency/slides.pdf

Problem Structure — DAC, e.g.

Consider a CSP with 3 **variables** in this order: $x \prec y \prec z$

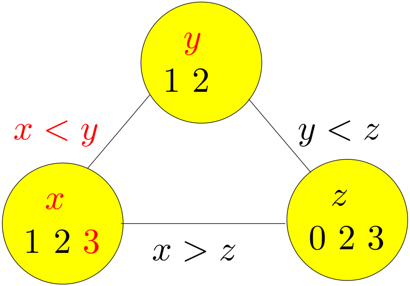
- **domains** $D_x = D_y = \{1, 2, 3\}$ and $D_z = \{0, 2, 3\}$
- **constraints** C: $x < y$, $y < z$, $x > z$



Problem Structure — DAC, e.g.

Consider a CSP with 3 **variables** in this order: $x \prec y \prec z$

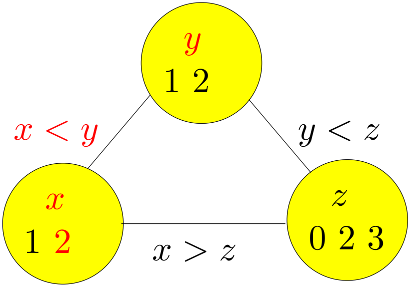
- **domains** $D_x = D_y = \{1, 2, 3\}$ and $D_z = \{0, 2, 3\}$
- **constraints** C: $x < y$, $y < z$, $x > z$



Problem Structure — DAC, e.g.

Consider a CSP with 3 **variables** in this order: $x \prec y \prec z$

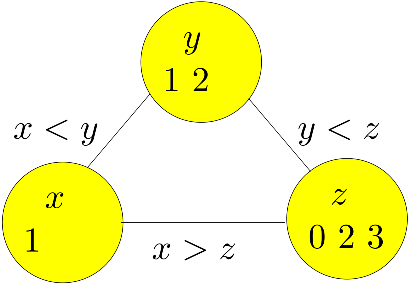
- **domains** $D_x = D_y = \{1, 2, 3\}$ and $D_z = \{0, 2, 3\}$
- **constraints** C: $x < y$, $y < z$, $x > z$



Problem Structure — DAC, e.g.

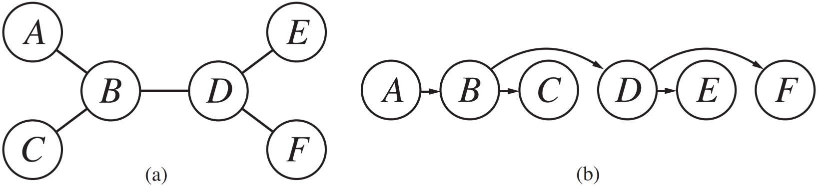
Consider a CSP with 3 **variables** in this order: $x \prec y \prec z$

- **domains** $D_x = D_y = \{1, 2, 3\}$ and $D_z = \{0, 2, 3\}$
- **constraints** C: $x < y$, $y < z$, $x > z$



Problem Structure

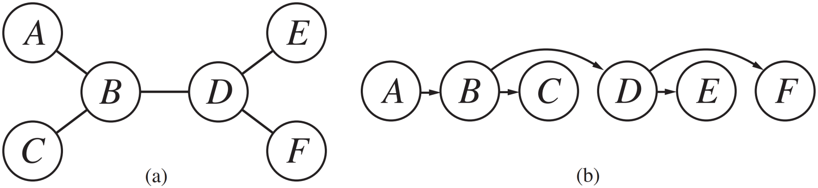
To solve a **tree-structured CSP**, first pick any **variable** to be the root of the tree, and choose an ordering of the **variables** such that each **variable** appears after its parent in the **tree** – called a **topological sort** of the variables.



(a) The **constraint graph** of a **tree-structured CSP**

(b) A linear ordering of the **variables consistent** with the **tree** with A as the root – a **topological sort**

Problem Structure



Any tree with n nodes has $n-1$ arcs, so make this **graph directed arc-consistent** in $O(n)$ steps, each of which must compare up to d possible **domain values** for two **variables**, for a total time of $O(nd^2)$.

- Once we have a **directed arc-consistent graph**, just down the list of **variables** and choose any remaining value.
- Since each link from a parent to its child is **arc consistent**, for any value we choose for the parent, there will be a valid value left to choose for the child - no **backtracking**; move linearly through the **variables** – the Tree CSP Solver

Problem Structure

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or failure
  inputs: *csp*, a CSP with components X, D, C

  $n \leftarrow$ number of variables in X
  $assignment \leftarrow$ an empty assignment
  $root \leftarrow$ any variable in X
  $X \leftarrow$ TOPOLOGICALSORT(X, $root$)
  for j = n **down to** 2 **do**
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent **then return** *failure*
  for i = 1 **to** n **do**
    $assignment[X_i] \leftarrow$ any consistent value from D_i
    if there is no consistent value **then return** *failure*
  return *assignment*