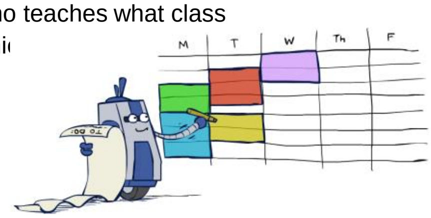
Constraint Satisfaction Problems

- Constraint Satisfaction problems are a class of problems where variables need to be assigned values while satisfying some conditions.
- Constraint satisfaction problems (CSPs):
  - Set of variables $\{X_1 , X_2 , \ldots , X_n\}$
  - Set of Domains one for each variable $\{D_1 , D_2 , \ldots , D_n\}$
  - Set of constraints C
- Allows useful general-purpose algorithms with more power than standard search algorithms
  - Can be solved by general search algorithms but inefficient

Real-World CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., whi

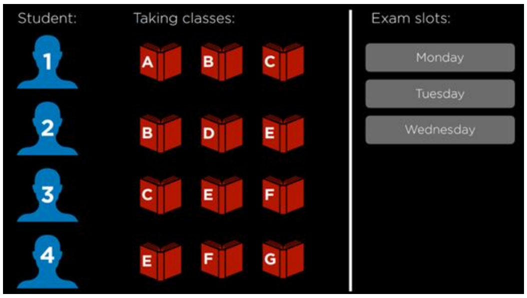


- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!

- Many real-world problems involve real-valued variables…

Scheduling

- Each of students 1-4 is taking three courses from A, B, …, G. Each course needs to have an exam, and the possible days for exams are Monday, Tuesday, and Wednesday. However, the same student can't have two exams on the same day.
  - the variables are the courses,
  - the domain is the days,
  - constraints are which courses can't be scheduled to have an exam on the same day because the same student is taking them.



Constraint Satisfaction

Constraint Satisfaction problems are a class of problems where variables need to be assigned values while satisfying some conditions.

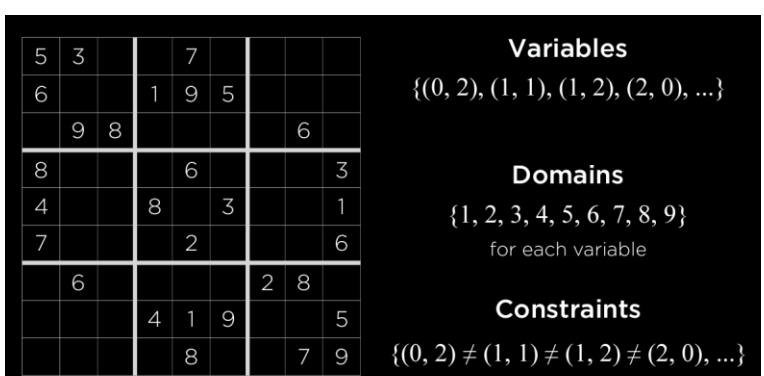
Constraints satisfaction problems have the following properties:

- Set of variables $(x_1, x_2, \ldots, x_n)$
- Set of domains for each variable $\{D_1, D_2, \ldots, D_n\}$
- Set of constraints C

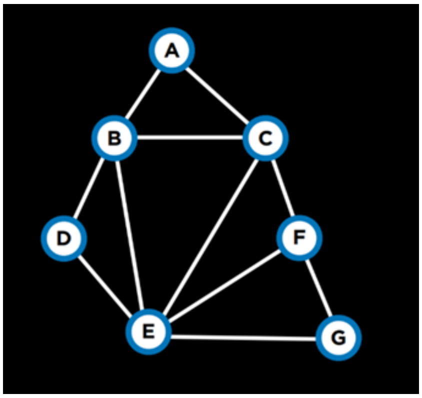
Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (reduces domains),
    - e.g.: Course A exam must be on a Monday
  - Binary constraints involve pairs of variables, e.g.:
    - e.g.: Student A takes courses A and B
  - Higher-order constraints involve 3 or more variables:
    e.g., Sudoku
- Preferences (**Soft Constraints**); Requirements (**Hard Constraints)**:
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems

Example: Sudoku



Constraint Graph

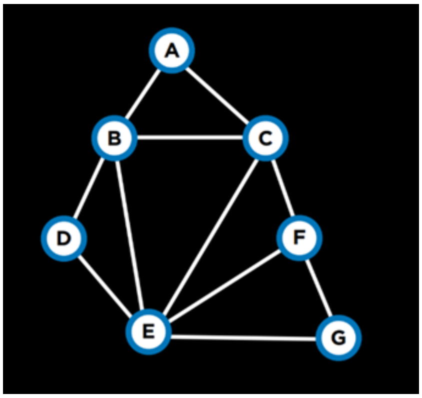


Variables: {A,B,C,D,F,G,}

Domains: {Monday, Tuesday, Wednesday} for each variable

Constraints: {A≠B, A≠C, B≠C, B≠D, B≠E, C≠E, C≠F, D≠E, E≠F, E≠G, F≠G}

Examples: Unary constraint



unary constraint
{A ≠ Monday}

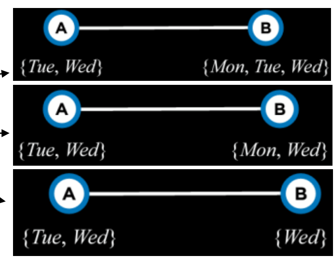
binary constraint
{A ≠ B}

Node consistency: Reduce Domains

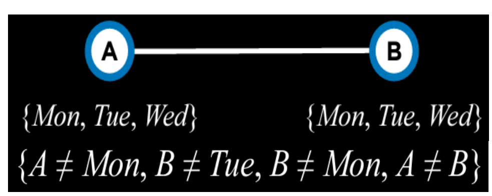
- node consistency when all the values in a variable's domain satisfy the variable's unary constraints

{A ≠ Mon,
B ≠ Tue,
B ≠ Mon,

A ≠ B}



Arc Consistency



- **Arc consistency** is when all the values in a variable's domain satisfy the variable's binary constraints (note "arc" to refer to an "edge").
- **To make X arc-consistent with respect to Y, remove elements from X's domain until every choice for X has a possible choice for Y.**

AC – remove Weds from A

RETURN TO THIS LATER

Constraint Satisfaction Problems as Search

Constraints satisfaction problems have the following properties:

- Set of variables (x_1, x_2, …, x_n)
- Set of domains for each variable {D_1, D_2, …, D_n}
- Set of constraints C

Standard Search Formulation

- **Initial state:** empty assignment (all variables don't have any values assigned to them).
- **Actions:** add a {variable = value} to assignment; that is, give some variable a value.
- **Transition model**: shows how adding the assignment changes the assignment. There is not much depth to this: the transition model returns the state that includes the assignment following the latest action.
- **Goal test:** check if all variables are assigned a value and all constraints are satisfied.
- **Path cost function**: all paths have the same cost. As we mentioned earlier, as opposed to typical search problems, optimization problems care about the solution and not the route to the solution.

DFS and BFS are very inefficient!

Example: Map Coloring, 3-color Australia map

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{red, green, blue\}$
- Constraints: adjacent regions must have different colors

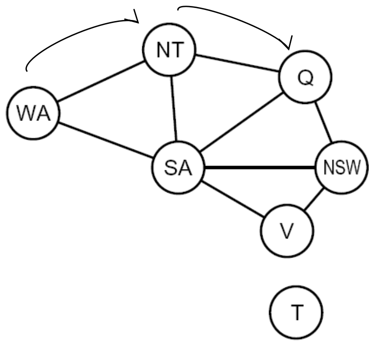
  Implicit: $WA \neq NT$

  Explicit: $(WA, NT) \in \{(red, green), (red, blue), \ldots\}$
- Solutions are assignments satisfying all constraints, e.g.:

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

Western Australia
Northern Territory
Queensland
New South Wales
Victoria
South Australia
Tasmania*

Constraint Graphs

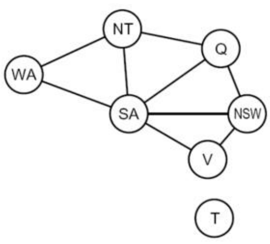


Western Australia
Northern Territory
Queensland
New South Wales
Victoria
South Australia
Tasmania*

Exercise:

- What will DFS and BFS do on the graph starting ordering the states: WA: Western Australia, NT Northern Territory, …
- Three colors: Red, Green, Blue

Start: No colors



WA: Western Australia

NT: Northern Territory

Q: Queensland

NSW: New South Wales
V: Victoria

SA: South Australia

Backtracking Search

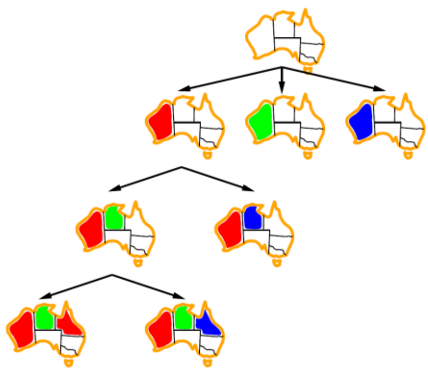
- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering -> better branching factor!
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - “Incremental goal test”
- Depth-first search with these two improvements is called *backtracking search* (not the best name)
- Can solve n-queens for n ≈ 25

Backtracking Search for CSP

```
function Backtrack(csp)
        return Rec_Backtrack({}, csp)
function Rec_Backtrack(assignment, csp)
if assignment complete:
    return assignment
var = Select-Unassigned-Var(assignment, csp)
for value in Domain-Values(var, assignment, csp):
    if value consistent with assignment:
        add {var = value} to assignment
        result = Rec_Backtrack(assignment, csp)
        if result ≠ failure:
            return result
        remove {var = value} from assignment
return failure
```

- Backtracking = DFS + variable-ordering + fail-on-violation

Backtracking Example



Improving Backtracking

- General-purpose ideas give huge gains in speed… but it's all still NP-hard
- Ordering:
  - Which variable should be assigned next? (MRV)
  - In what order should its values be tried? (LCV)
- Filtering: Can we detect inevitable failure early?
  - When assigning values to variables, check adjacent nodes and reduce domains
- Structure: Can we exploit the problem structure?

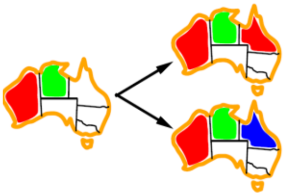
Variable Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?
- Also called "**most constrained variable**"
- "Fail-fast" ordering

Value Ordering: Least Constraining Value

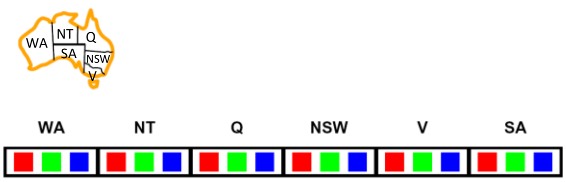
- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)



- Why least rather than most?
- Combining these ordering ideas makes 1000 queens feasible

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



[Demo: coloring -- forward checking]

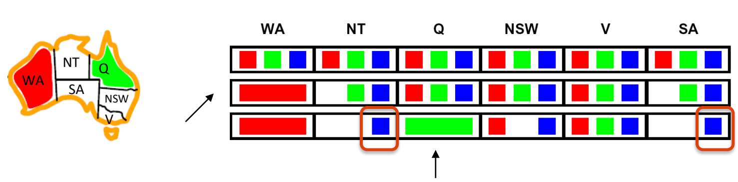
CAL POLY
SAN LUIS OBISPO

Computer Science Department

23

Filtering: Constraint Propagation

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



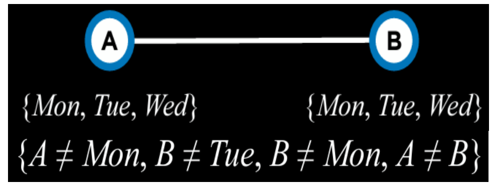
- Note: NT and SA cannot both be blue!
- Why didn't we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

CAL POLY
SAN LUIS OBISPO

Computer Science Department

24

Recall:
Arc Consistency



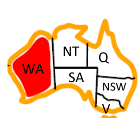
- **<u>Arc consistency</u>** is when all the values in a variable's domain satisfy the variable's binary constraints (note "arc" to refer to an "edge").
- **To make X <u>arc-consistent</u> with respect to Y, remove elements from X's domain until every choice for X has a possible choice for Y.**

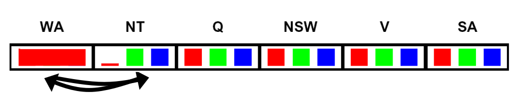
AC – remove Weds from A

RETURN TO THIS LATER

Consistency of A Single Arc

- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint





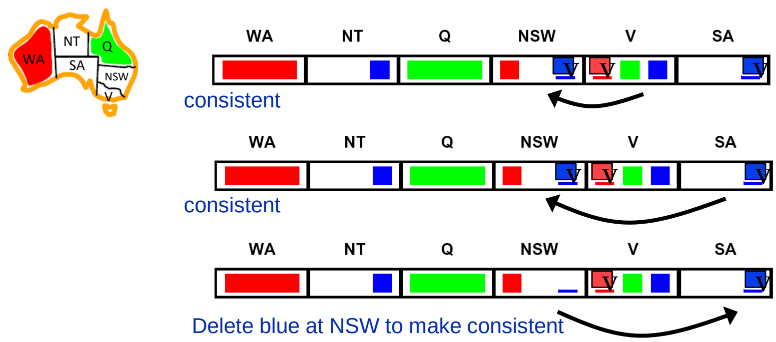
Delete from the tail!

Forward checking?
Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP

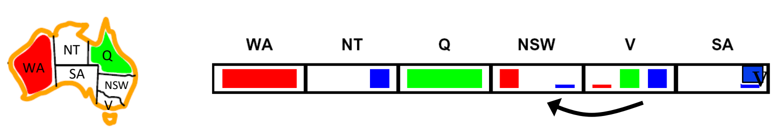
Remember: Delete from the tail!

- A simple form of propagation makes sure all arcs are consistent:



Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

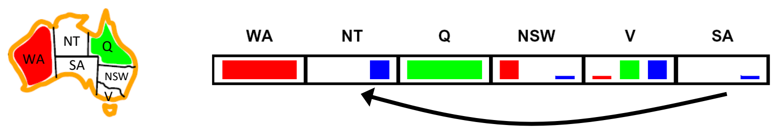


- But blue has been removed from NSW so we need to relook V →NSW
  Now red must be removed from V since it would be inconsistent with red in NSW

Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:

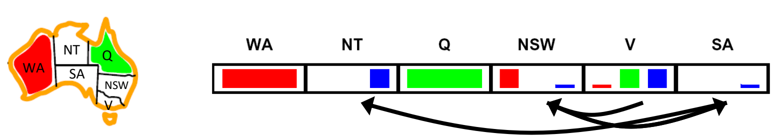


- But now SA and NT are inconsistent and have to delete the blue at SA and now there is an empty domain, so now solution is possible
  Now need to backtrack

Remember: Delete from the tail!

Arc Consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:



- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

Arc Consistency: Revise

```
function REVISE(csp, X, Y):
    revised = false
    for x in X.domain:
    if no y in Y.domain satisfies constraint for (X, Y):
        delete x from X.domain
        revised = true
    return revised
```


Arc Consistency: AC-3

```
function AC-3(csp):
    queue = all arcs in csp
    while queue non-empty:
    (X, Y) = DEQUEUE(queue)
    if REVISE(csp, X, Y):
        if size of X.domain == 0:
            return false
        for each Z in X.neighbors - {Y}:
            ENQUEUE(queue, (Z, X))
    return true
```

Maintaining Arc Consistency: AC-3

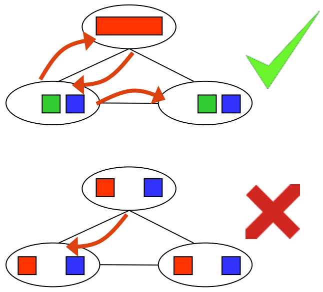
- AC-3: Algorithm for enforcing arc-consistency every time we make a new assignment
- When we make a new assignment to X, calls AC-3, starting with a queue of all arcs (Y, X) where Y is a neighbor of X

- Enforce arc-consistency after every new assignment of the backtracking search.
  - Specifically, after we make a new assignment to X, we will call the AC-3 algorithm
  - Start it with a queue of all arcs (*Y*,*X*) where Y is a neighbor of X (and not a queue of all arcs in the problem).
  - Every time a value is removed from the Domain of a variable, add node to the queue
  - When backtracking the domains must be restored to their previous values

Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

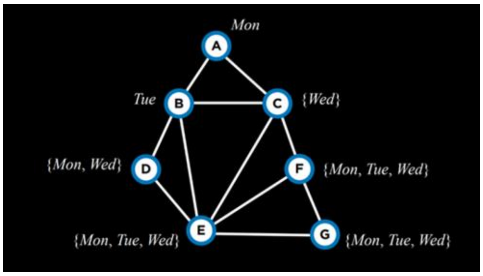
- Arc consistency still runs inside a backtracking search!



Stop

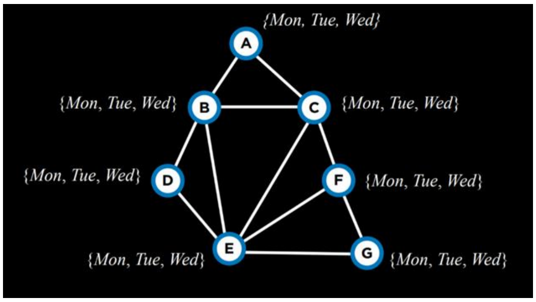
Choosing the variable to assigned next: Heuristics-1

- **Minimum Remaining Values (MRV)**
  - The idea here is that if a variable's domain was constricted by inference, and now it has only one value left (or even if it's two values), then making this assignment might reduce the number of backtracks we need to do later. That is, if this assignment leads to failure, fail soon and not backtrack later!



Choosing the variable to assigned next: Heuristics-2

- **Max Degree heuristic**
  - Recall that the degree of a node (variable) is how many arcs connect a variable to other variables.
  - Choosing the variable with the highest degree, with one assignment, we constrain multiple other variables, speeding the algorithm's process. (e.g start at node E)

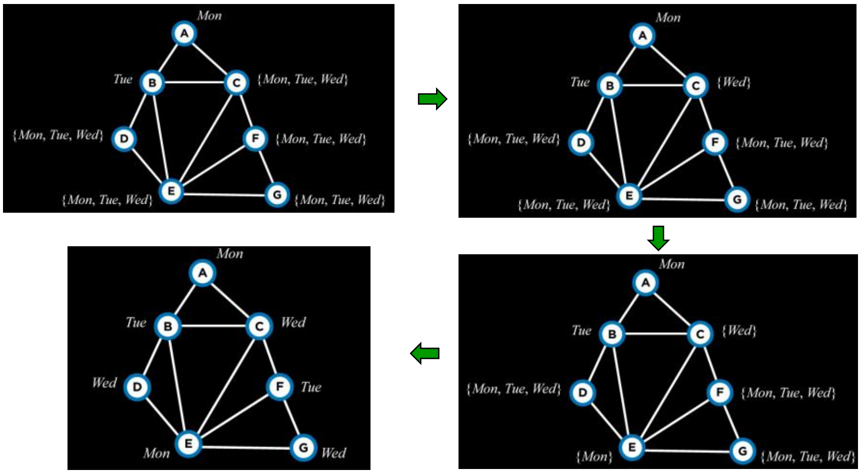


Interleave

```
function Backtrack(assignment, csp):
if assignment complete:
    return assignment
var = Select-Unassigned-Var(assignment, csp)
for value in Domain-Values(var, assignment, csp):
    if value consistent with assignment:
        add {var = value} to assignment
        inferences = Inference(assignment, csp)
        if inferences ≠ failure:
            add inferences to assignment
        result = Backtrack(assignment, csp)
        if result ≠ failure:
            return result
        remove {var = value} and inferences from assignment
return failure
```

(Inference)

idea



Choosing Domain-Values: Least Constraining Values Heuristic

Least Constraining Values heuristic:

- select the value that will constrain the least other variables.
- While in the degree heuristic we wanted to use the variable that is more likely to constrain other variables, here we want this variable to place the least constraints on other variables.
- The point is to locate the variable that could be the largest potential source of trouble (the variable with the highest degree), and then render it the least troublesome that we can (assign the least constraining value to it).

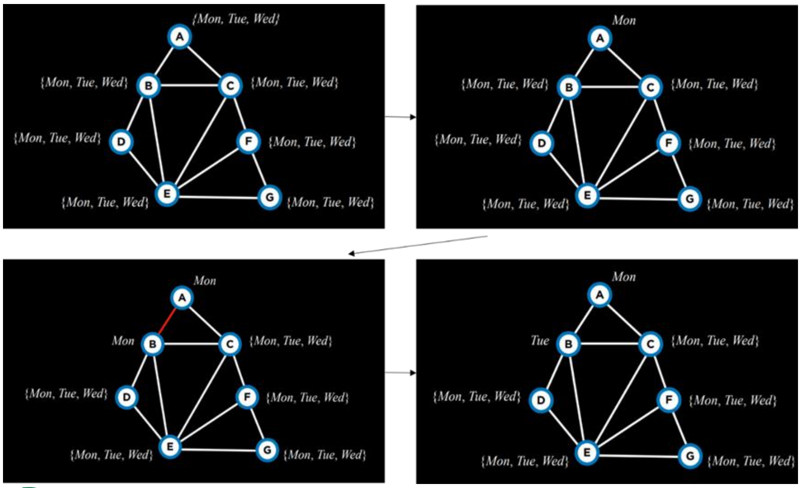
Slides and Course Materials modified from

University of California, Berkeley
[These slides adapted from Dan Klein and Pieter Abbeel]

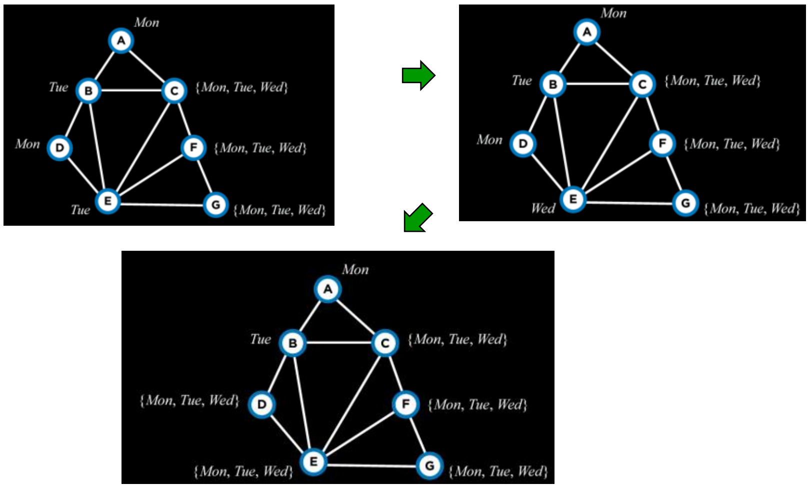
CS50 Course from Harvard
???

CSC 480 from Cal Poly
Franz Kurfess

Backtracking Example



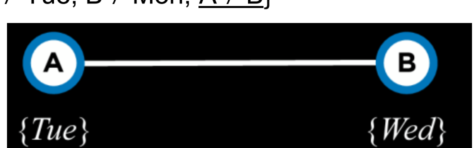
Backtracking Example - 2



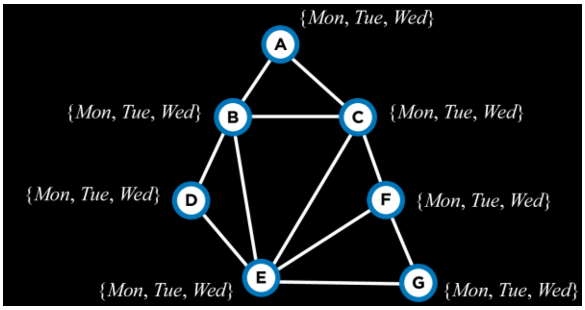
Arc consistency

- Arc consistency when all the values in a variable's domain satisfy the variable's binary constraints

{A ≠ Mon, B ≠ Tue, B ≠ Mon, A ≠ B}



Example: Arc consistency



Arc consistency alone will not solve the problem

Arc Consistency: AC-3

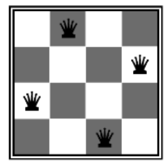
```
function AC-3(csp):
    queue = all arcs in csp
    while queue non-empty:
    (X, Y) = DEQUEUE(queue)
    if REVISE(csp, X, Y):
        if size of X.domain == 0:
            return false
        for each Z in X.neighbors - {Y}:
            ENQUEUE(queue, (Z, X))
    return true
```

Example: N-Queens

- Formulation 1:
  - Variables:
  - Domains:
  - Constraints:

X_{ij}

$\{0, 1\}$



$$\forall i, j, k \ \ (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \ \ (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \ \ (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \ \ (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

$$\sum_{i,j} X_{ij} = N$$

CAL POLY
SAN LUIS OBISPO

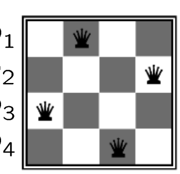
Computer Science Department

47

Example: N-Queens

- Formulation 2:
  - Variables: Q_k
  - Domains: $\{1, 2, 3, \ldots N\}$
  - Constraints:

Q_1
Q_2
Q_3
Q_4



Implicit: $\forall i, j \ \ \text{non-threatening}(Q_i, Q_j)$

Explicit: $(Q_1, Q_2) \in \{(1,3), (1,4), \ldots\}$

$\ldots$

CAL POLY
SAN LUIS OBISPO

Computer Science Department

48

Enforcing Arc Consistency in a CSP: AC-3

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue

function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi]; removed ← true
   return removed
```

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard – why?