Lecture 4: Constraint Satisfaction Problems

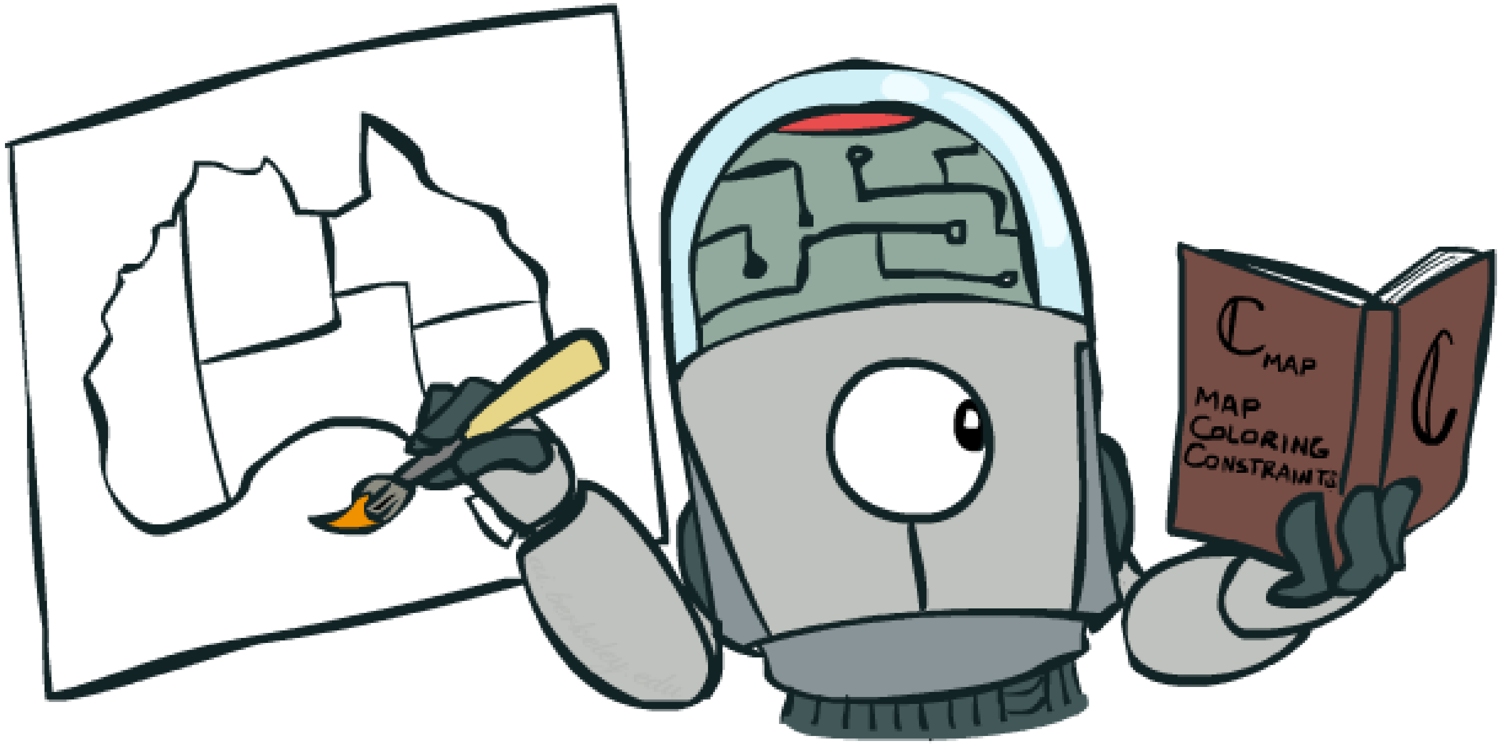
Shuai Li

John Hopcroft Center, Shanghai Jiao Tong University

https://shuaili8.github.io

https://shuaili8.github.io/Teaching/CS410/index.html

Part of slide credits: CMU AI & http://ai.berkeley.edu



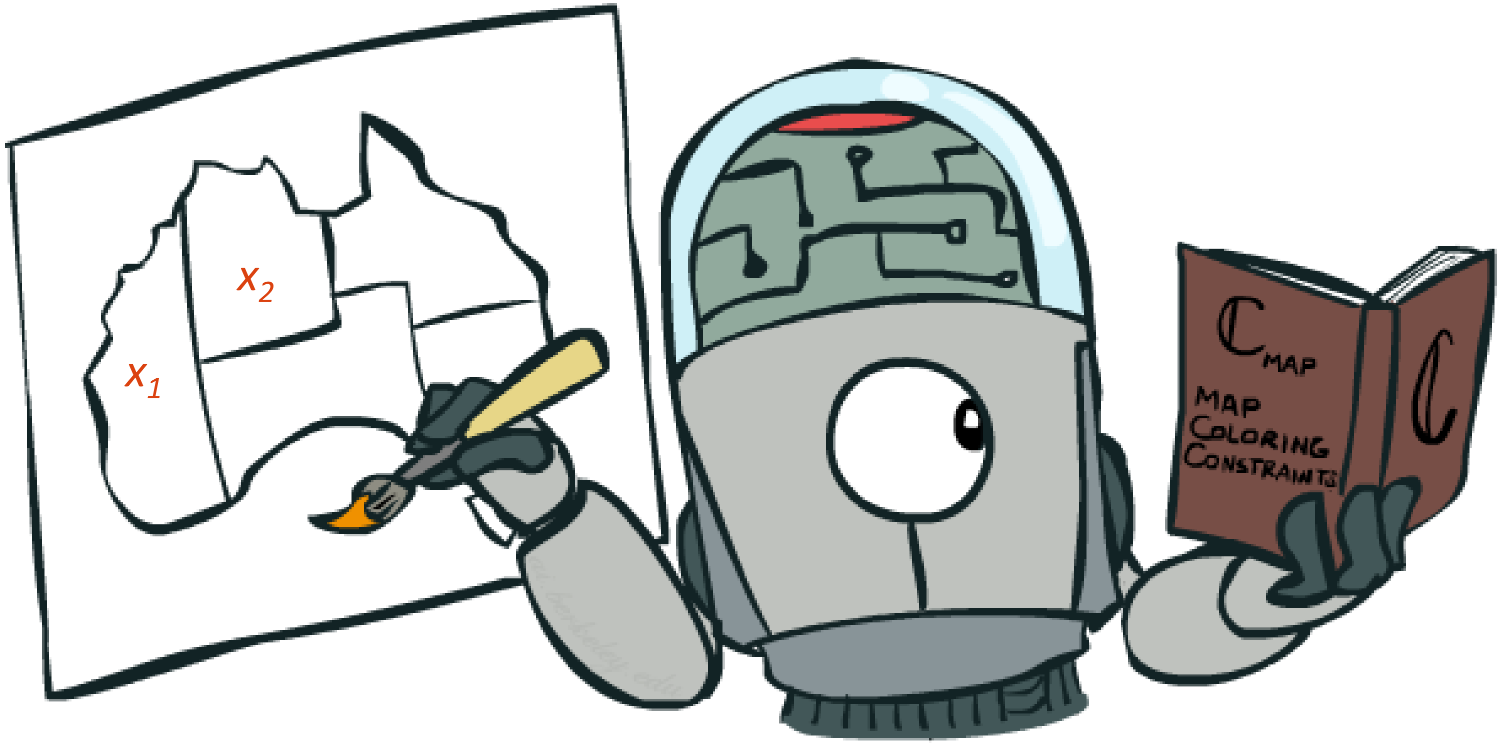
Constraint Satisfaction Problems

Constraint Satisfaction Problems

N variables

domain D

constraints



states

partial assignment

goal test

complete; satisfies constraints

successor function

assign an unassigned variable

What is Search For?

- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space

- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance

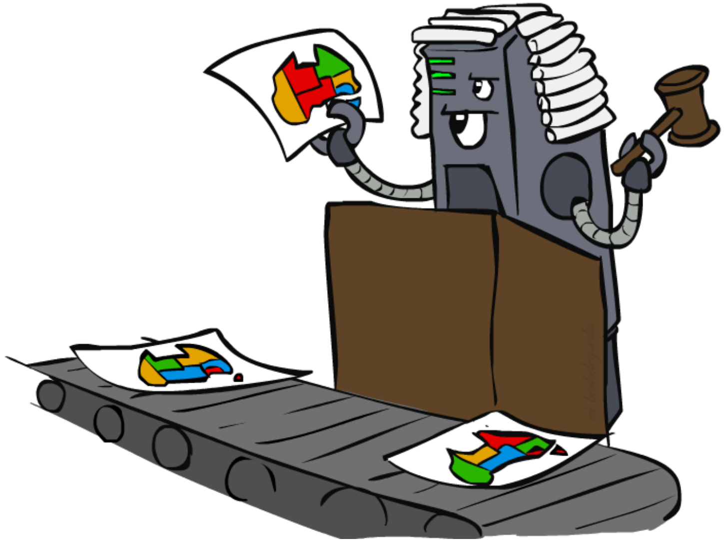
- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are specialized for identification problems

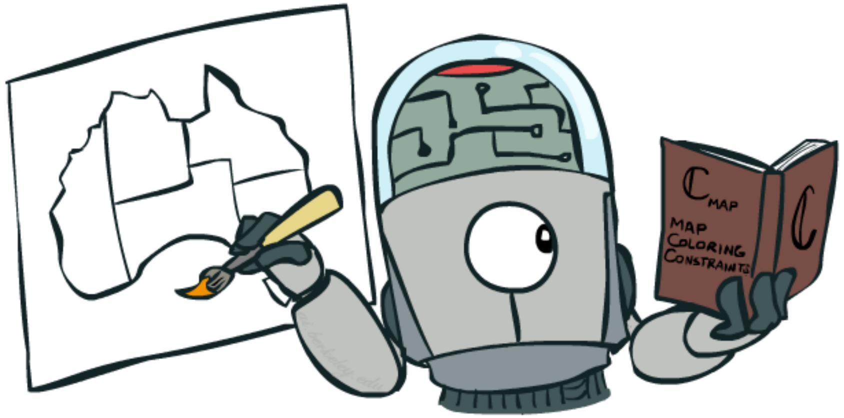
Constraint Satisfaction Problems

- Standard search problems:
  - State is a “black box”: arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything

- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

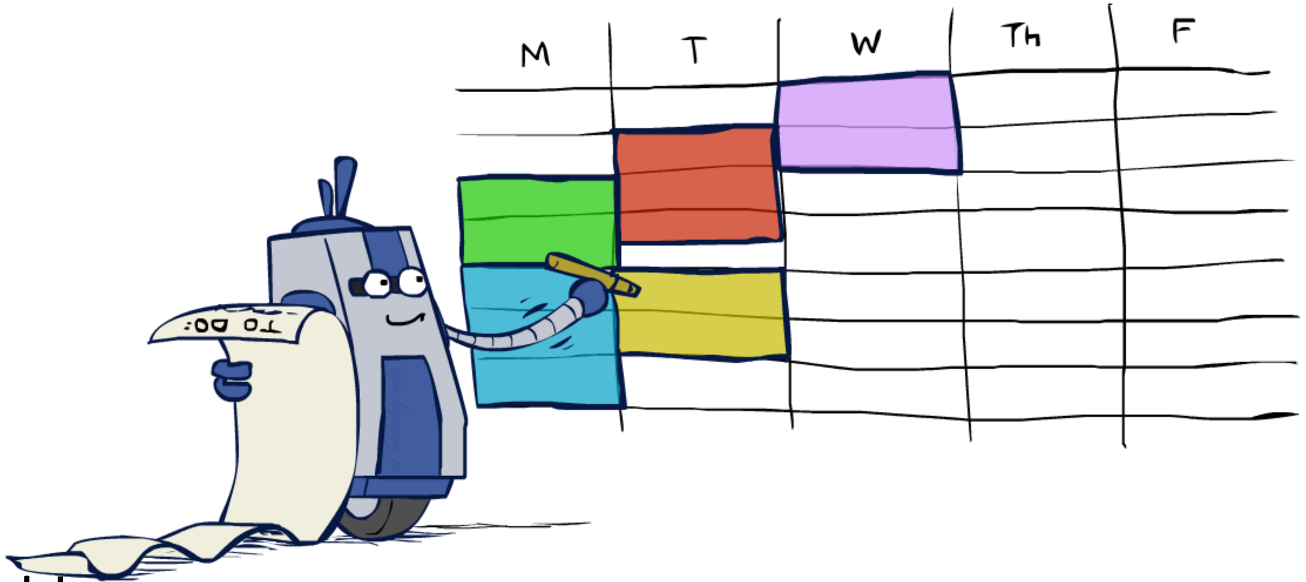
- Allows useful general-purpose algorithms with more power than standard search algorithms





Why study CSPs?

- Many real-world problems can be formulated as CSPs
- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!



- Sometimes involve real-valued variables…

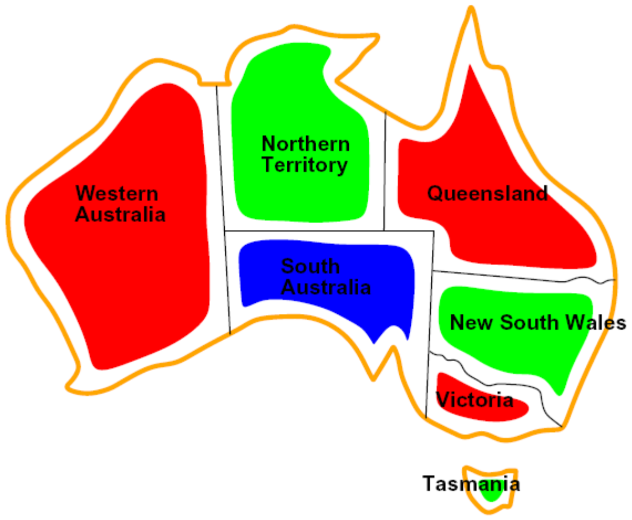
Example: Map Coloring

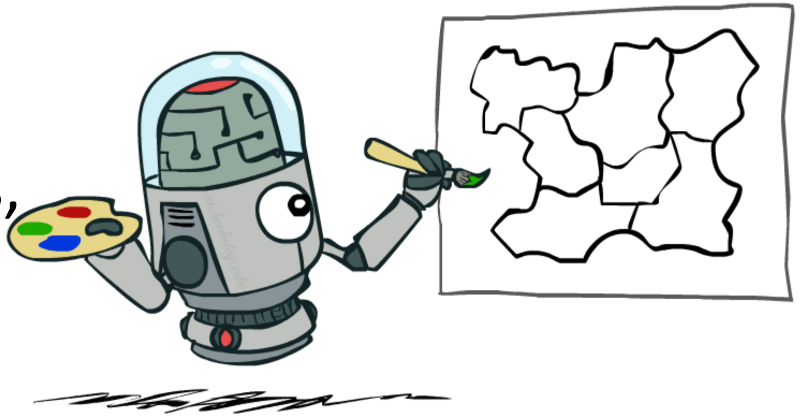
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: D={red, green, blue}

- Constraints: adjacent regions must have different colors:
  - Implicit: WA≠NT
  - Explicit: (WA,NT)∈{(red, green), (red, blue), …}

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

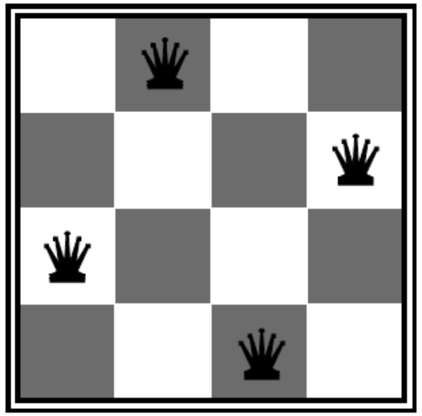
{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}

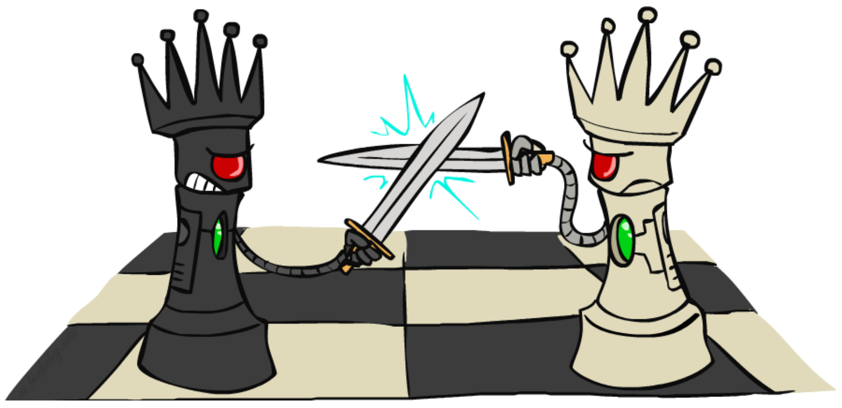




Example: N-Queens

- Formulation 1:
  - Variables: X_{ij}
  - Domains: $\{0,1\}$
  - Constraints:

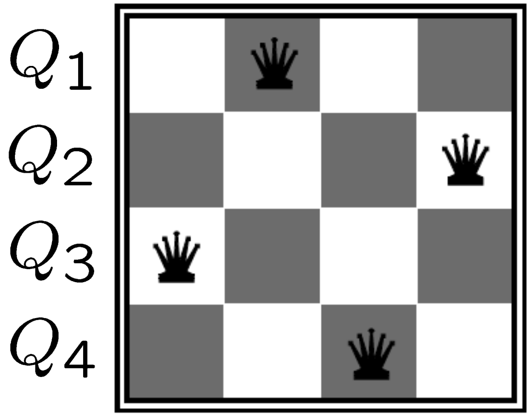




$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) \in \{(0,0),(0,1),(1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j-k}) \in \{(0,0),(0,1),(1,0)\}$$

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

Example: N-Queens 2



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

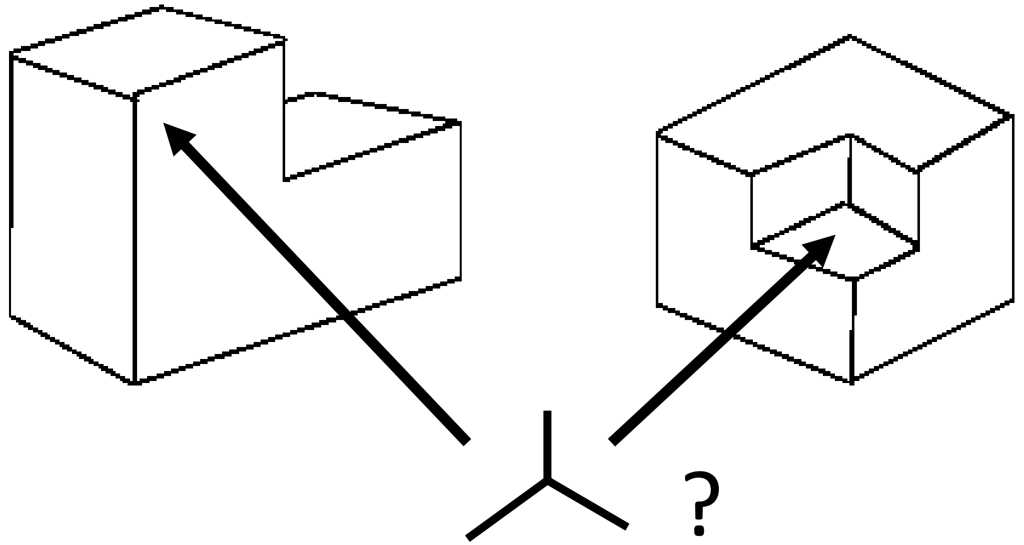
Implicit: $\forall i, j \;\; \text{non-threatening}(Q_i, Q_j)$

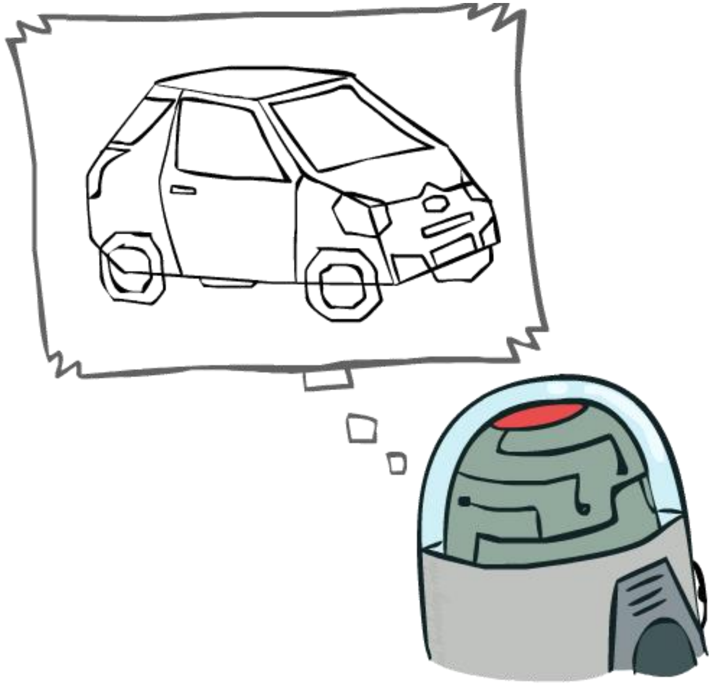
Explicit: $(Q_1, Q_2) \in \{(1,3),(1,4),\ldots\}$

$\cdots$

Example: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects
- An early example of an AI computation posed as a CSP





- Approach:
  - Each intersection is a variable
  - Adjacent intersections impose constraints on each other
  - Solutions are physically realizable 3D interpretations

Example: Cryptarithmetic

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

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

- Constraints:

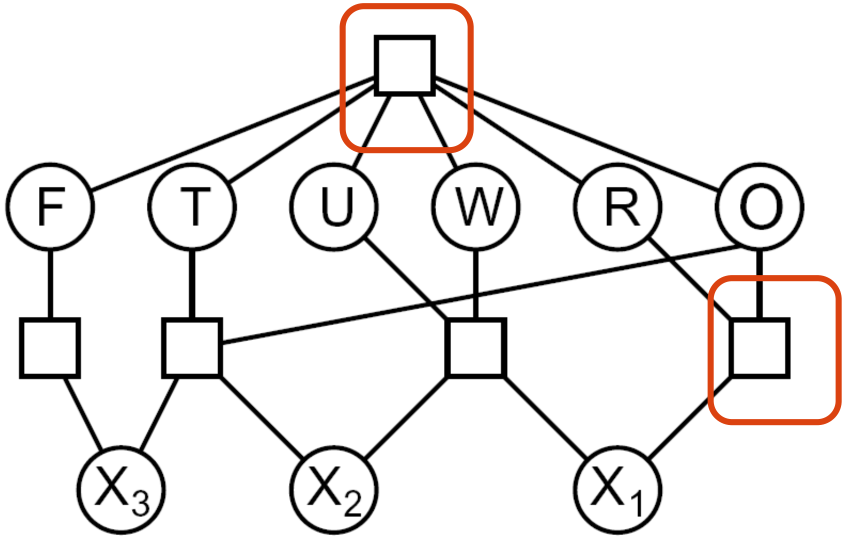
  $\boxed{\text{alldiff}(F, T, U, W, R, O)}$

  $\boxed{O + O = R + 10 \cdot X_1}$

  $\cdots$

X_1

```
    T W O
+   T W O
---------
  F O U R
```



Example: Sudoku

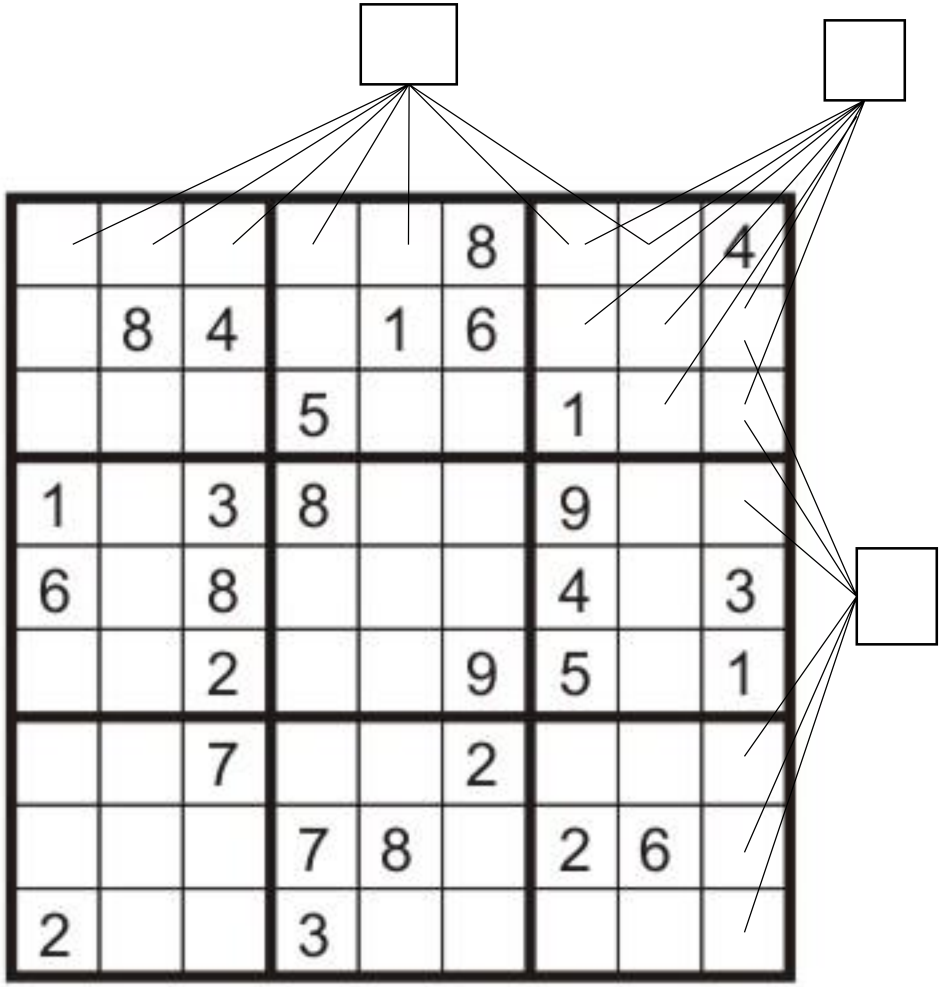
- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:

  9-way alldiff for each column

  9-way alldiff for each row

  9-way alldiff for each region

  (or can have a bunch of pairwise inequality constraints)

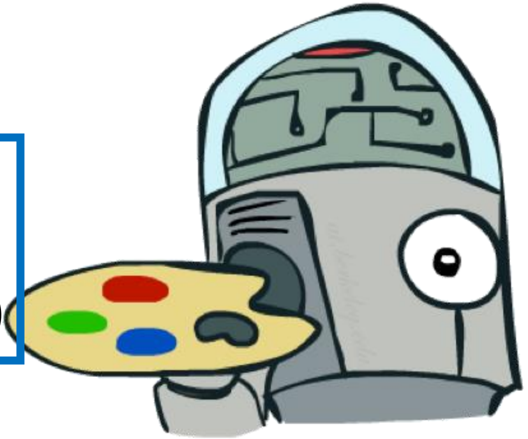


Varieties of CSPs

- Discrete Variables   We will cover in this lecture
  - Finite domains
    - Size d means $O(d^n)$ complete assignments
    - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end times for each job
    - Linear constraints solvable, nonlinear undecidable

Related with linear programming

- Continuous variables
  - E.g., start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by LP methods



Varieties of Constraints 2

- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), e.g.:

    $$SA \neq green$$ Focus of this lecture

  - Binary constraints involve pairs of variables, e.g.:

    $$SA \neq WA$$

  - Higher-order constraints involve 3 or more variables:
    e.g., cryptarithmetic column constraints

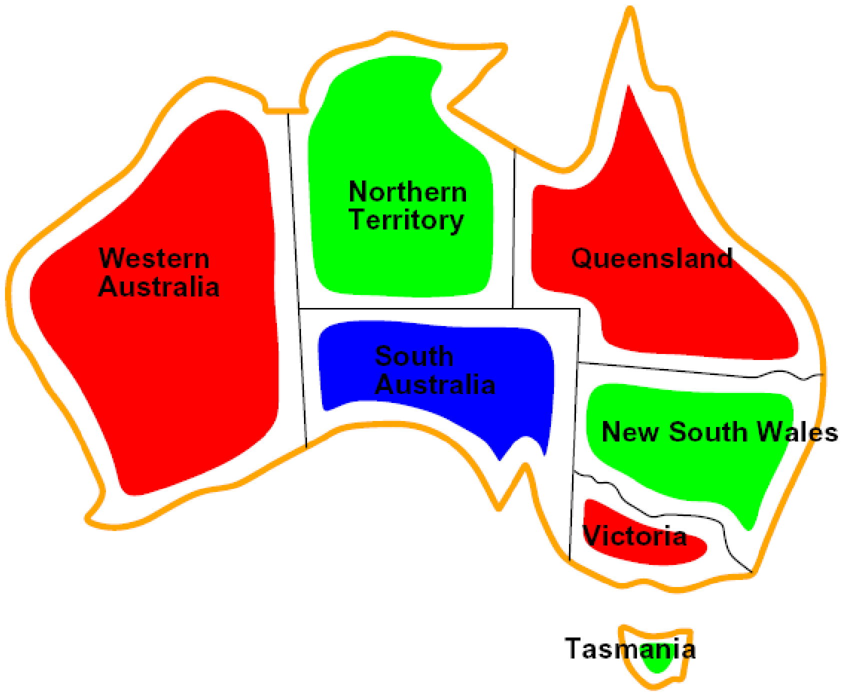
- Preferences (soft constraints):
  - E.g., red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (We'll ignore these until we get to Bayes' nets)

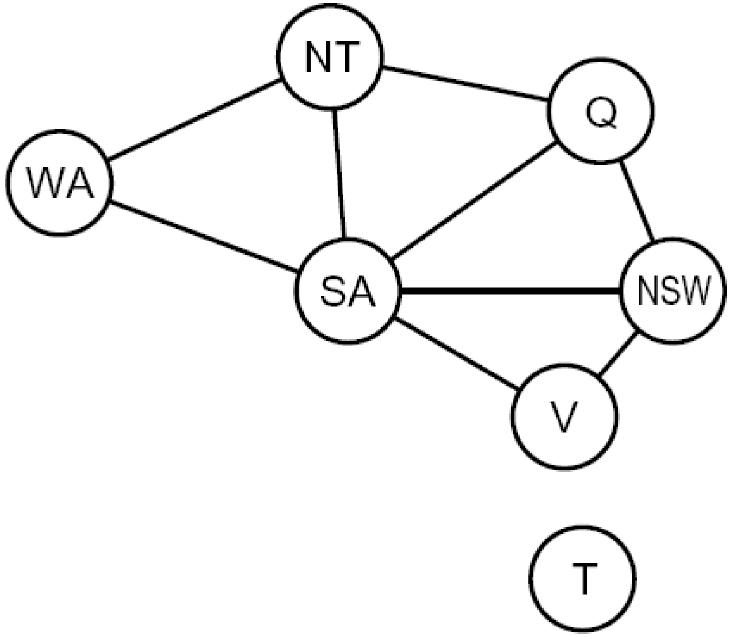
Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables

- Binary constraint graph: nodes are variables, arcs show constraints

- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!





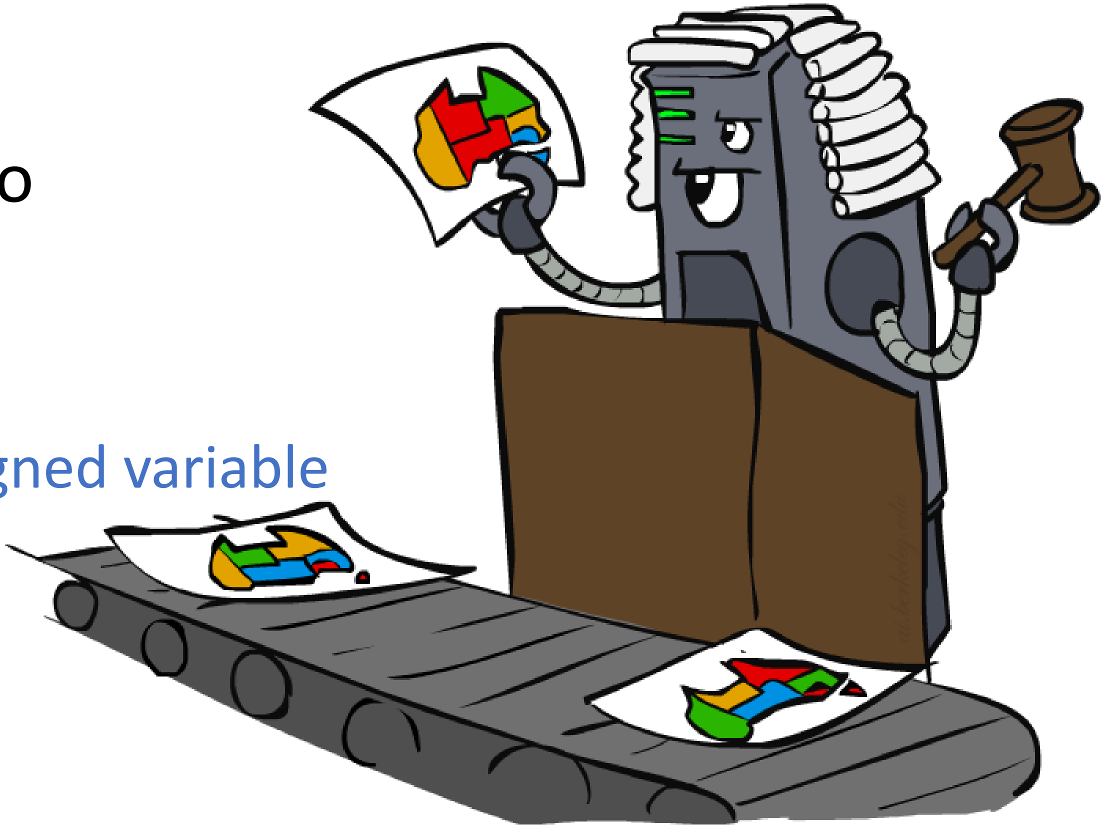
Solving CSPs

Standard Search Formulation

- Standard search formulation of CSPs

- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable →Can be any unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints

- We'll start with the straightforward, naïve approach, then improve it

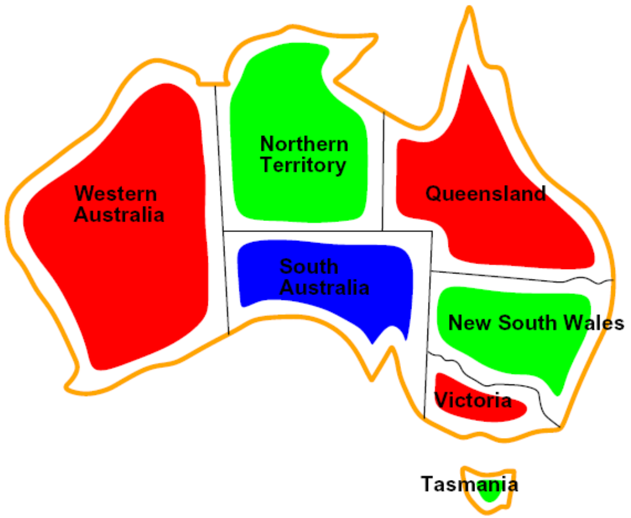


Search Methods: BFS

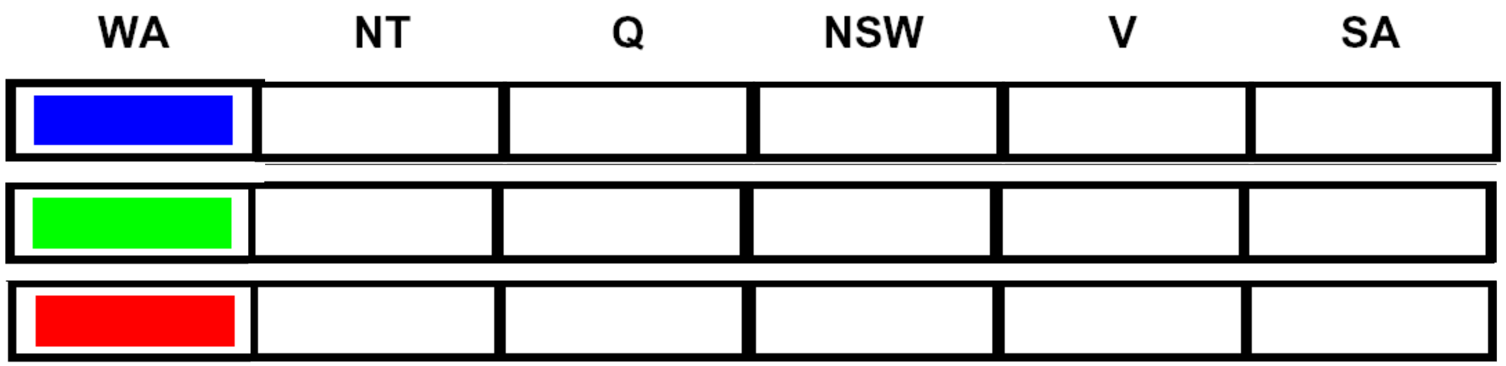
- What would BFS do?

{}

{WA=g} {WA=r} … {NT=g} …

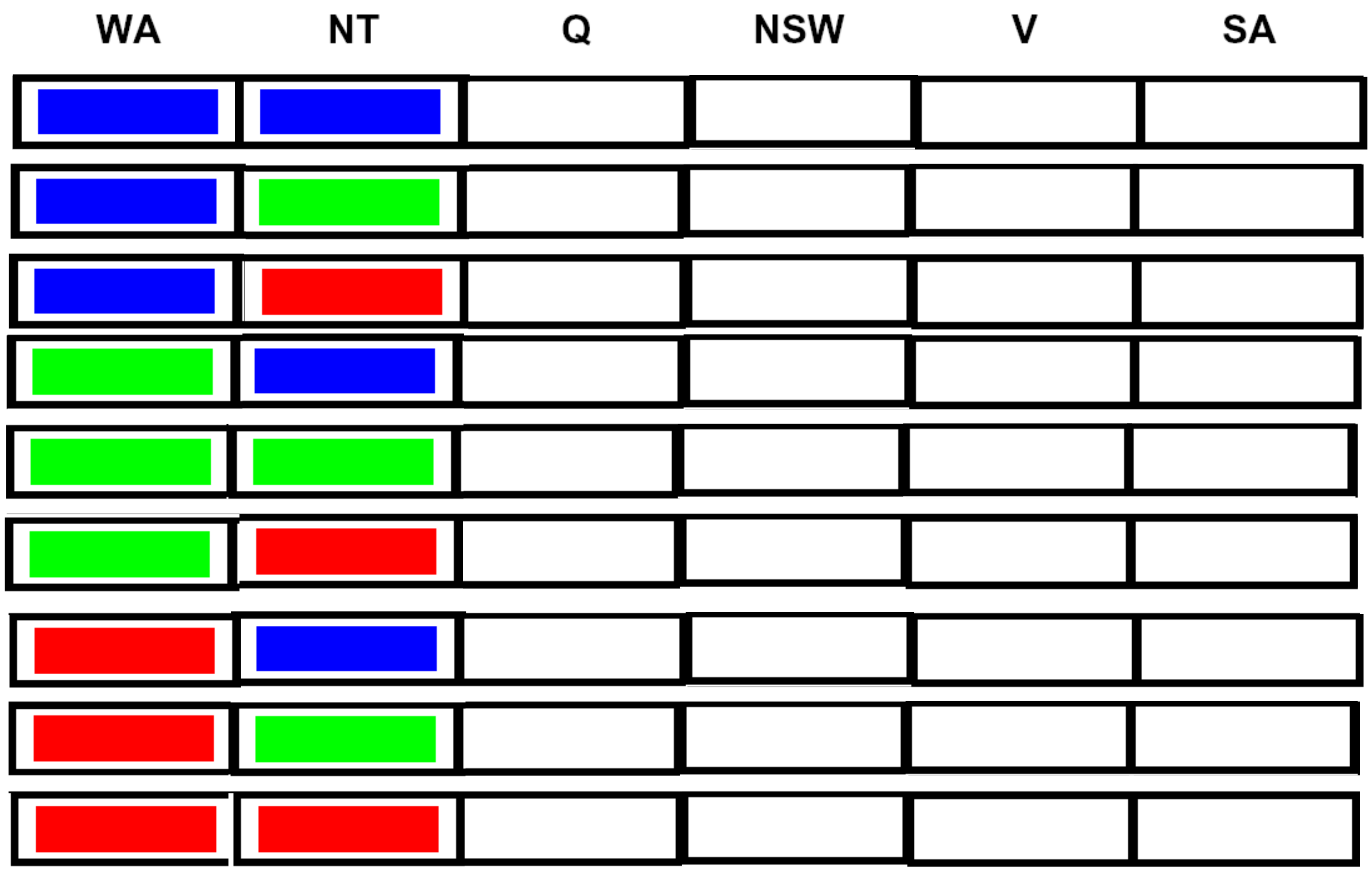


Search Methods: BFS 2



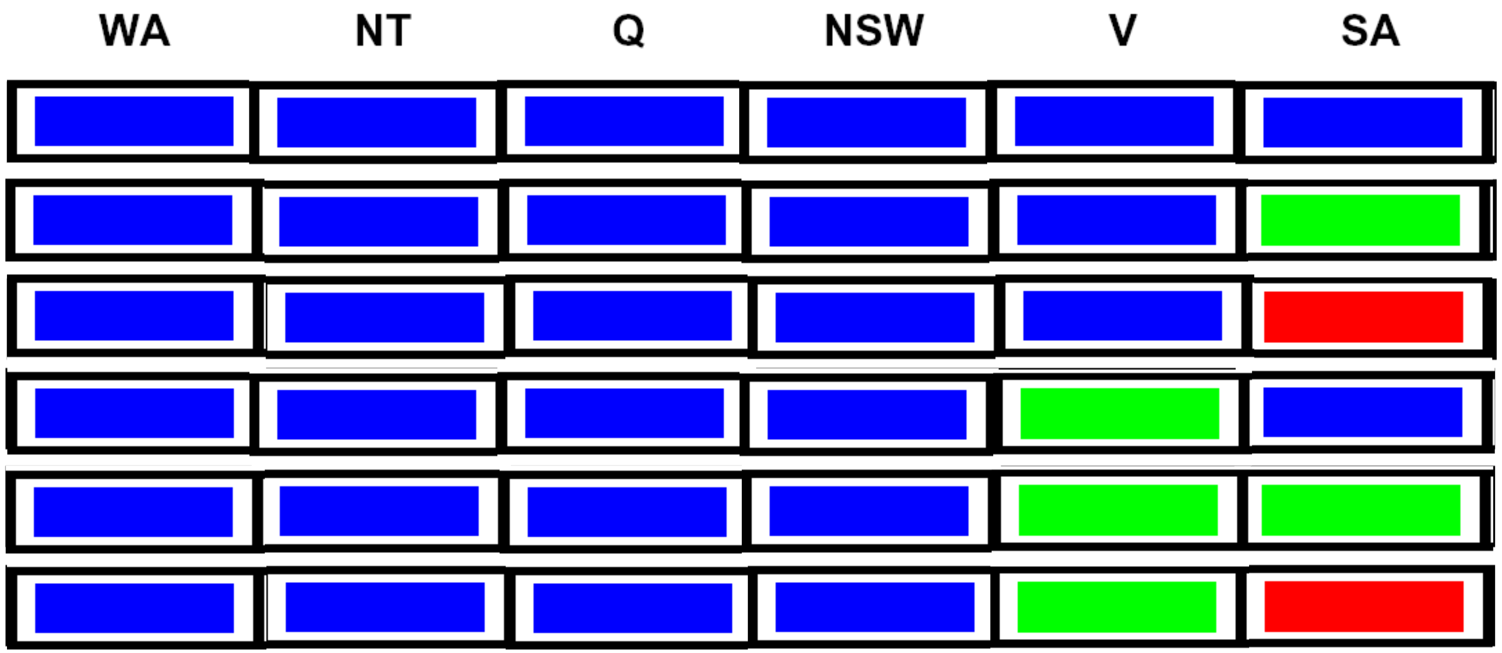
- Any one variable

Search Methods: BFS 3



- Any two variables

Search Methods: BFS 4

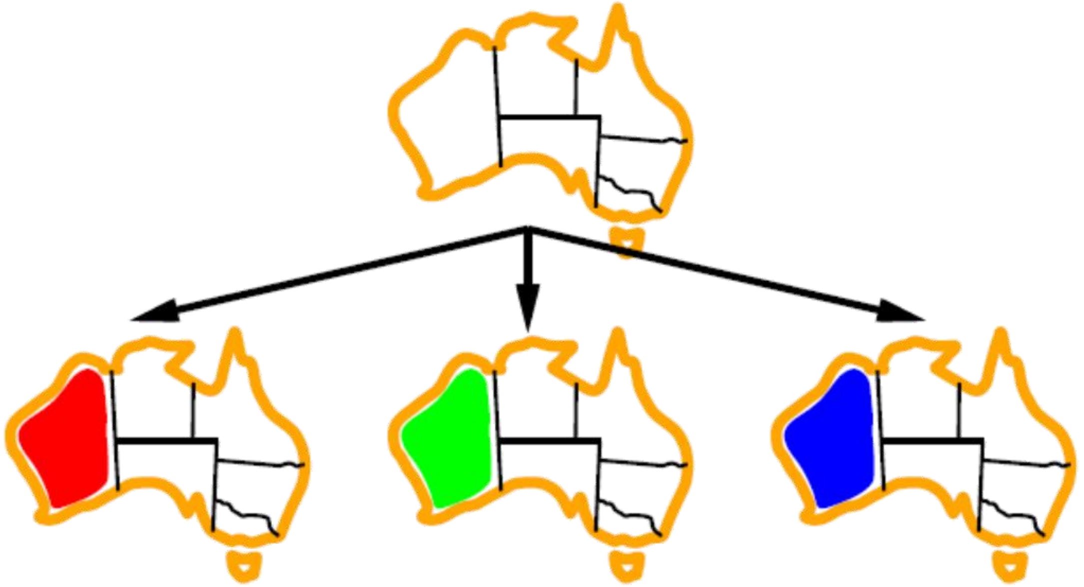


…

- Any assignment for all variables

Search Methods: DFS

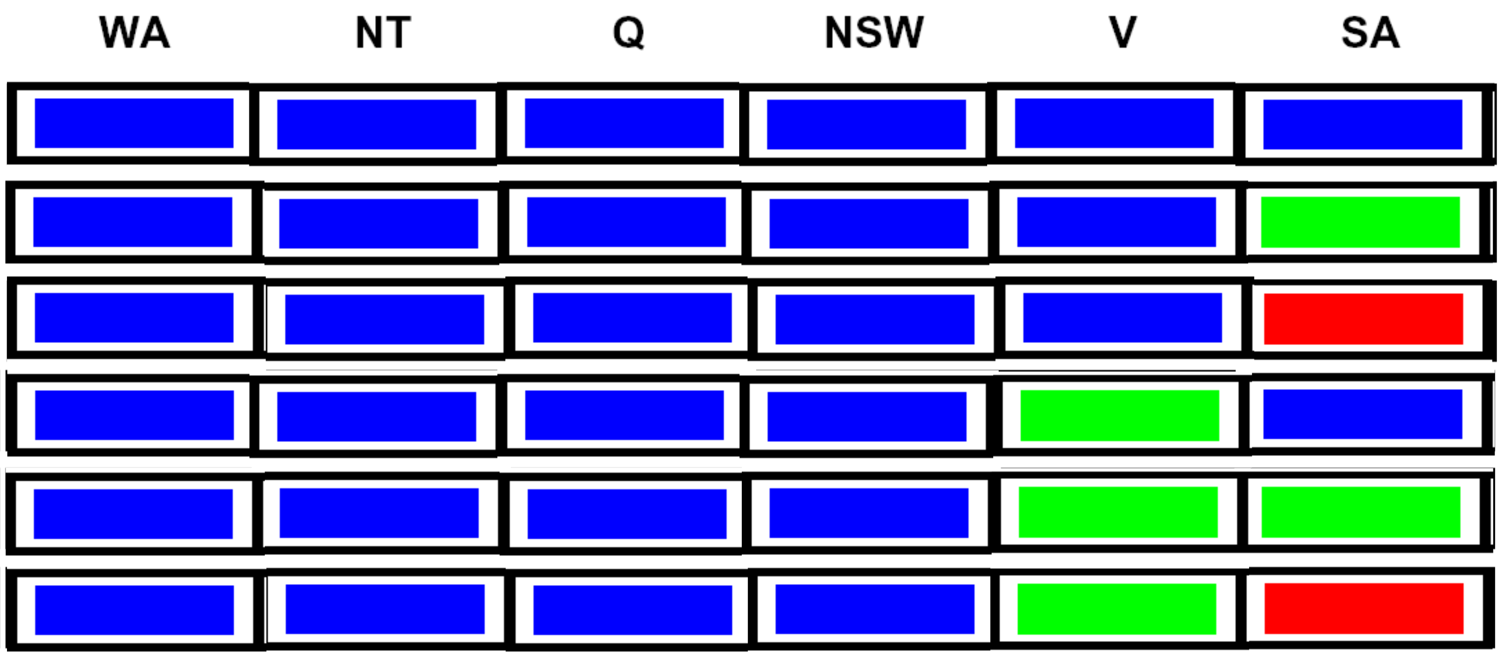
- At each node, assign a value from the domain to the variable
- Check feasibility (constraints) when the assignment is complete



- What problems does the naïve search have?

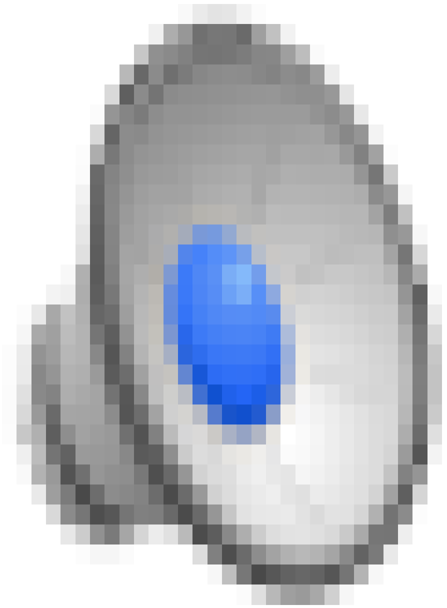
[Demo: coloring -- dfs]

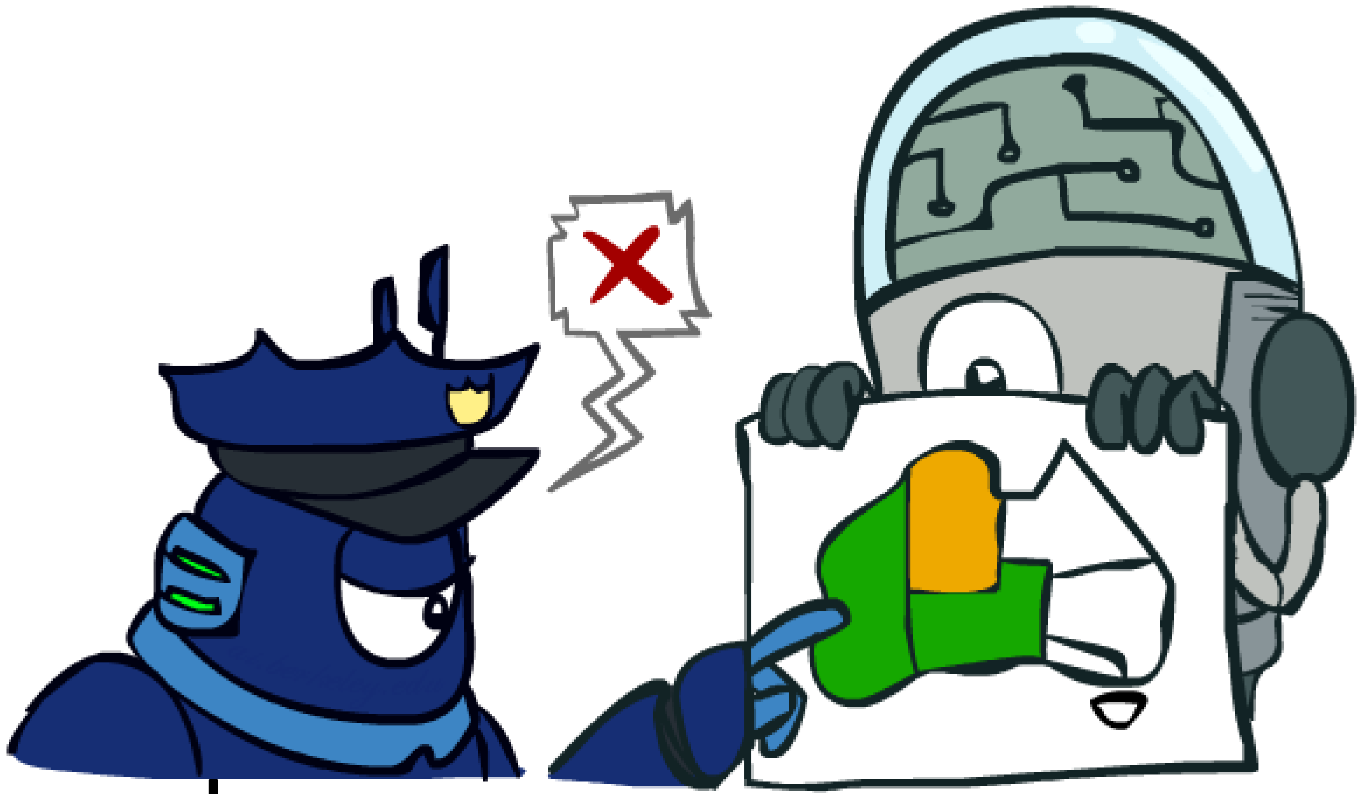
Search Methods: DFS 2



...

Video of Demo Coloring -- DFS



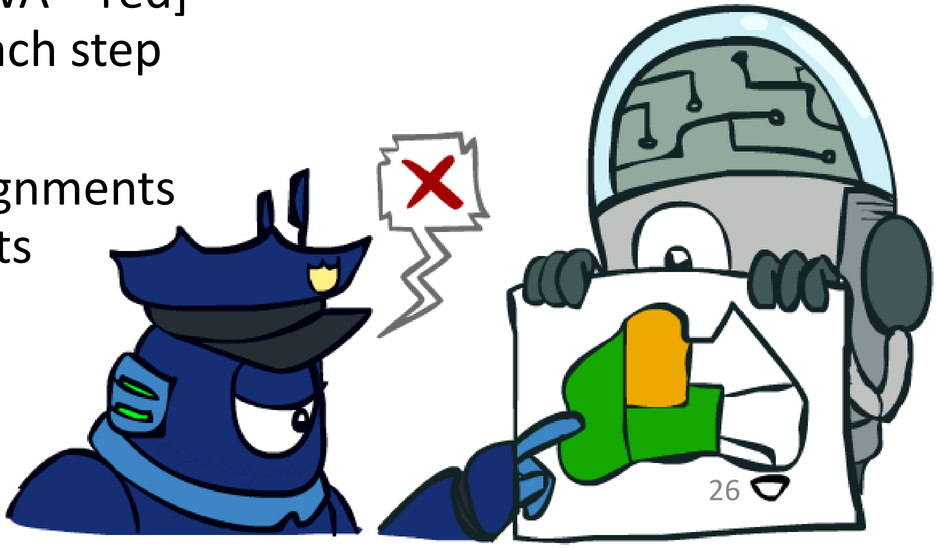


Backtracking Search

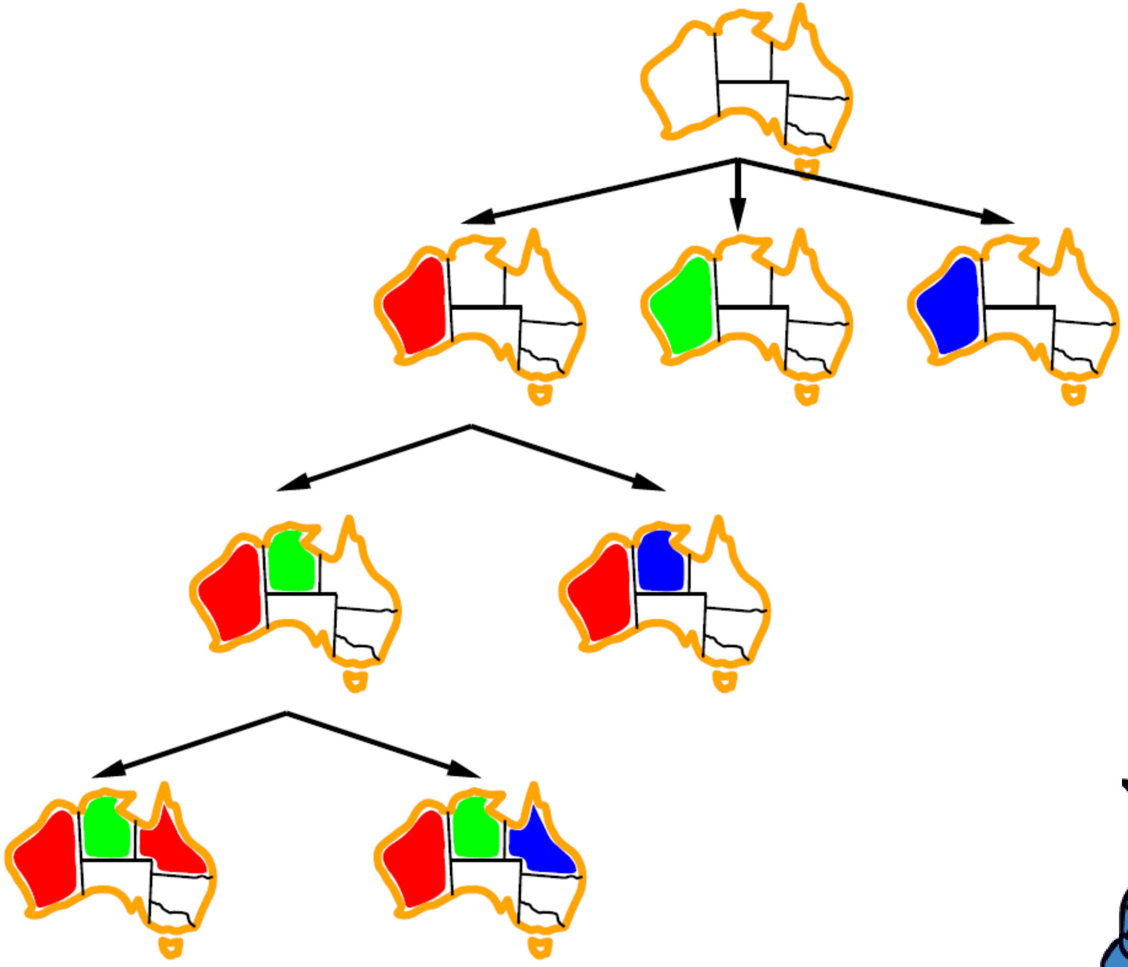
Backtracking Search

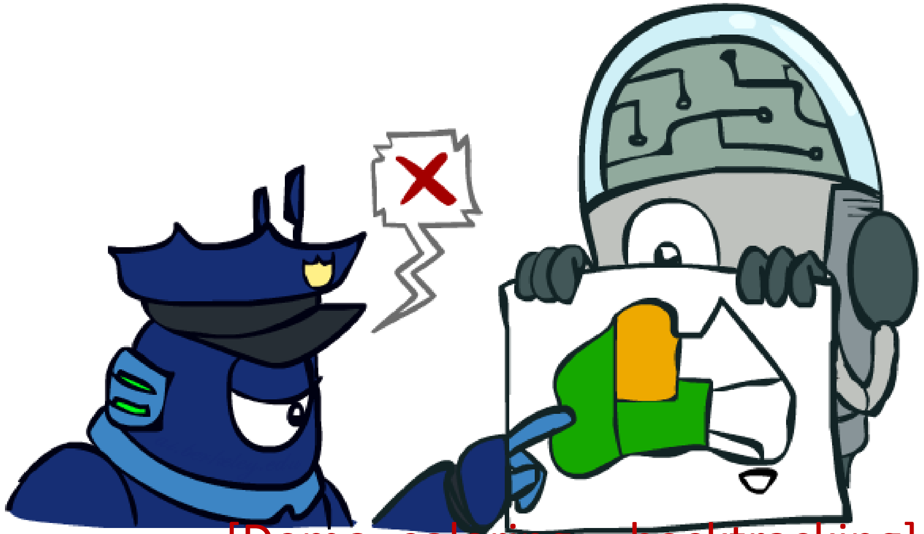
- Backtracking search is the basic uninformed algorithm for solving CSPs
- Backtracking search = DFS + two improvements

- 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”
- Can solve N-queens for $N \approx 25$



Example





[Demo: coloring -- backtracking]

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return RECURSIVE_BACKTRACKING({}, csp)

function RECURSIVE_BACKTRACKING(assignment, csp) returns a solution, or failure
    if assignment is complete then
        return assignment
    var ⟵ SELECT_UNASSIGNED_VARIABLE(VARIABLES[csp], assignment, 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
            result ⟵ RECURSIVE_BACKTRACKING(assignment, csp)
            if result ≠ failure then
                return result
            remove {var=value} from assignment
    return failure
```

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return RECURSIVE_BACKTRACKING({}, csp)

function RECURSIVE_BACKTRACKING(assignment, csp) returns a solution, or failure
    if assignment is complete then
        return assignment
    var ⟵ SELECT_UNASSIGNED_VARIABLE(VARIABLES[csp], assignment, 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
            result ⟵ RECURSIVE_BACKTRACKING(assignment, csp)
            if result ≠ failure then
                return result
            remove {var=value} from assignment
    return failure
```

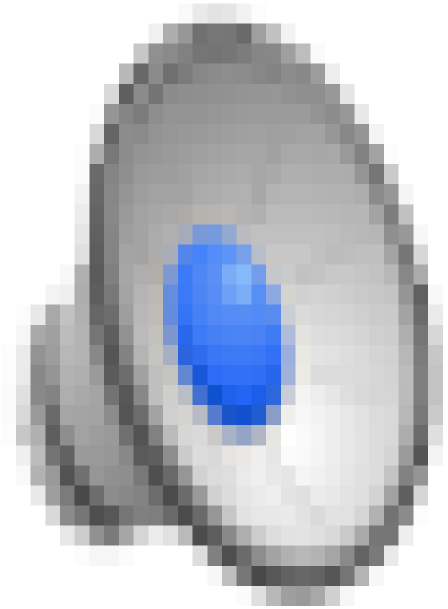
No need to check consistency for a complete assignment

What are choice points?

Checks consistency at each assignment

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

Video of Demo Coloring – Backtracking



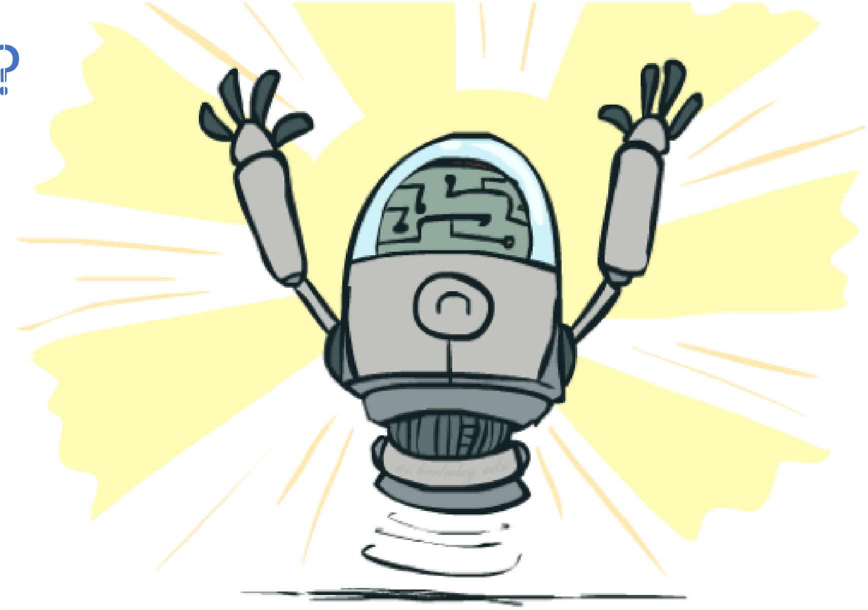
Improving Backtracking

- General-purpose ideas give huge gains in speed

- Filtering: Can we detect inevitable failure early?

- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

- Structure: Can we exploit the problem structure?

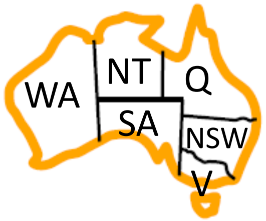


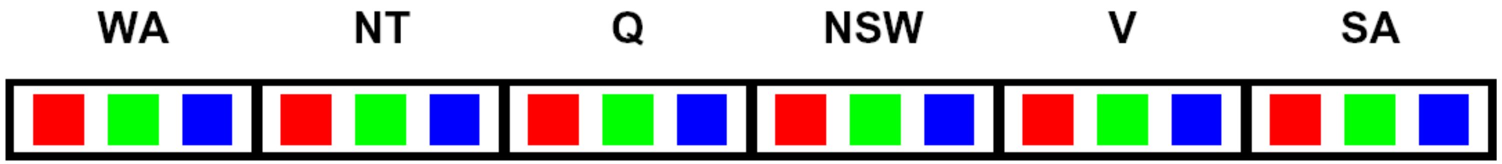
Filtering

Keep track of domains for unassigned variables and cross off bad options

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 failure is detected if some variables have no values remaining

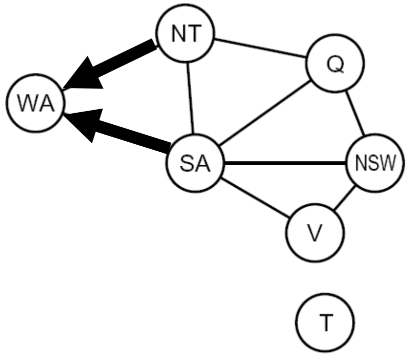


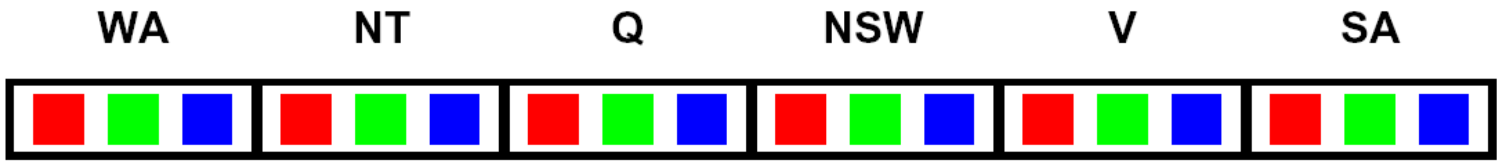


[Demo: coloring -- forward checking]

Filtering: Forward Checking 2

- 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



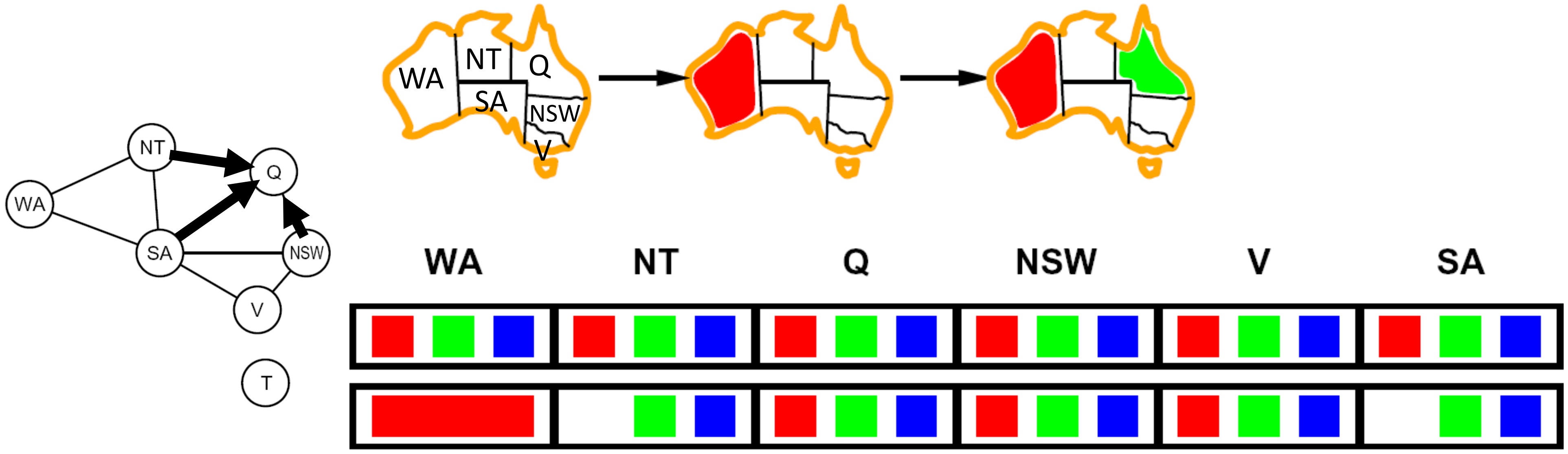


Recall: Binary constraint graph for a binary CSP (i.e., each constraint has most two variables): nodes are variables, edges show constraints

[Demo: coloring -- forward checking]

Filtering: Forward Checking 3

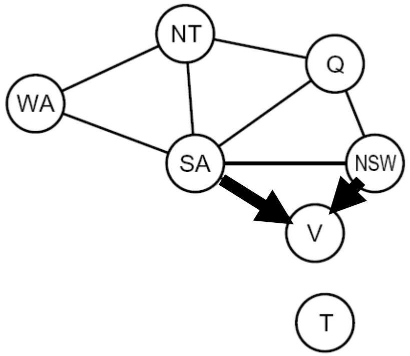
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

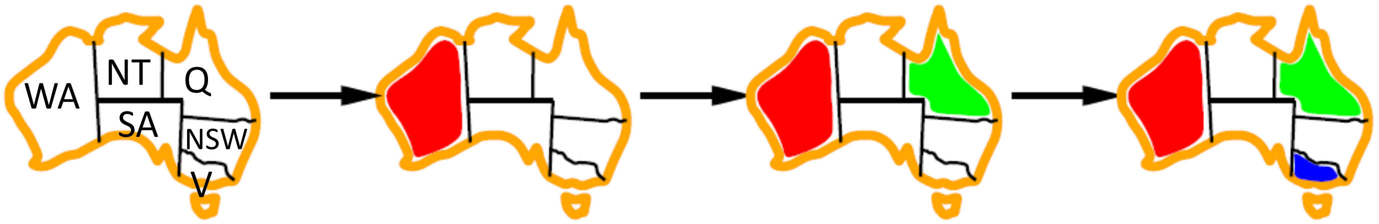


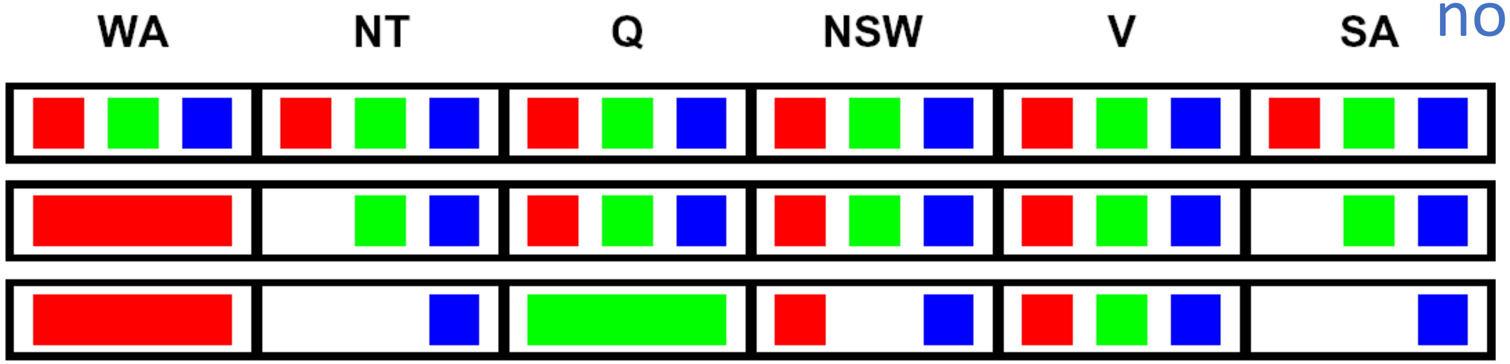
[Demo: coloring -- forward checking]

Filtering: Forward Checking 4

- 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



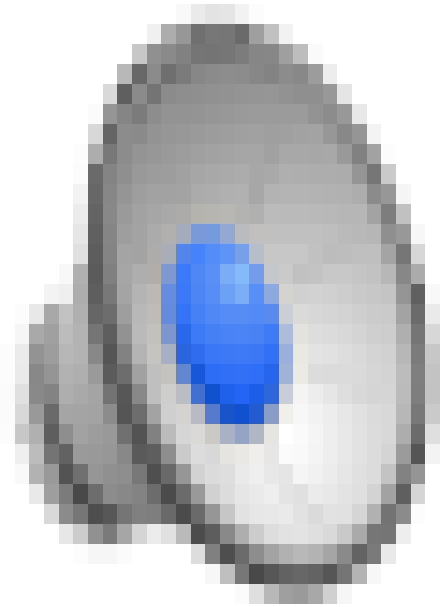




FAIL – variable with no possible values

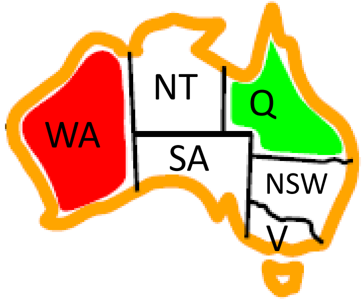
[Demo: coloring -- forward checking]

Video of Demo Coloring – Backtracking with Forward Checking



Filtering: Constraint Propagation

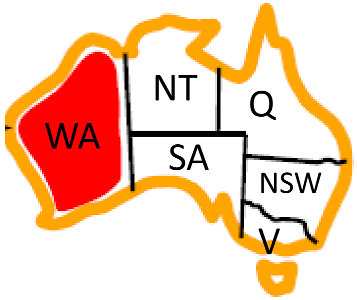
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

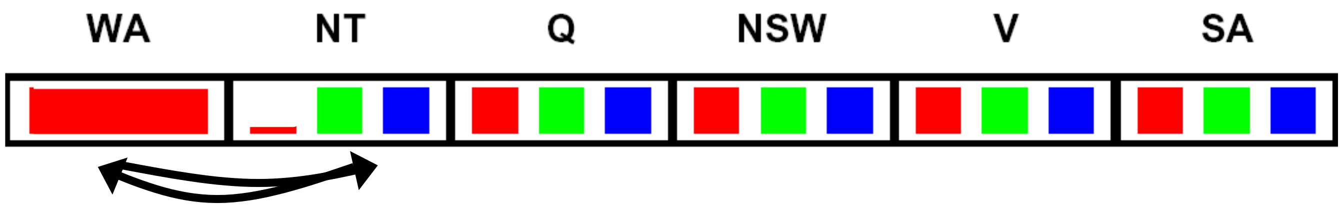


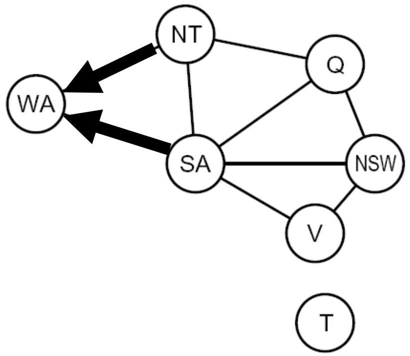
- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

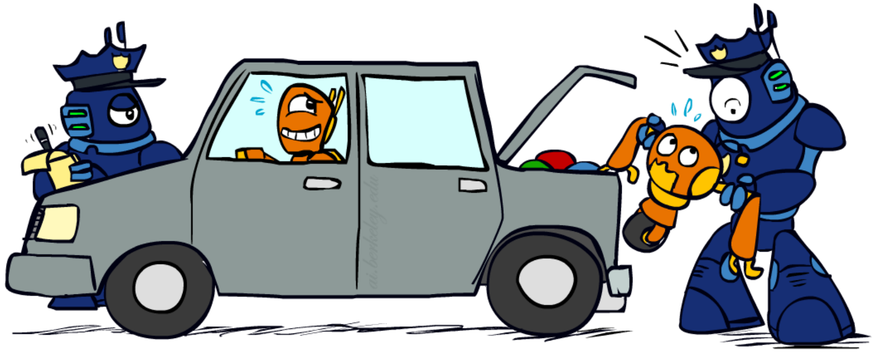
Consistency of A Single Arc

- An arc X $\rightarrow$ 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!

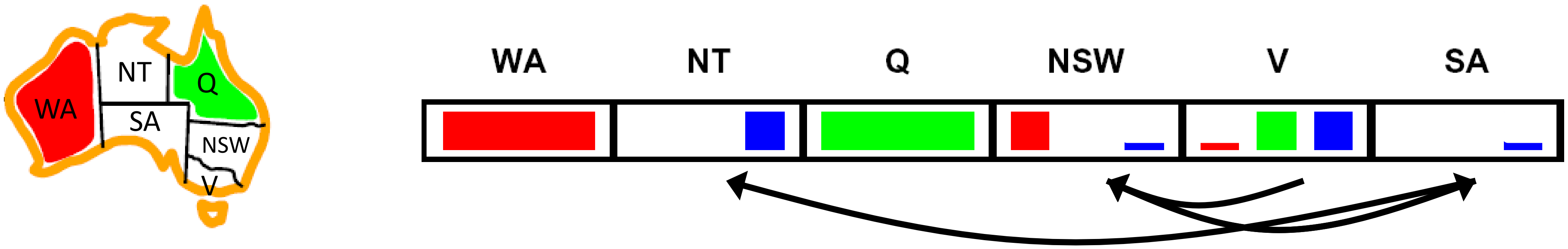
Forward checking?

A special case

Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP

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

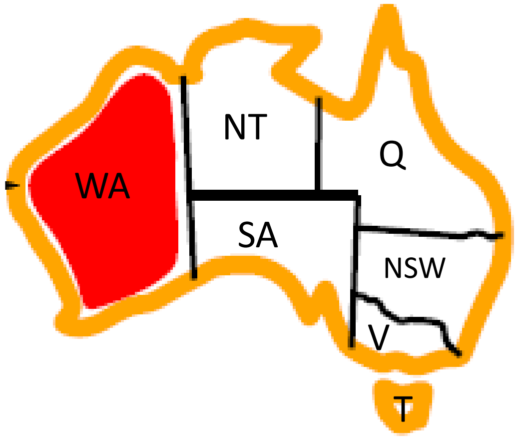


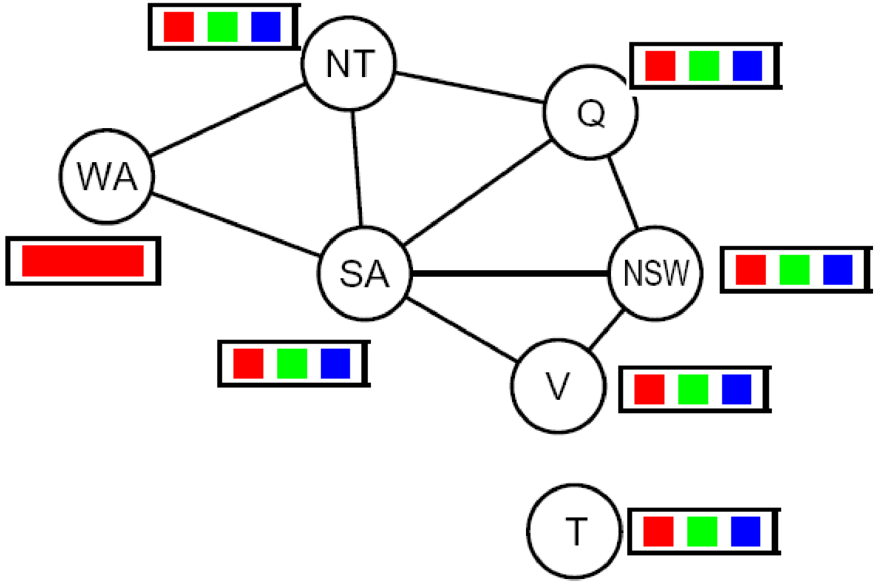
- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

Arc Consistency of Entire CSP 2

- A simplistic algorithm: Cycle over the pairs of variables, enforcing arc-consistency, repeating the cycle until no domains change for a whole cycle
- AC-3 (Arc Consistency Algorithm #3):
  - A more efficient algorithm ignoring constraints that have not been modified since they were last analyzed





```
function AC-3(csp) returns the CSP, possibly with reduced domains
    initialize a queue of all the arcs in csp
    while queue is not empty do
        (X_i, X_j) ⟵ REMOVE_FIRST(queue)
        if REMOVE_INCONSISTENT_VALUES(X_i, X_j) then
            for each X_k in NEIGHBORS[X_i] do
                add (X_k, X_i) to queue

function REMOVE_INCONSISTENT_VALUES(X_i, X_j)  returns true iff succeeds
    removed ⟵ false
    for each x in DOMAIN[X_i] do
        if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ⟷ X_j then
            delete x from DOMAIN[X_i]; removed ⟵ true
    return removed
```

```
function AC-3(csp) returns the CSP, possibly with reduced domains
    initialize a queue of all the arcs in csp
    while queue is not empty do
        (X_i, X_j) ⟵ REMOVE_FIRST(queue)
        if REMOVE_INCONSISTENT_VALUES(X_i, X_j) then
            for each X_k in NEIGHBORS[X_i] do
                add (X_k, X_i) to queue
```

Constraint Propagation!

```
function REMOVE_INCONSISTENT_VALUES(X_i, X_j)  returns true iff succeeds
    removed ⟵ false
    for each x in DOMAIN[X_i] do
        if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ⟷ X_j then
            delete x from DOMAIN[X_i]; removed ⟵ true
    return removed
```

function AC-3(csp) returns the CSP, possibly with reduced domains

initialize a queue of all the arcs in csp

while queue is not empty do

$(X_i, X_j) \longleftarrow$ REMOVE_FIRST(queue)

$\boxed{\text{if REMOVE\_INCONSISTENT\_VALUES}(X_i, X_j) \text{ then}}$

for each X_k in NEIGHBORS[X_i] do

add (X_k, X_i) to queue

- An arc is added after a removal of value at a node
- n node in total, each has $\leq d$ values
- Total times of removal: $O(nd)$
- After a removal, $\leq n$ arcs added
- Total times of adding arcs: $O(n^2 d)$

function REMOVE_INCONSISTENT_VALUES(X_i, X_j) returns true iff succeeds

removed $\longleftarrow$ false

$\boxed{\text{for each x in DOMAIN}[X_i] \text{ do}}$

if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$ then

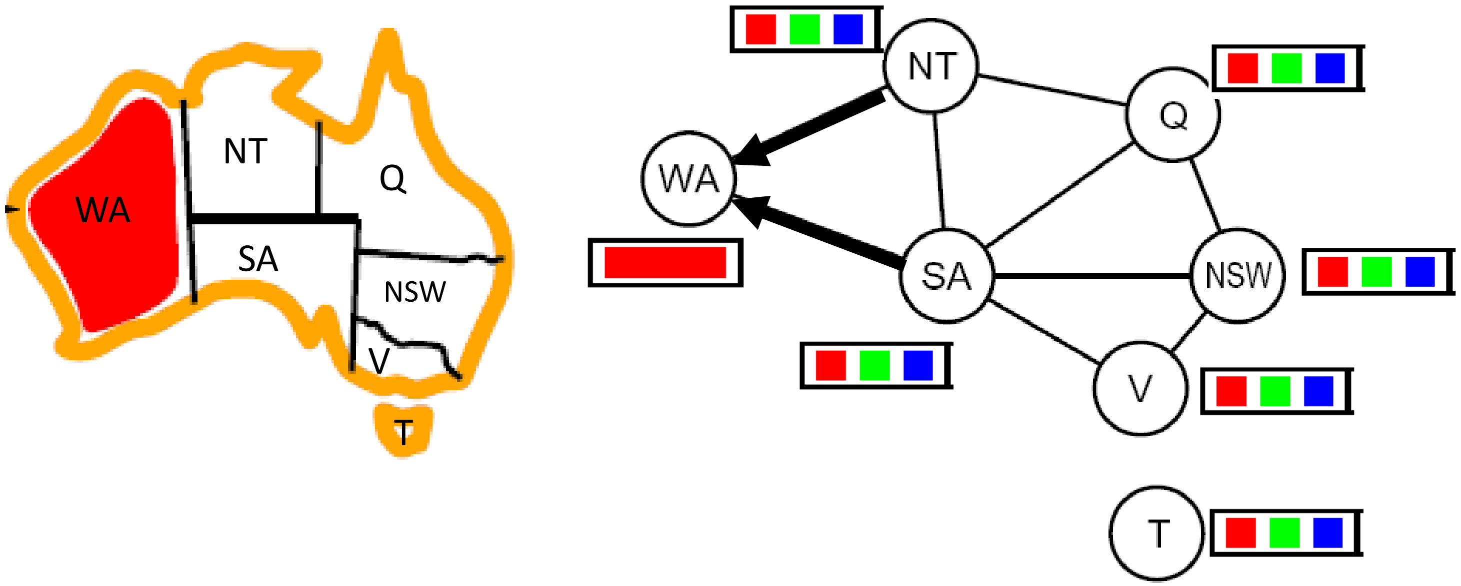
delete x from DOMAIN[X_i]; removed $\longleftarrow$ true

return removed

- Check arc consistency per arc: $O(d^2)$
- Complexity: $O(n^2 d^3)$
- Can be improved to $O(n^2 d^2)$

… but detecting all possible future problems is NP-hard – why?

Example of AC-3

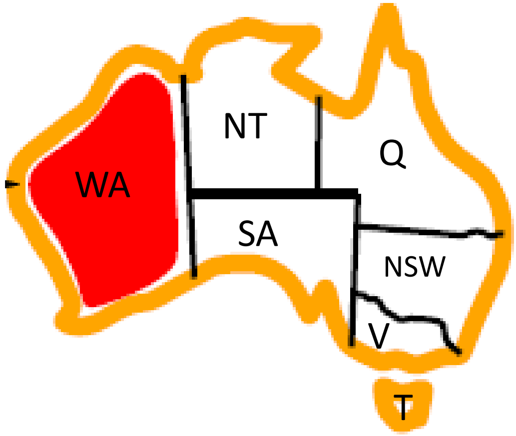


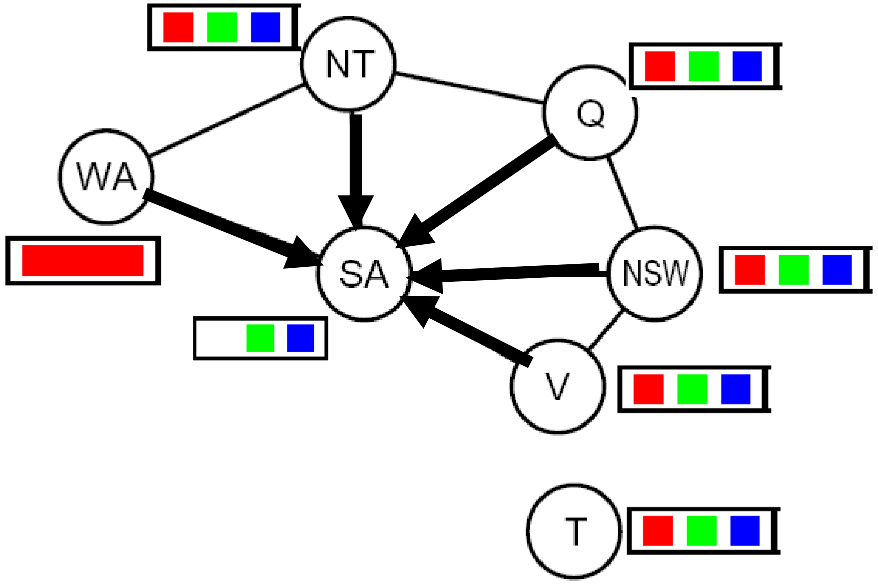
Queue:
SA->WA
NT->WA

Remember: Delete from the tail!

45

Example of AC-3 2





Queue:

~~SA->WA~~

NT->WA

WA->SA

NT->SA

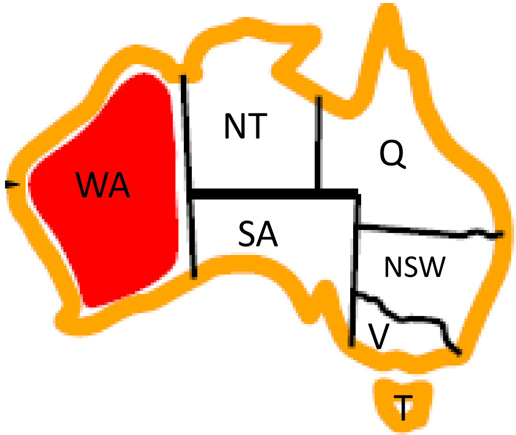
Q->SA

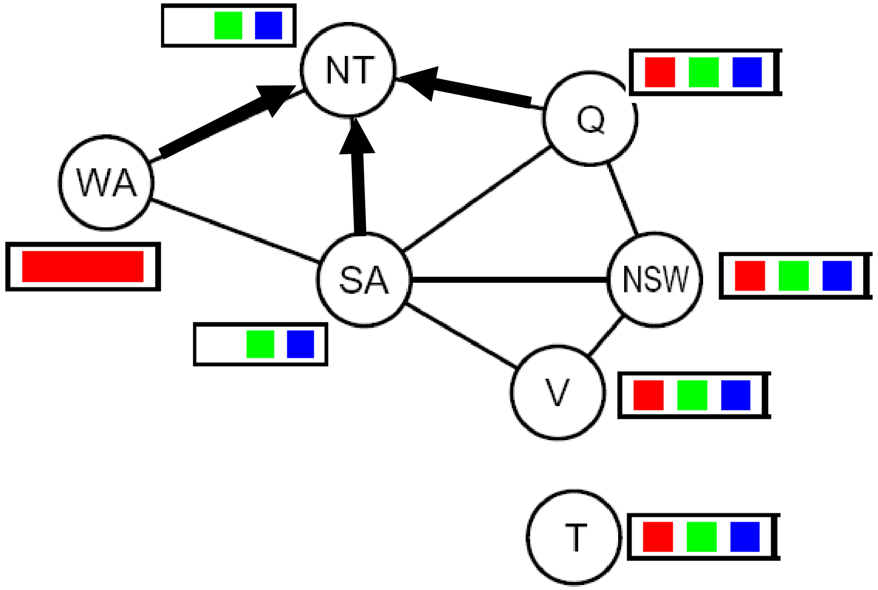
NSW->SA

V->SA

Remember: Delete from the tail!

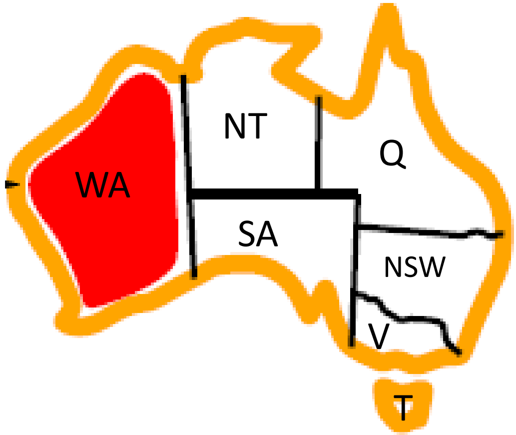
Example of AC-3 3

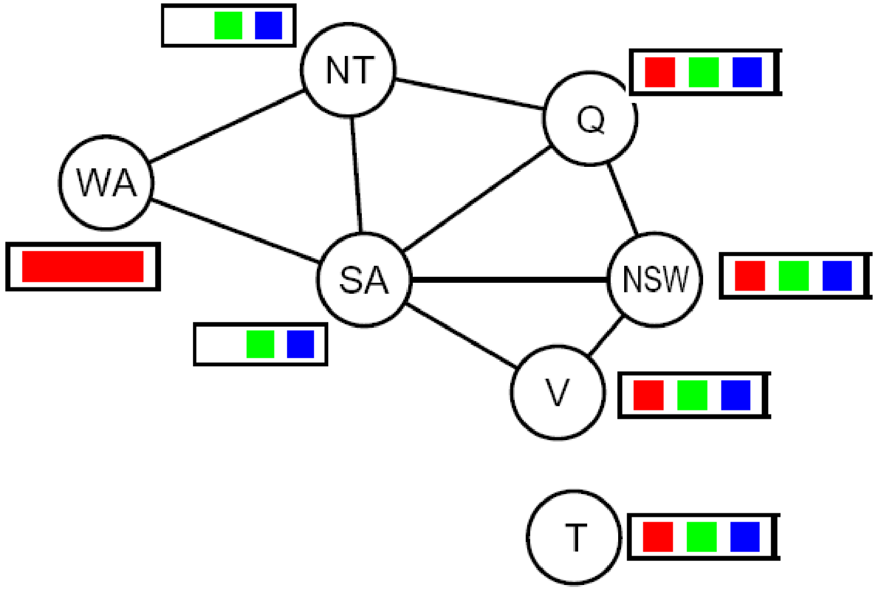




Queue:
~~SA->WA~~
~~NT->WA~~
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

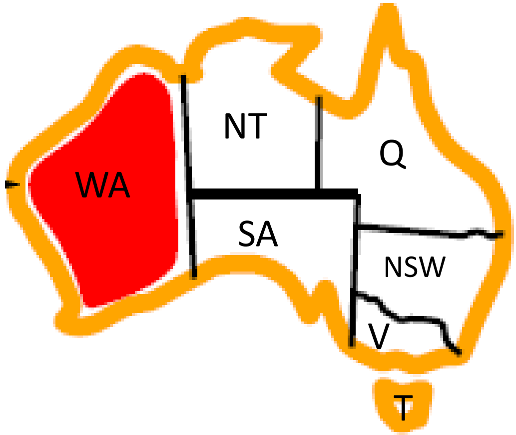
Example of AC-3 4

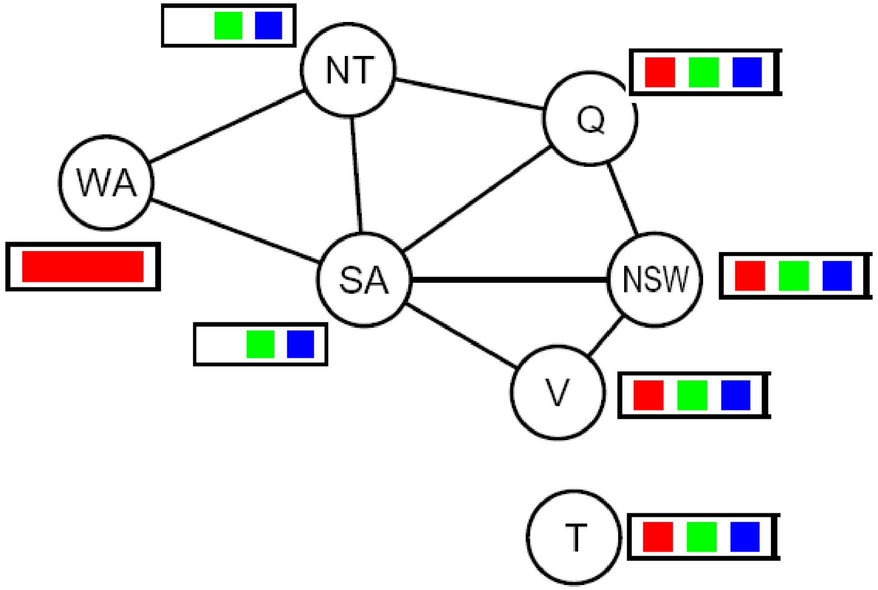




Queue:
~~SA->WA~~
~~NT->WA~~
~~WA->SA~~
NT->SA
Q->SA
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

Example of AC-3 5





Queue:

~~SA->WA~~

~~NT->WA~~

~~WA->SA~~

~~NT->SA~~

~~Q->SA~~

~~NSW->SA~~

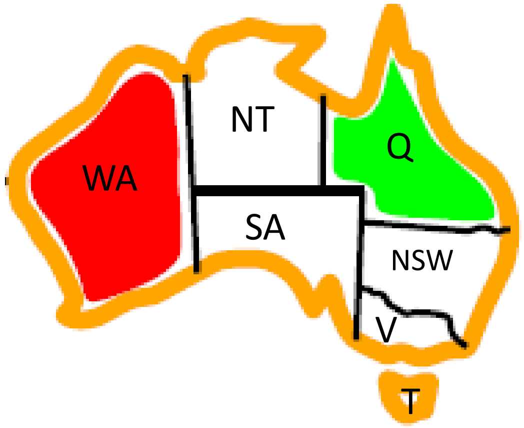
~~V->SA~~

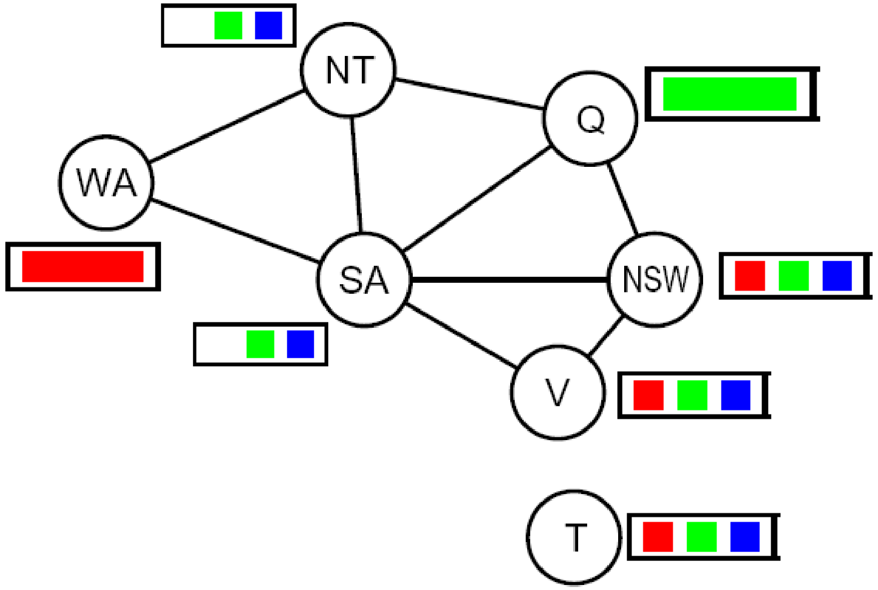
~~WA->NT~~

~~SA->NT~~

~~Q->NT~~

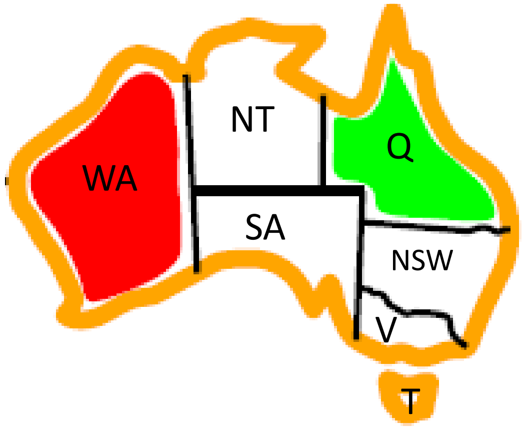
Example of AC-3 6

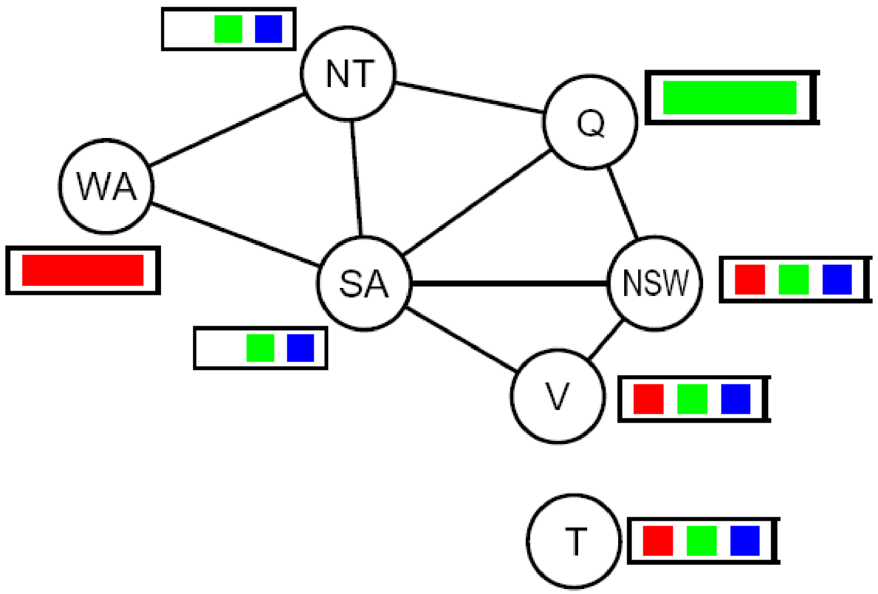




Queue:

Quiz: What would be added to the queue?



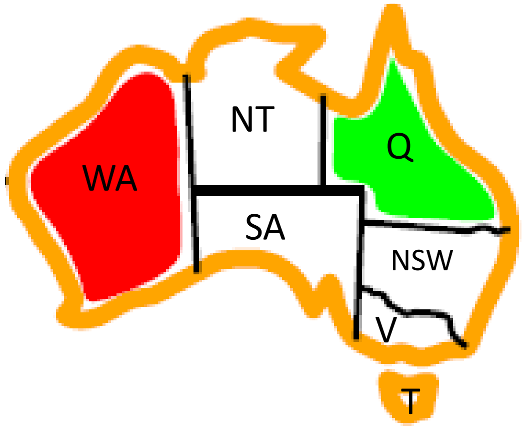


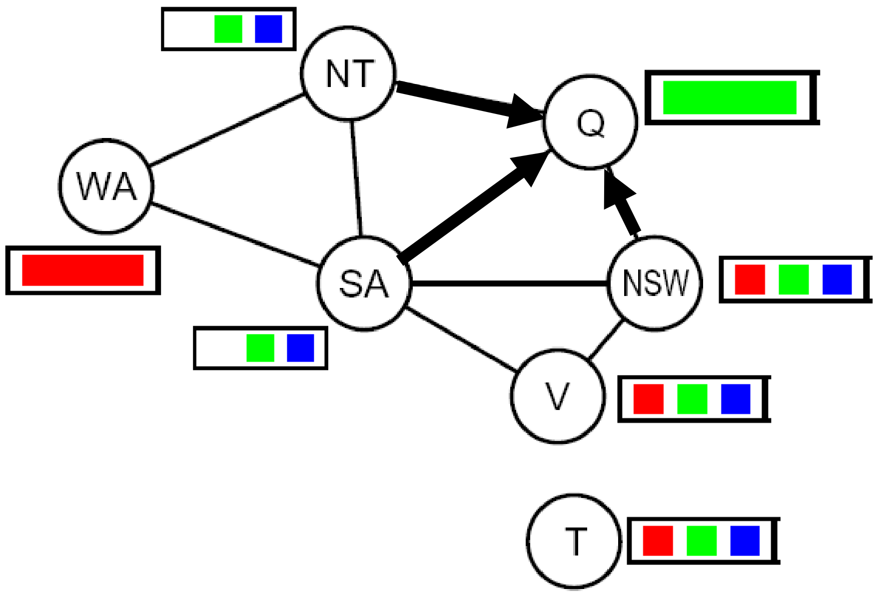
Queue:

A: NSW->Q, SA->Q, NT->Q

B: Q->NSW, Q->SA, Q->NT

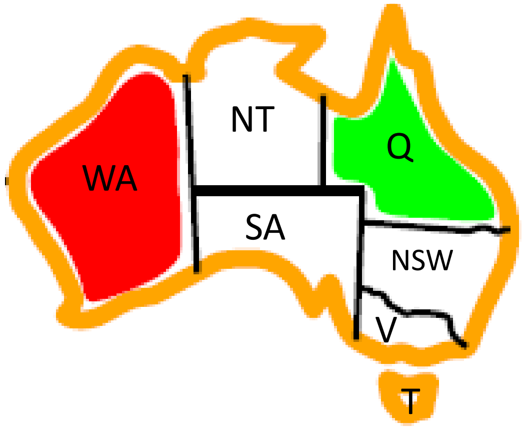
Example of AC-3 7

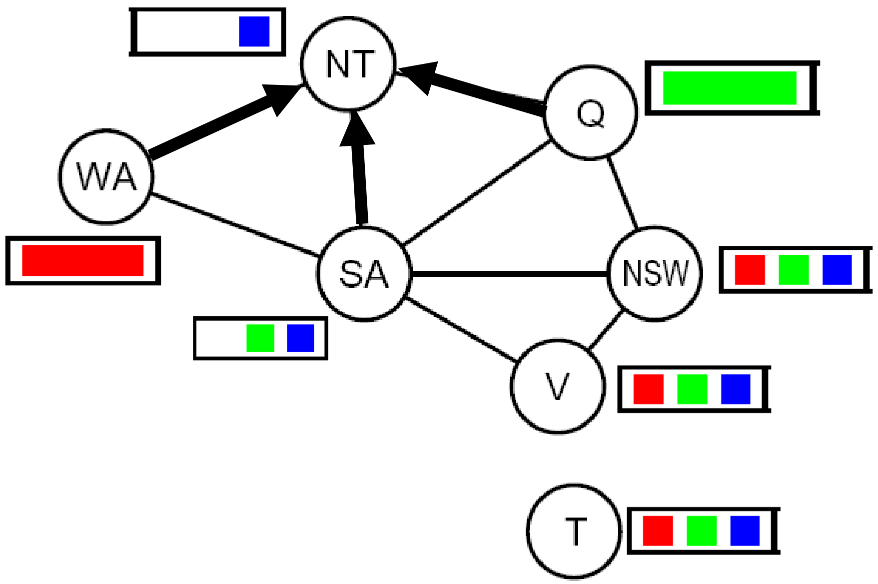




Queue:
NT->Q
SA->Q
NSW->Q

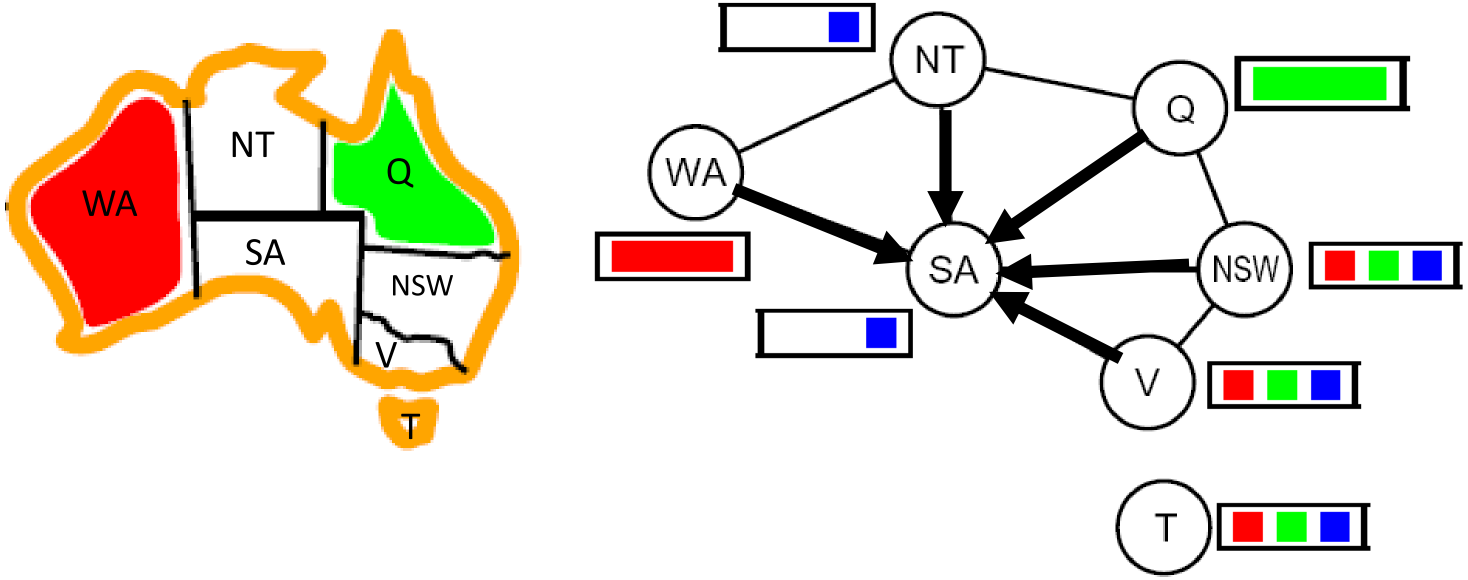
Example of AC-3 8





Queue:
~~NT->Q~~
SA->Q
NSW->Q
WA->NT
SA->NT
Q->NT

Example of AC-3 9



Queue:

~~NT->Q~~

~~SA->Q~~

NSW->Q

WA->NT

SA->NT

Q->NT

WA->SA

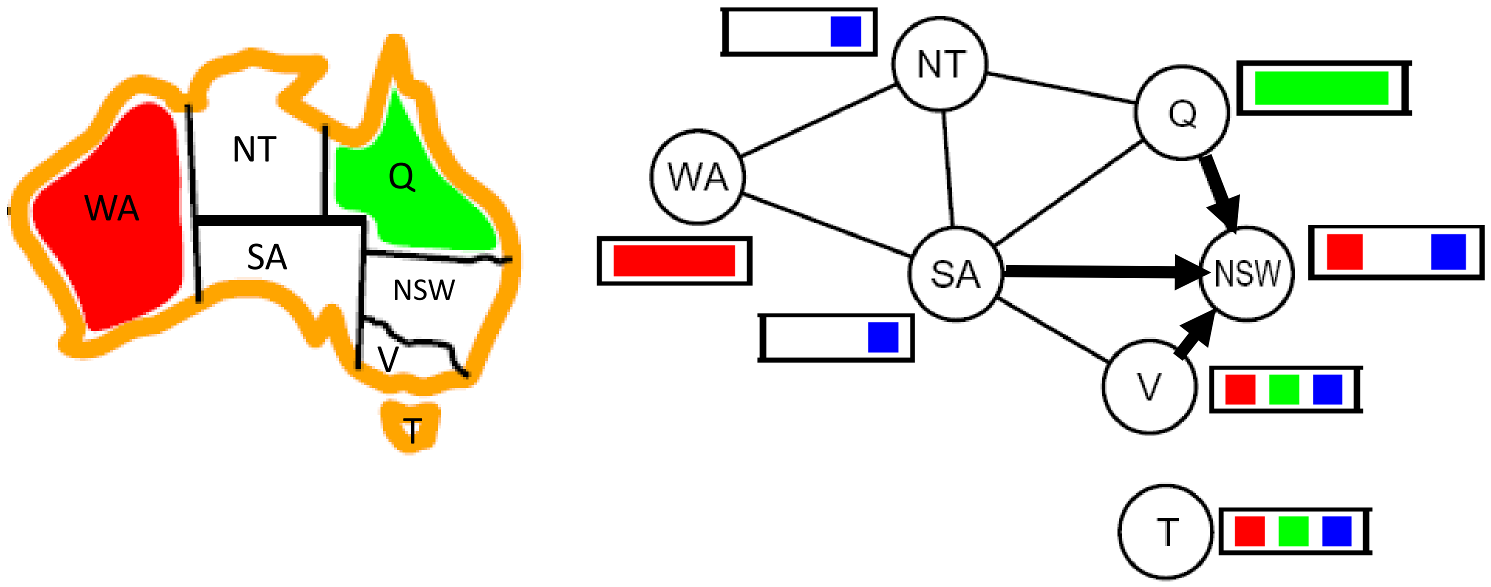
NT->SA

Q->SA

NSW->SA

V->SA

Example of AC-3 10



Queue:

~~NT->Q~~

~~SA->Q~~

~~NSW->Q~~

WA->NT

SA->NT

Q->NT

WA->SA

NT->SA

Q->SA

NSW->SA

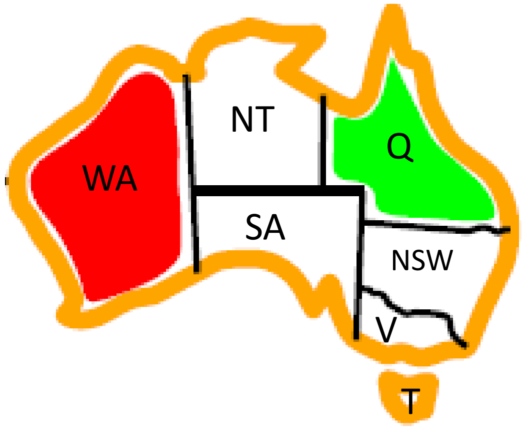
V->SA

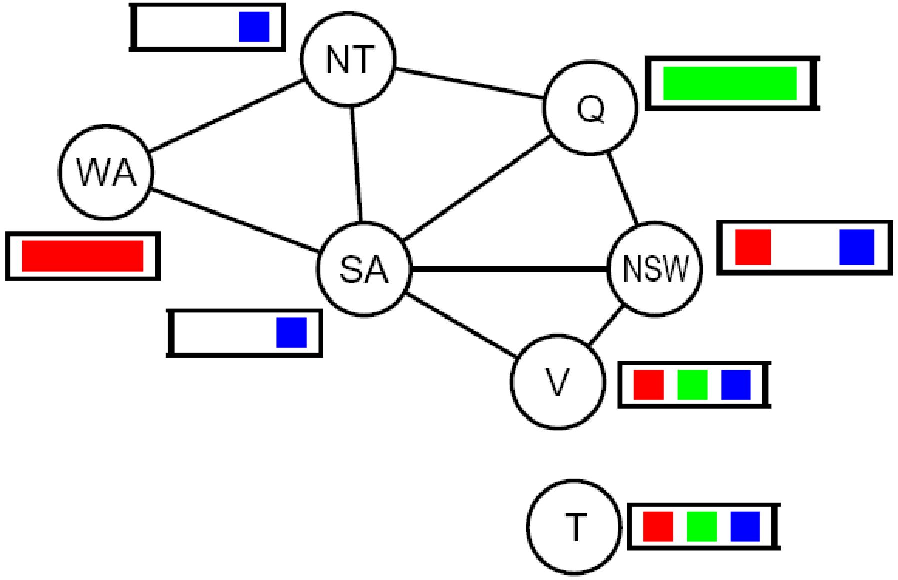
V->NSW

Q->NSW

SA->NSW

Example of AC-3 11





Queue:

~~NT->Q~~

~~SA->Q~~

~~NSW->Q~~

~~WA->NT~~

SA->NT

Q->NT

WA->SA

NT->SA

Q->SA

NSW->SA

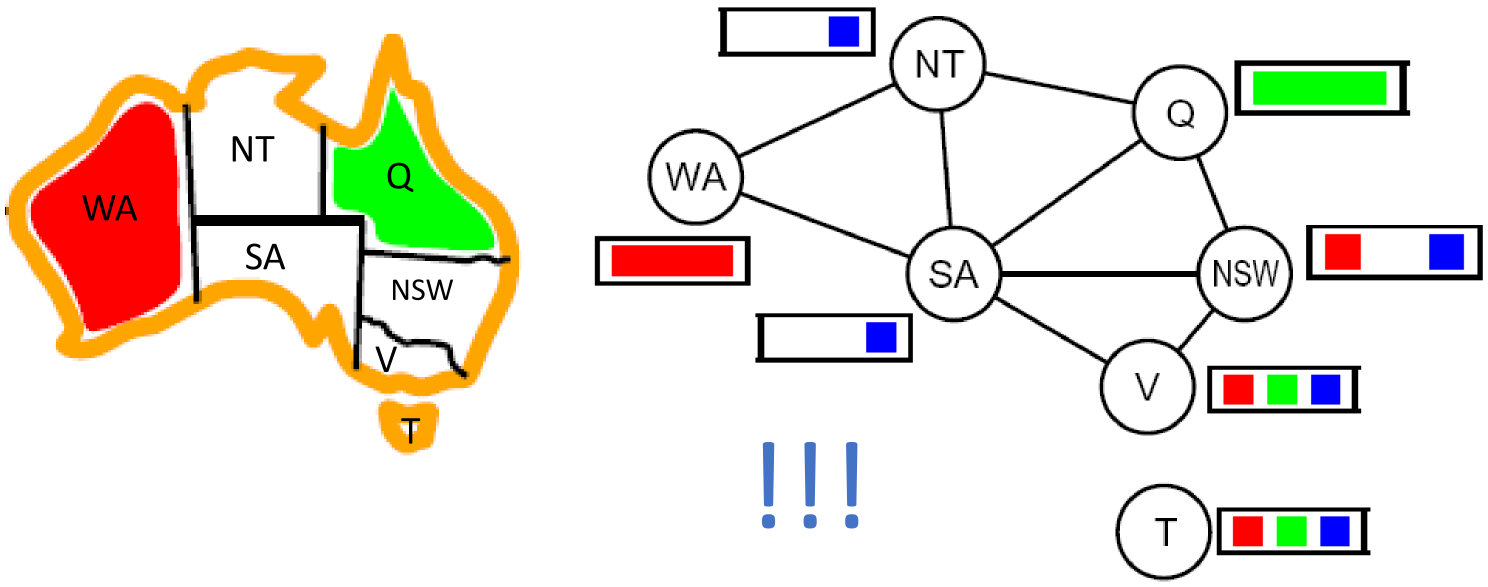
V->SA

V->NSW

Q->NSW

SA->NSW

Example of AC-3 12

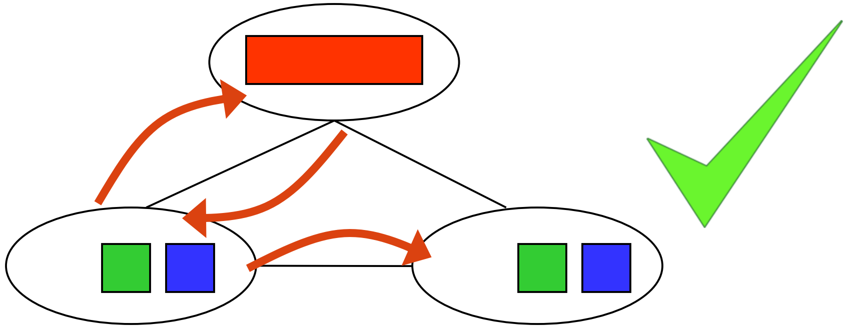


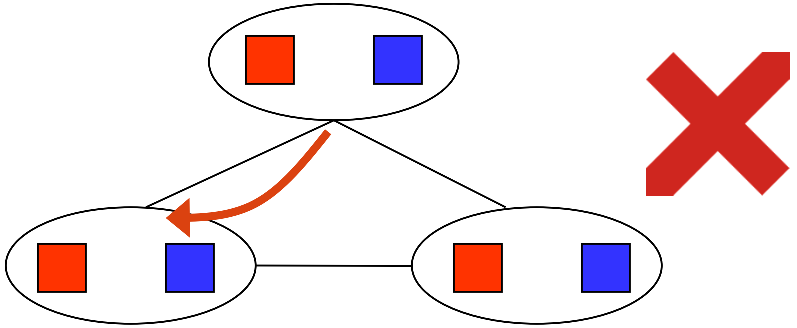
Queue:
~~NT->Q~~
~~SA->Q~~
~~NSW->Q~~
~~WA->NT~~
~~SA->NT~~
Q->NT
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA
V->NSW
Q->NSW
SA->NSW

- Backtrack on the assignment of Q
- Arc consistency detects failure earlier than forward checking

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)





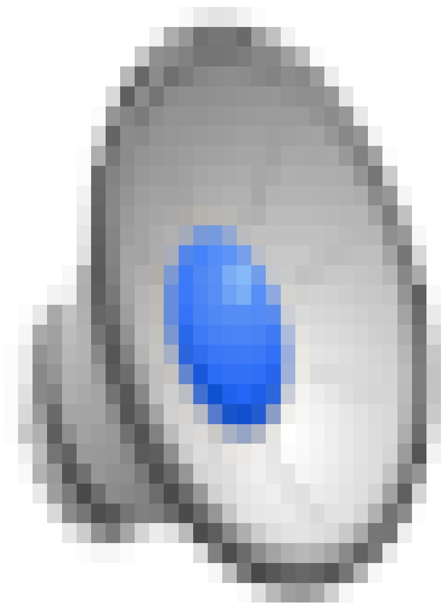
- Arc consistency still runs inside a backtracking search!
- And will be called many times

[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]

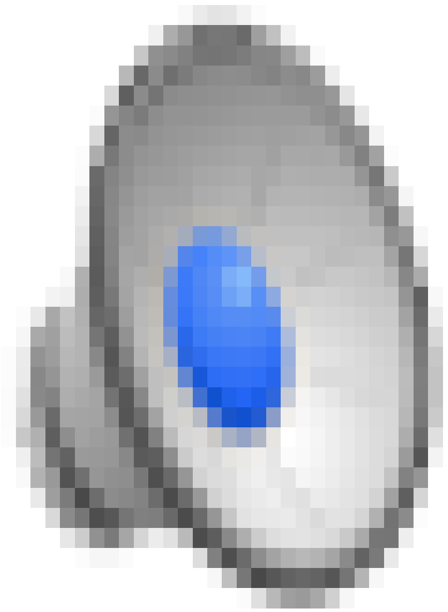
```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return RECURSIVE_BACKTRACKING({}, csp)

function RECURSIVE_BACKTRACKING(assignment, csp) returns a solution, or failure
    if assignment is complete then
        return assignment
    var ⟵ SELECT_UNASSIGNED_VARIABLE(VARIABLES[csp], assignment, 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
                                                                AC-3(csp)
            result ⟵ RECURSIVE_BACKTRACKING(assignment, csp)
            if result ≠ failure, then
                return result
            remove {var=value} from assignment
    return failure
```

Video of Demo Coloring – Backtracking with Forward Checking – Complex Graph

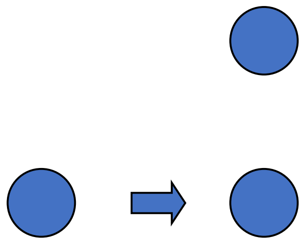


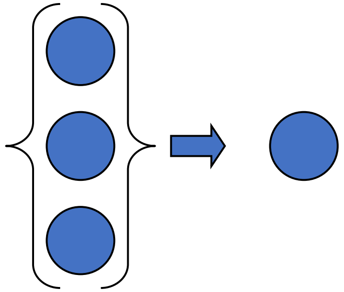
Video of Demo Coloring – Backtracking with Arc Consistency – Complex Graph

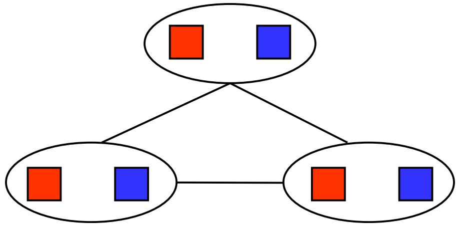


K-Consistency

- Increasing degrees of consistency
  - 1-Consistency (Node Consistency): Each single node's domain has a value which meets that node's unary constraints
  - 2-Consistency (Arc Consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - k-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the kth node.
- Higher k more expensive to compute
- (You need to know the k=2 case: arc consistency)







Strong K-Consistency

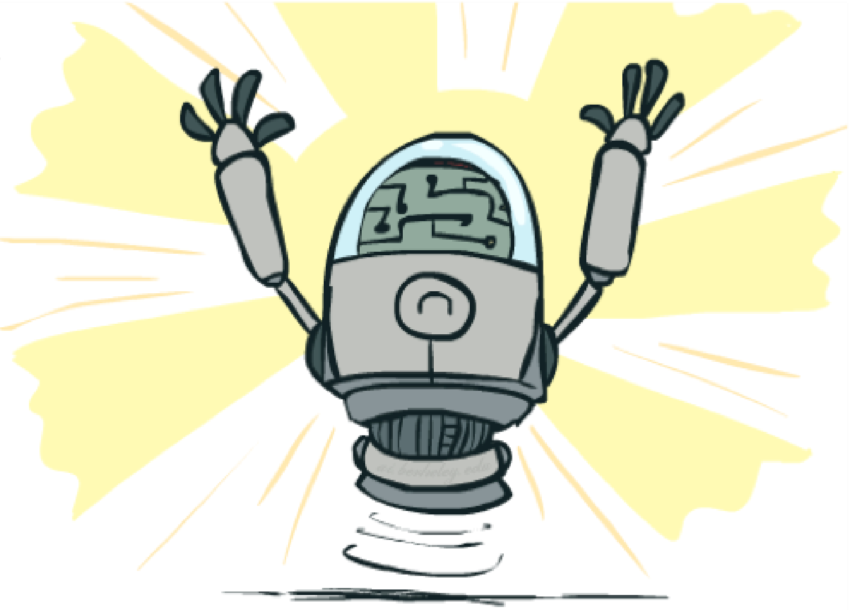
- Strong k-consistency: also k-1, k-2, … 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 2
  - …

- Lots of middle ground between arc consistency and n-consistency! (e.g. k=3, called path consistency)

Ordering

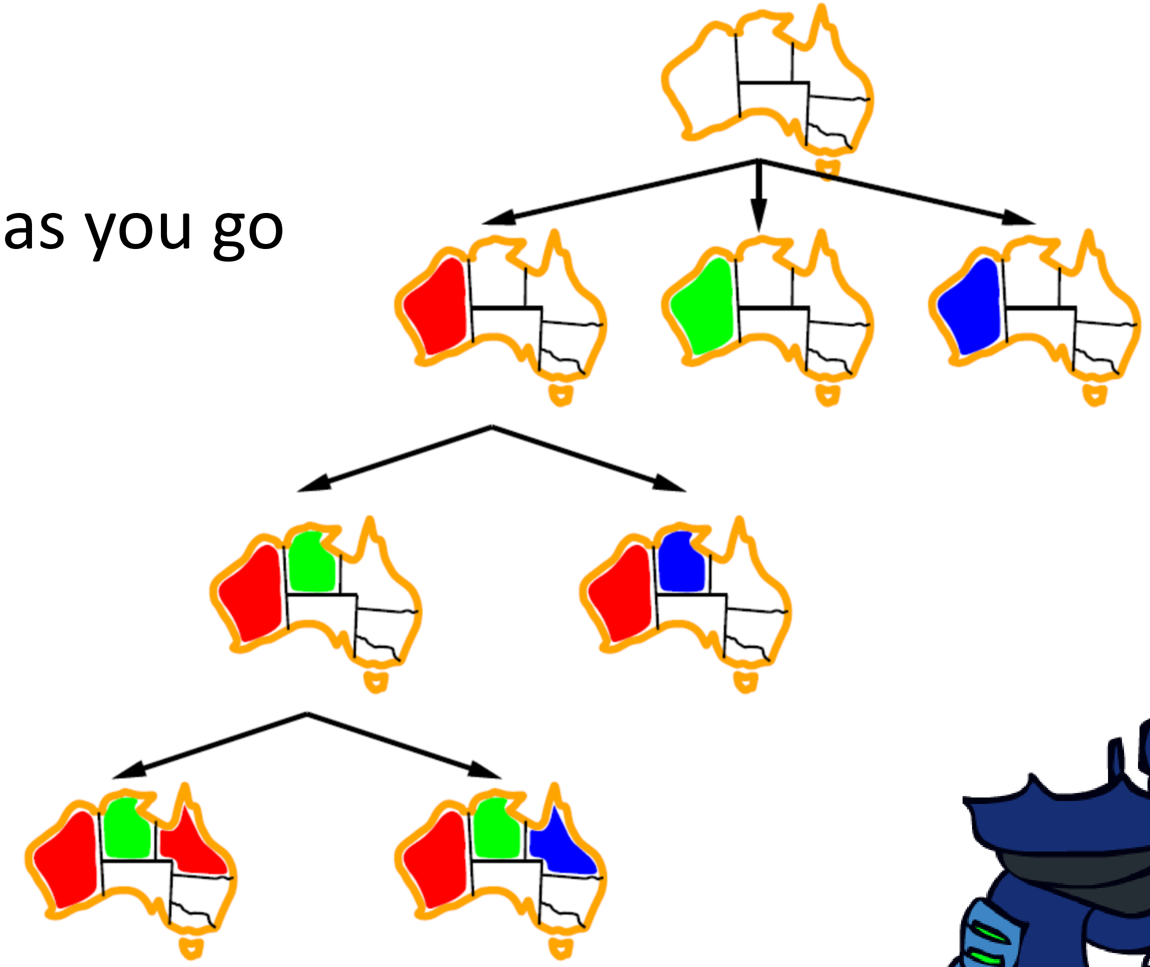
Improving Backtracking

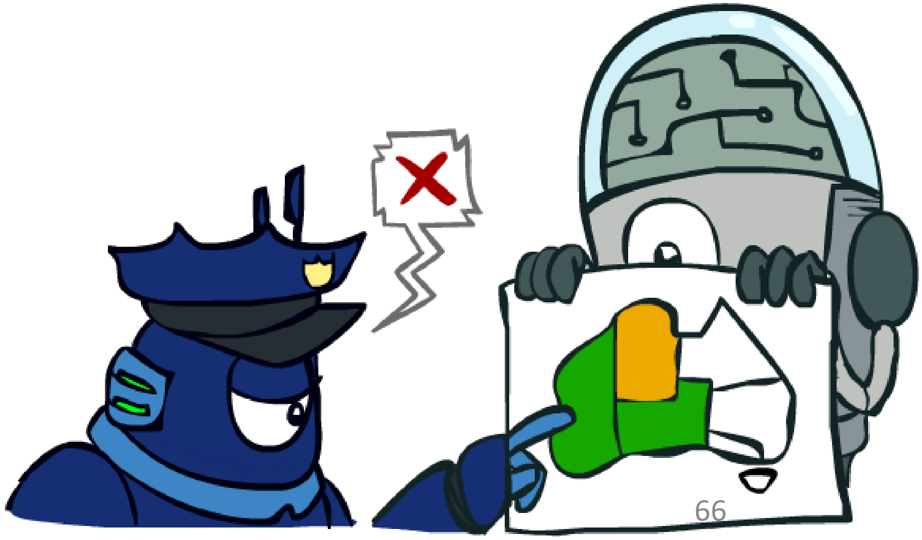
- General-purpose ideas give huge gains in speed
- Filtering: Can we detect inevitable failure early?
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Structure: Can we exploit the problem structure?



Backtracking Search

- fix ordering
- check constraints as you go



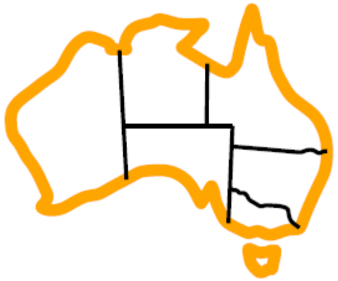


Quiz

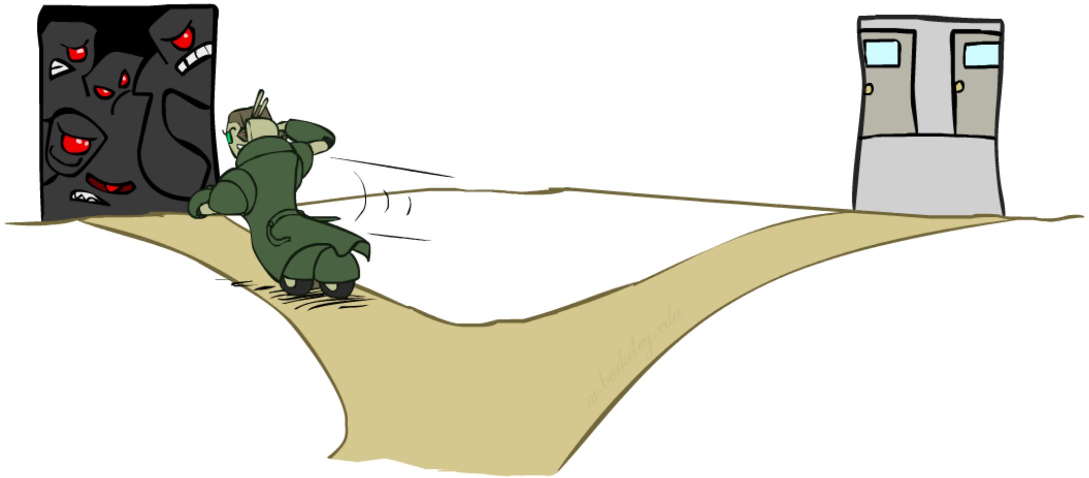
- What is good/bad to fix the ordering of variables?
- What is good/bad to fix the ordering of values?

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

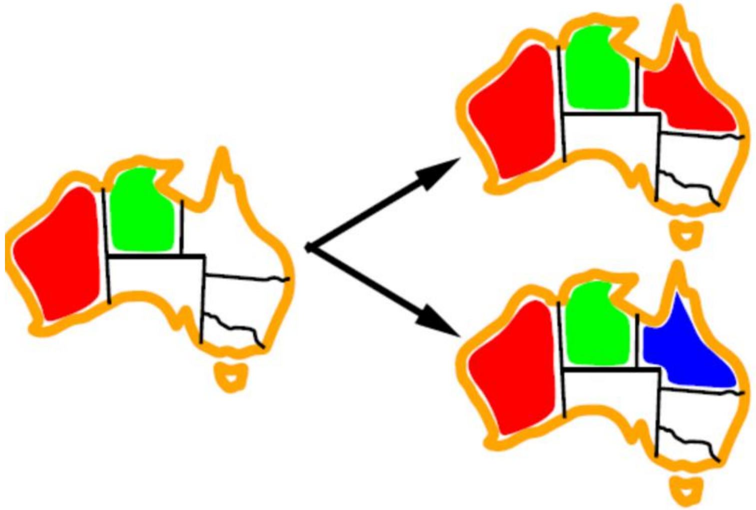


Demo: Coloring -- Backtracking + Forward Checking + Ordering

- Backtracking + Forward Checking + Minimum Remaining Values (MRV)

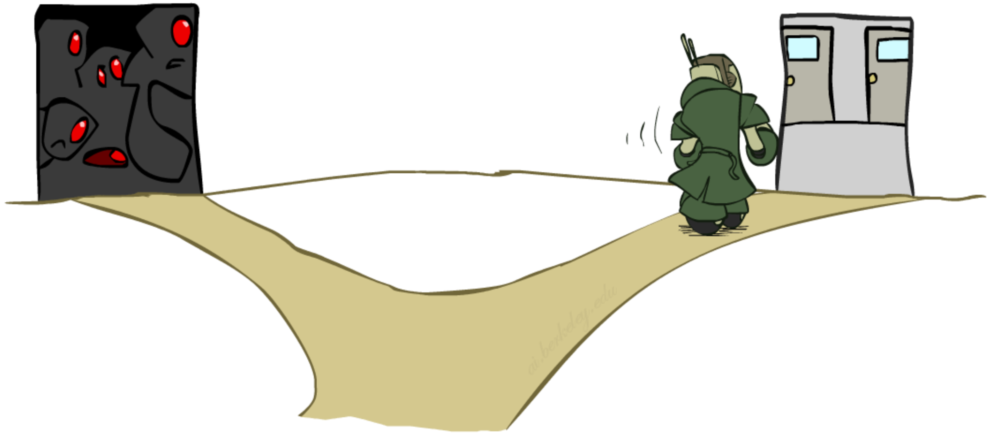
Ordering: Least Constraining Value

- 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



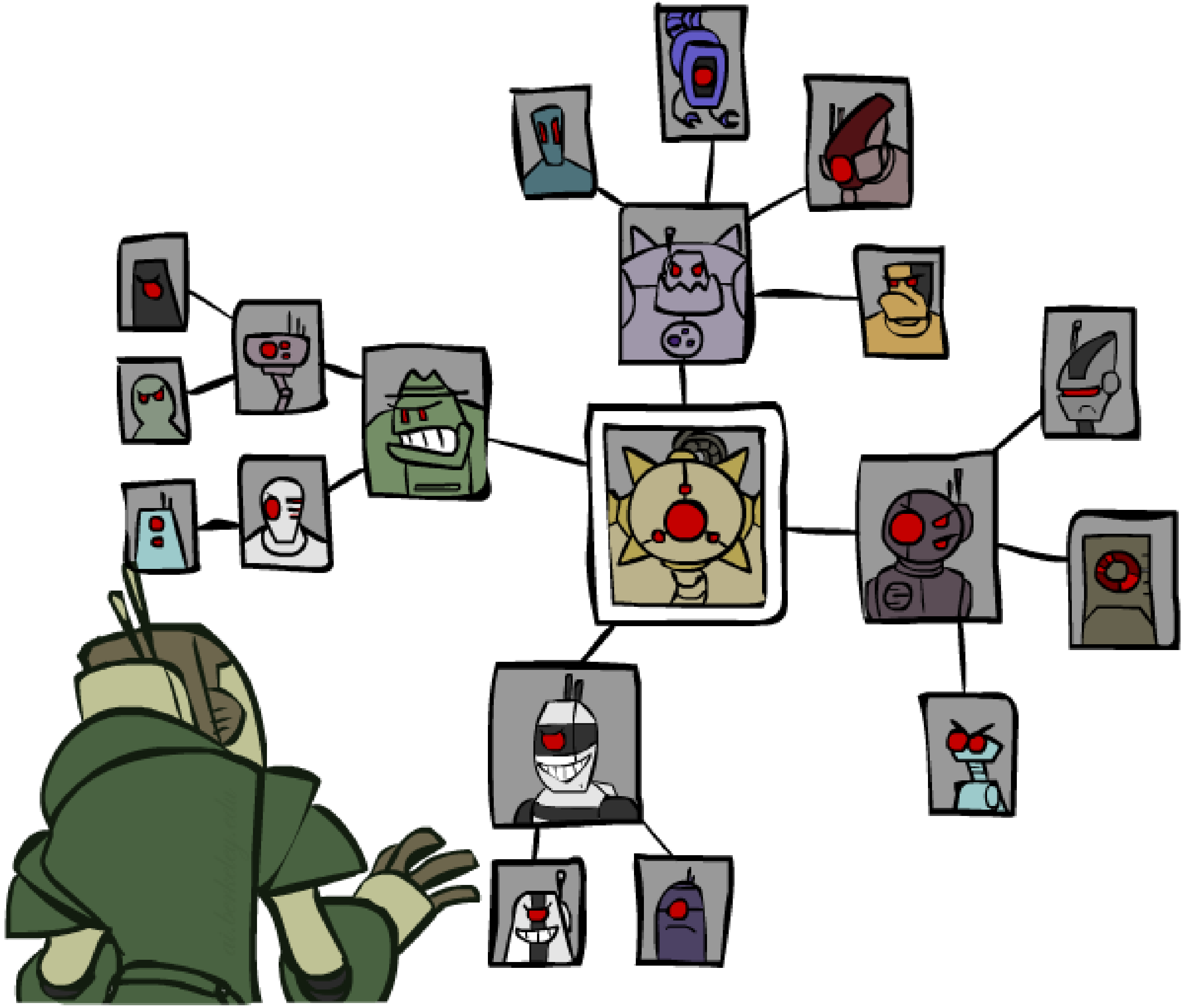
Demo: Coloring -- Backtracking + Forward Checking + Ordering

- Backtracking + AC-3 + MRV + LCV

Quiz

- How we order variables and why
- How we order values and why
- Why different on variables and values

Structure



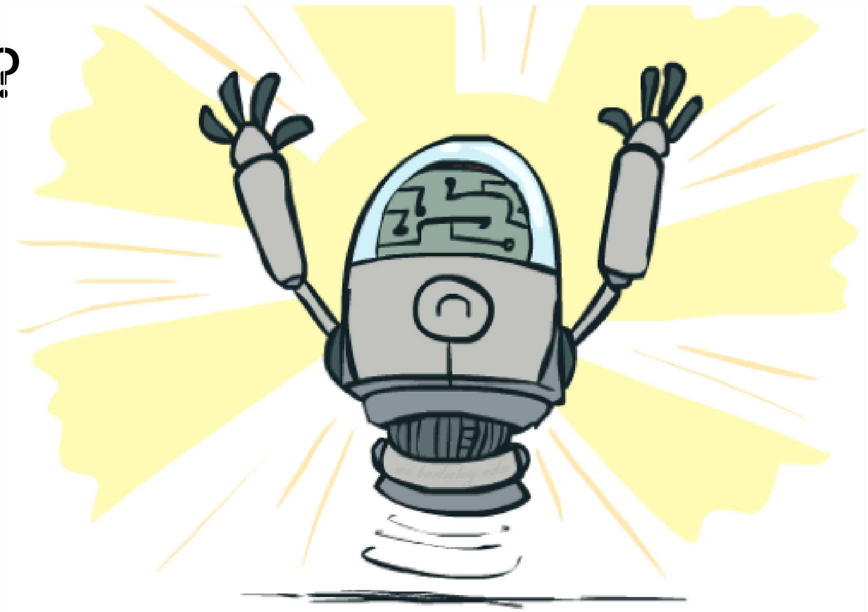
Improving Backtracking

- General-purpose ideas give huge gains in speed

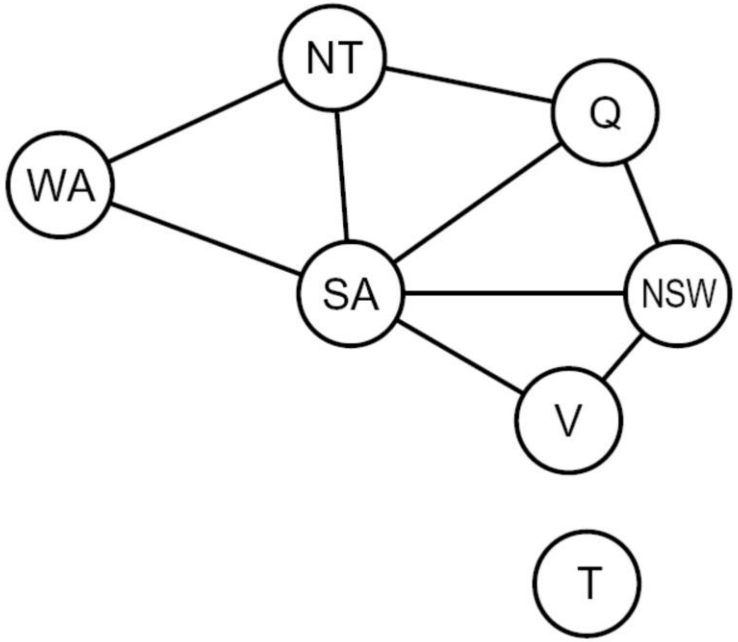
- Filtering: Can we detect inevitable failure early?

- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

- Structure: Can we exploit the problem structure?



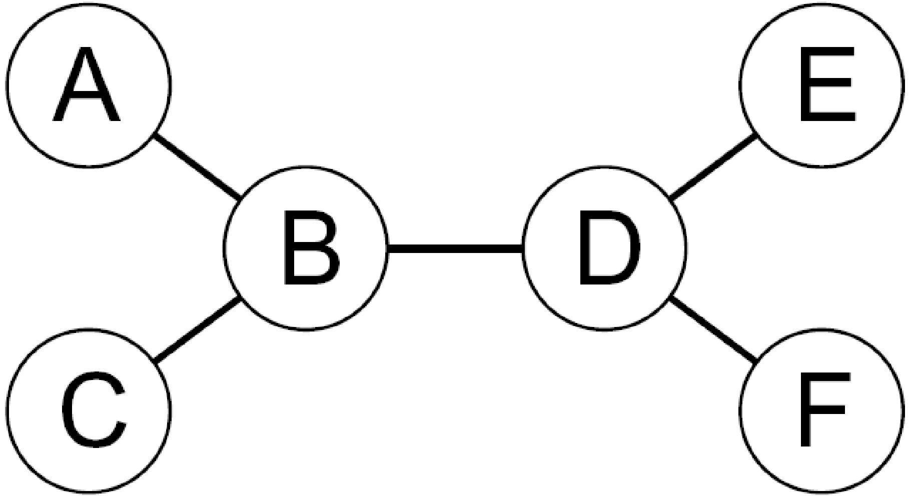
Problem Structure



- For general CSPs, worst-case complexity with backtracking algorithm is $O(d^n)$
- When the problem has special structure, we can often solve the problem more efficiently

- Special Structure 1: Independent subproblems
  - Example: Tasmania and mainland do not interact
  - Connected components of constraint graph
  - Suppose a graph of n variables can be broken into subproblems, each of only c variables:
    - Worst-case complexity is $O((n/c)(d^c))$, linear in n
    - E.g., n = 80, d = 2, c =20
    - 2^{80} = 4 billion years at 10 million nodes/sec
    - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec

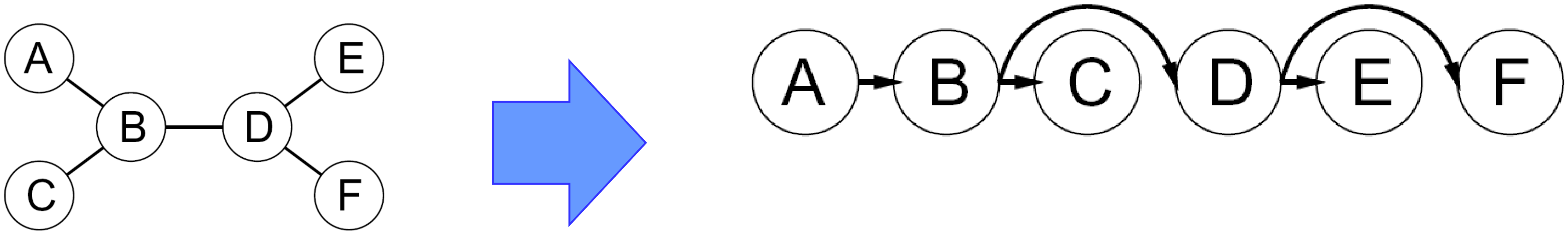
Tree-Structured CSPs



- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$
  - How?
- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

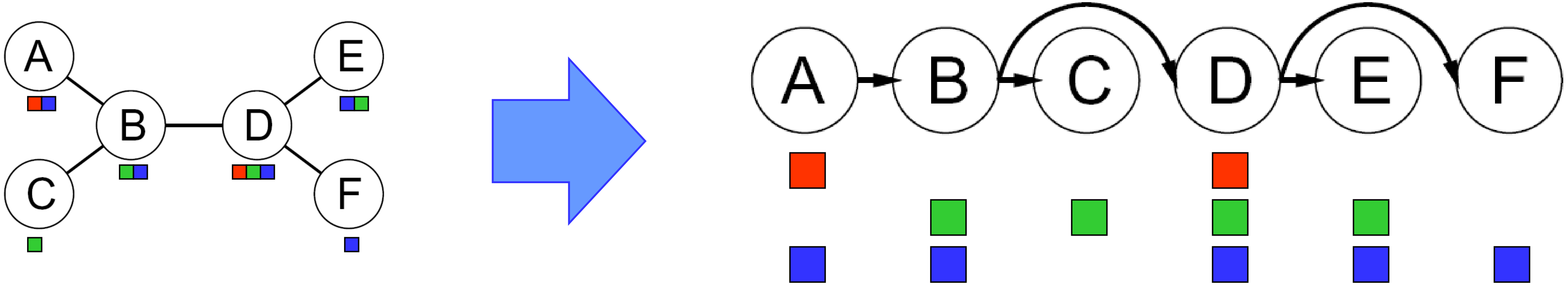
Tree-Structured CSPs 2

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



Tree-Structured CSPs 3

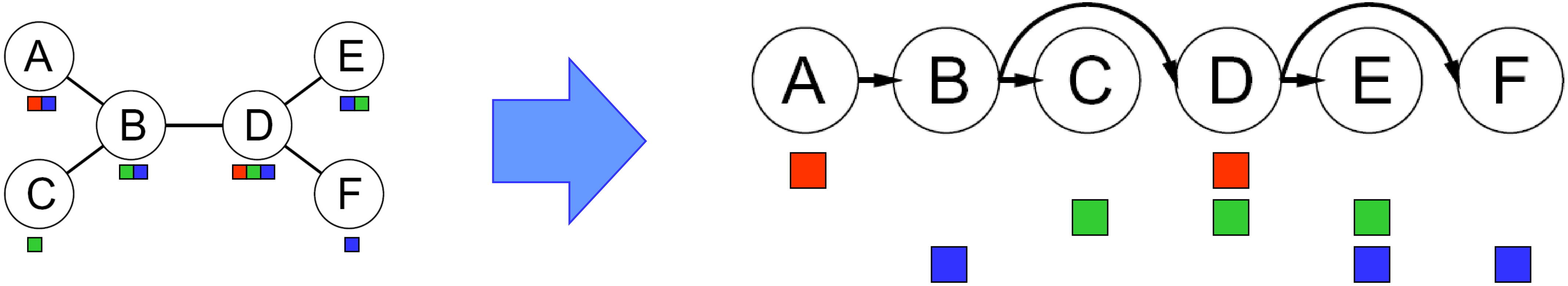
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - Remove backward: For $i = n: 2$, apply RemoveInconsistent(Parent(X_i),X_i)

Tree-Structured CSPs 4

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



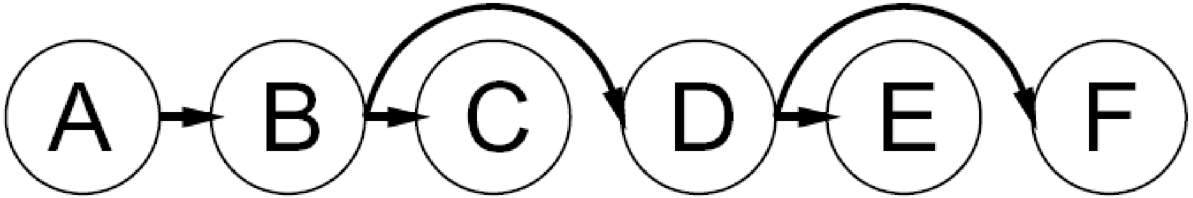
  - Remove backward: For $i = n: 2$, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For $i = 1: n$, assign X_i consistently with Parent(X_i)

Remove backward $O(nd^2)$: $O(d^2)$ per arc and $O(n)$ arcs
Assign forward $O(nd)$: $O(d)$ per node and $O(n)$ nodes

- Runtime: $O(nd^2)$ (why?)
- Can always find a solution when there is one (why?)

Tree-Structured CSPs 5

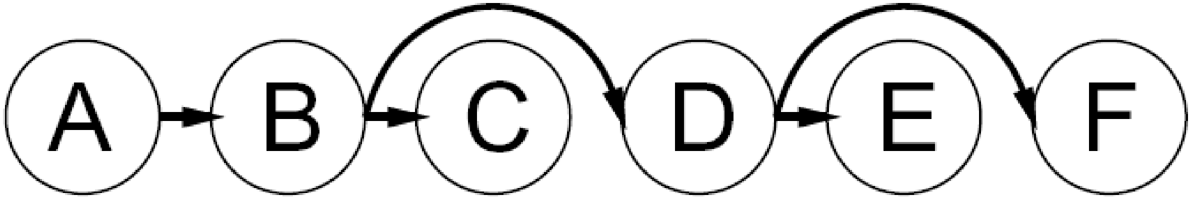
- Remove backward: For $i = n: 2$, apply RemoveInconsistent(Parent(X_i),X_i)



- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: During backward pass, every node except the root node was “visited” once
  - a. $\text{Parent}(X_i) \rightarrow X_i$ was made consistent when X_i was visited
  - b. After that, $\text{Parent}(X_i) \rightarrow X_i$ kept consistent until the end of the backward pass

Tree-Structured CSPs 6

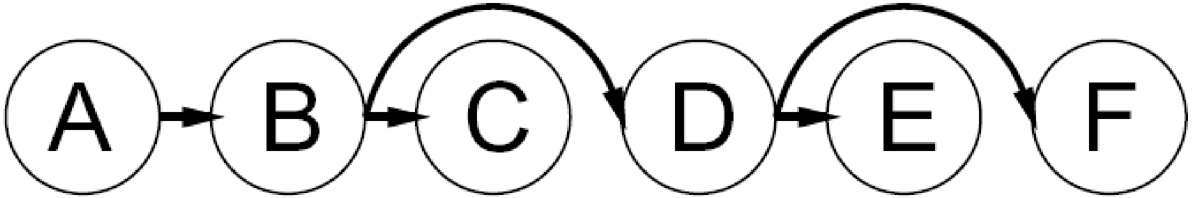
- Remove backward: For $i = n: 2$, apply RemoveInconsistent(Parent(X_i),X_i)



- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: During backward pass, every node except the root node was "visited" once
  - a. Parent$(X_i) \rightarrow X_i$ was made consistent when X_i was visited
    - When X_i was visited, we enforced arc consistency of Parent$(X_i) \rightarrow X_i$ by reducing the domain of Parent(X_i). By definition, for every value in the reduced domain of Parent(X_i), there was some x in the domain of X_i which could be assigned without violating the constraint involving Parent(X_i) and X_i
  - b. After that, Parent$(X_i) \rightarrow X_i$ kept consistent until the end of the backward pass

Tree-Structured CSPs 7

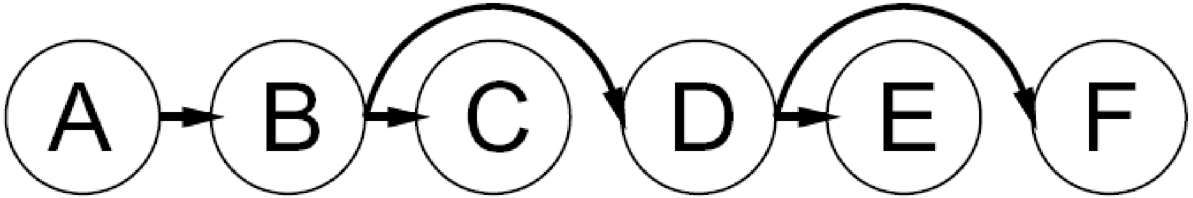
- Remove backward: For $i = n{:}\,2$, apply RemoveInconsistent(Parent(X_i),X_i)



- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: During backward pass, every node except the root node was "visited" once.
  - a. Parent$(X_i) \rightarrow X_i$ was made consistent when X_i was visited
  - b. After that, Parent$(X_i) \rightarrow X_i$ kept consistent until the end of the backward pass
    - Domain of X_i would not have been reduced after X_i is visited because X_i's children were visited before X_i. Domain of Parent(X_i) could have been reduced further. Arc consistency would still hold by definition.

Tree-Structured CSPs 8

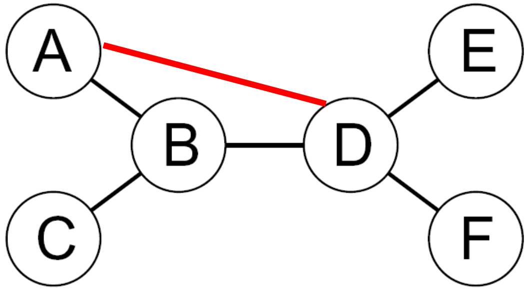
- Assign forward: For i=1:n, assign X_i consistently with Parent(X_i)

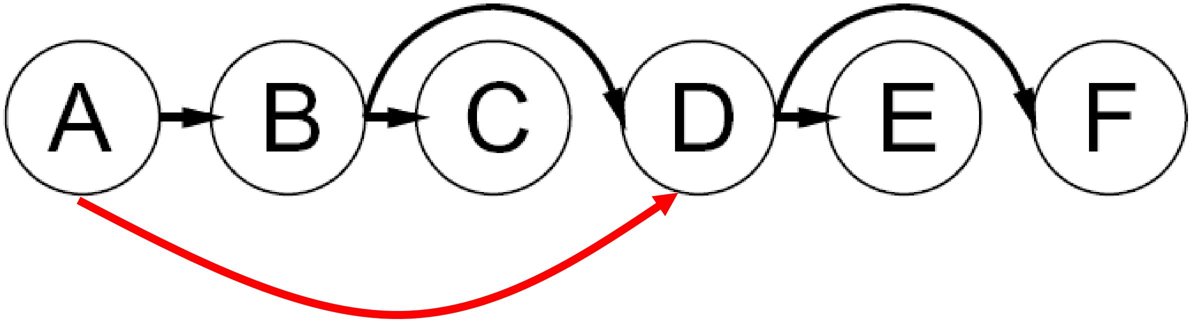


- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Follow the backtracking algorithm (on the reduced domains and with the same ordering). Induction on position Suppose we have successfully reached node X_i. In the current step, the potential failure can only be caused by the constraint between X_i and Parent(X_i), since all other variables that are in a same constraint of X_i have not assigned a value yet. Due to the arc consistency of Parent$(X_i) \to X_i$, there exists a value x in the domain of X_i that does not violate the constraint. So we can successfully assign value to X_i and go to the next node. By induction, we can successfully assign a value to a variable in each step of the algorithm. A solution is found in the end.

What if there are cycles

- Why doesn't this algorithm work with cycles in the constraint graph?



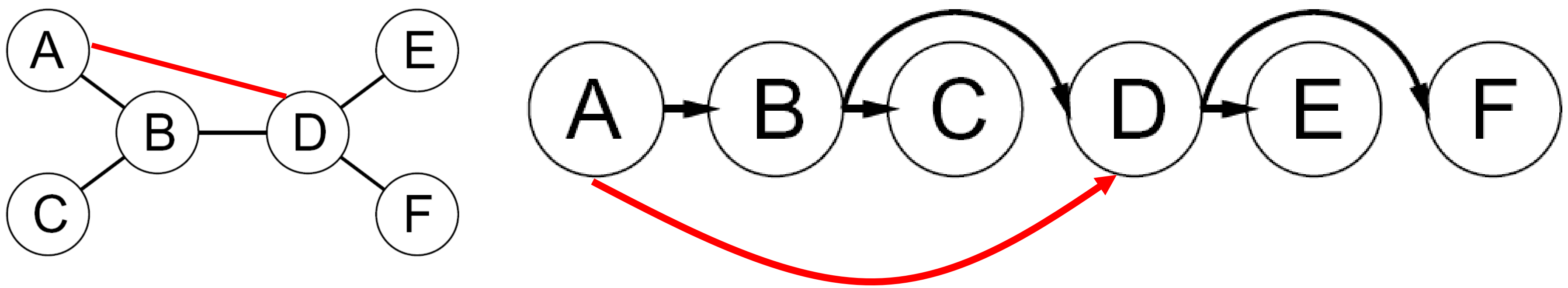


- We can still apply the algorithm (choose an arbitrary order and draw "forward" arcs).
- For remove backward, what would happen?
- For assign forward, what would happen?

Note: We'll see a similar idea with Bayes' nets in later lectures

What if there are cycles 2

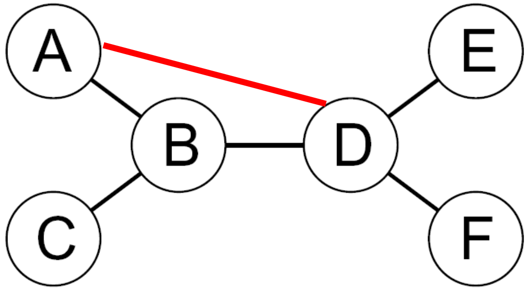
- Why doesn't this algorithm work with cycles in the constraint graph?

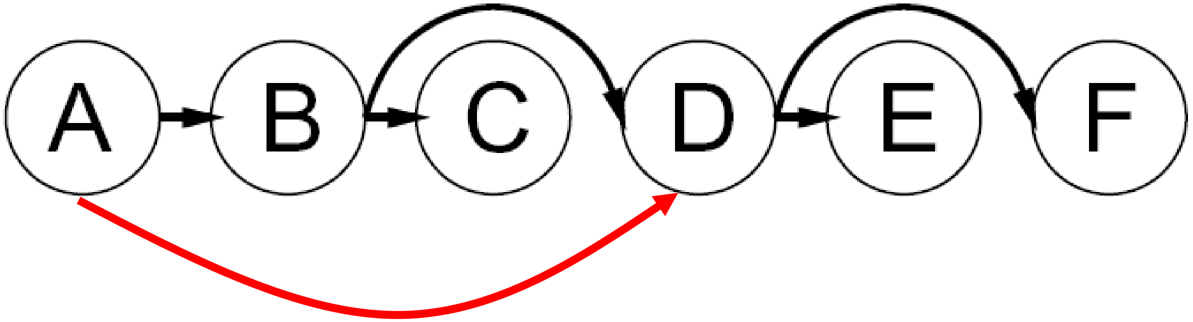


- We can still apply the algorithm (choose an arbitrary order and draw "forward" arcs).
- For remove backward, what would happen?
- We can enforce all arcs pointing to X_i when X_i is visited. The complexity is $O(n^2d^2)$. After backward pass, the reduced domains do not exclude any solution and all the forward arcs are consistent
- For assign forward, what would happen?

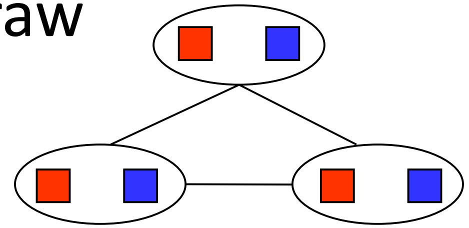
What if there are cycles 3

- Why doesn't this algorithm work with cycles in the constraint graph?

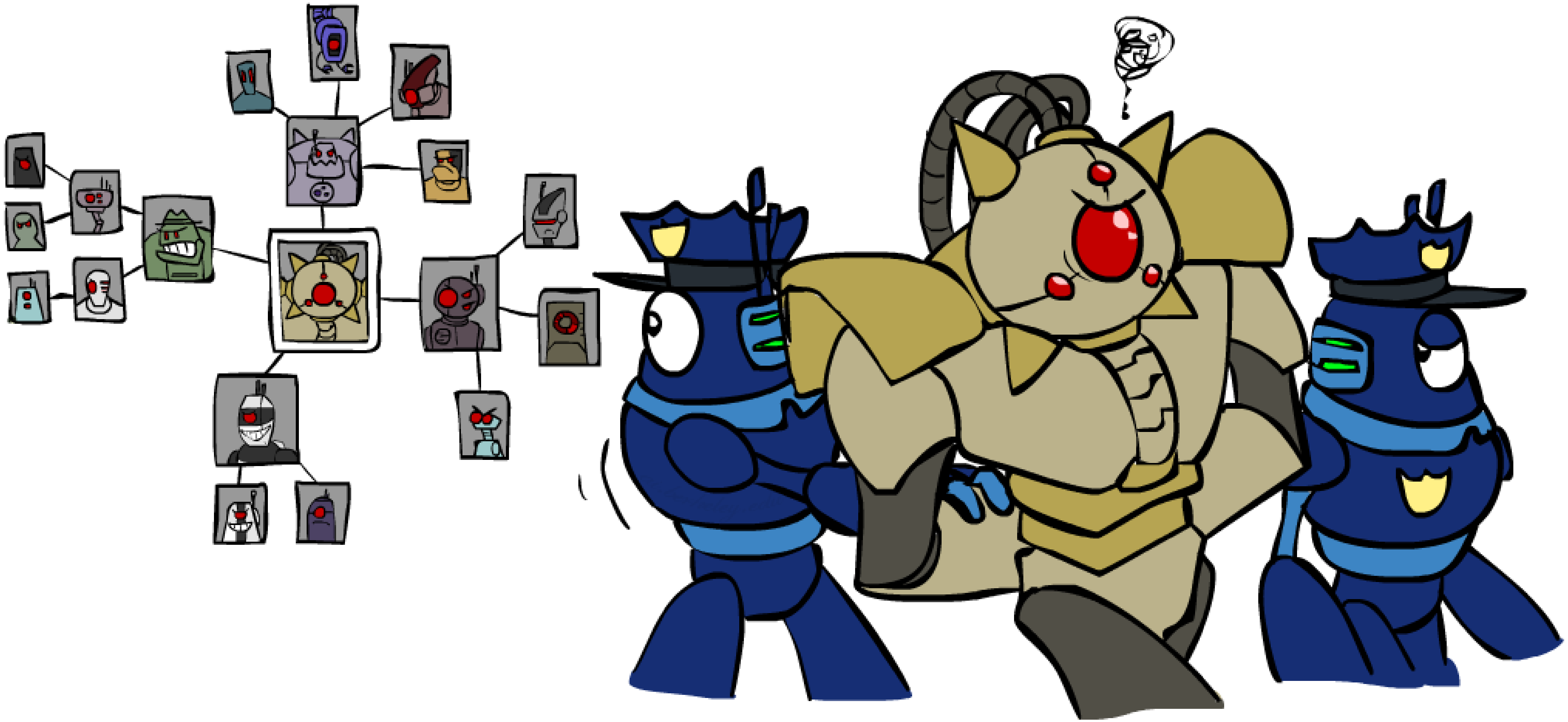




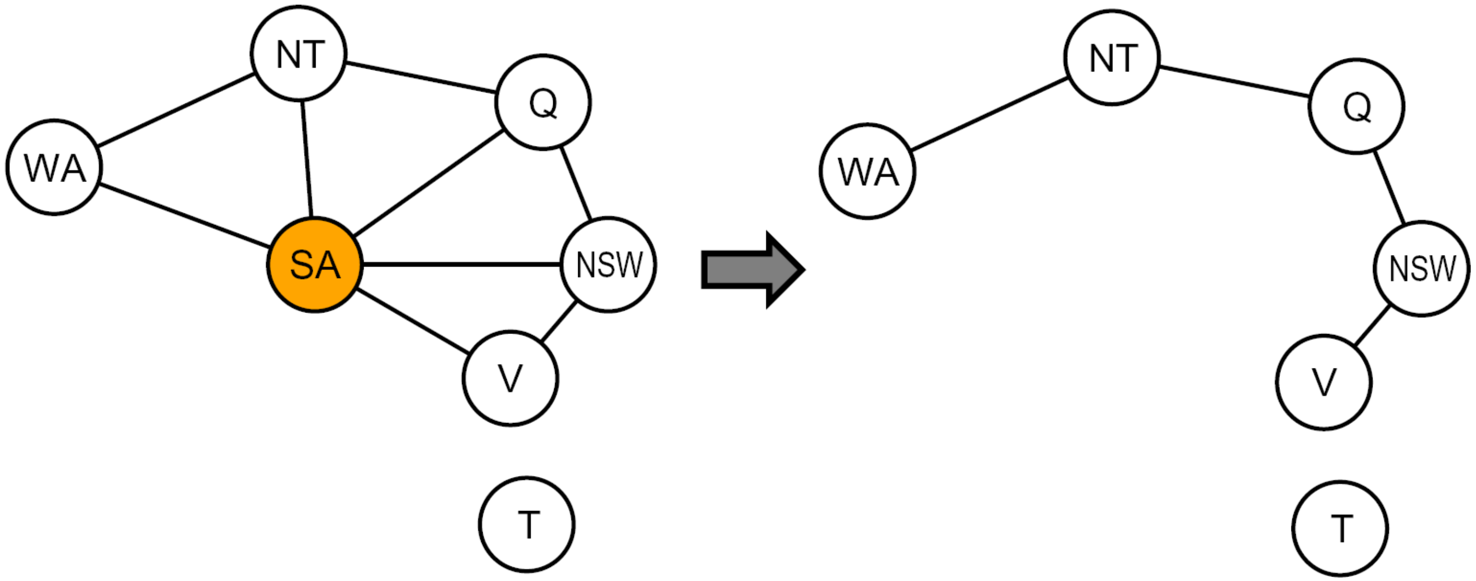
- We can still apply the algorithm (choose an arbitrary order and draw "forward" arcs).
- For remove backward, what would happen?
- For assign forward, what would happen?
- In a step of assigning values, we may encounter failure because we need to make sure the constraints involving the current node and any parent node is satisfied, which could be impossible. Therefore, we may need to backtrack.



Improving Structure



Nearly Tree-Structured CSPs



- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c gives runtime $O((d^c) (n\text{-}c) d^2)$, very fast for small c

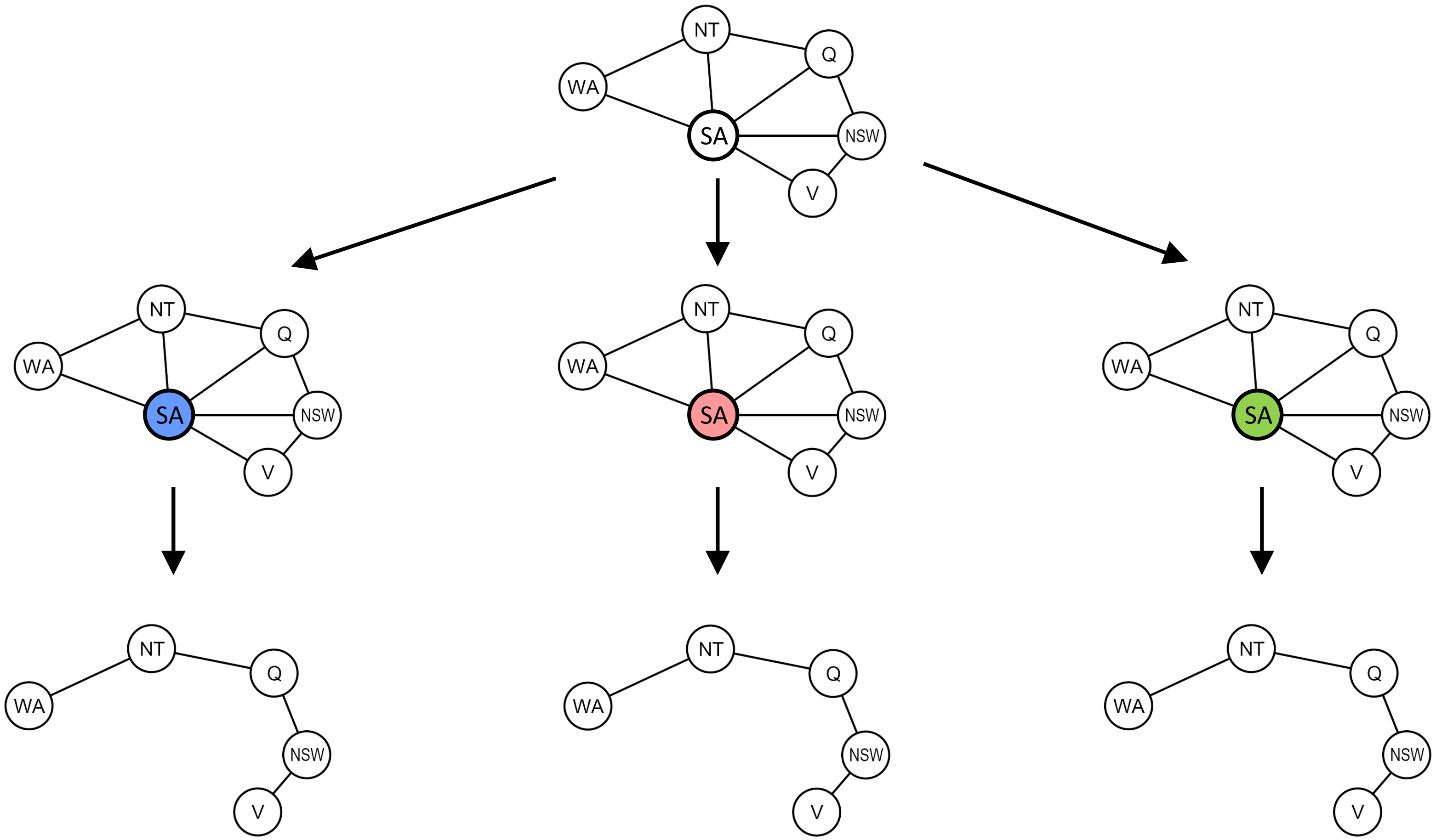
Cutset Conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

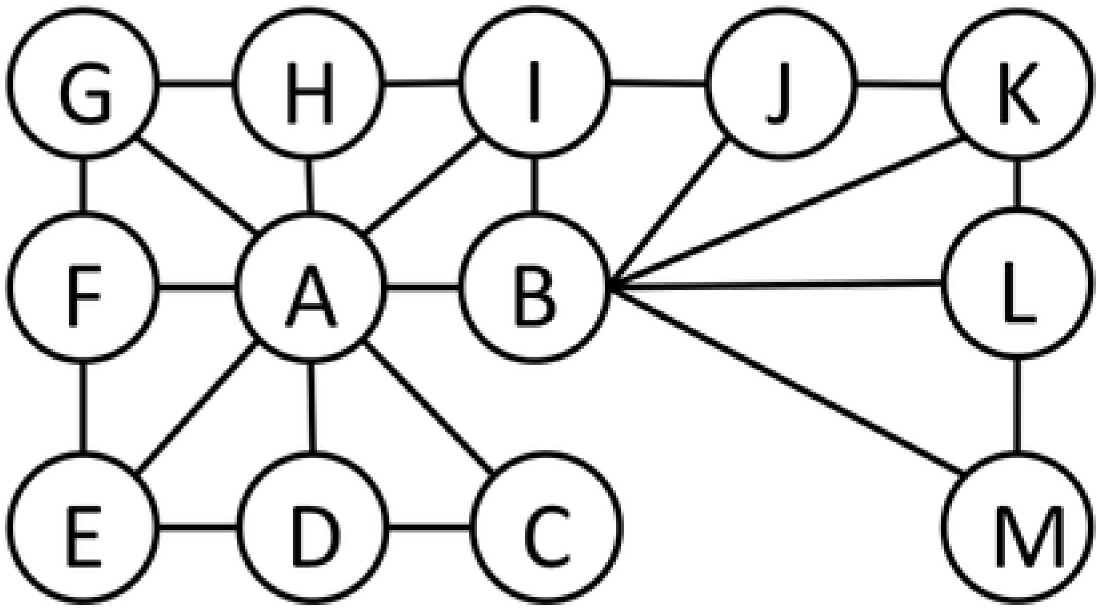
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)

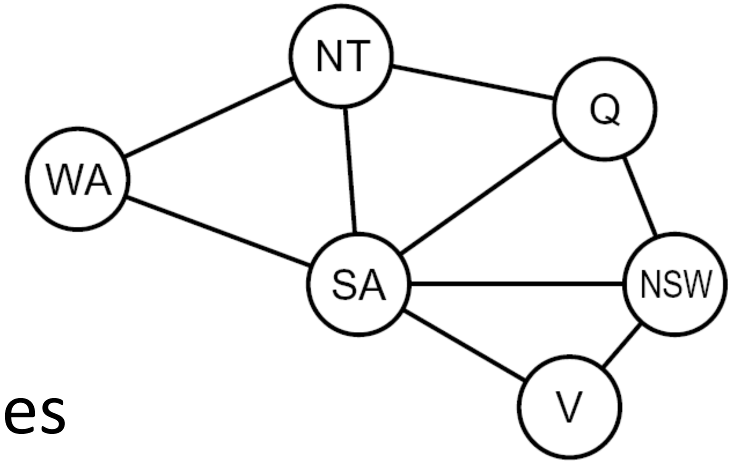


Quiz

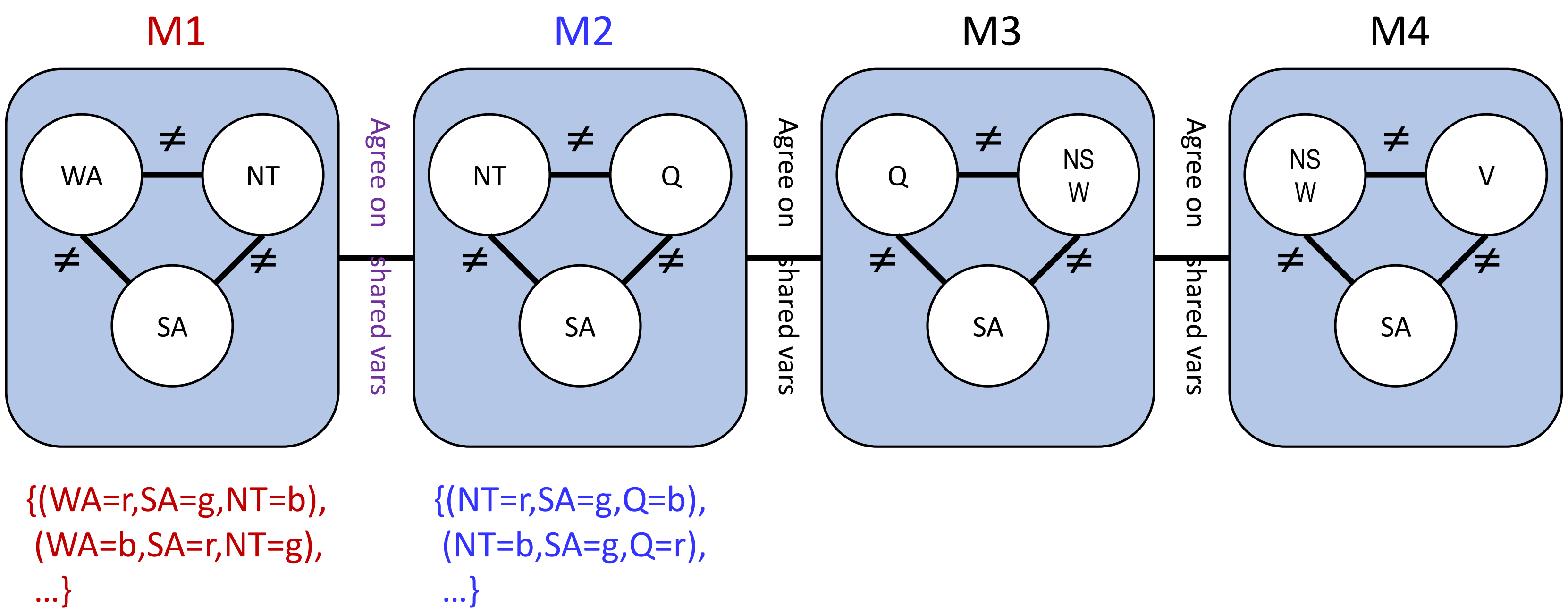
- Find the smallest cutset for the graph below



Tree Decomposition



- Idea: create a tree-structured graph of mega-variables
- Each mega-variable encodes part of the original CSP
- Subproblems overlap to ensure consistent solutions



Non-binary CSPs

Example: Cryptarithmetic

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

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

- Constraints:

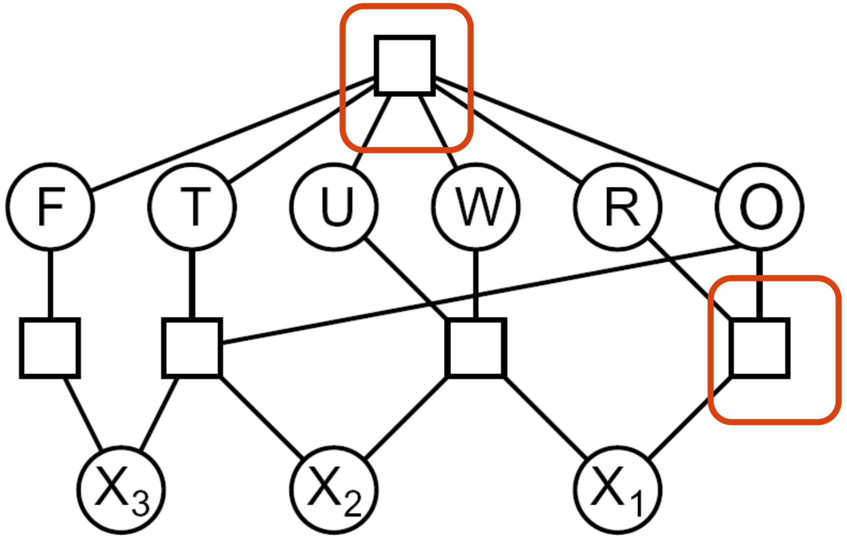
  $\boxed{\text{alldiff}(F, T, U, W, R, O)}$

  $\boxed{O + O = R + 10 \cdot X_1}$

  $\cdots$

X_1

```
    T W O
+   T W O
---------
  F O U R
```



Constraint graph for non-binary CSPs

- Variable nodes: nodes to represent the variables
- Constraint nodes: auxiliary nodes to represent the constraints
- Edges: connects a constraint node and its corresponding variables

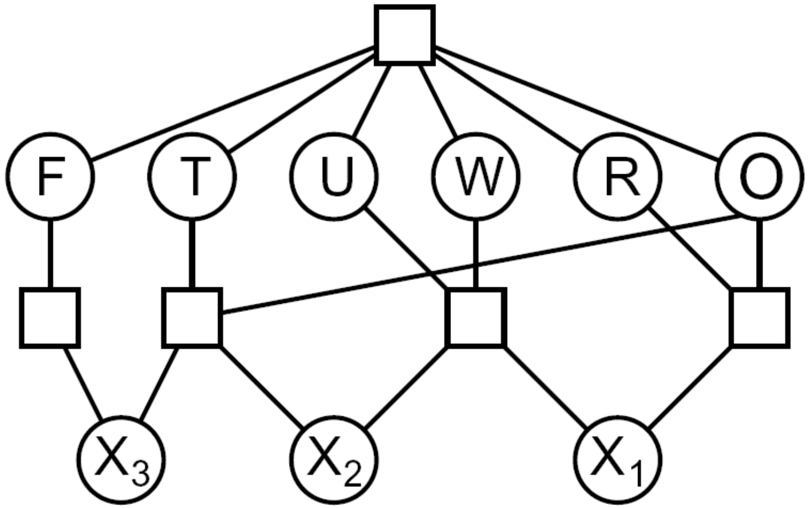
- Constraints:

  alldiff(F, T, U, W, R, O)

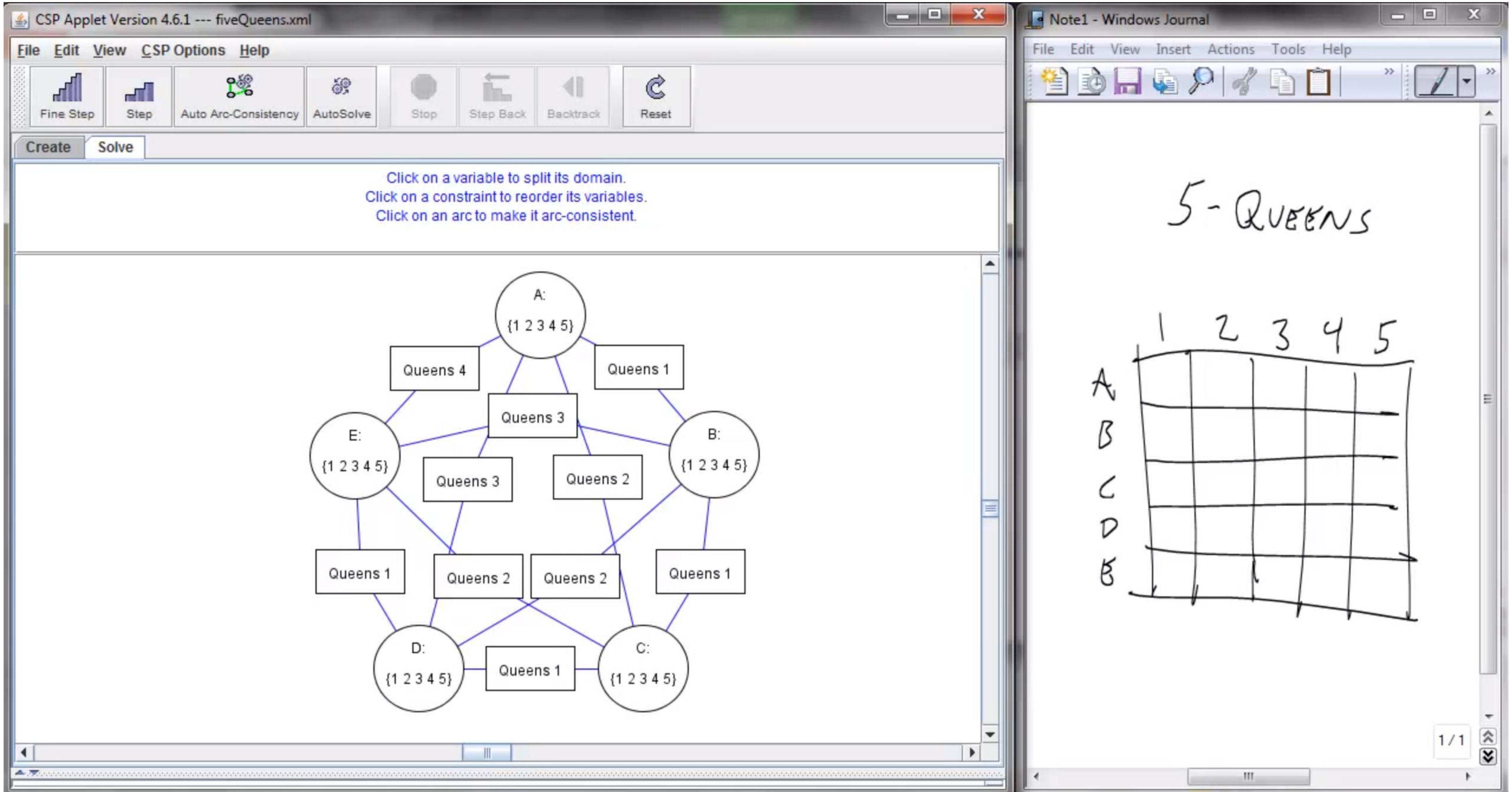
  $O + O = R + 10 \cdot X_1$

  $\cdots$

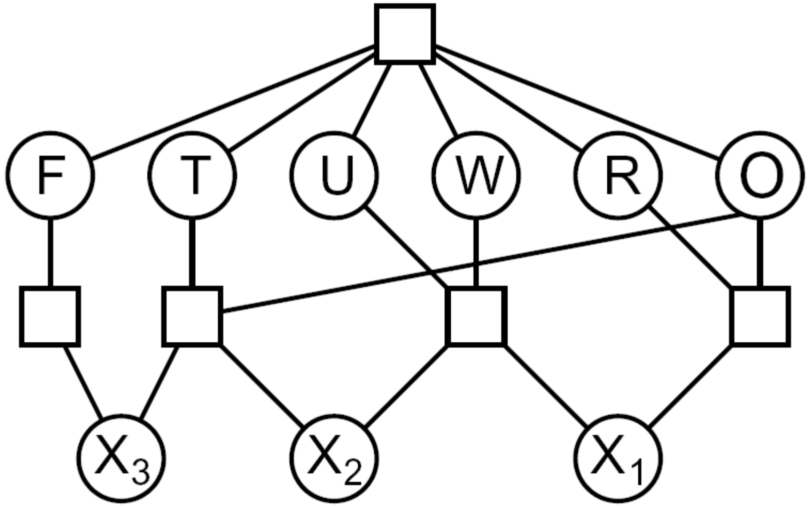
```
   T W O
 + T W O
 -------
 F O U R
```



Example: N-Queens



Solve non-binary CSPs



- Naïve search?
  - Yes!
- Backtracking?
  - Yes!
- Forward Checking?
  - Need to generalize the original FC operation
  - (nFC0) After a variable is assigned a value, find all constraints with only one unassigned variable and cross off values of that unassigned variable which violate the constraint
  - There exist other ways to do generalized forward checking

Solve non-binary CSPs 2

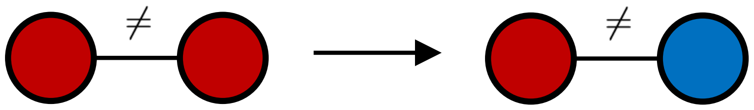
- AC-3? Need to generalize the definition of AC and enforcement of AC
- Generalized arc-consistency (GAC)
  - A non-binary constraint is GAC iff for every value for a variable there exist consistent value combinations for all other variables in the constraint
  - Reduced to AC for binary constraints
- Enforcing GAC
  - Simple schema: enumerate value combination for all other variables
  - O(d^k) on k-ary constraint on variables with domains of size d
- There are other algorithms for non-binary constraint propagation, e.g., (i,j)-consistency [Freuder, JACM 85]

Local Search

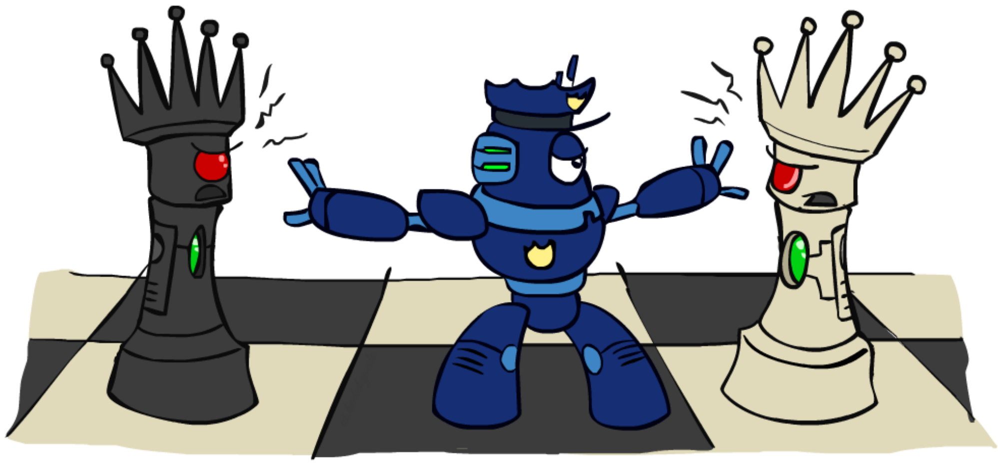
- Can be applied to identification problems (e.g., CSPs), as well as some planning and optimization problems
- Typically use a complete-state formulation
  - e.g., all variables assigned in a CSP (may not satisfy all the constraints)
- Different “complete”:
  - An assignment is complete means that all variables are assigned a value
  - An algorithm is complete means that it will output a solution if there exists one

Iterative Algorithms for CSPs

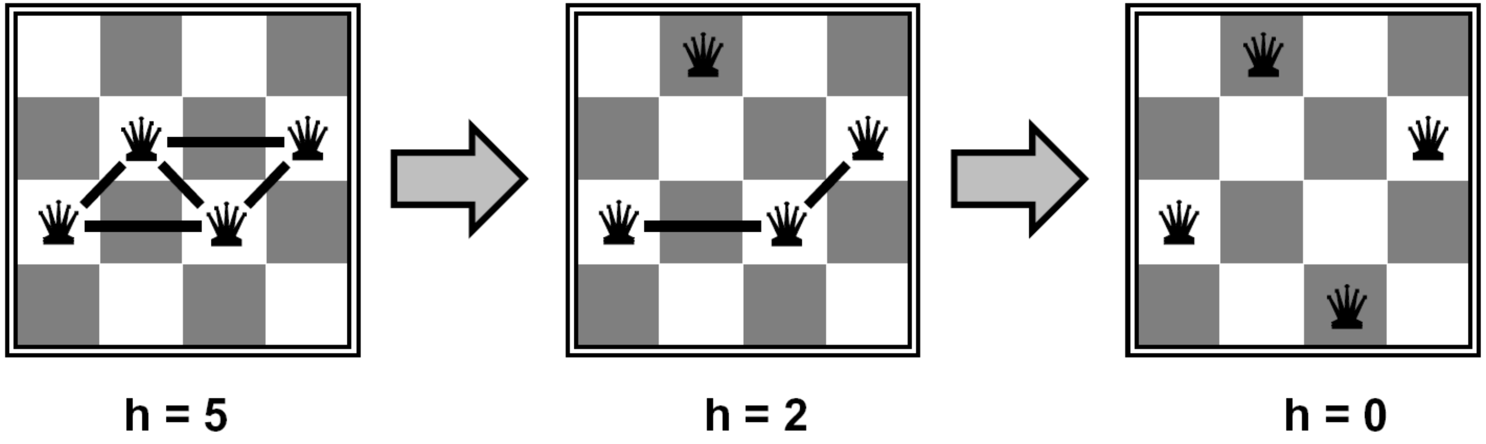
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.



- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic
    - Choose a value that violates the fewest constraints
    - v.s., hill climb with h(x) = total number of violated constraints (break tie randomly)

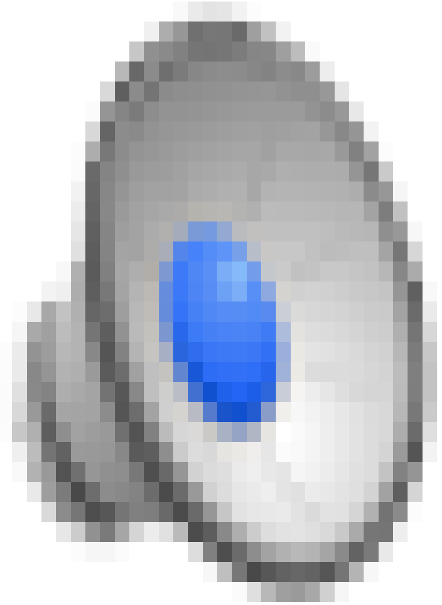


Example: 4-Queens

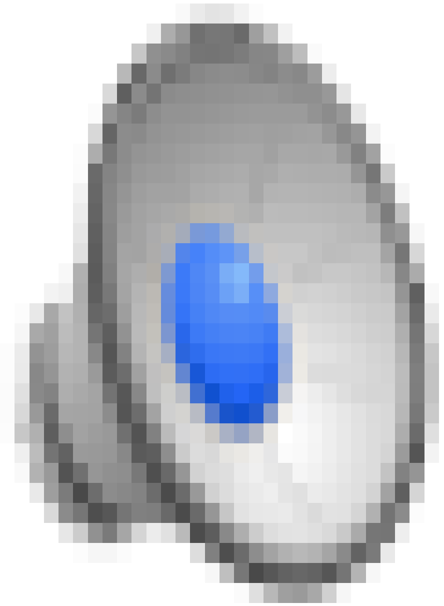


- States: 4 queens in 4 columns (4^4 = 256 states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: h(n) = number of attacks

Video of Demo Iterative Improvement – n Queens



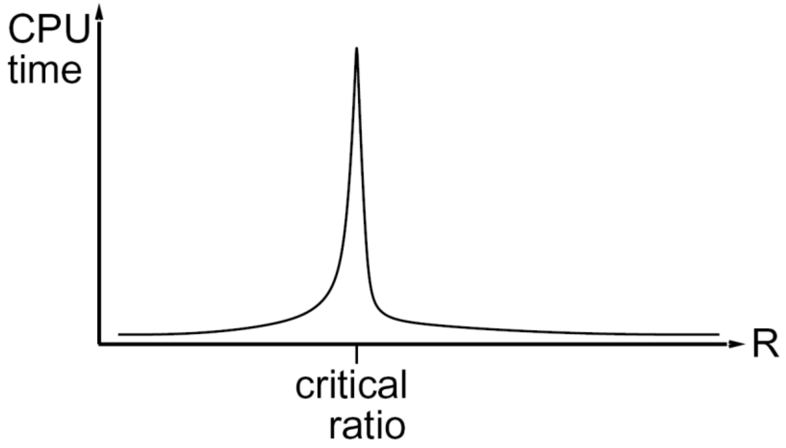
Video of Demo Iterative Improvement – Coloring

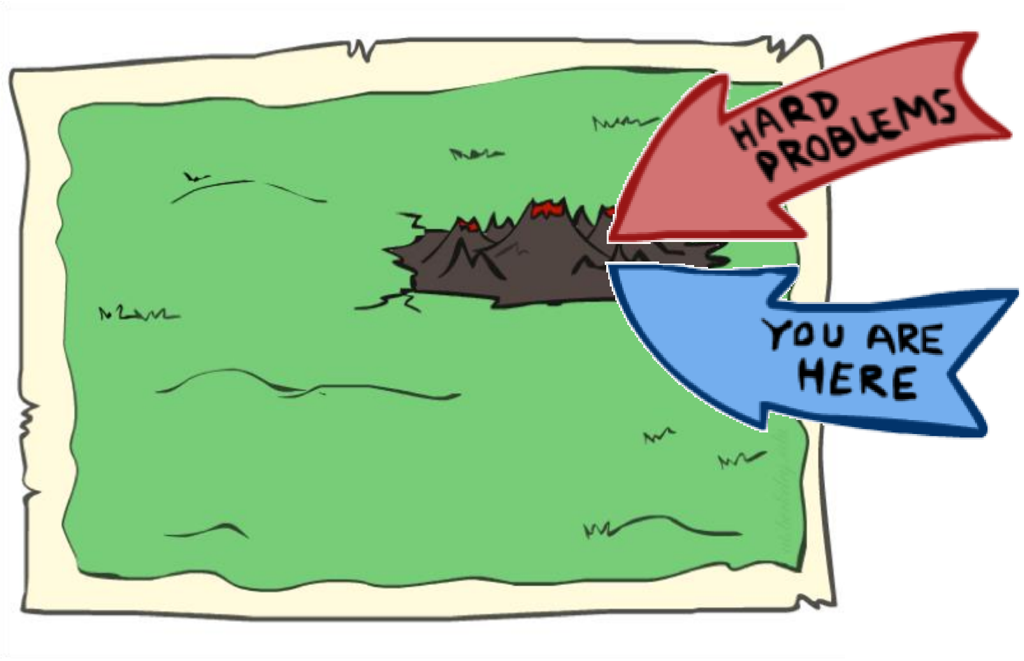


Performance of Min-Conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., n = 10,000,000)!
- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

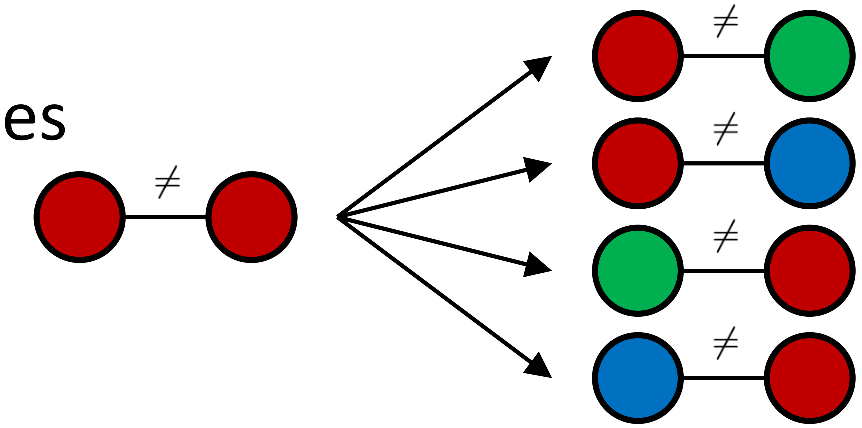
$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$





Local Search vs Tree Search

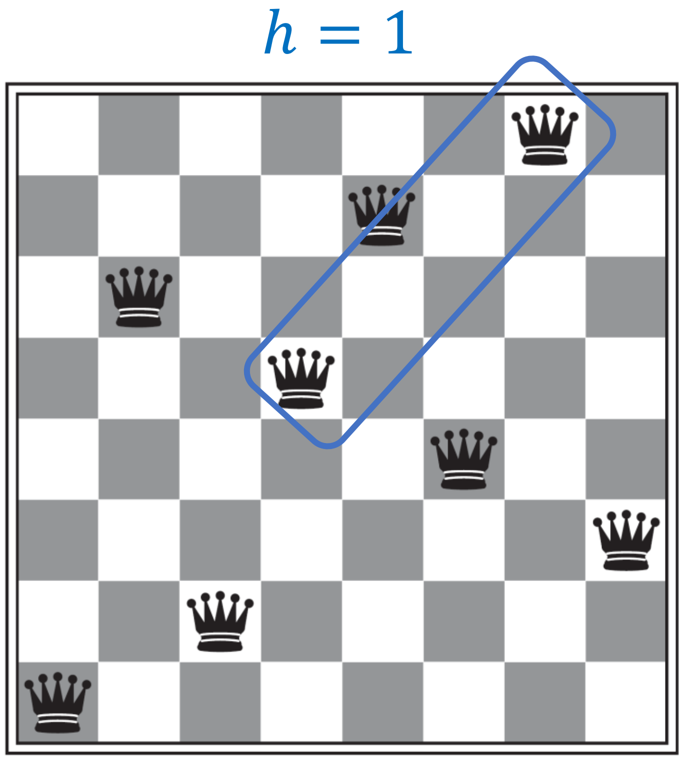
- Tree search keeps unexplored alternatives on the fringe (ensures completeness)
- Local search: improve a single option until you can't make it better (no fringe!)
- New successor function: local changes



- Generally much faster and more memory efficient (but incomplete and suboptimal)

Example

- Local search may get stuck in a local optima



Hill Climbing

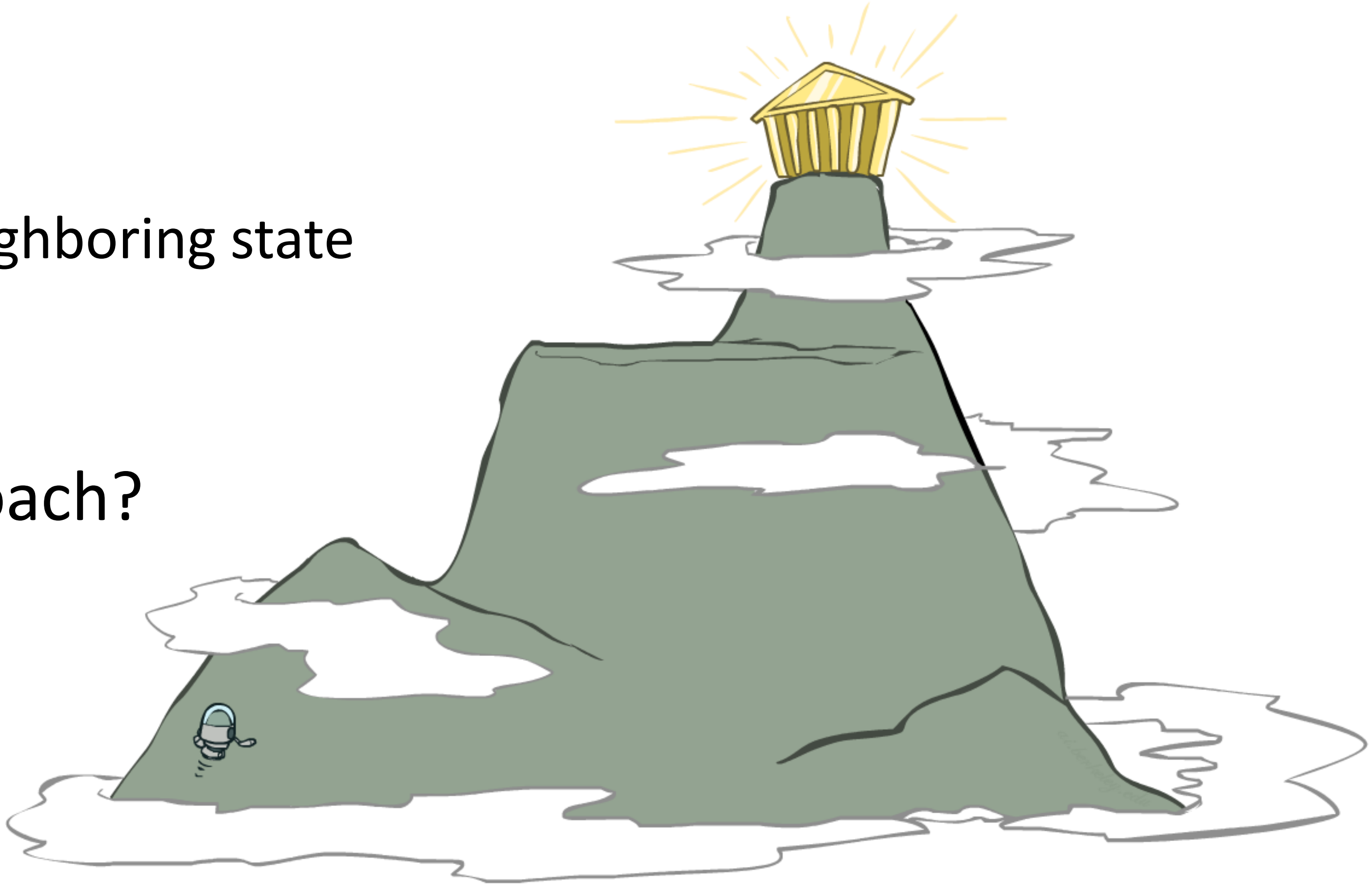
- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no for current, quit

- What's bad about this approach?

  Complete? No!

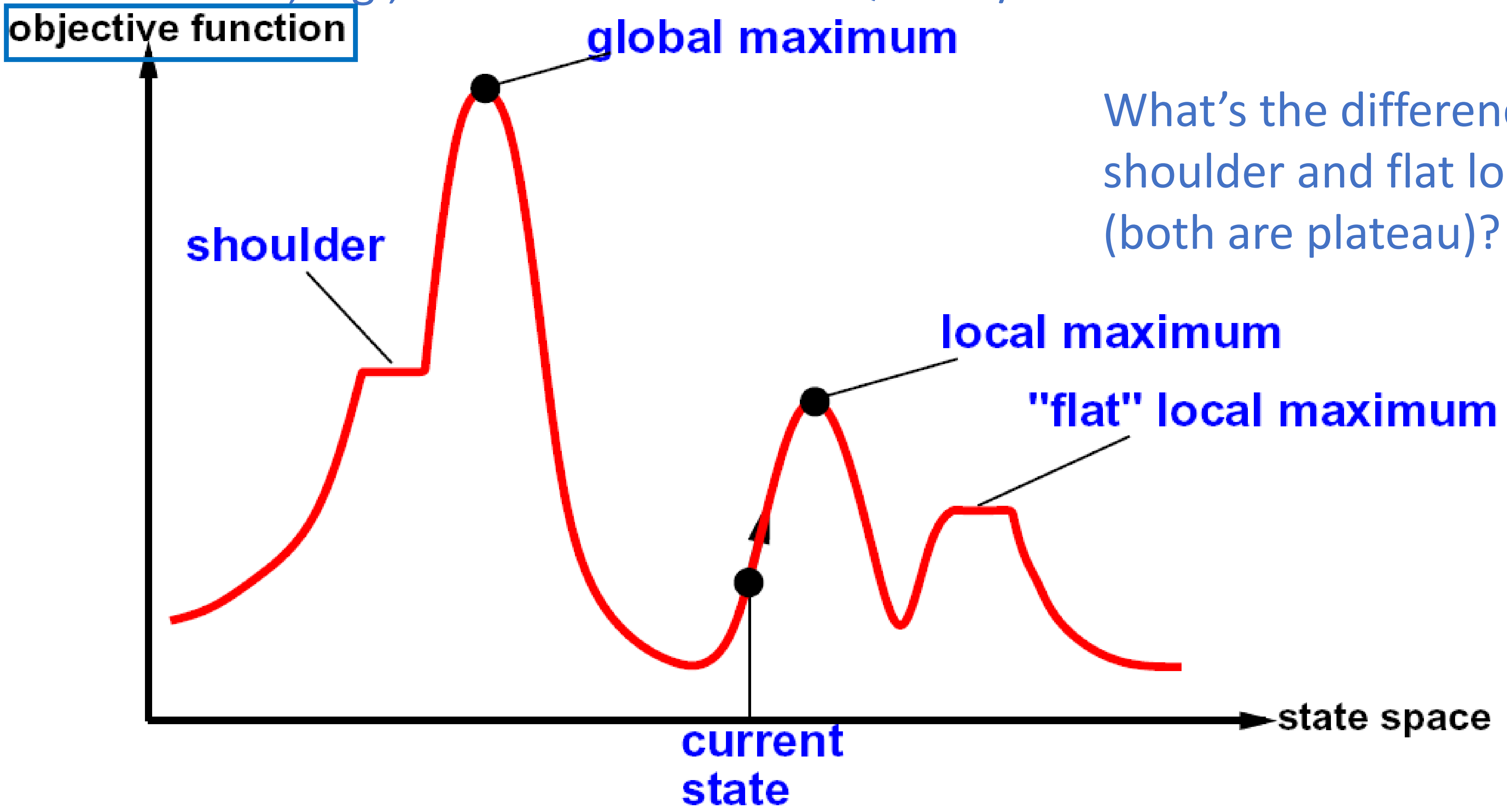
  Optimal? No!

- What's good about it?



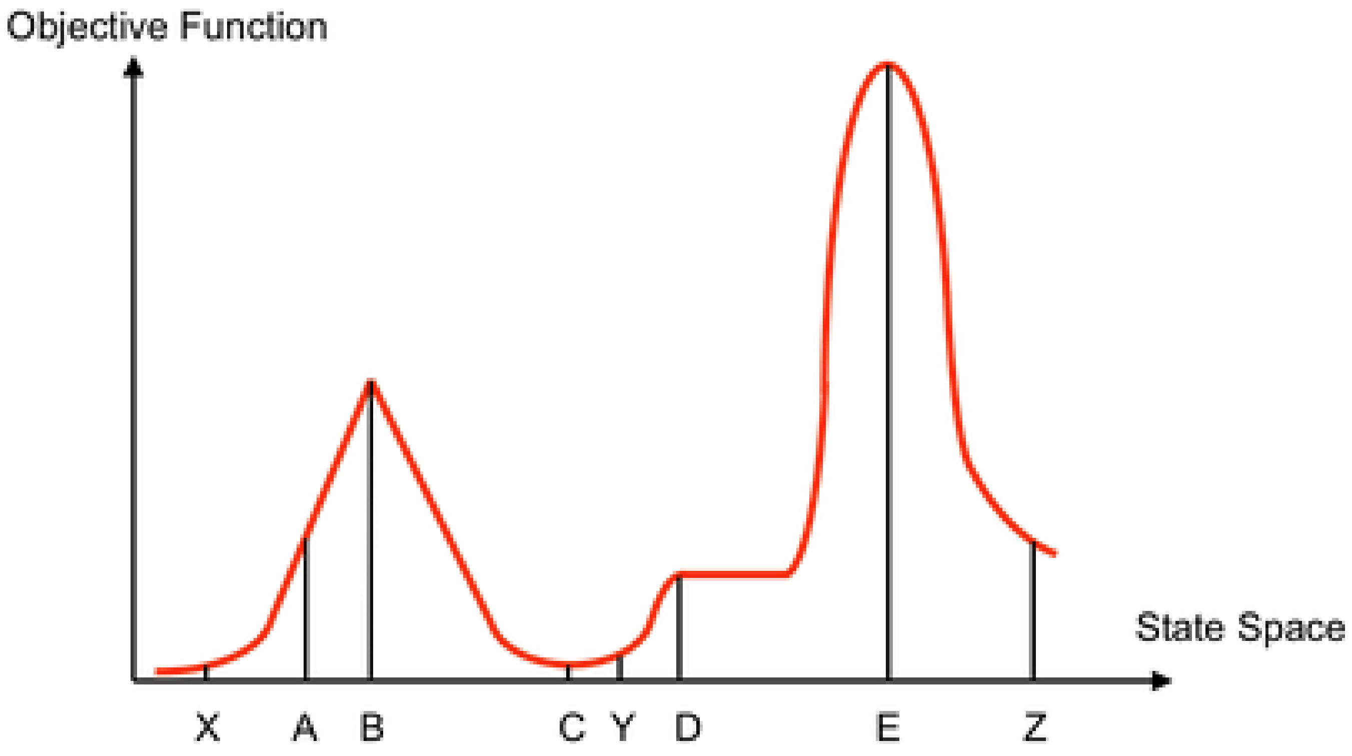
Hill Climbing Diagram

In identification problems, could be a function measuring how close you are to a valid solution, e.g., $-1 \times$ #conflicts in n-Queens/CSP



What's the difference between shoulder and flat local maximum (both are plateau)?

Quiz



- Starting from X, where do you end up ?
- Starting from Y, where do you end up ?
- Starting from Z, where do you end up ?

Hill Climbing (Greedy Local Search)

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

current ← MAKE-NODE(*problem*.INITIAL-STATE)
loop do
  neighbor ← a highest-valued successor of *current*
  if neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
  current ← *neighbor*

How to apply Hill Climbing to n-Queens? How is it different from Iterative Improvement?

Define a state as a board with n queens on it, one in each column
Define a successor (neighbor) of a state as one that is generated by moving a single queen to another square in the same column

Hill Climbing (Greedy Local Search) 2

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

  current ← MAKE-NODE(*problem*.INITIAL-STATE)
  loop do
    $\boxed{\textit{neighbor} \leftarrow \text{a highest-valued successor of } \textit{current}}$
    if neighbor.VALUE ≤ current.VALUE **then** $\boxed{\textbf{return}\ \textit{current}.\text{STATE}}$
    current ← *neighbor*

What if there is a tie?

Typically break ties randomly

What if we do not stop here?

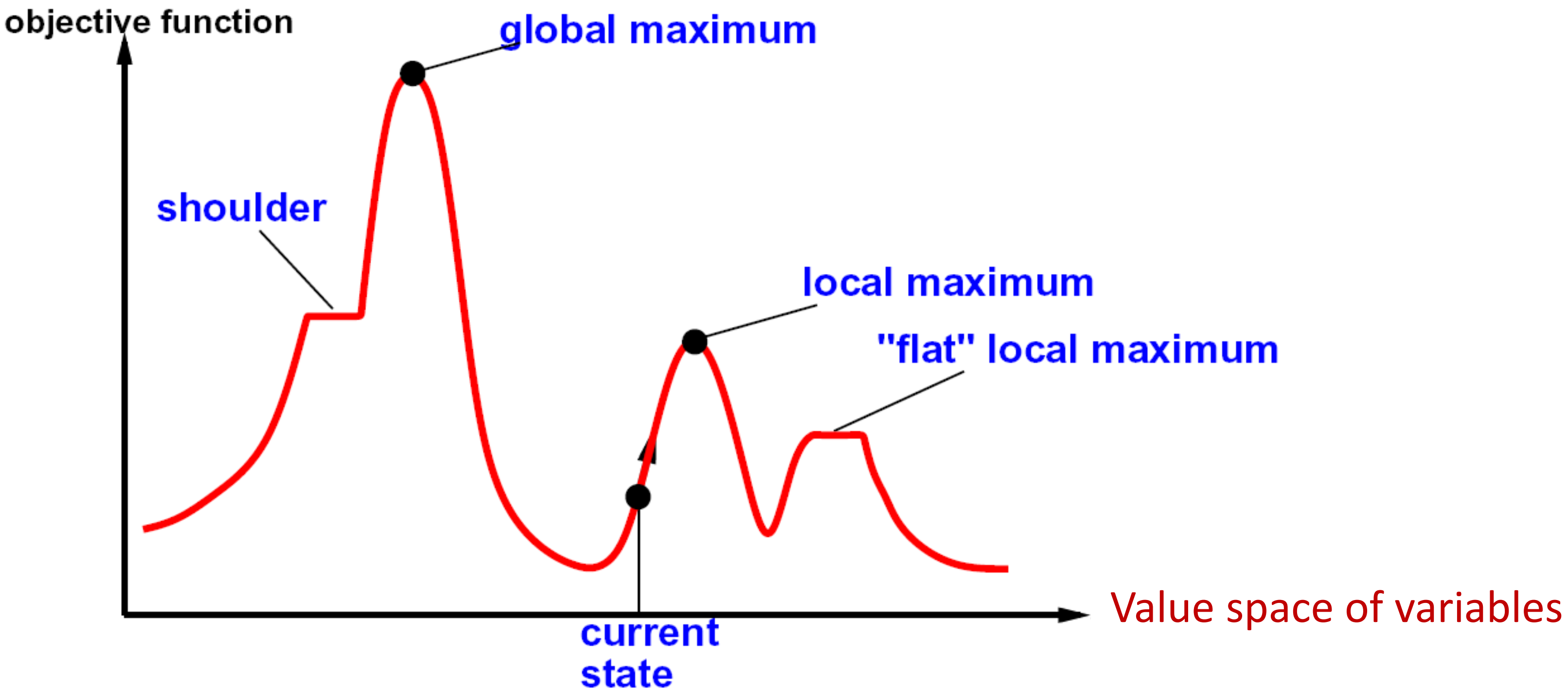
- In 8-Queens, steepest-ascent hill climbing solves 14% of problem instances
  - Takes 4 steps on average when it succeeds, and 3 steps when it fails
- When allow for ≤100 consecutive sideway moves, solves 94% of problem instances
  - Takes 21 steps on average when it succeeds, and 64 steps when it fails

Variants of Hill Climbing

- Random-restart hill climbing
  - “If at first you don’t succeed, try, try again.”
  - Complete!
  - What kind of landscape will random-restarts hill climbing work the best?
- Stochastic hill climbing
  - Choose randomly from the uphill moves, with probability dependent on the “steepness” (i.e., amount of improvement)
  - Converge slower than steepest ascent, but may find better solutions
- First-choice hill climbing
  - Generate successors randomly (one by one) until a better one is found
  - Suitable when there are too many successors to enumerate

Variants of Hill Climbing 2

- What if variables are continuous, e.g. find $x \in [0,1]$ that maximizes $f(x)$?
  - Gradient ascent
    - Use gradient to find best direction
    - Use the magnitude of the gradient to determine how big a step you move

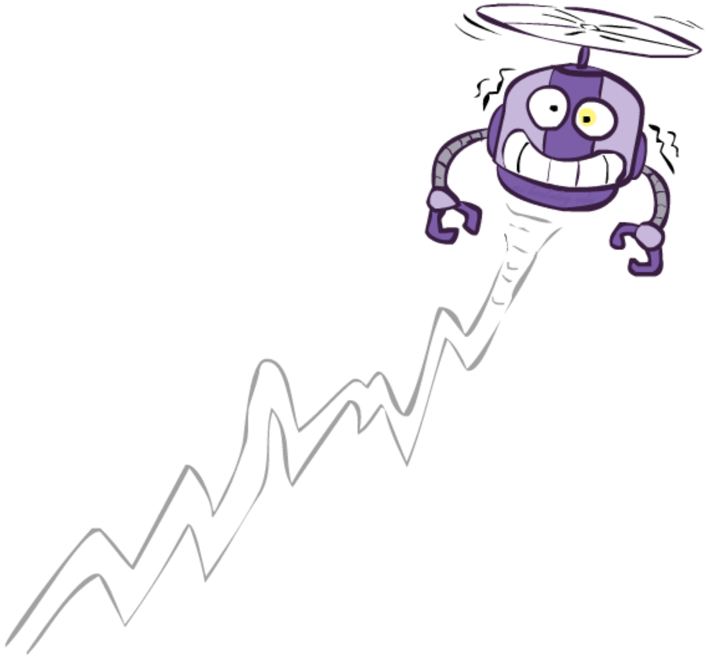


Random Walk

- Uniformly randomly choose a neighbor to move to
- Complete but inefficient!
- Stop according to the goal test

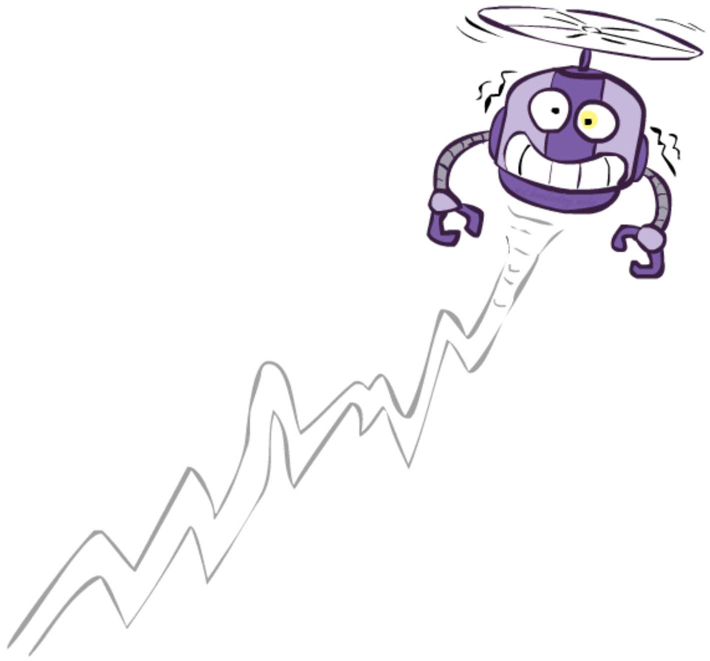
Simulated Annealing

- Combines random walk and hill climbing
- Complete and efficient
- Inspired by statistical physics
- Annealing – Metallurgy
  - Heating metal to high temperature then cooling
  - Reaching low energy state
- Simulated Annealing – Local Search
  - Allow for downhill moves and make them rarer as time goes on
  - Escape local maxima and reach global maxima



Simulated Annealing 2

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on



function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
        schedule, a mapping from time to "temperature"
  local variables: *current*, a node
        next, a node
        T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow schedule[t]$
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

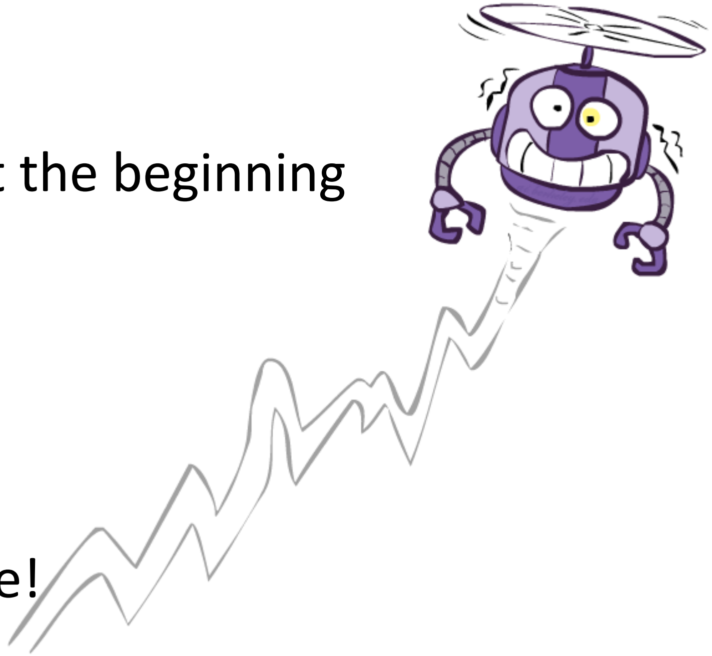
Control the change of temperature T (↓ over time)

Almost the same as hill climbing except for a *random* successor

Unlike hill climbing, move downhill with some prob.

Simulated Annealing 3

- $\mathbb{P}[\text{move downhill}] = e^{\Delta E/T}$
  - Bad moves are more likely to be allowed when T is high (at the beginning of the algorithm)
  - Worse moves are less likely to be allowed



- Theoretical guarantee:
  - Stationary distribution: $p(x) \propto e^{\frac{E(x)}{kT}}$
  - If T decreased slowly enough, will converge to optimal state!

- Is this an interesting guarantee?

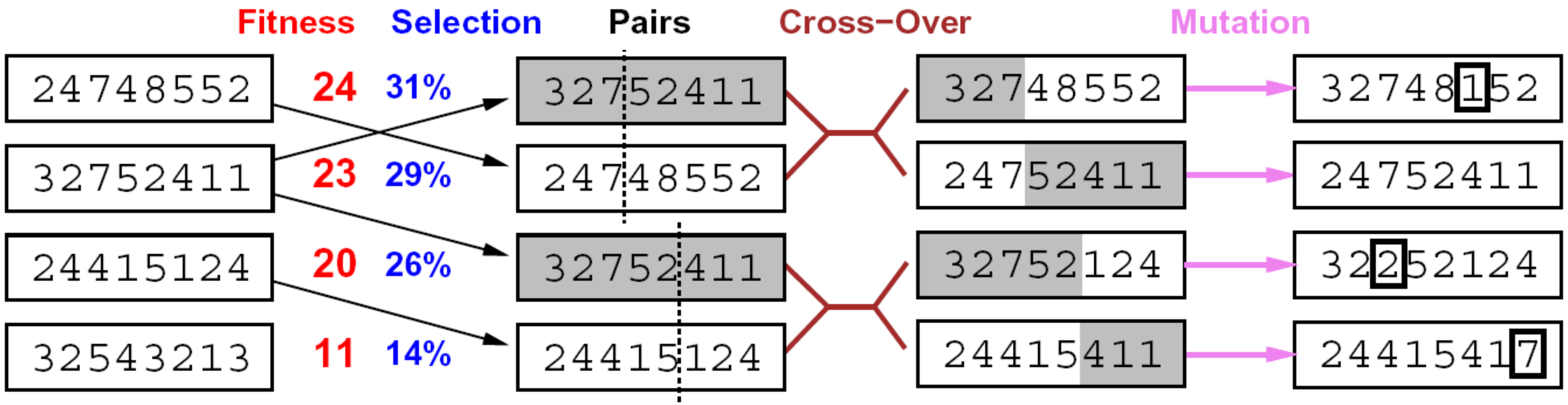
- Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - People think hard about *ridge operators* which let you jump around the space in better ways

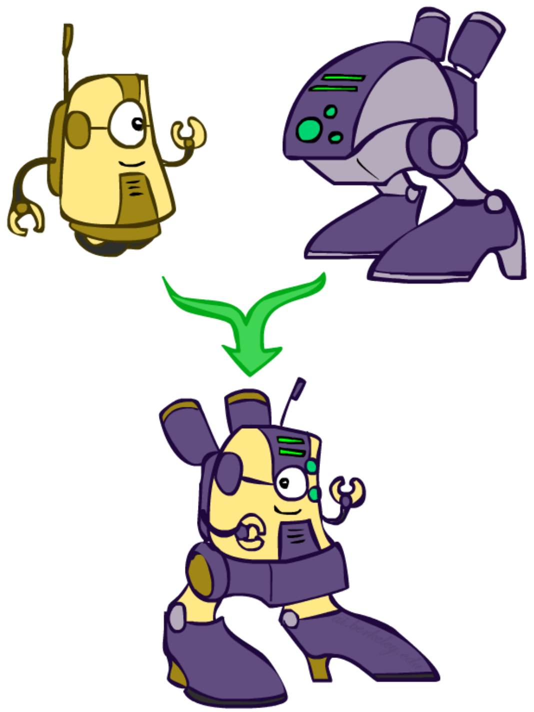
Genetic Algorithms

- Inspired by evolutionary biology
  - Nature provides an objective function (reproductive fitness) that Darwinian (达尔文) evolution could be seen as attempting to optimize

- A variant of stochastic beam search
  - Successors are generated by combining two parent states instead of modifying a single state (sexual reproduction rather than asexual reproduction)

Genetic Algorithms 2

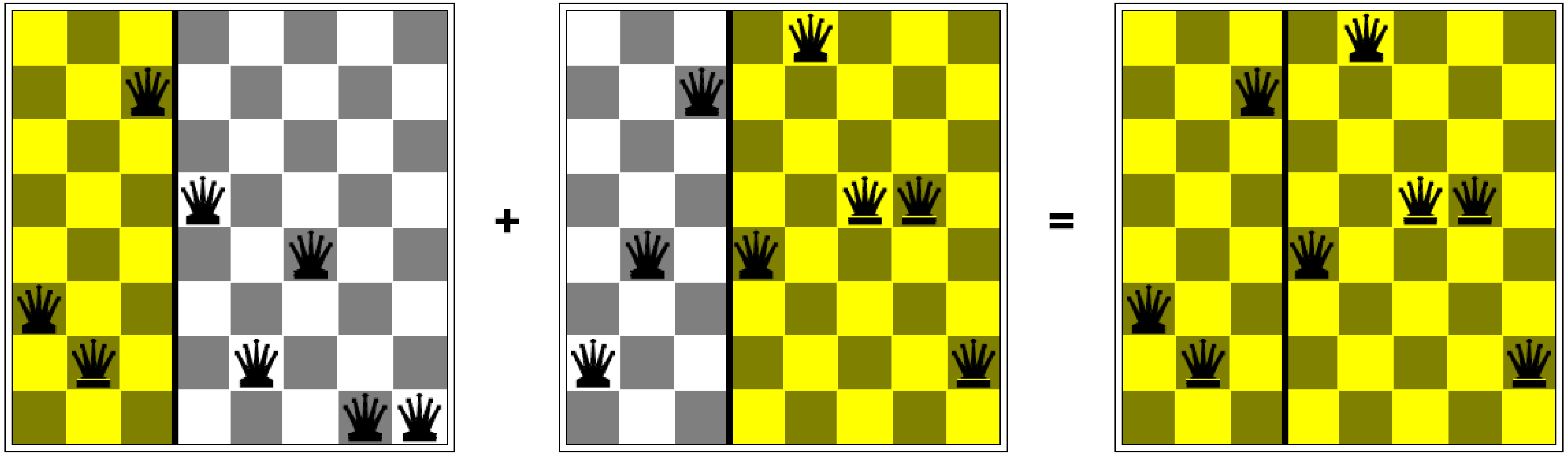




- State Representation: 8-digit string, each digit in $\{1..8\}$
- Fitness Function: #Nonattacking pairs
- Selection: Select k individuals randomly with probability proportional to their fitness value (random selection with replacement)
- Crossover: For each pair, choose a crossover point $\in \{1..7\}$, generate two offsprings by crossing over the parent strings
- Mutation (With some prob.): Choose a digit and change it to a different value in $\{1..8\}$

What if k is an odd number?

Example: N-Queens



- Why does crossover make sense here?
- When wouldn't it make sense?
- What would mutation be?
- What would a good fitness function be?

Genetic Algorithms 3

- Start with a population of k individuals (states)
- In each iteration
  - Apply a fitness function to each individual in the current population
  - Apply a selection operator to select k pairs of parents
  - Generate k offsprings by applying a crossover operator on the parents
  - For each offspring, apply a mutation operation with a (usually small) independent probability

- For a specific problem, need to design these functions and operators
- Successful use of genetic algorithms require careful engineering of the state representation!
- Possibly the most misunderstood, misapplied (and even maligned) technique around

Genetic Algorithms 4

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

How is this different from the illustrated procedure on 8-Queens?

Exercise: Traveling Salesman Problem

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?
- Input: $c_{ij}, \forall i, j \in \{0, \dots, n-1\}$
- Output: A ordered sequence $\{v_0, v_1, \dots, v_n\}$ with $v_0 = 0$, $v_n = 0$ and all other indices show up exactly once

- Question: How to apply Local Search algorithms to this problem?

Local Search: Summary

- Maintain a constant number of current nodes or states, and move to “neighbors” or generate “offsprings” in each iteration
  - Do not maintain a search tree or multiple paths
  - Typically do not retain the path to the node

- Advantages
  - Use little memory
  - Can potentially solve large-scale problems or get a reasonable (suboptimal or almost feasible) solution

Summary

Shuai Li

https://shuaili8.github.io

- CSPs
  - a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constraints
  - Planning vs Identification problems
- Basic solution: backtracking search
- Speed-ups:
  - Filtering: Forward checking & arc consistency
  - Ordering: MRV & LCV
  - Structure: Independent subproblems/Trees
- Local Search
  - Iterative algorithm/hill climb/simulated annealing/genetic algorithm

Questions?