

Constraint Satisfaction

Chen-Yu Wei

Constraint Satisfaction Problems (CSP)

- Variables: X_1, X_2, \dots, X_N
- Domains: $\text{Domain}_1, \dots, \text{Domain}_N$
 - X_i takes values in Domain_i
- Constraints: specifying the relations between the variables
- Solution: An assignment $\{X_1: v_1, X_2: v_2, \dots, X_N: v_N\}$ that satisfies all constraints

Example: Map Coloring

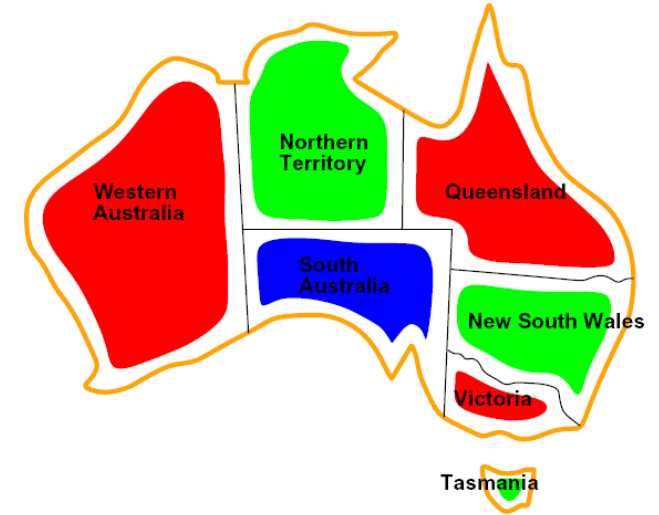
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors

Implicit: $WA \neq NT$

Explicit: $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), \dots\}$

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

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



Example: Cryptarithmic

- Variables:

$F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

- Domains:

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

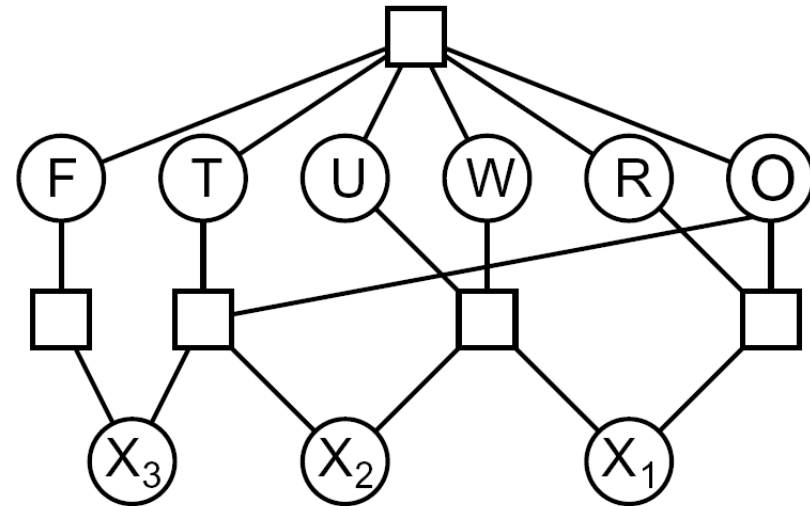
- Constraints:

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

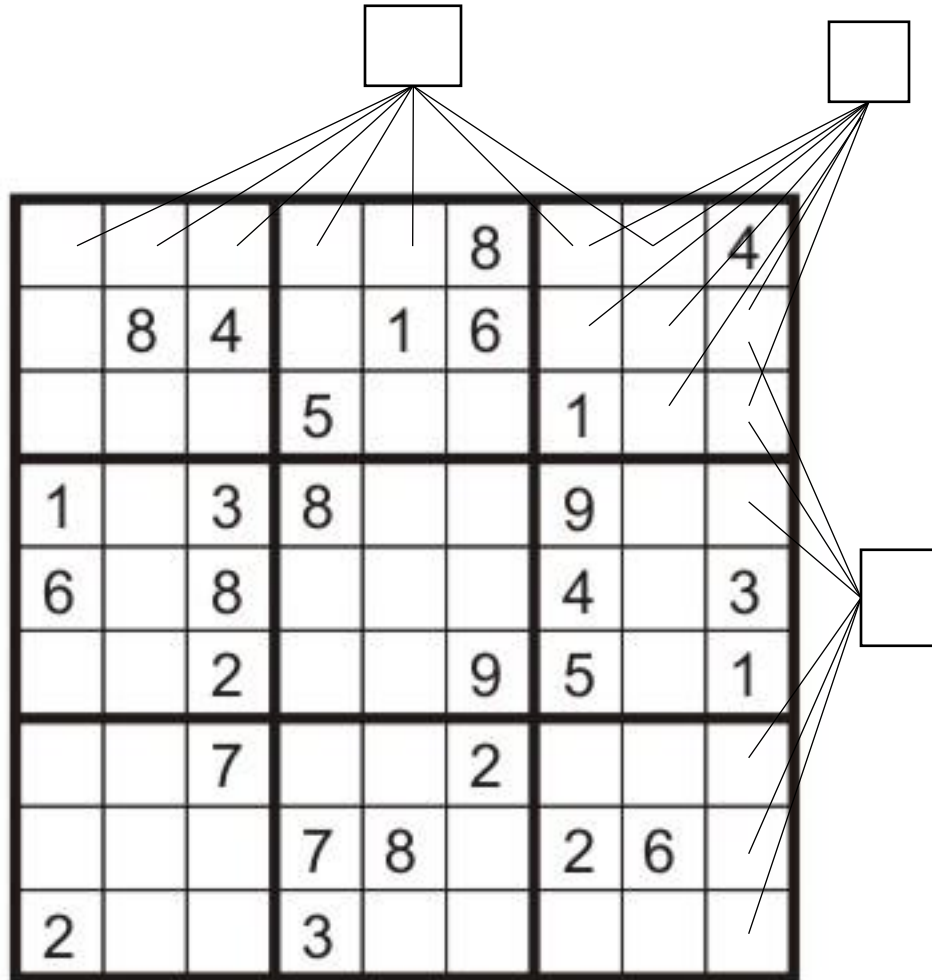
$O + O = R + 10 \cdot X_1$

...

$$\begin{array}{r} T \ W \ O \\ + \ T \ W \ O \\ \hline F \ O \ U \ R \end{array}$$



Example: Sudoku



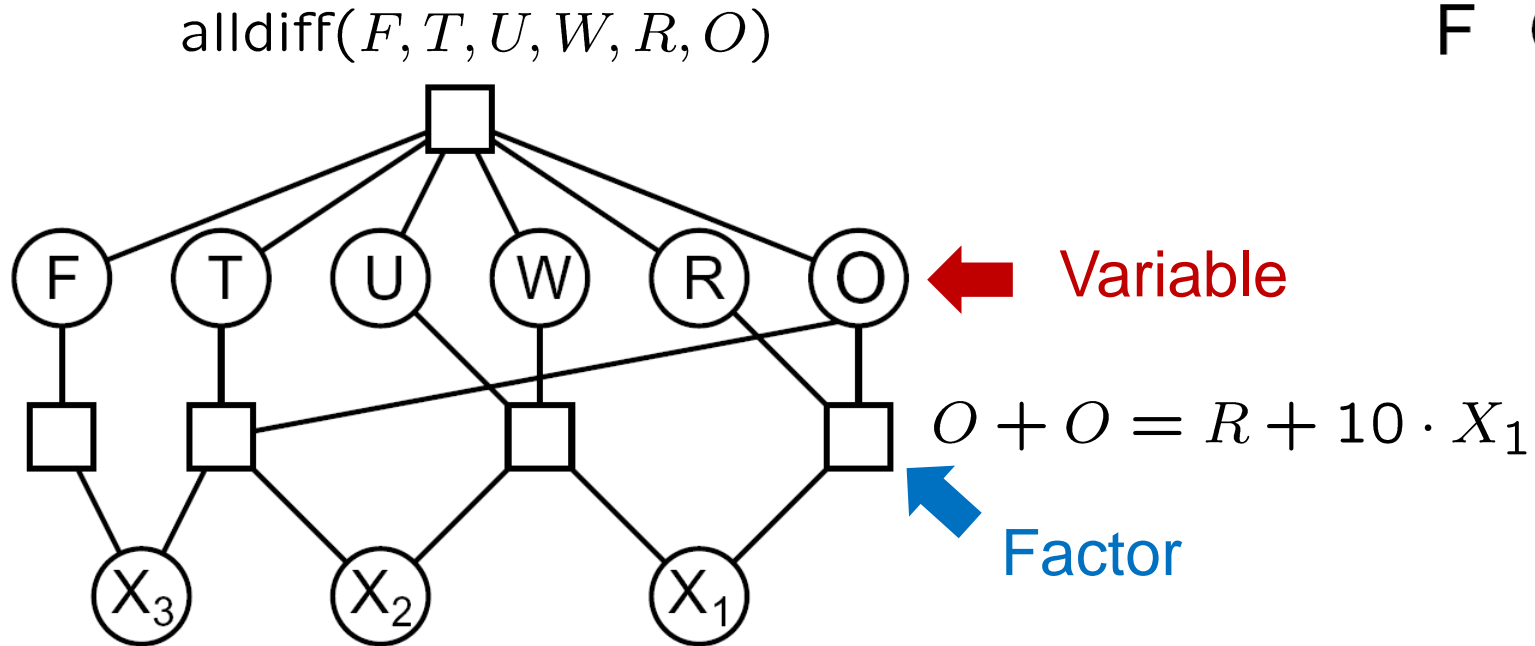
- Variables: Each (open) square
- Domains: $\{1, 2, \dots, 9\}$
- Constraints:
 - 9-way alldiff for each column
 - 9-way alldiff for each row
 - 9-way alldiff for each region
 - (or can have a bunch of pairwise inequality constraints)

Real-World 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
 - ...
-
- Many real-world problems involve real-valued variables

Factor Graph

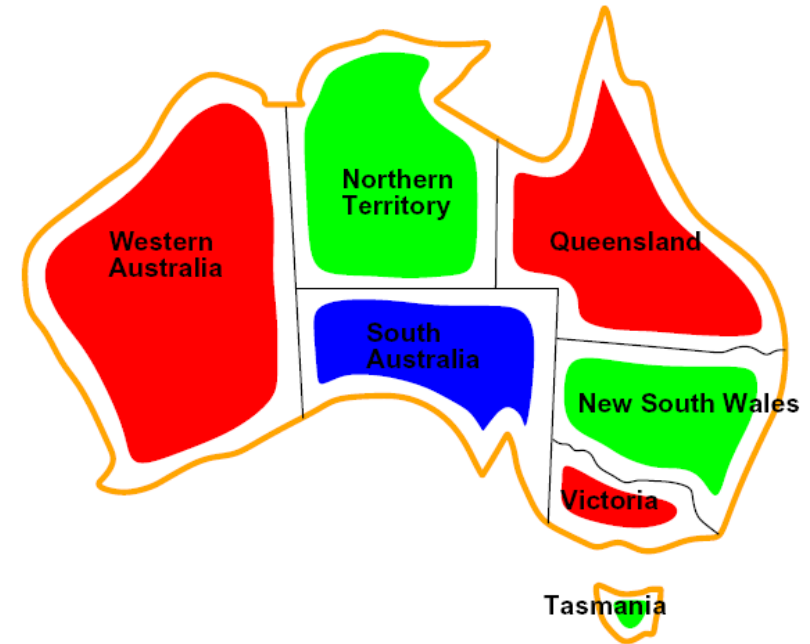
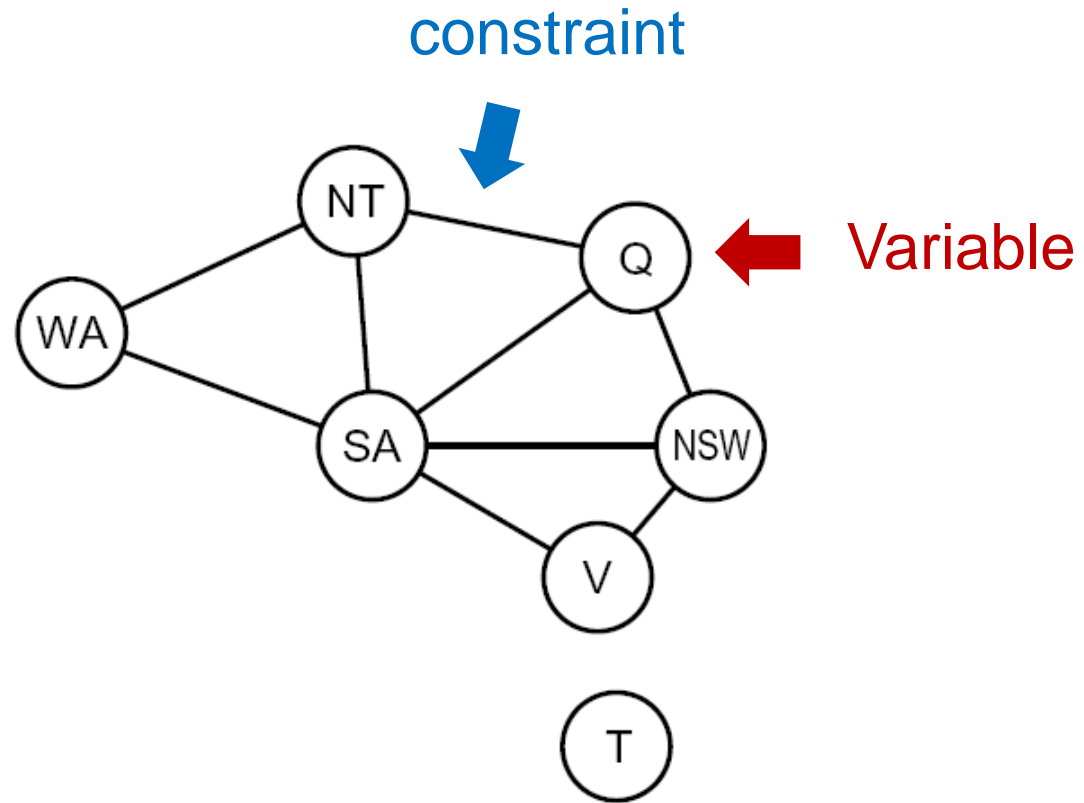
$$\begin{array}{r} T W O \\ + T W O \\ \hline F O U R \end{array}$$



$$f_1(O, R, X_1) = \begin{cases} 1, & \text{if } O + O = R + 10X_1 \\ 0, & \text{otherwise} \end{cases}$$

$$\text{Weight} = \prod_{i=1}^4 f_i(F, T, U, W, R, O, X_1, X_2, X_3)$$

Constraint Graph

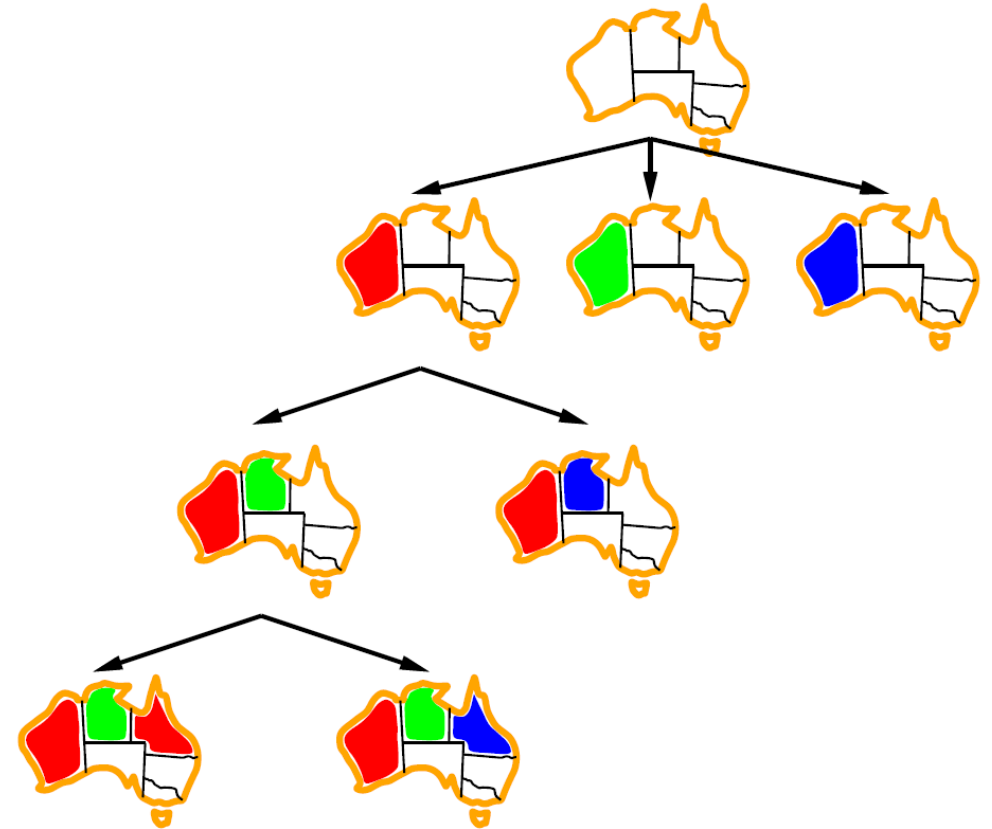


(more convenient for **binary constraint** CSPs)

Every constraint involves at most 2 variables

How to Solve CSP?

- Treat it as a search problem
 - Assign one variable at a time
 - **State:** A partial assignment
 - **Action:** Assign value to an unassigned variable
 - **Goal test:** check whether all constraints are satisfied
- But there's more structure to leverage
 - Variable ordering doesn't matter
 - Variables are interdependent in a local way
- We will start from known search algorithms, and try to speed it up



Backtracking Search

Backtracking Search

Backtracking search = DFS + failure-on-violation

BacktrackingSearch($\{ \}$, Domain) returns an assignment or reports failure.

BacktrackingSearch(x , Domain):

If x is a complete assignment: **return** x .

Let X_i be the next unassigned variable.

For each value $v \in \text{Domain}_i$:

$x' \leftarrow x \cup \{X_i: v\}$

If x' violates constraints: **continue**

return **BacktrackingSearch**(x' , Domain)

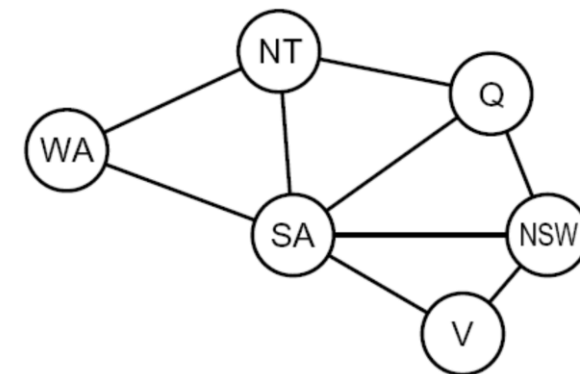
return failure

Improving Backtracking Search

- Forward checking
- Maintaining arc consistency (more powerful than forward checking)
- Dynamic ordering

Vanilla Backtracking Search

Suppose we assign the variables in the order of
WA, Q, V, NT, NSW, SA



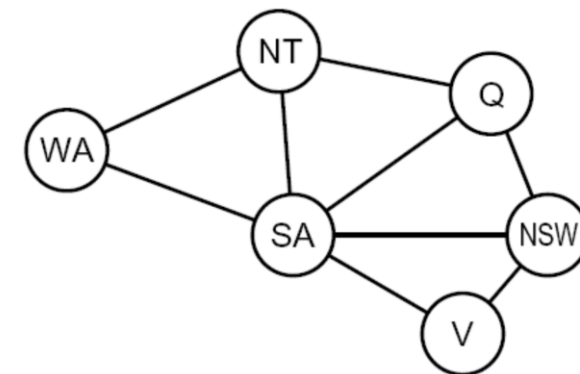
WA	NT	Q	NSW	V	SA

No valid assignment for **SA**.

Then the algorithm backtracks to try other assignments...

Forward Checking

Cross off values that violate a constraint when added to the existing assignment



WA	NT	Q	NSW	V	SA
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

Inconsistency found for **SA** (even though we haven't reached the layer of SA).

Forward Checking

BacktrackingSearch(x , Domain):

If x is a complete assignment: **return** x .

Let X_i be the next unassigned variable.

For each value $v \in \text{Domain}_i$:

$x' \leftarrow x \cup \{X_i: v\}$

If x' violates constraints: **continue**

return **BacktrackingSearch**(x' , Domain')

return failure

Forward Checking

BacktrackingSearch(x , Domain):

If x is a complete assignment: **return** x .

Let X_i be the next unassigned variable.

For each value $v \in \text{Domain}_i$:

$x' \leftarrow x \cup \{X_i: v\}$

Domain', Consistent = **ForwardChecking**(x' , X_i , v , Domain)

If not Consistent: **continue**

return **BacktrackingSearch**(x' , Domain')

return failure

ForwardChecking (x' , X_i , v , Domain):

Domain' \leftarrow Domain

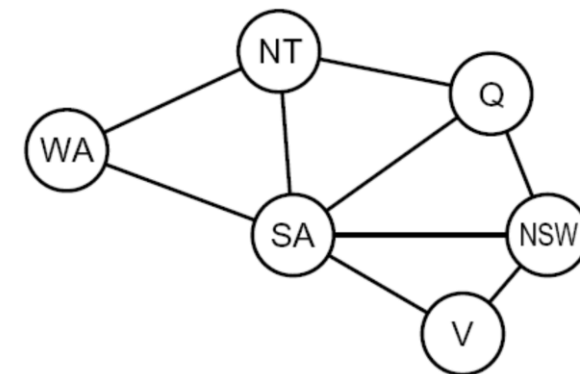
For all X_j that is unassigned in x' and connected to X_i :

Delete values in Domain' _{j} that are inconsistent with $\{X_i: v\}$

If Domain' _{j} is empty: **return** Domain', *False*

return Domain', *True*

Can We Prune Even More?



With forward checking:

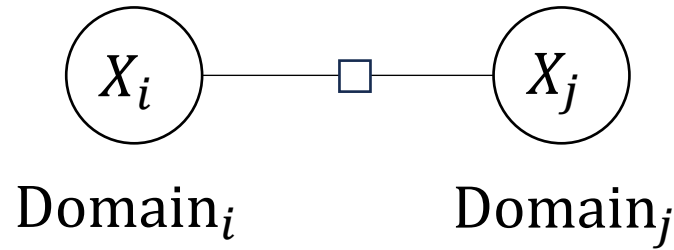


After assigning **Q** with green, **NT** and **SA**'s domains are left with only blue.

But **NT** and **SA** are neighbors, so there is no consistent assignment from here.

How can we detect such inconsistency at this step?

Arc Consistency



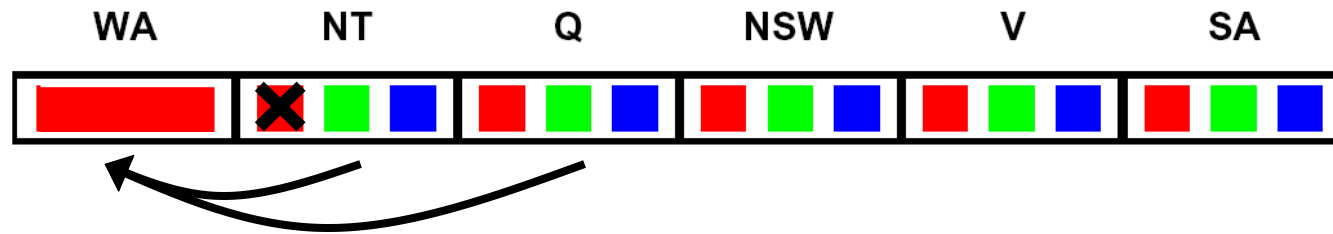
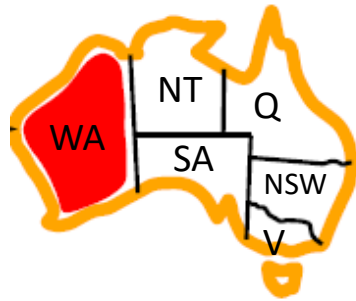
Fact. Let $v \in \text{Domain}_i$ be such that **for all $w \in \text{Domain}_j$, $\{X_i: v, X_j: w\}$ violates the constraint on (X_i, X_j)** . Then we can remove v from Domain_i .

Definition (Arc Consistency on $X_i \rightarrow X_j$). For all $v \in \text{Domain}_i$, there is some $w \in \text{Domain}_j$ such that $\{X_i: v, X_j: w\}$ satisfies the constraint on (X_i, X_j) .

Idea to prune more: keep checking whether we can remove elements from any Domain using the fact above. (i.e., always maintaining arc consistency)

Maintaining Arc Consistency (MAC)

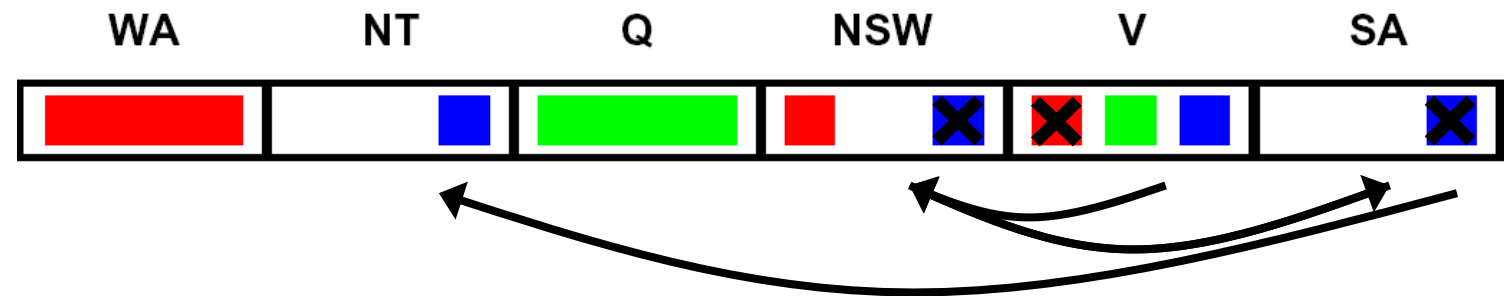
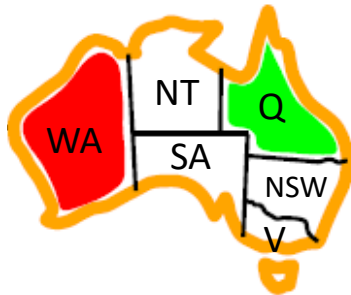
Forward checking: maintaining arc consistency **from unassigned variables to newly assigned variables.**



Maintaining Arc Consistency (MAC)

We can prune more if we ensure arc consistency for all arcs.

Remember: Delete from the tail!



If X's domain changes, neighbors of X need to be rechecked!

Maintaining Arc Consistency (MAC)

AC3:

queue \leftarrow initial queue

while queue not empty:

$(X_i, X_j) \leftarrow \text{POP}(\text{queue})$

if arc $X_i \rightarrow X_j$ is not consistent:

 Revise Domain_i to make it consistent

if Domain_i is empty: **return** *False*

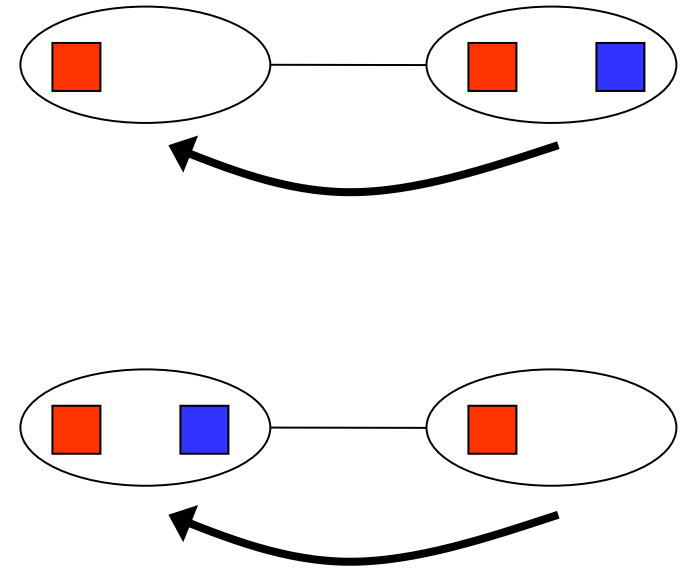
for each X_k connected to X_i :

 add (X_k, X_i) to queue

return *True*

Arc Consistency in Map Coloring Problems

- Useful when there is only one color left at the arc head
- Actually, this is also the only useful case in the map coloring problem

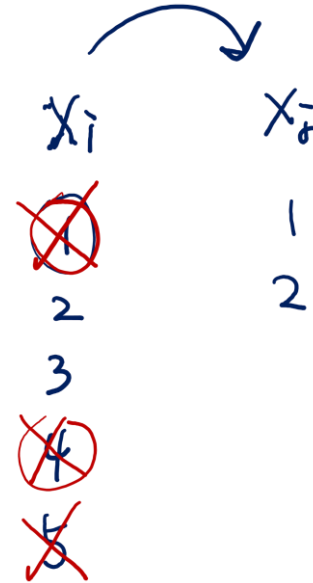


Arc Consistency in Other Problems

$X_i \in \text{Domain}_i = \{1,2,3,4,5\}$

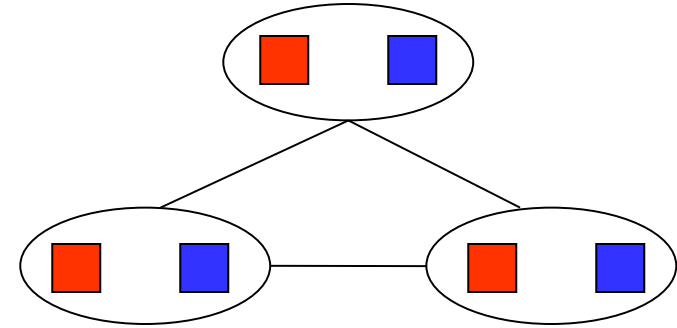
$X_j \in \text{Domain}_j = \{1,2\}$

Constraint: $X_i + X_j = 4$



Limitations of Arc Consistency

- Some failure modes cannot be detected by arc consistency.
- Therefore, we still need “backtracking”.



Maintaining Arc Consistency (MAC)

Combining AC3 with backtracking search:

BacktrackingSearch(x , Domain):

If x is a complete assignment: **return** x .

Let X_i be the next unassigned variable.

For each value $v \in \text{Domain}_i$:

$x' \leftarrow x \cup \{X_i: v\}$

Domain', Consistent = **AC3**(x' , X_i , v , Domain)

If not Consistent: **continue**

return **BacktrackingSearch**(x' , Domain')

return failure

Maintaining Arc Consistency (MAC)

AC3($x', X_i, v, \text{Domain}$):

$\text{Domain}' \leftarrow \text{Domain}$

$\text{queue} \leftarrow \{(X_j, X_i) \text{ for all } X_j \text{ that is unassigned in } x' \text{ and connected to } X_i\}$

while queue not empty:

$(X_k, X_\ell) \leftarrow \text{POP}(\text{queue})$

if arc $X_k \rightarrow X_\ell$ is not consistent:

Revise Domain'_k to make it consistent

if Domain'_k is empty: **return** $\text{Domain}', \text{False}$

for each X_m that is unassigned in x' and connected to X_i :

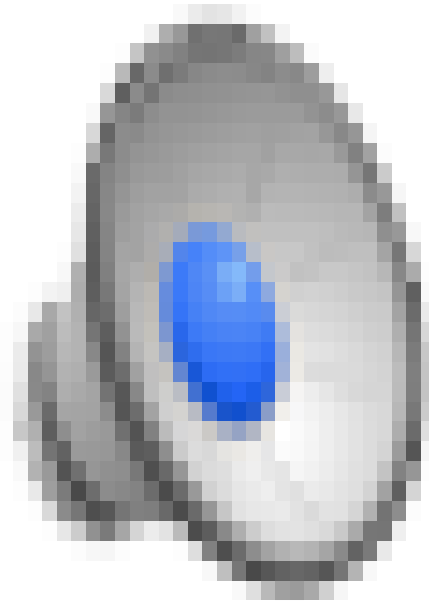
add (X_m, X_k) to queue

return $\text{Domain}', \text{True}$

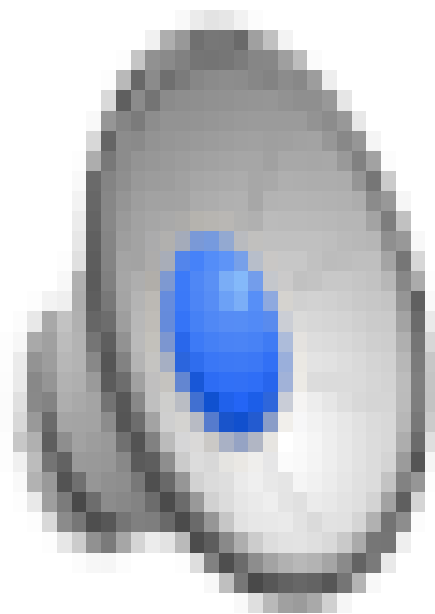
Maintaining Arc Consistency (MAC)

Drawbacks of backtracking search with arc-consistency check?

Video: Forward Checking



Video: Maintaining Arc Consistency



K-Consistency

2-Consistency (= Arc-Consistency)

- For any assignment $X_i = v_i$, there exists an assignment of $X_j = v_j$ such that $\{X_i = v_i, X_j = v_j\}$ satisfies the constraint on (X_i, X_j) .

3-Consistency

- For any consistent assignment $\{X_i = v_i, X_j = v_j\}$, there exists an assignment $X_k = v_k$ such that $\{X_i = v_i, X_j = v_j, X_k = v_k\}$ satisfies constraints on $(X_i, X_k), (X_j, X_k), (X_i, X_j, X_k)$.

K-Consistency

- For any consistent assignment $\{X^{(1)} = v^{(1)}, \dots, X^{(k-1)} = v^{(k-1)}\}$, there exists an assignment $X^{(k)} = v^{(k)}$ such that $\{X^{(1)} = v^{(1)}, \dots, X^{(k)} = v^{(k)}\}$ satisfies all constraints among $X^{(1)}, \dots, X^{(k)}$.

“Naked Triples” Strategy in Sudoku

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

$\{1, 8\}$ $\{1, 3, 8\}$

$\{3, 8\}$

- In any units (i.e., row, column, or box), find three squares each have a domain that contains the same three numbers or subset of these three numbers e.g., the three domains might be $\{1, 8\}$, $\{3, 8\}$, $\{1, 3, 8\}$
- Eliminate these three numbers from the domains of all other squares in the same unit.
- The idea is similar to checking K-consistency: For any assignment that assigns 1 or 3 or 8 to other squares (blue dots on the left figure), there is no feasible solution for the chosen three squares (red dots). Therefore, 1, 3, 8 can be eliminated from other squares's domains.

Improving Backtracking Search

- Forward checking
- Maintaining arc consistency (more powerful than forward checking)
- **Dynamic ordering**

Ordering

BacktrackingSearch(x , Domain):

If x is a complete assignment: **return** x .

Let X_i be the next unassigned variable.

← Which **variable** should we pick first?

For each value $v \in \text{Domain}_i$:

← Which **value** should we try first?

$x' \leftarrow x \cup \{X_i: v\}$

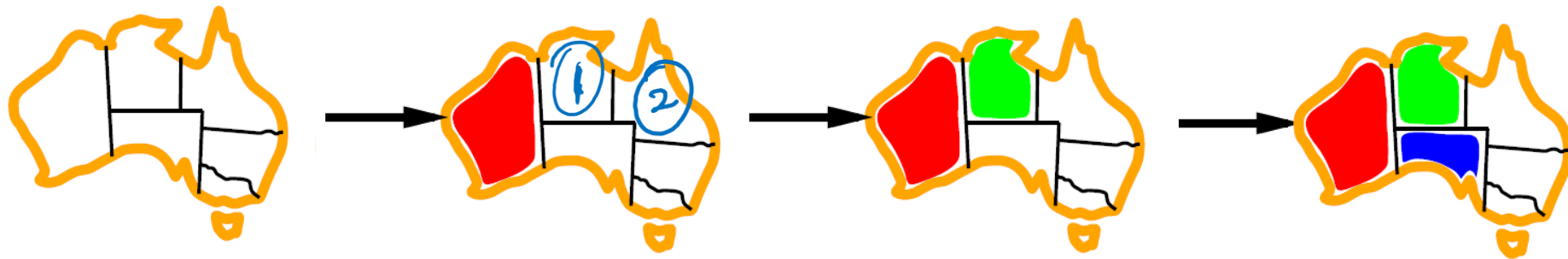
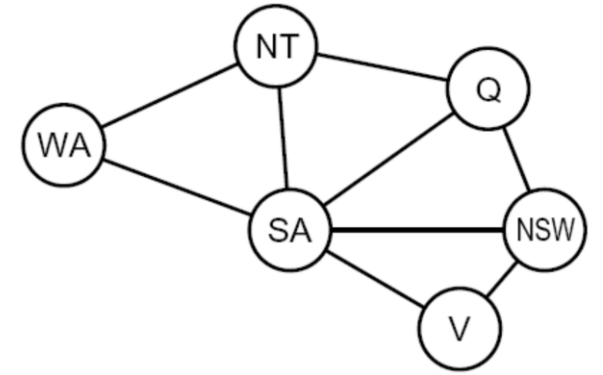
Domain', Consistent = **AC3**(x' , X_i , v , Domain)

If not Consistent: **continue**

return **BacktrackingSearch**(x' , Domain')

return failure

Variable Ordering



Variable Ordering

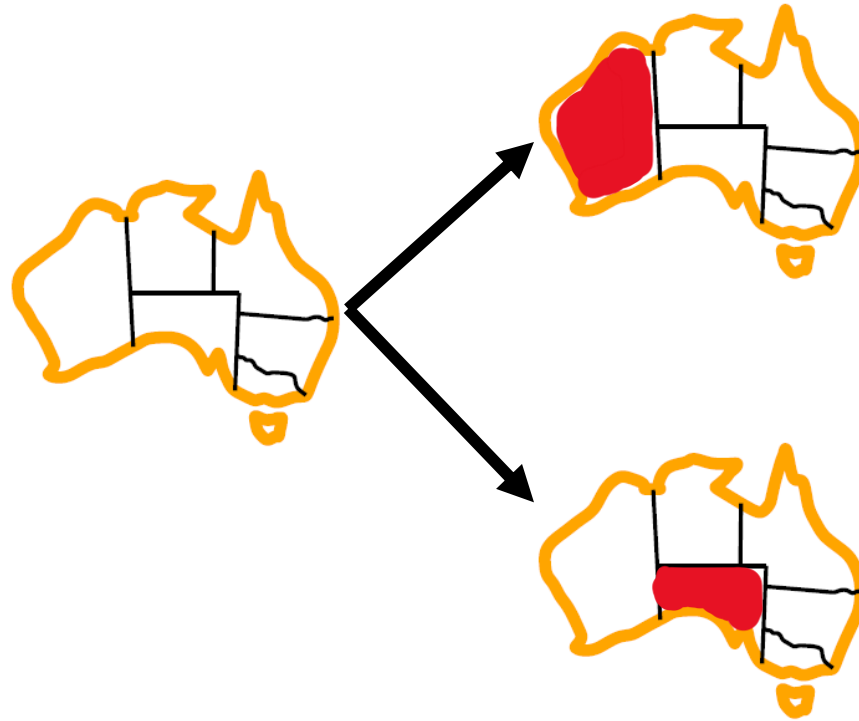
Minimum Remaining Value (MRV) heuristic

Choose variable that has the fewest left values in its domain.

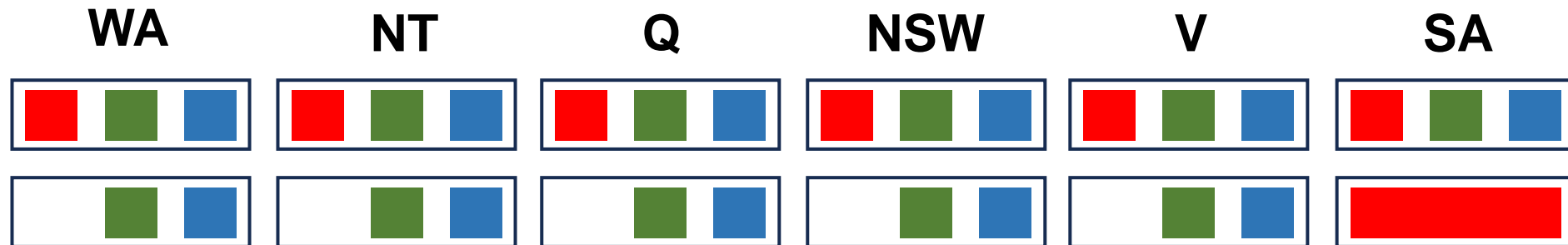
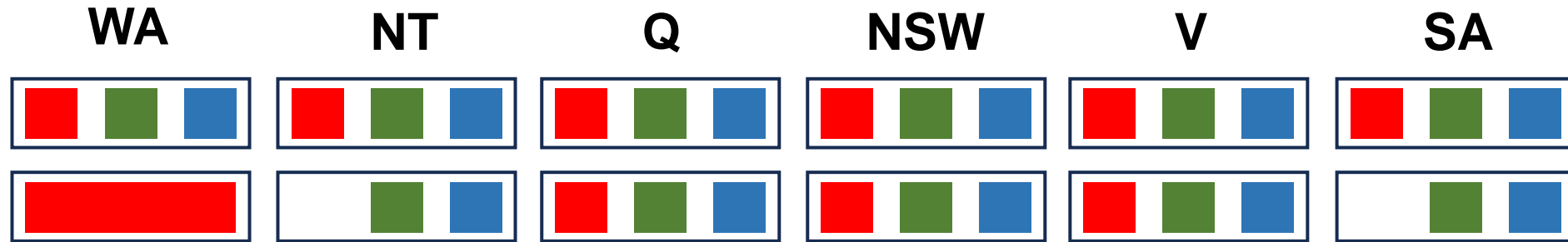
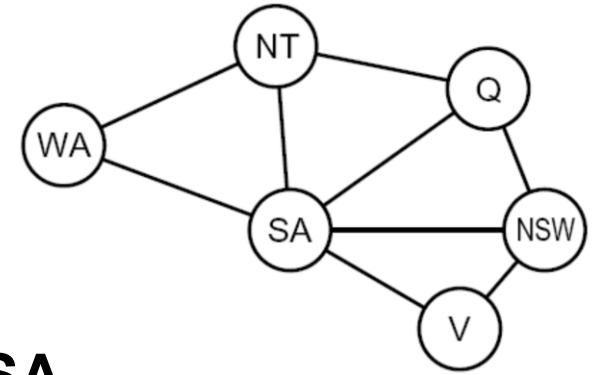
Why?

- Must assign **every** variable
- If going to fail, fail early → more pruning

Variable Ordering



Variable Ordering



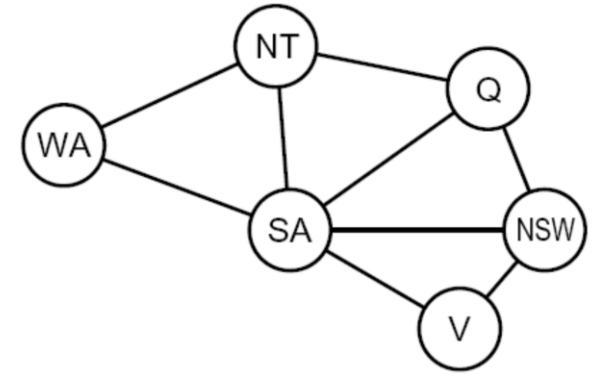
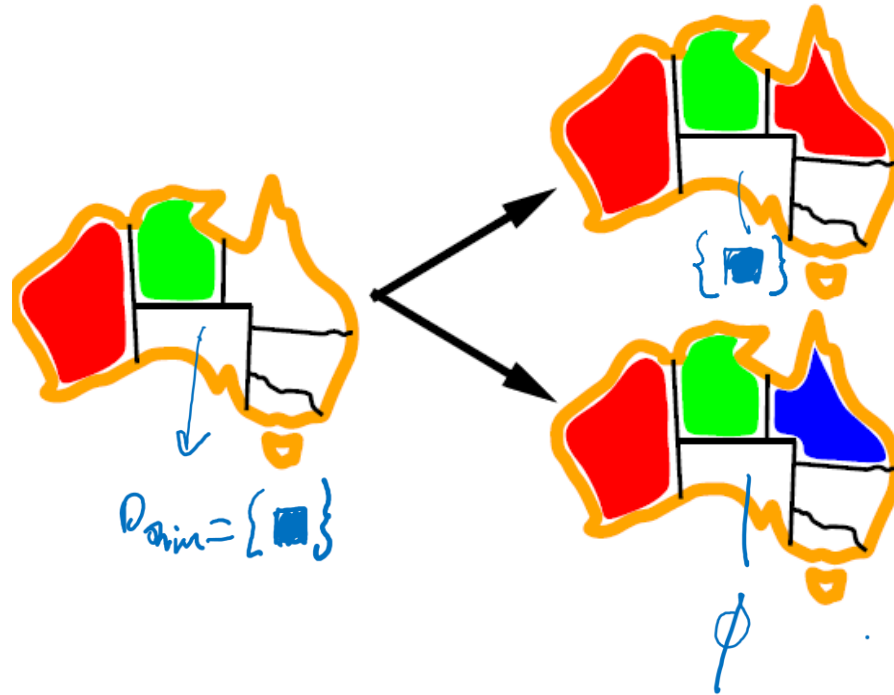
Variable Ordering

Degree heuristic

Choose variable that is involved in the largest number of constraint.

Could be a good tie-breaking strategy along with MRV.

Value Ordering




Value Ordering

Least Constrained Value (LCV) heuristic

Choose the value that rules out the fewest values in the remaining variables.

Can be estimated by forward checking
or arc-consistency checking



Why?

- Needs to choose some value
- Choosing value most likely to lead to solution
- Unlike variable ordering, there is no need to consider all values

Ordering

Minimum Remaining Value (MRV) heuristic

Choose variable that has the fewest left values in its domain.

Degree heuristic

Choose variable that is involved in the largest number of constraint.

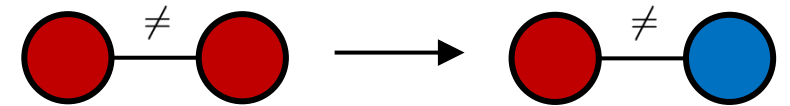
Least Constrained Value (LCV) heuristic

Choose the value that rules out the fewest values in the remaining variables.

Local Search

Iterative Improvement

- Start from some complete assignment that may not satisfy all constraints
- Modify the assignment, trying to resolve violations



Iterative Improvement

MinConflict (MaxSteps):

$x \leftarrow$ an initial complete assignment

for iter = 1 to MaxStep:

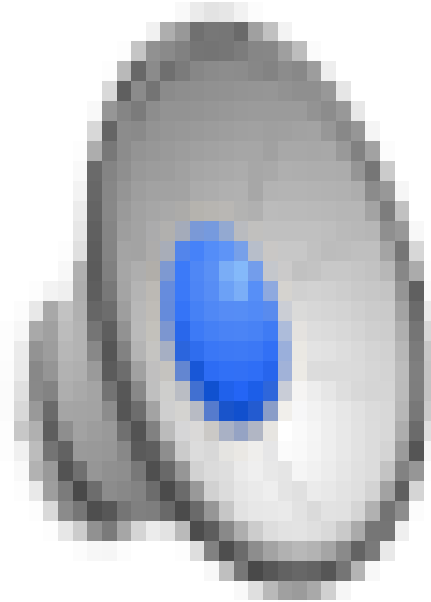
if x is a solution **then return** x

 Let X_i be a randomly chosen conflicted variable

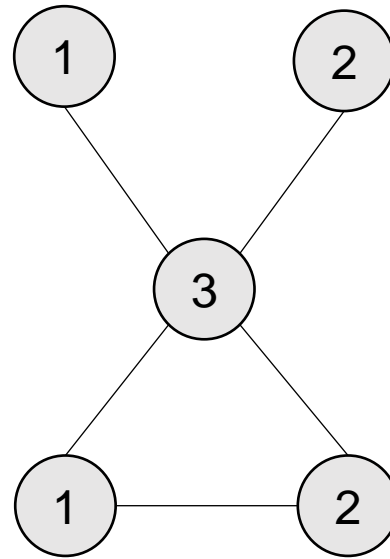
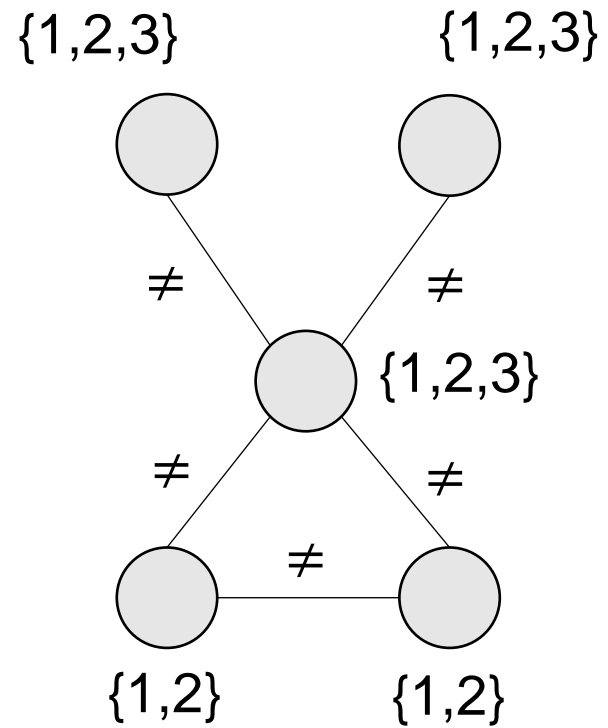
 Let v be the value for X_i that **minimizes conflicts**

 Reassign $X_i = v$ in x

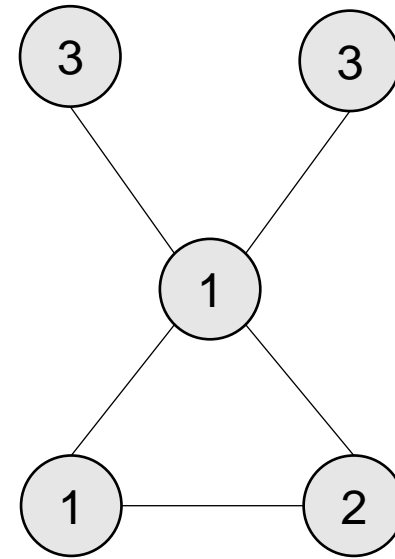
Iterative Improvement



Failure of MinConflict



A feasible solution



An unfortunate initialization for MinConflict

Rescue

For local search algorithms like MinConflict, we will randomly generate multiple initial assignments. For each initial assignment, we will only run it for MaxStep iterations before giving up.

Homework 2

Xuhui Kang, Matthew Landers

Deadline: 11:59PM, September 30

Homework 2

1. Choice Questions (10 points)
 - a. 10 questions.
 - b. Answer directly on Gradescope
 - c. The same requirements as the last time.
 - d. 1 feedback box for the course

1. Program Questions (25 points)

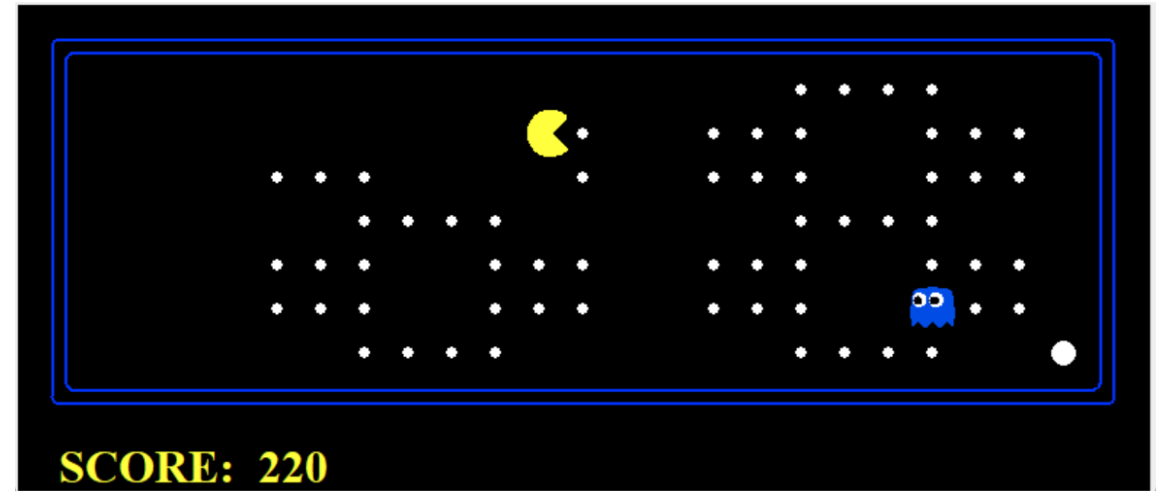
Question 1 (4 points): Reflex Agent

Your agent should easily and reliably clear the layout:

Run 10 times for evaluation:

TO receive full credits:

- Wins all the 10 times
- Average score is greater than 1k.

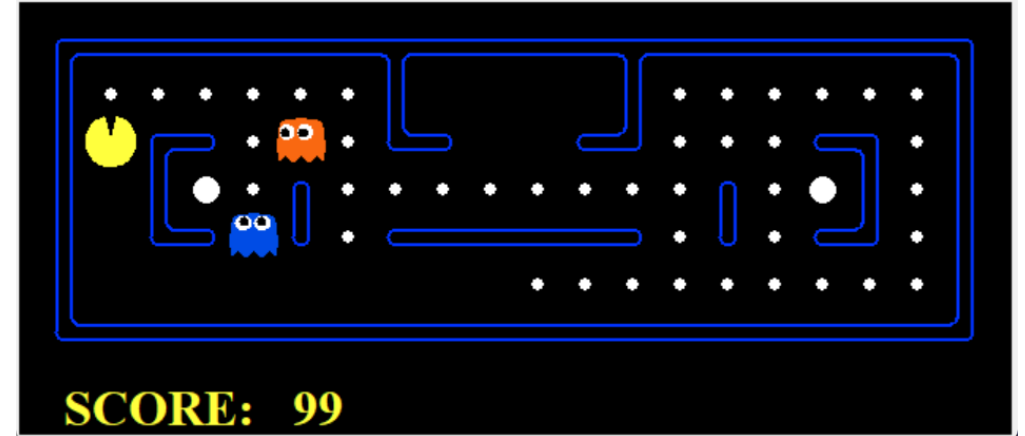


Question 2 (5 points): Minimax

You will write an adversarial search agent.

Work with any number of ghosts:

- Check the number of ghosts first.
- Index 0 is always the Pacman.



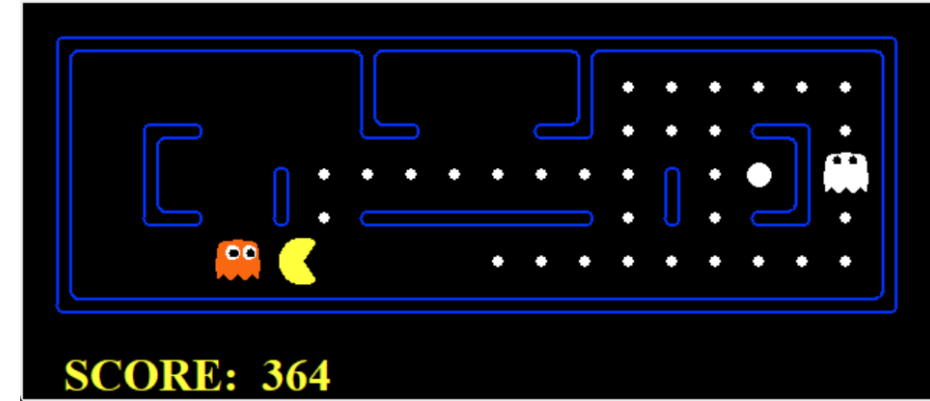
Correct implementation will lead to Pacman losing the game in some tests.

In large boards, Pacman will be good at not dying but bad at winning. It is not a problem.

When the Pacman believes that his death is unavoidable. Hw will try to end the game as soon as possible.

Question 3 (5 points): Alpha-Beta Pruning

Uses alpha-beta pruning to more efficiently explore the minimax tree.



Alpha-Beta Implementation

α : MAX's best option on path to root
 β : MIN's best option on path to root

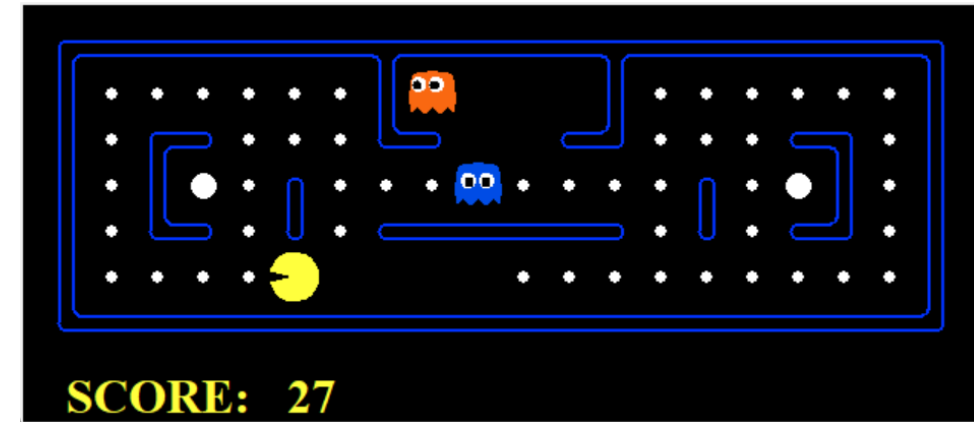
```
def max-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize  $v = -\infty$   
    for each successor of state:  
         $v = \max(v, \text{value}(\text{successor}, \alpha, \beta))$   
        if  $v > \beta$  return  $v$   
         $\alpha = \max(\alpha, v)$   
    return  $v$ 
```

```
def min-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize  $v = +\infty$   
    for each successor of state:  
         $v = \min(v, \text{value}(\text{successor}, \alpha, \beta))$   
        if  $v < \alpha$  return  $v$   
         $\beta = \min(\beta, v)$   
    return  $v$ 
```

Question 4 (5 points): Expectimax

Implement Expectimax Agent to modeling the probabilistic behavior of agents who may make suboptimal choices.

Correct implementation will lead to Pacman losing the game in some tests.



Question 5 (6 points): Evaluation Function

The evaluation function should evaluate states, rather than actions like your reflex agent evaluation function did.

You can use your search code in last project for evaluation.

