Constraint Satisfaction

Chen-Yu Wei

Revised Assignment Policies

- 7 free late days (originally 5)
- Late days for each homework cannot exceed 7
  - We will release / discuss solutions one week after the deadline
  - **Exception:** Homework3.choices does not allow any late submission.
- Further extensions are made in very rare cases
  - Doctor proof or proof of emergency is required
- We count the late days for the choices part and programming part separately
  - Each part has 7 free late days
- Need NOT notify me when you want to use free late days

Constraint Satisfaction Problems (CSP)

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

Example: Map Coloring

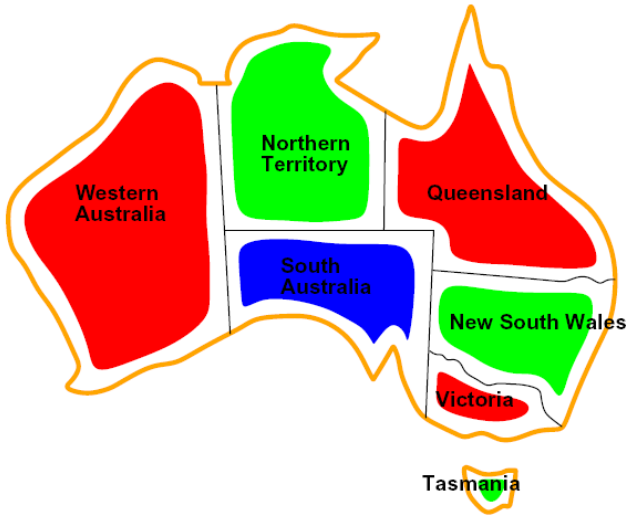
- Variables: $\mathsf{WA,\ NT,\ Q,\ NSW,\ V,\ SA,\ T}$
- Domains: $D = \{\mathsf{red}, \mathsf{green}, \mathsf{blue}\}$
- Constraints: adjacent regions must have different colors

  Implicit: $\mathsf{WA} \neq \mathsf{NT}$

  Explicit: $(\mathsf{WA}, \mathsf{NT}) \in \{(\mathsf{red}, \mathsf{green}), (\mathsf{red}, \mathsf{blue}), \ldots\}$

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

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



Example: Cryptarithmetic

- Variables:

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

- Domains:

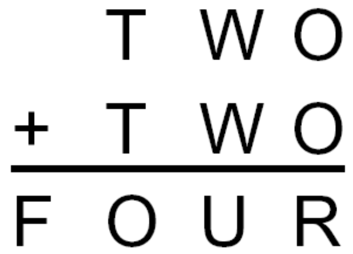
$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

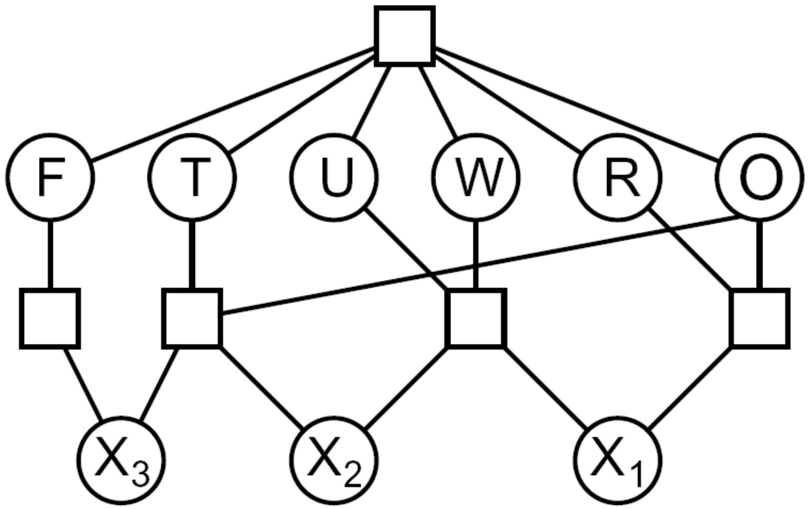
- Constraints:

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

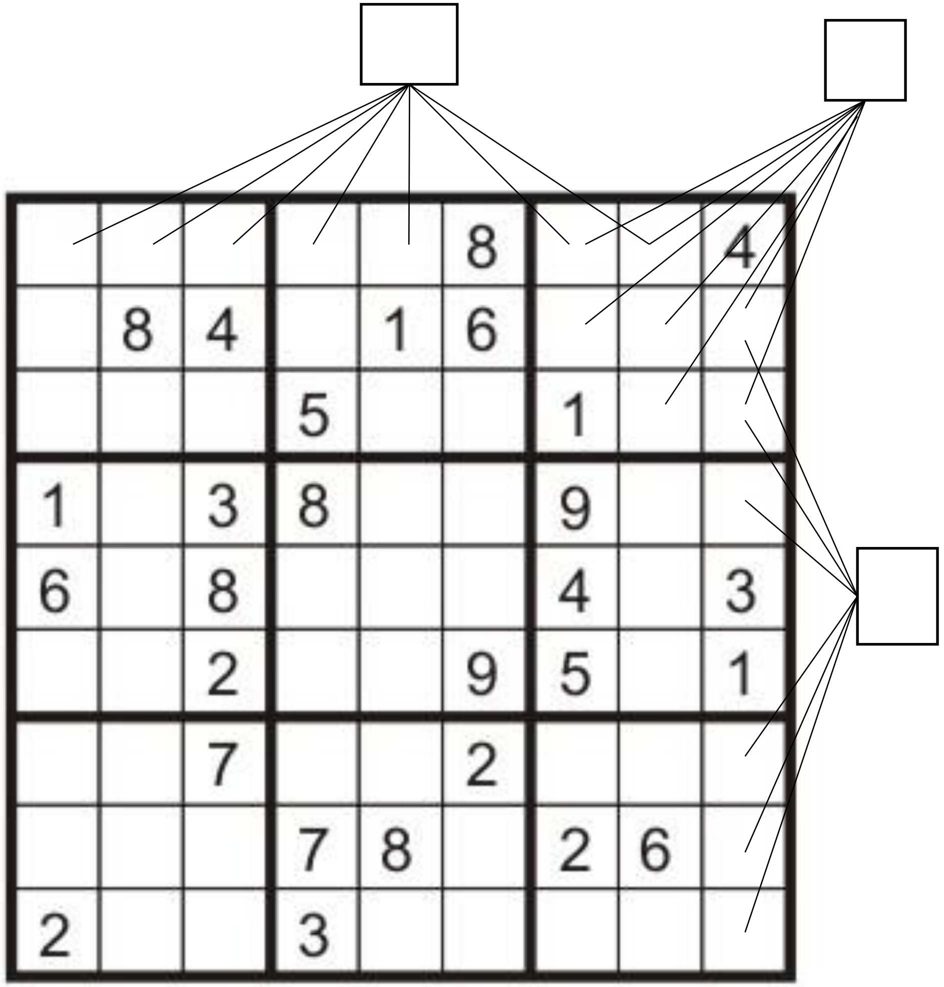
$$O + O = R + 10 \cdot X_1$$

$$\ldots$$





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)

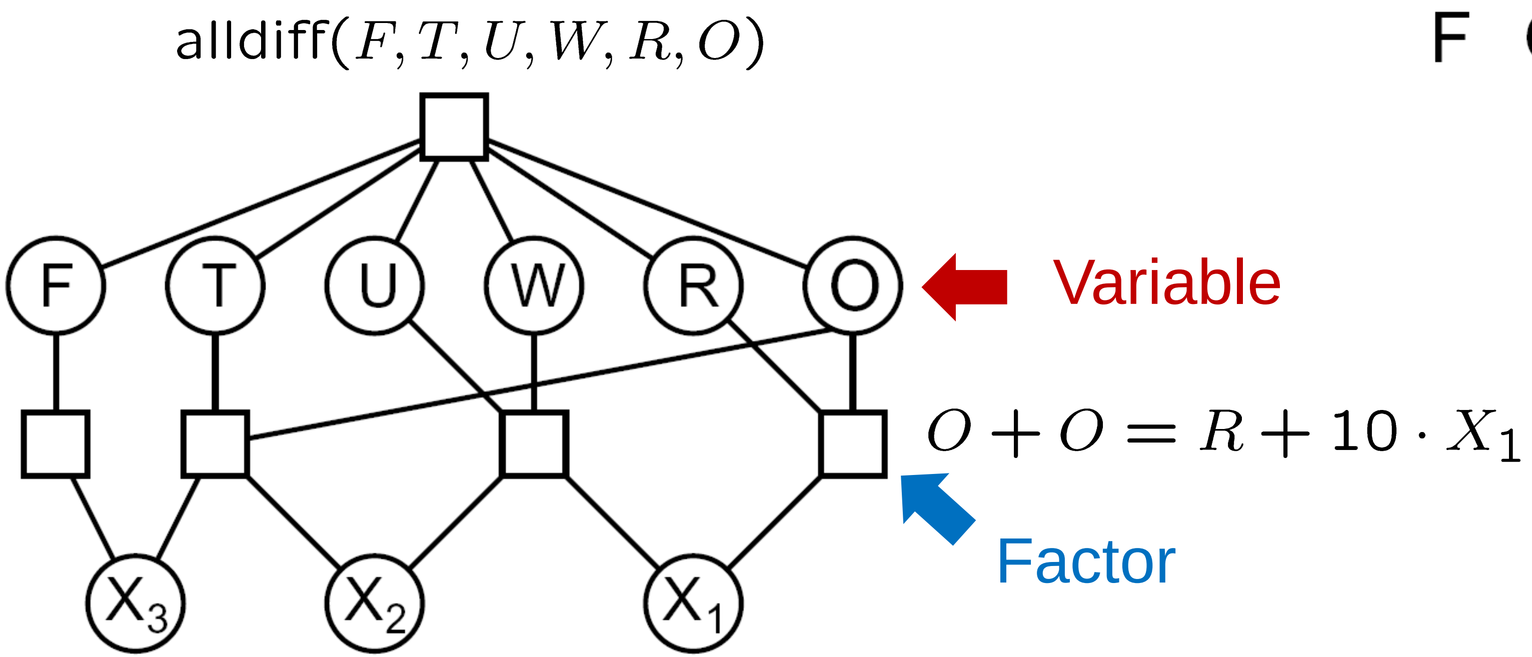
Real-World CSPs

- 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

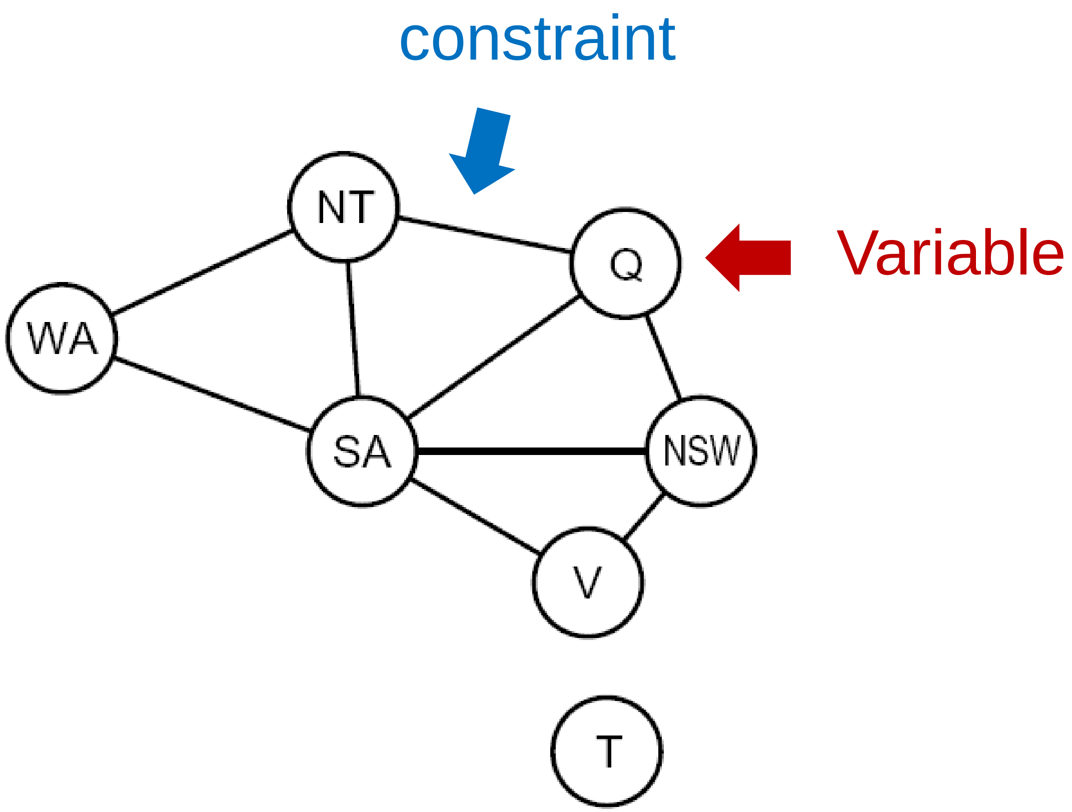
```
   T W O
+  T W O
--------
 F O U R
```

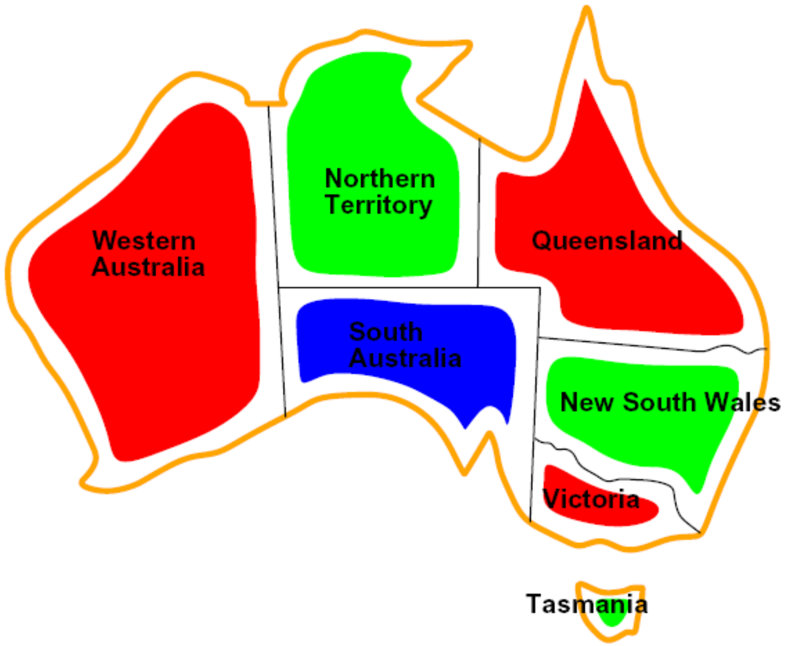


$$f_1(O, R, X_1) = \begin{cases} 1, & \text{if } O + O = R + 10X_1 \\ 0, & \text{otherwise} \end{cases}$$

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

Constraint Graph





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

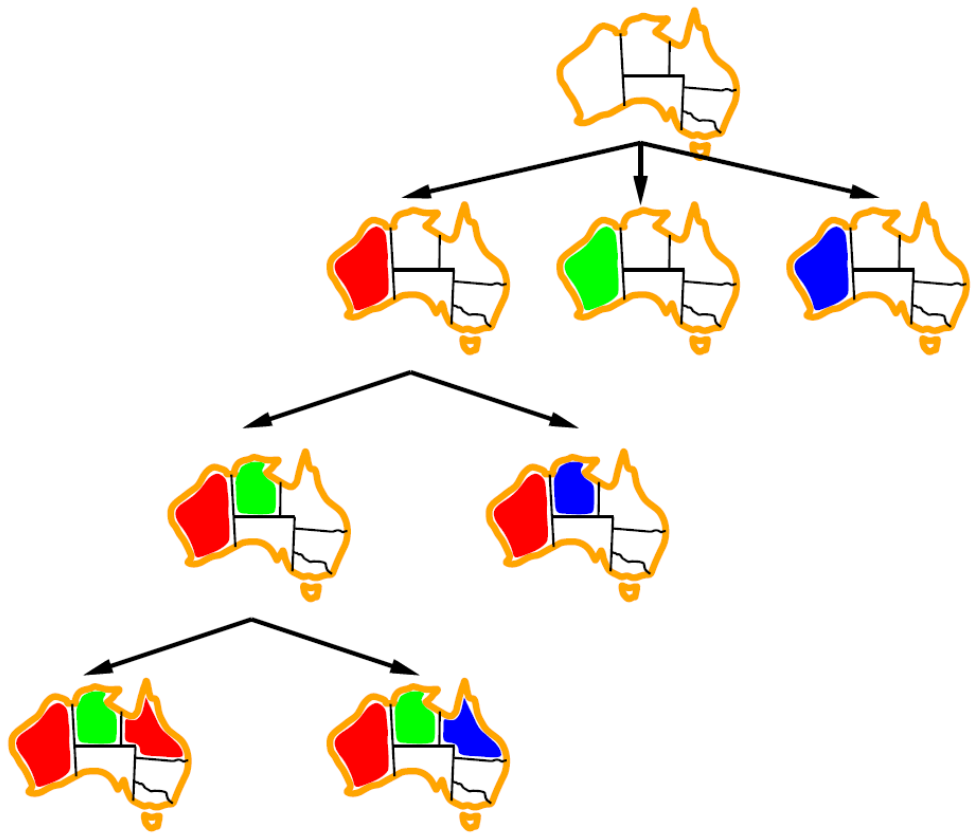
Every constraint involves at most 2 variables

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 constraint 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



X_1, X_2, X_3, X_4

$x = \{X_1 : 3,\ X_3 : 2\}$

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 $v \in \mathrm{Domain}_i$:

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

If x' violates constraints: **continue**

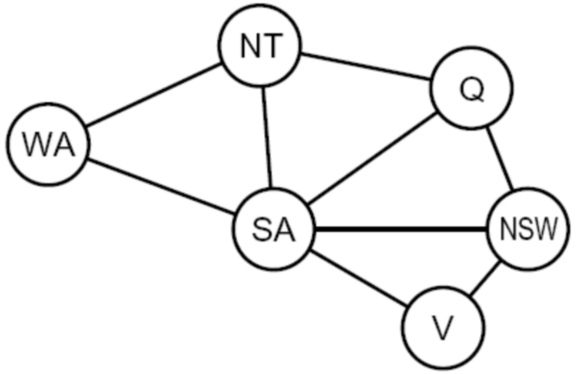
return **BacktrackingSearch**(x', Domain)

return failure

Improving Backtracking Search

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

Vanilla Backtracking Search

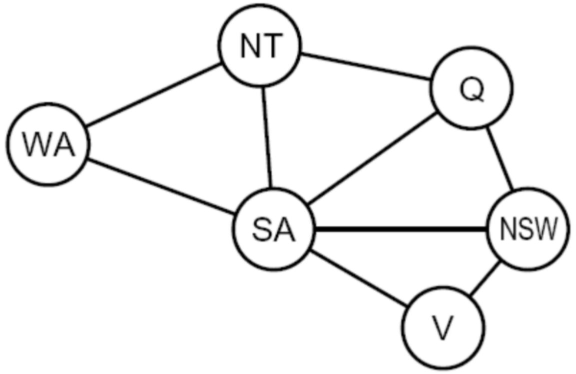


Suppose we assign the variables in the order of

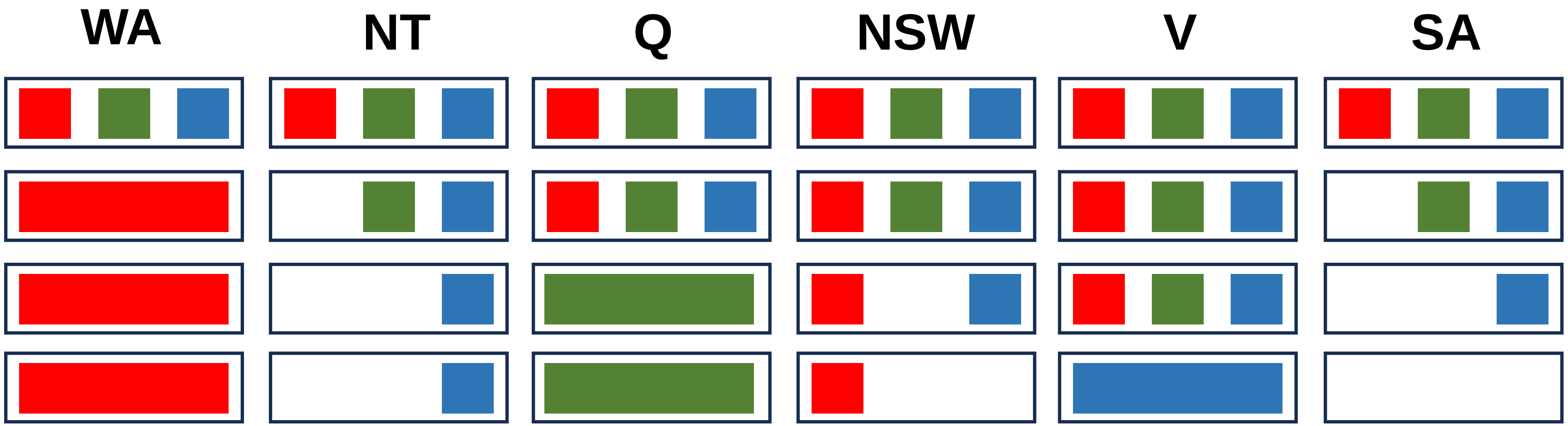
WA, Q, V, NT, NSW, SA

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



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 $v \in \mathrm{Domain}_i$:

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

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 $v \in \text{Domain}_i$:

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

$\text{Domain}', \ \text{Consistent} =$ **ForwardChecking**$(x', X_i, v, \text{Domain})$

If not Consistent: **continue**

return BacktrackingSearch(x', Domain')

return failure

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

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

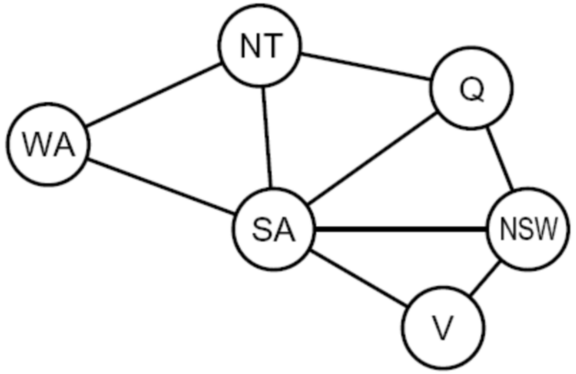
For all X_j that is unassigned in x' and connected to X_i:

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

If Domain'_j is empty: return $\text{Domain}', \mathit{False}$

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

Can We Prune Even More?



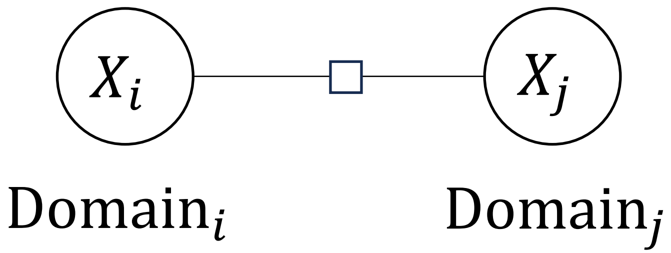
With forward checking:

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



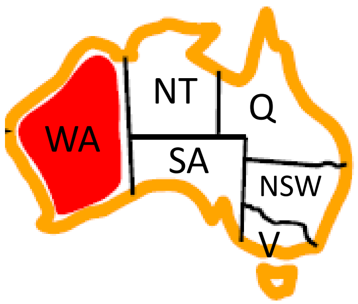
Fact. Let $v \in \mathrm{Domain}_i$ be such that for all $w \in \mathrm{Domain}_j$, $\{X_i : v,\ \ X_j : w\}$ violates the constraint on (X_i, X_j). Then we can remove v from Domain_i.

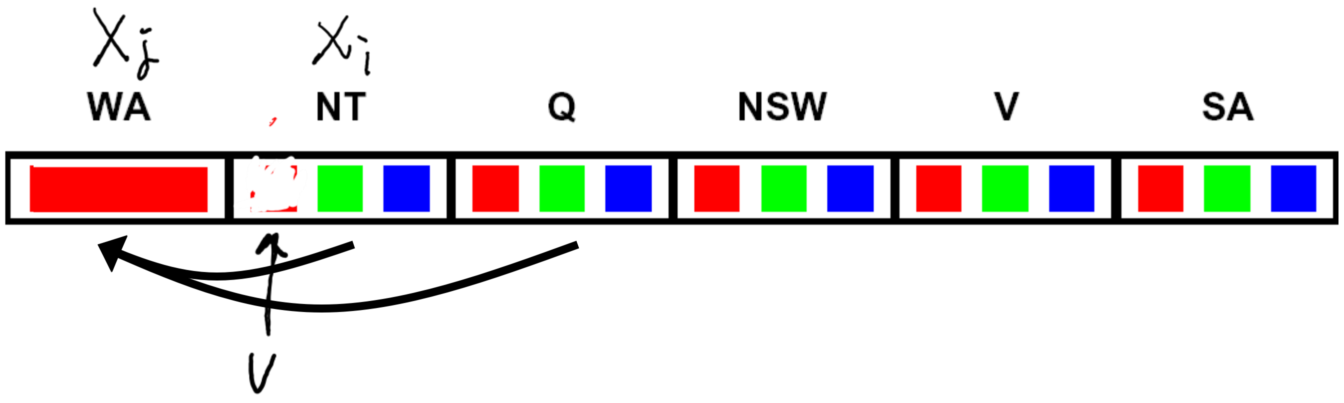
Definition (Arc Consistency on $X_i \to X_j$). For all $v \in \mathrm{Domain}_i$, there is some $w \in \mathrm{Domain}_j$ such that $\{X_i : v,\ \ X_j : w\}$ satisfies the constraint on (X_i, X_j).

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.

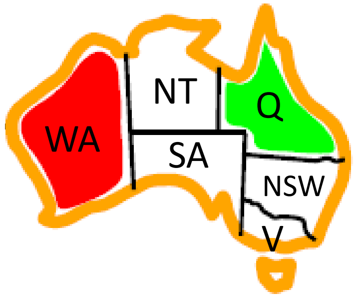


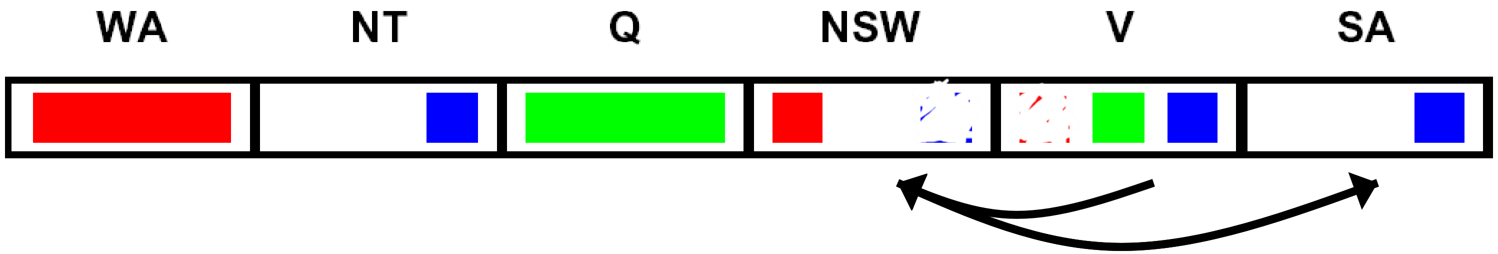


Maintaining Arc Consistency (MAC)

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!
- Arc consistency detects failure earlier than forward checking.
- What's the downside of enforcing arc consistency?

Maintaining Arc Consistency (MAC)

BacktrackingSearch(x, Domain):

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

Let X_i be the next unassigned variable.

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

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

$\mathrm{Domain}', \ \mathrm{Consistent} =$ **AC3**$(x', X_i, v, \mathrm{Domain})$

If not $\mathrm{Consistent}$: **continue**

return BacktrackingSearch(x', Domain')

return failure

Maintaining Arc Consistency (MAC)

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

Domain′ ← Domain

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

while queue not empty:

$(X_k, X_\ell) \leftarrow$ POP(queue)

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

Revise Domain'_k to make it consistent

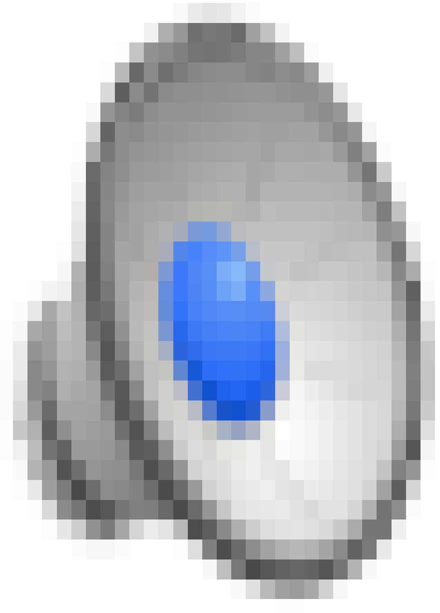
if Domain'_k is empty: **return** $\text{Domain}', False$

for each $X_m \neq X_\ell$ that is unassigned in x' and connected to X_i:

add (X_m, X_k) to queue

return $\text{Domain}', True$

Video: Forward Checking



Video: Maintaining Arc Consistency

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.

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

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

$\text{Domain}', \text{ Consistent} =$ **AC3**$(x', X_i, v, \text{Domain})$

If not Consistent: **continue**

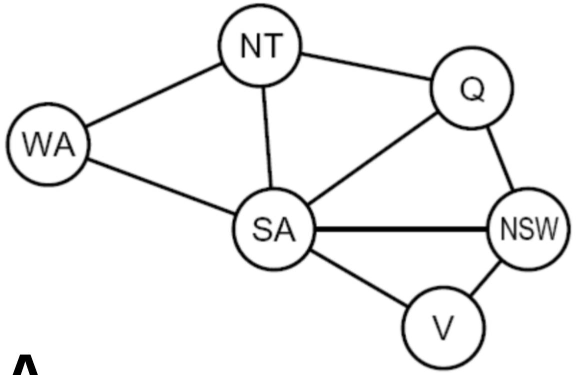
return **BacktrackingSearch**(x', Domain')

return failure

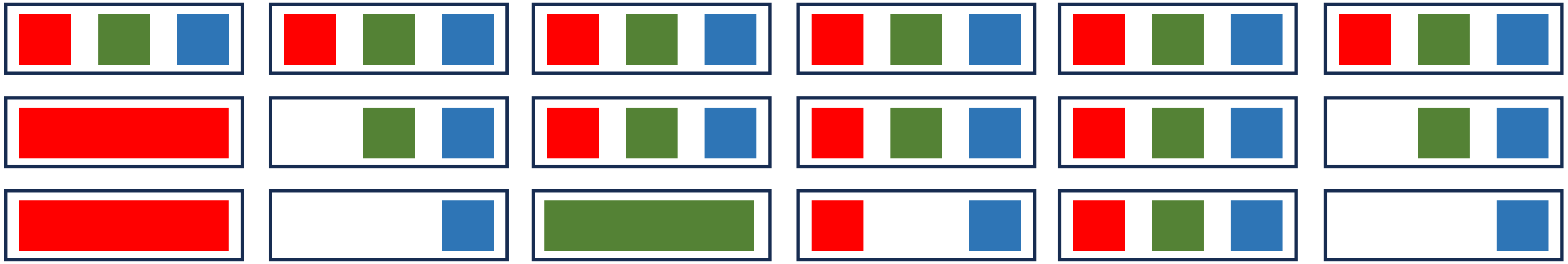
Which **variable** should we pick first?

Which **value** should we try first?

Variable Ordering



WA NT Q NSW V SA



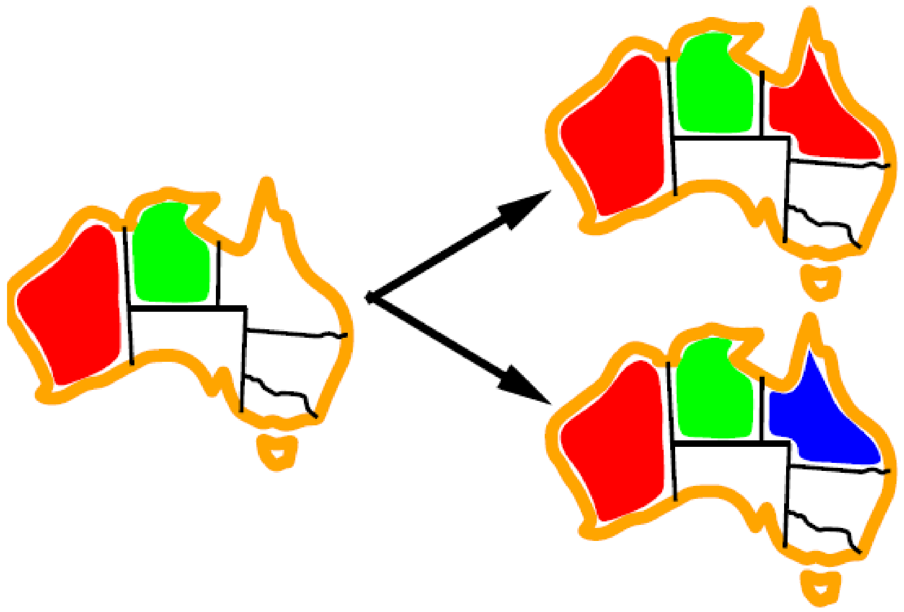
Variable Ordering

Most Constrained Variable (MCV)

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

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

Value Ordering



Value Ordering

Least Constrained Value (LCV)

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

Why?

- Needs to choose some value
- Choosing value most likely to lead to solution

Ordering

Most Constrained Variable (MCV)

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

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

Least Constrained Value (LCV)

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

- Needs to choose some value
- Choosing value most likely to lead to solution

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)

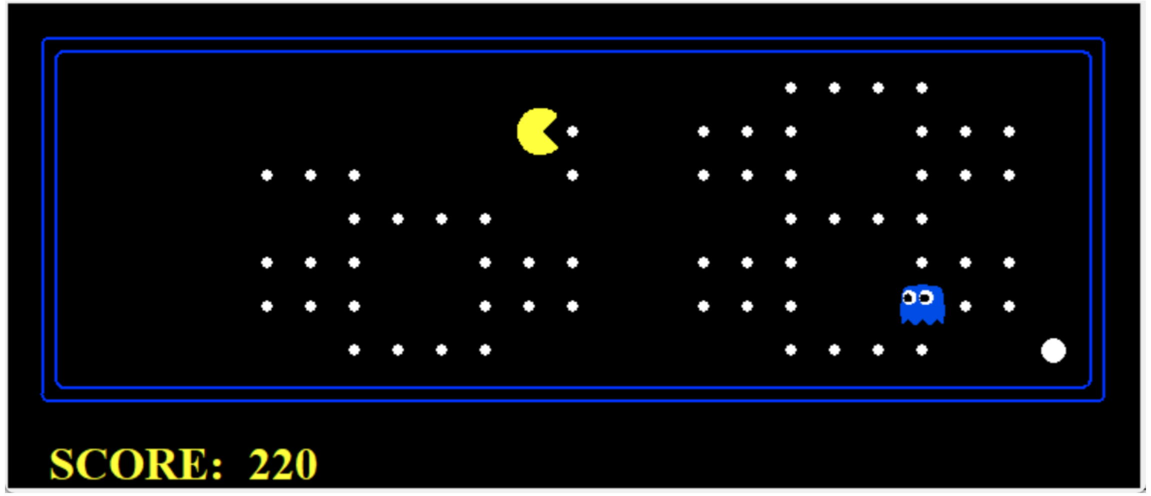
Question 1 (4 points): Reflex Agent

Your agent should easily and reliably clear the layout:

Run 10 times for evaluation:

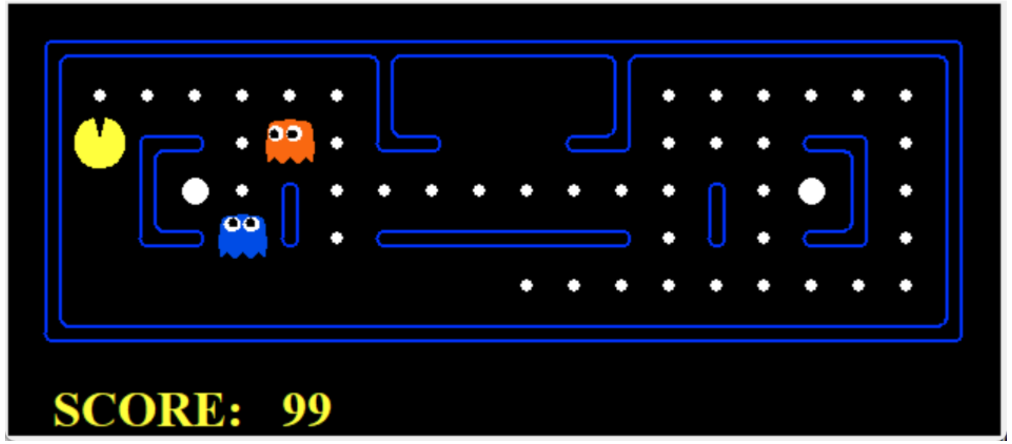
TO receive full credits:

- 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.

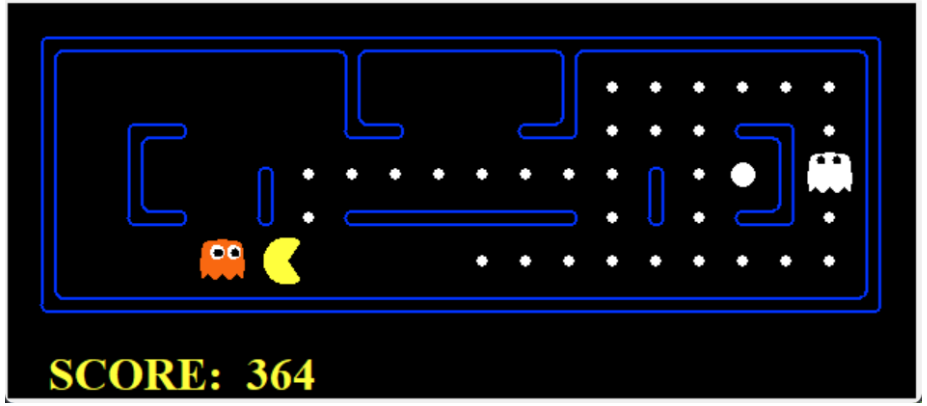
Correct implementation will lead to Pacman losing the game in some tests.

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

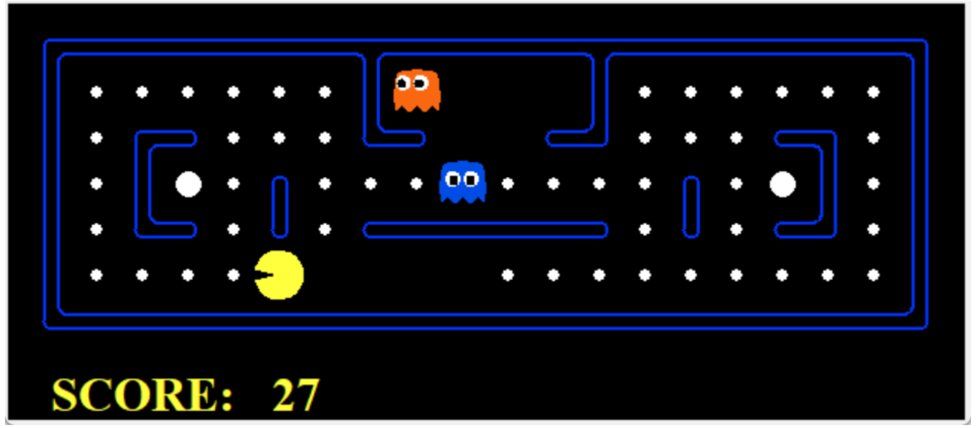
α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v > β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v < α return v
        β = min(β, 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.