Artificial Intelligence

CS4365 --- Fall 2022

Midterm 2 Review

Instructor: Yunhui Guo

Midterm 2

- 5 problems

- Topics:
  - Constraint satisfaction problem
  - Min-max search
  - Propositional logic

Constraint Satisfication Problems (CSP)

- A powerful representation for (discrete) search problems.

- A Constraint Satisfication Problem (CSP) is defined by:
  - X is a set of n variables X_1, X_2, ..., X_n, each defined by a finite domain D_1, D_2, ..., D_n of possble values
  - C is a set of constraints C_1, C_2, ..., C_m. Each C_i involves some subset of the variables; specifices the allowable combinations of values for that subset.

A solution is an assigment of values to the variables that satisfies all constraints.

Map-Coloring Problem

- Variables:

  WA, NT, Q, NSW, V, SA, T

- Domains:

  {red, green, blue}

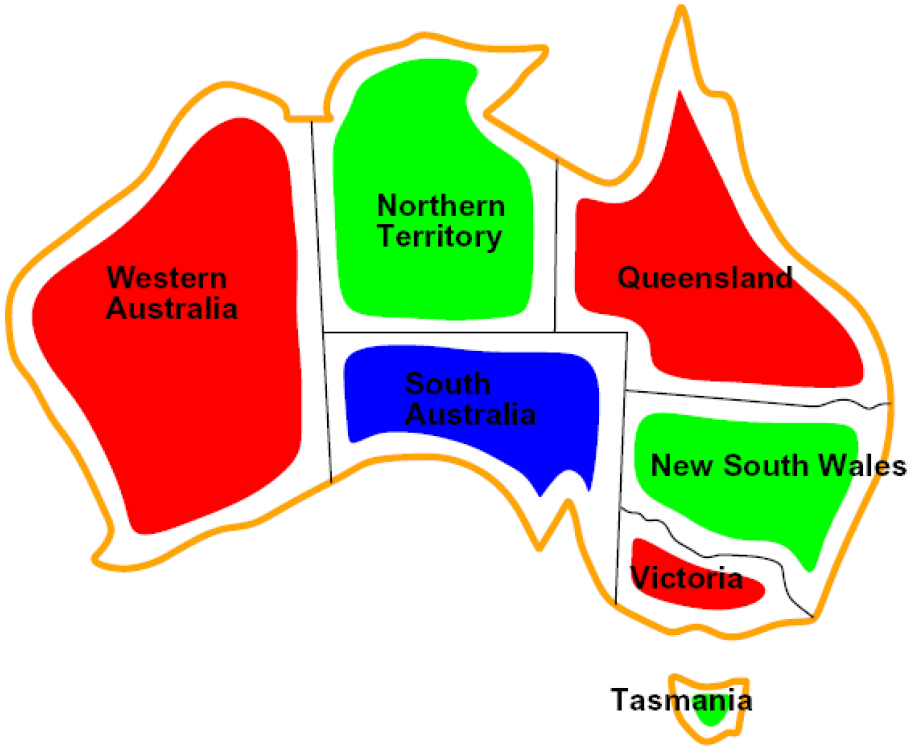
- Constraints:

  Adjacent regions must have diffe-

  rent colors

  Implicit: WA ≠ NT

  Explicit: (WA, NT) ∈ {(red, green), {red, blue}, ...}



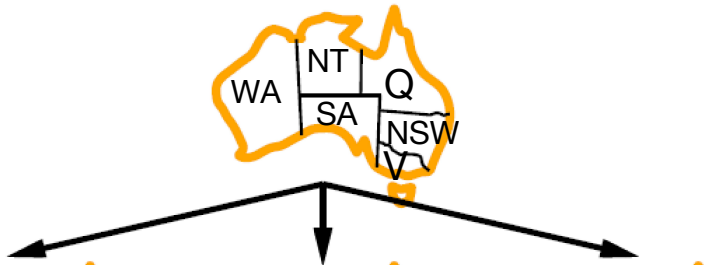
How to View a CSP as a Search Problem?

- Initial State -- state in which all the variables are unassigned.
- Successor function -- assign a value to a variable from a set
- Goal test -- check if all the variables are assigned and all the constraints are satsified. We call the assignment is complete.
- Path cost -- assumes constant cost for each step

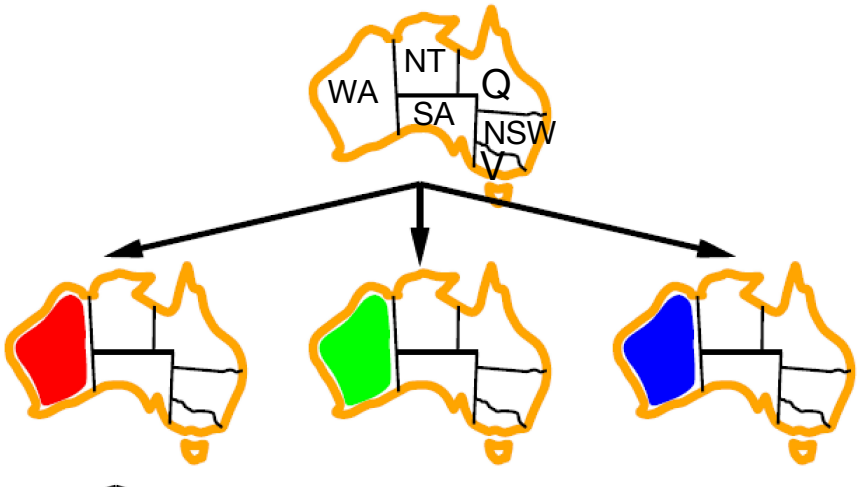
The Backtracking Search Algorithm

```
Backtrack(assignment, csp):
    If assignment is complete then return assignment
    var ← select one unassigned variable
    for each value in the domain of var do
        if value is consistent with assignment then
            add {var = value} to assignment
            result ← Backtrack(assignment, csp)
            if result ≠ failure then
                return result
            remove {var = value} from assignment
    return failure
```

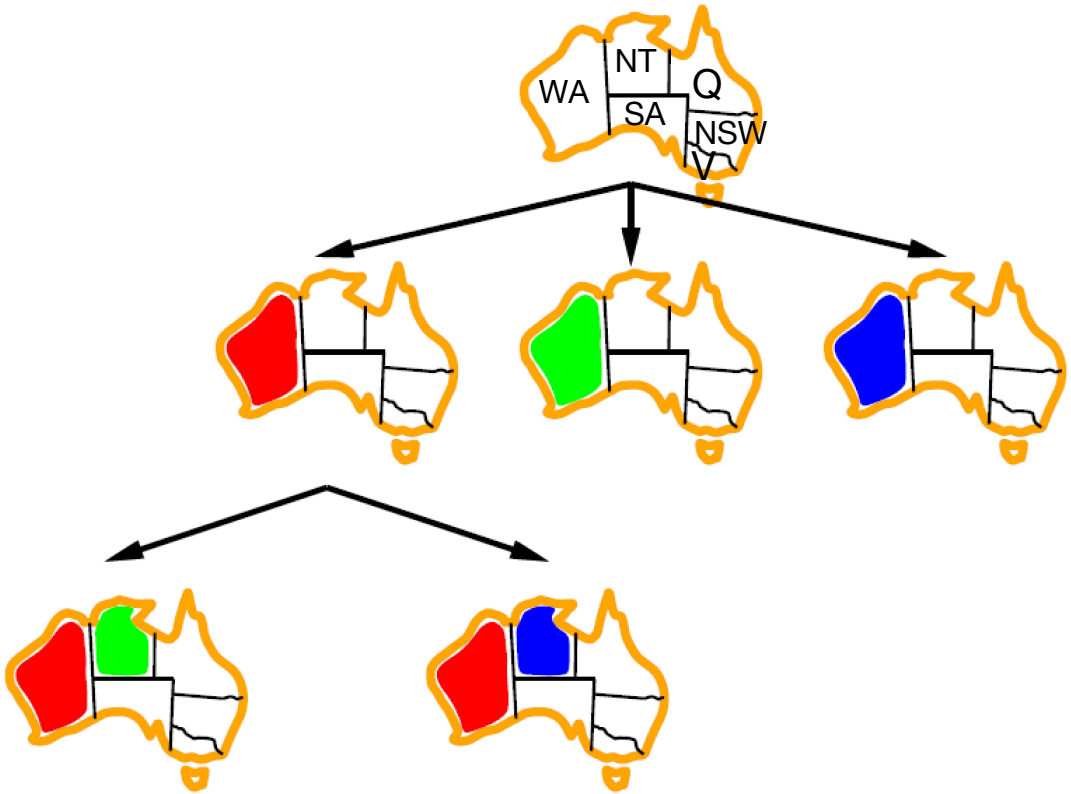
Backtracking Example



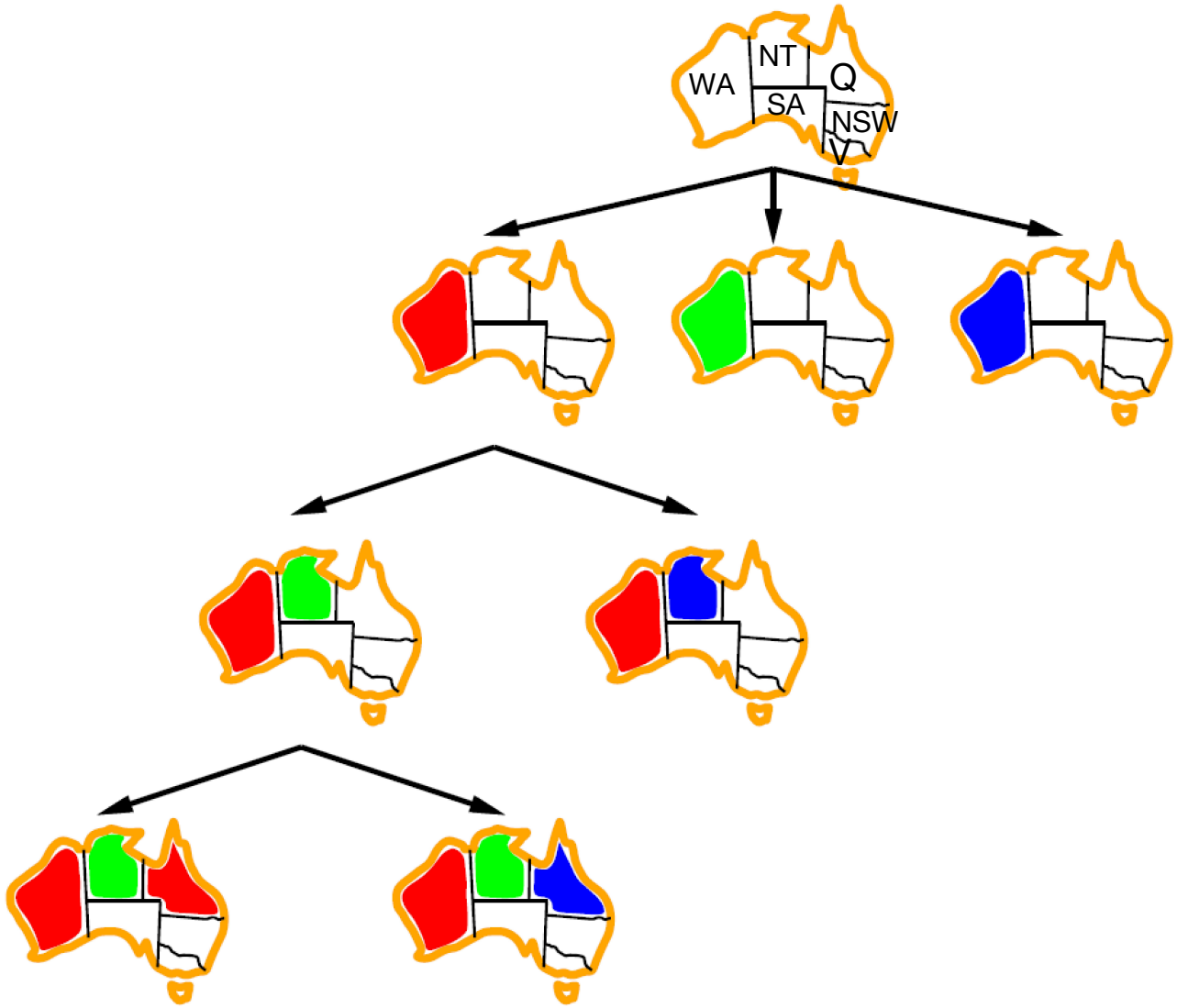
Backtracking Example



Backtracking Example



Backtracking Example



The Backtracking Search Algorithm

```
Backtrack(assignment, csp):
    If assignment is complete then return assignment
    var ← select one unassigned variable  ← which one to select?
    for each value in the domain of var do ← which one to use?
        if value is consistent with assignment then
            add {var = value} to assignment     ← detect failure early
            result ← Backtrack(assignment, csp)
            if result ≠ failure then
                return result
            remove {var = value} from assignment
    return failure
```

Improving Backtracking Efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
   - Choose the most constrained variable, and break ties by choosing the most constraining variable

2. In what order should its values be tried?
   - Choose the least constraining value

3. Can we detect inevitable failure early?
   - Forward checking, constraint propagation

Improving Backtracking Efficiency

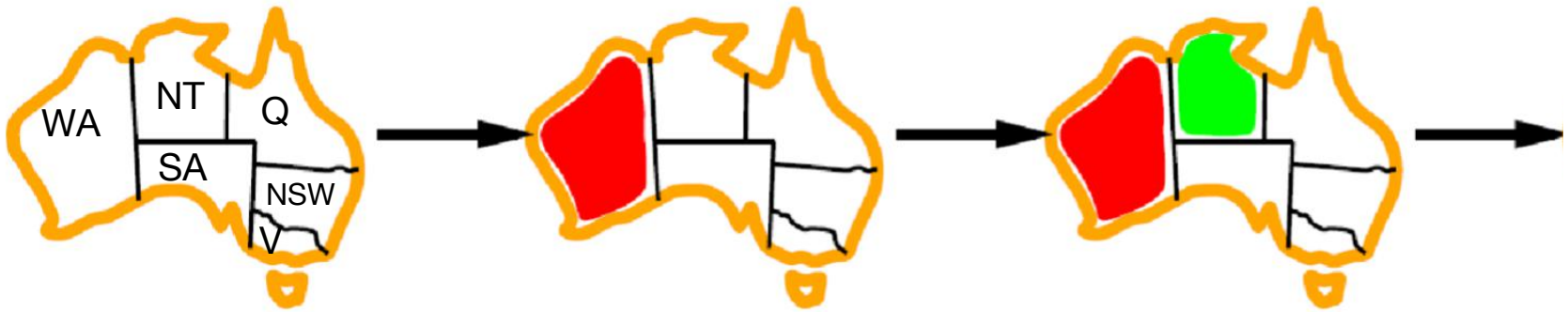
General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
   - Choose the most constrained variable, and break ties by choosing the most constraining variable

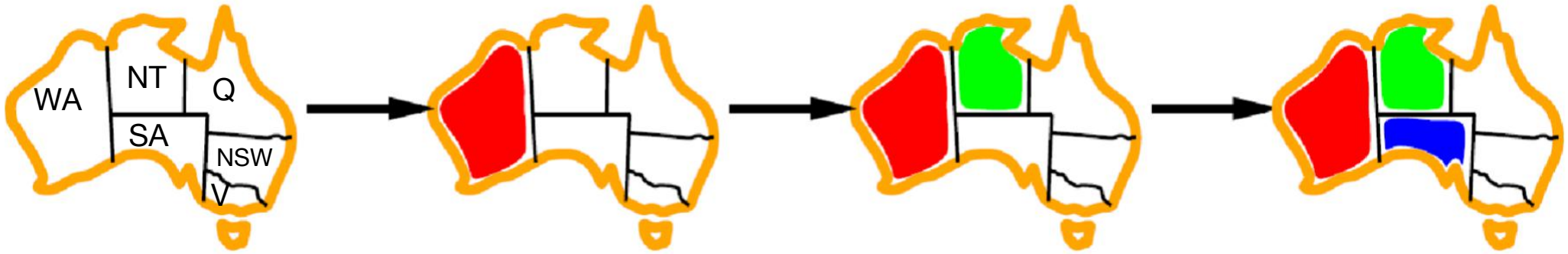
2. In what order should its values be tried?
   - Choose the least constraining value

3. Can we detect inevitable failure early?
   - Forward checking, constraint propagation

Variable Selection Heuristic 1: Most Constrained Variable



Variable Selection Heuristic 1: Most Constrained Variable



Most constrained variable:

Choose the variable with the fewest legal values

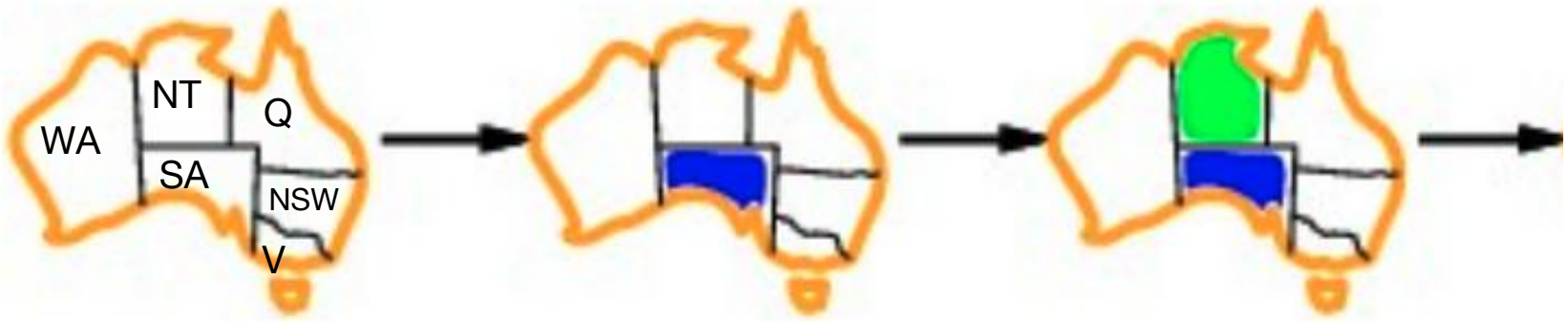
a.k.a. mininum remaining values (MRV) heuristic

- Why min rather than max?
- "Fail-fast" ordering
- 1000× faster or more

Reduce the branching factor

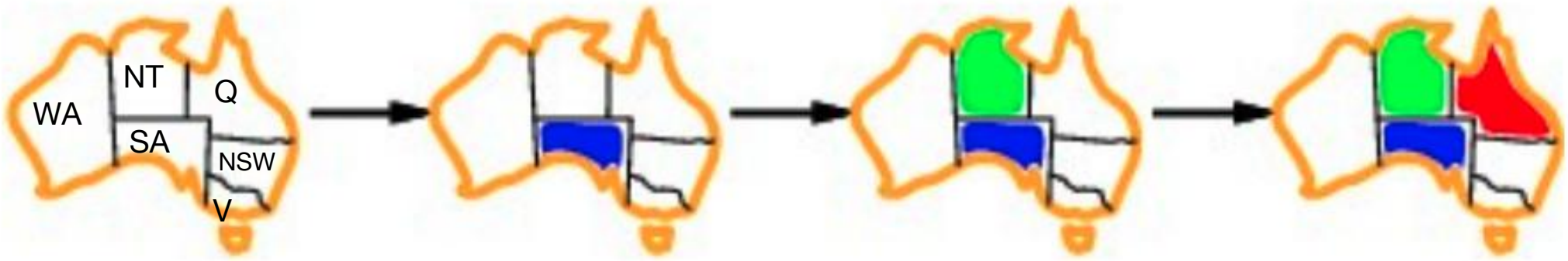
We use forward checking in this example, generally just based on the domain size

Variable Selection Heuristic 2: Most Constraining Variable



- Tie-breaker among most constrained variables

Variable Selection Heuristic 2: Most Constraining Variable



- Tie-breaker among most constrained variables

Most constraining variable:

choose the variable with the most constraints on remaining variables

Reduce branching factor in the future

Improving Backtracking Efficiency

General-purpose methods can give huge gains in speed:

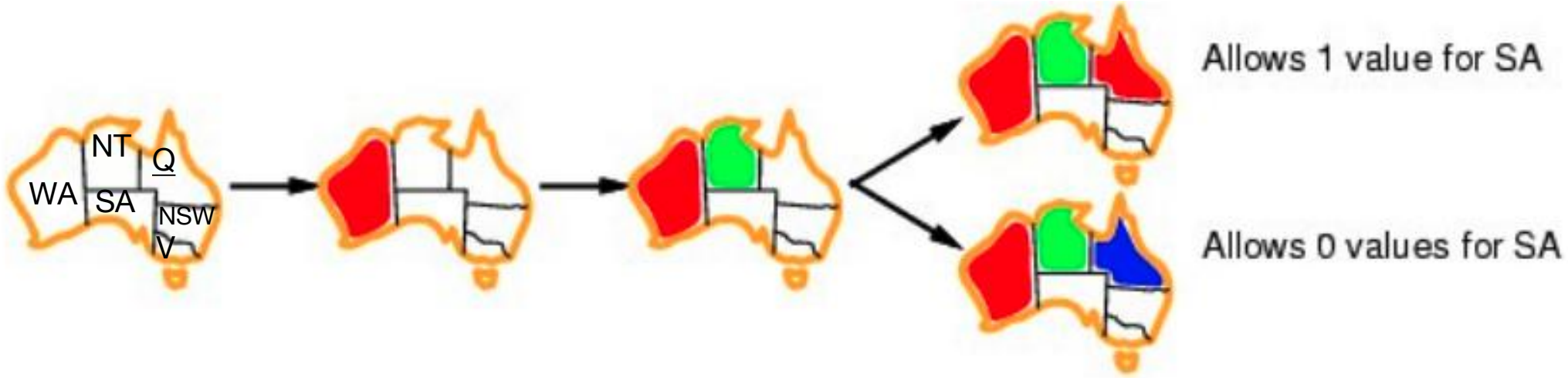
1. Which variable should be assigned next?
   - Choose the most constrained variable, and break ties by choosing the most constraining variable

2. In what order should its values be tried?
   - Choose the least constraining value

3. Can we detect inevitable failure early?
   - Forward checking, constraint propagation

Value Select Heuristic: Least Constraining Value

Value Select Heuristic: Least Constraining Value



- Given a variable, choose the least constraining value:

  the one that rules out the fewest values in the remaining (unassigned) variables

- Why least rather than most? More likely to find a solution early

Improving Backtracking Efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
   - Choose the most constrained variable, and break ties by choosing the most constraining variable

2. In what order should its values be tried?
   - Choose the least constraining value

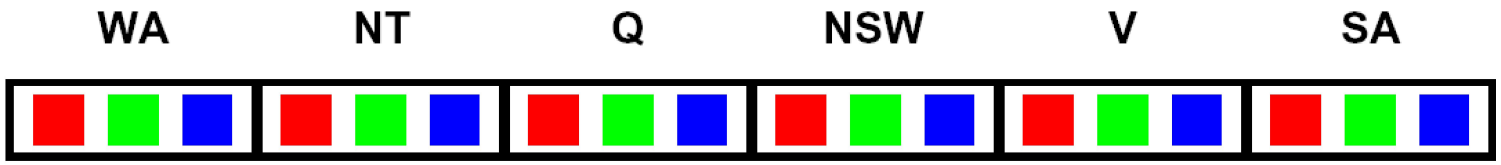
3. Can we detect inevitable failure early?
   - Forward checking, constraint propagation

Consistency-Enforcing Procedure 1: Forward Checking

- Idea:

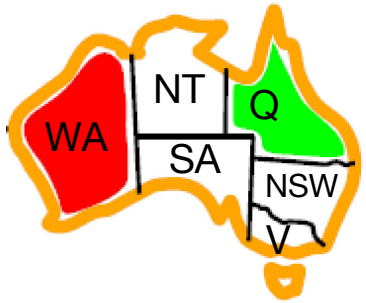
  Keep track of remaining legal values for unsigned variables

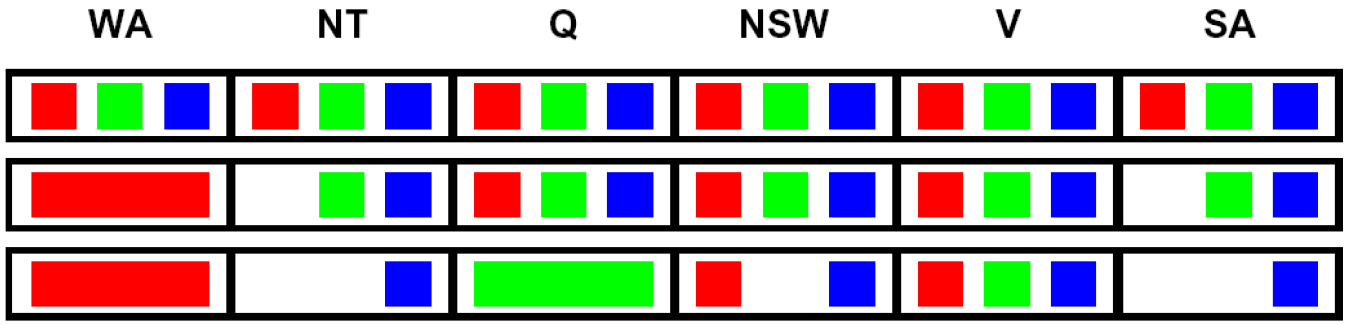
  Terminate search when any variable has no legal values



Consistency-Enforcing Procedure 2: Constraint Propagation

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

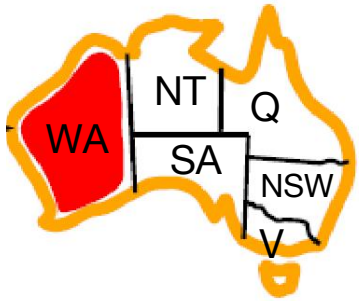


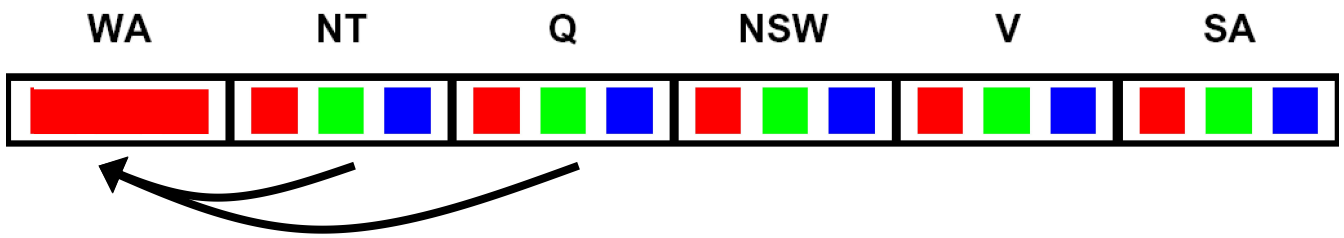


- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constrains locally.

Arc Consistency

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

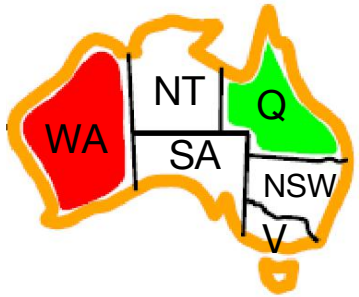


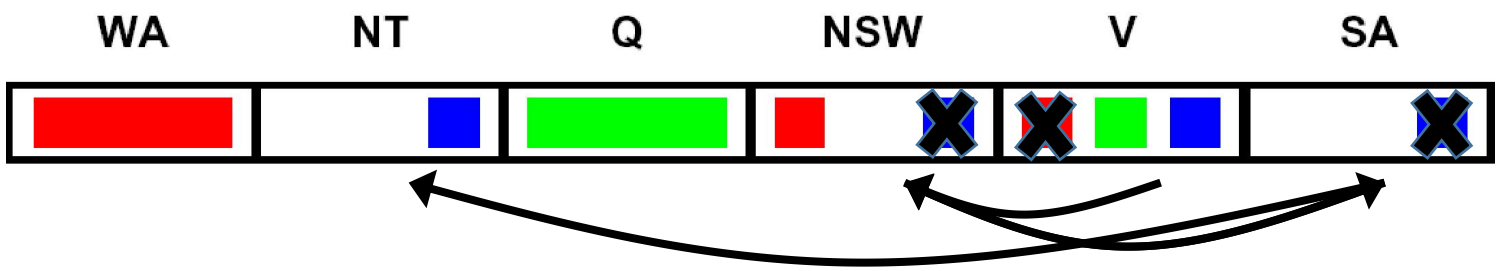


- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc Consistency

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





- 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

Consistency-Enforcing Procedures

- Forward checking
- Constraint propagation
- Variable/value ordering heuristics

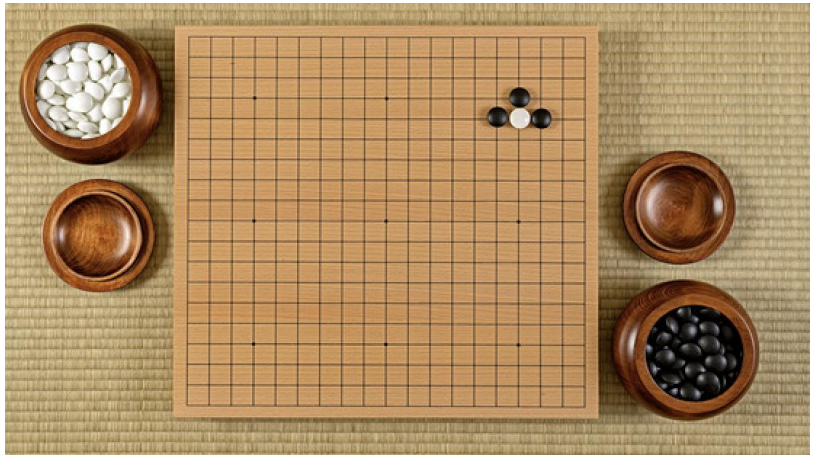
- They can used in combination to improve search process.

General Purpose CSP Algorithm

1. If all values have been successfully assigned then stop, else go on to 2.
2. Apply the consistency enforcing procedure (e.g. forward checking if feeling computationally mean, or constraint propagation if extravagant.)
3. If a deadend is detected then backtrack (e.g., DFS-type backtrack.). Else go on to step 4.
4. Select the next variable to be assigned (using your variable ordering heuristic)
5. Select a promising value (using your value ordering heuristic)
6. Create a new search state. Cache the info you need for backtracking. And go back to 1.

Deterministic Games

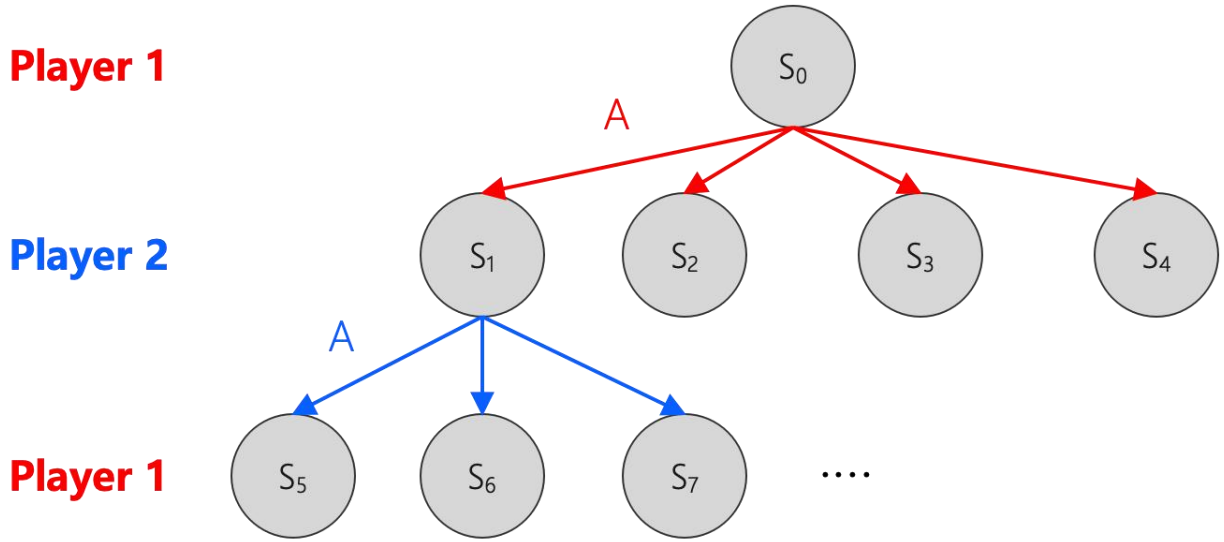
- One possible formulations
  - States: S (start at s_0)
  - Players: P = {1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: S x A → S
  - Terminal Test: S → {True, False}
  - Terminal Utilities: SxP → R



- Solution for a player is a policy: S → A

Game Playing as Search

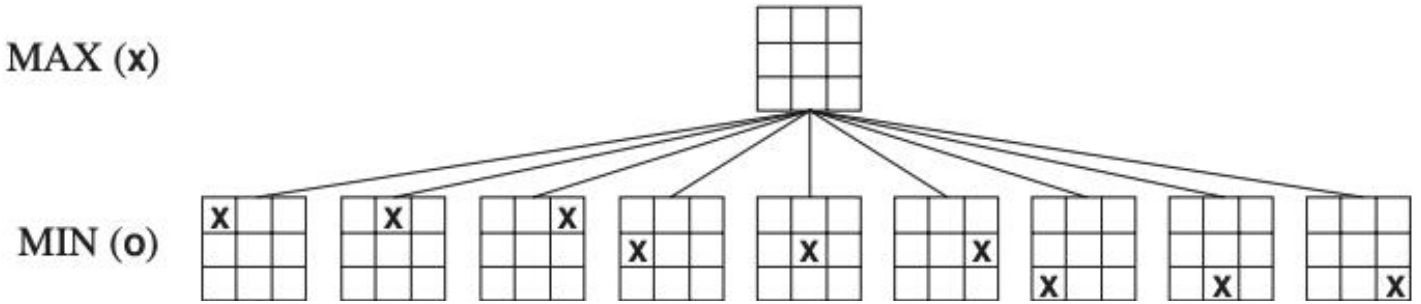
- We can list all the possible actions and states
- In each step, play 1 searches for an action which leads to the maximum utility



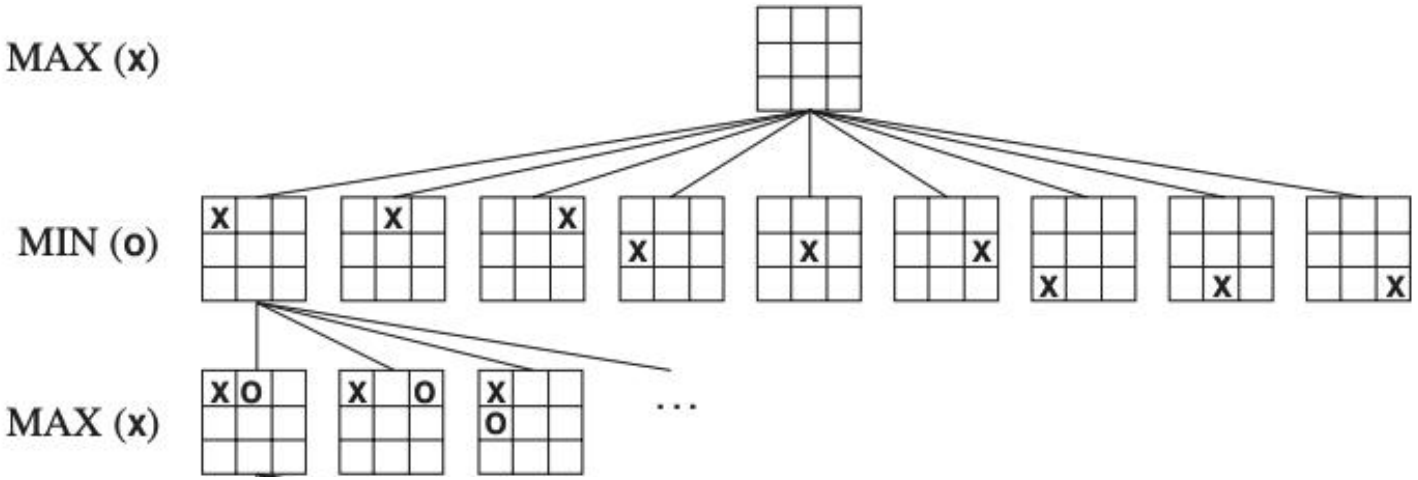
Search Tree for Tic-Tac-Toe

MAX (x)

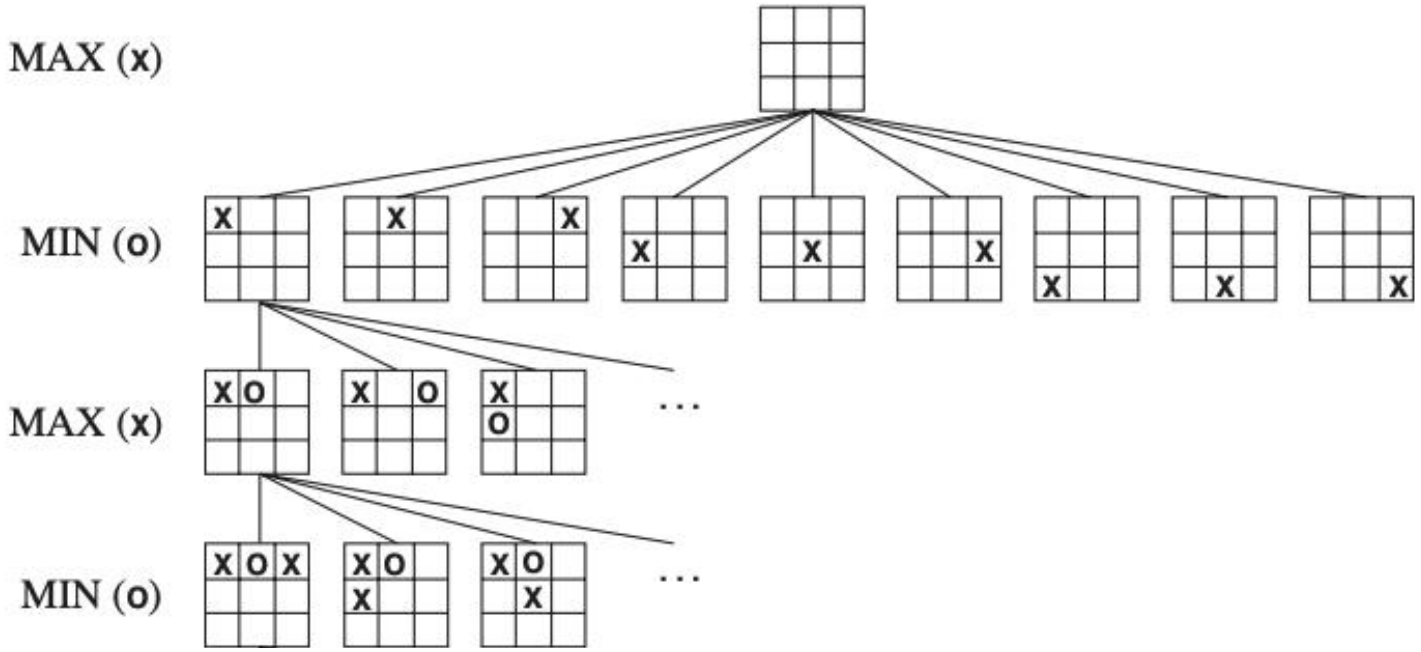
Search Tree for Tic-Tac-Toe



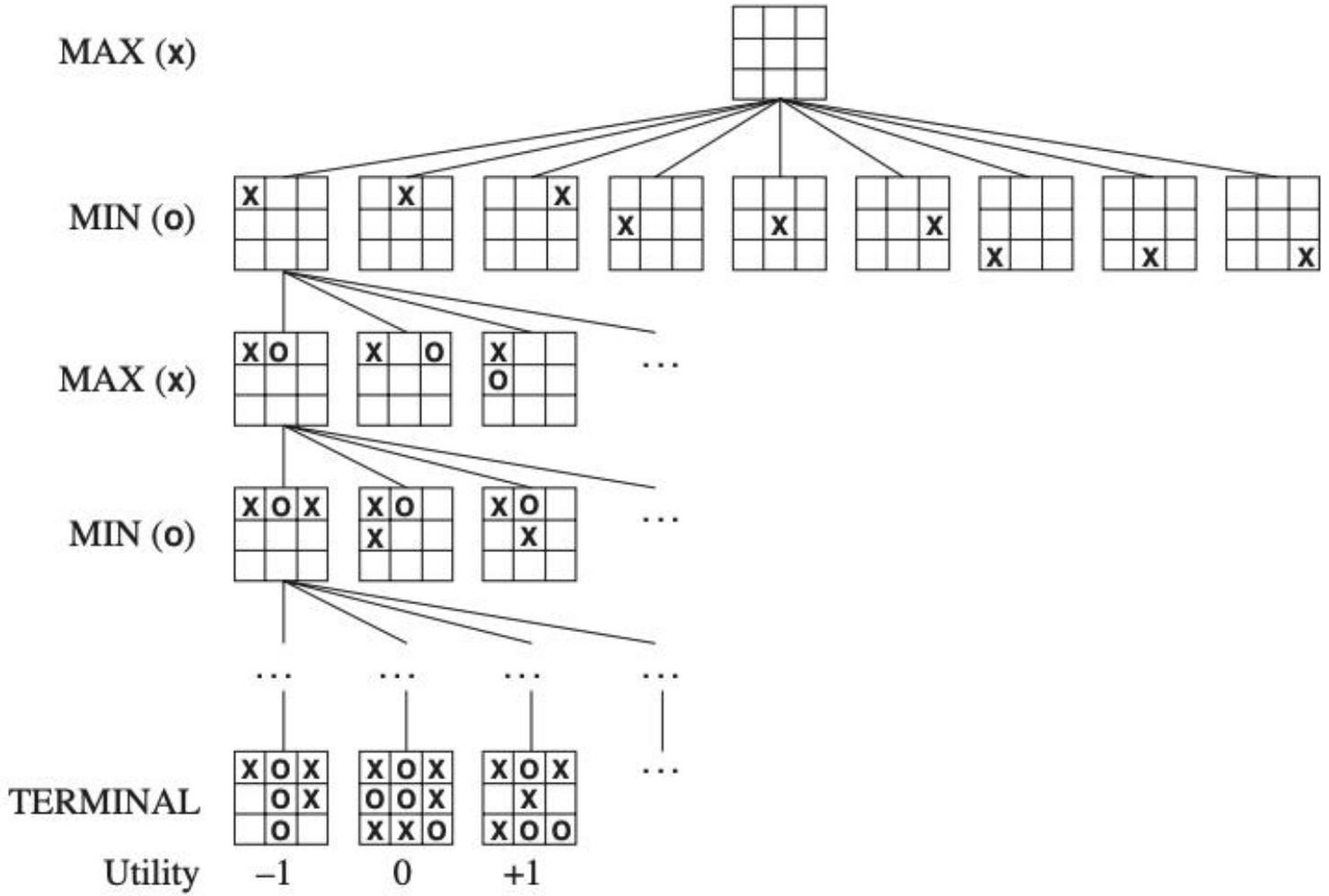
Search Tree for Tic-Tac-Toe



Search Tree for Tic-Tac-Toe



Search Tree for Tic-Tac-Toe



Tic-Tac-Toe

- High values are good for MAX and bad for MIN.
- It is MAX's job to use the search tree and utility values to determine the best move.
- Root is initial position. Next level are all moves player 1 (MAX) can make; tree is from Max's viewpoint. Next level are all possible responses from player 2 (MIN).
- Max has to find a strategy that will lead to a winning terminal state regardless of what Min does. Strategy has to include the correct move for Max for each possible move by Min.

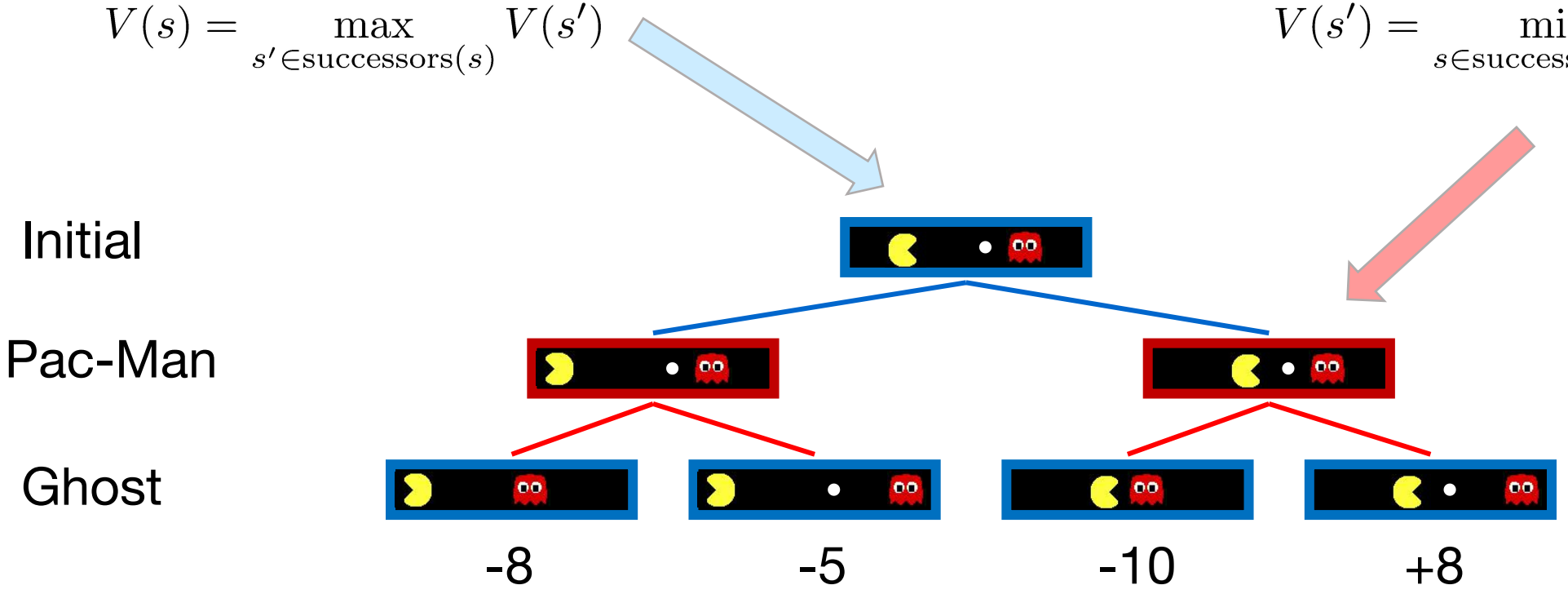
Minimax Values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



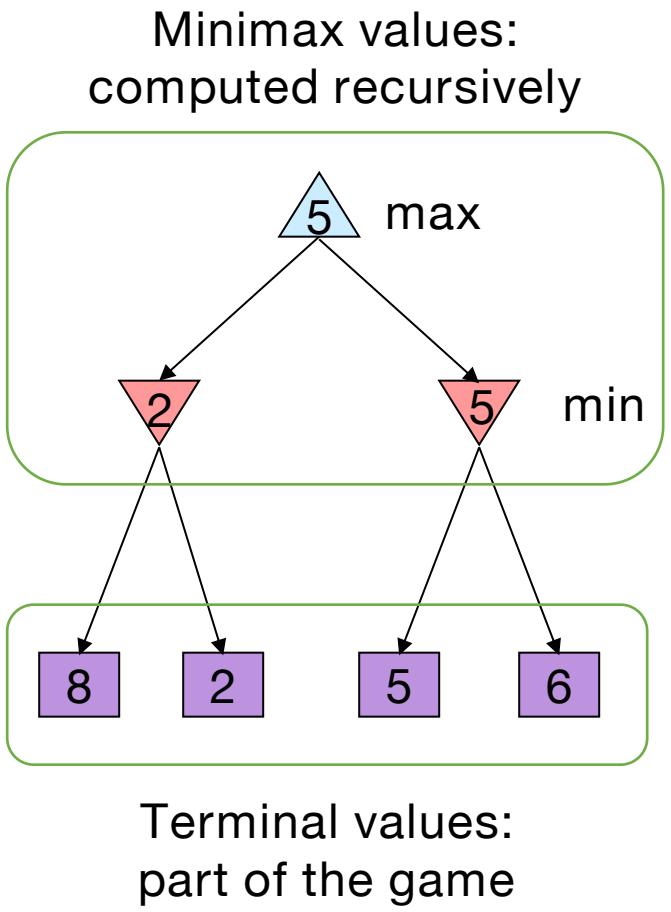
Terminal States:

$$V(s) = \text{known}$$

Minimax Search

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- Compute each node's minimax value:
  - the best achievable utility against a rational (optimal) adversary



Simplifed Minimax Algorithm

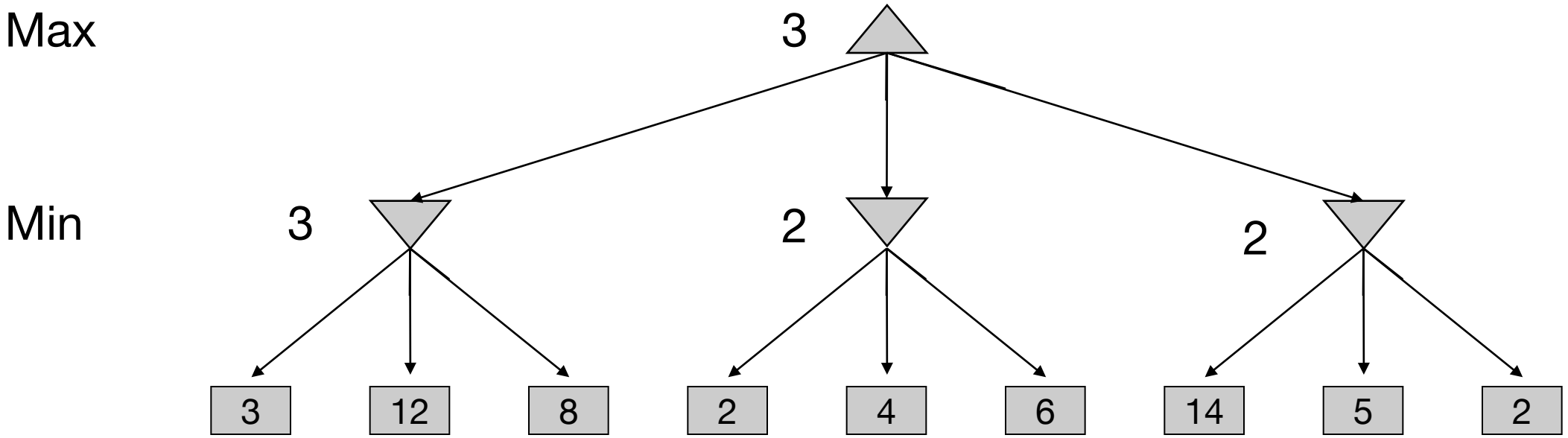
1. Expand the entire tree below the root

2. Evaluate the terminal nodes as wins for the minimizer or maximimizer

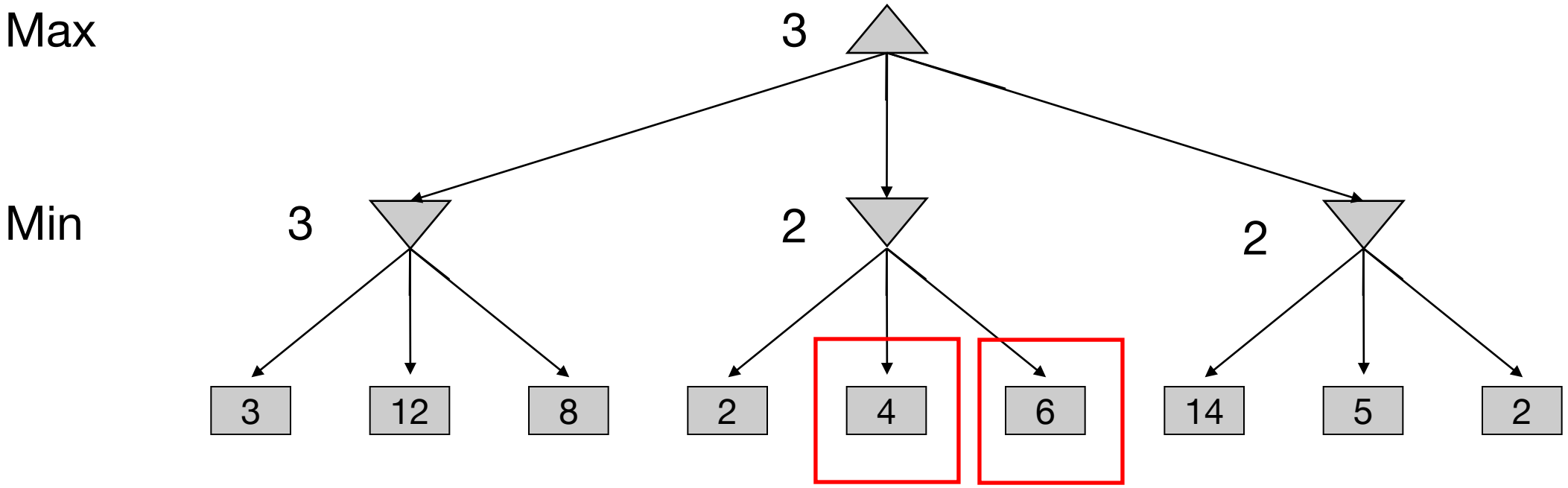
3. Select an unlabeled node, n, all of whose children have been assigned values. If there is no such node, we're done — return the value assigned to the root.

4. If n is a minimizer move, assign it a value that is the minimum of the values of its children. If n is a maximizer move, assign it a value that is the maximum of the values of its children. Return to Step 3.

Another Example



α - β pruning



α - β Pruning

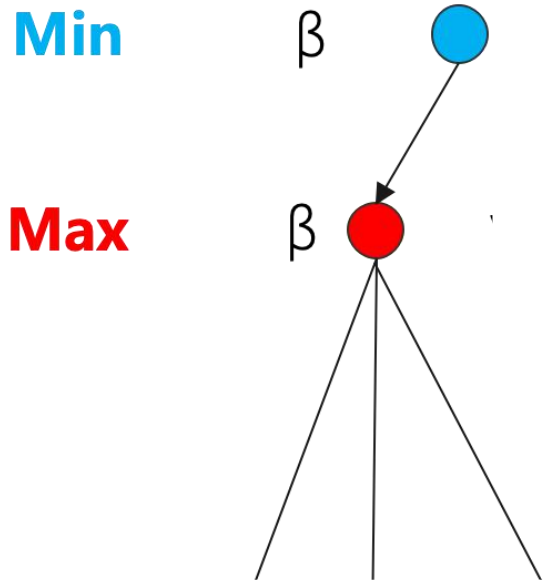
Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above

Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above

α - β Pruning

Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above

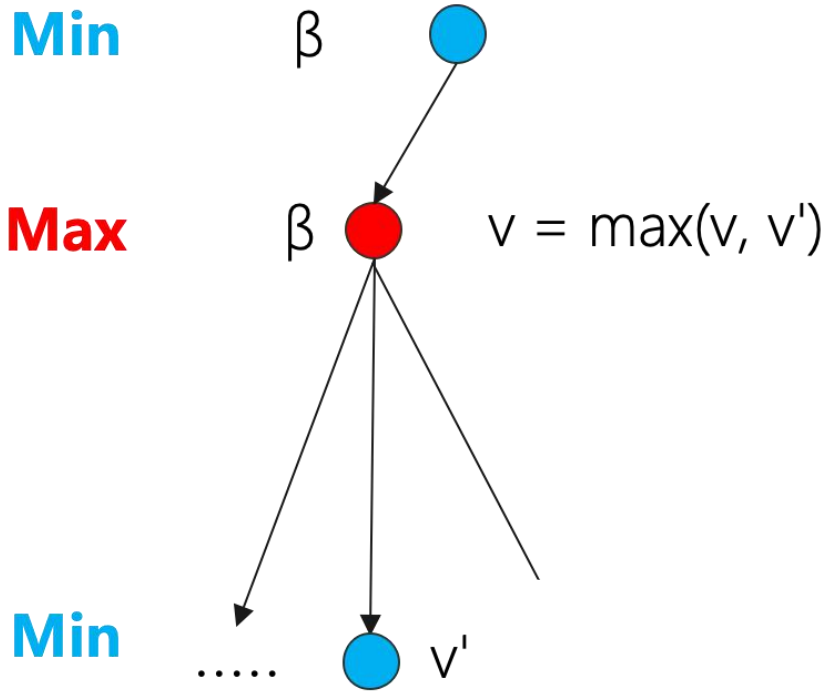
Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above



α - β Pruning

Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above

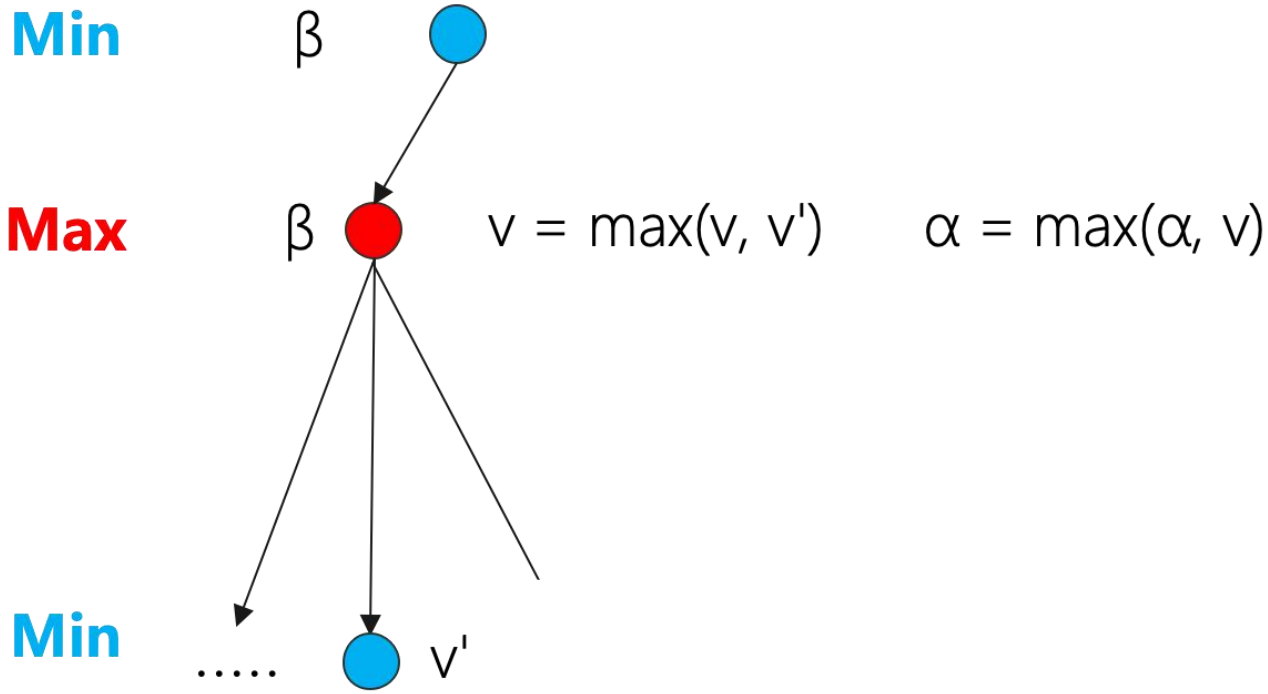
Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above



α - β Pruning

Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above

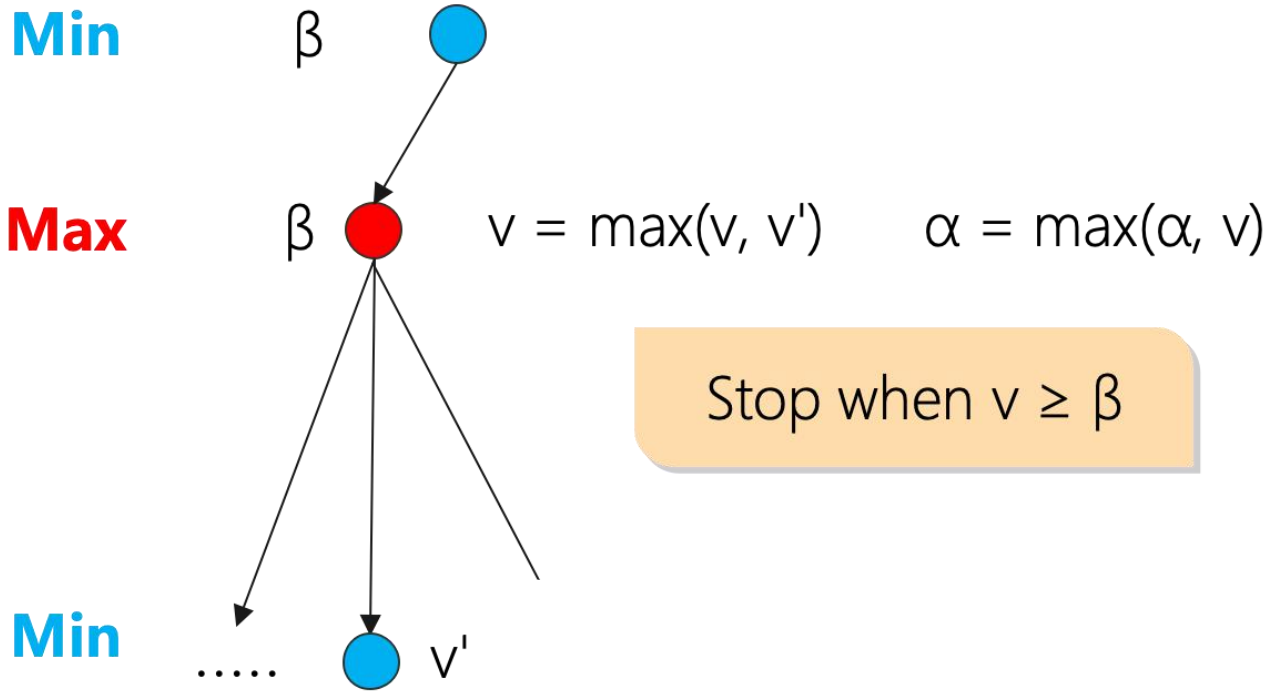
Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above



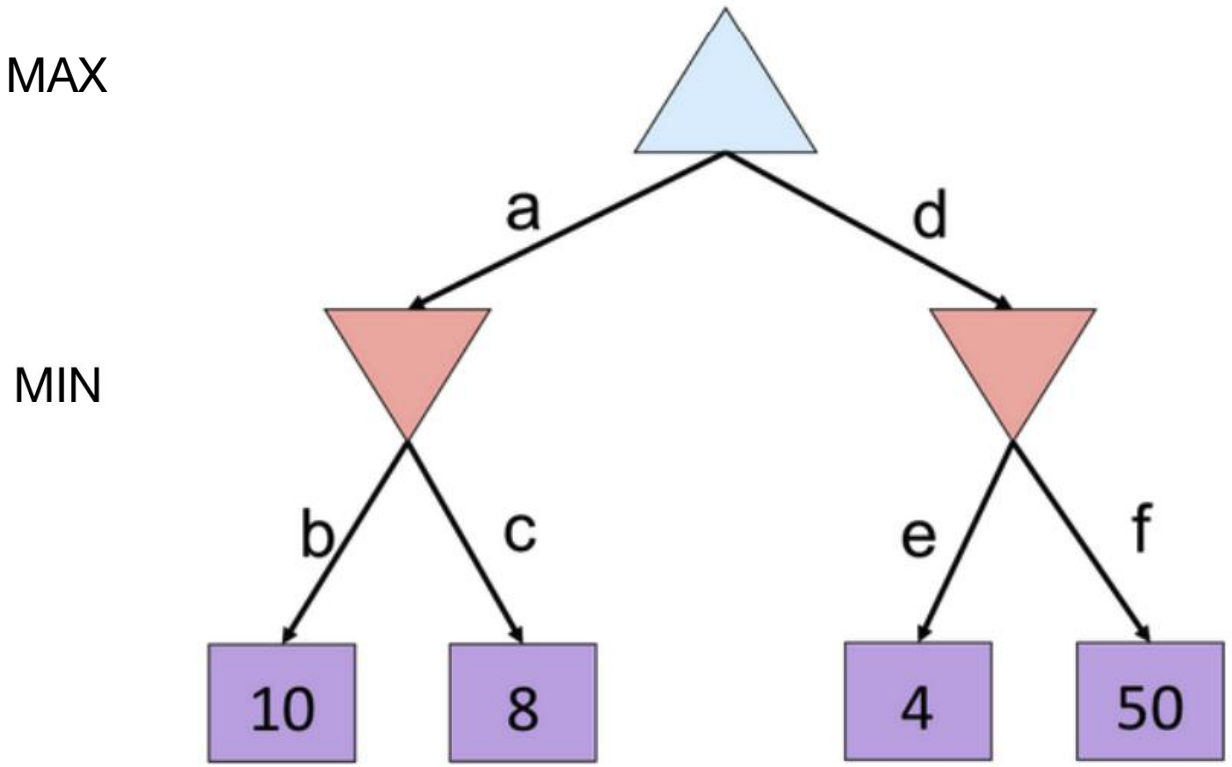
α - β Pruning

Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above

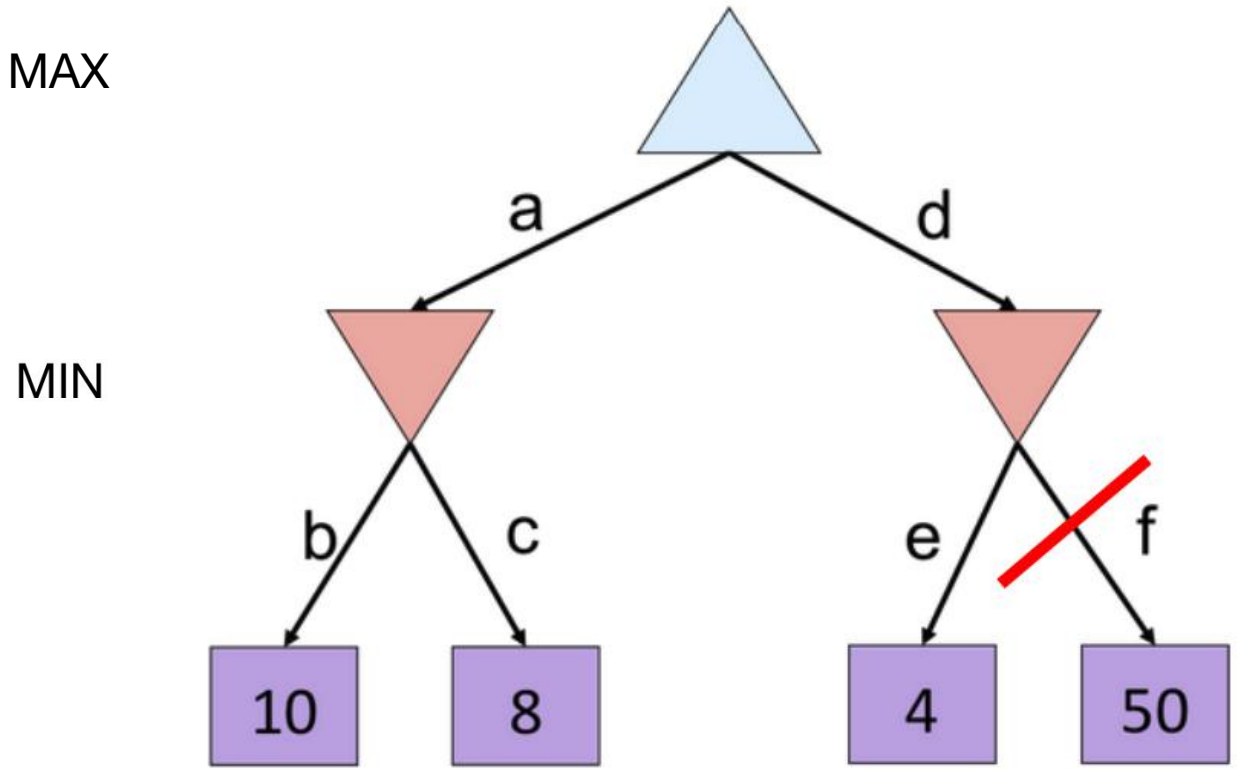
Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above



α - β Search Quiz



α - β Search Quiz

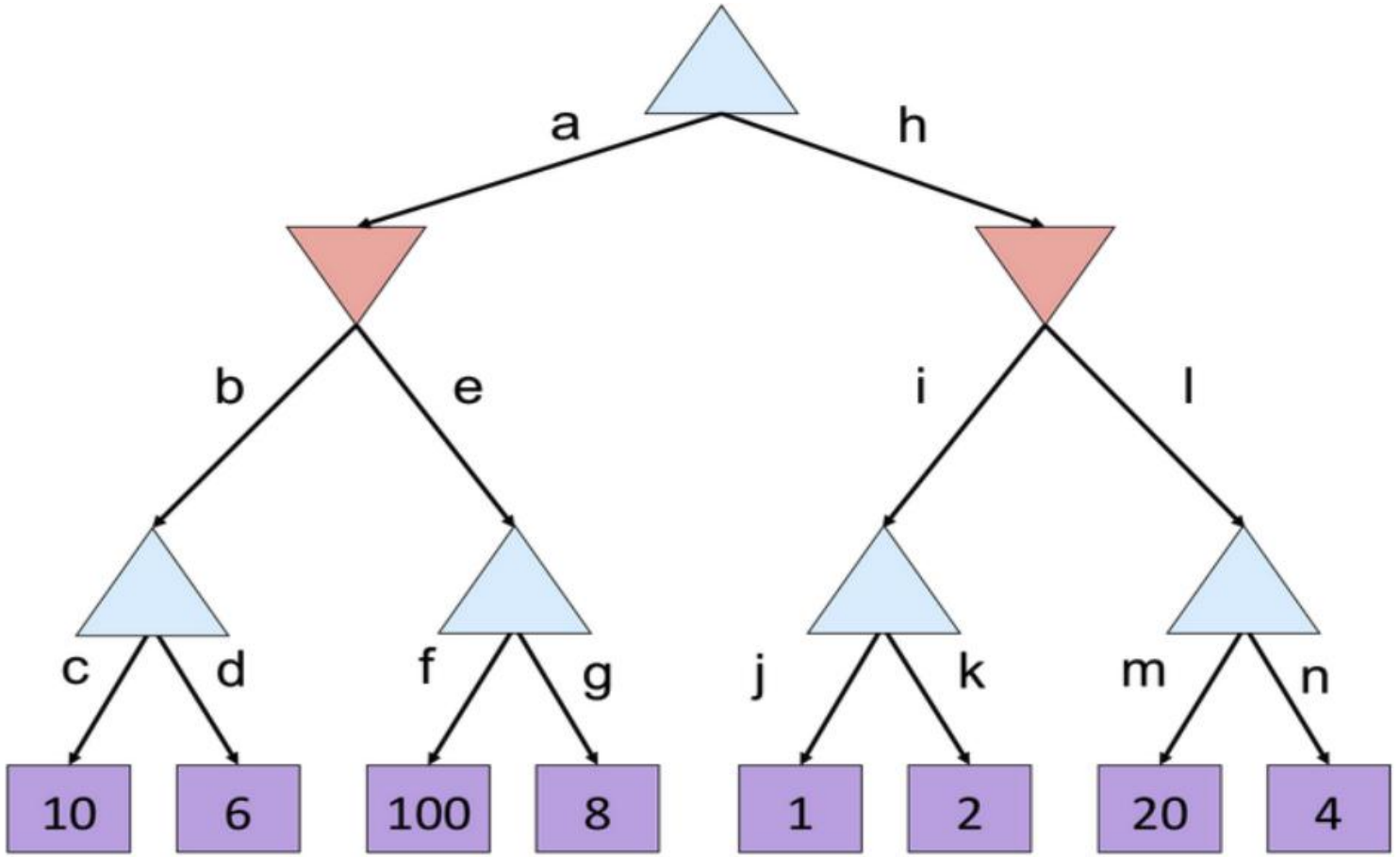


α - β Search Quiz

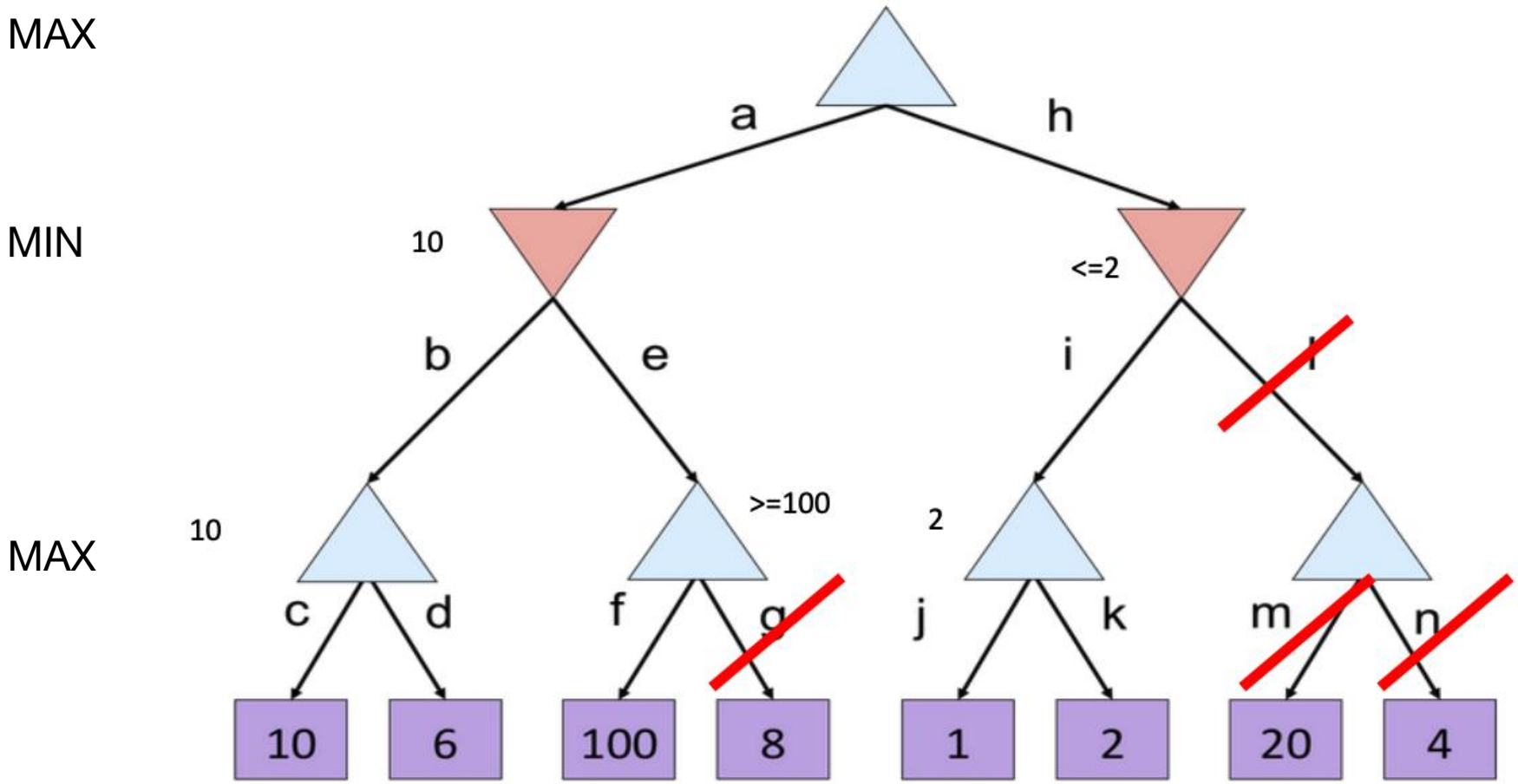
MAX

MIN

MAX



α - β Search Quiz



Logic

- Logic:
  - defines a formal language for logical reasoning
- It gives us a tool that helps us to understand how to construct a valid argument
- Logic Defines:
  - the meaning of statements
  - the rules of logical inference

Logic as a Knowledge Representation

Three components:

- syntax: specifies which sentences can be constructed in a given formal logic
  - E.g. $x + y = 4$
- semantics: specifies what a sentence means
  - $x + y = 4$ is True if $x = 2$ and $y = 2$
- proof theory: a set of general purpose rules that allow efficient derivation of new information from the sentences in the knowledge base

Logic as a Knowledge Representation

Model: a truth assignment to every propositonal symbol

Logic entailment:

A sentence follows logically from another sentence:

$$\alpha \models \beta$$

In every model in which α is true, β is also true.

Logic as a Knowledge Representation

- Logic inference:
  - Given a knowledge base KB and a sentence α
  - Does a KB semantically entail α? KB $\models$ α

One possible approach:

Model Checking: enumerate all the possible models to check if α is true in all models in which KB is true

KR Language: Propositional Logic

- Literal: an atomic formula or its negation
  - Positive literal: P, Q
  - Negative literal: ¬P, ¬Q

- Syntax: build sentences from atomic propositions, using connectives:
  - $\wedge$: and
  - $\vee$: or
  - ¬ : not
  - $\Rightarrow$: implies
  - $\Leftrightarrow$: equivalence (biconditional)

KR Language: Propositional Logic

Syntax: build sentences from atomic propositions, using connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$

(and / or / not / implies / equivalence (biconditional))

E.g.: $\neg P$

$Q \wedge R$

$(\neg P \vee (Q \wedge R)) \Rightarrow S$

KR Language: Propositional Logic

- Clause: a disjunction of literals

  E.g.: $Q \vee R$

- Conjunctive normal form (CNF): a conjunction of clauses

  E.g.: $(Q \vee R) \wedge (P \vee R)$

- Every formula can be equivalently written as a formula in conjunctive normal form

  $(Q \wedge R) \vee P \quad \rightarrow \quad (Q \vee P) \wedge (R \vee P)$

Semantics

Semantics specifies what something means.

In propositional logic, the semantics (i.e., meaning) of a sentence is the set of interpretations (i.e., truth assignments) in which the sentence evaluates to True.

Example:

The semantics of the sentence $P \vee Q \Rightarrow R$ is

- P is True, Q is True, R is True
- P is True , Q is False, R is True
- P is False , Q is True , R is True
- P is False , Q is False , R is True
- P is False , Q is False , R is False

Evaluating a sentence under interpretation I

We can evaluate a sentence using a truth table

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
$false$	$false$	$true$	$false$	$false$	$true$	$true$
$false$	$true$	$true$	$false$	$true$	$true$	$false$
$true$	$false$	$false$	$false$	$true$	$false$	$false$
$true$	$true$	$false$	$true$	$true$	$true$	$true$

Evaluating a sentence under interpretation I

We can evaluate a sentence using a truth table

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Note: $\Rightarrow$ is somewhat counterintuitive

What's the true value of “5 is even implies Sam is smart”

If P is True, then I claim Q is True

Three Important Concepts

- Logic Equivalence
- Validity
- Satisfiability

Logic Equivalence

- Two sentences are equivalent if they are true in the same set of models.

- We write this as $\alpha \equiv \beta$. $\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$

For example:

I. If Lisa is in Denmark, then she is in Europe

II. If Lisa is not in Europe, then she is not in Denmark

Logic Equivalence

- $(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$ commutativity of $\wedge$
- $(\alpha \vee \beta) \equiv (\beta \vee \alpha)$ commutativity of $\vee$
- $((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$ associativity of $\wedge$
- $((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$ associativity of $\vee$
- $\neg(\neg \alpha) = \alpha$ double-negation
- $(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$ contraposition
- $(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta)$ implication elimination
- $(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$ biconditional elimination

Logic Equivalence

- $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$ De Morgan
- $\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$ De Morgan
- $(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$ distributivity of $\wedge$ over $\vee$
- $(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$ distributivity of $\vee$ over $\wedge$

These equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics.

Validity

- Some sentences are very true! For example

1) True 2) $P \Rightarrow P$ 3) $(P \wedge Q) \Rightarrow Q$

A valid sentence is one whose meaning includes every possible interpretation.

$$((P \vee H) \wedge (\neg H)) \Rightarrow P$$

P	H	$P \vee H$	$(P \vee H) \wedge \neg H$	$((P \vee H) \wedge \neg H) \Rightarrow P$
False	*False*	*False*	*False*	*True*
False	*True*	*True*	*False*	*True*
True	*False*	*True*	*True*	*True*
True	*True*	*True*	*False*	*True*

The truth table shows that $((P \vee H) \wedge (\neg H)) \Rightarrow P$ is valid

We write $\models ((P \vee H) \wedge (\neg H)) \Rightarrow P$

Satisfiability

- An unsatisfiable sentence is one whose meaning has no interpretation (e.g., $P \wedge \neg P$)
- A satisfiable sentence is one whose meaning has at least one interpretation.
- A sentence must be either satisfiable or unsatisfiable but it can't be both.
- If a sentence is valid then it's satisfiable.
- If a sentence is satisfiable then it may or may not be valid.

Convert to CNF

- Convert $(B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1} \wedge P_{1,2}$ into CNF
- $((B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})) \wedge \neg B_{1,1} \wedge P_{1,2}$ (biconditional)
- $((\neg B_{1,1} \vee (P_{1,2} \vee P_{2,1})) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) \wedge \neg B_{1,1} \wedge P_{1,2}$ (Implication elimination)
- $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) \wedge \neg B_{1,1} \wedge P_{1,2}$ (De Morgan)
- $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) \wedge \neg B_{1,1} \wedge P_{1,2}$ (Distri.)