CS 161 Intro. To Artificial Intelligence

Week 4, Discussion 1C

Li Cheng Lan

Today’s Topics

- Constraint Satisfaction Problem (CSP)
  - Formulation of CSPs
  - Backtrack Search
  - Techniques for improving CSP solution
  - Tree-structured CSPs
- Game Playing
  - Formulation as Search
  - Minimax Algorithm
  - Alpha-beta Pruning
  - Expect Minimax for Nondeterministic Games
- Propositional Logic

Today's Topics

- **Constraint Satisfaction Problem (CSP)**
  - Formulation of CSPs
  - Backtrack Search
  - Techniques for improving CSP solution
  - Tree-structured CSPs
- Game Playing
  - Formulation as Search
  - Minimax Algorithm
  - Alpha-beta Pruning
  - Expect Minimax for Nondeterministic Games
- Propositional Logic

CSP - Formulation

Components of Constraint Satisfaction Problem (CSP):

X is a set of variables, $\{X_1, \ldots, X_n\}$.

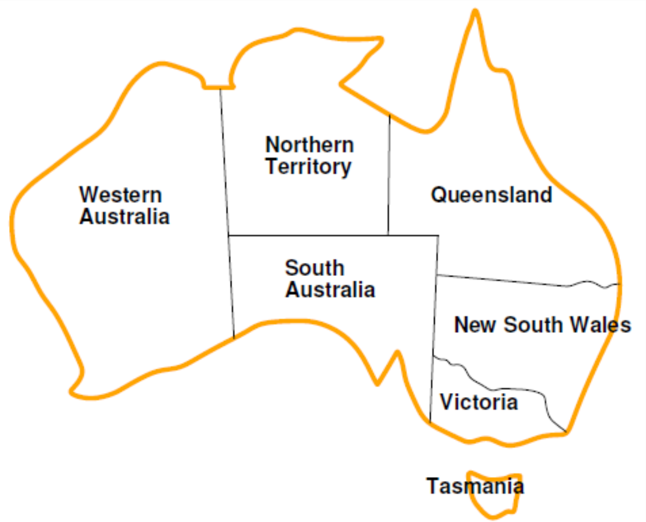
D is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.

C is a set of constraints that specify allowable combinations of values.

- Each domain Di consists of a set of allowable values {v1,...,vk} for the corresponding Xi
- A **state** in CSP: an assignment of values to some or all variables
  - Partial assignment: assign values to only some of the variables
  - <u>Complete</u> assignment: every variable is assigned (otherwise partial assignment)
  - <u>Consistent</u>/Legal assignment: an assignment that does not violate any constraints
- A **solution** in CSP: a consistent, complete assignment

CSP – Formulation Example

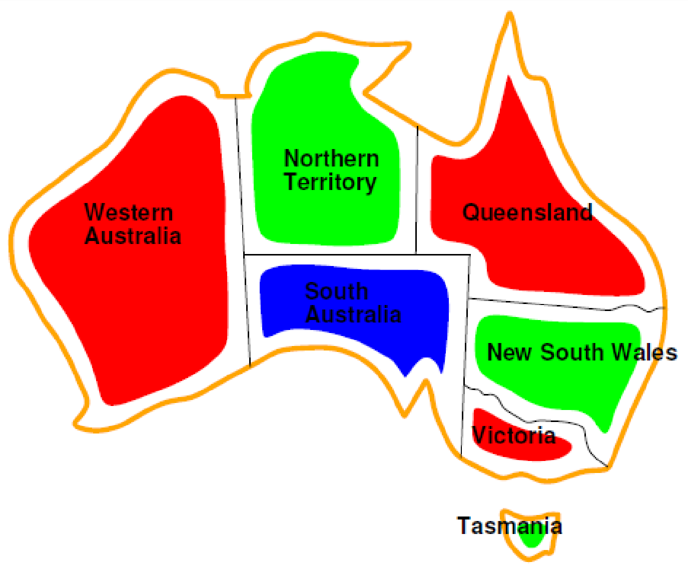
Map Coloring:



- **Variables**: WA, NT, Q, NSW, V , SA, T
- **Domains**: Di = {red, green, blue}
- **Constraints**: adjacent regions must have different colors
  - E.g. WA ≠ NT (if the language allows this),
  - E.g. (WA,NT) ∈ {(red, green), (red, blue), (green, red), (green, blue), . . .}

CSP – Formulation Example

Map Coloring:



- Solutions are assignments satisfying all constraints,
  - e.g., {WA=red, NT =green, Q=red, NSW =green, V =red, SA=blue, T =green}

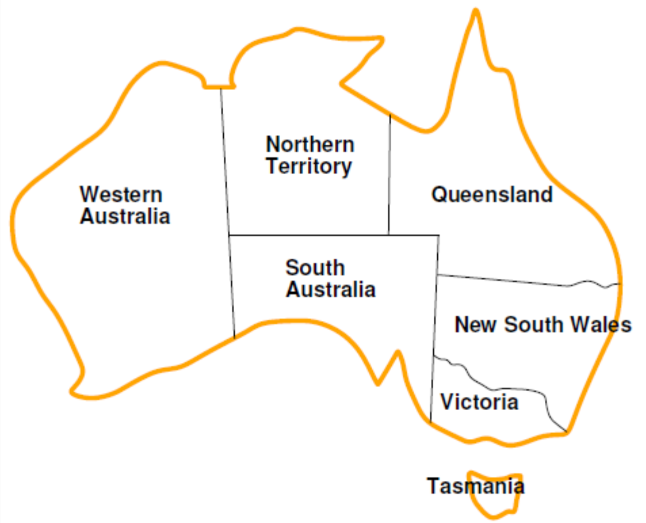
Varieties in CSP

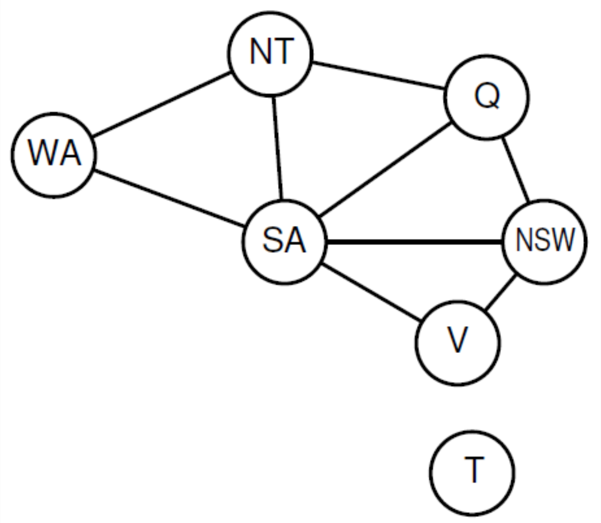
- Unary constraints: involve a single variable,
  - e.g., SA $\neq$ green
- **Binary** constraints: involve pairs of variables, ← look at in this class
  - e.g., SA $\neq$ WA
- Higher-order(Global) constraints: involve 3 or more variables,
  - e.g., Alldif (all of the variables involved in the constraint must have different values)
- Preferences (soft constraints):
  - e.g., red is better than green
  - often representable by a cost for each variable assignment

    → constrained optimization problems

Constraint Graph

- For **Binary CSP** -- each constraint relates at most two variables
- Constraint graph: nodes are variables, arcs show constraints
  - General-purpose CSP algorithms use the graph structure to speed up search.





Backtrack Search

Backtracking search: the basic uninformed algorithm for CSPs

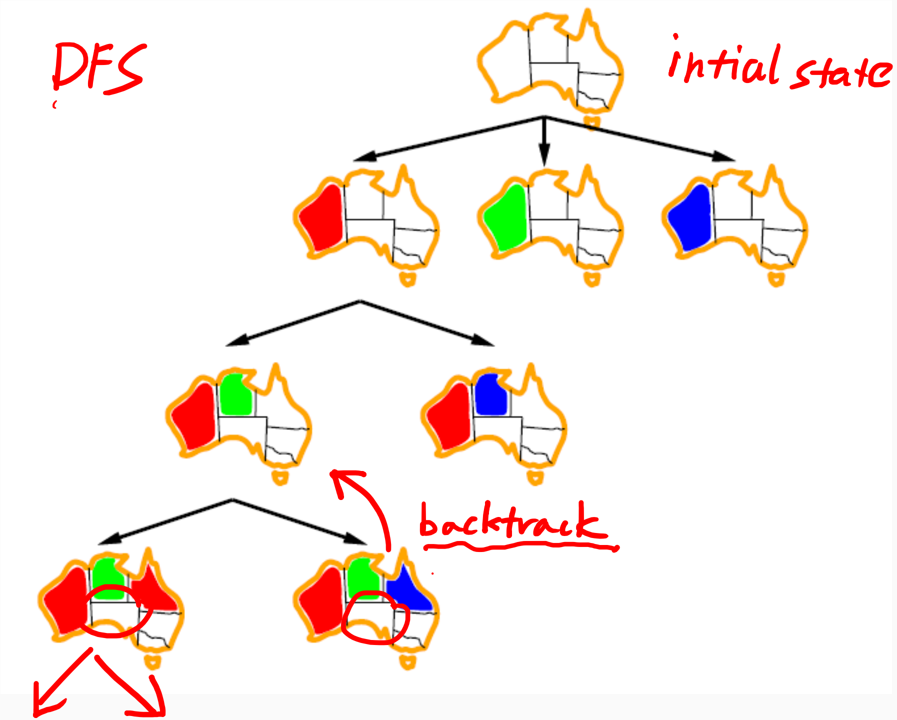
- It's basically a depth-first search for CSPs with single-variable assignments:
  - Chooses values for one variable at a time and **backtracks when a variable has no legal values left to assign**

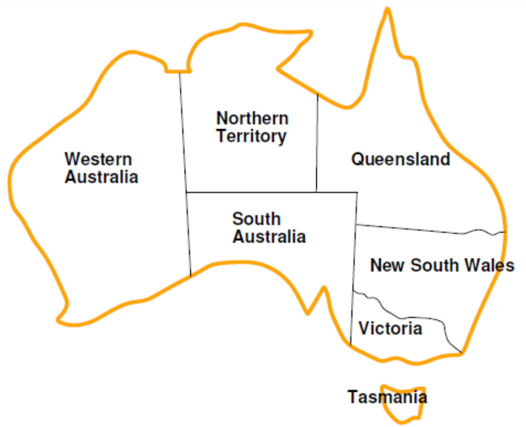
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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
```

Backtrack Search - Example

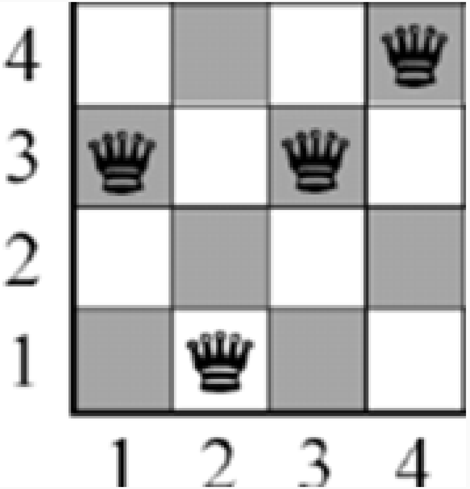
- When to backtrack? → when a variable has no legal values left to assign





Backtrack Search - Example

- 4-Queens Puzzle (assume each queen in each column) as a CSP:
  - Variables: Q1, Q2, Q3, Q4 -- row indices of each queen
  - Domains Di = {1, 2, 3, 4}
  - Constraints:
    - $Qi \neq Qj$ (cannot be in same row)
    - $|Qi - Qj| \neq |i - j|$ (or same diagonal)



- Backtracking search:
  - (Q1, Q2, Q3, Q4):
  - (1,X,X,X) → (1, 3,X,X) → No legal assign for Q3, backtracking
  - (1, 4,X,X) → (1, 4, 2,X) → No legal assign for Q4, backtracking
  - (1, 4, 3, X) → Not a legal assign for Q3, backtracking
  - (2,X,X,X) → (2, 4,X,X) → (2, 4, 1,X) → (2, 4, 1, 3), Bingo! ← solution

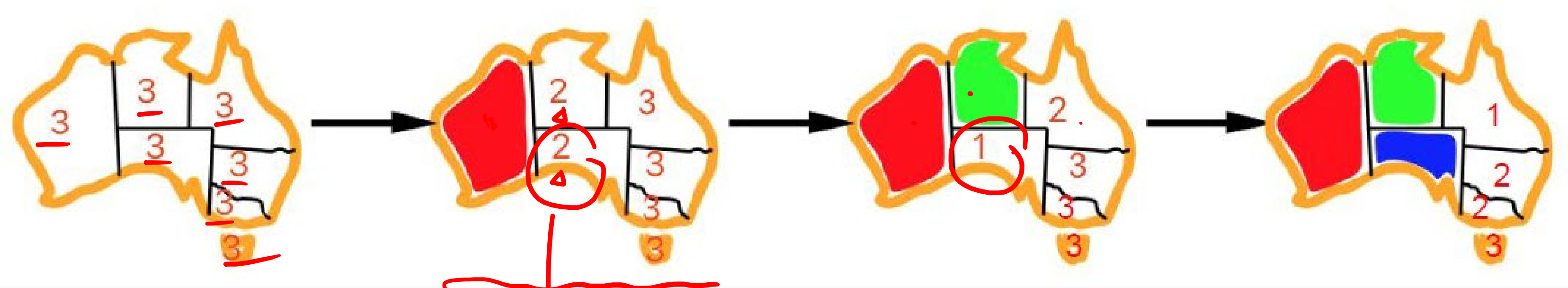
Improving Backtracking Efficiency

3 techniques (heuristics) to improve efficiency:

- How to select unassigned **variable**?
  - Most constrained variable / Minimum remaining values (MRV)
  - Most constraining variable / Degree heuristic
- In what order should we assign **values** to each variable?
  - Least constraining value

Improving Backtracking Efficiency

- How to select unassigned **variable**?
  - Most constrained variable / Minimum remaining values (MRV):
    - choose the **variable** with the fewest legal values
    - If no legal values left, fail immediately



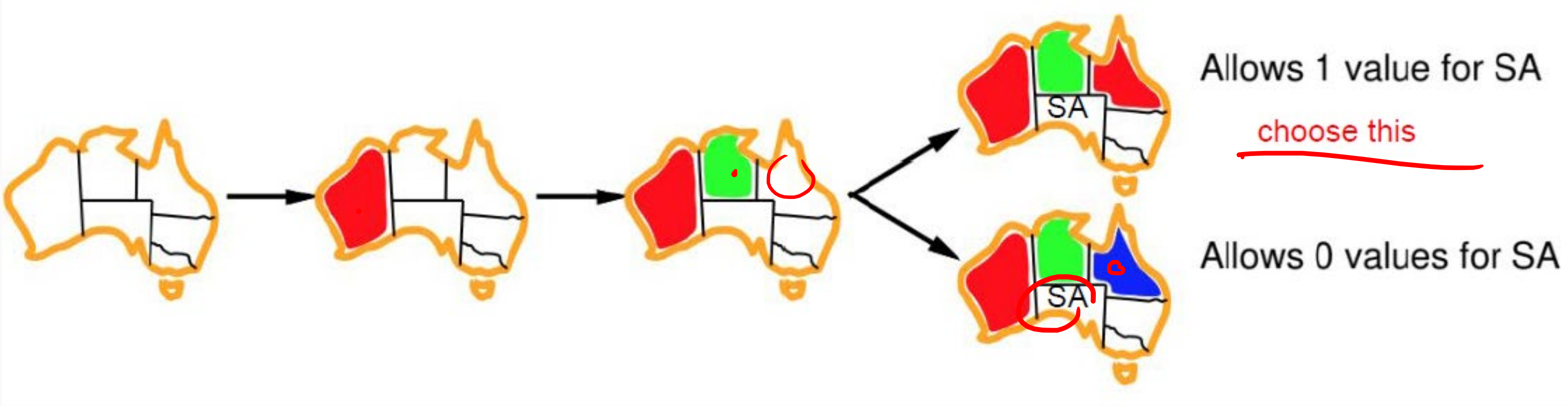
choose this
based on degree heuristic

Improving Backtracking Efficiency

- How to select unassigned **variable**?
  - Most constrained variable / Minimum remaining values (MRV)
  - Most constraining variable / Degree heuristic:
    - Choose the **variable** with the most constraints on remaining variables
    - Attempt to reduce branching factor on future choice
    - **Useful as a tie-breaker**

Improving Backtracking Efficiency

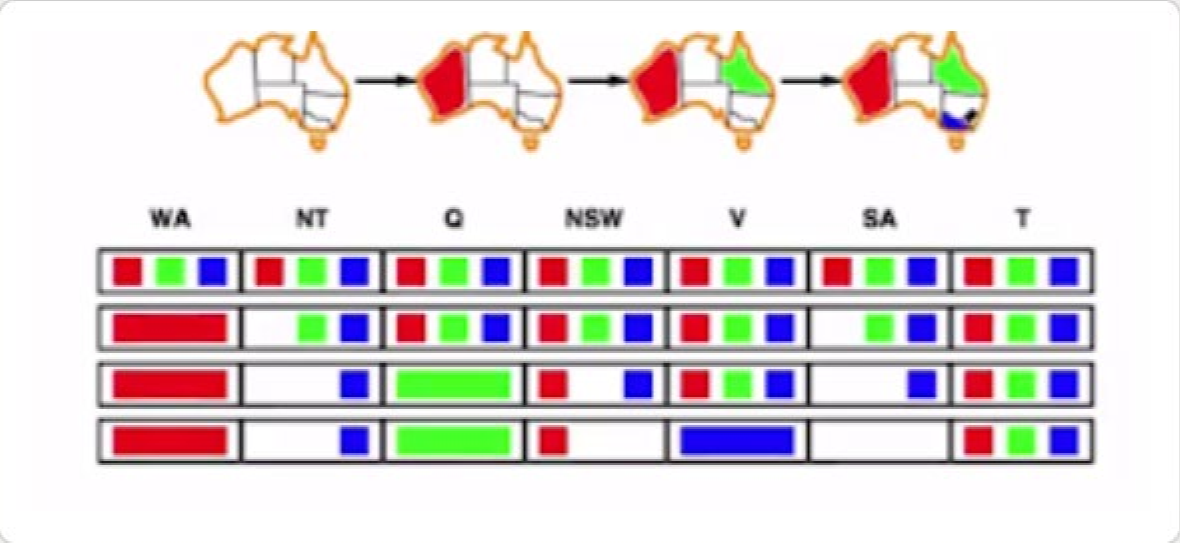
- In what order should we assign **values** to each variable?
  - Least constraining value:
    - Choose the **value** that rules out the fewest values in the remaining variables
    - Leave the maximum flexibility for subsequent variable assignments



Early Failure Detection

Two methods to detect failures by doing domain reductions:

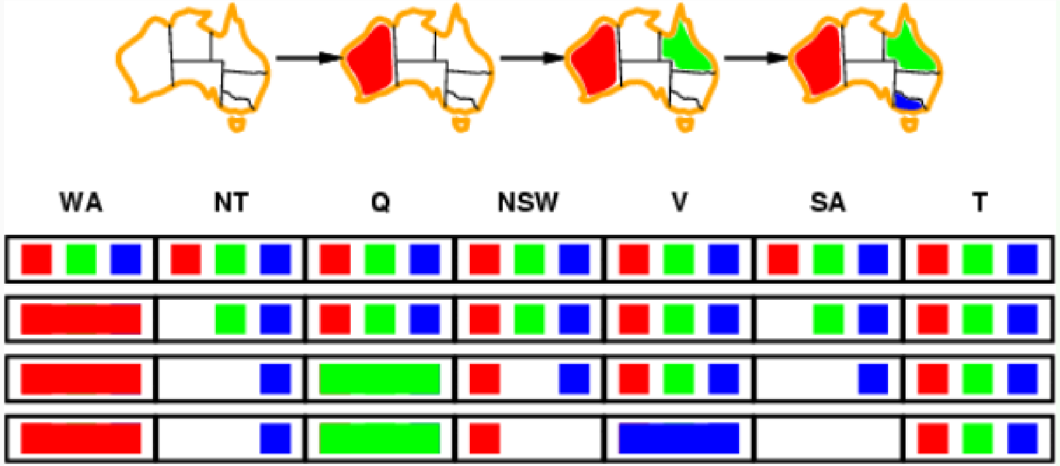
- **Forward checking**
- Arc consistency



Early Failure Detection

Two methods to detect failures by doing domain reductions:

- Forward checking
- **Arc consistency**



Arc Consistency (AC)

because unidirectional

n variables: $2(n-1)$ arcs

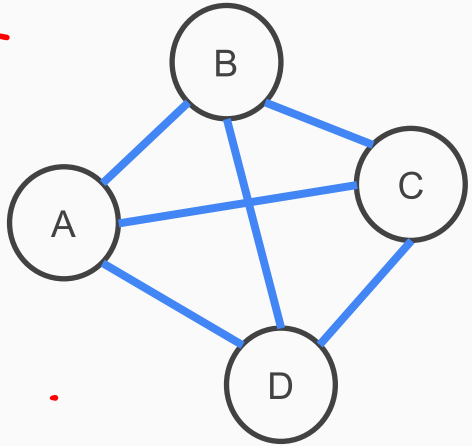
$1 \quad 2 \quad \cdots \quad n$

$2\times \quad 0 \quad 1 \quad n-1$

$(n-1)n$

Evaluation of AC:

- Notations:
  - n: # of variables
  - d: largest domain size
  - c: # of constraints
- Time complexity: $\mathbf{O(n^2 d^3)}$
  - Checking consistency of one arc: $O(d^2)$
    - Look at different combinations of values
  - At most $(n^2 - n)$ arcs: $O(n^2)$ or O(c) ← 2*c constraints
  - Each arc (X_i, X_j) can be inserted at most d times
    - X_i has at most d values to delete



Arc Consistency (AC) – AC3 algorithm

Before Backtracking Search.

AC3 (*not covered in class*) -- reduce domain size based on arc consistency before search start:

- Maintains a queue (set) of all arcs
- Pop an arbitrary arc ($X_i \leftarrow X_j$) and check D_i (the domain of X_i)
  - D_i unchanged
    - Move to next
  - D_i becomes smaller
    - Add to queue all arcs ($X_k \leftarrow X_i$) where X_k is a neighbor of X_i
    - The change in D_i might enable further reductions in the domains of D_k , even if we have previously considered X_k and D_k
  - D_i is empty
    - Fail!

Time complexity: $O(n^2 d^3)$

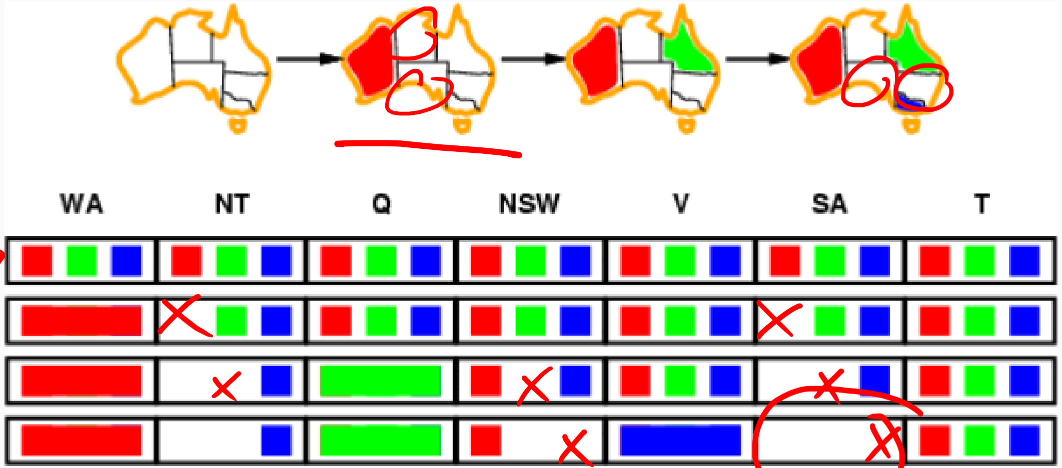
Forward Checking (FC)

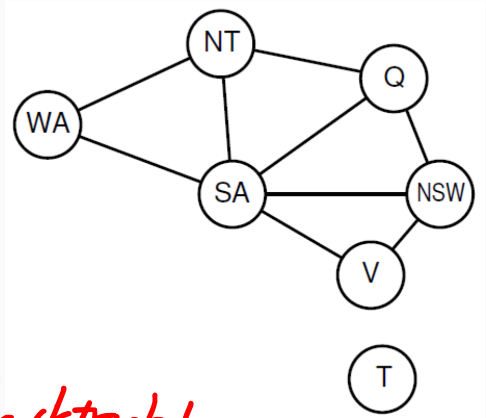
Inside backtracking search:

Forward Checking (during search):

- Keep track of remaining legal values for unassigned variables that are connected to current variable.

  → **Variable-level arc consistency**
- Terminates when any variable has no legal values
  - Then backtrack!





Forward Checking (FC)

- Doing FC every time we assign a value to a variable:
- If a variable has no legal value, do backtrack

BT: DFS-based search algorithm

FC: technique to rule out some invalid values
↳ increase efficiency of BT

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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
```

Forward checking after this

Comparison of AC and FC

Techniques

AC3: → check all the arcs

- **Before** search
- **initialization**: push **all** arcs in the queue → AC3
- **maintain the queue**: when some variable X_i's domain size change, push (X_k <- X_i) into the queue, where X_k is neighbor of X_i. ➔ Do both POP and PUSH!

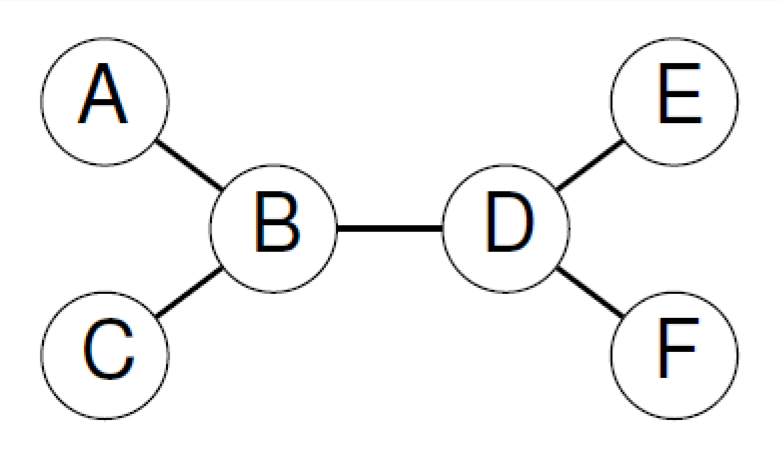
FC: → check arcs related to current variable.

- **During** search
- **initialization**: when assign value to X, push all X's neighbor into the queue. (neighbor <- X)
- **maintain the queue**: queue won't add anything after initialized. ➔ ONLY POP, NO PUSH!

Tree-structured CSPs

Theorem: If the constraint graph has no loops (tree-structured), the CSP can be solved in $\mathbf{O}(nd^2)$ time

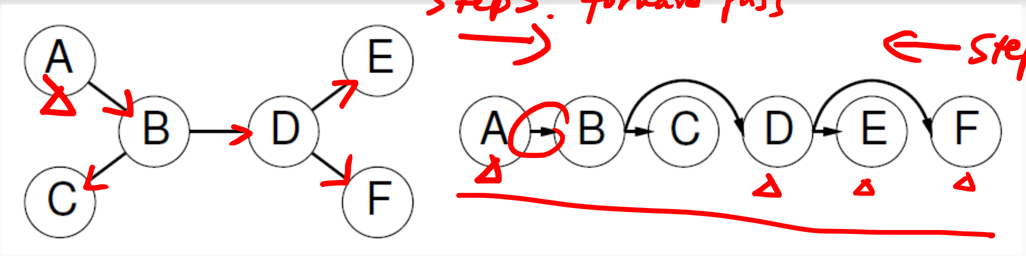
- n: # of variables/nodes
- d: largest domain size



Tree-structured CSPs

step 3. forward pass

step 2. backward pass



Algorithm with time complexity $\mathbf{O}(nd^2)$: n nodes → (n-1) edges → O(n) for edges

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering

$O(d^2)$ for each edge

2. For j from n down to 2, apply Remove-Inconsistent-Values(Parent(X_j), X_j) as follows:

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

→ check arc consistency.

3. For j from 1 to n, assign X_j consistently with Parent(X_j)

Today's Topics

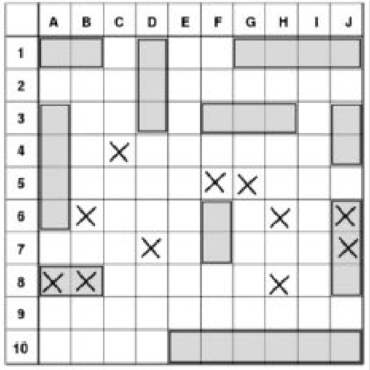
- Constraint Satisfaction Problem (CSP)
  - Formulation of CSPs
  - Backtrack Search
  - Techniques for improving CSP solution
  - Tree-structured CSPs
- **Game Playing**
  - Formulation as Search
  - Minimax Algorithm
  - Alpha-beta Pruning
  - Expected Minimax for Nondeterministic Games
- Propositional Logic

Types of Games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

Go: Perfect and Deterministic

Monopoly: Perfect, Chance Introduced



Battleship: Imperfect and Deterministic

Bridge: Imperfect, Chance Introduced

Game as a Search Problem

Can we use search strategies to win games?

- Require to make some decision when calculating the optimal decision is infeasible
- The **solution** will be a **strategy** that specifies a move for every possible opponent reply
- Challenges:
  - Very, very large search space
  - Time limits

Game as a Search Problem

- **S0**: The initial state, which specifies how the game is set up at the start.
- **PLAYER(s)**: Defines which player has the move in a state.
- **ACTIONS(s)**: Returns the set of legal moves in a state.
- **RESULT(s, a)**: The transition model, which defines the result of a move.
- **TERMINAL-TEST(s)**: A terminal test, which is true when the game is over and false otherwise.
  - States where the game has ended are called terminal states. utility values
- **UTILITY(s, p)**: A utility function that defines the final numeric value for a game that ends in terminal state s for a player p.
  - Also called an objective function or payoff function.
  - In chess, the outcome is a win, loss, or draw, with values $+1$, 0, or $1/2$.

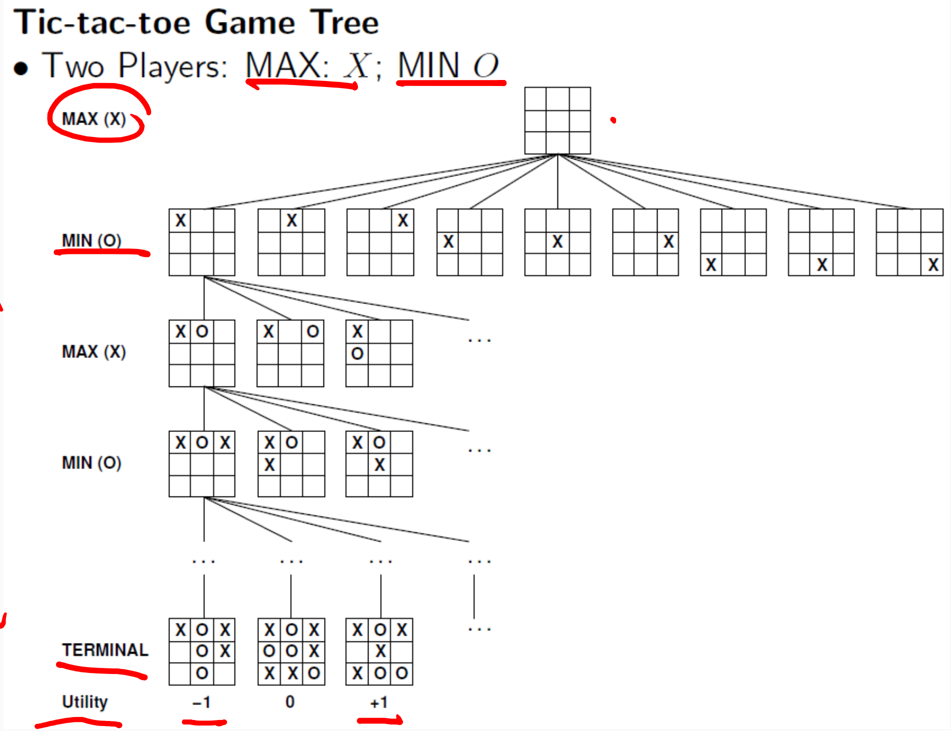
Optimal Decisions in Games

How to find the <u>optimal decision</u> in a <u>deterministic</u>, <u>perfect-information</u> game?

Idea: choose the move with highest achievable payoff against the best play of the other player

Partial Game Tree:

- Top node is the initial state
- Giving alternating moves by MAX and MIN



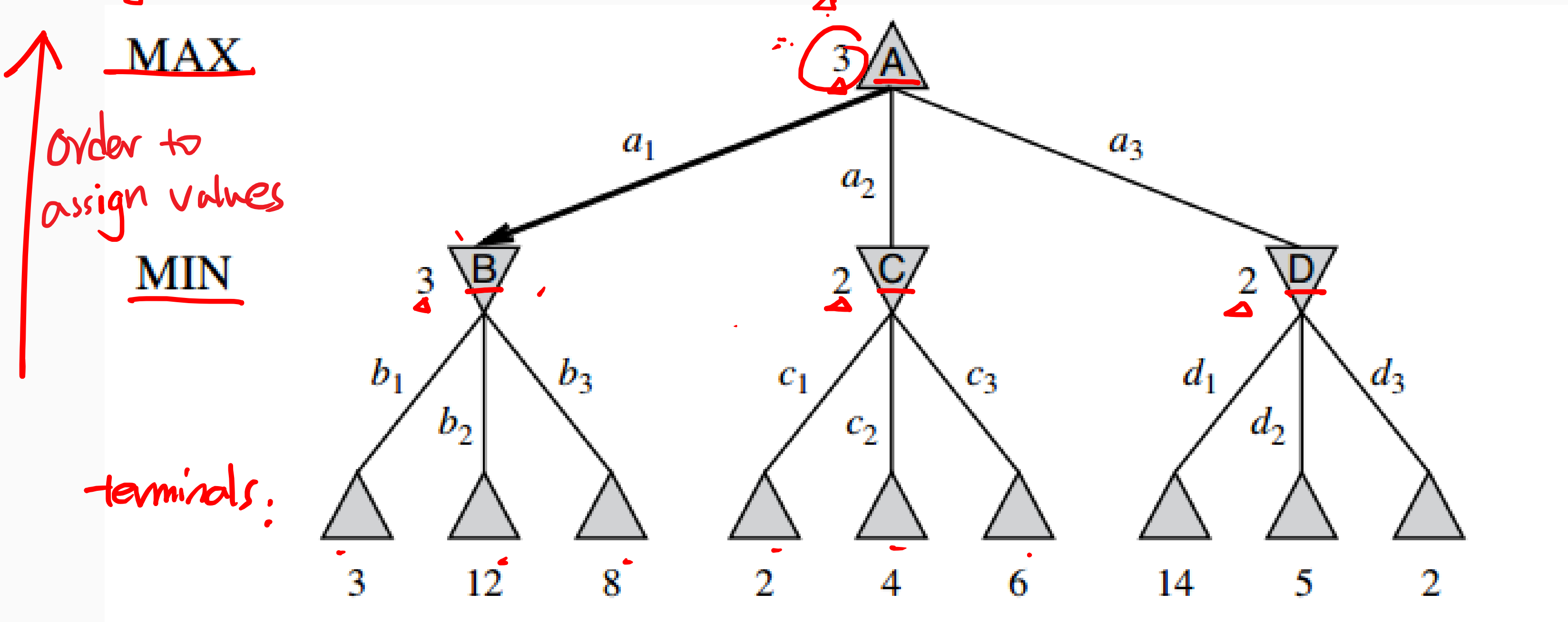
Minimax Algorithm

- Imagine we are MAX
- We refer to the payoff as MINIMAX value, at each step
  - MAX wants MINIMAX value to be as <u>big</u> as possible
  - MIN wants MINIMAX value to be as <u>small</u> as possible

$$
\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}
$$

Minimax Algorithm

: takes max value among child nodes   : takes min value among child nodes



Minimax Algorithm

Evaluation:

- Complete (if tree is finite)
- Optimal (since we are against an optimal opponent)
- Time complexity: **O**$(\boldsymbol{b^m})$
  - b: max # of children nodes for one parent node
  - m: max depth of the state space
- Space complexity: **O**$(\boldsymbol{bm})$ ➔ depth-first exploration

Depth-first

Actually don't need to explore every path and compute MINIMAX for every node!

- Increase the efficiency
- Use Alpha-beta pruning

Alpha-beta Pruning

Minimax: a way of finding an optimal move in a two player game.

Alpha-beta pruning: finding the optimal minimax solution while avoiding searching subtrees of moves which won't be selected.

- α : maximum lower bound of possible solutions
- β : minimum upper bound of possible solutions
- If N is estimated value of the node, then $\alpha \leq N \leq \beta$

Alpha-beta Pruning

α : maximum lower bound of possible solutions

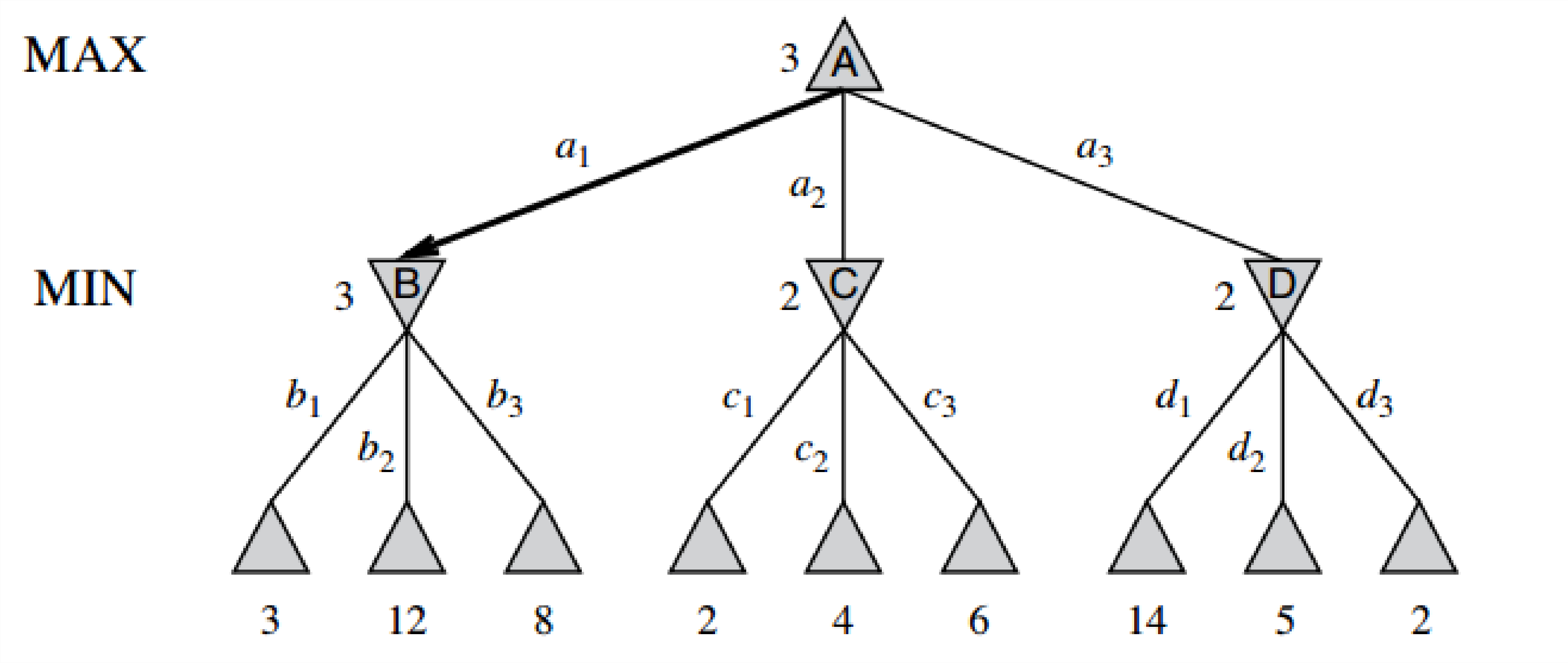
β : minimum upper bound of possible solutions

If N is estimated value of the node, then $\alpha \leq N \leq \beta$

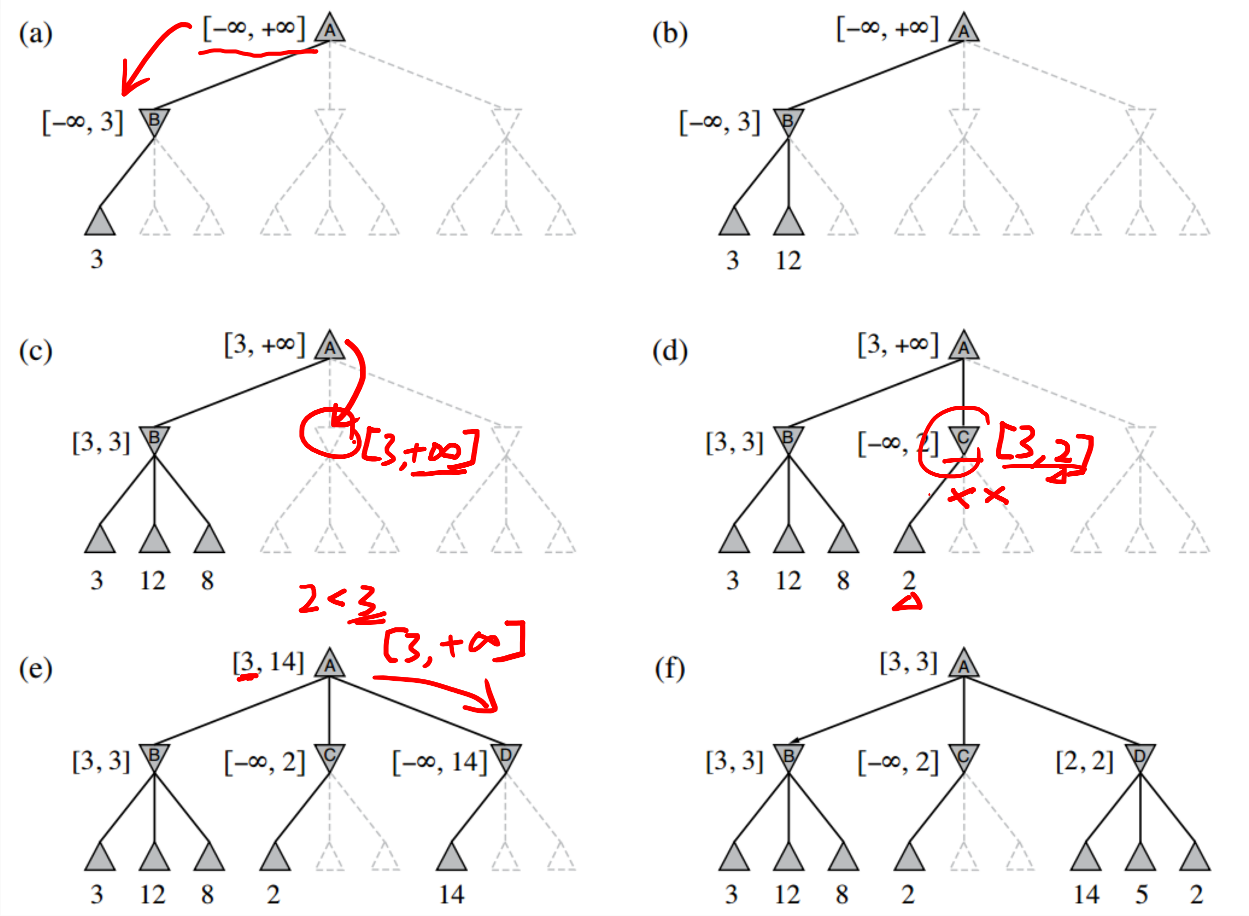
Steps:

- During the search, each node carries an upper bound α a lower bound β
- Pushing bound upward:
  - When a child returns, it pushes its value onto the parent (**always tighten the bound**)
  - Max player will modify its lower bound, and min player will modify its upper bound
- Pushing bound downward (**both lower and upper**) and prune:
  - If min parent, max children, when $\alpha_{children}$ >= β_{parent} , prune!
  - If max parent, min children, when α_{parent} >= $\beta_{children}$, prune!

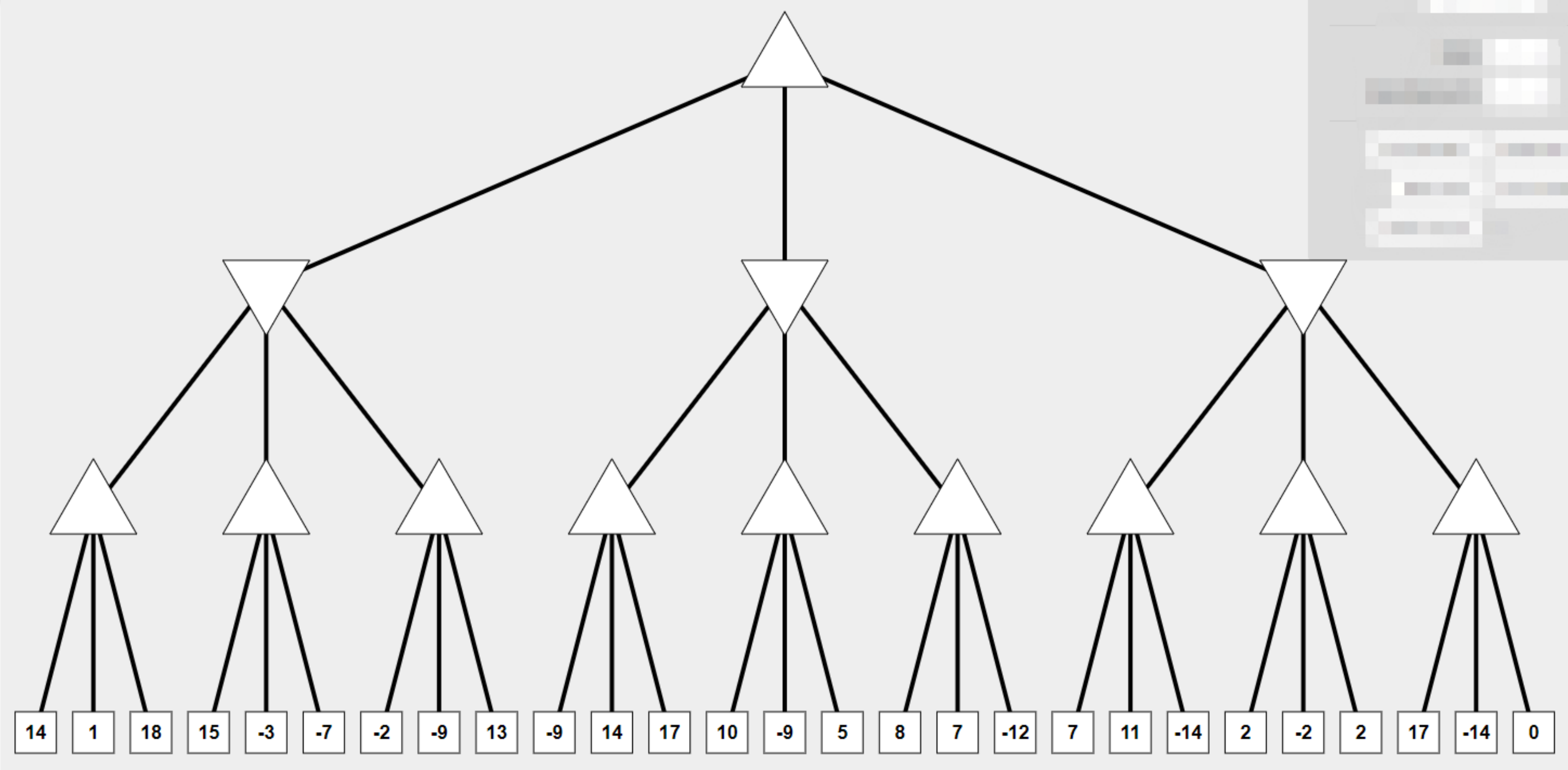
Alpha-beta Pruning - Example



Alpha-beta Pruning - Example



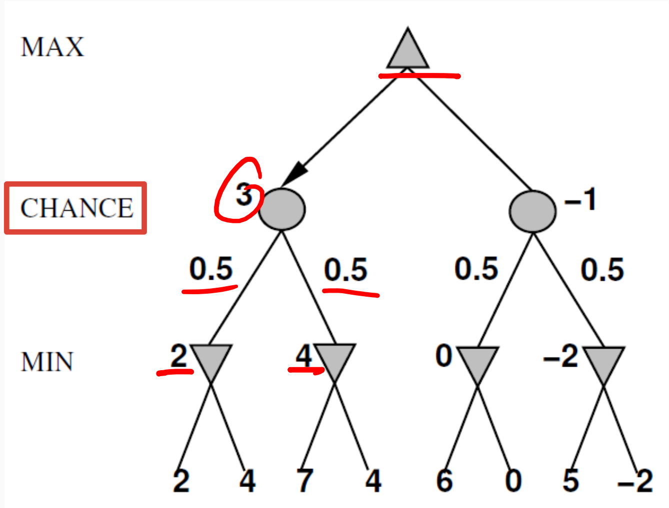
Alpha-beta Pruning - Practices



- More exercises: http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/

Nondeterministic Game

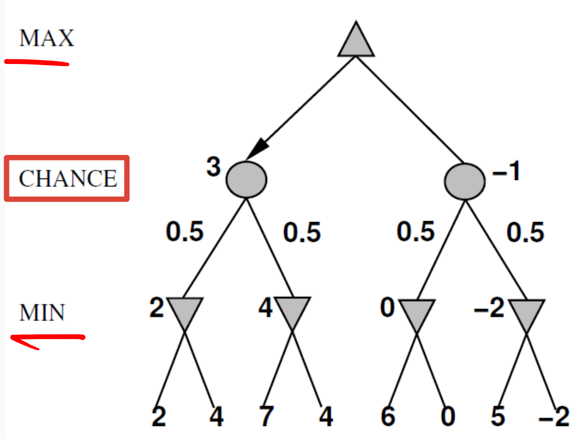
- In deterministic games with perfect information, Minimax Algorithm gives perfect play
- What if the game is nondeterministic with perfect information?
  - In nondeterministic games, chances are introduced
  - For example:
    - Two people MAX and Min play a game
    - Flip a coin after each player make a decision
    - The result of coin flipping changes the state



Expected Minimax Algorithm

- In nondeterministic games, EXPECTMINIMAX gives perfect play
    - Just like MINIMAX, except we must also handle chance nodes

if *state* is a MAX node **then**
  return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
if *state* is a MIN node **then**
  return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
if *state* is a chance node **then**
  return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)

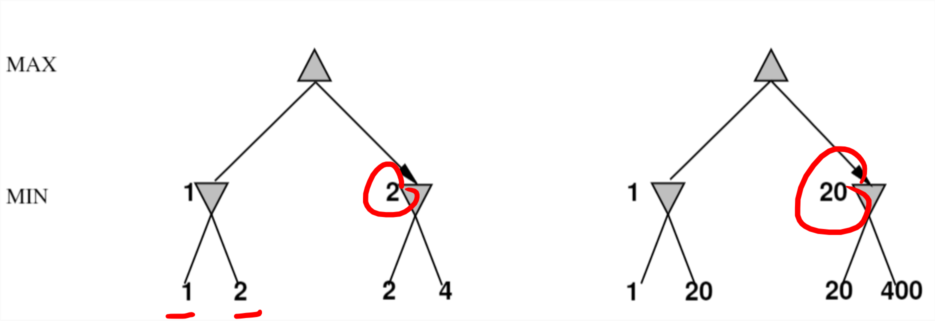


$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}(\text{RESULT}(s,r)) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$

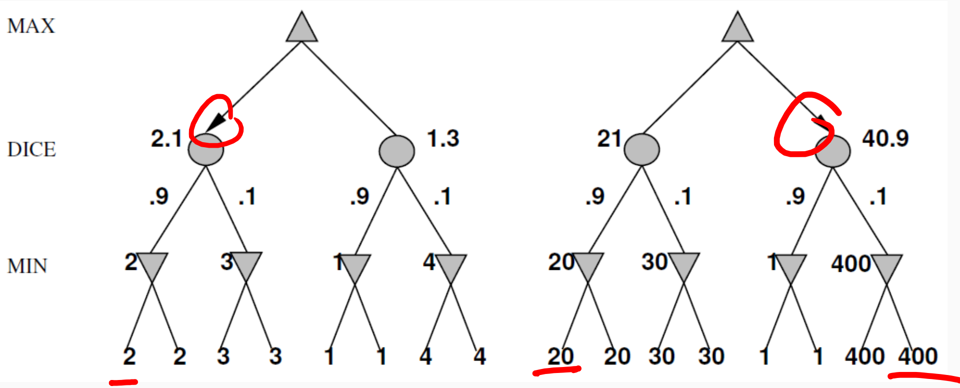
Expected Minimax Algorithm

- Unlike MINIMAX algorithm where only **order** of terminal nodes matters, in EXPECTMINIMAX algorithm, **exact values** of terminal nodes also matter!

MINIMAX



EXPECTMINIMAX



Today’s Topics

- Constraint Satisfaction Problem (CSP)
  - Formulation of CSPs
  - Backtrack Search
  - Techniques for improving CSP solution
  - Tree-structured CSPs
- Game Playing
  - Formulation as Search
  - Minimax Algorithm
  - Alpha-beta Pruning
  - Expected Minimax for Nondeterministic Games
- **Propositional Logic**

Logic

- Logic: knowledge representation language
  - Represent human knowledge as "**sentences**" (a.k.a *axiom*)
    - **Knowledge base (KB)**: a set of sentences
- Examples
  - **Propositional logic**
    - Boolean logic
  - **First-order logic**
    - Quantifiers ∀, ∃, objects and relations
- Key components in Logic
  - Syntax: how to write sentences
  - Semantics: how to interpret sentences
  - Reasoning/Inference: What new knowledge can be derived from known facts

B or ¬B

Propositional Logic - Syntax

Syntax:

- Atomic sentence
  - A single propositional symbol, like A (A can be True or False)

    A B
- Logical connectives
  - ¬ not
  - ∧ and (**conjunction**)
  - ∨ or (**disjunction**)
  - ⇒ $(or \rightarrow)$ implication
  - ⇔ if and only if
- Complex sentence
  - $A \vee B$, $A \vee \neg C \Rightarrow B$, ...
- A special type of sentence: Horn clause

Syntax Forms

Syntax Forms:

CNF (Conjunction Normal Form): $(A \vee \neg B) \wedge (A \vee \neg C \vee D)$

"∧"

- CNF consists of **clauses** that are connected by conjunction.
  - "∨" **Clauses**: disjunctions of **literals** (a symbol or its negation).

    A, ¬A
- $(A \vee \neg B) \wedge (A \vee \neg C \vee D)$
  - 2 clauses: $(A \vee \neg B)$, $(A \vee \neg C \vee D)$
  - 4 variables: A, B, C, D
  - Literals: $A, \neg B, \neg C, D$

DNF (Disjunction Normal Form): $(A \wedge \neg B) \vee (A \wedge \neg C \wedge D)$

- All propositional sentences can be converted to CNF/DNF.
- We will mainly use CNF. For most algorithms, you will need to standardize the sentence by converting it to CNF first.

Horn Clause

Horn Clause:

A Horn clause is a clause (a disjunction of literals) with at most one positive

- **¬A V ¬B V ¬C V D**
  - **A ^ B ^ C =>D**

Horn Form: When KB (knowledge base) = **conjunction** of Horn clauses

Why do we care about Horn clause?

- It's a special type! If the sentences are Horn clauses, inference can be done in linear time (exponential for general sentences)

Semantics

P ⇒ Q And
Q⇒P

- Answers when is a sentence true:

$P \Rightarrow Q$ is equivalent to $\neg P \vee Q$

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*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

Questions?

- My slides take the following materials as references:
  - Shirley Chen’s slides
  - Yewen Wang’s (Winter 2020’s TA) slides
  - Prof. Quanquan Gu’s (Winter 2020) slides

Thank you!