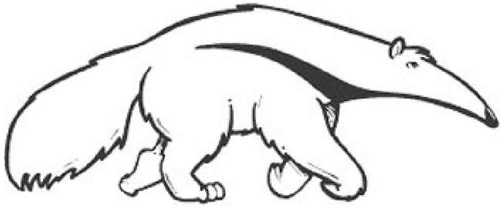
Final Review

CS171, Fall Quarter, 2018
Introduction to Artificial Intelligence
Prof. Richard Lathrop



Read Beforehand: R&N All Assigned Reading

BREN:ICS
INFORMATION AND COMPUTER SCIENCES
UNIVERSITY of CALIFORNIA IRVINE

Review Adversarial (Game) Search Chapter 5.1-5.4

- Minimax Search with Perfect Decisions (5.2)
  - Impractical in most cases, but theoretical basis for analysis
- Minimax Search with Cut-off (5.4)
  - Replace terminal leaf utility by heuristic evaluation function
- Alpha-Beta Pruning (5.3)
  - The fact of the adversary leads to an advantage in search!
- Practical Considerations (5.4)
  - Redundant path elimination, look-up tables, etc.

Games as Search

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over
  - Winner gets reward, loser gets penalty.
  - “Zero sum” means the sum of the reward and the penalty is a constant.

- Formal definition as a search problem:
  - **Initial state:** Set-up specified by the rules, e.g., initial board configuration of chess.
  - **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):** Transition model defines the result of a move.
  - (**2^{nd} ed.: Successor function:** list of (move,state) pairs specifying legal moves.)
  - **Terminal-Test(s):** Is the game finished? True if finished, false otherwise.
  - **Utility function(s,p):** Gives numerical value of terminal state s for player p.
    - E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
    - E.g., win (+1), lose (0), and draw (1/2) in chess.

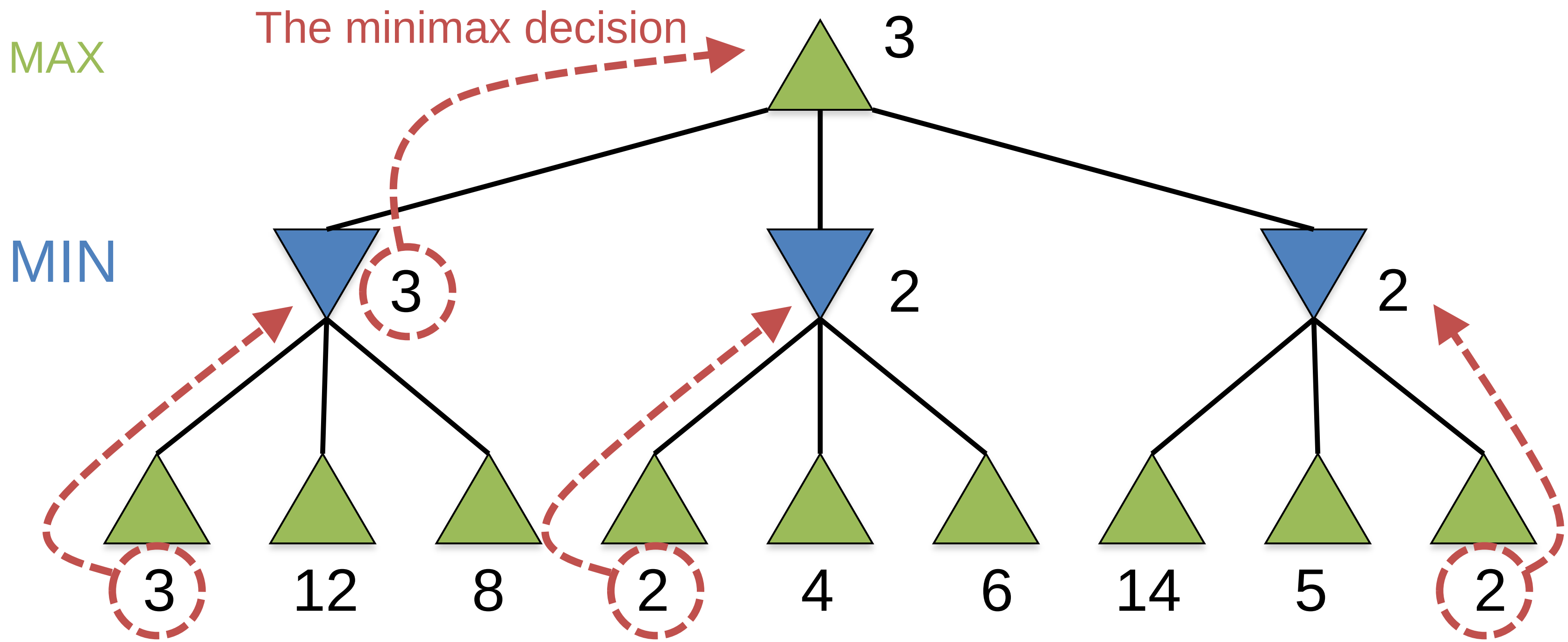
- MAX uses search tree to determine “best” next move.

An optimal procedure: The Min-Max method

Will find the optimal strategy and best next move for Max:

- 1. Generate the whole game tree, down to the leaves.
- 2. Apply utility (payoff) function to each leaf.
- 3. Back-up values from leaves through branch nodes:
  - a Max node computes the Max of its child values
  - a Min node computes the Min of its child values
- 4. At root: choose move leading to the child of highest value.

Two-ply Game Tree



Minimax maximizes the utility of the worst-case outcome for MAX

Pseudocode for Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
return arg max_{a∈ACTIONS(state)} MIN-VALUE(Result(state,a))
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a in ACTIONS(state) do
     v ← MAX(v,MIN-VALUE(Result(state,a)))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a in ACTIONS(state) do
     v ← MIN(v,MAX-VALUE(Result(state,a)))
  return v
```

Properties of minimax

- **Complete?**
  - Yes (if tree is finite).

- **Optimal?**
  - Yes (against an optimal opponent).
  - Can it be beaten by an opponent playing sub-optimally?
    - No. (Why not?)

- **Time complexity?**
  - $O(b^m)$

- **Space complexity?**
  - O(bm) (depth-first search, generate all actions at once)
  - O(m) (backtracking search, generate actions one at a time)

Cutting off search

MinimaxCutoff is identical to MinimaxValue except

1. Terminal? is replaced by Cutoff?
2. Utility is replaced by Eval

Does it work in practice?

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov

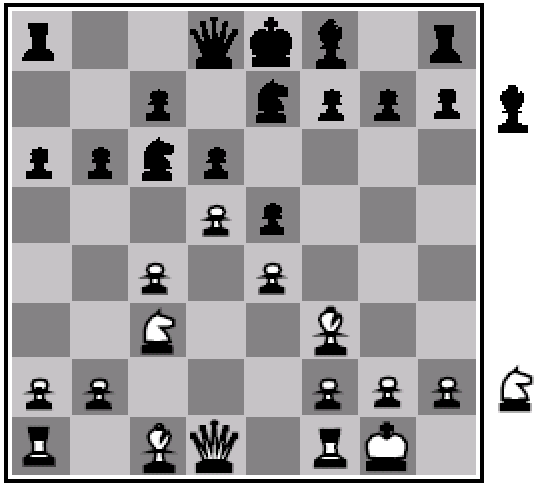
Static (Heuristic) Evaluation Functions

- An Evaluation Function:
  - Estimates how good the current board configuration is for a player.
  - Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces

- Typical values from -infinity (loss) to +infinity (win) or [-1, +1].

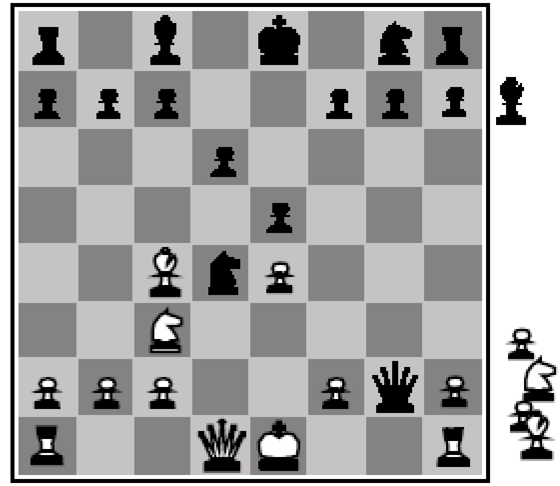
- If the board evaluation is X for a player, it's -X for the opponent
  - "Zero-sum game"

Evaluation functions



Black to move

White slightly better



White to move

Black winning

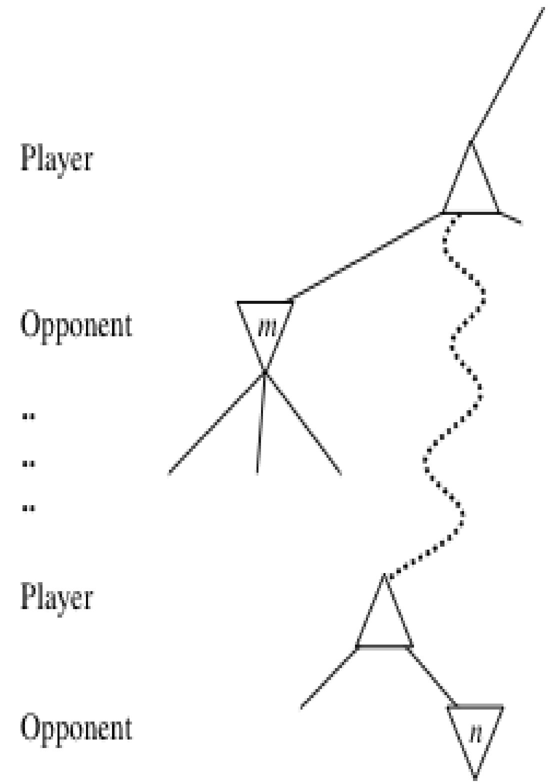
For chess, typically *linear* weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

General alpha-beta pruning

- Consider a node *n* in the tree ---
- If player has a better choice at:
  - Parent node of n
  - Or any choice point further up
- Then *n* will never be reached in play.
- Hence, when that much is known about *n*, it can be pruned.



Alpha-beta Algorithm

- Depth first search
  - only considers nodes along a single path from root at any time

α = highest-value choice found at any choice point of path for MAX
(initially, α = −infinity)
β = lowest-value choice found at any choice point of path for MIN
(initially, β = +infinity)

- Pass current values of α and β down to child nodes during search.
- Update values of α and β during search:
  - MAX updates α at MAX nodes
  - MIN updates β at MIN nodes
- Prune remaining branches at a node when $\alpha \geq \beta$

Pseudocode for Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** *an action*
  inputs: *state*, current state in game
  $\boxed{v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty)}$
  return the *action* in ACTIONS(*state*) with value *v*

---

function $\boxed{\text{MAX-VALUE}(state, \alpha, \beta)}$ **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for *a* in ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(Result(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

(MIN-VALUE is defined analogously)

When to Prune?

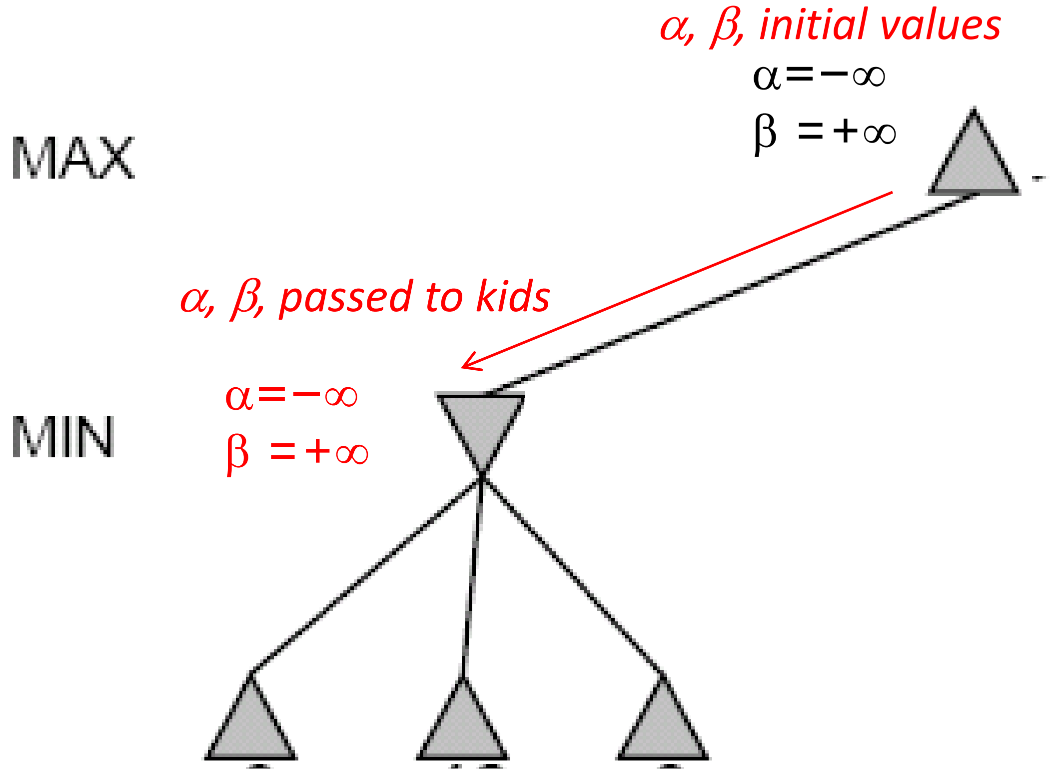
- **Prune whenever $\alpha \geq \beta$.**
  - Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
    - **Max nodes update alpha** based on children's returned values.
  - Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors.
    - **Min nodes update beta** based on children's returned values.

α/β Pruning vs. Returned Node Value

- Some students are confused about the use of α/β pruning vs. the returned value of a node
- α/β are used **ONLY FOR PRUNING**
  - α/β have no effect on anything other than pruning
  - IF (α >= β) THEN prune & return current node value
- Returned node value = "best" child seen so far
  - Maximum child value seen so far for MAX nodes
  - Minimum child value seen so far for MIN nodes
  - If you prune, return to parent "best" child so far
- Returned node value is received by parent

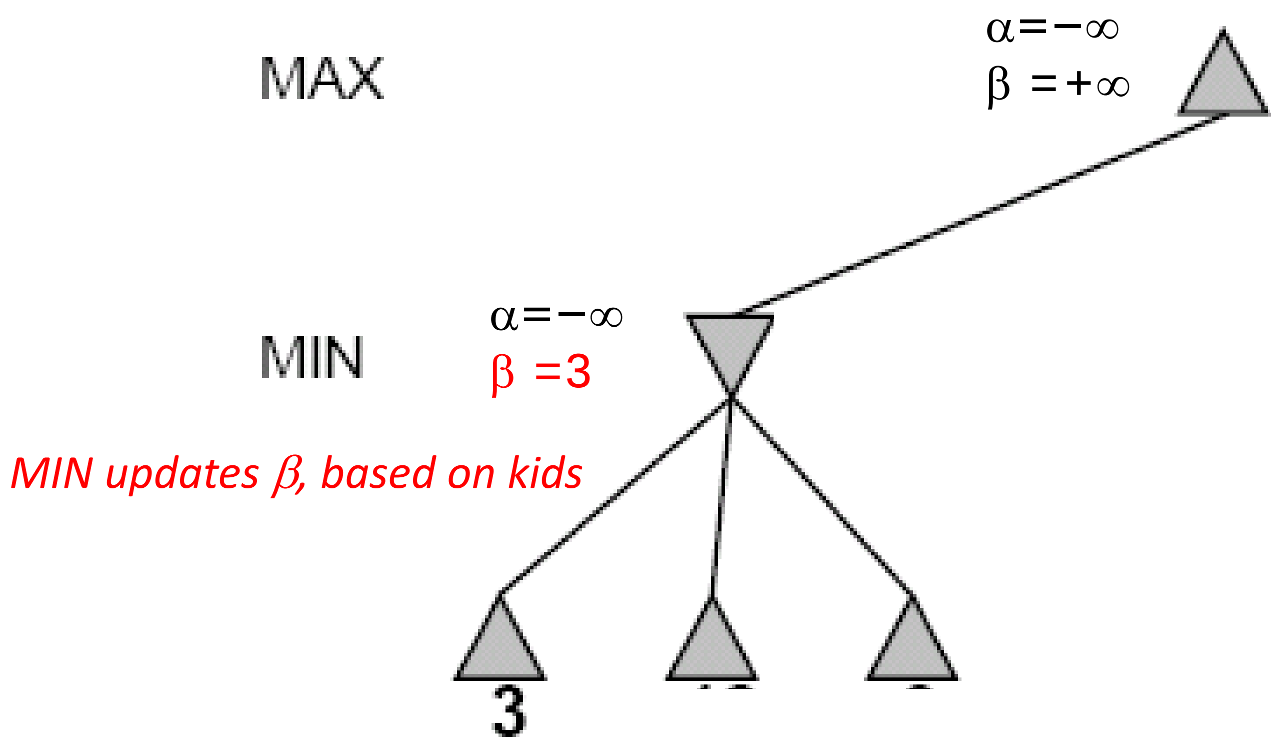
Alpha-Beta Example Revisited

Do DF-search until first leaf

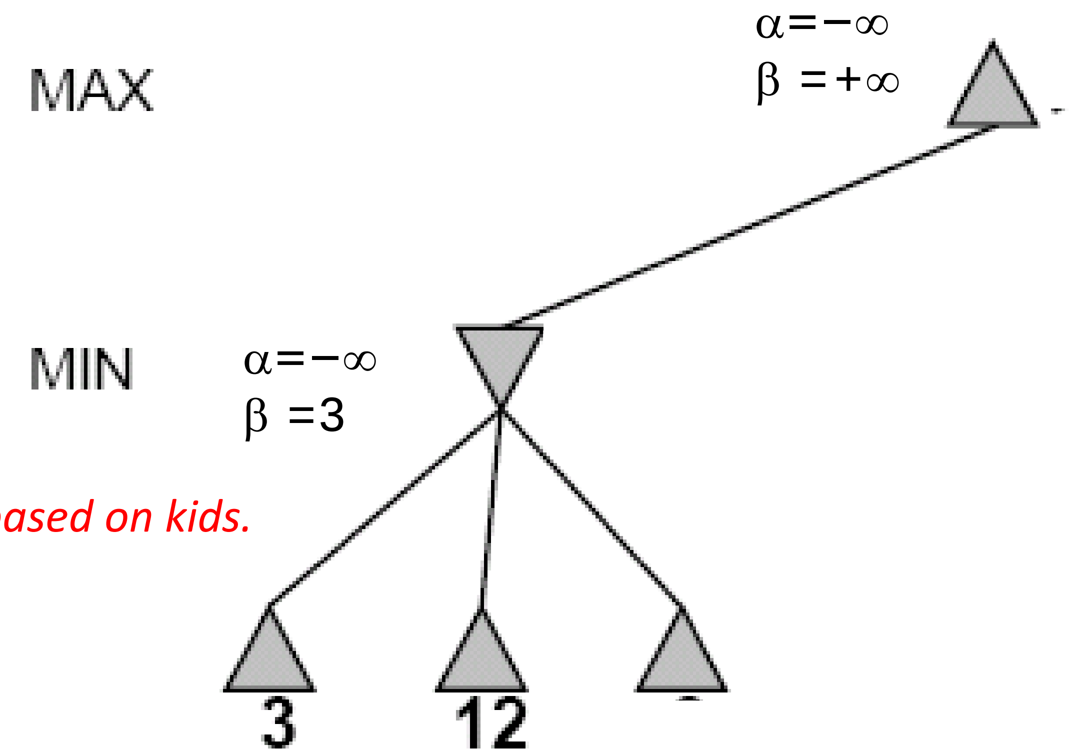


<u>Review Detailed Example of Alpha-Beta Pruning in lecture slides.</u>

Alpha-Beta Example (continued)

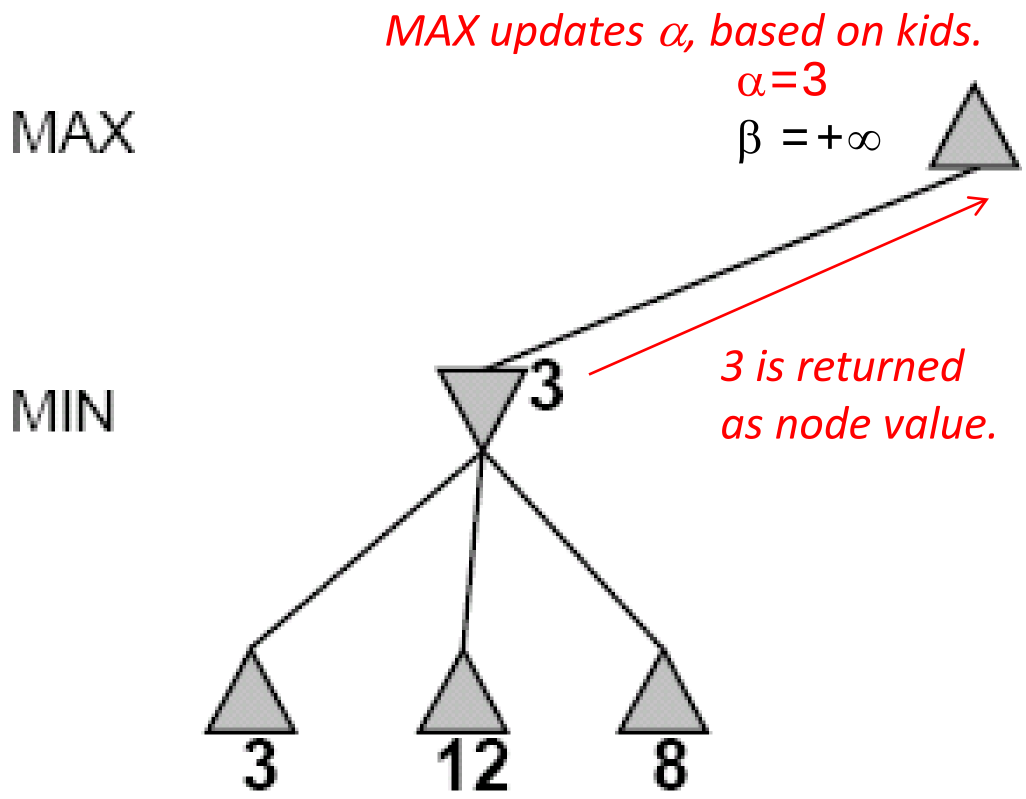


Alpha-Beta Example (continued)

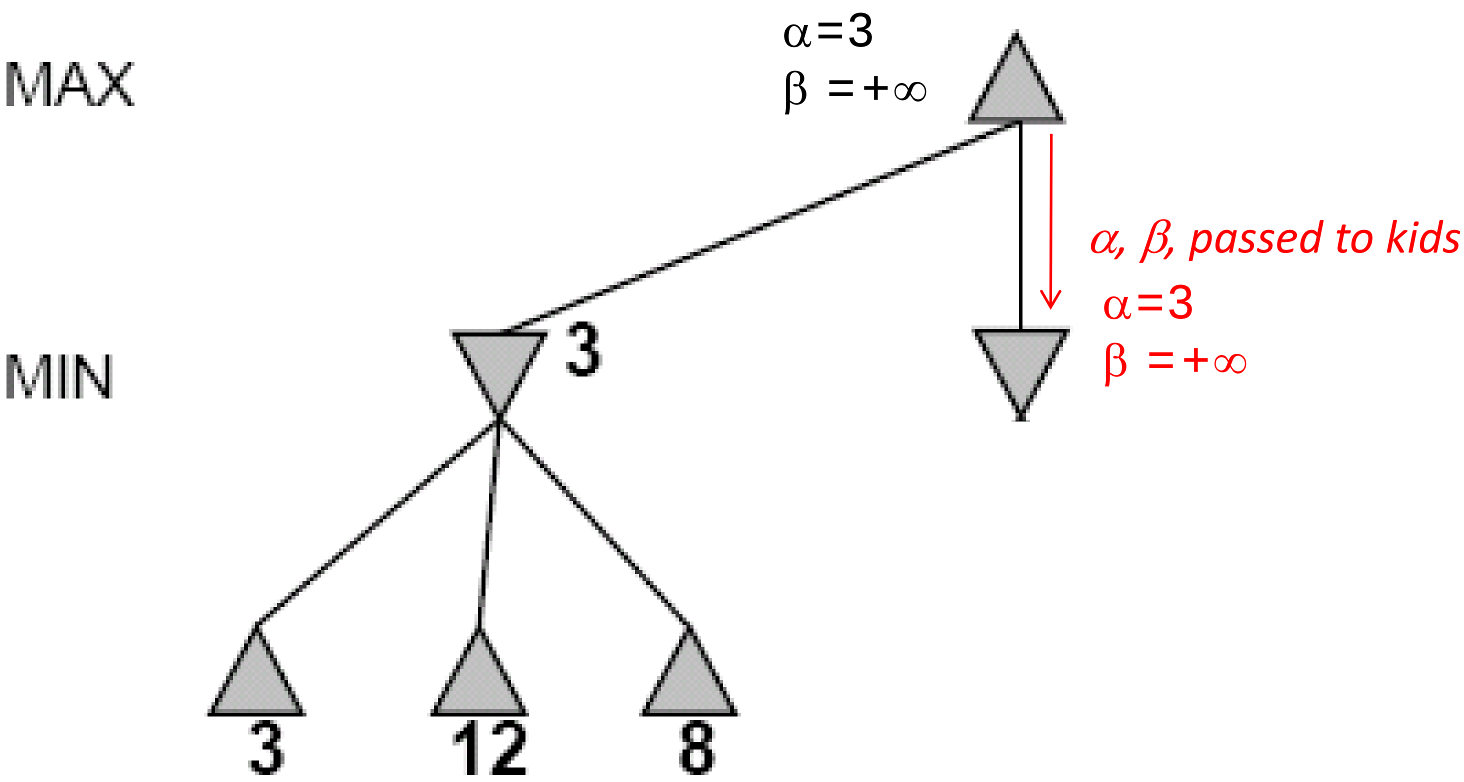


MIN updates β, based on kids.
No change.

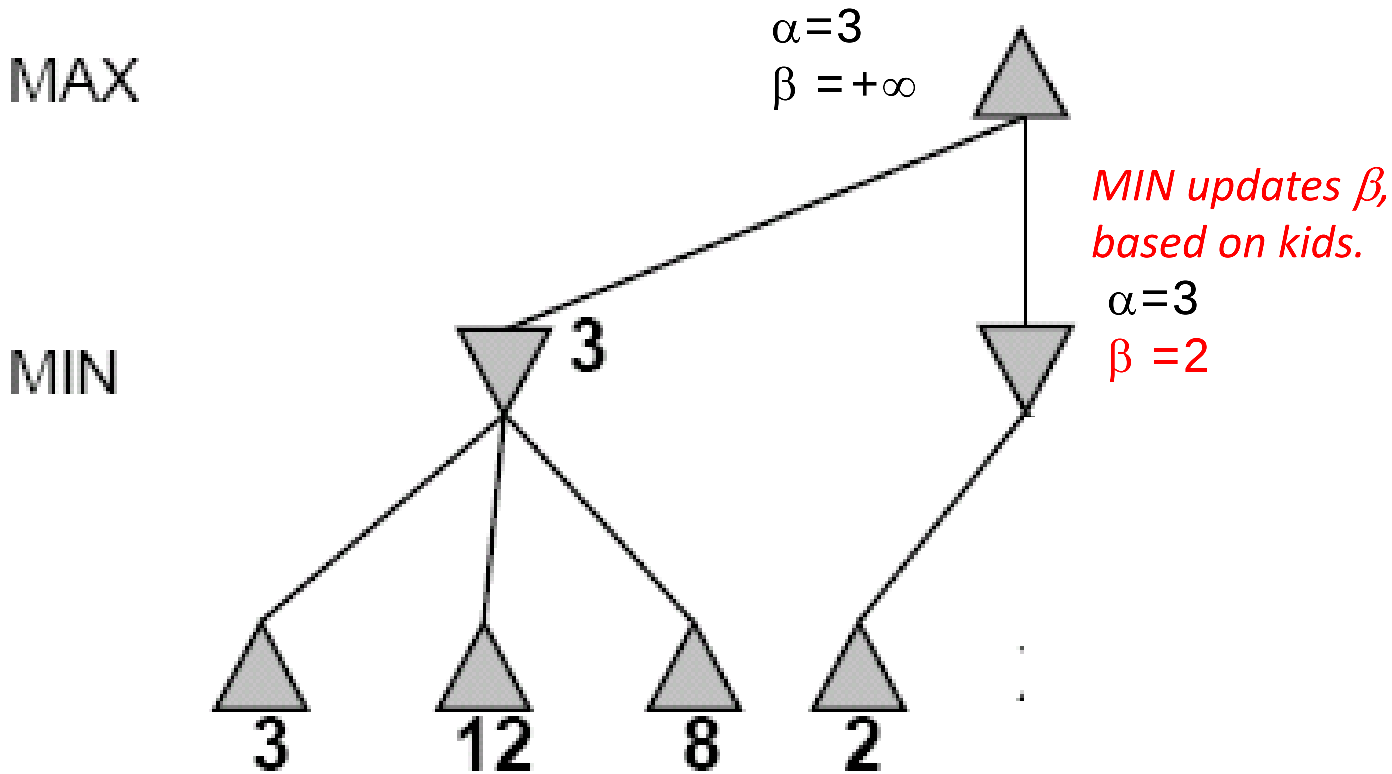
Alpha-Beta Example (continued)



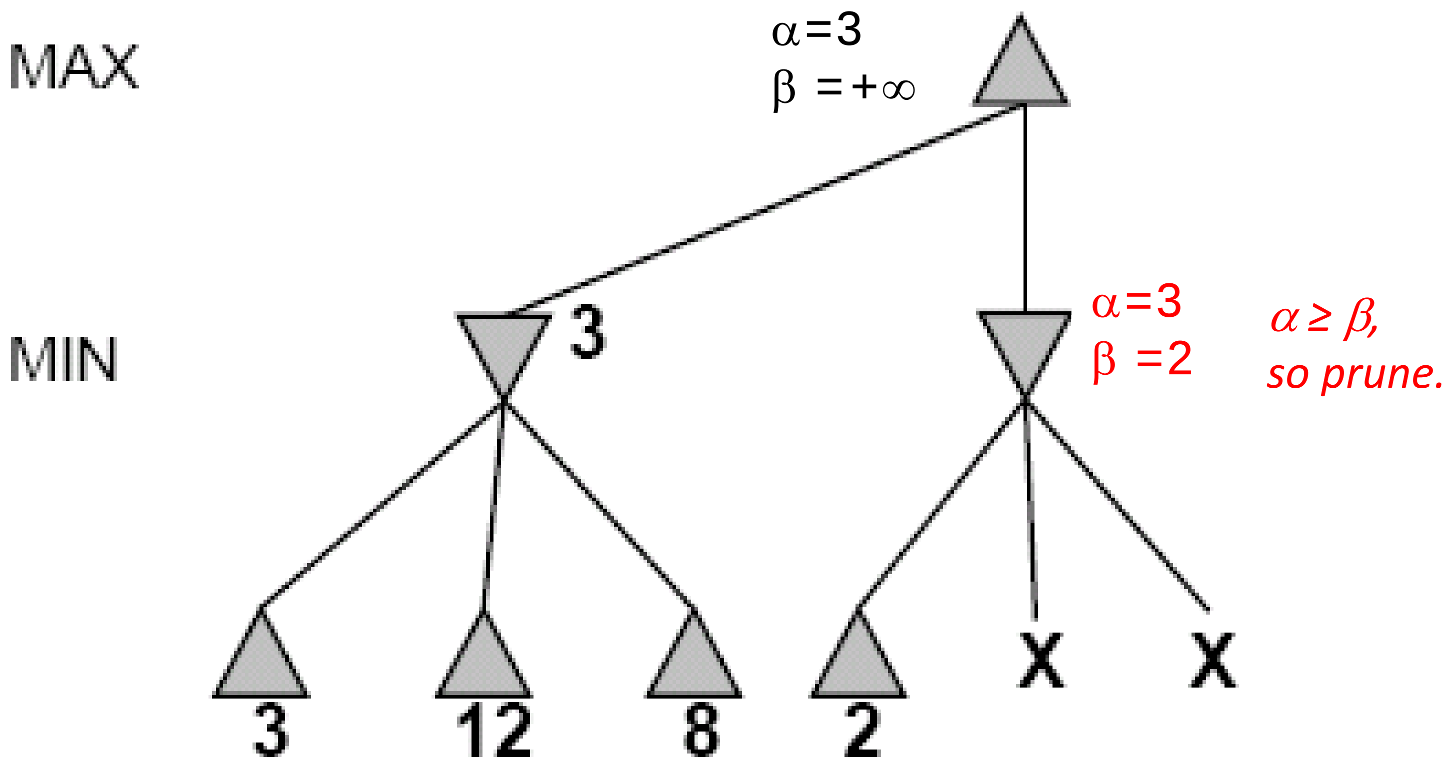
Alpha-Beta Example (continued)



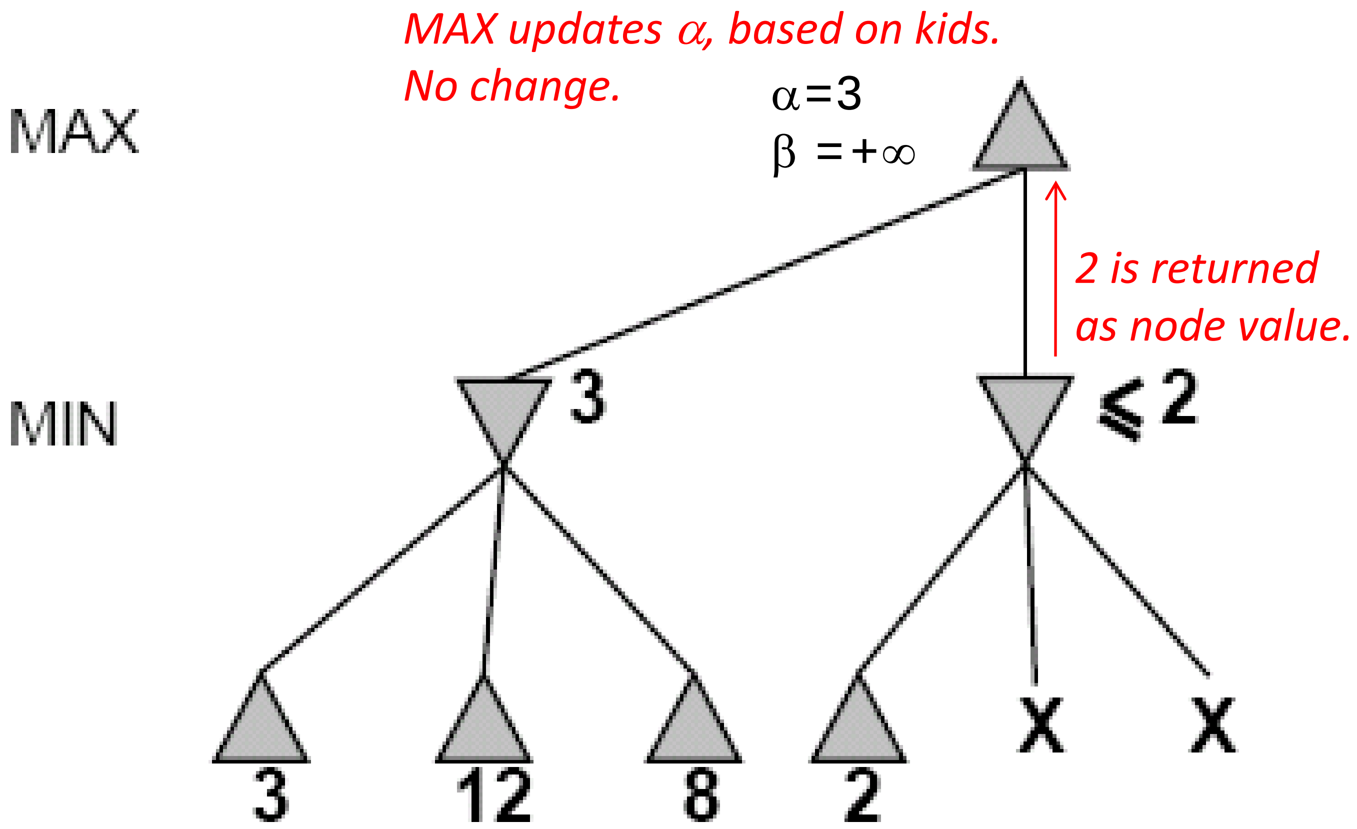
Alpha-Beta Example (continued)



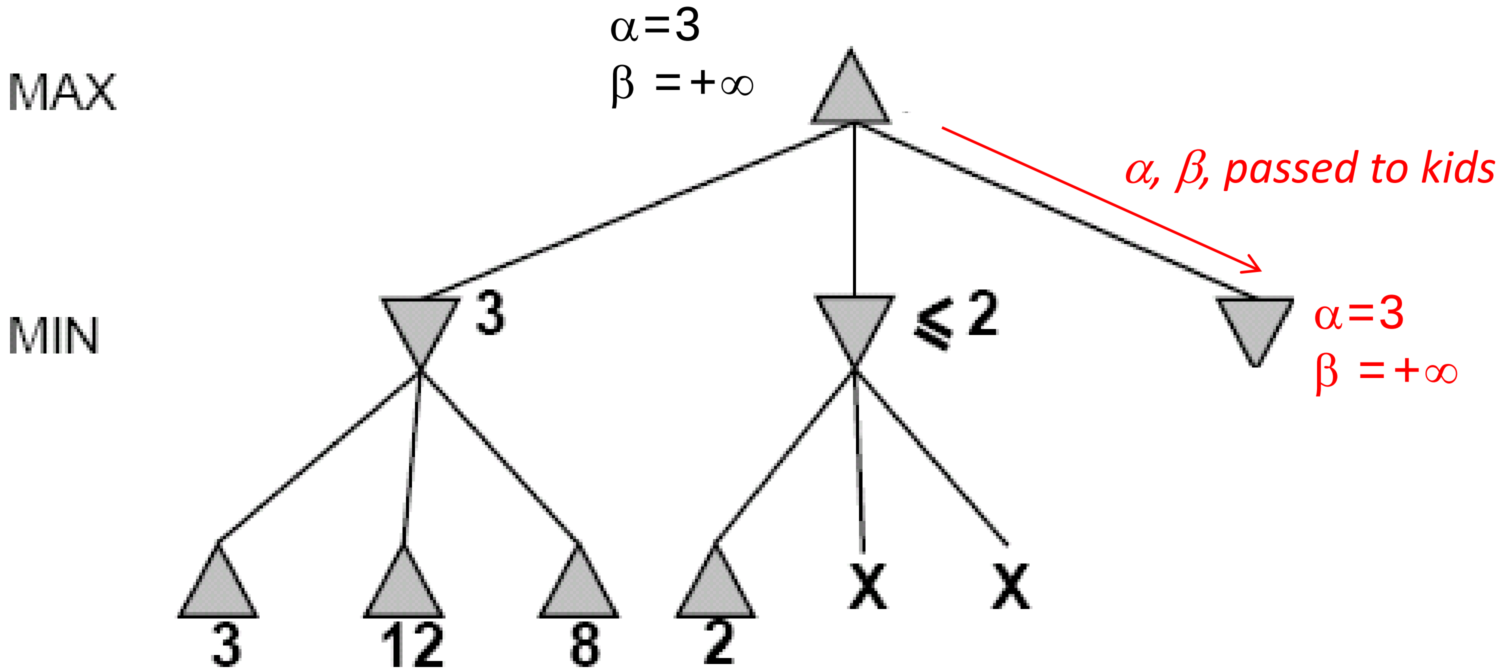
Alpha-Beta Example (continued)



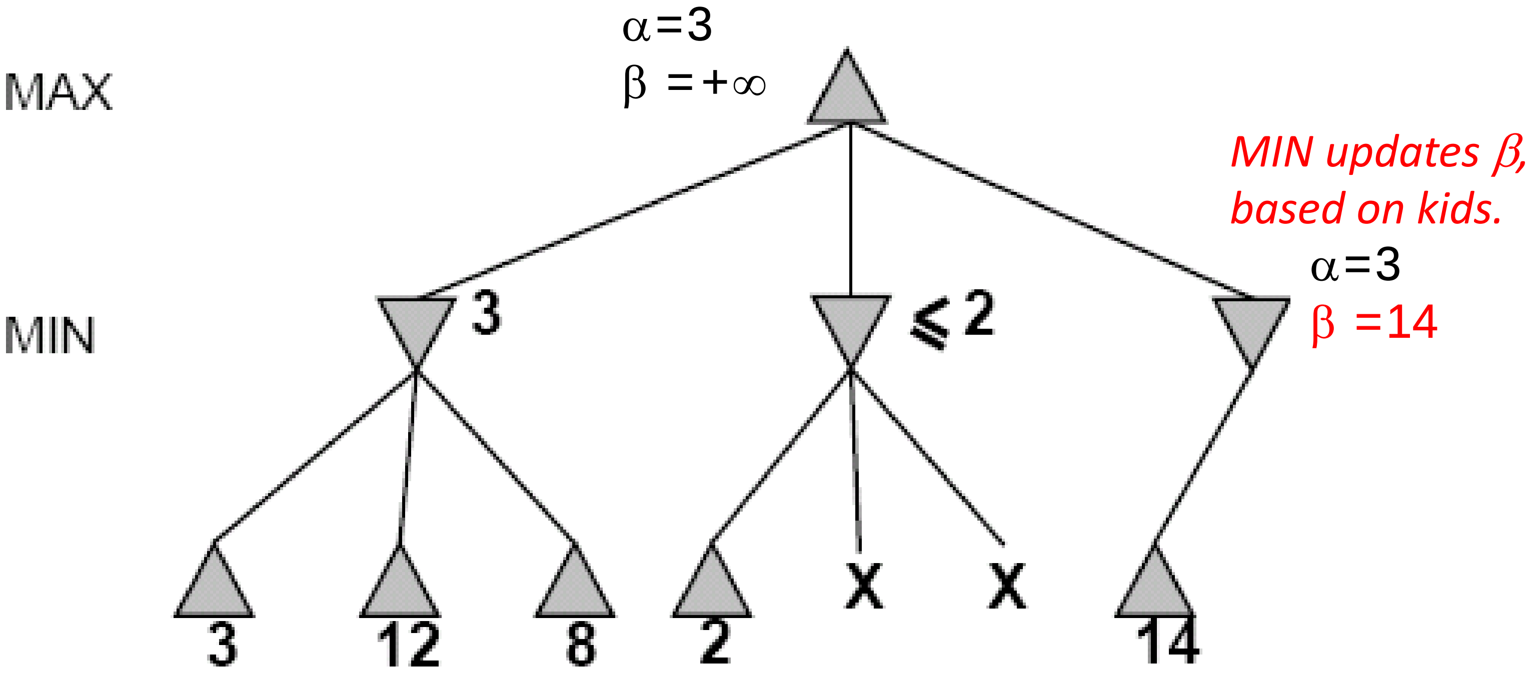
Alpha-Beta Example (continued)



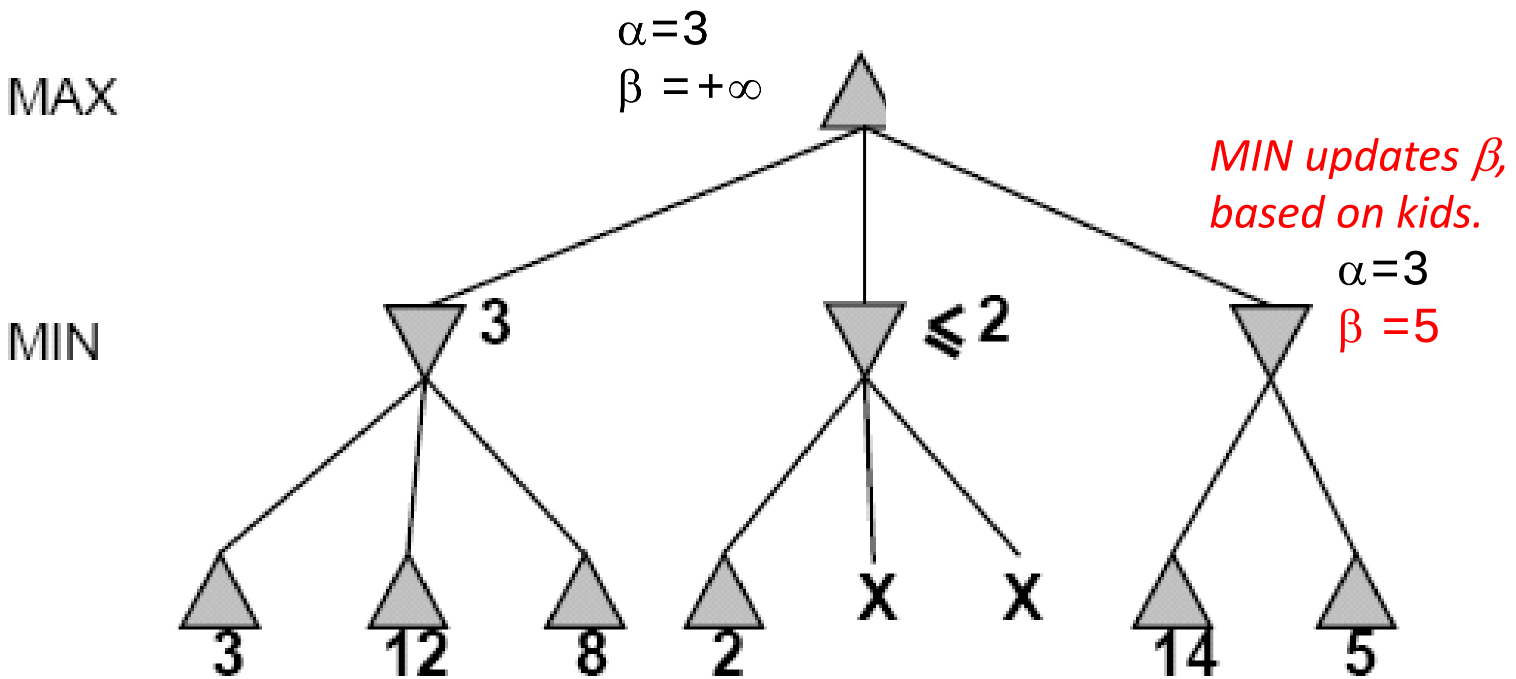
Alpha-Beta Example (continued)



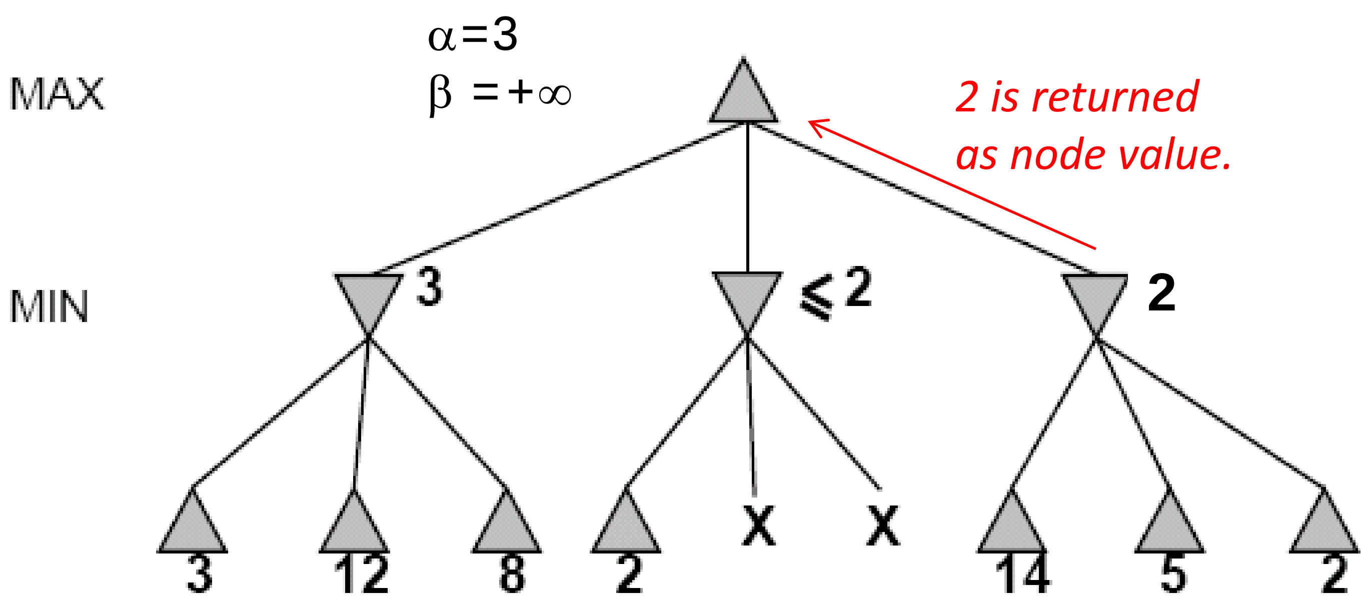
Alpha-Beta Example (continued)



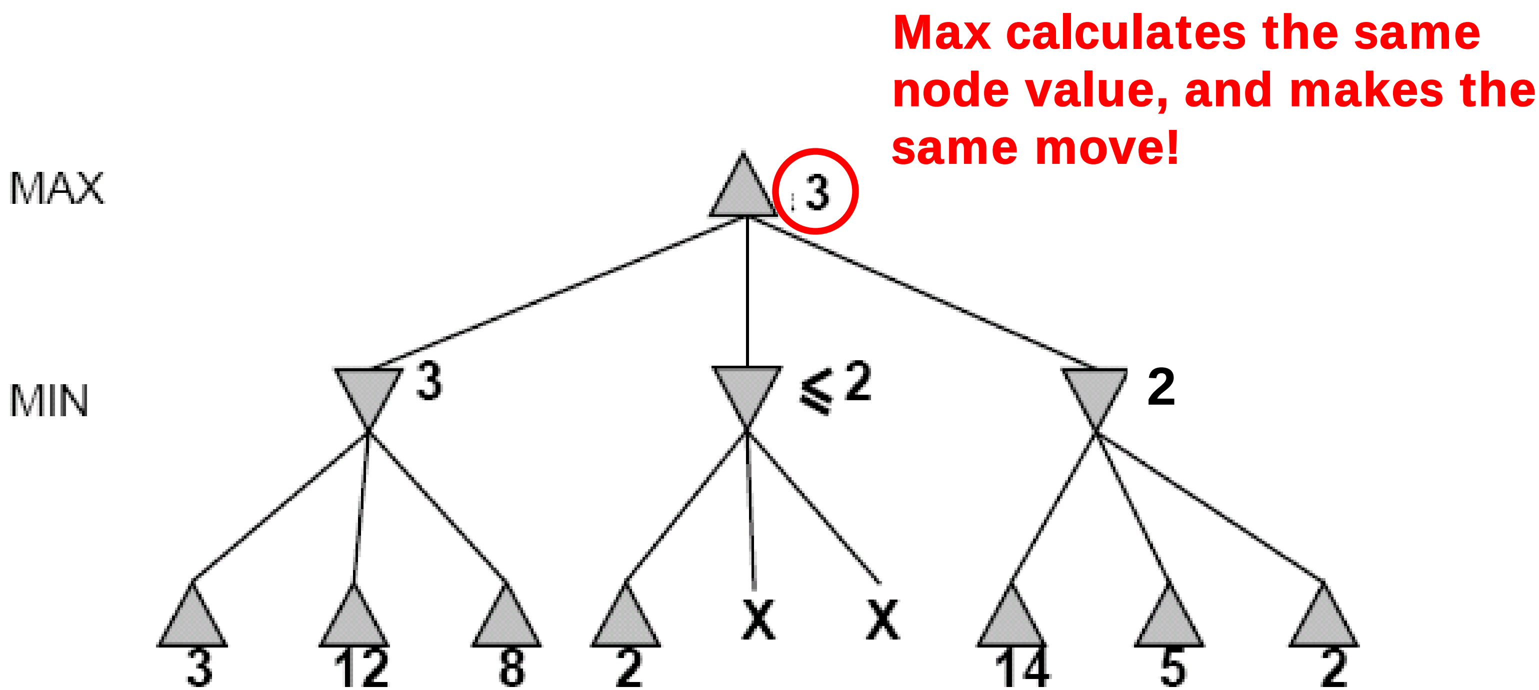
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Review Detailed Example of Alpha-Beta Pruning in lecture slides.

Review Constraint Satisfaction
R&N 6.1-6.4 (except 6.3.3)

- What is a CSP?

- Backtracking search for CSPs
  - Choose a variable, then choose an order for values
  - Minimum Remaining Values (MRV), Degree Heuristic (DH), Least Constraining Value (LCV)

- Constraint propagation
  - Forward Checking (FC), Arc Consistency (AC-3)

- Local search for CSPs
  - Min-conflicts heuristic

Constraint Satisfaction Problems

- What is a CSP?
  - Finite set of variables, $X_1, X_2, \ldots, X_n$
  - Nonempty domain of possible values for each: $D_1, \ldots, D_n$
  - Finite set of constraints, $C_1, \ldots, C_m$
    - Each constraint C_i limits the values that variables can take, e.g., $X_1 \neq X_2$
  - Each constraint C_i is a pair: C_i = (*scope*, *relation*)
    - Scope = tuple of variables that participate in the constraint
    - Relation = list of allowed combinations of variables

      May be an explicit list of allowed combinations

      May be an abstract relation allowing membership testing & listing

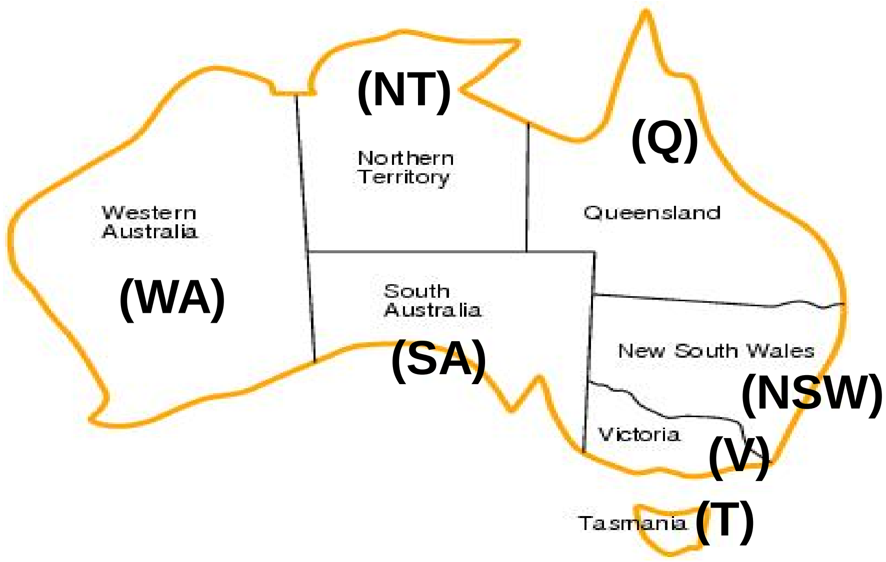
- CSP benefits
  - Standard representation pattern
  - Generic goal and successor functions
  - Generic heuristics (no domain-specific expertise required)

CSPs --- what is a solution?

- A ***state*** is an ***assignment*** of values to some variables.
  - ***Complete*** assignment
    - = every variable has a value.
  - ***Partial*** assignment
    - = some variables have no values.
  - ***Consistent*** assignment
    - = assignment does not violate any constraints

- A ***solution*** is a ***complete*** and ***consistent*** assignment.

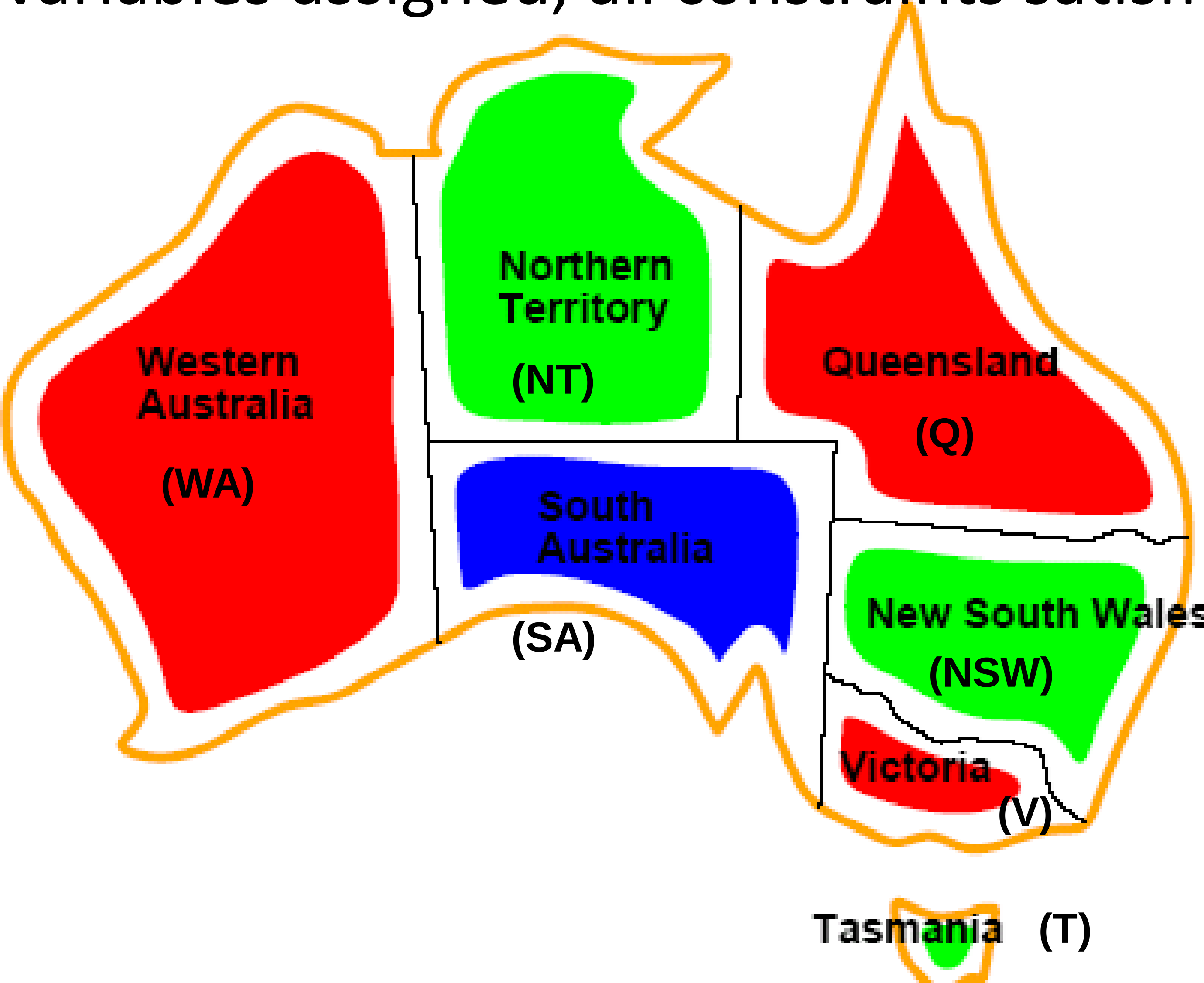
CSP example: map coloring



- **<u>Variables:</u>** *WA, NT, Q, NSW, V, SA, T*
- **<u>Domains:</u>** D_i=*{red,green,blue}*
- **<u>Constraints:</u>** Adjacent regions must have different colors, e.g., $WA \neq NT$.

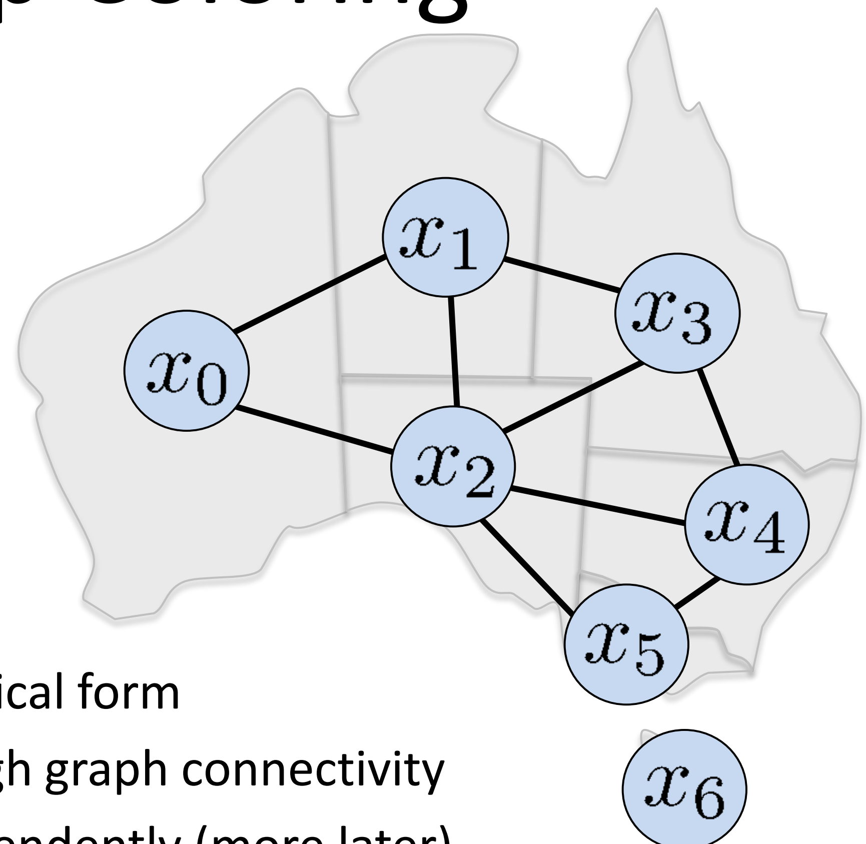
Example: Map coloring solution

All variables assigned, all constraints satisfied.

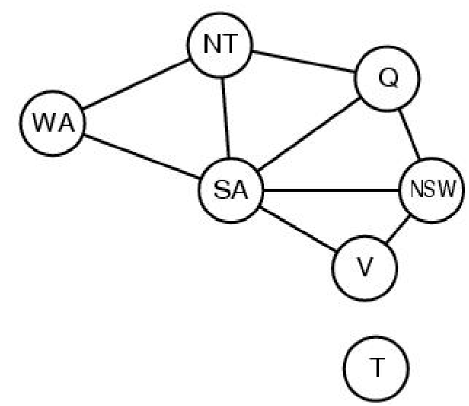


Example: Map Coloring

- Constraint graph
  - Vertices: variables
  - Edges: constraints
    (connect involved variables)



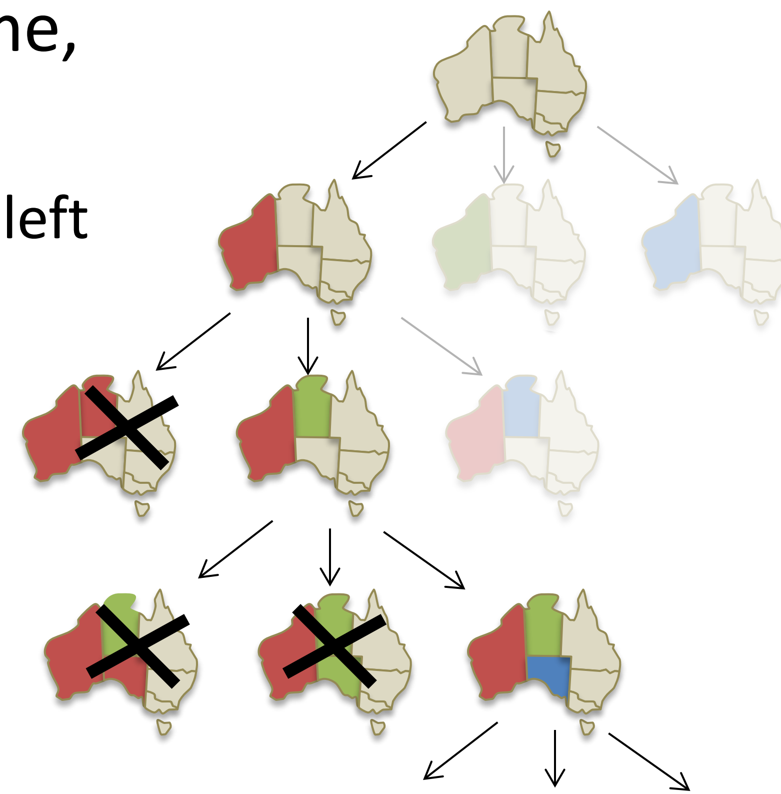
- Graphical model
  - Abstracts the problem to a canonical form
  - Can reason about problem through graph connectivity
  - Ex: Tasmania can be solved independently (more later)



- Binary CSP
  - Constraints involve at most two variables
  - Sometimes called “pairwise”

Backtracking search

- Similar to depth-first search
  - At each level, pick a single variable to expand
  - Iterate over the domain values of that variable
- Generate children one at a time,
  - One child per value
  - Backtrack when no legal values left
- Uninformed algorithm
  - Poor general performance

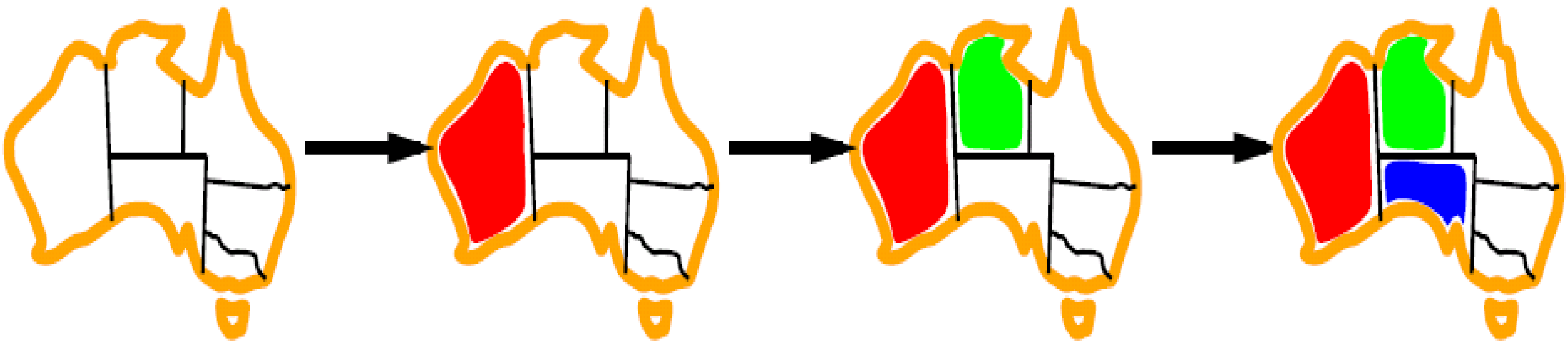


Backtracking search (Figure 6.5)

```
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({} , csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return 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 according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RRECURSIVE-BACTRACKING(assignment, csp)
            if result ≠ failure  then return result
            remove {var=value} from assignment
    return failure
```

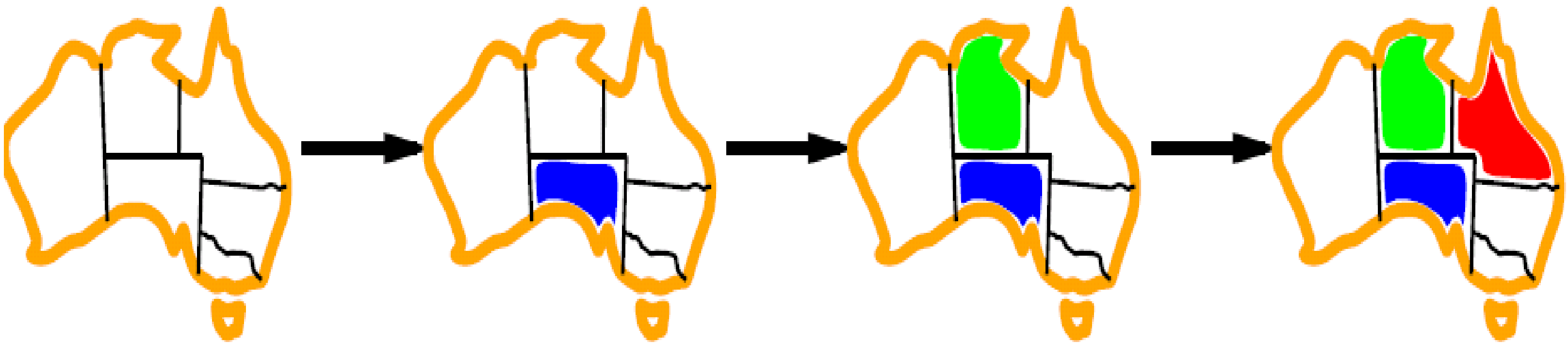
Minimum remaining values (MRV)



var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)

- A.k.a. most constrained variable heuristic
- *Heuristic Rule*: choose variable with the fewest legal moves
  - e.g., will immediately detect failure if X has no legal values

Degree heuristic for the initial variable



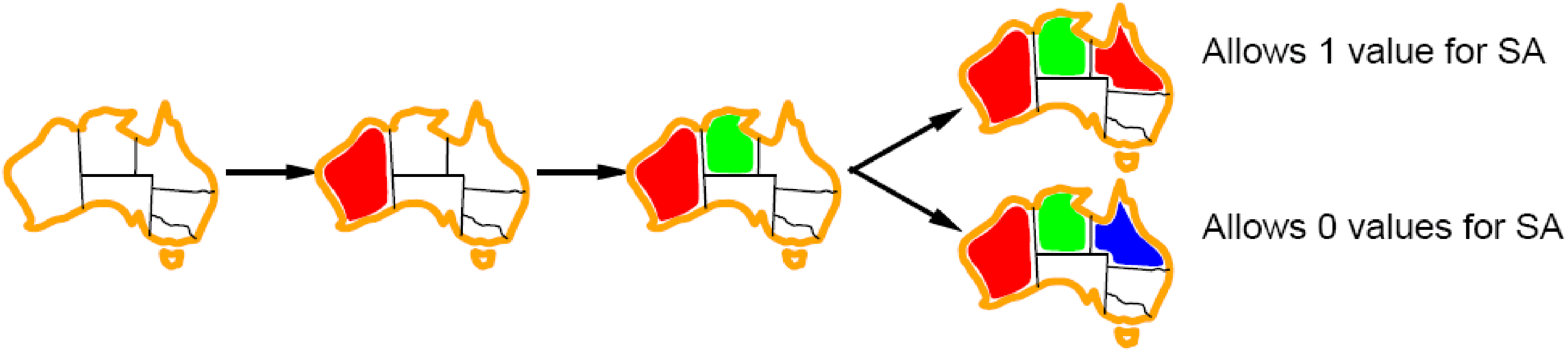
- *Heuristic Rule*: select variable that is involved in the largest number of constraints on other unassigned variables.
- Degree heuristic can be useful as a tie breaker.
- *In what order should a variable's values be tried?*

Backtracking search (Figure 6.5)

function BACKTRACKING-SEARCH(*csp*) **return** a solution or failure
  return RECURSIVE-BACKTRACKING(*{}* , *csp*)

function RECURSIVE-BACKTRACKING(*assignment, csp*) **return** 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* according to CONSTRAINTS[*csp*] **then**
    add *{var=value}* to assignment
    result ← RRECURSIVE-BACTRACKING(*assignment, csp*)
    if *result* ≠ *failure* **then return** *result*
    remove *{var=value}* from *assignment*
  return *failure*

Least constraining value for value-ordering



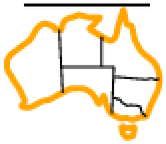
- Least constraining value heuristic
- Heuristic Rule: given a variable choose the least constraining value
  - leaves the maximum flexibility for subsequent variable assignments

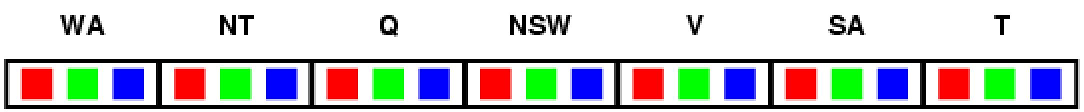
Look-ahead: Constraint propagation

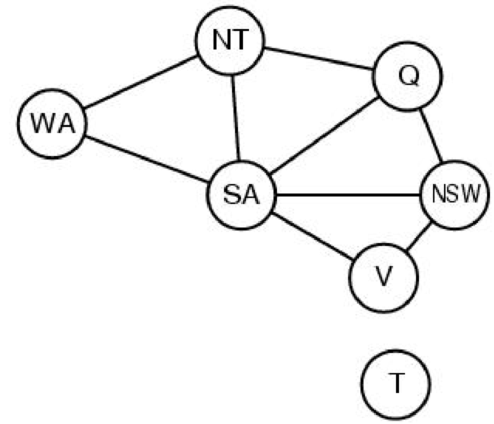
- Intuition:
  - Some domains have values that are inconsistent with the values in some other domains
  - Propagate constraints to remove inconsistent values
  - Thereby reduce future branching factors
- Forward checking
  - Check each unassigned neighbor in constraint graph
- Arc consistency (AC-3 in R&N)
  - Full arc-consistency everywhere until quiescence
  - Can run as a preprocessor
    - Remove obvious inconsistencies
  - Can run after each step of backtracking search
    - Maintaining Arc Consistency (MAC)

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - ONLY check neighbors of most recently assigned variable

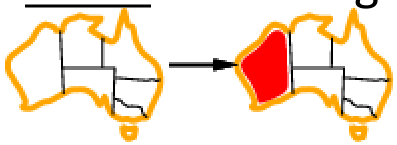


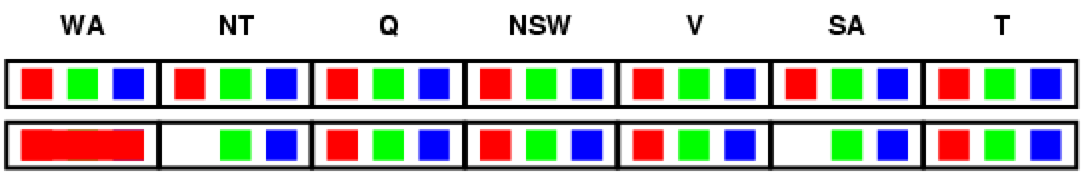


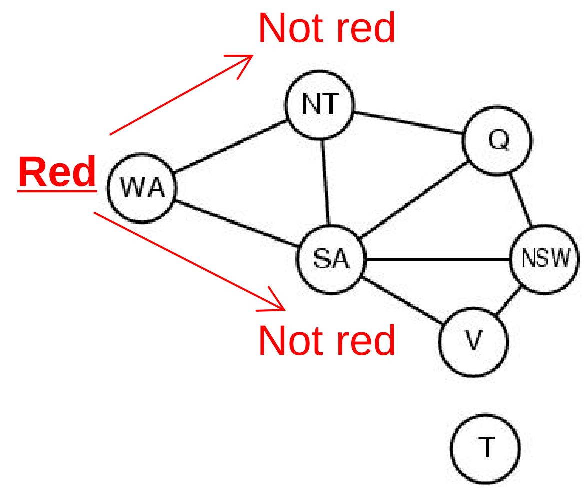


Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - <u>ONLY</u> check neighbors of <u>most recently assigned variable</u>







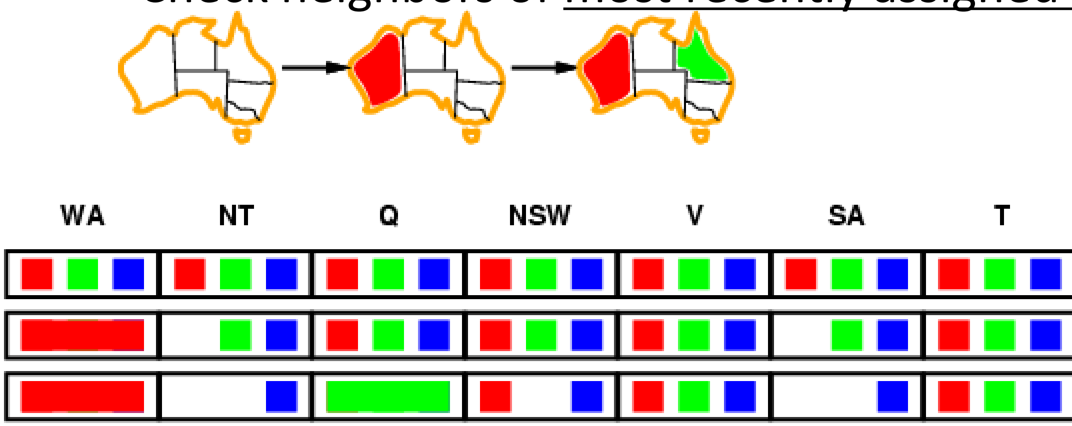
Assign {WA = red}

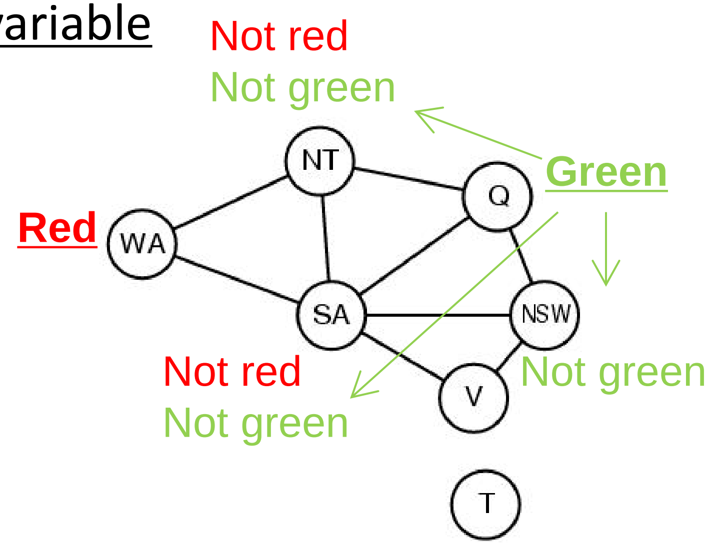
Effect on other variables (neighbors of WA):

- NT can no longer be red
- SA can no longer be red

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - Check neighbors of <u>most recently assigned variable</u>





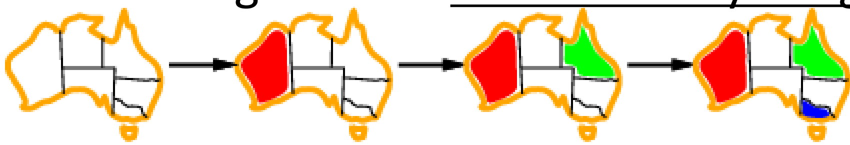
Assign {Q = green}
Effect on other variables (neighbors of Q):

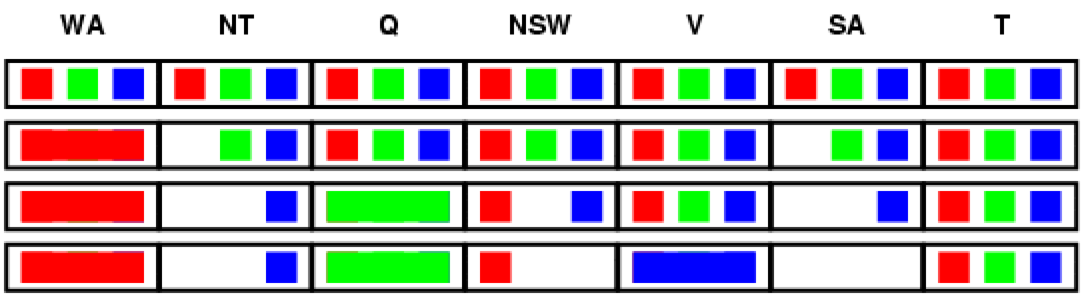
- NT can no longer be green
- SA can no longer be green
- NSW can no longer be green

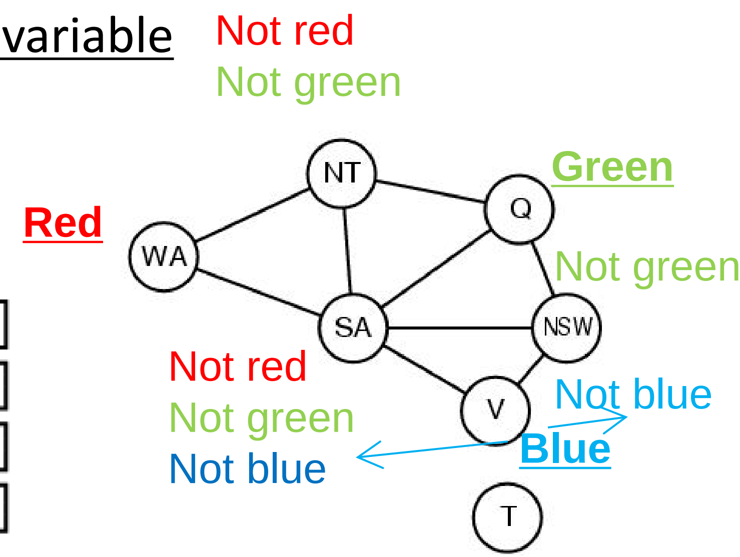
<u>(We already have failure, but FC is too simple to detect it now)</u>

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Backtrack when any variable has no legal values
  - Check neighbors of <u>most recently assigned variable</u>







Assign {V = blue}
Effect on other variables (neighbors of V):

- NSW can no longer be blue
- SA can no longer be blue **(no values possible!)**

Forward checking has detected that this partial assignment is inconsistent with any complete assignment

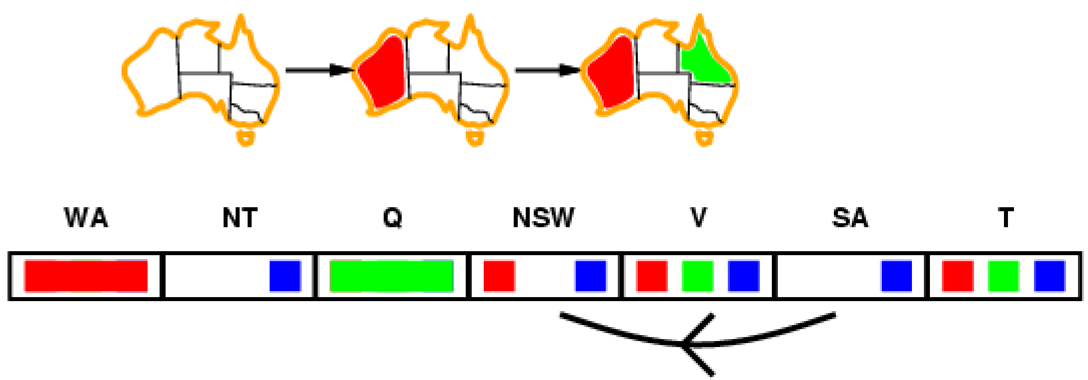
Arc consistency (AC-3) algorithm

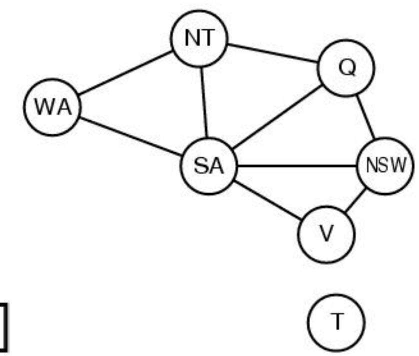
- An Arc $X \rightarrow Y$ is consistent iff for *every* value x of X there is *some* value y of Y that is consistent with x
- Put all arcs $X \rightarrow Y$ on a queue
  - Each undirected constraint graph arc is two directed arcs
  - Undirected $X — Y$ becomes directed $X \rightarrow Y$ and $Y \rightarrow X$
  - $X \rightarrow Y$ and $Y \rightarrow X$ both go on queue, separately
- Pop one arc $X \rightarrow Y$ and remove any inconsistent values from X
- If any change in X, put all arcs $Z \rightarrow X$ back on queue, where Z is any neighbor of X that is not equal to Y
- Continue until queue is empty

Arc consistency (AC-3)

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff (iff = if and only if)

  for every value x of X there is some allowed value y *for* Y *(note: directed!)*





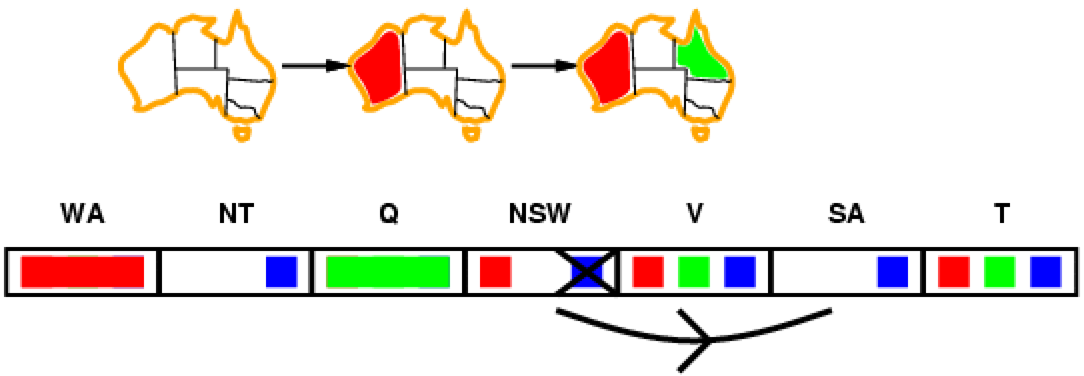
- Consider state after WA=red, Q=green
  - $SA \rightarrow NSW$ is consistent because

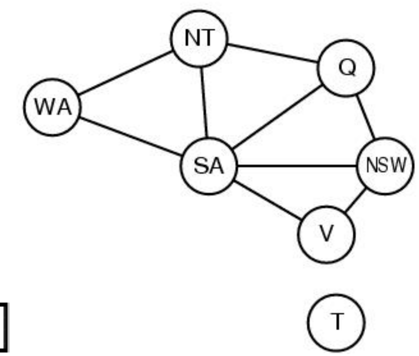
    SA = blue and NSW = red satisfies all constraints on SA and NSW

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value x of X there is some allowed value y *for* Y *(note: directed!)*





If *X* loses a value, neighbors of *X* need to be rechecked

- Consider state after WA=red, Q=green
  - NSW $\rightarrow$ SA consistent if

    NSW = red and SA = blue

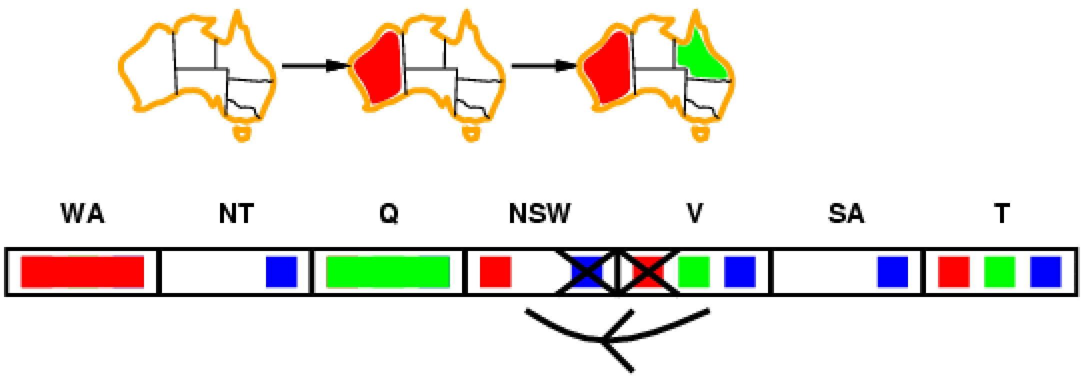
    NSW = blue and SA = ??? => NSW = blue can be pruned

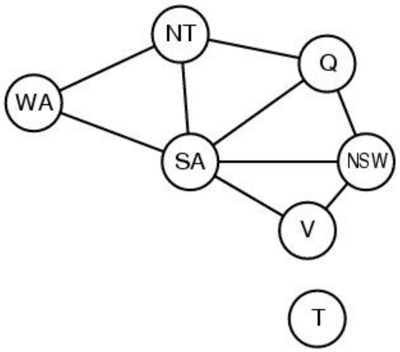
    No current domain value for SA is consistent

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \to Y$ is consistent iff

  for every value *x* of *X* there is some allowed value *y for Y* *(note: directed!)*



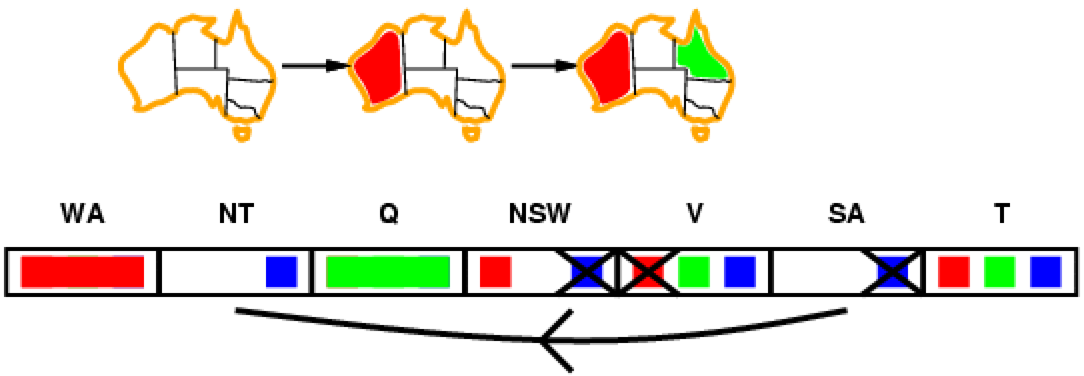


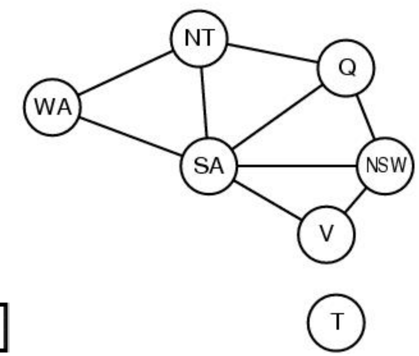
- **Enforce arc consistency:**
  - arc can be made consistent by removing blue from NSW
- **Continue to propagate constraints:**
  - Check V $\to$ NSW : not consistent for V = red; remove red from V

Arc consistency

- Simplest form of propagation makes each arc consistent
- $X \rightarrow Y$ is consistent iff

  for every value *x* of *X* there is some allowed value *y for Y* *(note: directed!)*





- Continue to propagate constraints
- SA $\rightarrow$ NT not consistent:
  - **And cannot be made consistent! Failure!**
- Arc consistency detects failure earlier than FC
  - But requires more computation: is it worth the effort?

Local search: min-conflicts heuristic

- Use complete-state representation
  - Initial state = all variables assigned values
  - Successor states = change 1 (or more) values

- For CSPs
  - allow states with unsatisfied constraints (unlike backtracking)
  - operators **reassign** variable values
  - hill-climbing with n-queens is an example

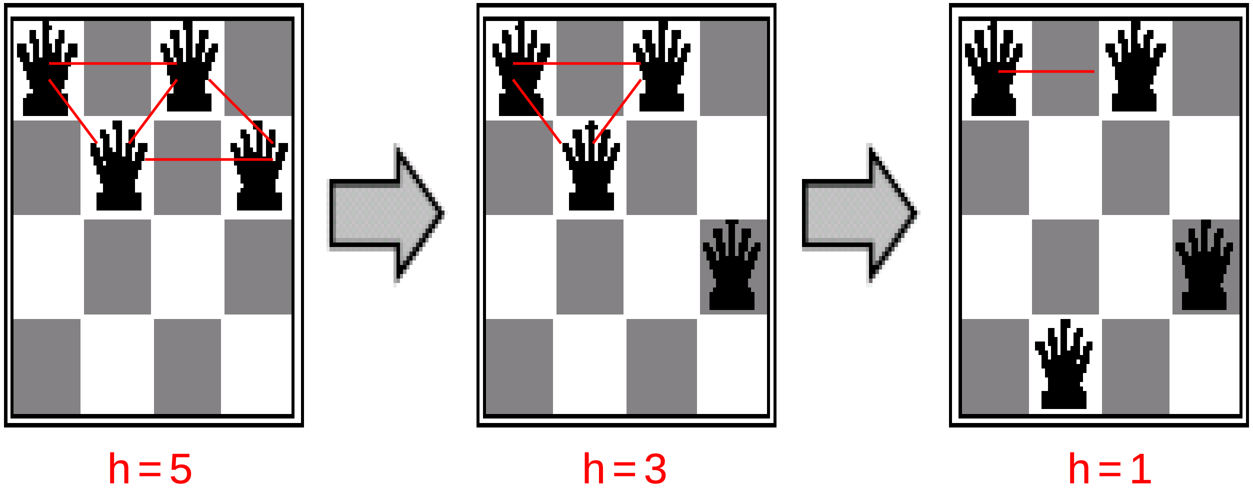
- **Variable selection:** randomly select any conflicted variable
- **Value selection:** <u>*min-conflicts heuristic*</u>
  - Select new value that results in a minimum number of conflicts with the other variables

Local search: min-conflicts heuristic

function MIN-CONFLICTS(*csp, max_steps*) **return** solution or failure
  inputs: *csp*, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  current ← a (random) initial complete assignment for *csp*
  for *i* = 1 to *max_steps* **do**
    if *current* is a solution for *csp* then return *current*
    var ← a randomly chosen, conflicted variable from VARIABLES[*csp*]
    value ← the value *v* for *var* that minimize CONFLICTS(*var,v,current,csp*)
    set *var = value* in *current*
  return *failure*

Min-conflicts example 1



Use of min-conflicts heuristic in hill-climbing.

Summary

- CSPs
  - special kind of problem: states defined by values of a fixed set of variables, goal test defined by constraints on variable values
- Backtracking = depth-first search, one variable assigned per node
- Heuristics: variable order & value selection heuristics help a lot
- Constraint propagation
  - does additional work to constrain values and detect inconsistencies
  - Works effectively when combined with heuristics
- Iterative min-conflicts is often effective in practice.
- Graph structure of CSPs determines problem complexity
  - e.g., tree structured CSPs can be solved in linear time.

Review Intro Machine Learning Chapter 18.1-18.4

- Understand Attributes, Target Variable, Error (loss) function, Classification & Regression, Hypothesis (Predictor) function
- What is Supervised Learning?
- Decision Tree Algorithm
- Entropy & Information Gain
- Tradeoff between train and test with model complexity
- Cross validation

Importance of representation

- Definition of “state” can be very important
- A good representation
  - Reveals important features
  - Hides irrelevant detail
  - Exposes useful constraints

  Most important
  - Makes frequent operations easy to do
  - Supports local inferences from local features
    - Called “soda straw” principle, or “locality” principle
    - Inference from features “through a soda straw”
  - Rapidly or efficiently computable
    - It’s nice to be fast

Terminology

- Attributes
  - Also known as features, variables, independent variables, covariates
- Target Variable
  - Also known as goal predicate, dependent variable, ...
- Classification
  - Also known as discrimination, supervised classification, ...
- Error function
  - Also known as objective function, loss function, ...

Inductive or Supervised learning

- Let x = input vector of attributes (feature vectors)
- Let f(x) = target label
  - The implicit mapping from x to f(x) is unknown to us
  - We only have training data pairs, D = {**x**, **f(x)**} available
- We want to learn a mapping from x to f(x)
  - Our hypothesis function is $h(x, \theta)$
  - $h(x, \theta) \approx f(x)$ for all training data points x
  - θ are the parameters of our predictor function h
- Examples:
  - $h(x, \theta) = \text{sign}(\theta_1 x_1 + \theta_2 x_2 + \theta_3)$ (perceptron)
  - $h(x, \theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$ (regression)
  - $h_k(x) = (x_1 \wedge x_2) \vee (x_3 \wedge \neg x_4)$

Empirical Error Functions

- $E(h) = \Sigma_x \text{ distance}[h(x, \theta) , f(x)]$

  Sum is over all training pairs in the training data D

  Examples:

  distance = squared error if h and f are real-valued (regression)

  distance = delta-function if h and f are categorical (classification)

  In learning, we get to choose

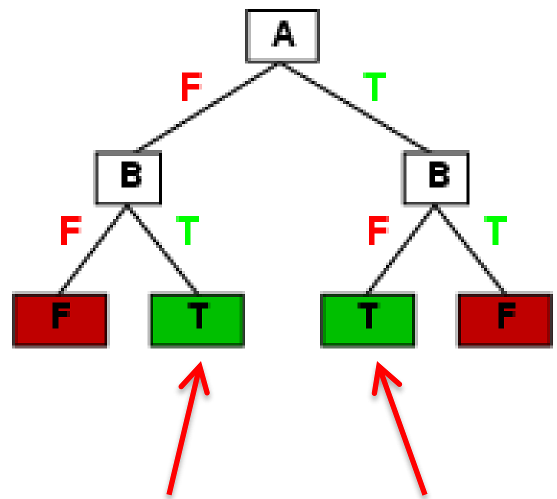
  1. what class of functions h(..) we want to learn
     – potentially a huge space! ("hypothesis space")

  2. what error function/distance we want to use
     - should be chosen to reflect real "loss" in problem
     - but often chosen for mathematical/algorithmic convenience

Decision Tree Representations

- Decision trees are fully expressive
  - Can represent any Boolean function (in DNF)
  - Every path in the tree could represent 1 row in the truth table
  - Might yield an exponentially large tree
    - Truth table is of size 2^d, where d is the number of attributes

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F



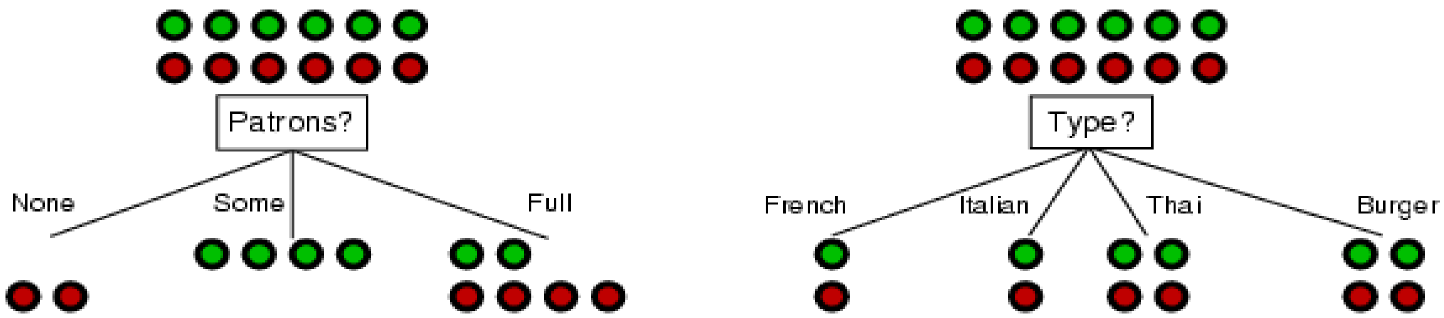
A xor B = $(\neg A \wedge B) \vee (A \wedge \neg B)$ in DNF

Pseudocode for Decision tree learning

```
function DTL(examples, attributes, default) returns a decision tree
   if examples is empty then return default
   else if all examples have the same classification then return the classification
   else if attributes is empty then return MODE(examples)
   else
      best ← CHOOSE-ATTRIBUTE(attributes, examples)
      tree ← a new decision tree with root test best
      for each value v_i of best do
         examples_i ← {elements of examples with best = v_i}
         subtree ← DTL(examples_i, attributes − best, MODE(examples))
         add a branch to tree with label v_i and subtree subtree
      return tree
```

Choosing an attribute

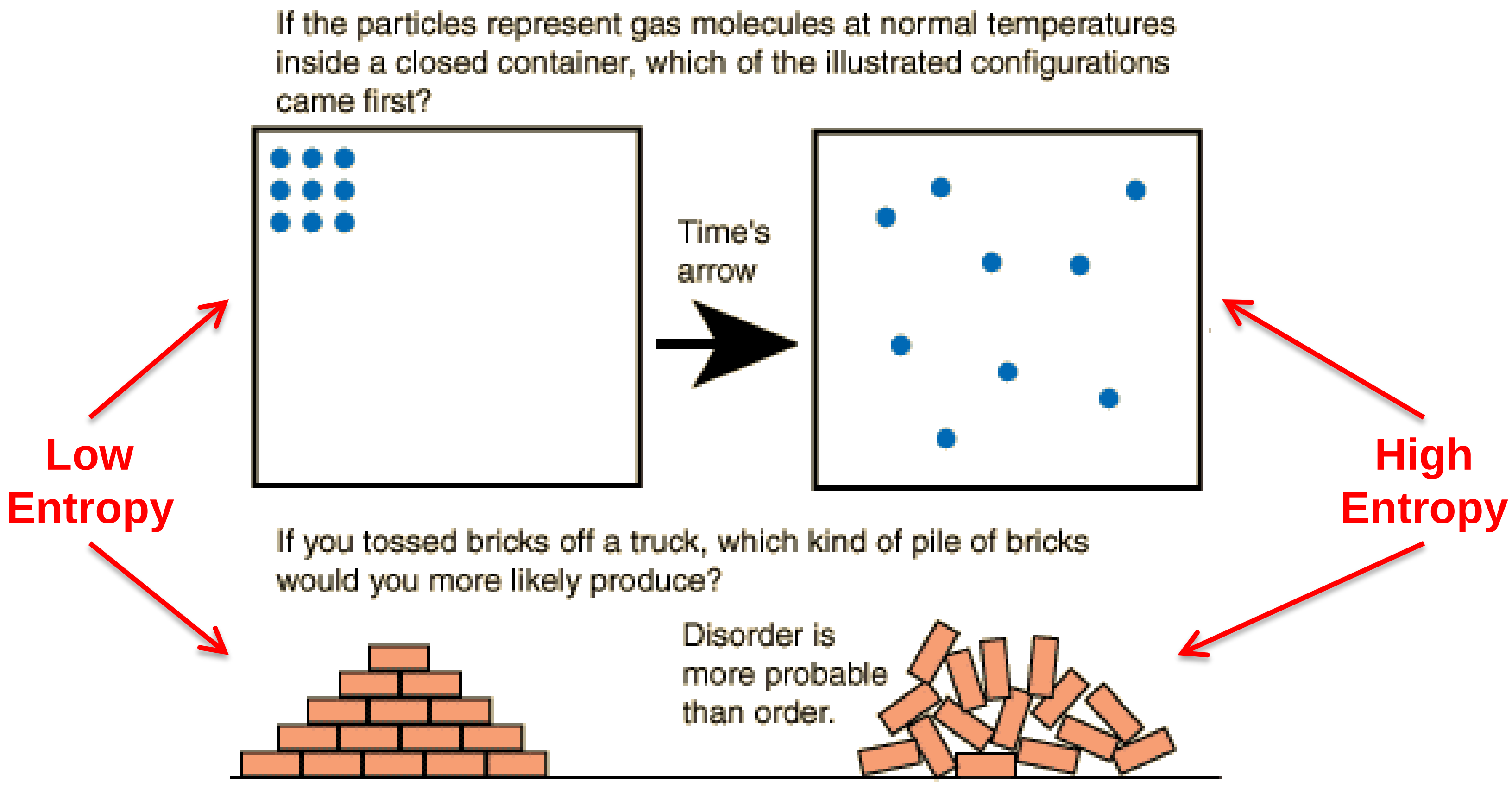
- Idea: a good attribute splits the examples into subsets that are (ideally) "all positive" or "all negative"



- *Patrons?* is a better choice
  - How can we quantify this?
  - One approach would be to use the classification error E directly (greedily)
    - Empirically it is found that this works poorly
  - **<u>Much better is to use information gain (next slides)</u>**
  - Other metrics are also used, e.g., Gini impurity, variance reduction
    - Often very similar results to information gain in practice

Entropy and Information

- **"Entropy" is a measure of randomness**
  = amount of disorder



https://www.youtube.com/watch?v=ZsY4WcQOrfk

Entropy, H(p), with only 2 outcomes

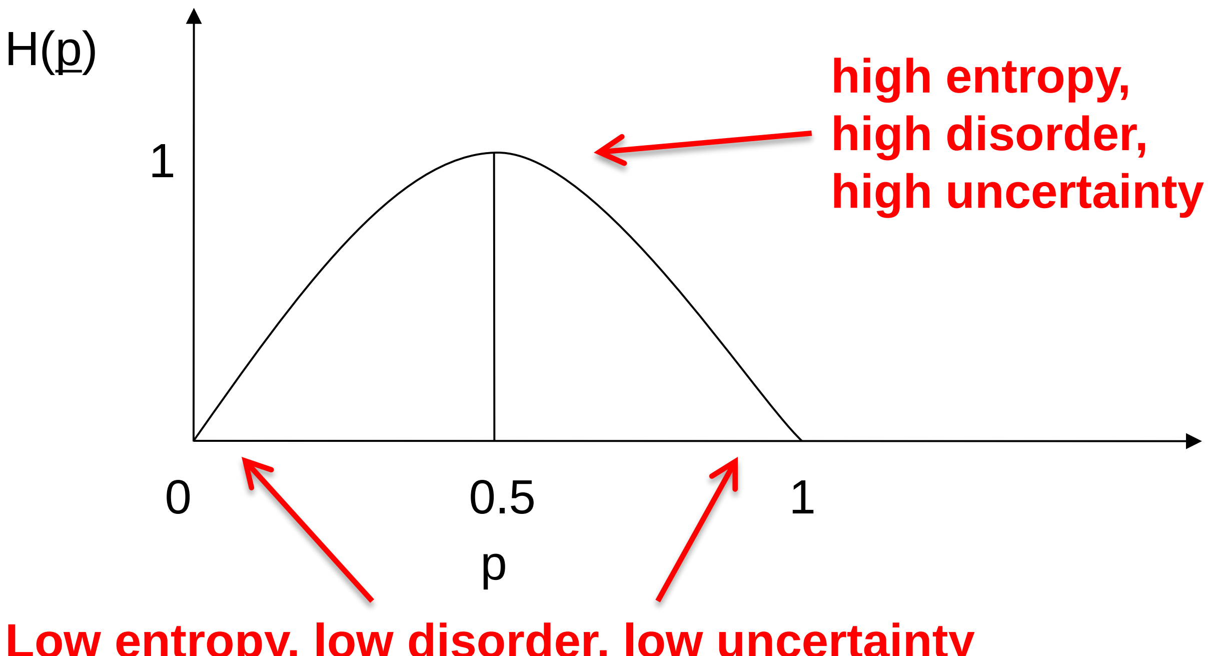
Consider 2 class problem:

p = probability of class #1,

1 – p = probability of class #2

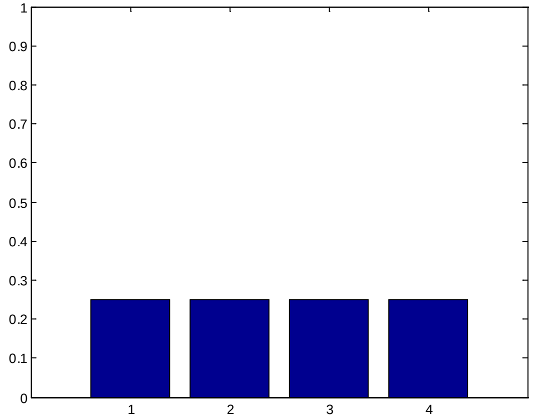
In binary case:

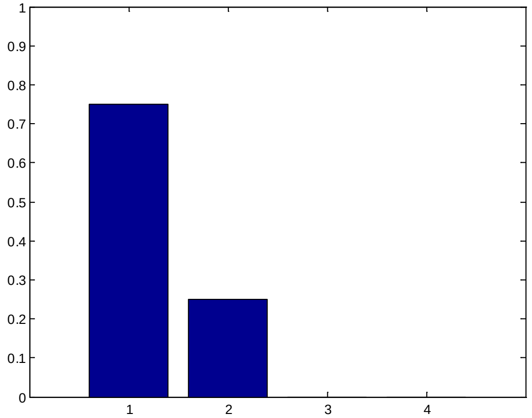
$$H(p) = -p \log p - (1-p) \log (1-p)$$

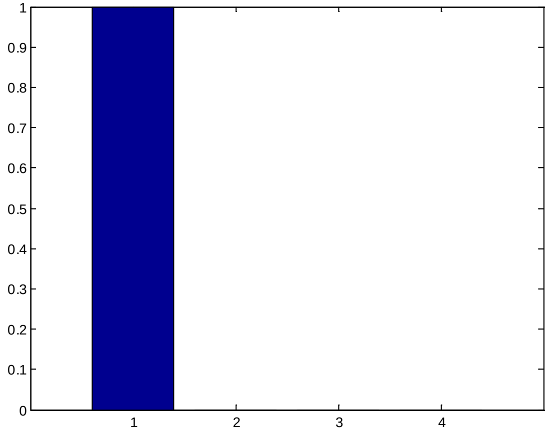


Entropy and Information

- **Entropy $H(X) = E[\log 1/P(X)] = \sum_{x \in X} P(x) \log 1/P(x)$**
  $= -\sum_{x \in X} P(x) \log P(x)$
  - Log base two, units of entropy are "bits"
  - If only two outcomes: $H(p) = -p \log(p) - (1-p) \log(1-p)$
- Examples:







H(x) = .25 log 4 + .25 log 4 + .25 log 4 + .25 log 4 = log 4 = 2 bits

Max entropy for 4 outcomes

H(x) = .75 log 4/3 + .25 log 4 = 0.8133 bits

H(x) = 1 log 1 = 0 bits

Min entropy

Information Gain

- H(P) = current entropy of class distribution P at a particular node,
  before further partitioning the data

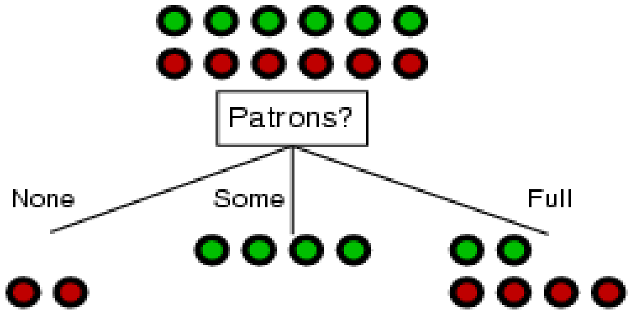
- H(P | A) = conditional entropy given attribute A
  = weighted average entropy of conditional class distribution,
  after partitioning the data according to the values in A

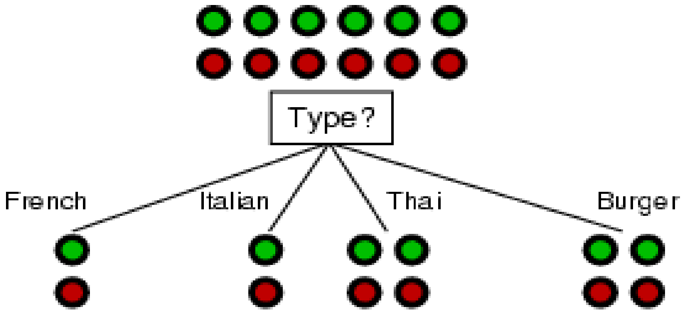
- Gain(A) = H(P) – H(P | A)
  - Sometimes written IG(A) = InformationGain(A)

- Simple rule in decision tree learning
  - **At each internal node, split on the node with the largest information gain [or equivalently, with smallest H(P|A)]**

- Note that by definition, conditional entropy can't be greater than the entropy, so Information Gain must be non-negative

Choosing an attribute

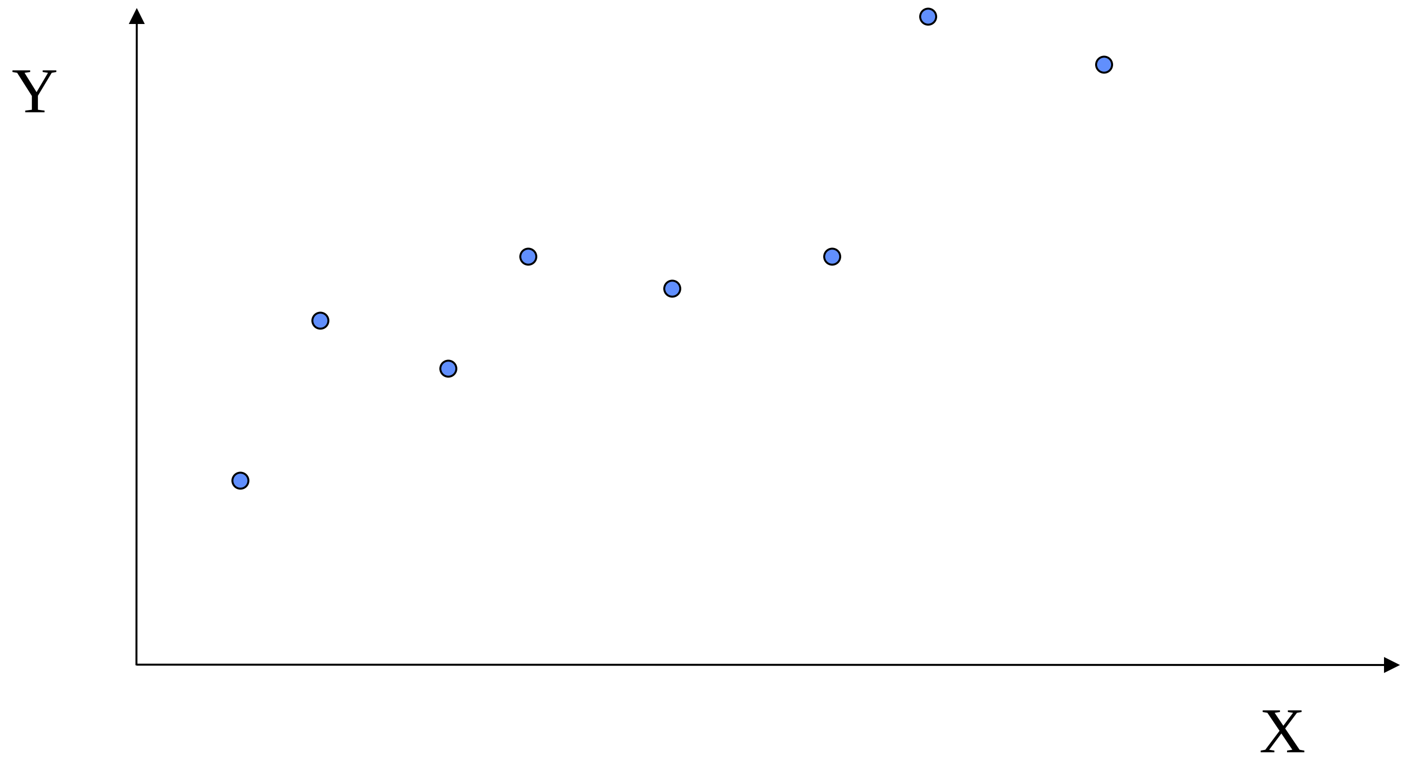




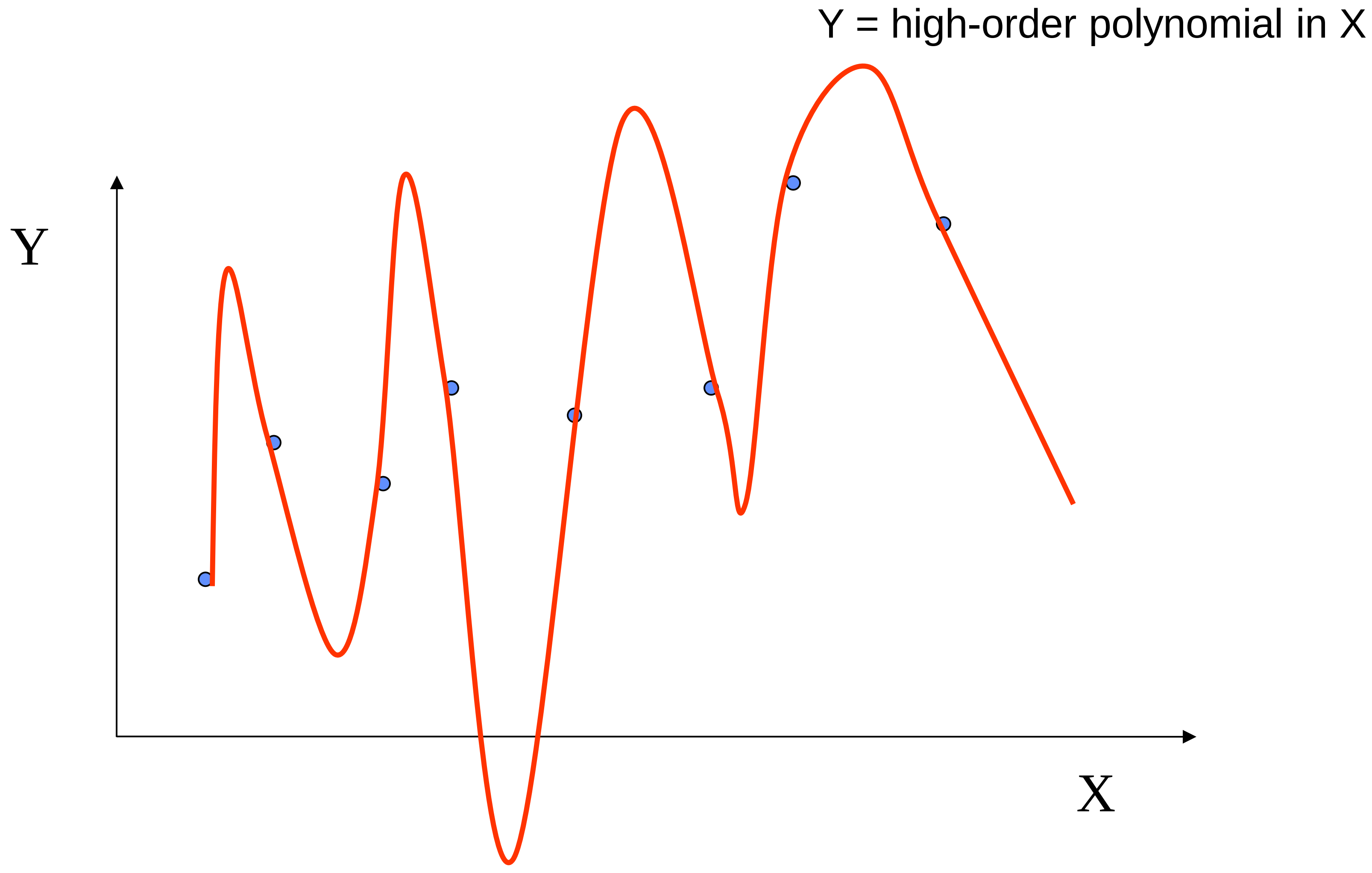
IG(Patrons) = 0.541 bits

IG(Type) = 0 bits

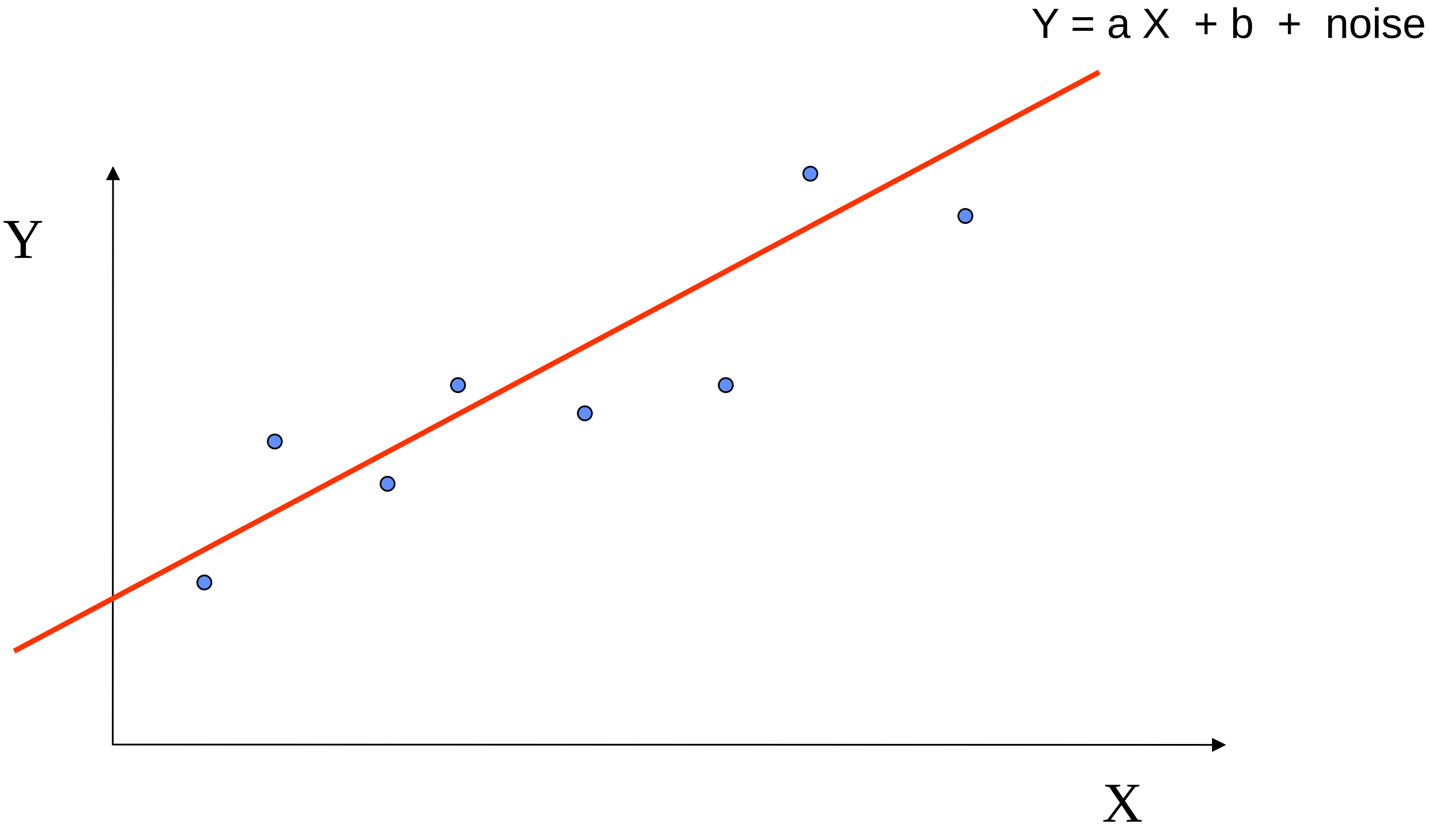
Overfitting and Underfitting



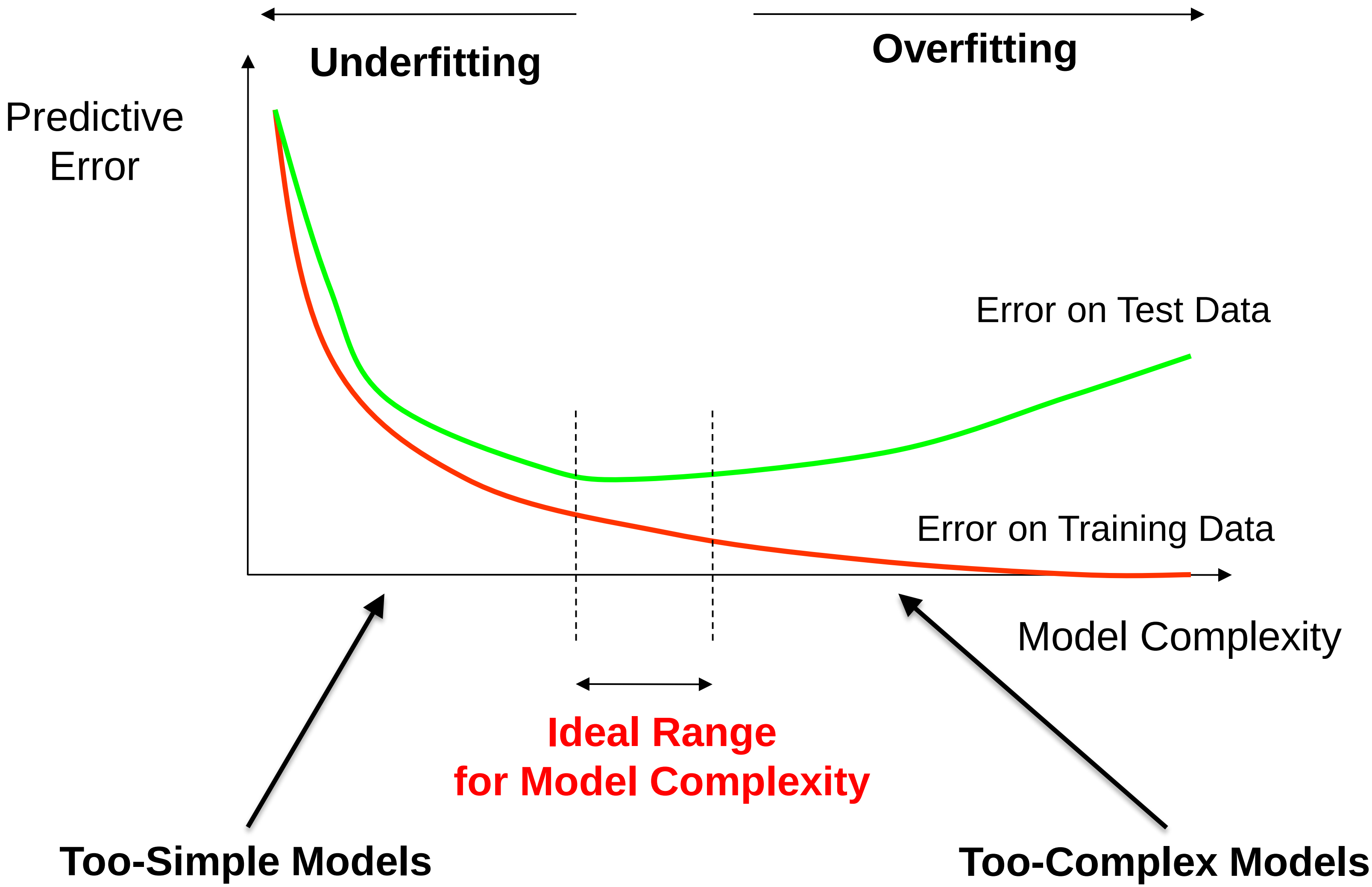
A Complex Model



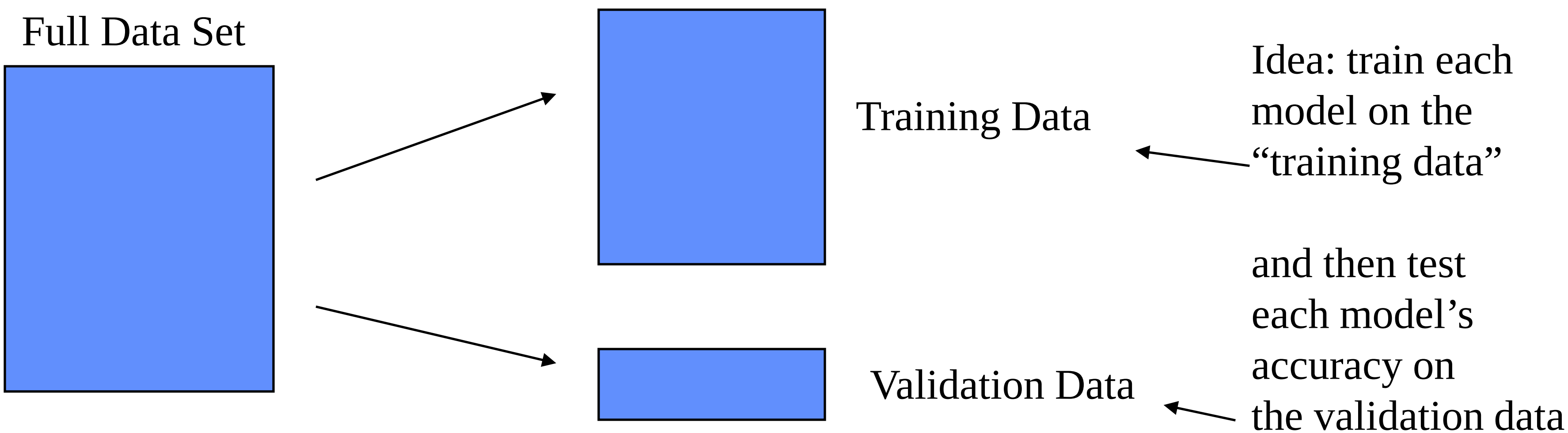
A Much Simpler Model



How Overfitting affects Prediction



Training and Validation Data

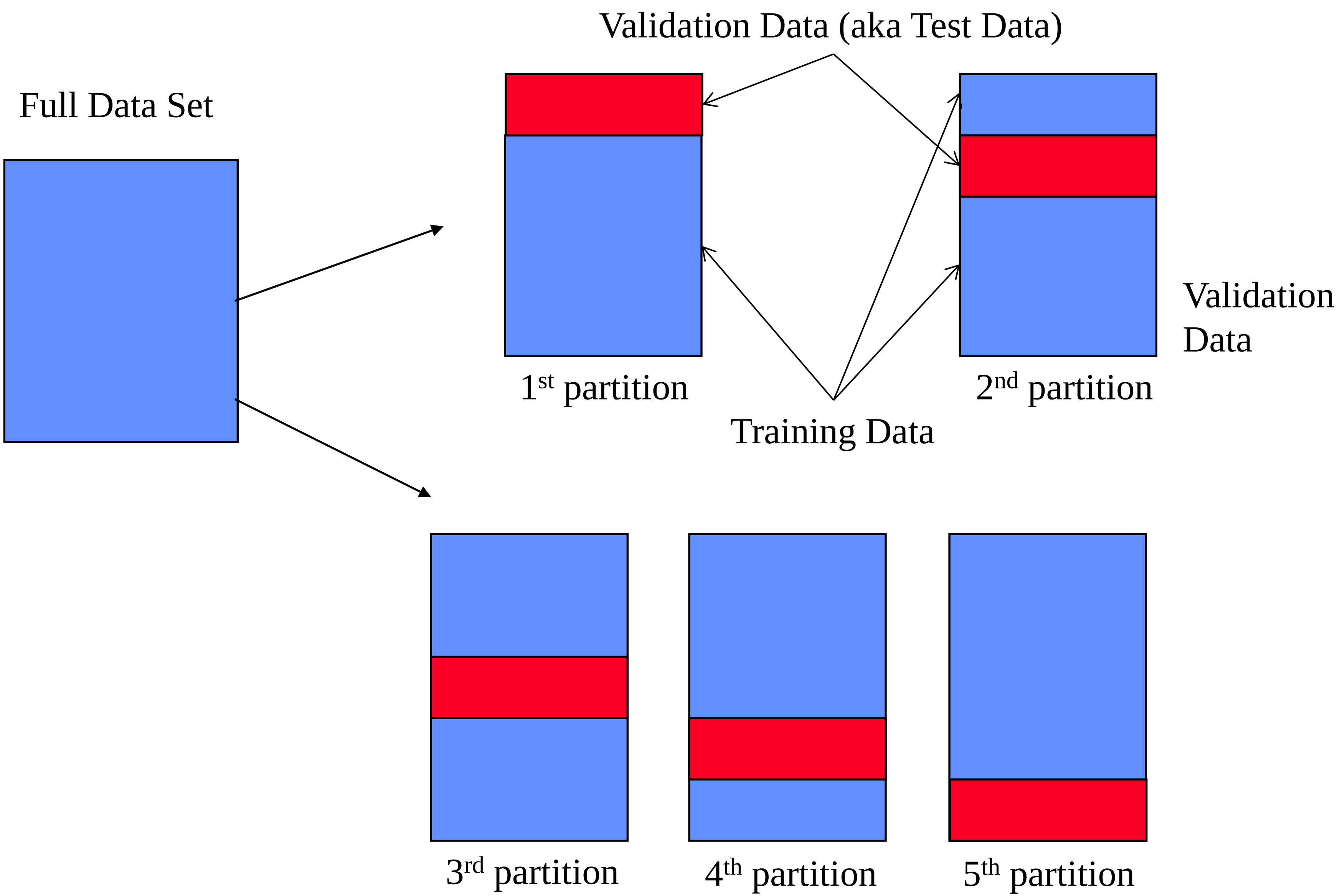


The k-fold Cross-Validation Method

- Why just choose one particular 90/10 "split" of the data?
  - In principle we could do this multiple times

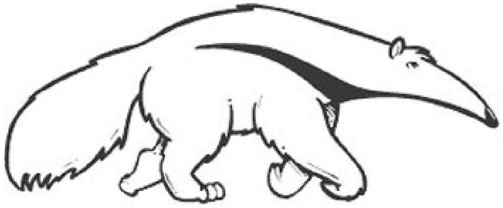
- "k-fold Cross-Validation" (e.g., k=10)
  - randomly partition our full data set into k disjoint subsets (each roughly of size n/k, n = total number of training data points)
    - for i = 1:10 (here k = 10)
      - train on 90% of data,
      - Acc(i) = accuracy on other 10%
    - end
    - Cross-Validation-Accuracy = $1/k \sum_i$ Acc(i)
  - choose the method with the highest cross-validation accuracy
  - common values for k are 5 and 10
  - Can also do "leave-one-out" where k = n

Disjoint Validation Data Sets



Final Review

CS171, Fall Quarter, 2018
Introduction to Artificial Intelligence
Prof. Richard Lathrop



Read Beforehand: R&N All Assigned Reading

BREN:ICS
INFORMATION AND COMPUTER SCIENCES

UNIVERSITY of CALIFORNIA IRVINE