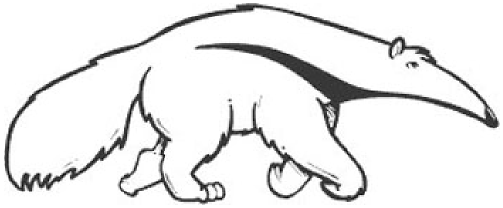
Final Review

CS171, Summer Session I, 2018
Introduction to Artificial Intelligence
Prof. Richard Lathrop



Read Beforehand: R&N All Assigned Reading

CS-171 Final Review

- **Local Search**
  - (4.1-4.2, 4.6; Optional 4.3-4.5)
- **Constraint Satisfaction Problems**
  - (6.1-6.4, except 6.3.3)
- **Machine Learning**
  - (18.1-18.12; 20.2.2)
- Questions on any topic
- Pre-mid-term material if time and class interest
- Please review your quizzes, mid-term, & old tests
  - At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; <u>the goal state itself is the solution</u>
  - Local search: widely used for <u>*very big*</u> problems
  - Returns good but <u>*not optimal*</u> solutions
  - <u>*Usually very slow*</u>, but can yield good solutions if you wait

- State space = set of "complete" configurations
- Find a complete configuration satisfying constraints
  - Examples: n-Queens, VLSI layout, airline flight schedules

- Local search algorithms
  - Keep a single "current" state, or small set of states
  - Iteratively try to improve it / them
  - Very memory efficient
    - keeps only one or a few states
    - You control how much memory you use

Random restart wrapper

- We'll use stochastic local search methods
  - Return different solution for each trial & initial state
- Almost every trial hits difficulties (see sequel)
  - Most trials will not yield a good result (sad!)
- Using many random restarts improves your chances
  - Many "shots at goal" may finally get a good one
- Restart a random initial state, *many times*
  - Report the best result found across *many* trials

Random restart wrapper

best_found ← **RandomState**()   // initialize to something

// now do repeated local search
loop do
  if (tired of doing it)
    then return *best_found*
  else
    result ← LocalSearch(**RandomState**())
    if (**Cost**(*result*) < **Cost**(*best_found*))
      // keep best result found so far
      then *best_found* ← *result*

You, as algorithm designer, write the functions named in red.

Typically, **“tired of doing it”** means that some resource limit has been exceeded, e.g., number of iterations, wall clock time, CPU time, etc.
It may also mean that result improvements are small and infrequent, e.g., less than 0.1% result improvement in the last week of run time.

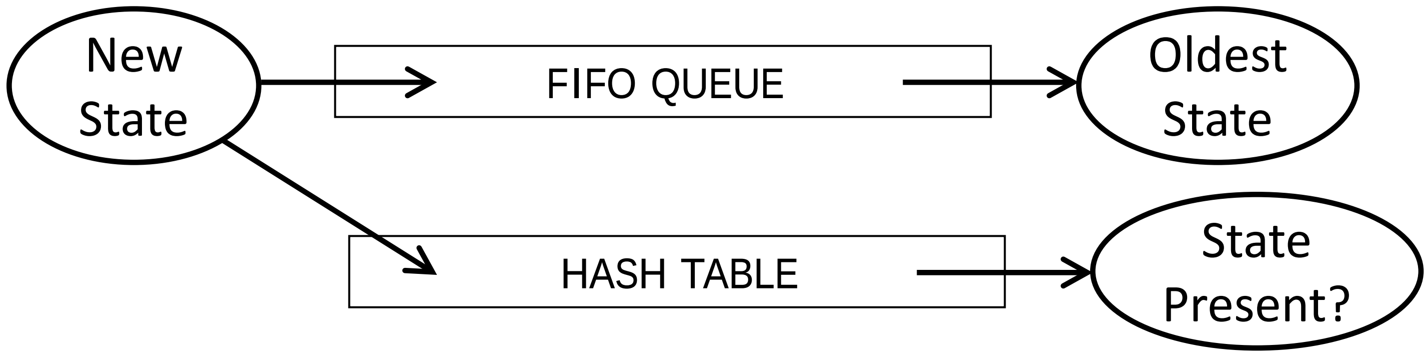
Tabu search wrapper

- Add recently visited states to a tabu-list
  - Temporarily excluded from being visited again
  - Forces solver away from explored regions
  - Less likely to get stuck in local minima (hope, in principle)

- Implemented as a hash table + FIFO queue
  - Unit time cost per step; constant memory cost
  - You control how much memory is used

- RandomRestart(TabuSearch (LocalSearch()))

Tabu search wrapper **(<u>inside</u> random restart!)**



best_found ← *current_state* ← **RandomState**()   // initialize
loop do   // now do local search
  if (tired of doing it) **then return** *best_found* **else**
    neighbor ← **MakeNeighbor**(*current_state*)
    if (*neighbor* is in *hash_table*) **then** discard *neighbor*
    else push *neighbor* onto *fifo*, pop *oldest_state*
      remove *oldest_state* from *hash_table*, insert *neighbor*
        current_state ← *neighbor*;
      if (**Cost**(*current_state*) < **Cost**(*best_found*))
        then *best_found* ← *current_state*

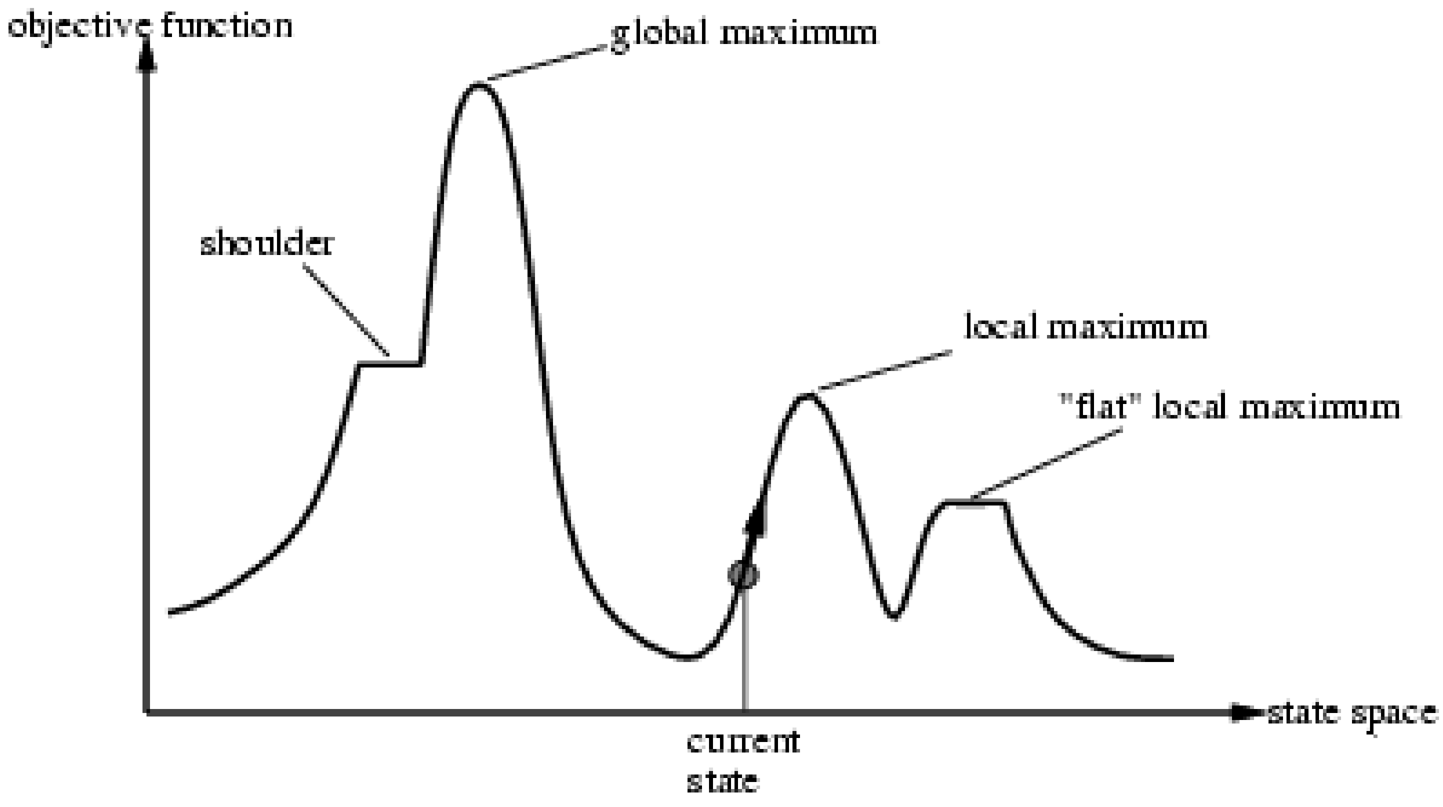
Local search algorithms

- Hill-climbing search
  - Gradient descent in continuous state spaces
  - Can use, e.g., Newton’s method to find roots
- Simulated annealing search
- Local beam search
- Genetic algorithms
- Linear Programming (for specialized problems)

Local Search Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the search space increases to high dimensionality.

- <u>Problems:</u> depending on state, can get stuck in local maxima
  - Many other problems also endanger your success!!



Local Search Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the search space increases to high dimensionality.

- Ridge problem: Every neighbor appears to be downhill
  - But the search space has an uphill!! (worse in high dimensions)

Ridge:
Fold a piece of paper and hold it tilted up at an unfavorable angle to every possible search space step. Every step leads downhill; but the ridge leads uphill.

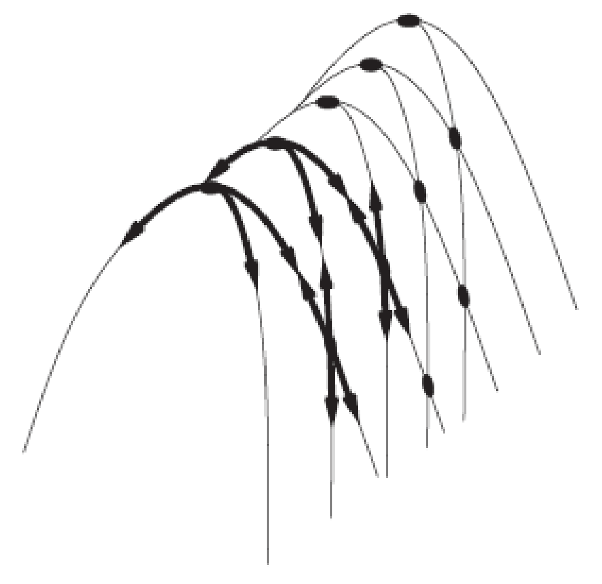


Figure 4.4 **FILES: figures/ridge.eps (Tue Nov 3 16:23:29 2009).** Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Hill-climbing search

You must shift effortlessly between maximizing value and minimizing cost

"…like trying to find the top of Mount Everest in a thick fog while suffering from amnesia"

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node
    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Equivalently:
"…a lowest-cost successor…"

Equivalently: "if **COST**[neighbor] ≥ **COST**[current] then …"

Simulated annealing (Physics!)

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency
-

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling prob. of downward steps

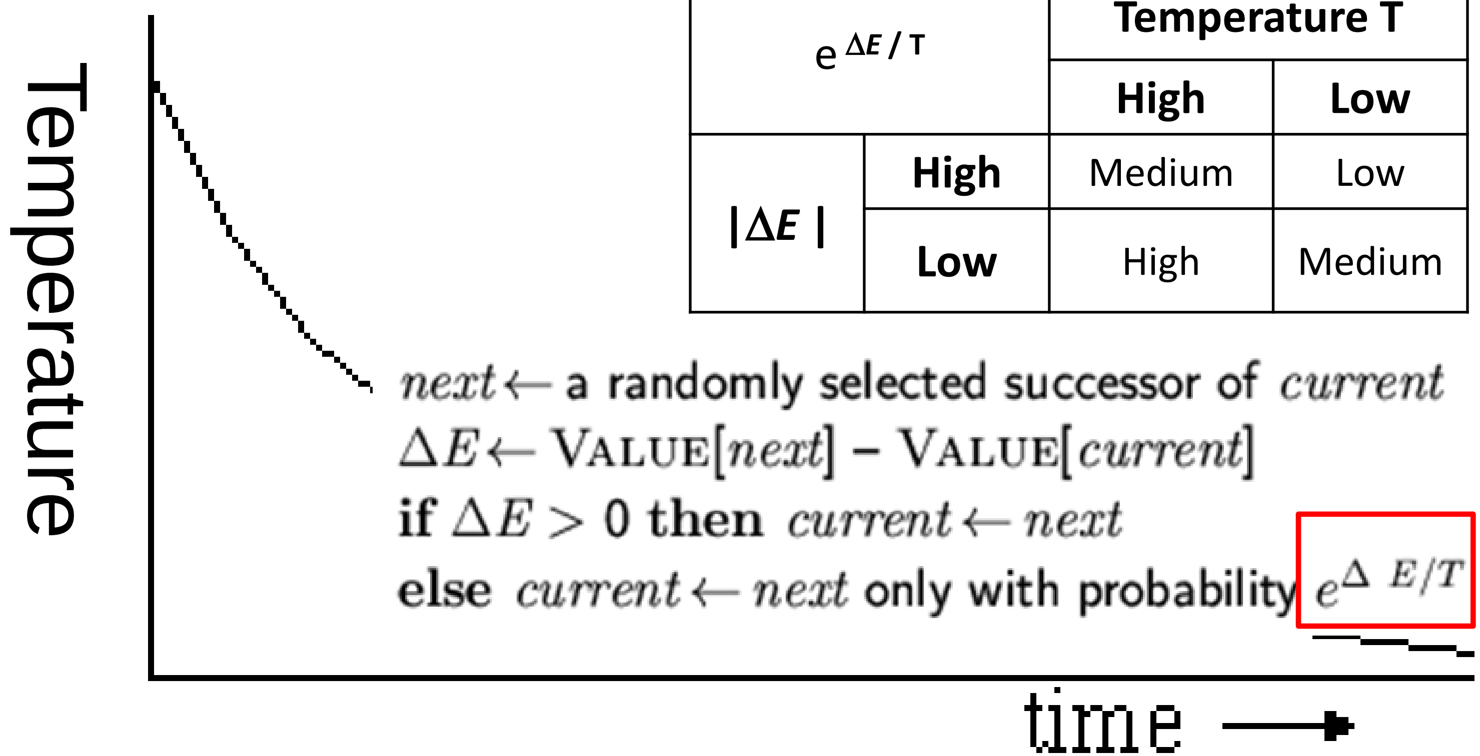
  current $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if $T = 0$ **then return** *current*
    next $\leftarrow$ a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* $\leftarrow$ *next*
    else *current* $\leftarrow$ *next* only with probability $e^{\Delta E/T}$

Improvement: Track the BestResultFoundSoFar. Here, this slide follows Fig. 4.5 of the textbook, which is simplified.

Probability(accept worse successor)

- Decreases as temperature T decreases
- Increases as |Δ E| decreases
- Sometimes, step size also decreases with T

(accept very bad moves early on; later, mainly accept “not very much worse”)

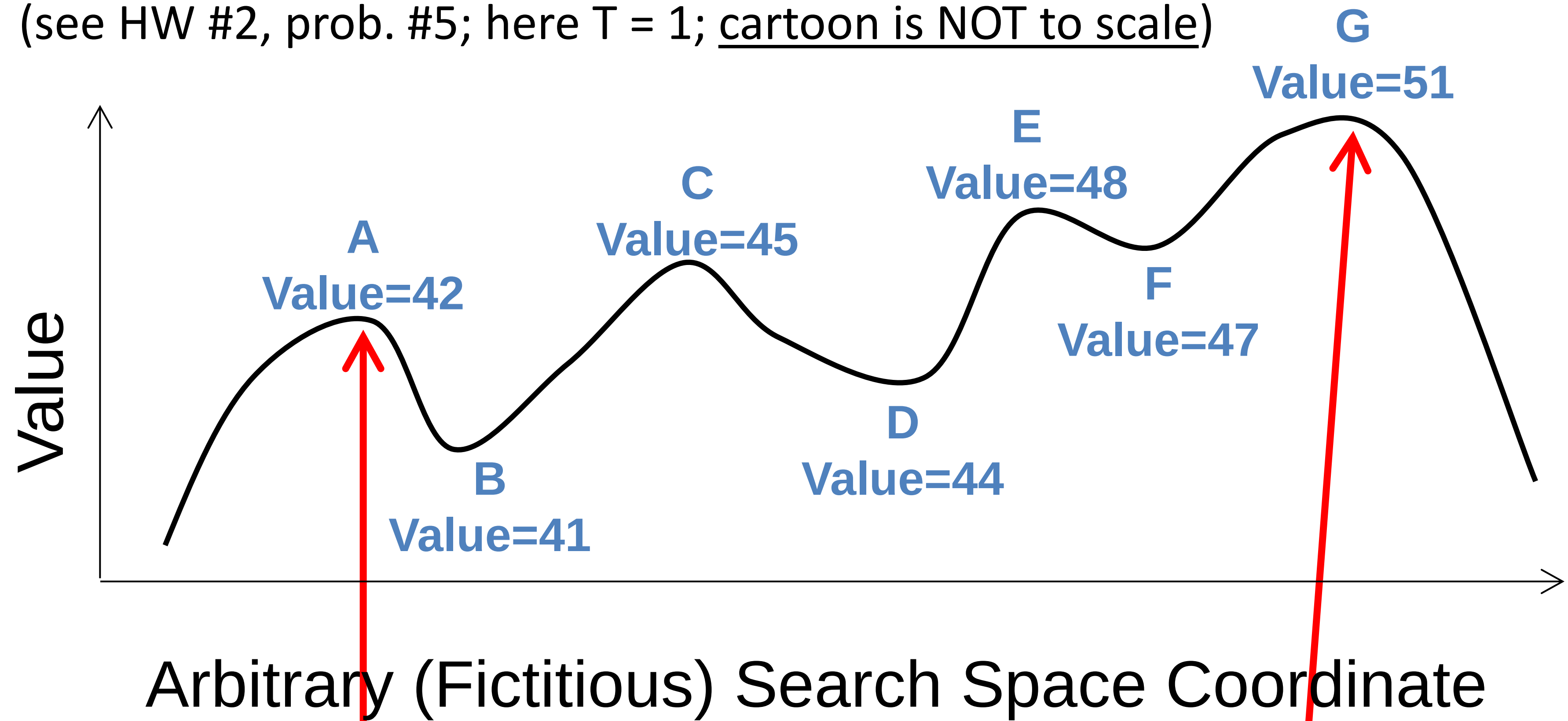


$e^{\Delta E / T}$		Temperature T: **High**	Temperature T: **Low**
$|\Delta E|$	**High**	Medium	Low
	Low	High	Medium

next ← a randomly selected successor of *current*
$\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
if $\Delta E > 0$ **then** *current* ← *next*
else *current* ← *next* only with probability $\boxed{e^{\Delta E/T}}$

Goal: “ratchet up” a bumpy slope

(see HW #2, prob. #5; here T = 1; <u>cartoon is NOT to scale</u>)

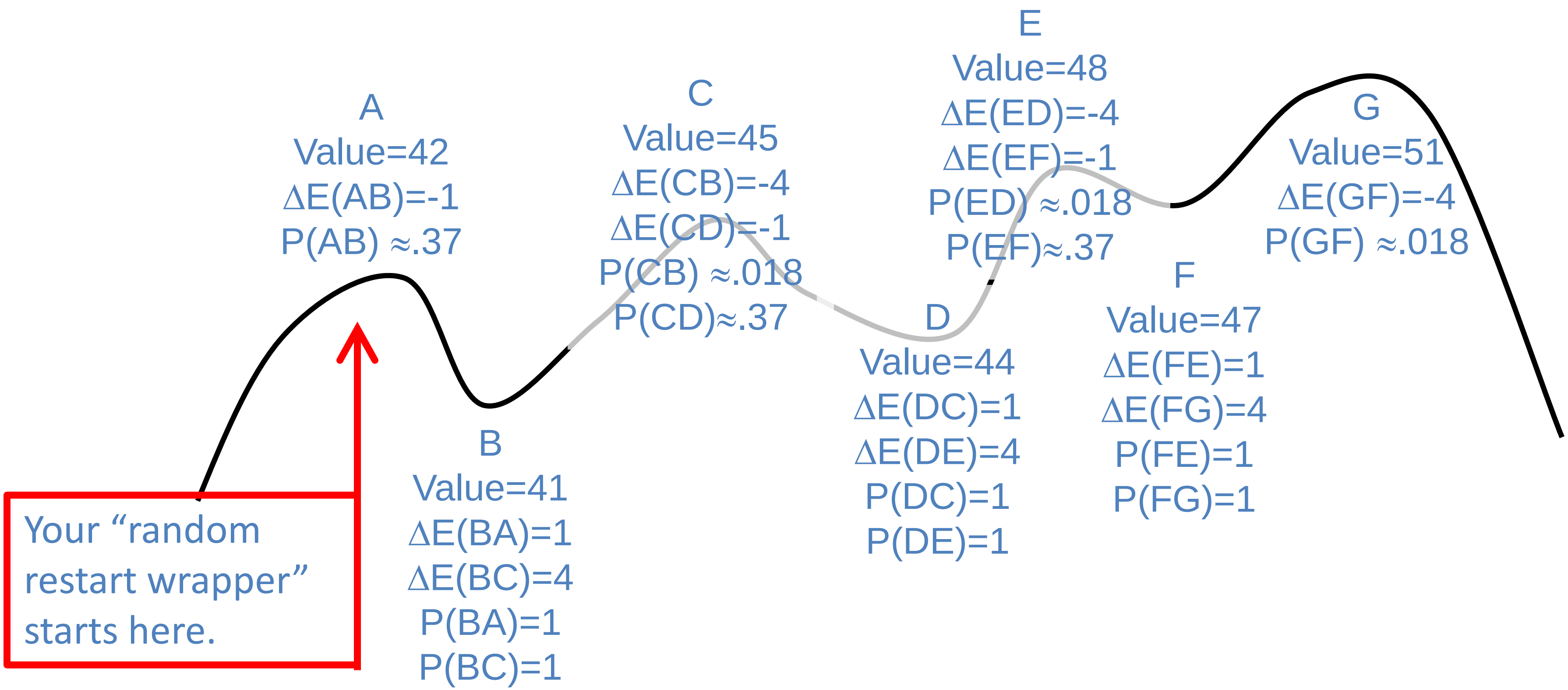


Your “random restart wrapper” starts here.

You want to get here. HOW??

This is an illustrative *cartoon*…

Goal: “ratchet up” a jagged slope



x	-1	-4
e^x	≈.37	≈.018

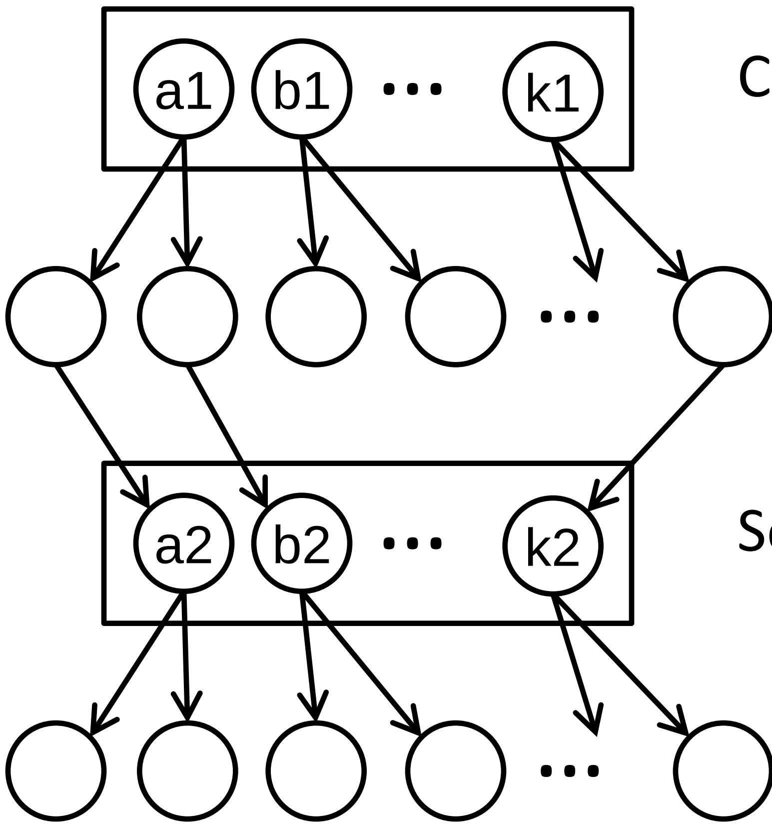
From A you will accept a move to B with P(AB) ≈.37.
From B you are equally likely to go to A or to C.
From C you are ≈20X more likely to go to D than to B.
From D you are equally likely to go to C or to E.
From E you are ≈20X more likely to go to F than to D.
From F you are equally likely to go to E or to G.
Remember best point you ever found (G or neighbor?).

This is an illustrative *cartoon*…

Local beam search

- Keep track of *k* states rather than just one
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.
- Concentrates search effort in areas believed to be fruitful
  - May lose diversity as search progresses, resulting in wasted effort

Local beam search



Create *k* random initial states

Generate their children

Select the *k* best children

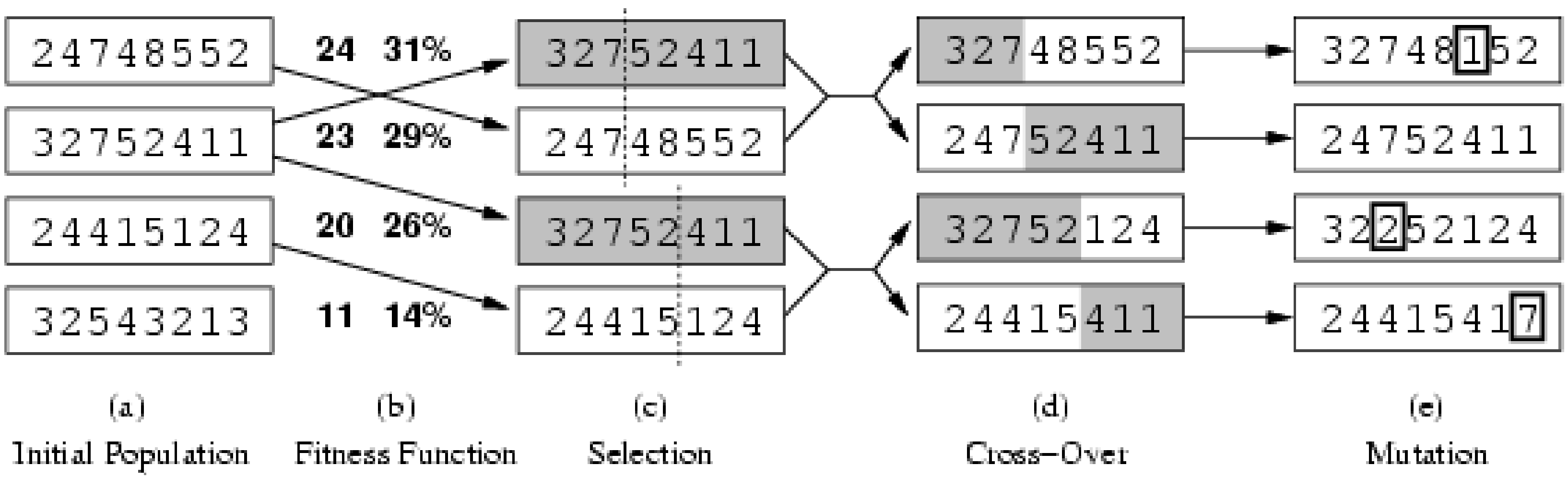
Repeat indefinitely…

Is it better than simply running *k* searches? Maybe…??

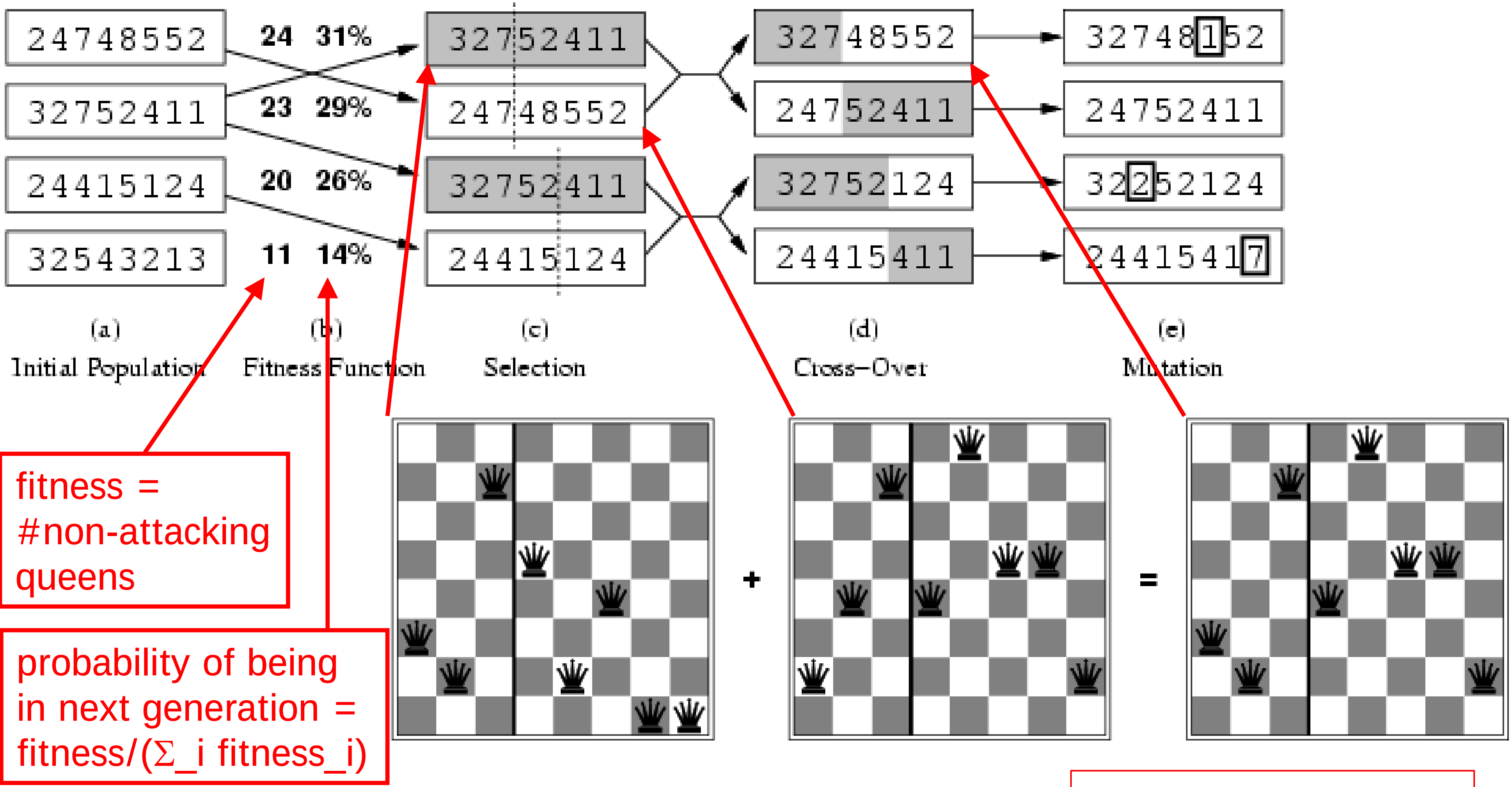
Genetic algorithms (<u>Darwin!!</u>)

- A state = a string over a finite alphabet (an **<u>individual</u>**)
  - A successor state is generated by combining two parent states
- Start with *k* randomly generated states (a **<u>population</u>**)
- **<u>Fitness</u>** function (= our heuristic objective function).
  - Higher fitness values for better states.
- **<u>Select</u>** individuals for next generation based on fitness
  - P(individual in next gen.) = individual fitness/total population fitness
- **<u>Crossover</u>** fit parents to yield next generation (**<u>offspring</u>**)
- **<u>Mutate</u>** the offspring randomly with some low probability

Genetic algorithms



- Fitness function (value): number of non-attacking pairs of queens (min = 0, max = $8 \times 7/2 = 28$)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29%; etc.



How to convert a fitness value into a probability of being in the next generation.

- Fitness function: #non-attacking queen pairs
  - min = 0, max = 8 × 7/2 = 28
- Σ_i fitness_i = 24+23+20+11 = 78
- P(child_1 in next gen.) = fitness_1/(Σ_i fitness_i) = 24/78 = 31%
- P(child_2 in next gen.) = fitness_2/(Σ_i fitness_i) = 23/78 = 29%; etc

CS-171 Final Review

- **Local Search**
  - (4.1-4.2, 4.6; Optional 4.3-4.5)
- **Constraint Satisfaction Problems**
  - (6.1-6.4, except 6.3.3)
- **Machine Learning**
  - (18.1-18.12; 20.2.2)
- Questions on any topic
- Pre-mid-term material if time and class interest
- Please review your quizzes, mid-term, & old tests
  - At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)

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

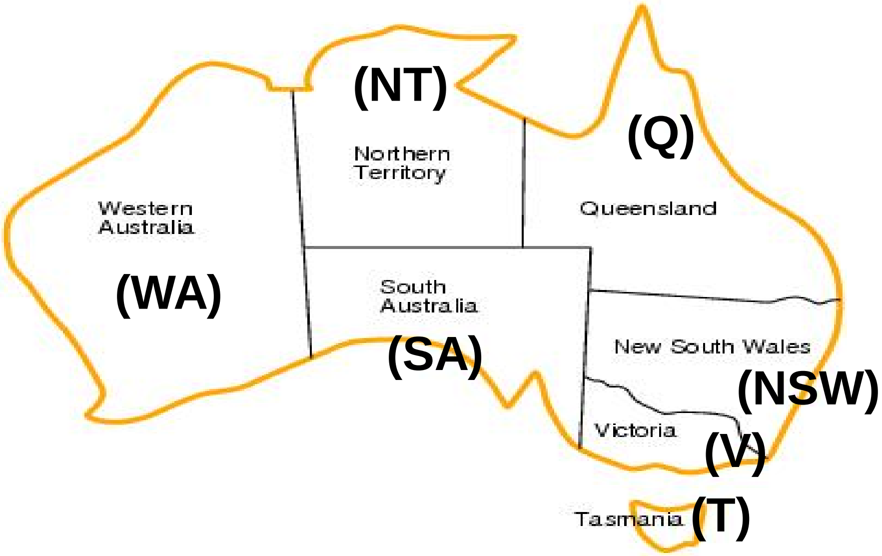
- CSP benefits
  - Standard representation pattern
  - Generic goal and successor functions
  - Generic heuristics (no domain-specific expertise required)

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.

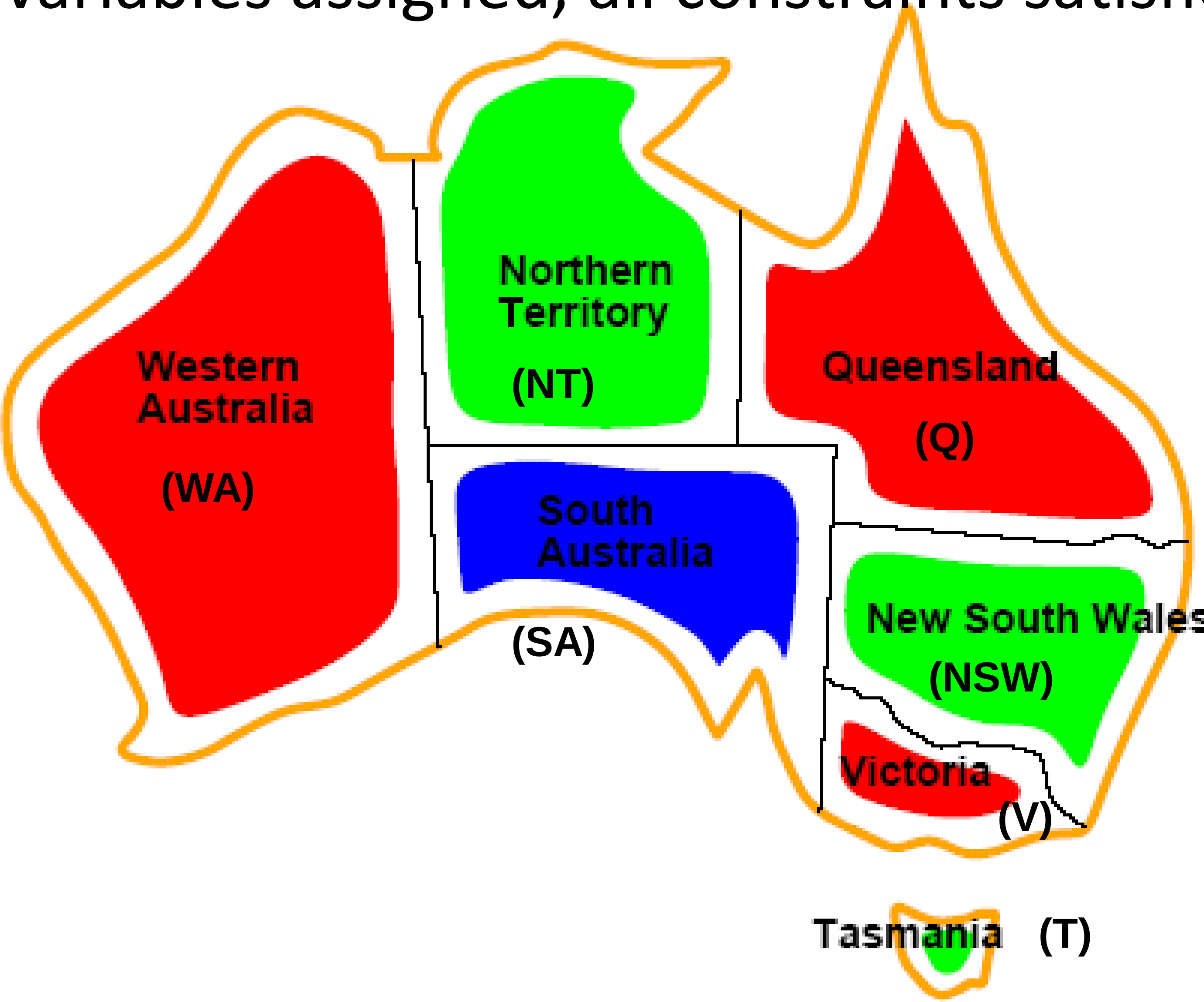
CSP example: map coloring



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

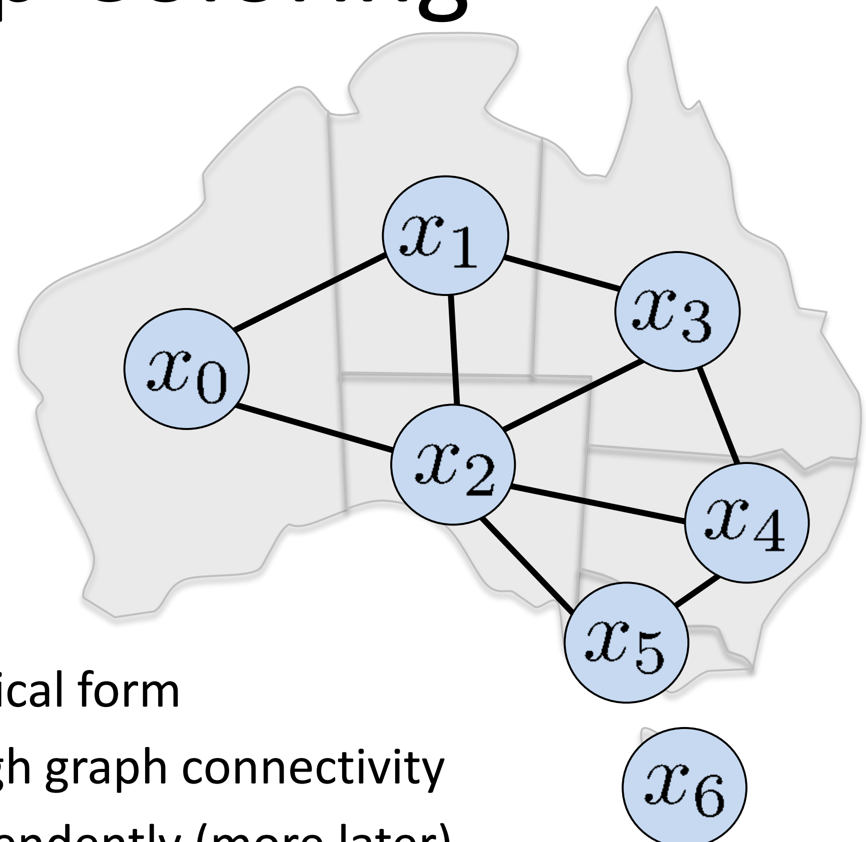
Example: Map coloring solution

All variables assigned, all constraints satisfied.

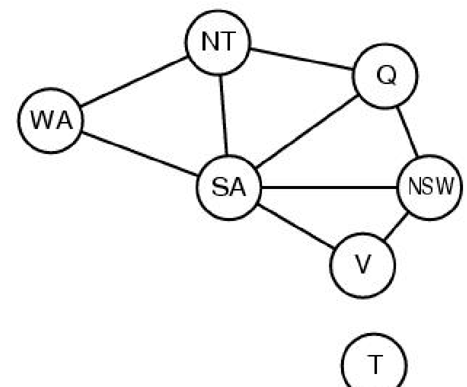


Example: Map Coloring

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



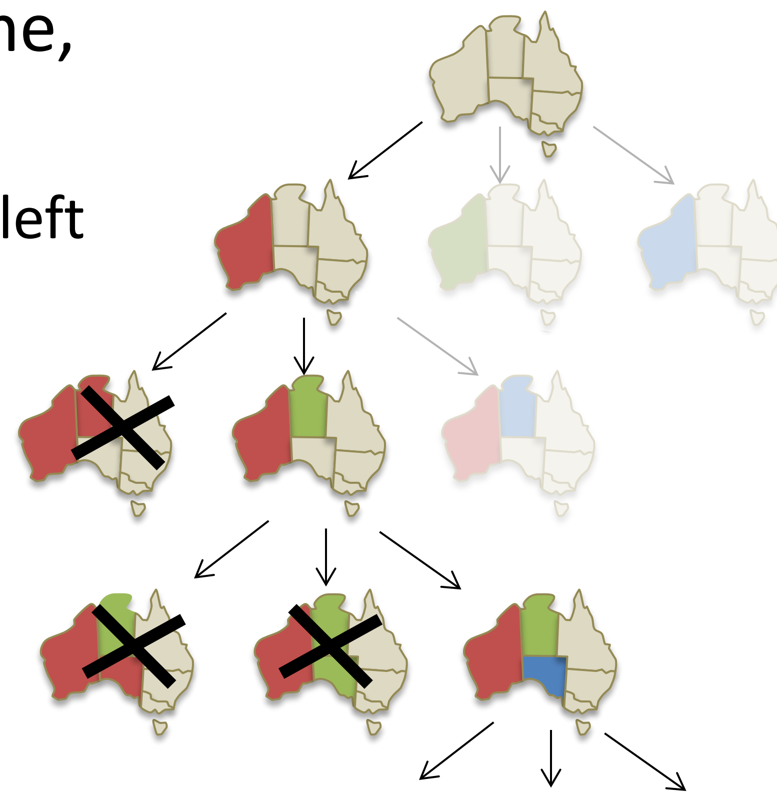
- Graphical model
  - Abstracts the problem to a canonical form
  - Can reason about problem through graph connectivity
  - Ex: Tasmania can be solved independently (more later)



- 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

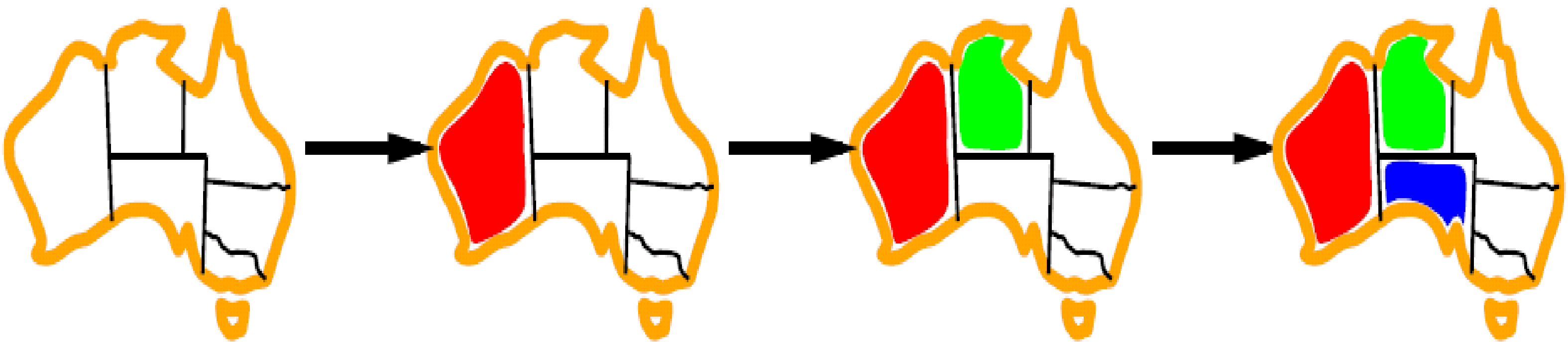


Backtracking search (Figure 6.5)

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*

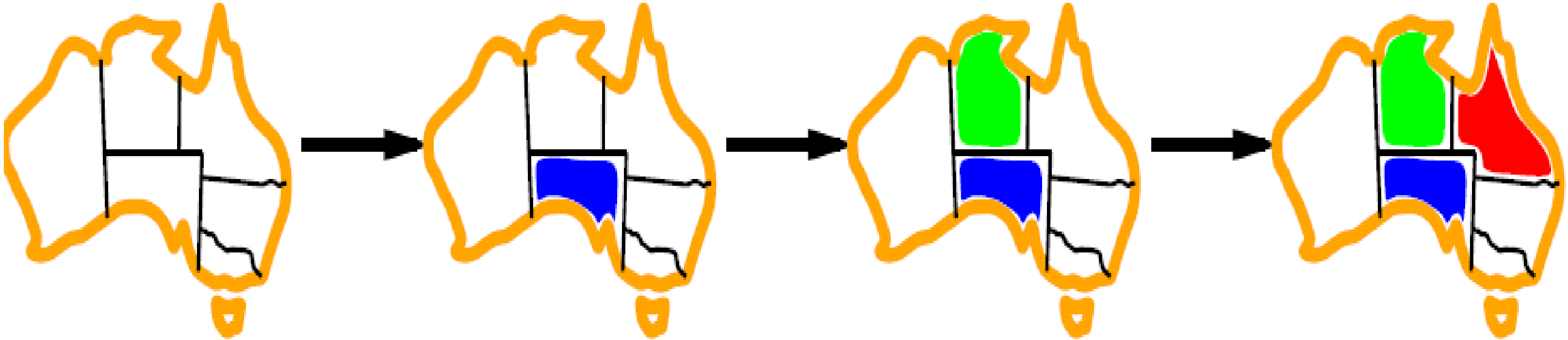
Minimum remaining values (MRV)



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



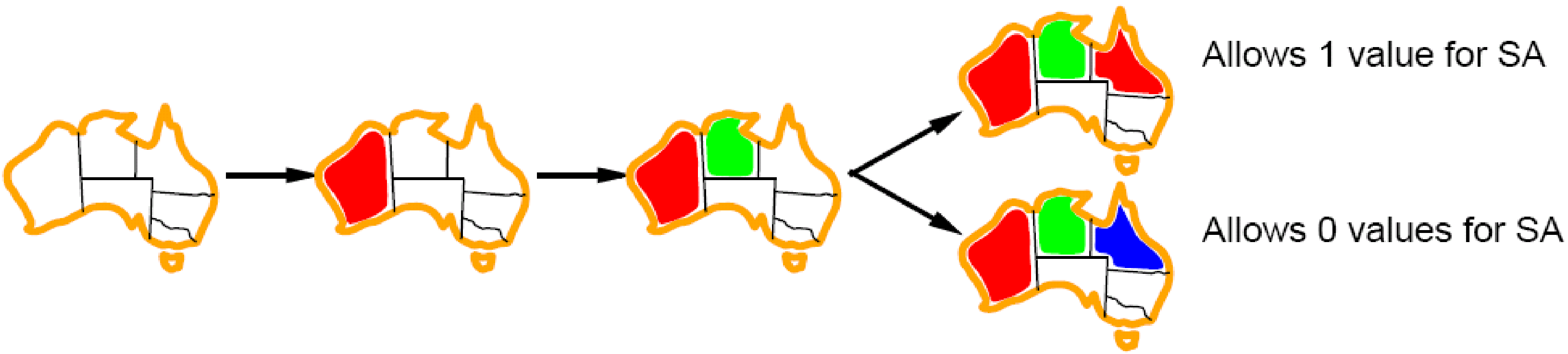
- *Heuristic Rule*: select variable that is involved in the largest number of constraints on other unassigned variables.
- Degree heuristic can be useful as a tie breaker.
- *In what order should a variable's values be tried?*

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

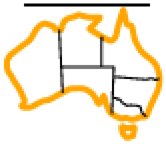
- Heuristic Rule: given a variable choose the least constraining value
  - leaves the maximum flexibility for subsequent variable assignments

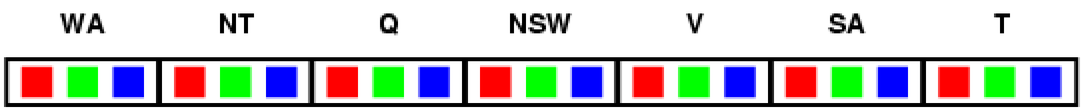
Look-ahead: Constraint propagation

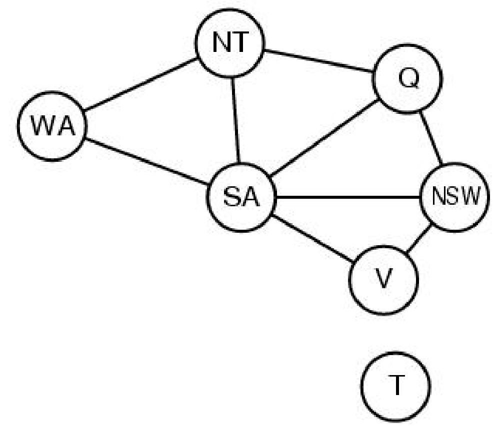
- Intuition:
  - Some domains have values that are inconsistent with the values in some other domains
  - Propagate constraints to remove inconsistent values
  - Thereby reduce future branching factors
- Forward checking
  - Check each unassigned neighbor in constraint graph
- Arc consistency (AC-3 in 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

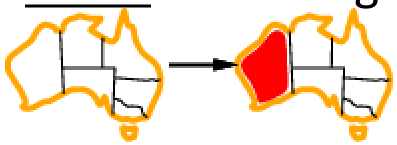


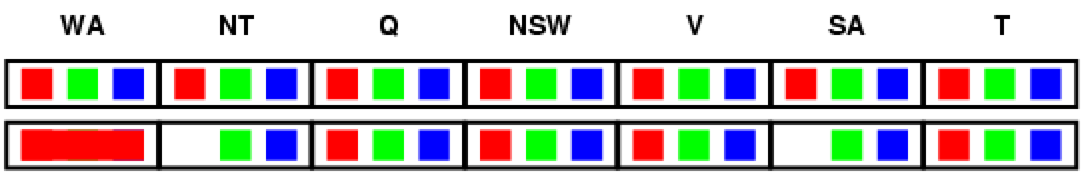


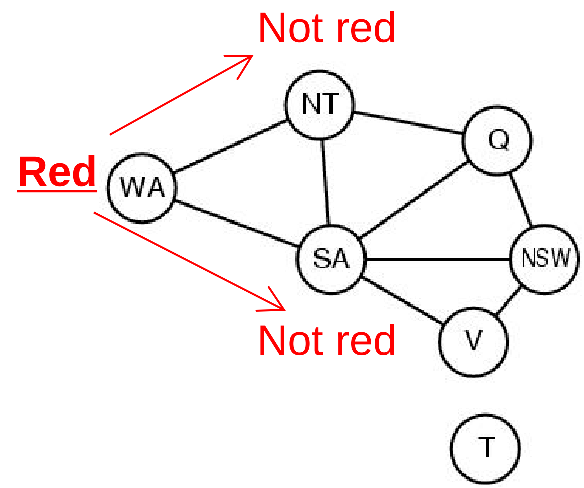


Forward checking

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







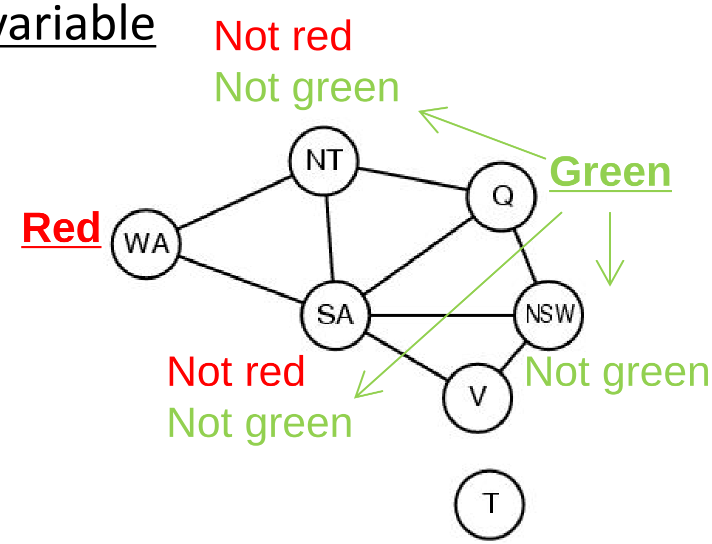
Assign {WA = red}

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>

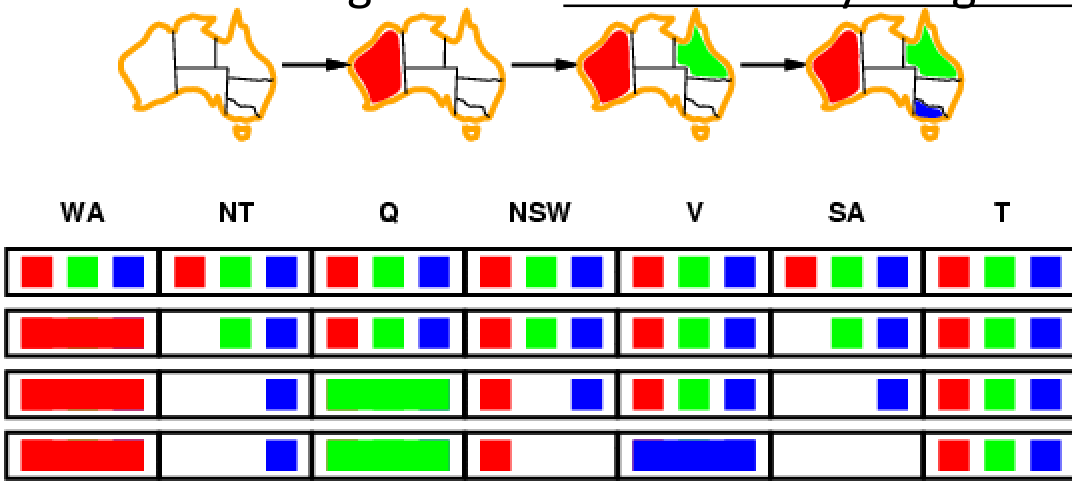


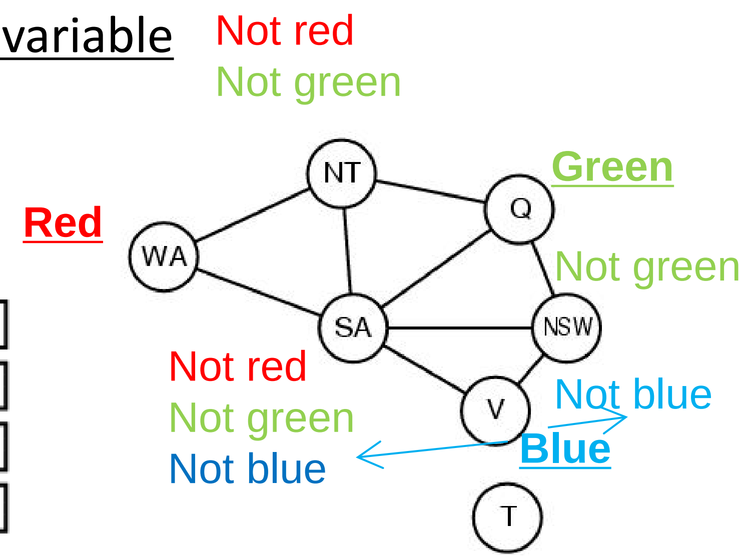
Assign {Q = green}
Effect on other variables (neighbors of Q):
- NT can no longer be green
- SA can no longer be green
- NSW can no longer be green

<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

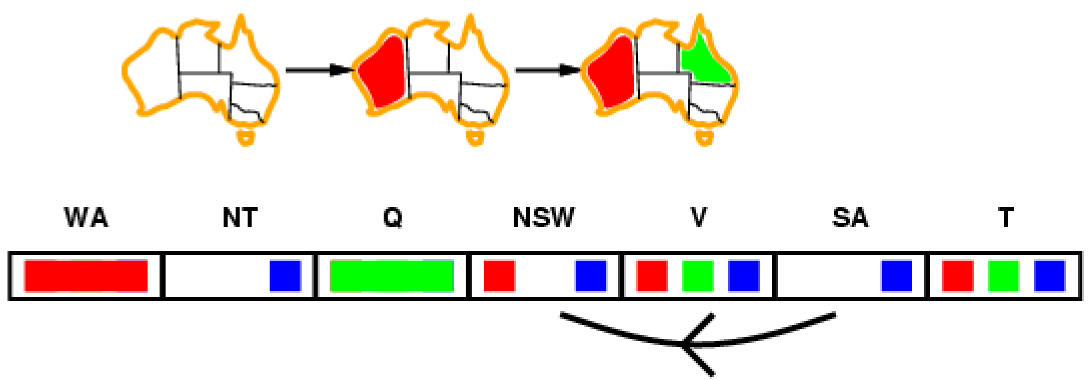
Arc consistency (AC-3) algorithm

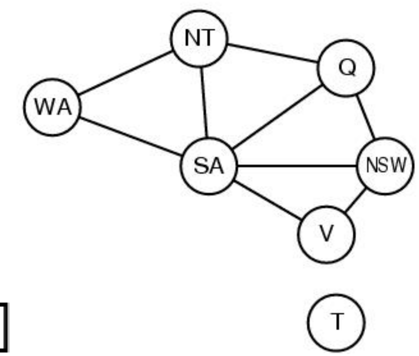
- An Arc $X \rightarrow Y$ is consistent iff for *every* value *x* of *X* there is *some* value *y* of *Y* that is consistent with *x*
- Put all arcs $X \rightarrow Y$ on a queue
  - Each undirected constraint graph arc is two directed arcs
  - Undirected X—Y becomes directed $X \rightarrow Y$ and $Y \rightarrow X$
  - $X \rightarrow Y$ 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!)*





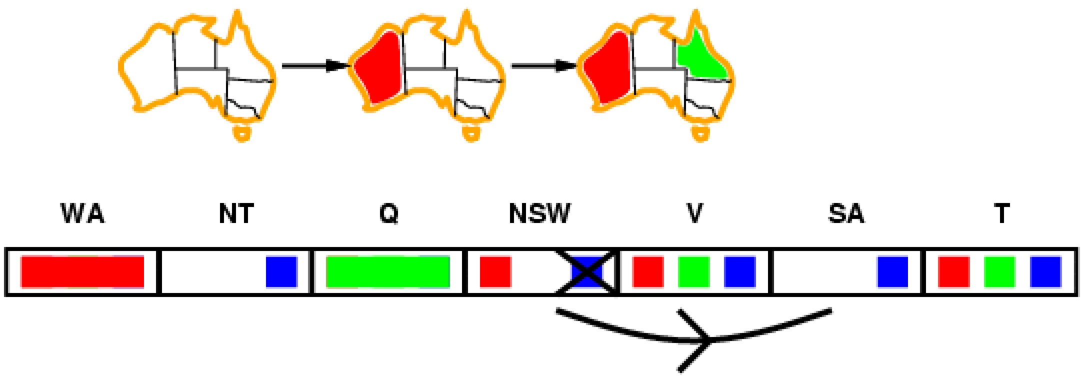
- Consider state after WA=red, Q=green
  - $SA \rightarrow NSW$ is consistent because

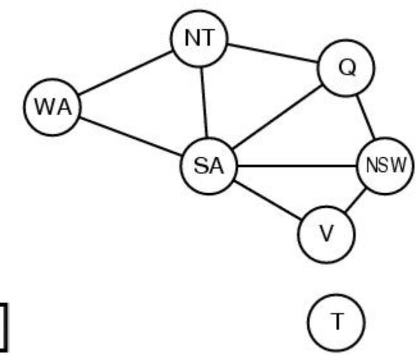
    SA = blue and NSW = red satisfies all constraints on SA and NSW

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

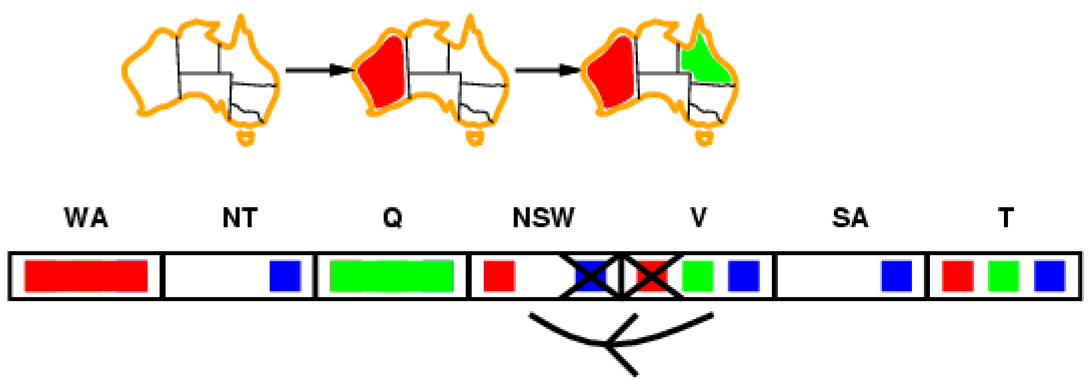
    NSW = blue and SA = ??? => NSW = blue can be pruned

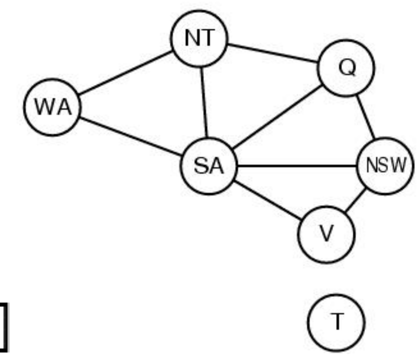
    No current domain value for SA is consistent

Arc consistency

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

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



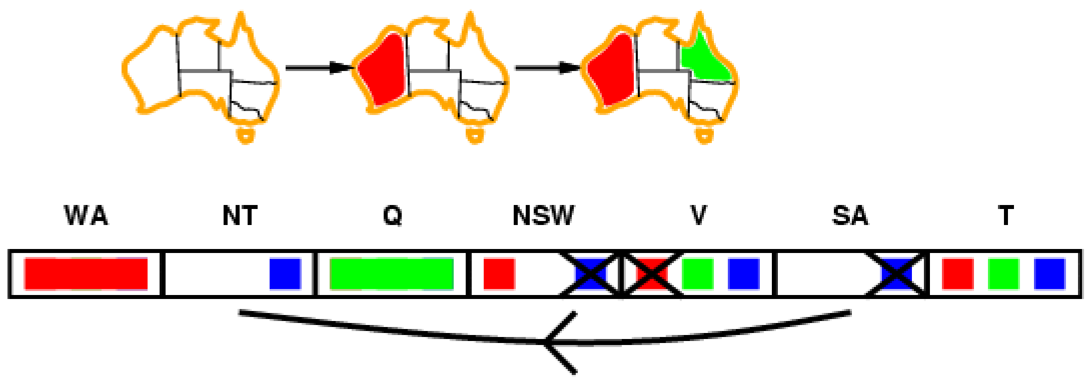


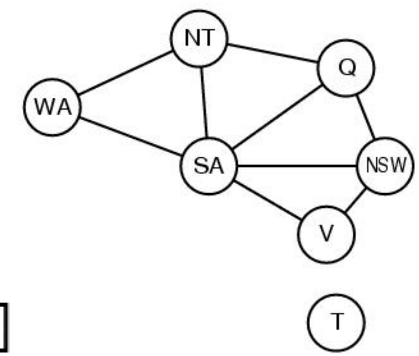
- **Enforce arc consistency:**
  – arc can be made consistent by removing blue from NSW
- **Continue to propagate constraints:**
  – Check V $\rightarrow$ NSW : not consistent for V = red; remove red from V

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

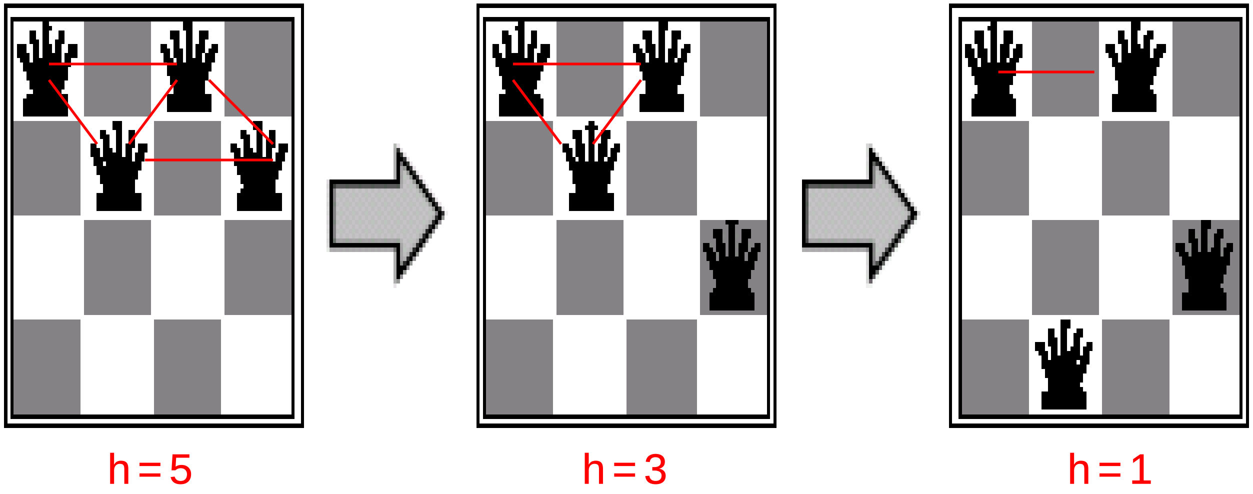
- **Variable selection:** randomly select any conflicted variable
- **Value selection:** <u>*min-conflicts heuristic*</u>
  - Select new value that results in a minimum number of conflicts with the other variables

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.

CS-171 Final Review

- **Local Search**
  - (4.1-4.2, 4.6; Optional 4.3-4.5)
- **Constraint Satisfaction Problems**
  - (6.1-6.4, except 6.3.3)
- **Machine Learning**
  - (18.1-18.12; 20.2.2)
- Questions on any topic
- Pre-mid-term material if time and class interest
- Please review your quizzes, mid-term, & old tests
  - At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)

Importance of representation

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

Most important

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, θ)
  - h(x, θ) ≈ 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

- $E(h) = \Sigma_x$ 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

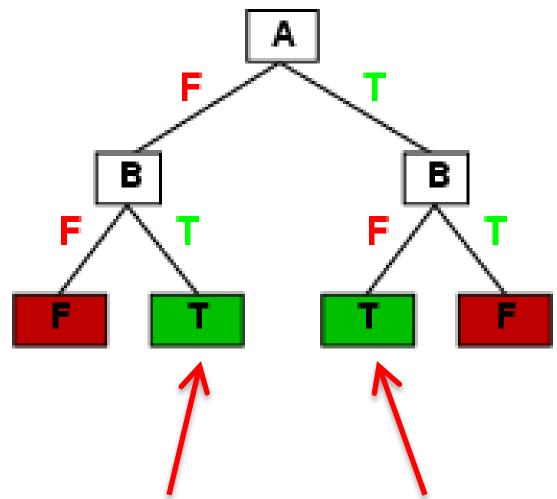
  1. what class of functions h(..) we want to learn
     – potentially a huge space! (“hypothesis space”)

  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

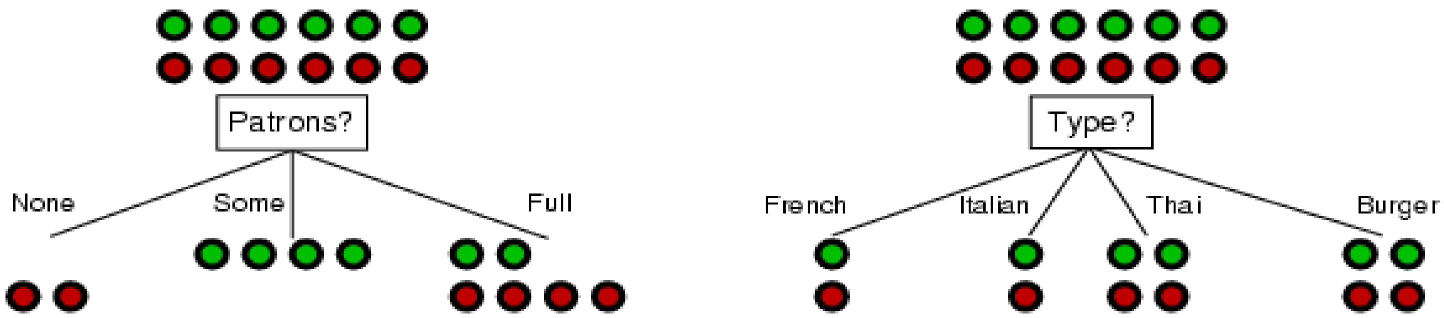
Pseudocode for Decision tree learning

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

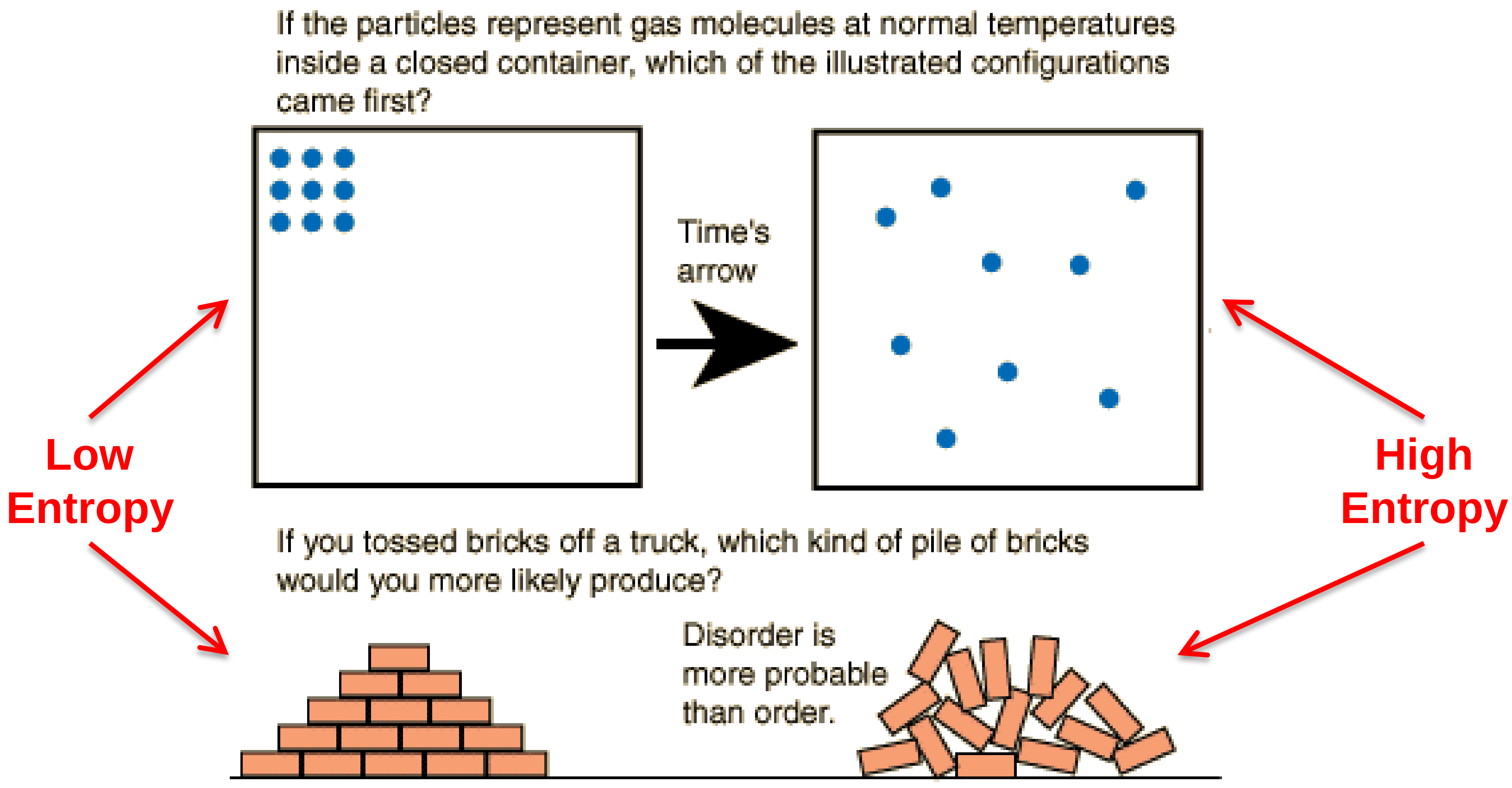
- Idea: a good attribute splits the examples into subsets that are (ideally) "all positive" or "all negative"



- *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

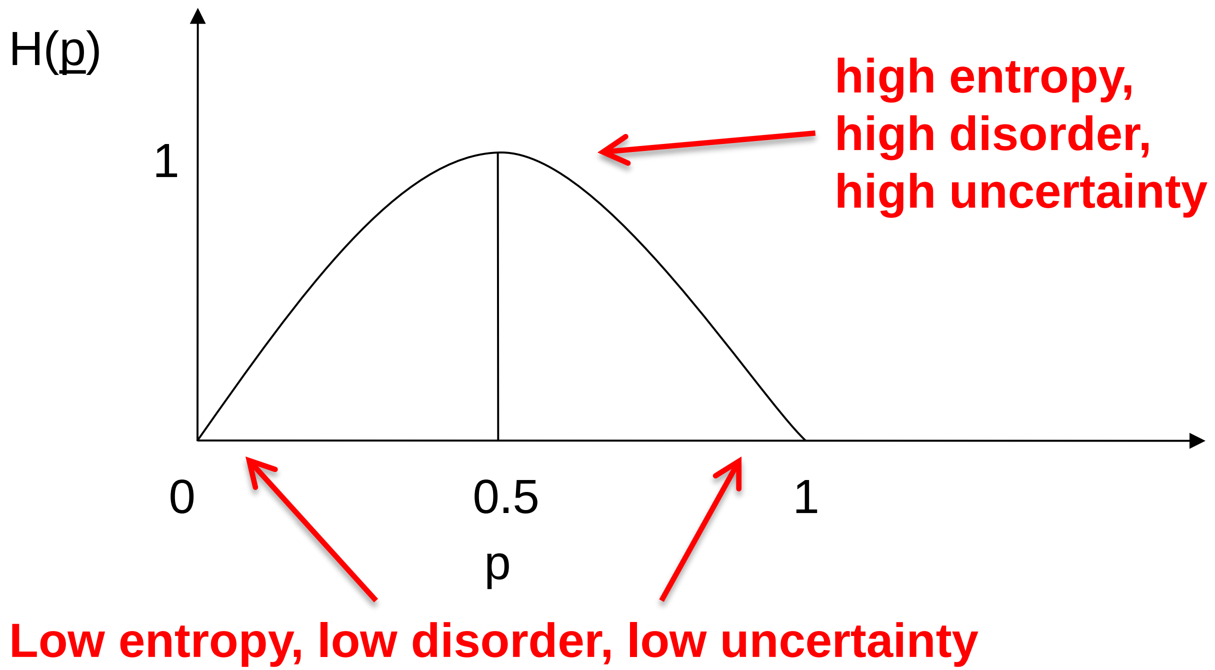
Consider 2 class problem:

p = probability of class #1,

1 – p = probability of class #2

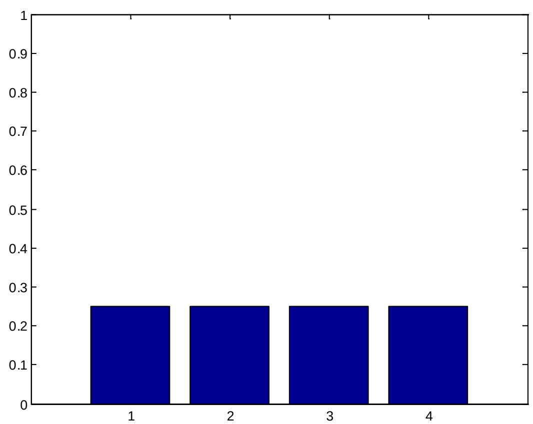
In binary case:

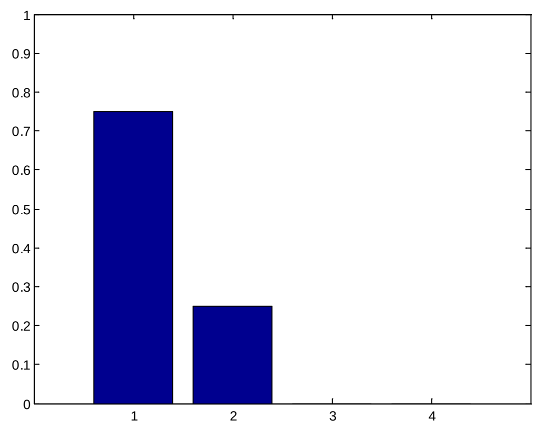
$$H(p) = -\,p \log p \;-\; (1-p) \log (1-p)$$

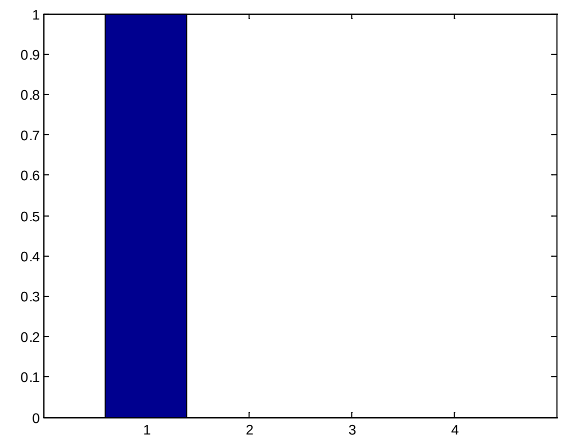


Entropy and Information

- **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

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

H(x) = 1 log 1 = 0 bits

Max entropy for 4 outcomes

Min entropy

Information Gain

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

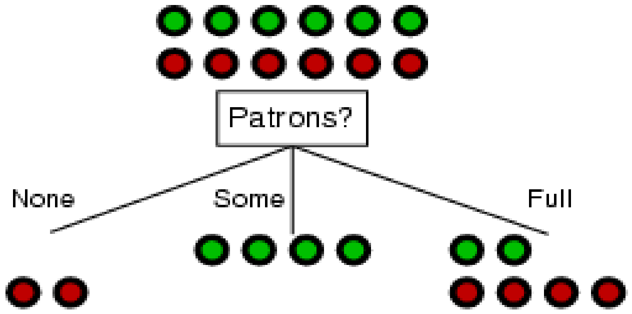
- H(P | A) = conditional entropy given attribute A
  = weighted average entropy of conditional class distribution,
  after partitioning the data according to the values in A

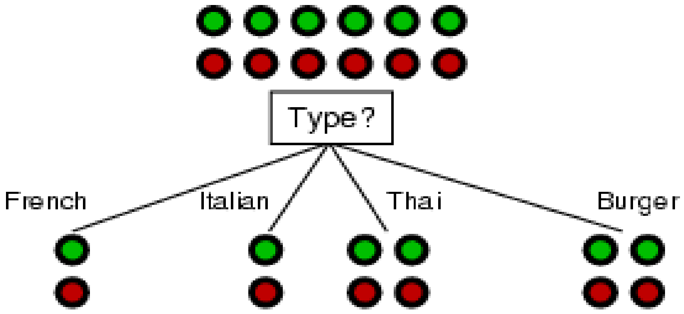
- Gain(A) = H(P) – H(P | A)
  - Sometimes written IG(A) = InformationGain(A)

- 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

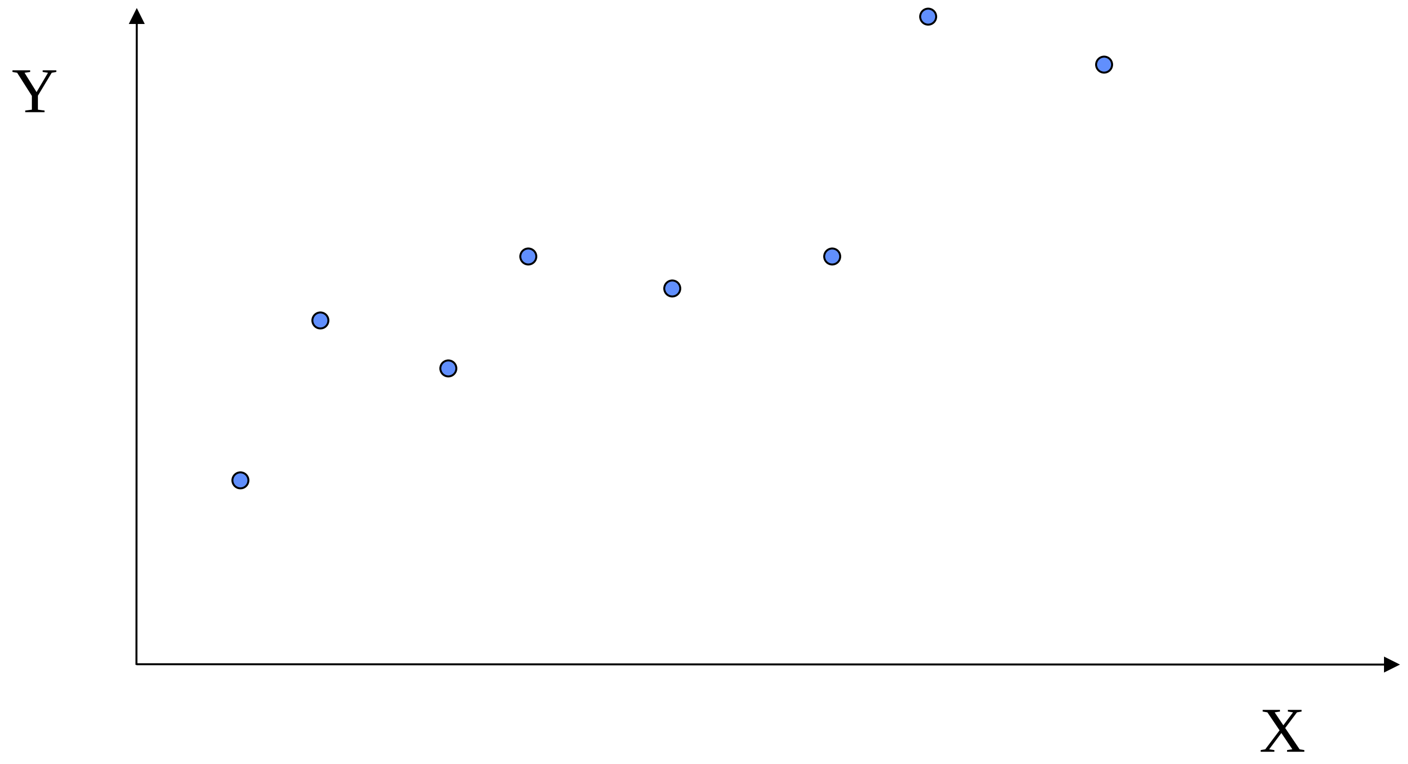




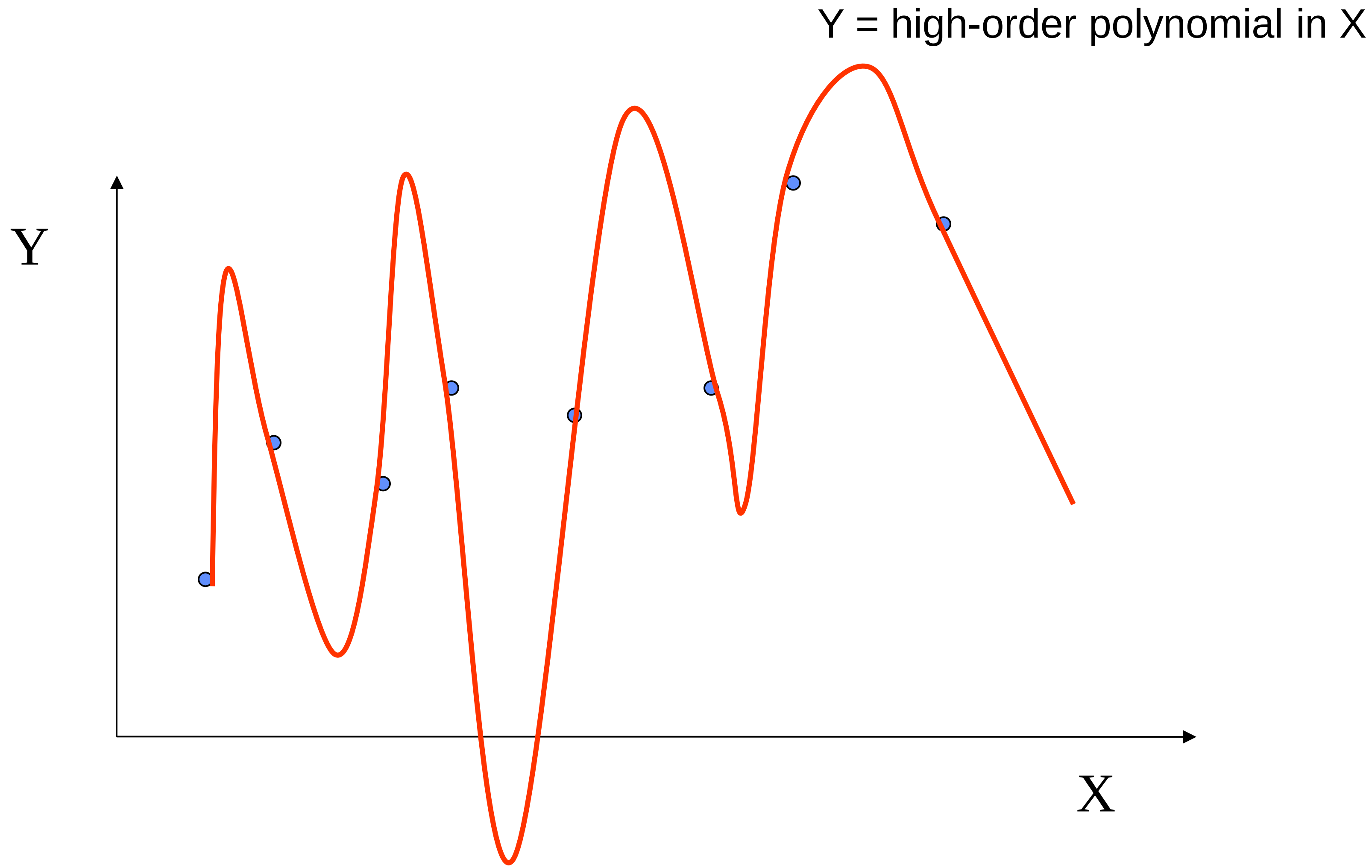
IG(Patrons) = 0.541 bits

IG(Type) = 0 bits

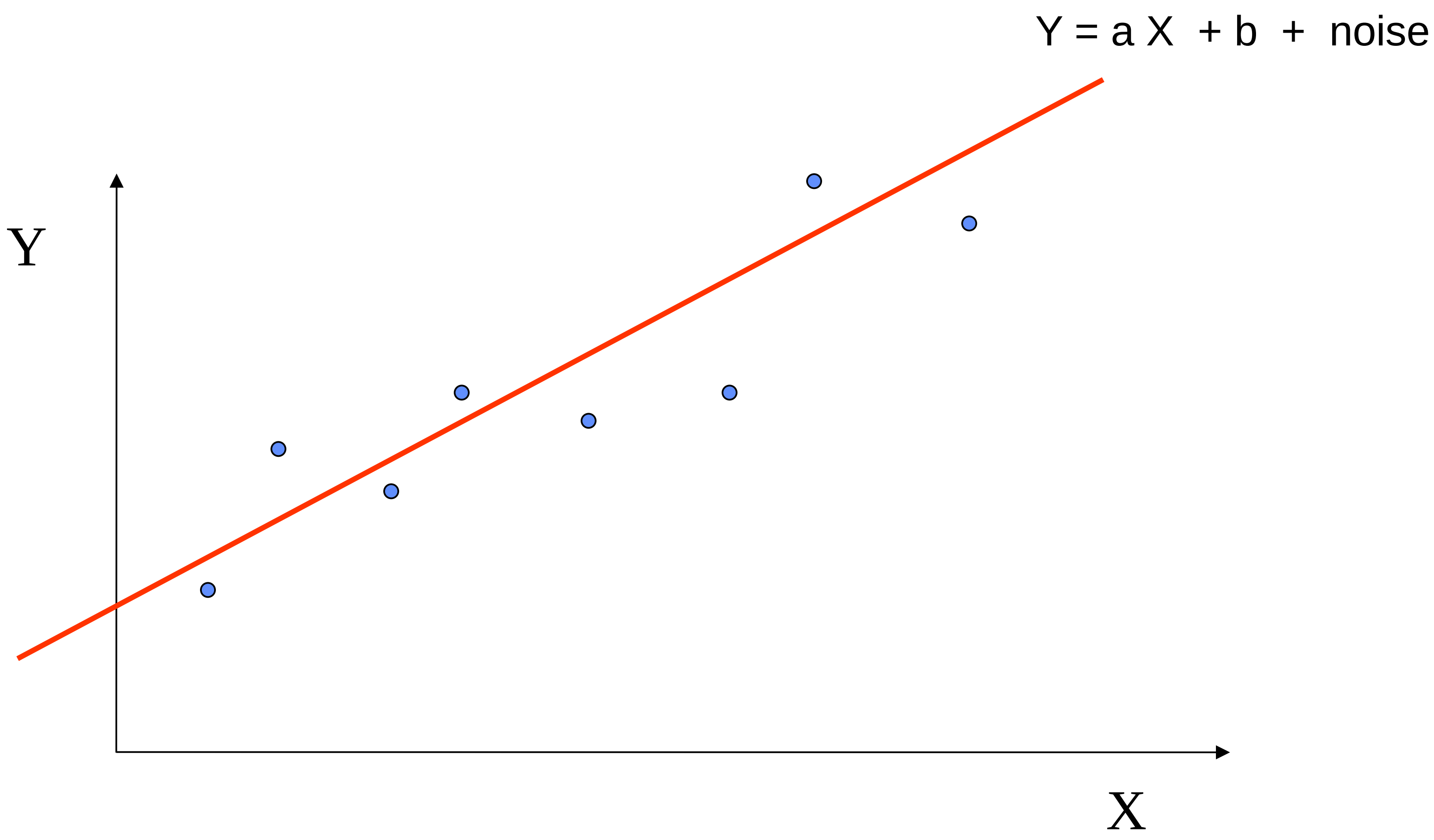
Overfitting and Underfitting



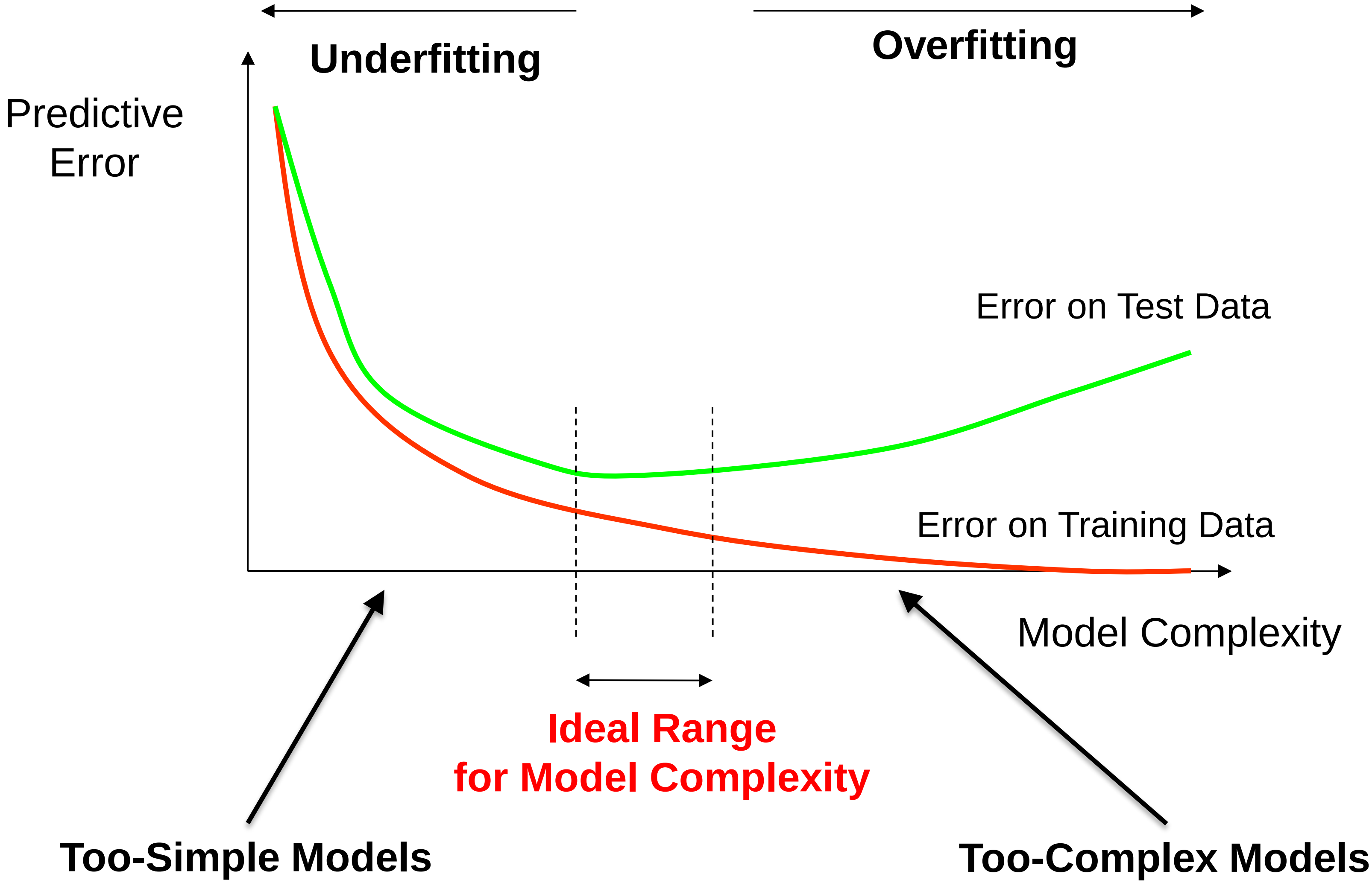
A Complex Model



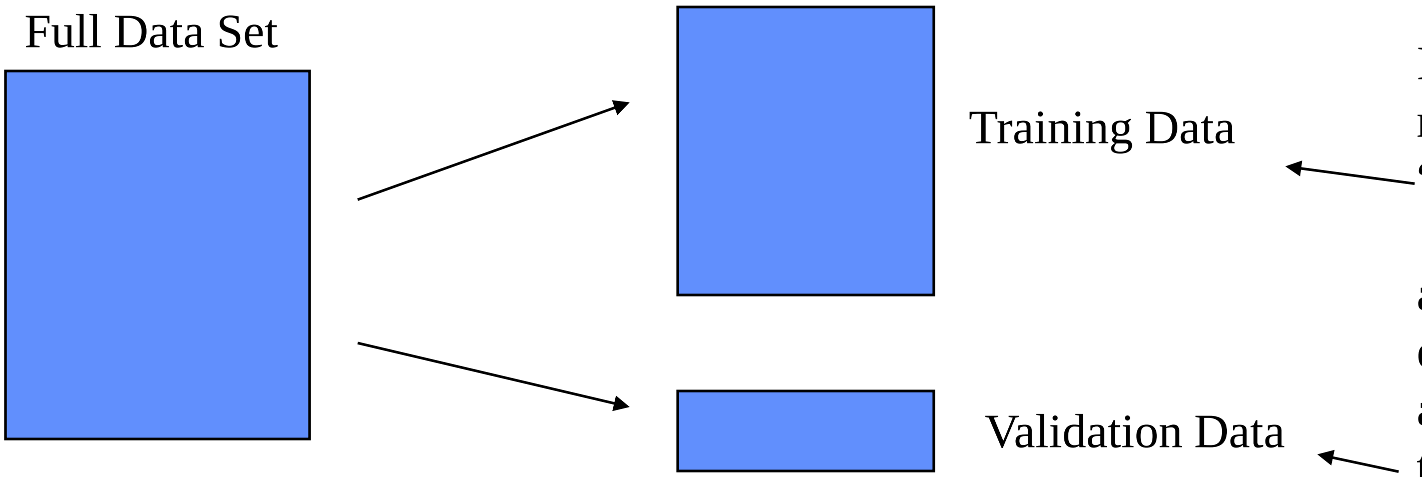
A Much Simpler Model



How Overfitting affects Prediction



Training and Validation Data



Idea: train each model on the "training data"

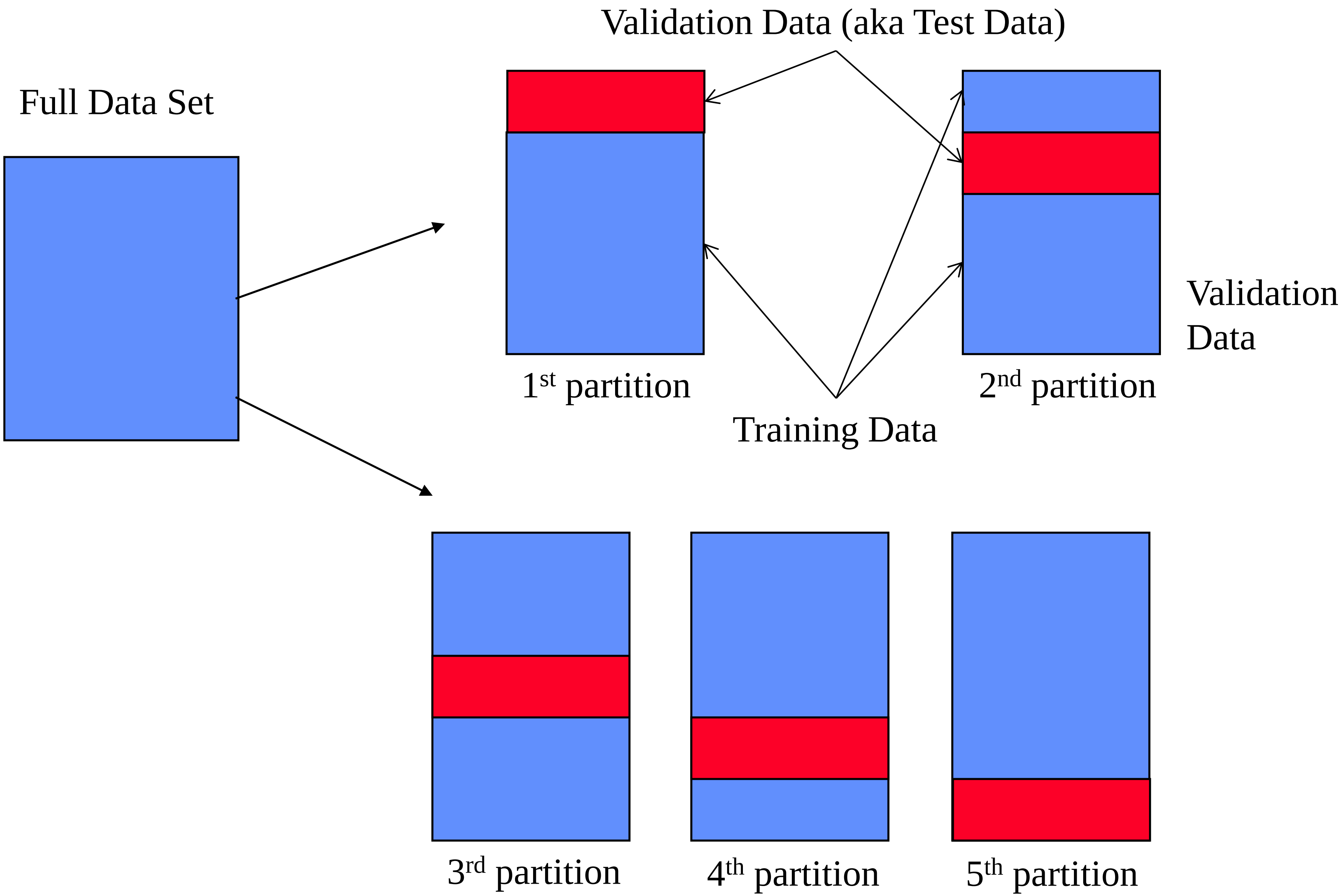
and then test each model's accuracy on the validation data

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 = $1/k \sum_i \text{Acc}(i)$
  - 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



Classification in Euclidean Space

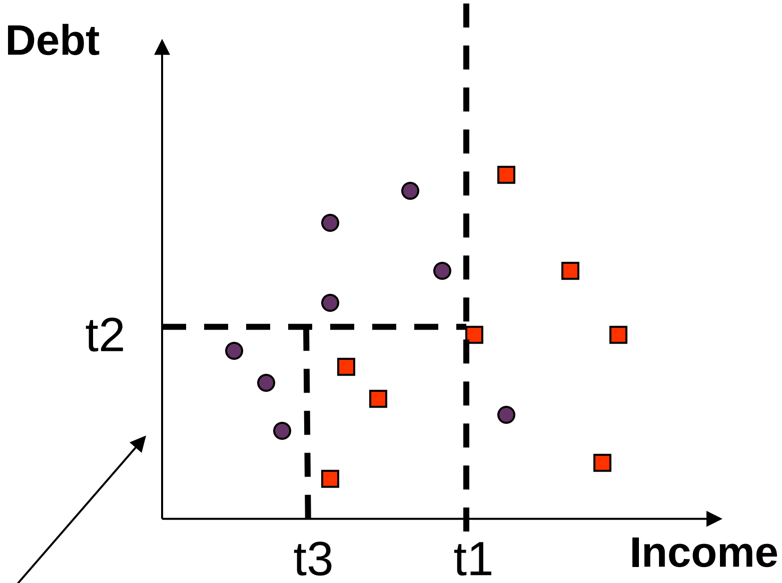
- A classifier is a partition of the space $\underline{x}$ into disjoint decision regions
  - Each region has a label attached
  - Regions with the same label need not be contiguous
  - For a new test point, find what decision region it is in, and predict the corresponding label

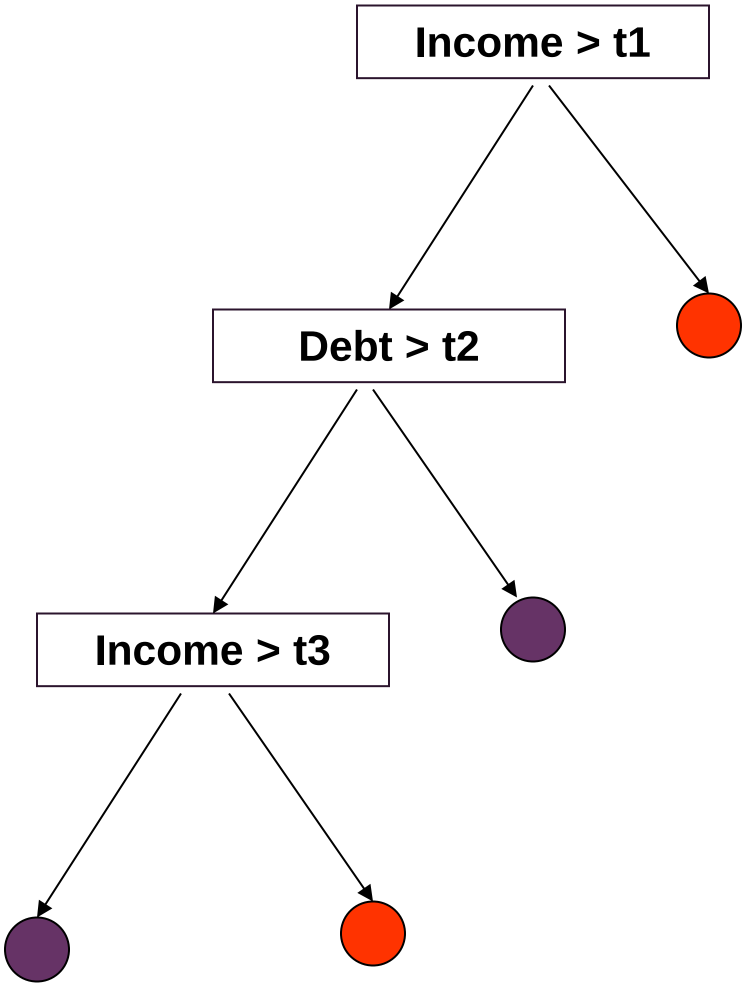
- Decision boundaries = boundaries between decision regions
  - The "dual representation" of decision regions

- We can characterize a classifier by the equations for its decision boundaries

- Learning a classifier $\Leftrightarrow$ searching for the decision boundaries that optimize our objective function

Decision Tree Example



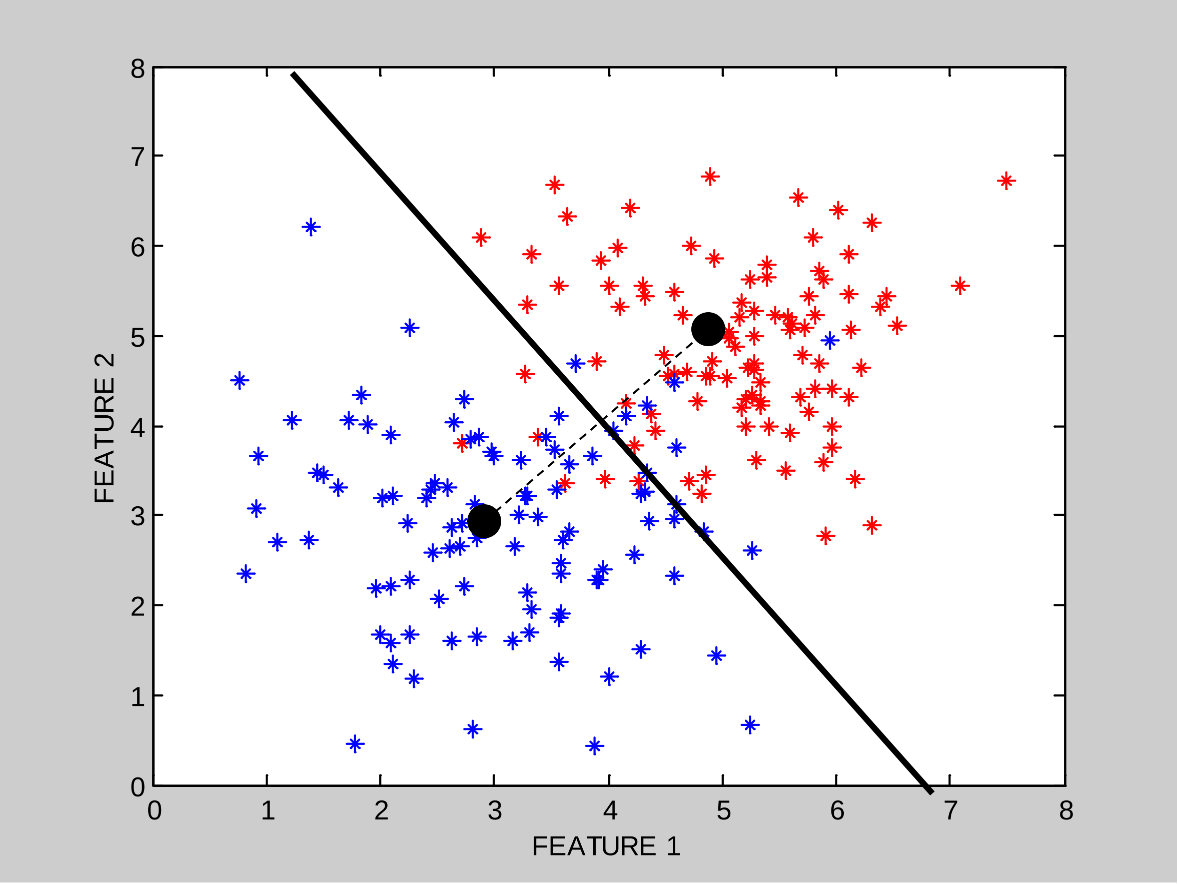


Note: tree boundaries are linear and axis-parallel

A Simple Classifier: Minimum Distance Classifier

- Training
  - Separate training vectors by class
  - Compute the mean for each class, $\underline{\mu}_k$,   k = 1,… m
- Prediction
  - Compute the closest mean to a test vector $\underline{x}'$ (using Euclidean distance)
  - Predict the corresponding class
- In the 2-class case, the decision boundary is defined by the locus of the hyperplane that is halfway between the 2 means and is orthogonal to the line connecting them
- This is a very simple-minded classifier – easy to think of cases where it will not work very well

Minimum Distance Classifier



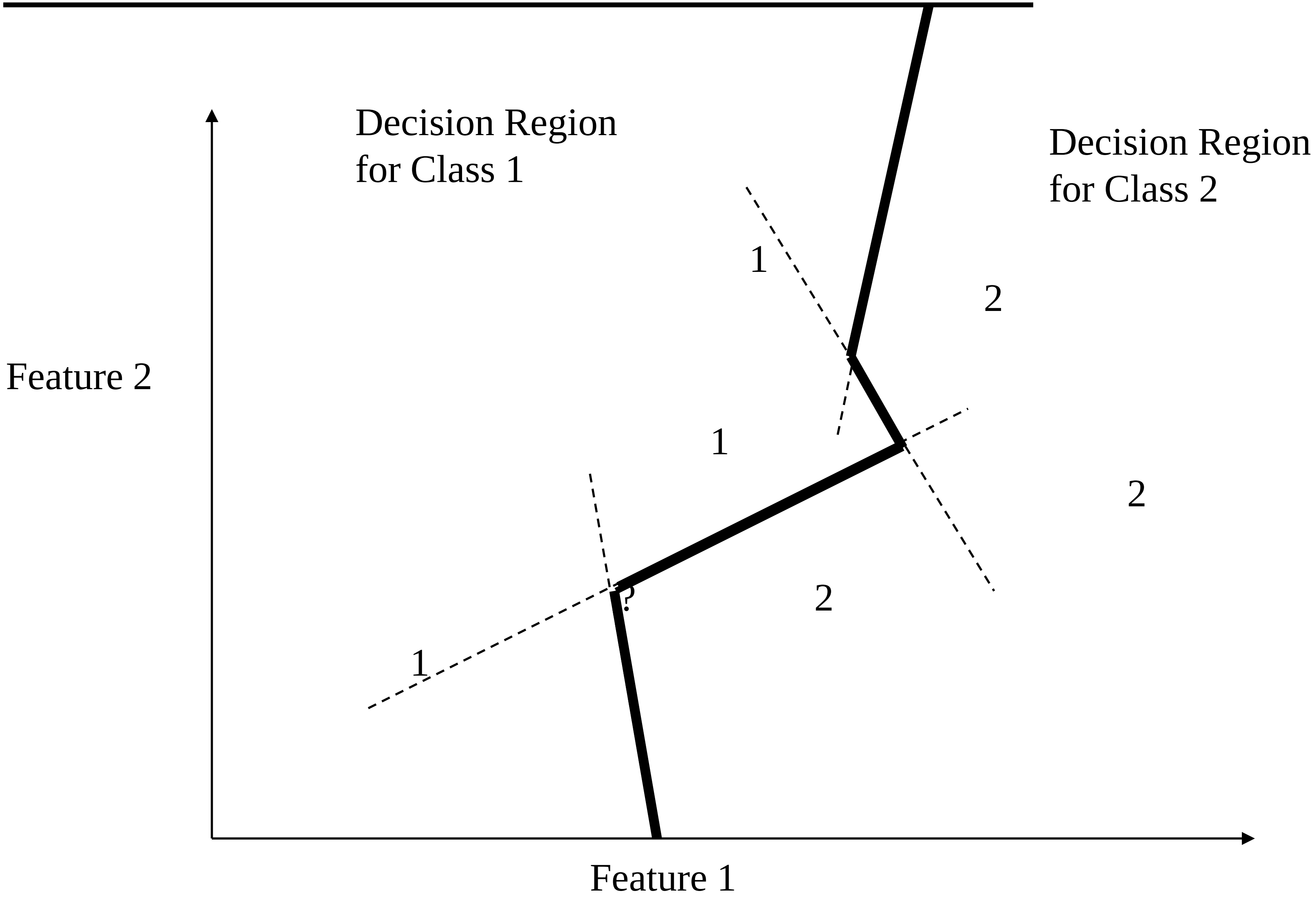
Another Example: Nearest Neighbor Classifier

- The nearest-neighbor classifier
  - Given a test point x', compute the distance between x' and each input data point
  - Find the closest neighbor in the training data
  - Assign x' the class label of this neighbor
  - (sort of generalizes minimum distance classifier to exemplars)

- If Euclidean distance is used as the distance measure (the most common choice), the nearest neighbor classifier results in piecewise linear decision boundaries

- Many extensions
  - e.g., kNN, vote based on k-nearest neighbors
  - k can be chosen by cross-validation

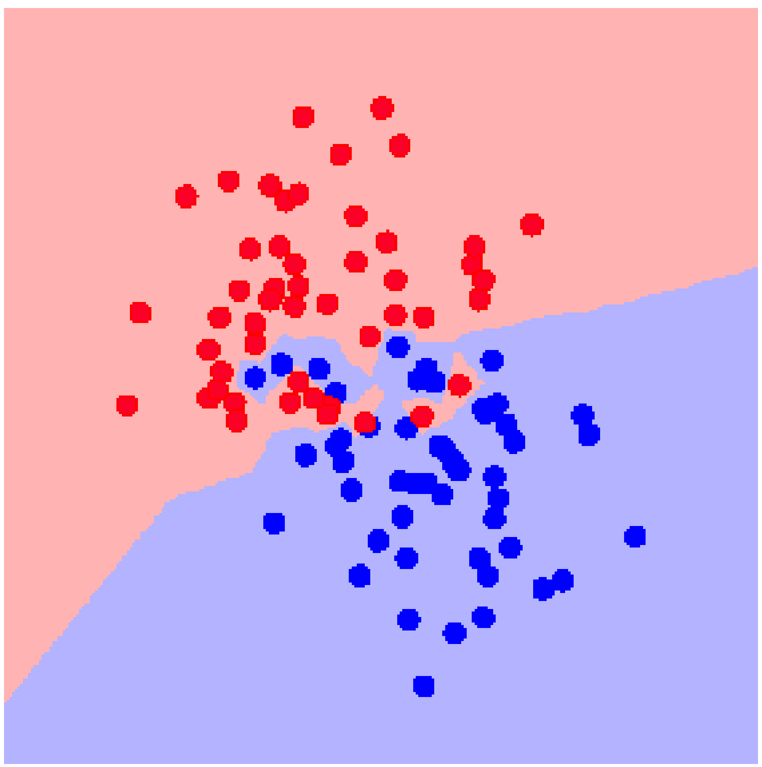
Overall Boundary = Piecewise Linear



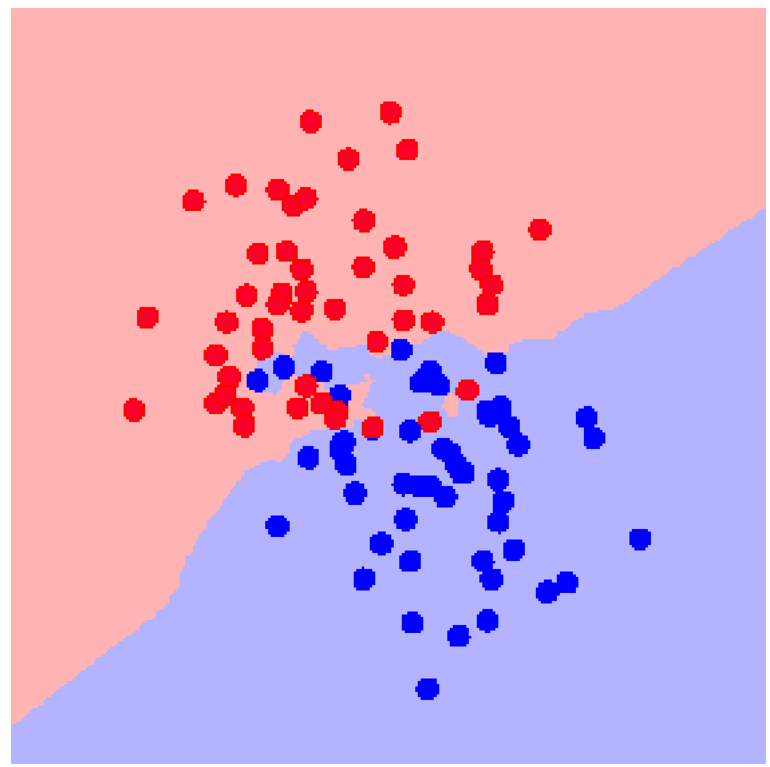
kNN Decision Boundary

- piecewise linear decision boundary
- Increasing k “simplifies” decision boundary
  - Majority voting means less emphasis on individual points

K = 1



K = 3

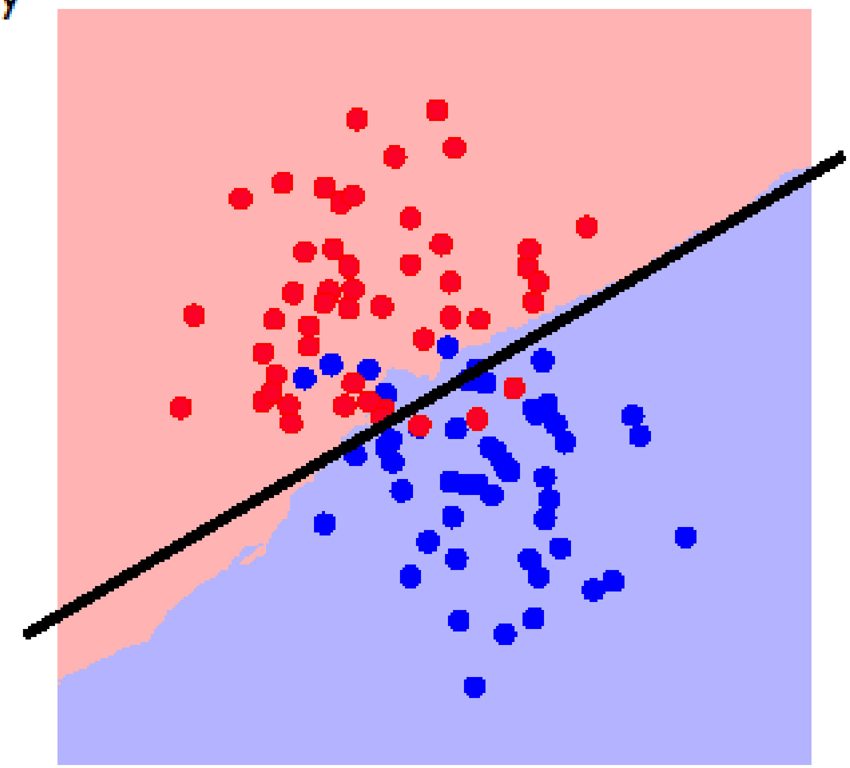


kNN Decision Boundary

- piecewise linear decision boundary
- Increasing k "simplifies" decision boundary
  - Majority voting means less emphasis on individual points

K = 25

- True ("best") decision boundary
  - In this case is linear
  - Compared to kNN: not bad!



Linear Classifiers

- Linear classifier ⇔ single linear decision boundary
  (for 2-class case)

- We can always represent a linear decision boundary by a linear equation:

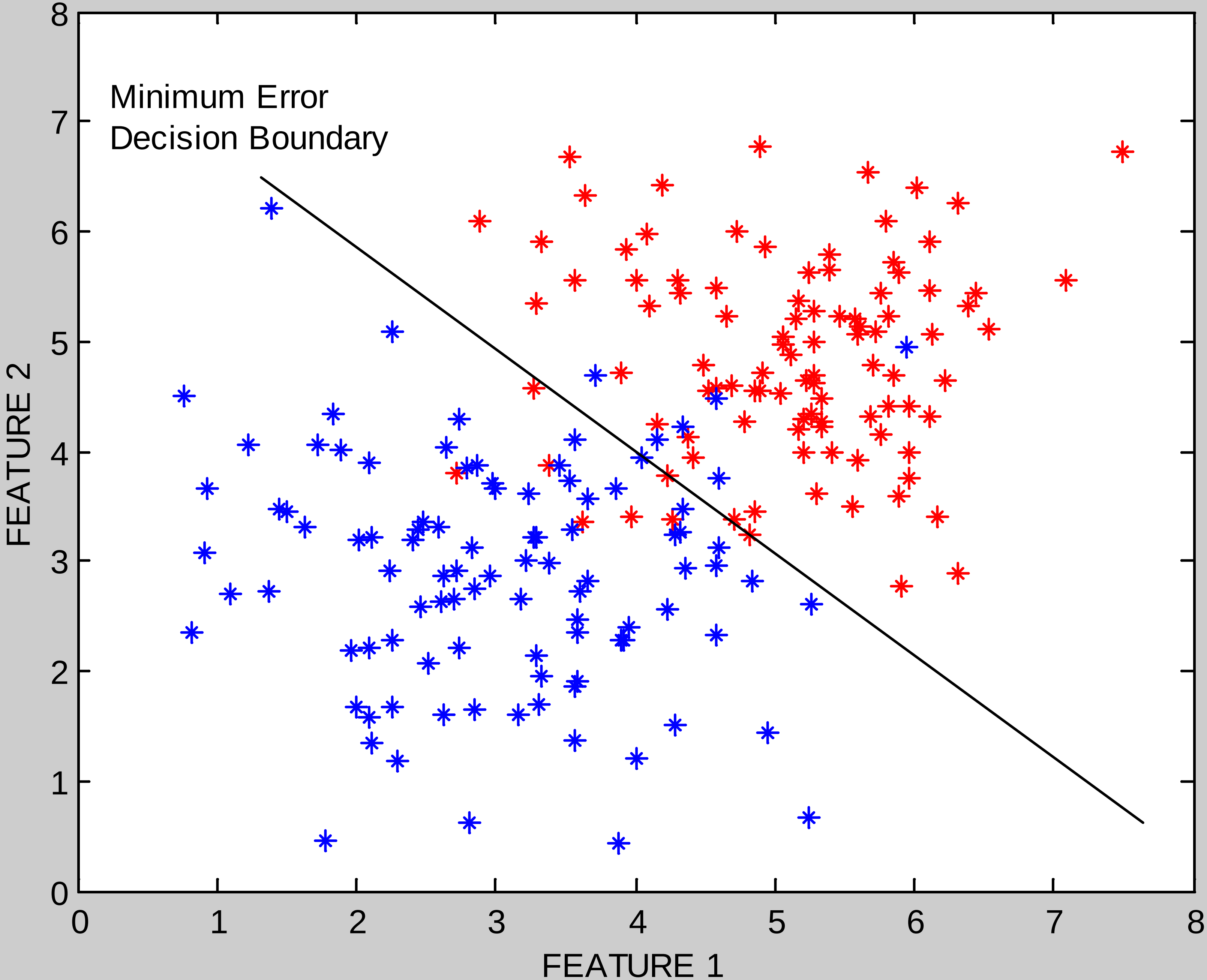
  $$w_1 x_1 + w_2 x_2 + \ldots + w_d x_d \quad = \Sigma\, w_j x_j \; = \; \underline{w}^t \, \underline{x} = 0$$

- In d dimensions, this defines a (d-1) dimensional hyperplane
  - d=3, we get a plane; d=2, we get a line

- For prediction we simply see if $\Sigma\, w_j x_j > 0$

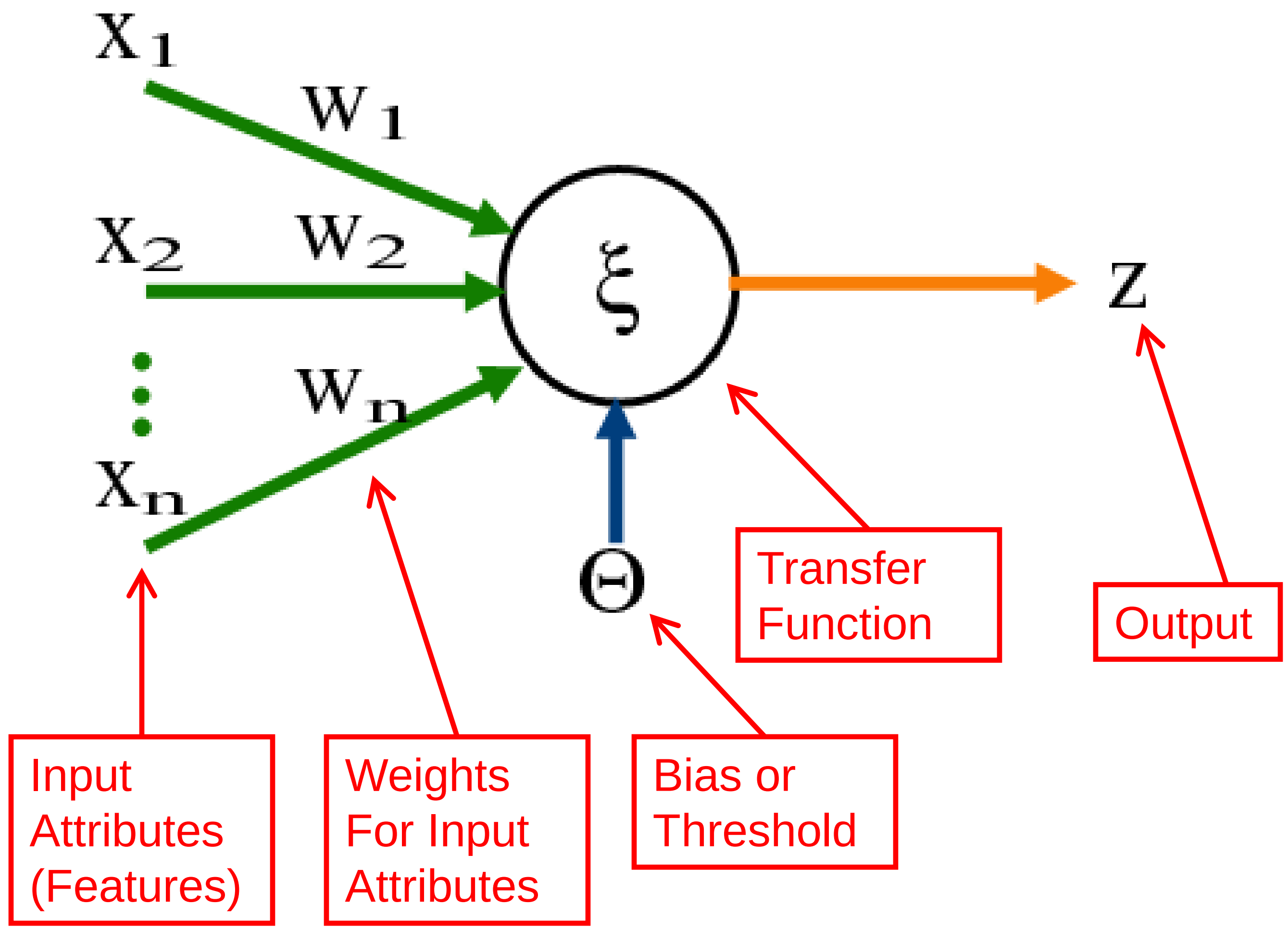
- The w_i are the weights (parameters)
  - Learning consists of searching in the d-dimensional weight space for the set of weights (the linear boundary) that minimizes an error measure
  - A threshold can be introduced by a “dummy” feature that is always one; it weight corresponds to (the negative of) the threshold

- Note that a minimum distance classifier is a special (restricted) case of a linear classifier



Minimum Error
Decision Boundary
FEATURE 1
FEATURE 2
0
1
2
3
4
5
6
7
8

The Perceptron Classifier (pages 729-731 in text)



The Perceptron Classifier (pages 729-731 in text)

- The perceptron classifier is just another name for a linear classifier for 2-class data, i.e.,

  $$\text{output}(\underline{x}) = \text{sign}(\Sigma\, w_j\, x_j)$$

- Loosely motivated by a simple model of how neurons fire

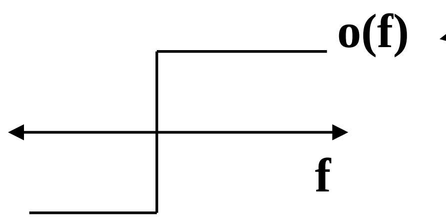
- For mathematical convenience, class labels are +1 for one class and -1 for the other

- Two major types of algorithms for training perceptrons
  - Objective function = classification accuracy ("error correcting")
  - Objective function = squared error (use gradient descent)

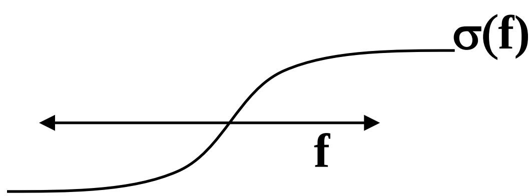
  - Gradient descent is generally faster and more efficient.

Two different types of perceptron output

x-axis below is $f(\underline{x}) = f$ = weighted sum of inputs
y-axis is the perceptron output



Thresholded output,
takes values +1 or -1



Sigmoid output, takes
real values between -1 and +1

The sigmoid is in effect an approximation
to the threshold function above, but
has a gradient that we can use for learning

Support Vector Machines (SVM): "Modern perceptrons" (section 18.9, R&N)

- A modern linear separator classifier
  - Essentially, a perceptron with a few extra wrinkles

- Constructs a **"maximum margin separator"**
  - A linear decision boundary with the largest possible distance from the decision boundary to the example points it separates
  - "Margin" = Distance from decision boundary to closest example
  - The "maximum margin" helps SVMs to generalize well

- Can embed the data in a non-linear higher dimension space
  - Constructs a linear separating hyperplane in that space
    - **This can be a non-linear boundary in the original space**
  - Algorithmic advantages and simplicity of linear classifiers
  - Representational advantages of non-linear decision boundaries

- **Currently most popular "off-the shelf" supervised classifier.**

Constructs a “maximum margin separator”

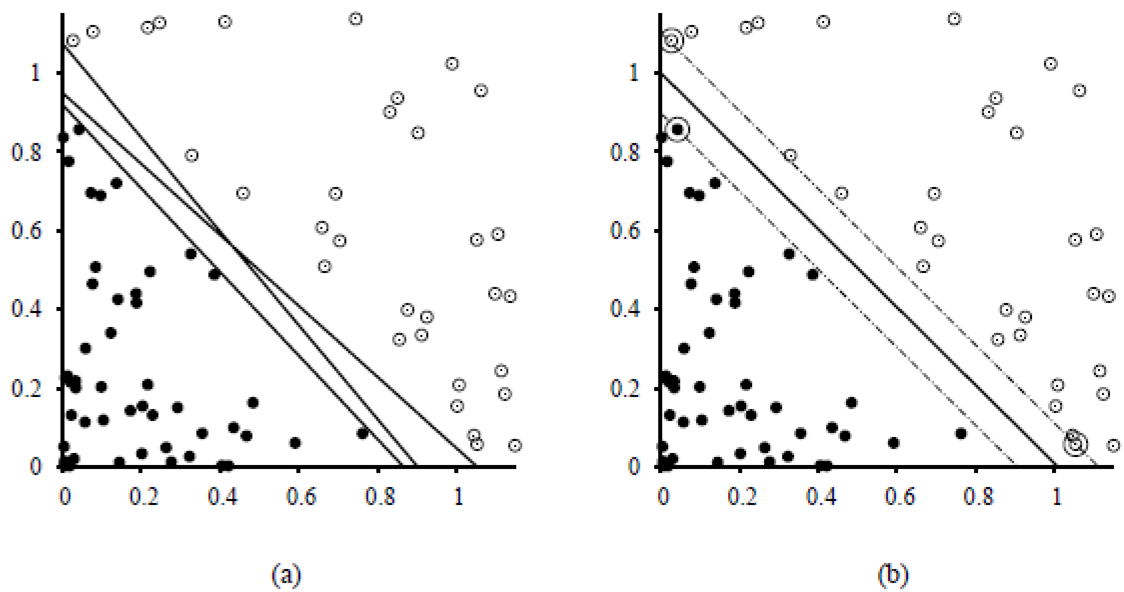


Figure 18.30 **FILES: .** Support vector machine classification: (a) Two classes of points (black and white circles) and three candidate linear separators. (b) The maximum margin separator (heavy line), is at the midpoint of the **margin** (area between dashed lines). The **support vectors** (points with large circles) are the examples closest to the separator.

Can embed the data in a non-linear higher dimension space

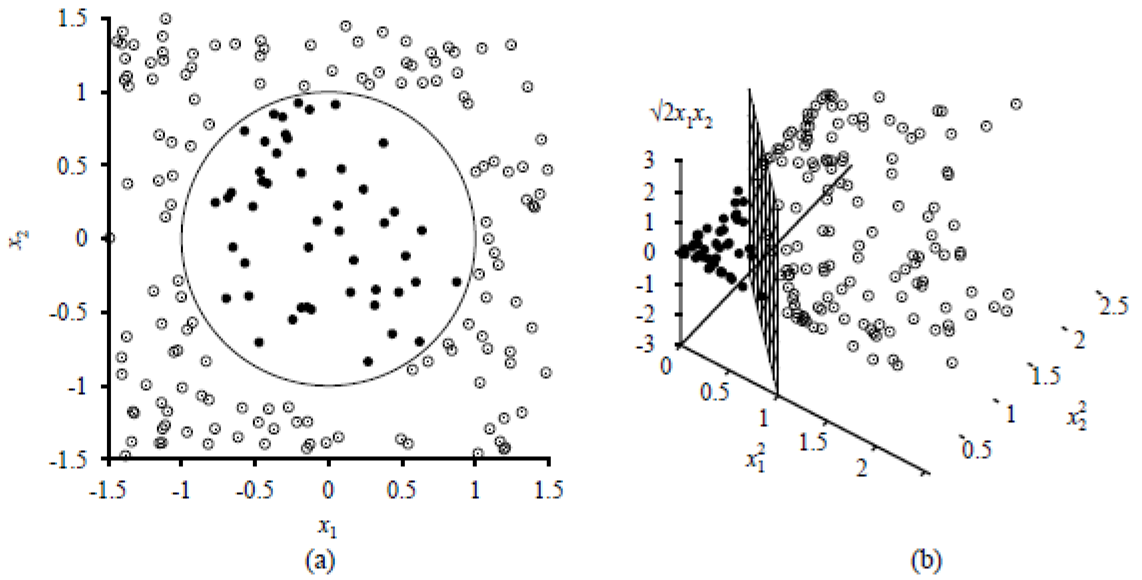


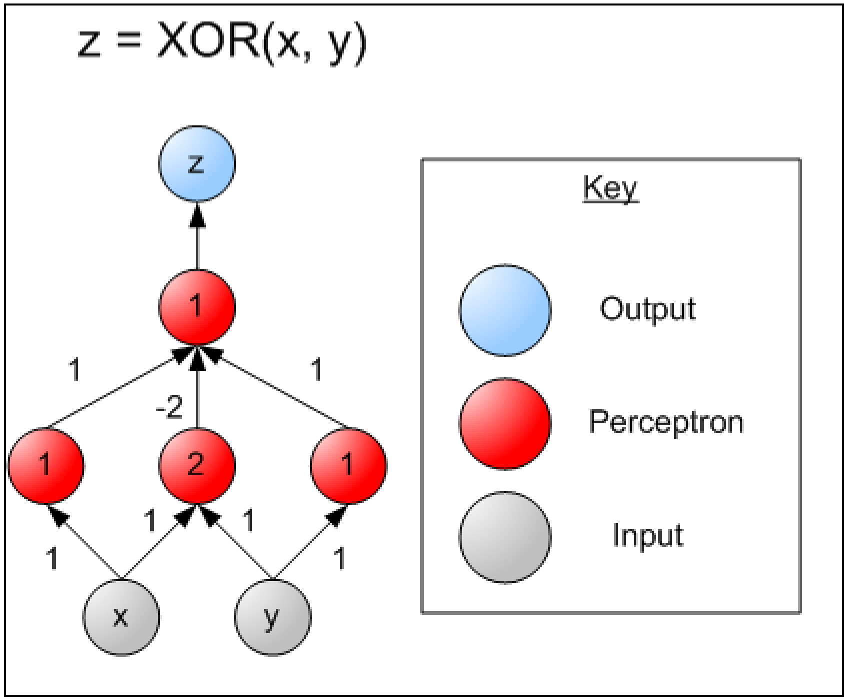
Figure 18.31 **FILES: .** (a) A two-dimensional training set with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions. Figure 18.29(b) gives a closeup of the separator in (b).

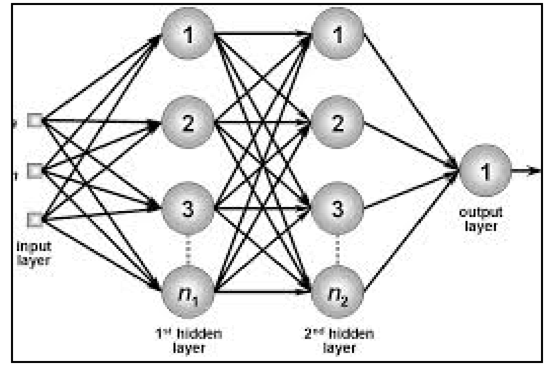
Multi-Layer Perceptrons (Artificial Neural Networks)

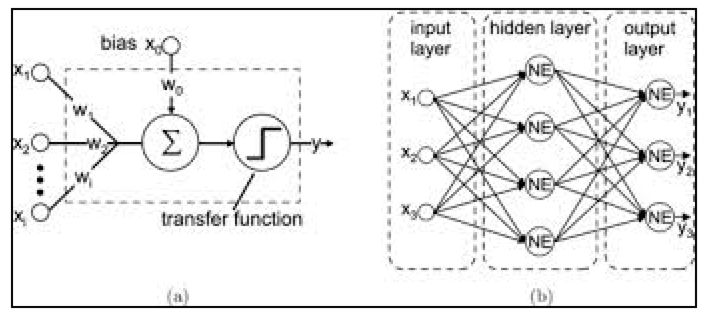
(sections 18.7.3-18.7.4 in textbook)

- What if we took K perceptrons and trained them in parallel and then took a weighted sum of their sigmoidal outputs?
  - This is a multi-layer neural network with a single "hidden" layer (the outputs of the first set of perceptrons)
  - If we train them jointly in parallel, then intuitively different perceptrons could learn different parts of the solution
    - They define different local decision boundaries in the input space
- What if we hooked them up into a general Directed Acyclic Graph?
  - Can create simple "neural circuits" (but no feedback; not fully general)
  - Often called neural networks with hidden units
- How would we train such a model?
  - Backpropagation algorithm = clever way to do gradient descent
  - Bad news: many local minima and many parameters
    - training is hard and slow
  - Good news: can learn general non-linear decision boundaries
  - Generated much excitement in AI in the late 1980's and 1990's
  - New current excitement with very large "deep learning" networks

Multi-Layer Perceptrons (Artificial Neural Networks)
(sections 18.7.3-18.7.4 in textbook)

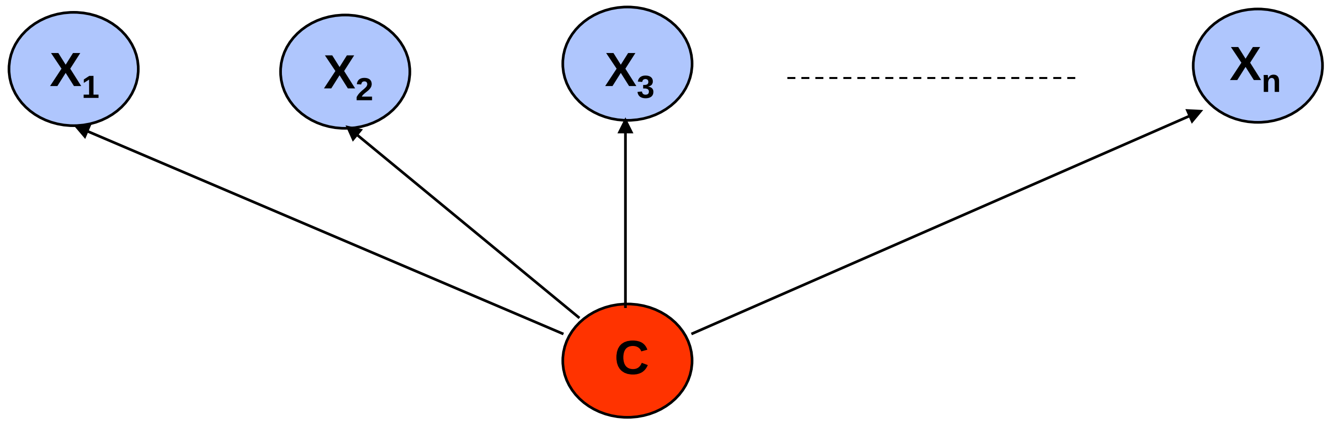






Naïve Bayes Model

(section 20.2.2 R&N 3rd ed.)



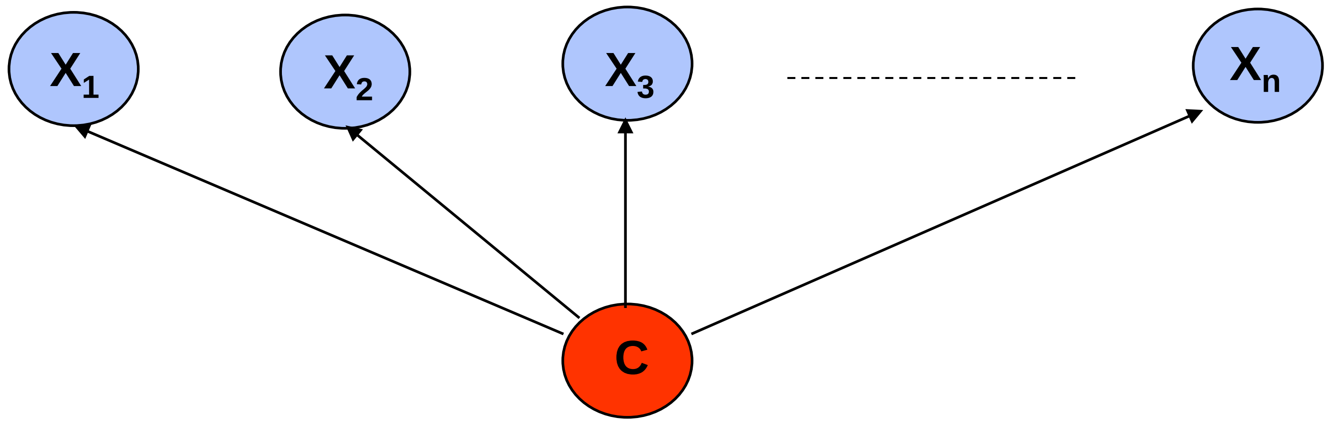
Basic Idea: We want to estimate $P(C \mid X_1,\ldots X_n)$, but it's hard to think about computing the probability of a class from input attributes of an example.

Solution: Use Bayes' Rule to turn $P(C \mid X_1,\ldots X_n)$ into a proportionally equivalent expression that involves only $P(C)$ and $P(X_1,\ldots X_n \mid C)$.
Then assume that feature values are conditionally independent given class, which allows us to turn $P(X_1,\ldots X_n \mid C)$ into $\Pi_i\ P(X_i \mid C)$.

We estimate $P(C)$ easily from the frequency with which each class appears within our training data, and we estimate $P(X_i \mid C)$ easily from the frequency with which each X_i appears in each class C within our training data.

Naïve Bayes Model

(section 20.2.2 R&N 3rd ed.)



Bayes Rule: $P(C \mid X_1,\ldots X_n)$ is proportional to $P(C)\ \Pi_i\ P(X_i \mid C)$
[note: denominator $P(X_1,\ldots X_n)$ is constant for all classes, may be ignored.]

Features Xi are conditionally independent given the class variable C

- choose the class value c_i with the highest $P(c_i \mid x_1,\ldots, x_n)$
- simple to implement, often works very well
- e.g., spam email classification: X's = counts of words in emails

Conditional probabilities $P(X_i \mid C)$ can easily be estimated from labeled date

- Problem: Need to avoid zeroes, e.g., from limited training data
- Solutions: Pseudo-counts, beta[a,b] distribution, etc.

Naïve Bayes Model (2)

$$P(C \mid X_1,\ldots X_n) = \alpha \prod P(X_i \mid C)\ P(C)$$

Probabilities P(C) and P(Xi | C) can easily be estimated from labeled data

P(C = cj) ≈ #(Examples with class label cj) / #(Examples)

P(Xi = xik | C = cj)
≈ #(Examples with Xi value xik and class label cj)
/ #(Examples with class label cj)

Usually easiest to work with logs

$$\log [\ P(C \mid X_1,\ldots X_n)\ ]$$
$$= \log \alpha + \sum [\ \log P(X_i \mid C) + \log P(C)\ ]$$

DANGER: Suppose ZERO examples with Xi value xik and class label cj ?
An unseen example with Xi value xik will NEVER predict class label cj !

Practical solutions: Pseudocounts, e.g., add 1 to every #() , etc.
Theoretical solutions: Bayesian inference, beta distribution, etc.

Classifier Bias — Decision Tree or Linear Perceptron?

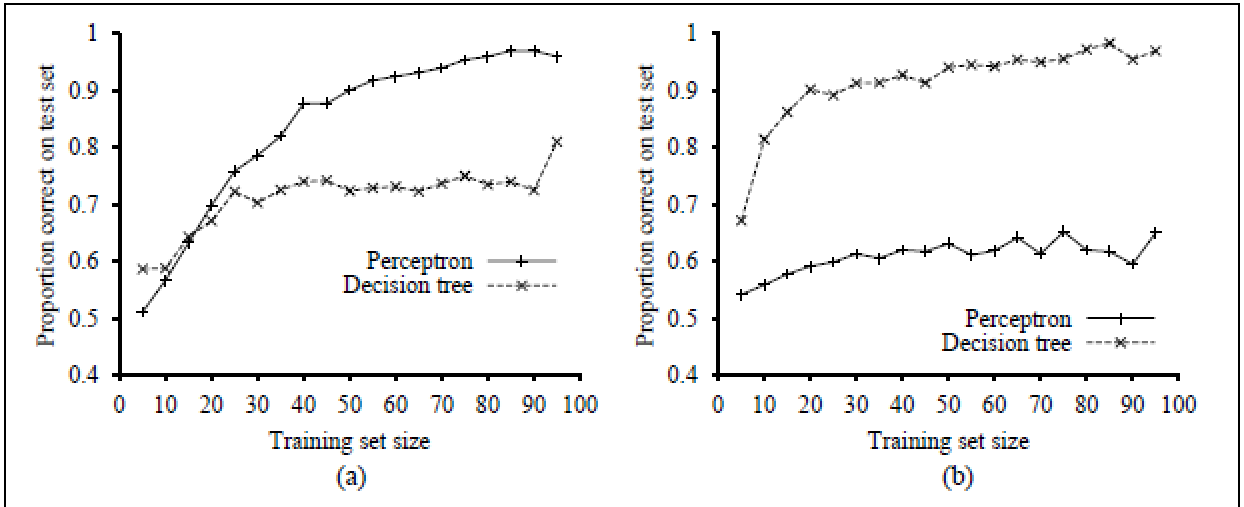
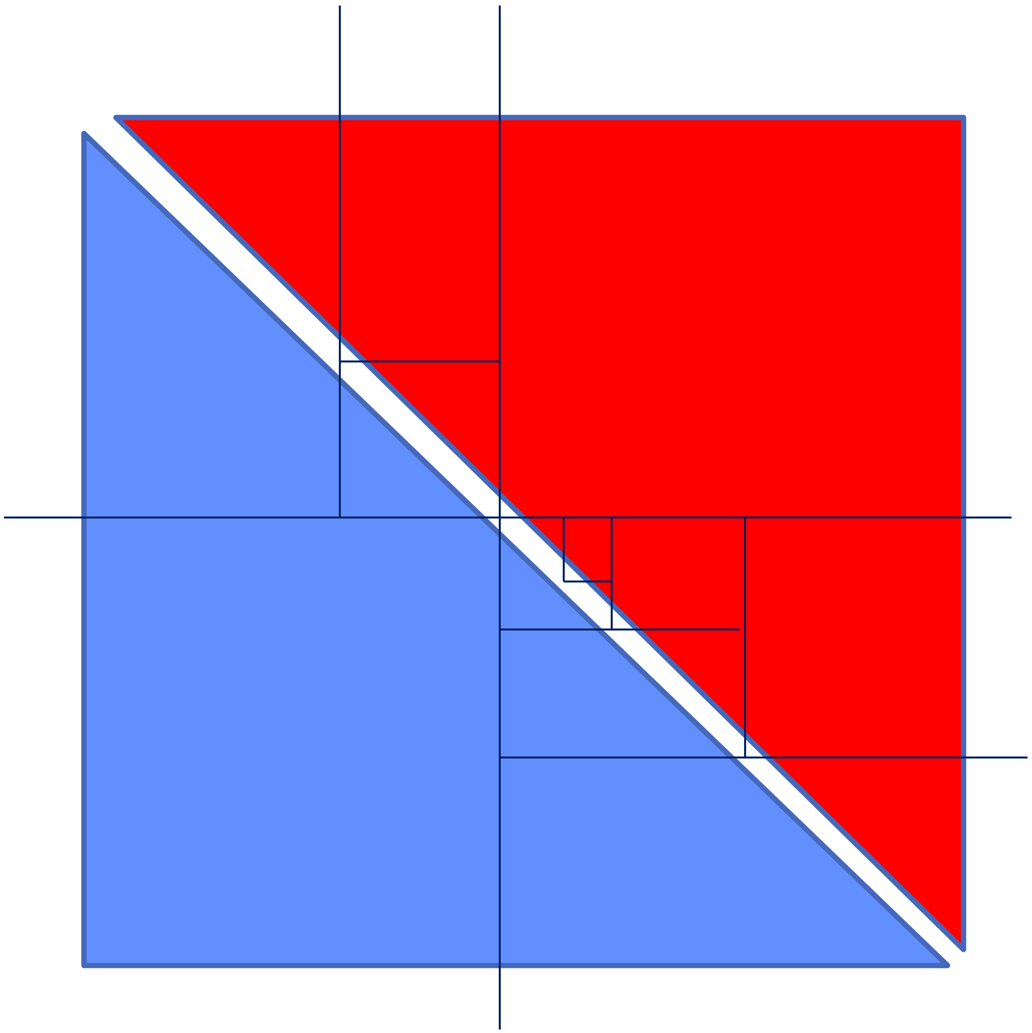
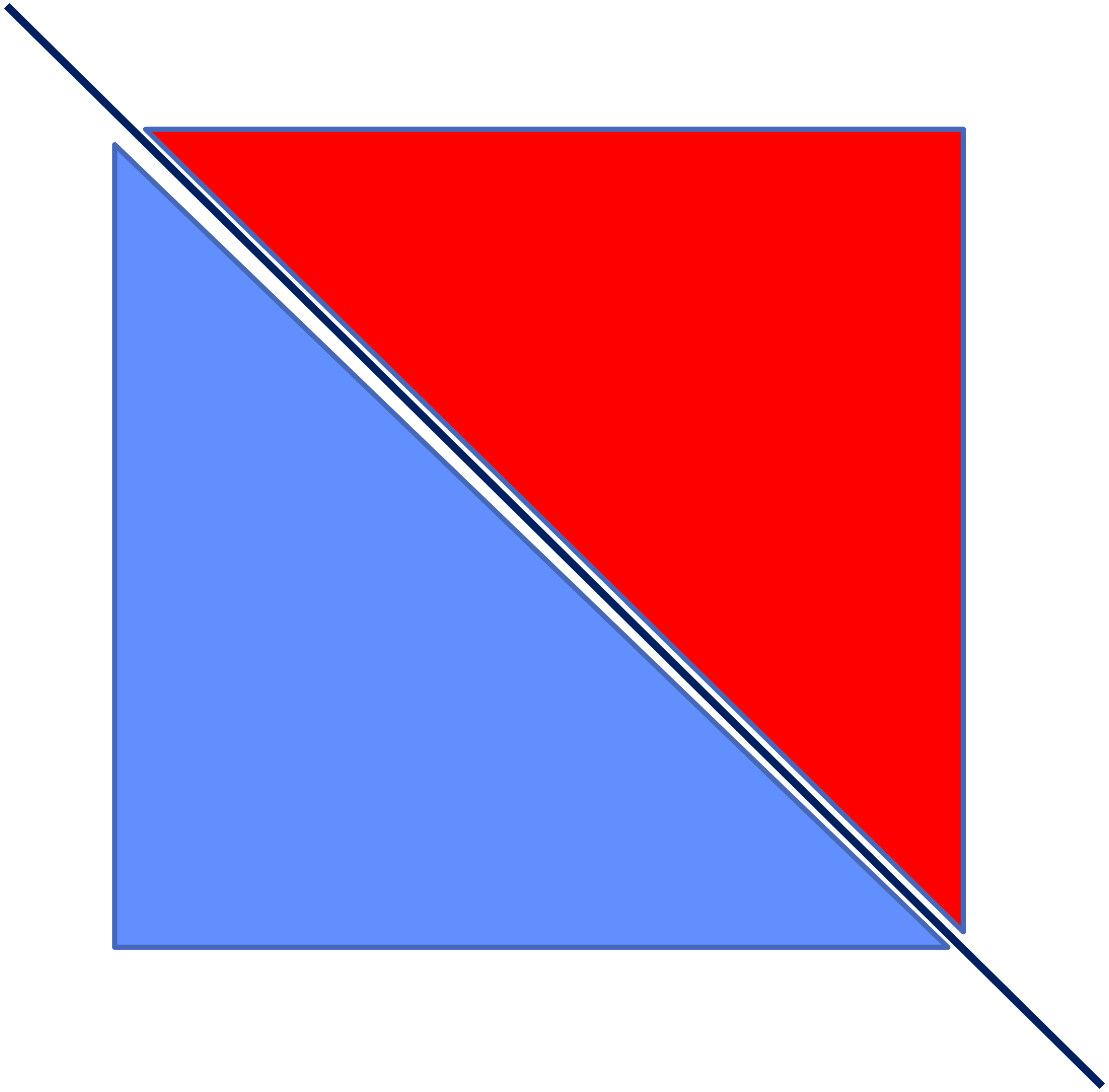


Figure 18.22 **FILES: .** Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the $WillWait$ predicate in the restaurant example.

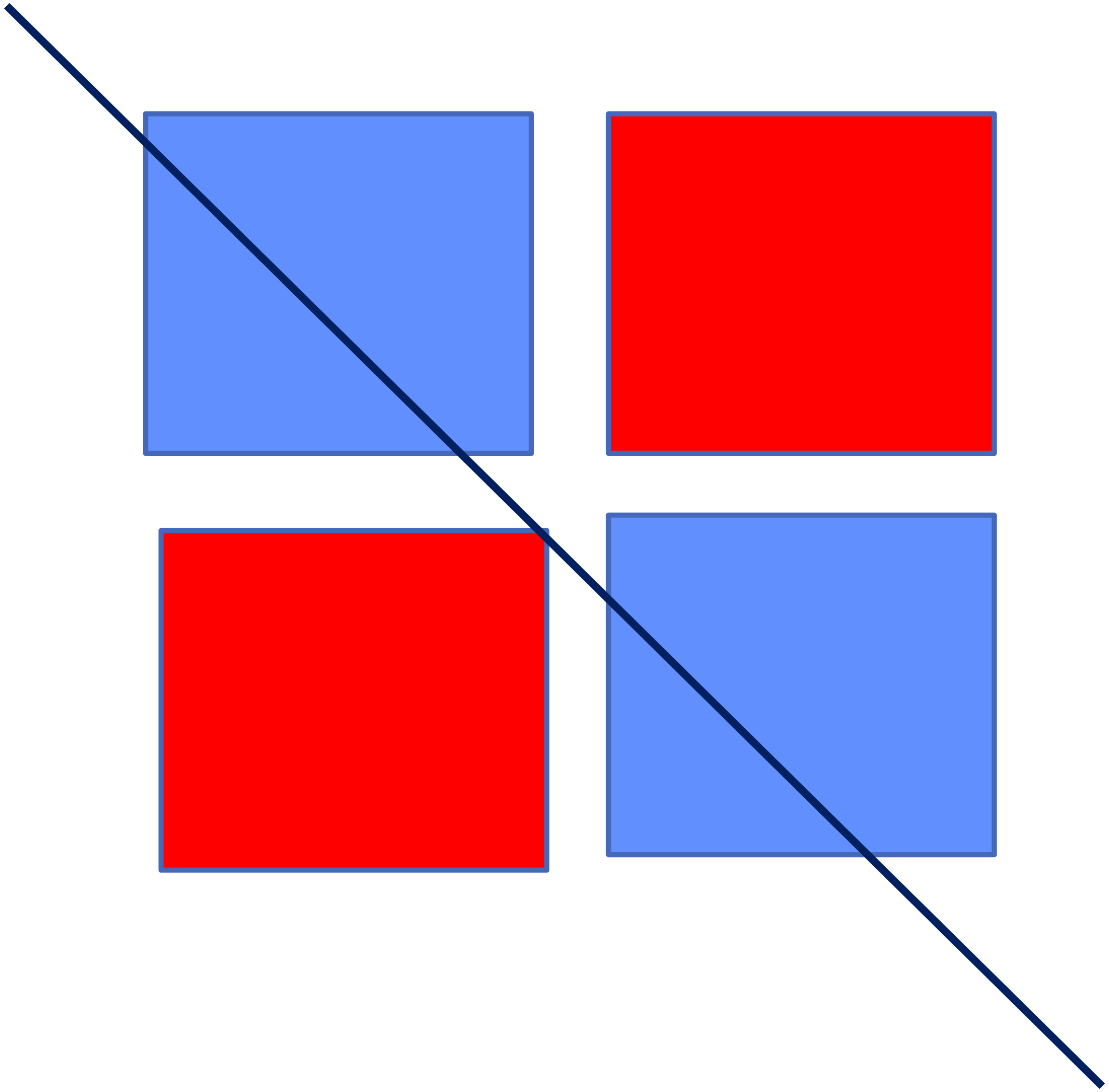
Classifier Bias — Decision Tree or Linear Perceptron?



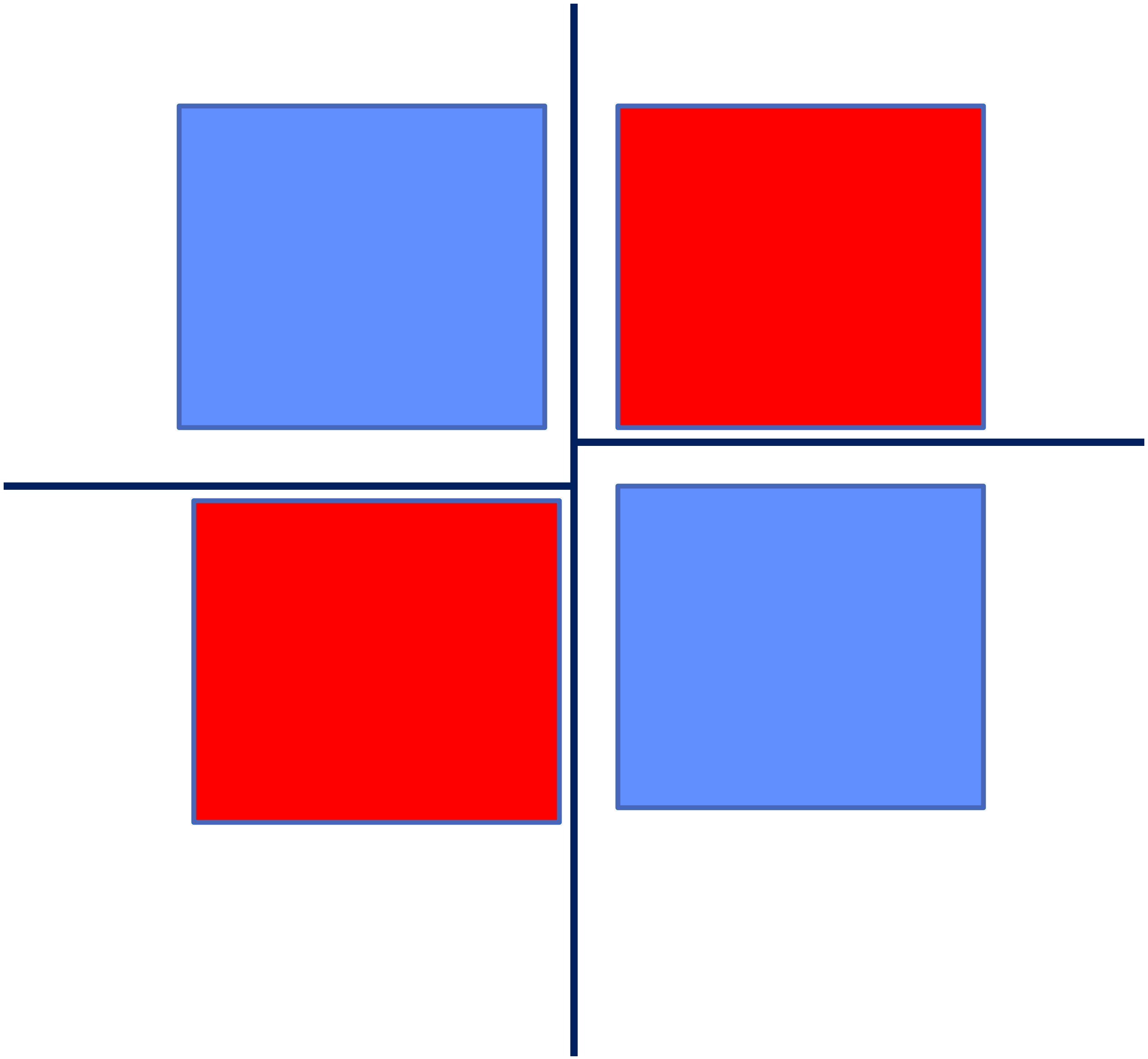
Classifier Bias — Decision Tree or Linear Perceptron?



Classifier Bias — Decision Tree or Linear Perceptron?



Classifier Bias — Decision Tree or Linear Perceptron?



CS-171 Final Review

- **Local Search**
  - (4.1-4.2, 4.6; Optional 4.3-4.5)
- **Constraint Satisfaction Problems**
  - (6.1-6.4, except 6.3.3)
- **Machine Learning**
  - (18.1-18.12; 20.2.2)
- Questions on any topic
- Pre-mid-term material if time and class interest
- Please review your quizzes, mid-term, & old tests
  - At least one question from a prior quiz or old CS-171 test will appear on the Final Exam (and all other tests)