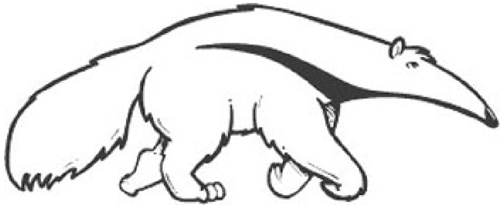
Introduction to Artificial Intelligence

CS171, Summer 1 Quarter, 2019
Introduction to Artificial Intelligence
Prof. Richard Lathrop



Read Beforehand: All assigned reading so far

BREN:ICS
INFORMATION AND COMPUTER SCIENCES

UNIVERSITY of CALIFORNIA IRVINE

CS-171 Final Review

- <u>**Machine Learning Classifiers**</u>
  - (R&N Ch. 18.5-18.12; 20.2)
- **Intro to Machine Learning**
  - (R&N Ch. 18.1-18.4)
- **Game (Adversarial) Search**
  - (R&N Ch. 5.1-5.4)
- **Local Search**
  - (R&N Ch. 4.1-4.2)
- **State Space Search**
  - (R&N Ch. 3.1-3.7)
- Questions on any topic
- Please review your quizzes & old tests

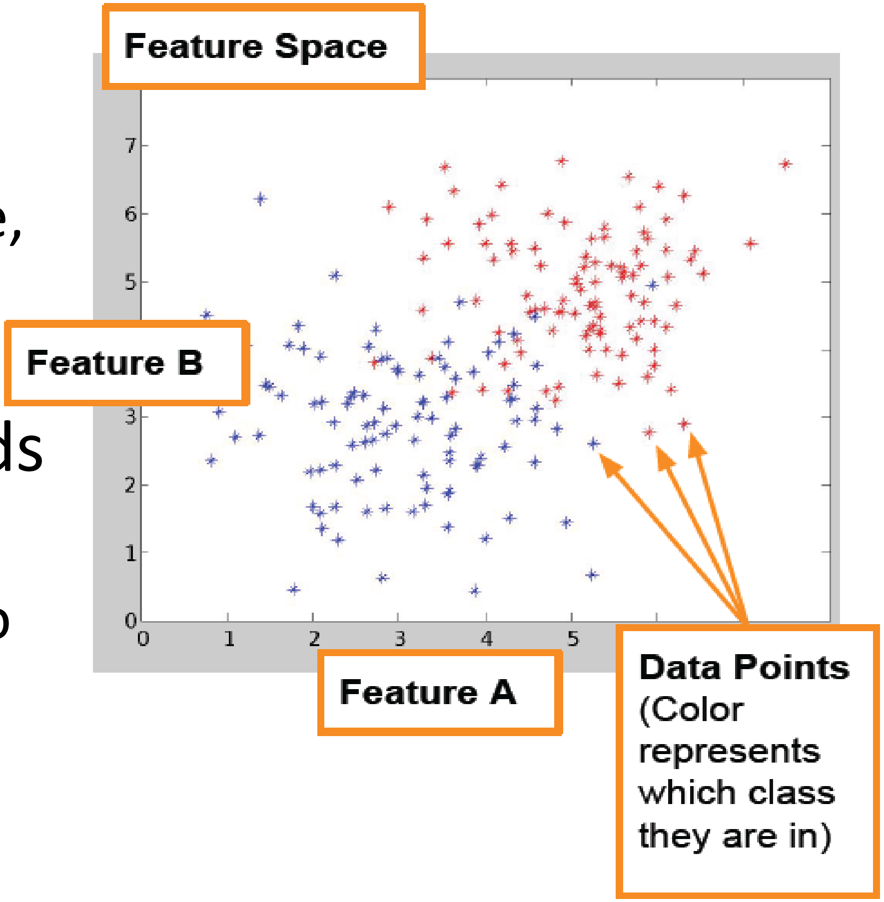
Review Machine Learning Classifiers Chapters 18.5-18.12; 20.2.2

- Decision Regions and Decision Boundaries

- Classifiers:
  - Decision trees
  - K-nearest neighbors
  - Perceptrons
  - Support vector Machines (SVMs), Neural Networks
  - Naïve Bayes

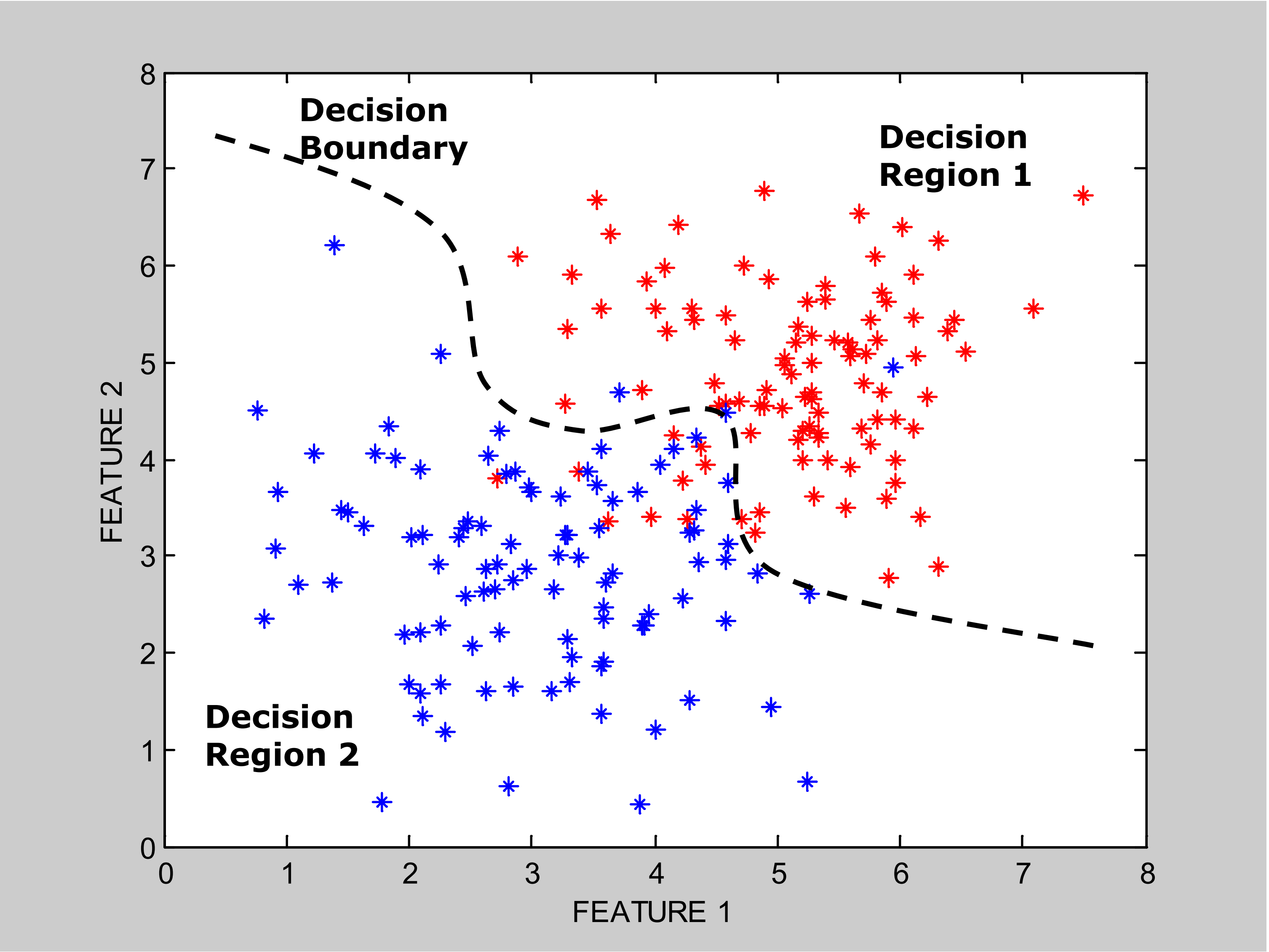
A Different View on Data Representation

- Data pairs can be plotted in "feature space"
- Each axis represents one feature.
  - This is a d dimensional space, where d is the number of features.
- Each data case corresponds to one point in the space.
  - In this figure we use color to represent their class label.



Decision Boundaries

Can we find a boundary that separates the two classes?



Classification in Euclidean Space

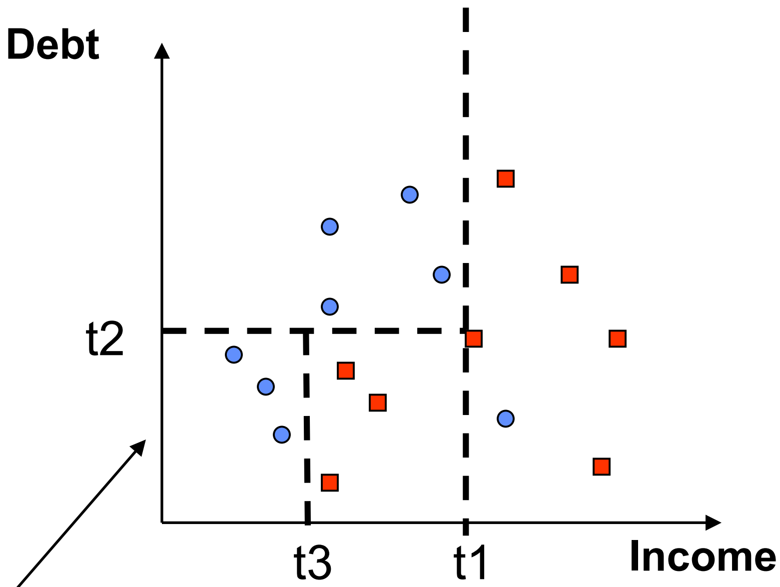
- A classifier is a partition of the feature space into disjoint decision regions
  - Each region has a label attached
  - Regions with the same label need not be contiguous
  - For a new test point, find what decision region it is in, and predict the corresponding label

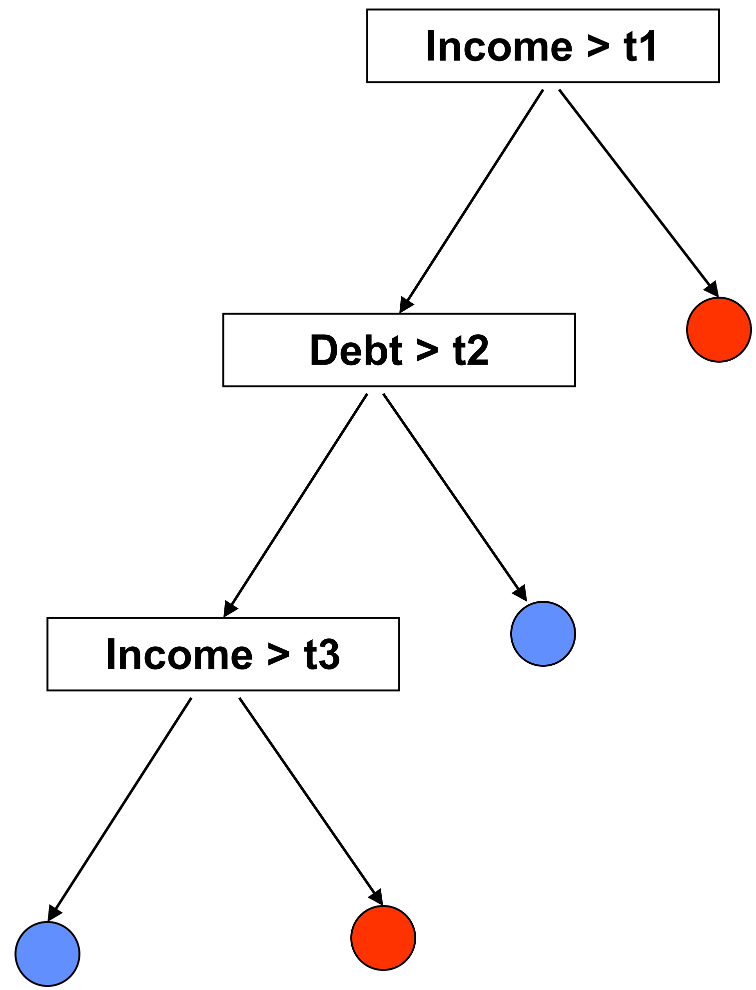
- Decision boundaries = boundaries between decision regions
  - The “dual representation” of decision regions

- 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

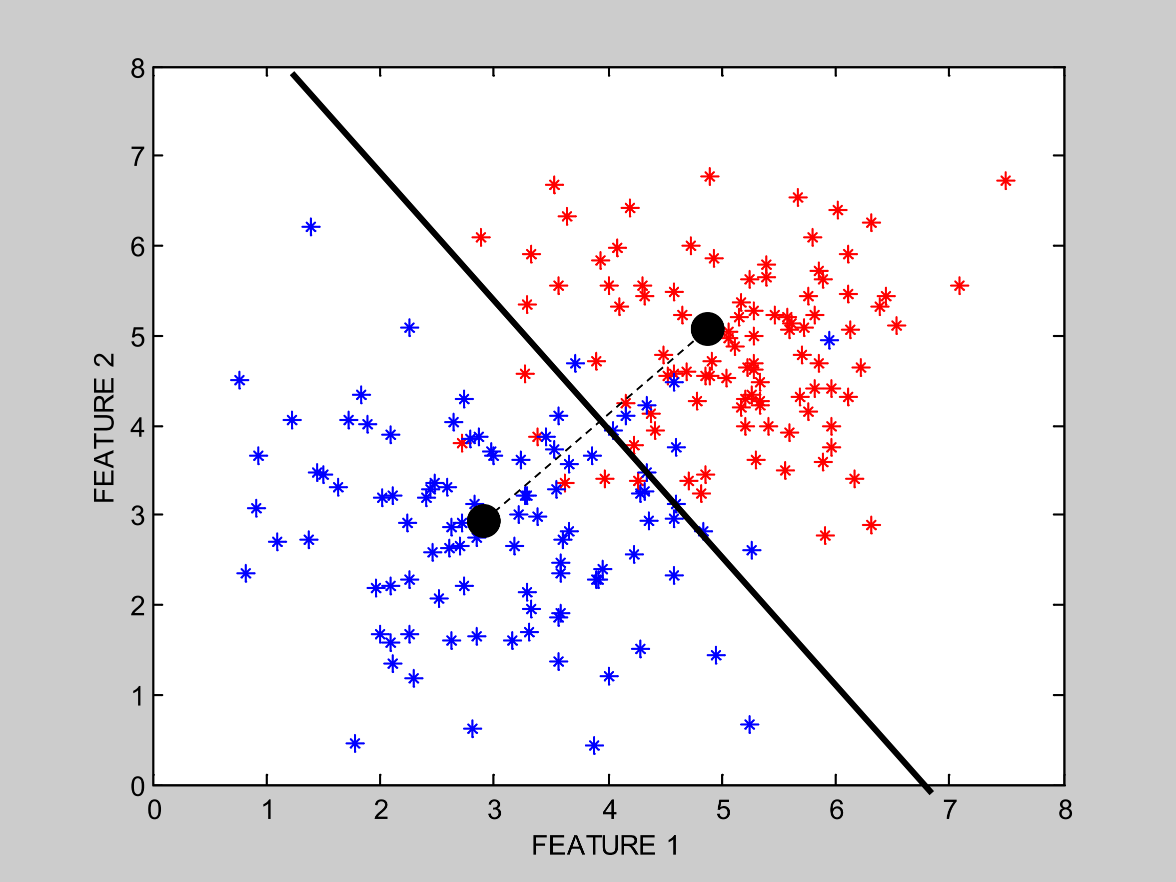
- Training
  - Separate training vectors by class
  - Compute the mean for each class, $\underline{\mu}_k$, $k = 1, \ldots m$

- 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 $\underline{x}'$, compute the distance between $\underline{x}'$ and each input data point
  - Find the closest neighbor in the training data
  - Assign $\underline{x}'$ the class label of this neighbor
  - (sort of generalizes minimum distance classifier to exemplars)

- The nearest neighbor classifier results in piecewise linear decision boundaries

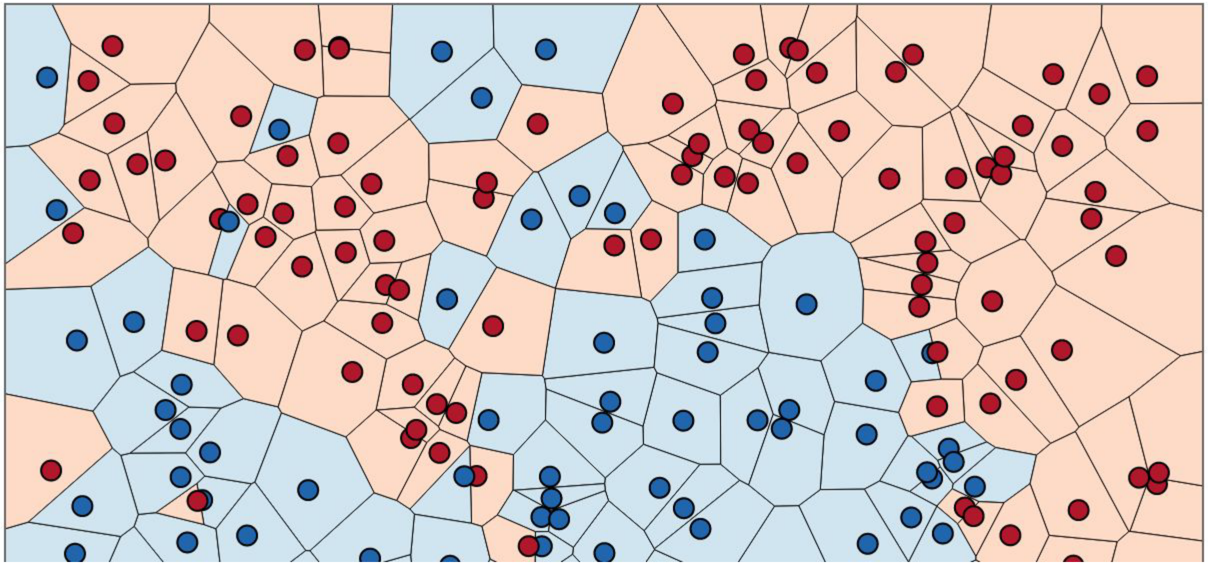
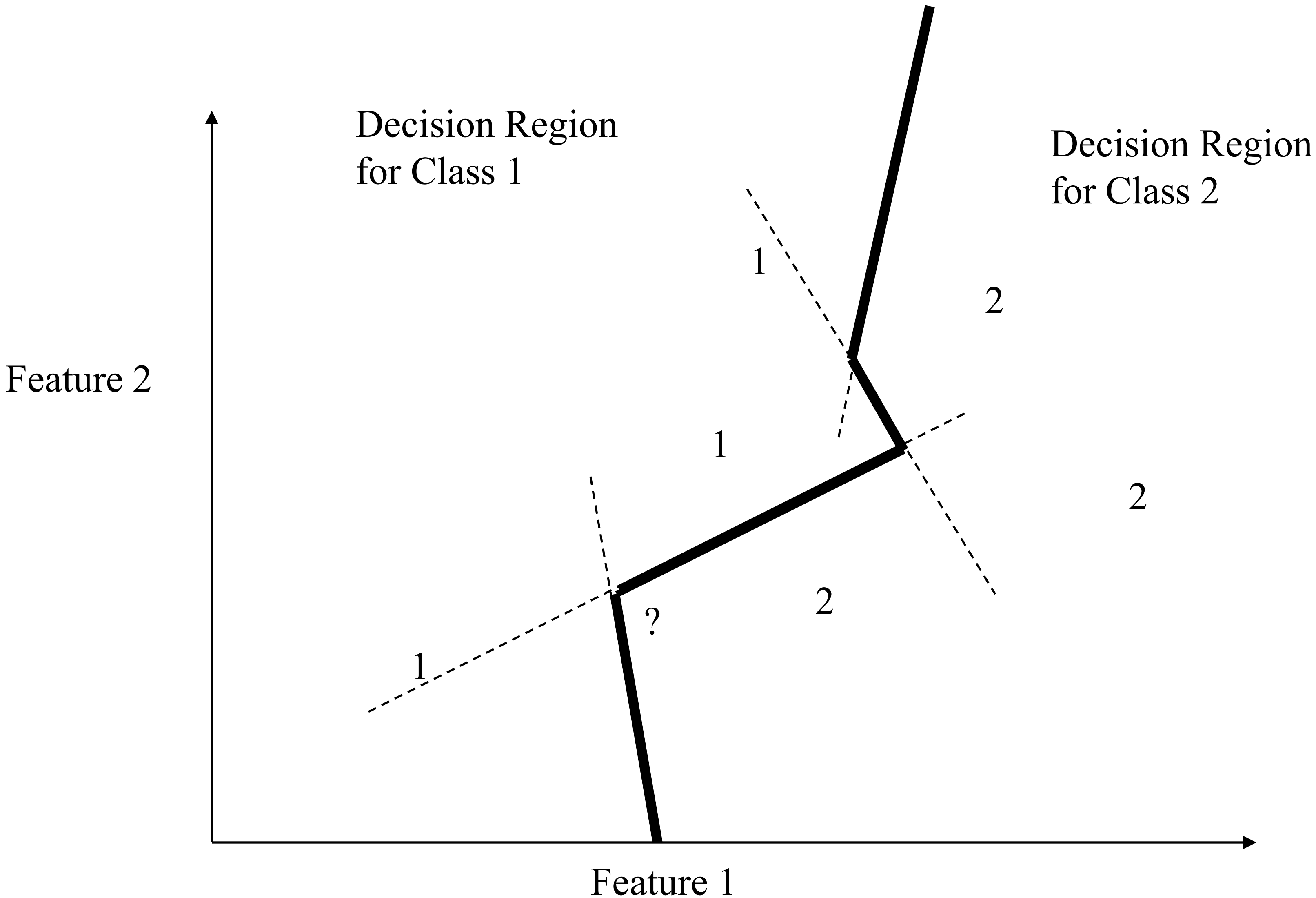


Image Courtesy: http://scott.fortmann-roe.com/docs/BiasVariance.html

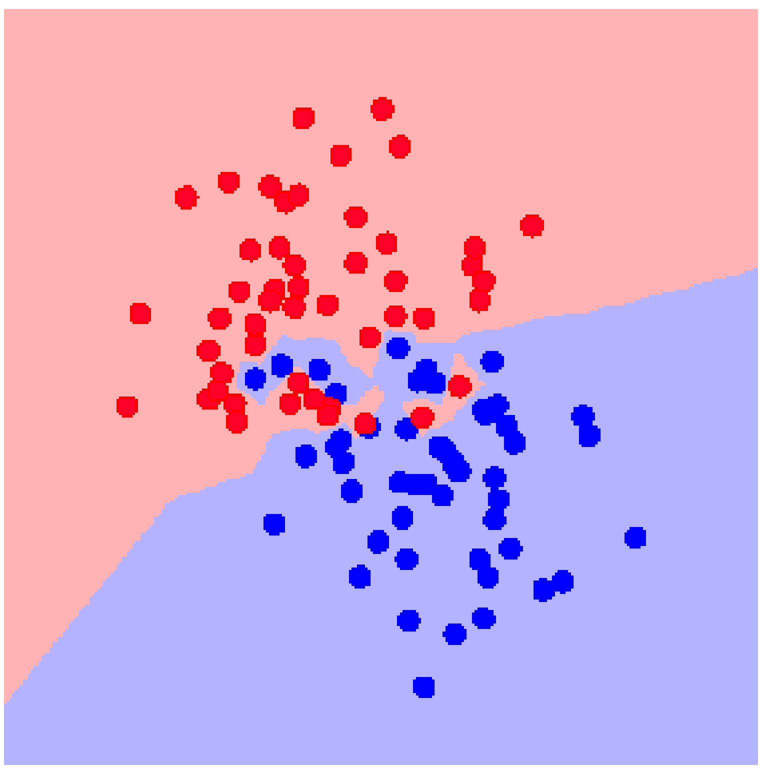
Overall Boundary = Piecewise Linear



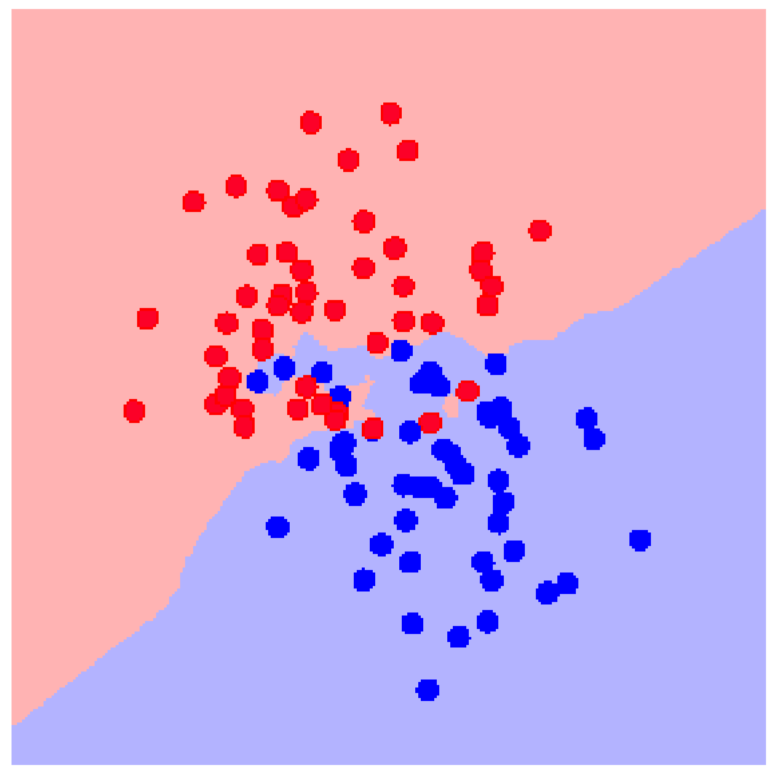
kNN Decision Boundary

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

K = 1



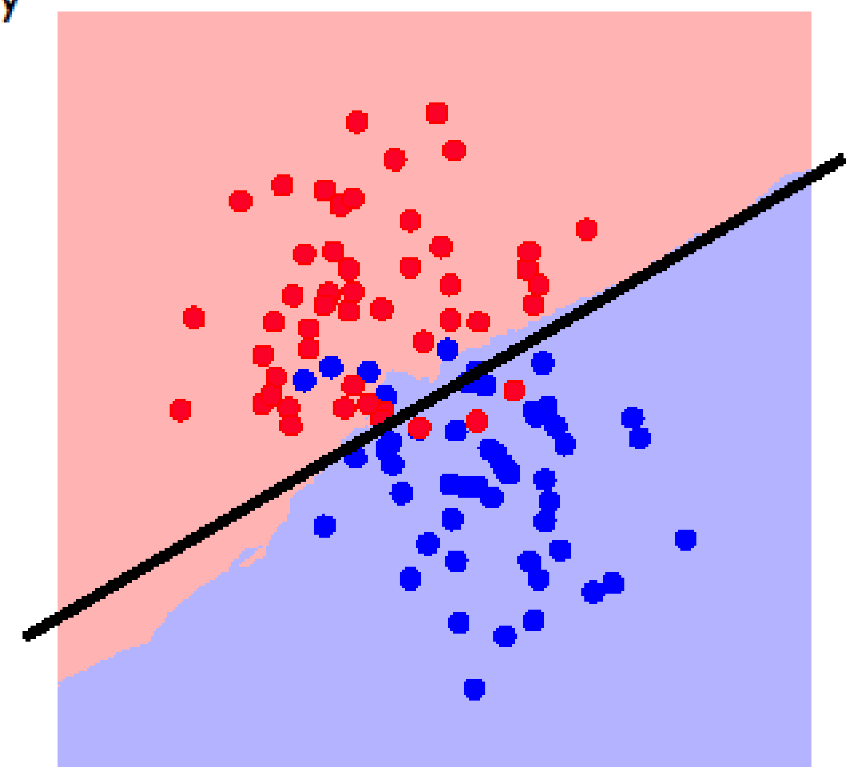
K = 3



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!

Larger K $\Longrightarrow$ Smoother boundary

Linear Classifiers

- Linear classifiers classification decision based on the value of a linear combination of the characteristics.
  - Linear decision boundary (single boundary for 2-class case)

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

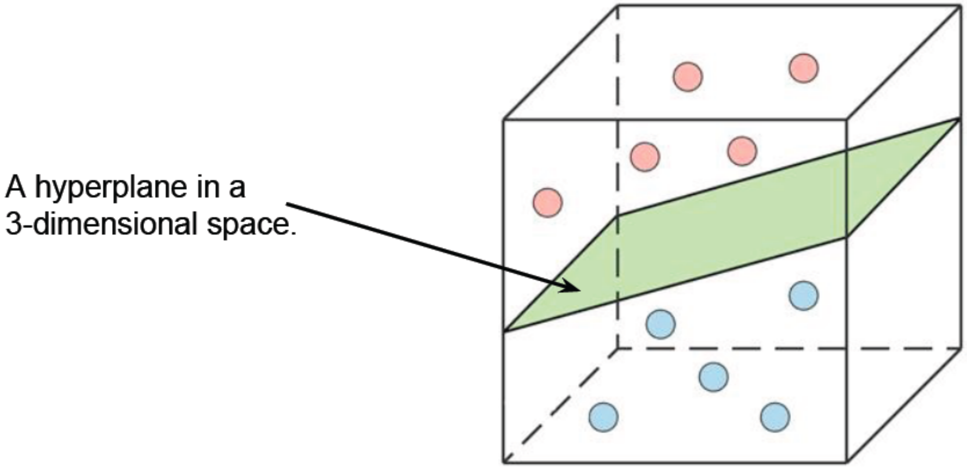
$$w_1 x_1 + w_2 x_2 + \ldots + w_d x_d = \sum_j w_j x_j = w^T x = 0$$

- The w_i are weights; the x_i are feature values

Linear Classifiers

$$w_1x_1 + w_2x_2 + \ldots + w_dx_d = \sum_j w_jx_j = w^Tx = 0$$

- This equation defines a hyperplane in d dimensions
  - A hyperplane is a subspace whose dimension is one less than that of its ambient space.
  - If a space is 3-dimensional, its hyperplanes are the 2-dimensional planes;
  - if a space is 2-dimensional, its hyperplanes are the 1-dimensional lines.



https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

Linear Classifiers

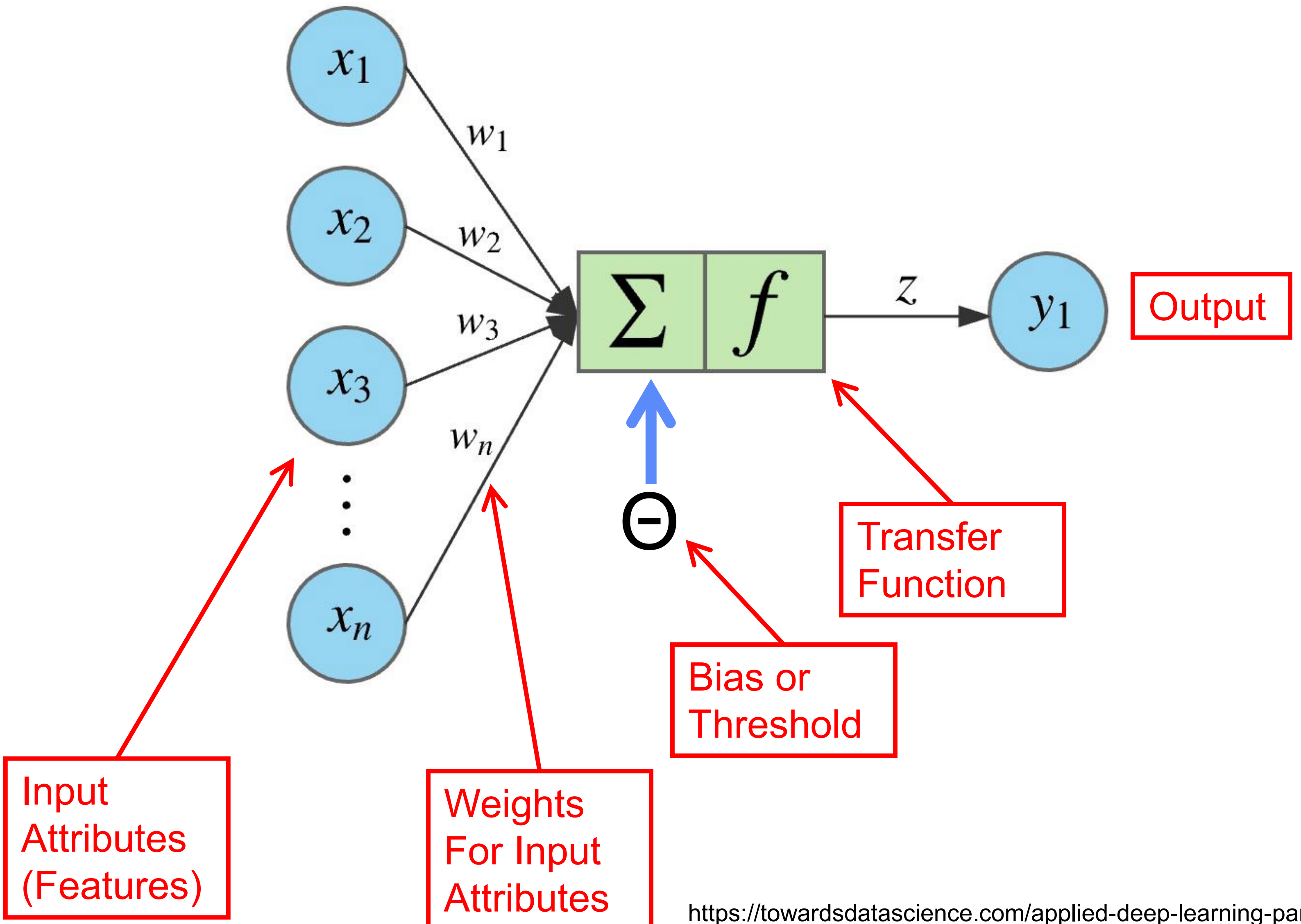
- For prediction we simply see if $\sum_j w_j x_j > 0$ for new data x.
  - If so, predict x to be positive
  - If not, predict x to be negative

- 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
  - The feature value is always 1.0
  - Its weight corresponds to (the negative of) the threshold

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

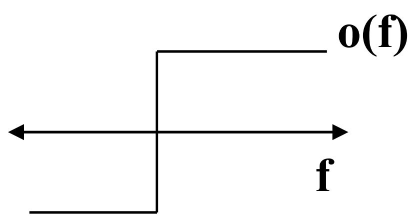
The Perceptron Classifier
(pages 729-731 in text)



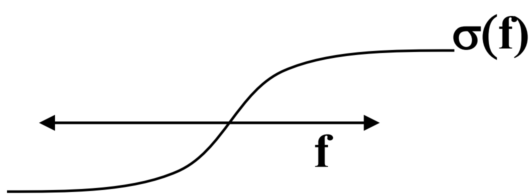
https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

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



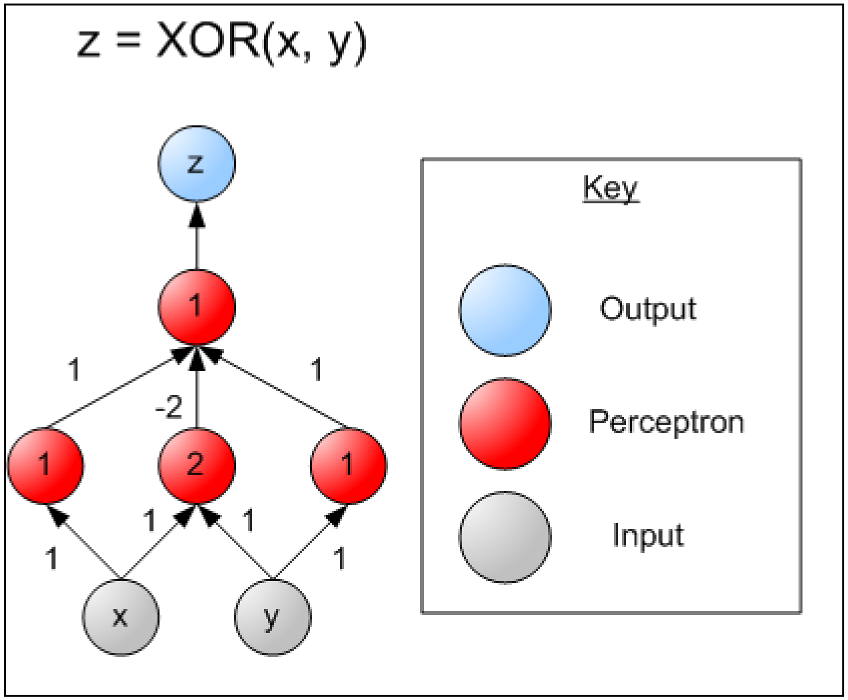
Sigmoid output, takes
real values between -1 and +1

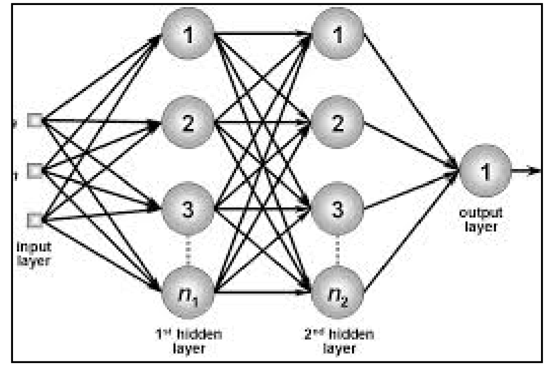
The sigmoid is in effect an approximation to the threshold function above, but has a gradient that we can use for learning

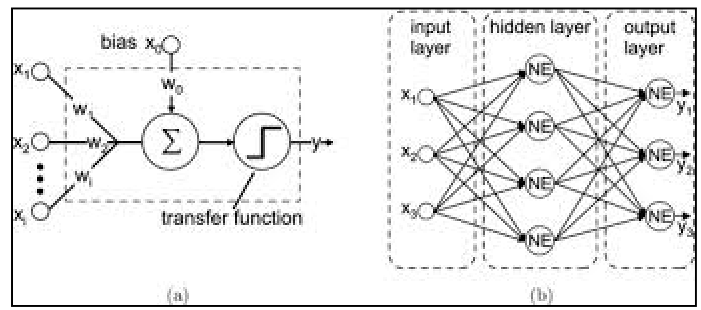
Sigmoid function is defined as

$$\sigma[\, f \,] = [\, 2 / (\, 1 + \exp[-\, f \,] \,) \,] - 1$$

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







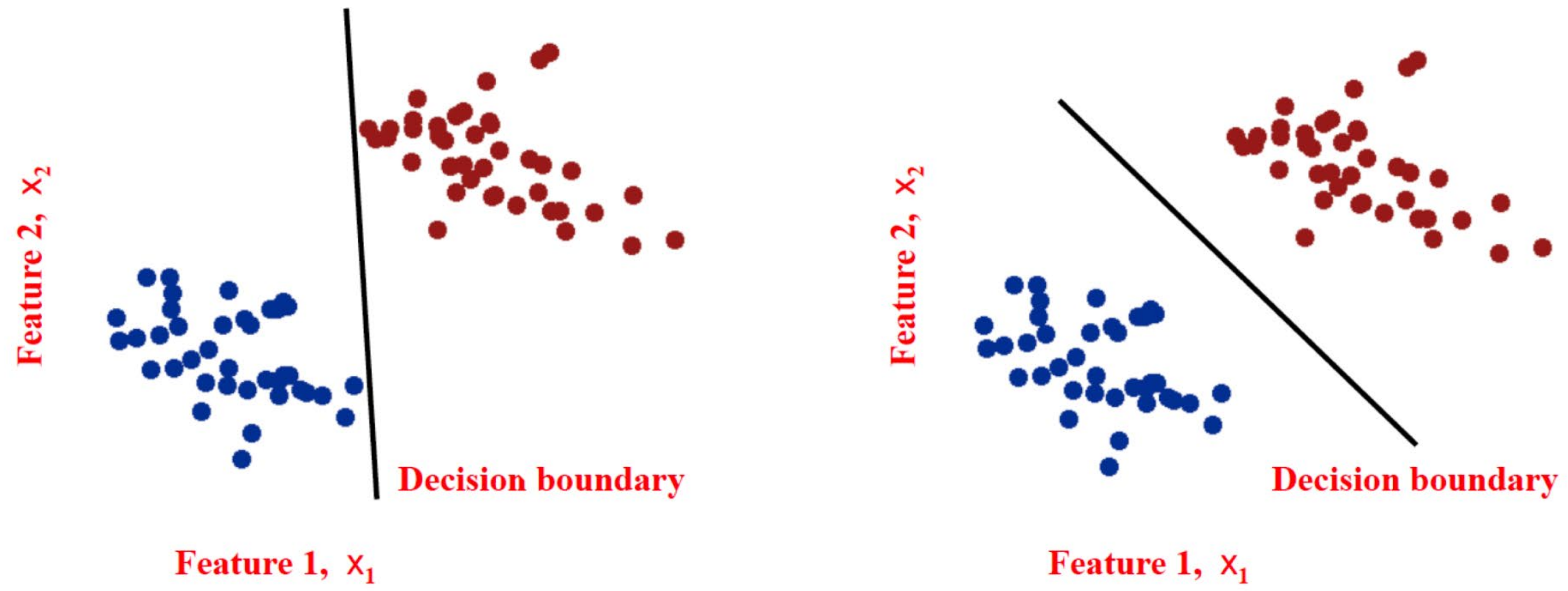
Multi-Layer Perceptrons (Artificial Neural Networks)
(sections 18.7.3-18.7.4 in textbook)

- 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

Which decision boundary is "better"?

- Both have zero training error (perfect training accuracy).
- But one seems intuitively better, more robust to error



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”

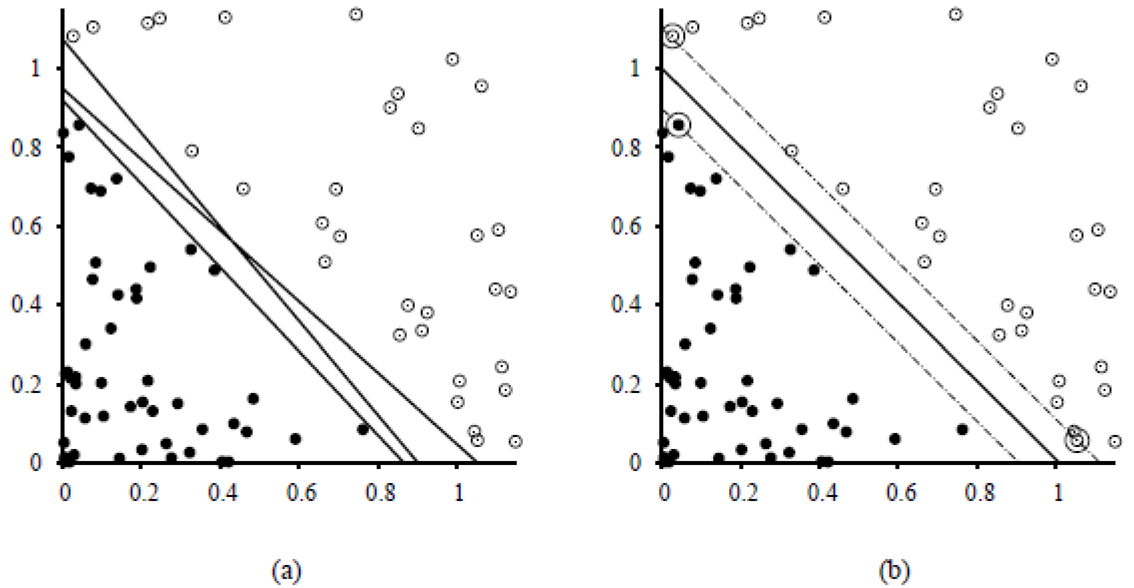


Figure 18.30 **FILES: .** Support vector machine classification: (a) Two classes of points (black and white circles) and three candidate linear separators. (b) The maximum margin separator (heavy line), is at the midpoint of the **margin** (area between dashed lines). The **support vectors** (points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

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

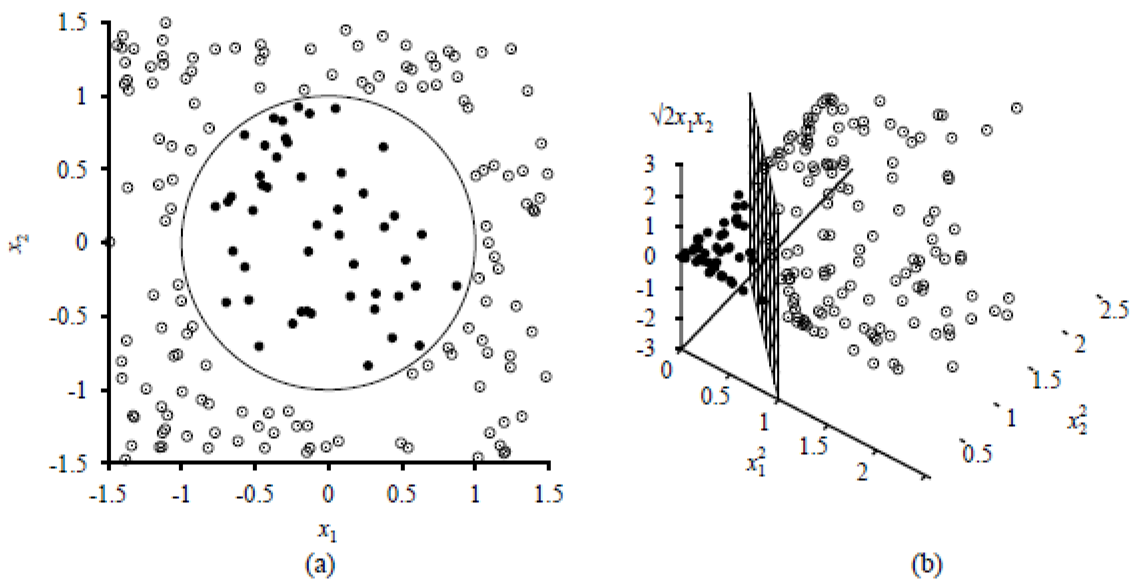
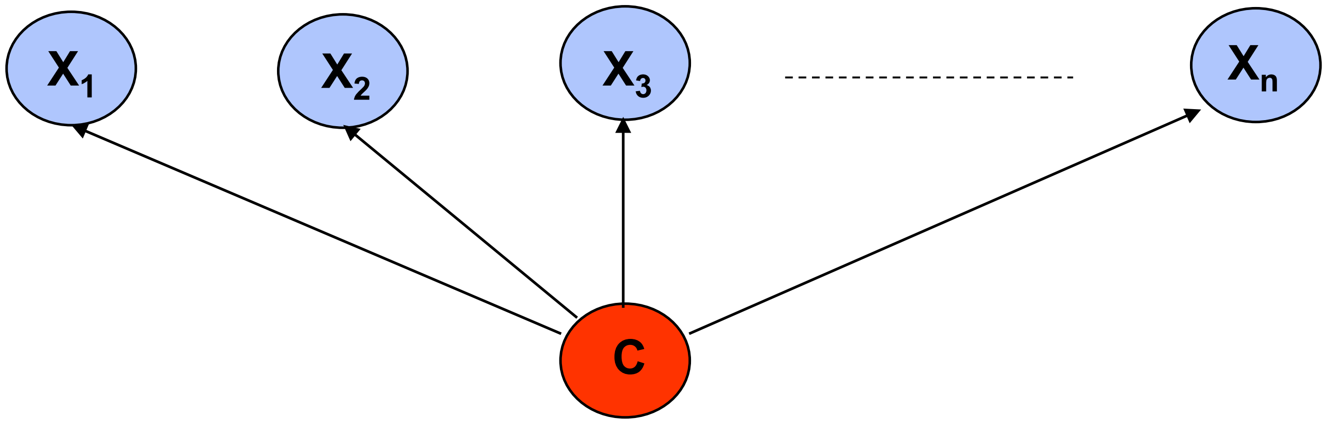


Figure 18.31 **FILES: .** (a) A two-dimensional training set with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$.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

Naïve Bayes Model (section 20.2.2 R&N 3rd ed.)



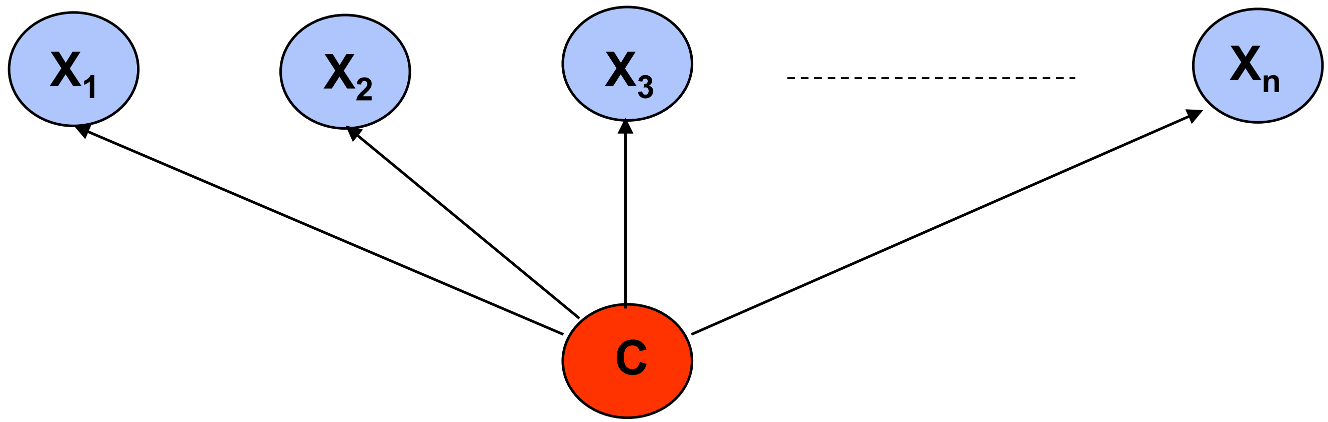
Basic Idea: We want to estimate $P(C \mid X_1,\ldots X_n)$, but it's hard to think about computing the probability of a class from input attributes of an example.

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 $\Pi_i \; P(X_i \mid C)$.

$$P(C \mid X_1,\ldots X_n) = P(C)\, P(X_1,\ldots X_n \mid C) \,/\, P(X_1,\ldots X_n) \propto P(C)\, \Pi_i \; P(X_i \mid C)$$

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.)



By 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)

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

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

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

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.

CS-171 Final Review

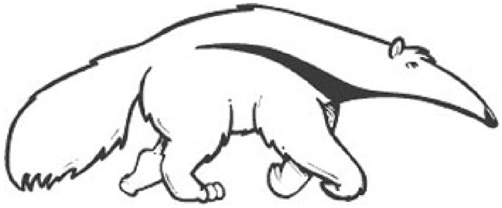
- **Machine Learning Classifiers**
  - (R&N Ch. 18.5-18.12; 20.2)
- **<u>Intro to Machine Learning</u>**
  - (R&N Ch. 18.1-18.4)
- **Game (Adversarial) Search**
  - (R&N Ch. 5.1-5.4)
- **Local Search**
  - (R&N Ch. 4.1-4.2)
- **State Space Search**
  - (R&N Ch. 3.1-3.7)
- Questions on any topic
- Please review your quizzes & old tests

Introduction to Machine Learning

CS171, Fall 2017

Introduction to Artificial Intelligence

TA Edwin Solares



Automated Learning

- Why learn?
  - Key to intelligence
  - Take real data → get feedback → improve performance → reiterate
  - USC Autonomous Flying Vehicle Project

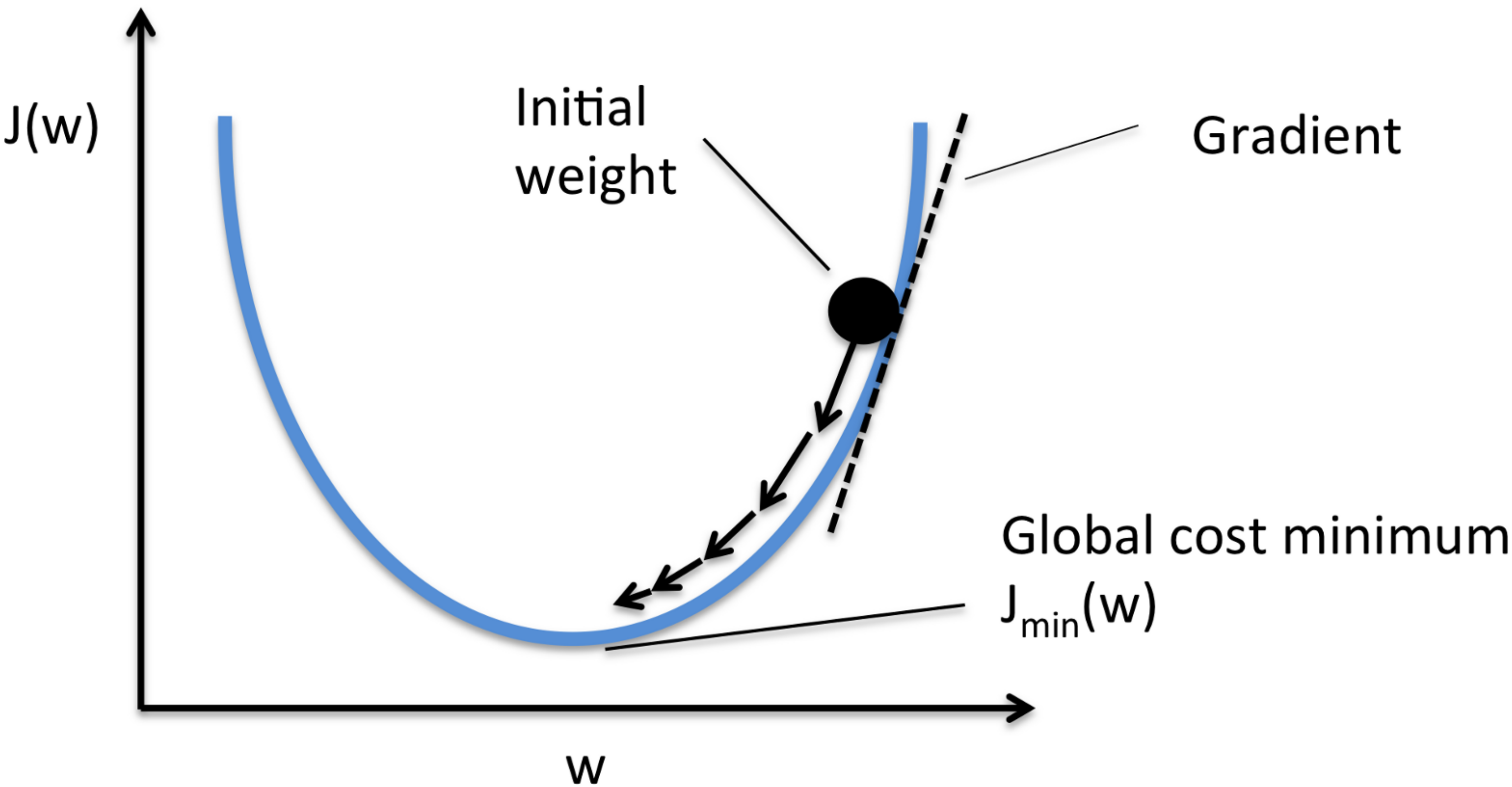
- Types of learning
  - Supervised learning: learn mapping: attributes → "target"
    - Classification: learn discreet target variable (e.g., spam email)
    - Regression: learn real valued target variable (e.g., stock market)

  - Unsupervised learning: no target variable; "understand" hidden data structure
    - Clustering: grouping data into K groups (e.g. K-means)
    - Latent space embedding: learn simple representation of the data (e.g. PCA, SVD)

  - Other types of learning
    - Reinforcement learning: e.g., game-playing agent
    - Learning to rank, e.g., document ranking in Web search
    - And many others….

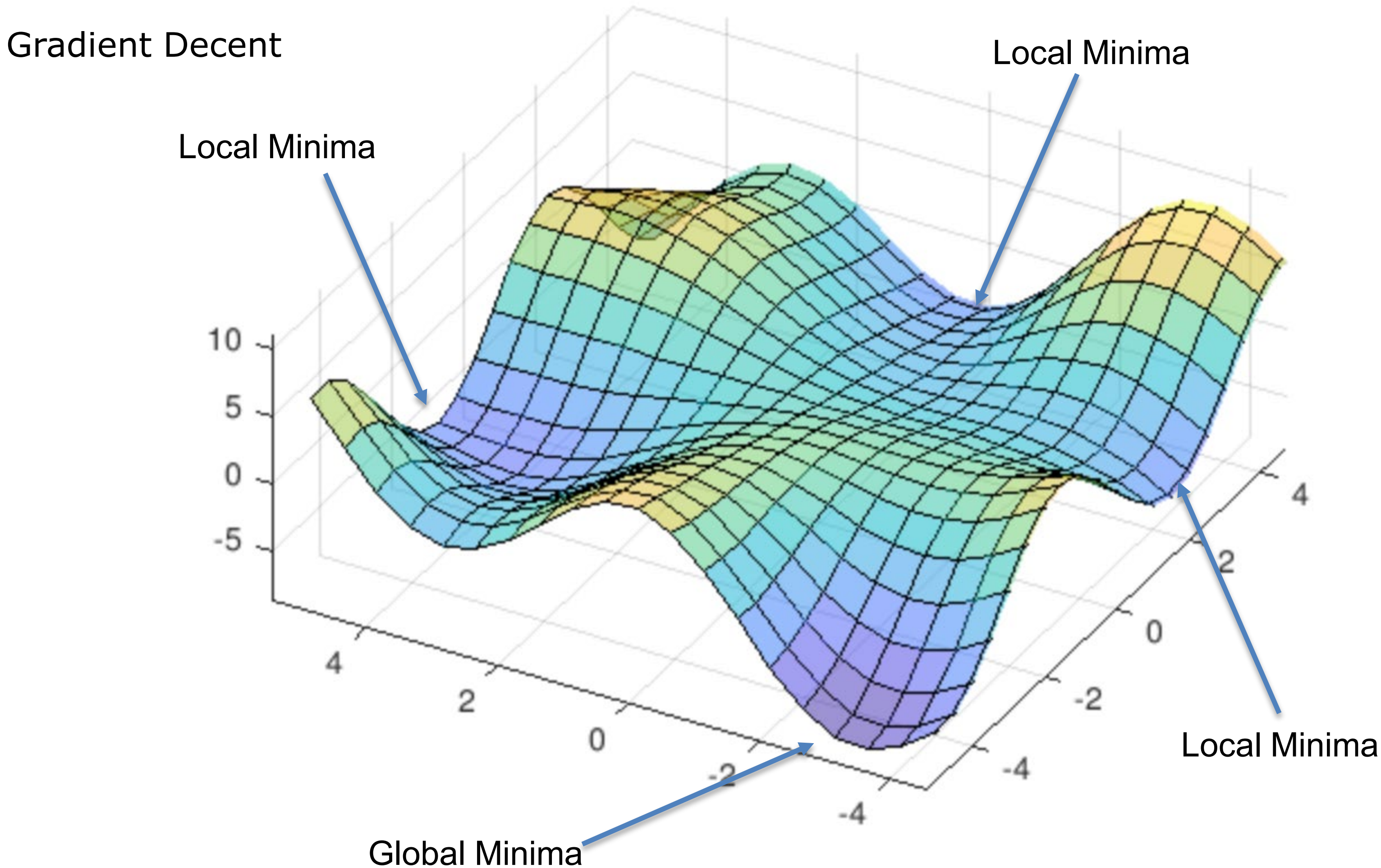
Minimization of Cost Function

Gradient Decent



Courtesy of Nvidia
Website

Minimization of Cost Function



Entertaining and informative way to learn about Neural Nets and Deep Learning
https://www.youtube.com/watch?v=p69khggr1Jo

Supervised Learning Terminology

- Attributes
  - Also known as features, variables, independent variables, covariates

- Target Variable
  - Also known as goal predicate, dependent variable, f(x), y …

- Classification
  - Also known as discrimination, supervised classification, …

- Error function
  - Objective function, loss function, …

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)
  - $h_k(x) = (x_1 \wedge x_2) \vee (x_3 \wedge \neg x_4)$

Inductive Learning as Optimization or Search

- Empirical error function:

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

- Empirical learning = finding h(x), or h(x, θ) that minimizes E(h)
  - In simple problems there may be a closed form solution
    - E.g., "normal equations" when h is a linear function of x, E = squared error
  - If E(h) is **differentiable** → continuous optimization problem using gradient descent, etc
    - E.g., multi-layer neural networks
  - If E(h) is **non-differentiable** (e.g., classification → systematic search problem through the *space of functions h*
    - E.g., decision tree classifiers

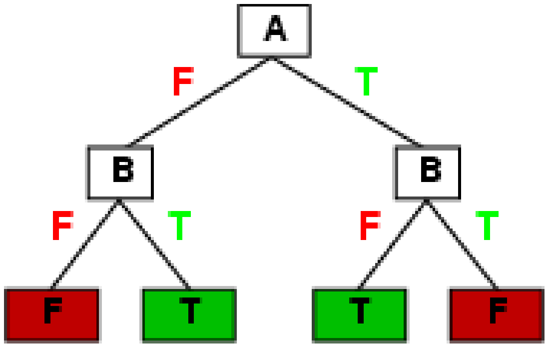
- Once we decide on what the functional form of h is, and what the error function E is, then machine learning typically reduces to a large search or optimization problem

- Additional aspect: we really want to learn a function h that will generalize well to new data, not just memorize training data – will return to this later

Decision Tree Representations

- Decision trees are fully expressive
  - can represent any Boolean function
  - Every path in the tree could represent 1 row in the truth table
  - Yields an exponentially large tree
    - Truth table with $\boldsymbol{2^d}$ rows, where $\boldsymbol{d}$ is the number of attributes

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



Decision Tree Representations

- Decision trees are DNF representations
  - often used in practice → result in compact approximate representations for complex functions
  - E.g., consider a truth table where most of the variables are irrelevant to the function

- Simple DNF formulae can be easily represented
  - E.g., $f = (A \wedge B) \vee (\neg A \wedge D)$
  - DNF = disjunction of conjunctions

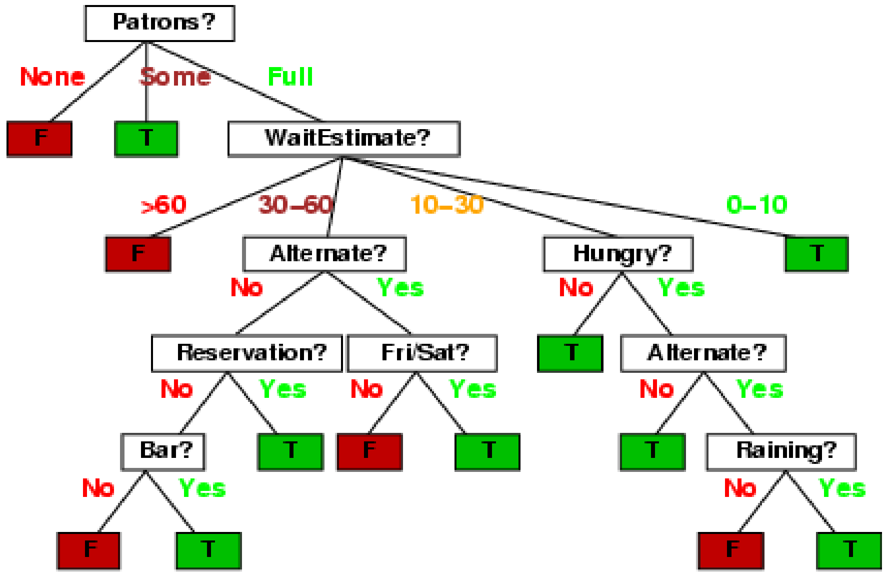
- Trees can be very inefficient for certain types of functions
  - Parity function: 1 only if an even number of 1's in the input vector
    - Trees are very inefficient at representing such functions
  - Majority function: 1 if more than ½ the inputs are 1's
    - Also inefficient

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
```

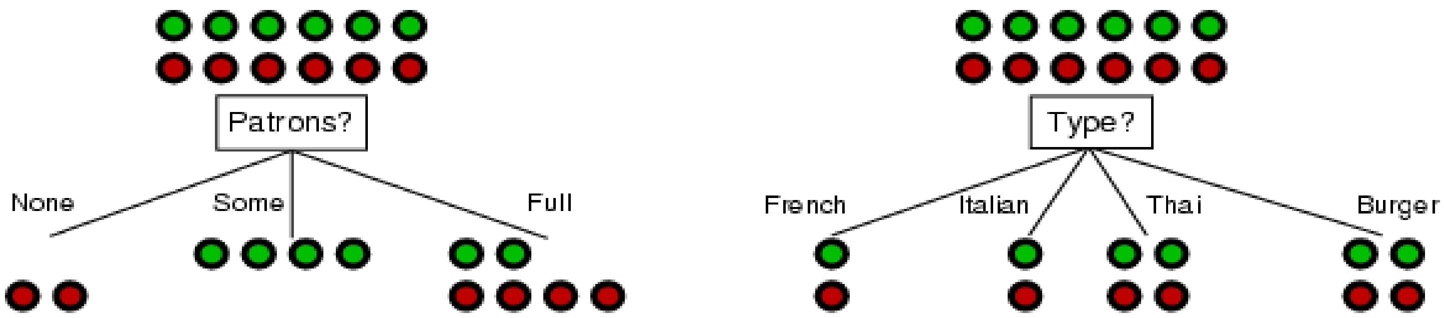
Decision Tree: Book Example

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*Wait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T



Choosing an attribute

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

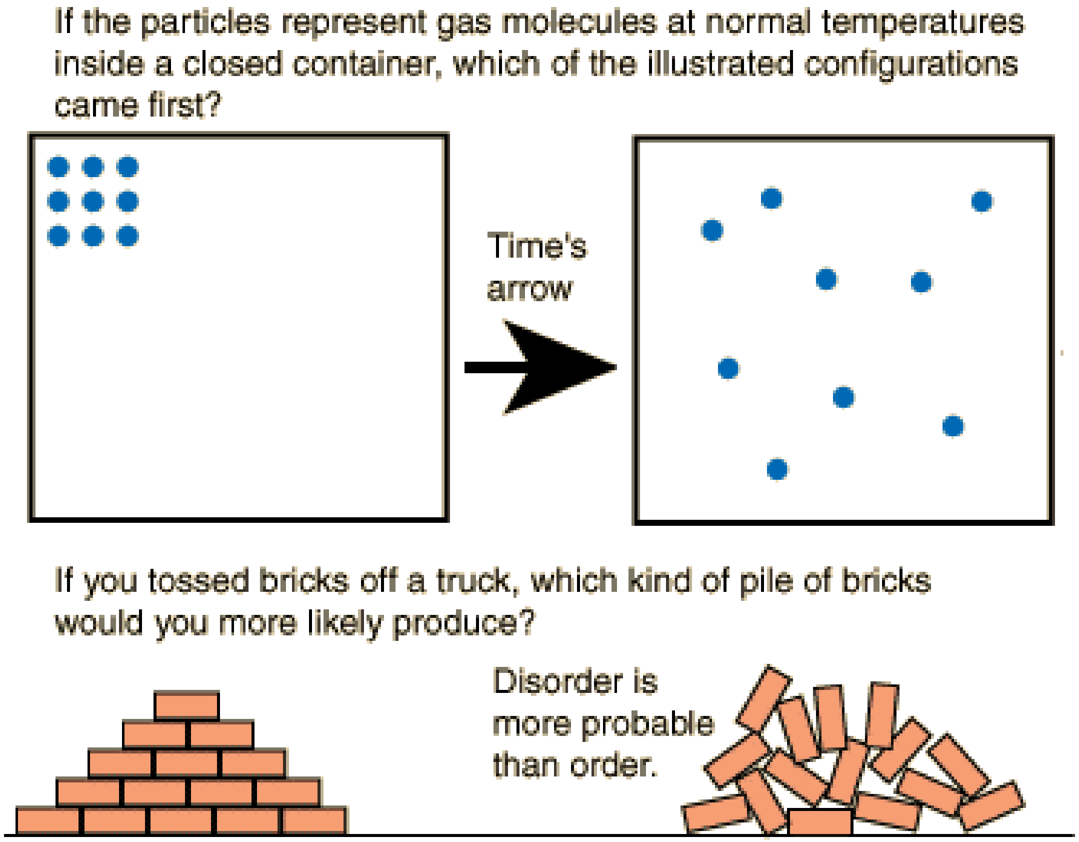


- *Patrons?* is a better choice
  - How can we quantify this?
  - One approach would be to use the classification error E directly (greedily)
    - Empirically it is found that this works poorly
  - Much better is to use information gain (next slides)

Entropy and Information

- "Entropy" is a measure of randomness

- In chemistry:



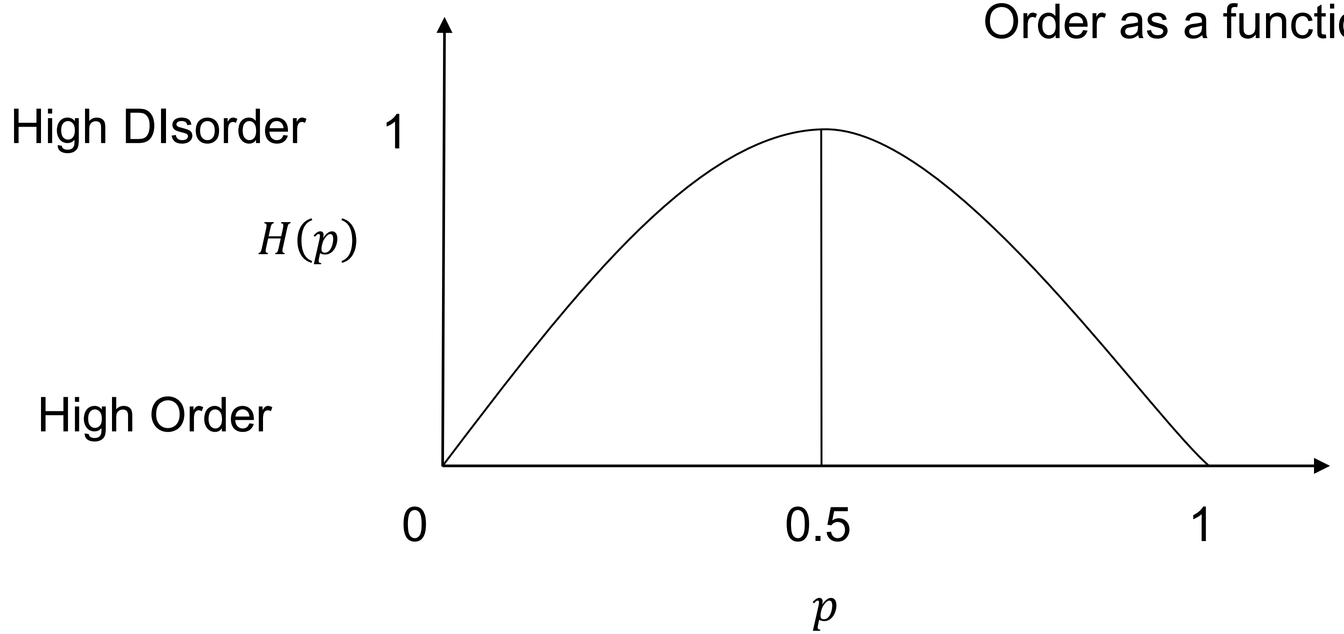
https://www.youtube.com/watch?v=ZsY4WcQOrfk

Entropy with only 2 outcomes

In binary case (2 outcomes)

$$H(p) = -p\, log_2(p) - (1-p)log_2(1-p)$$

Order as a function of *p*



For multiple outcomes we have $\max H(p) = -\log_2\left(\frac{1}{n}\right) = \log_2(n)$

Information Gain

- $H(p)$ = entropy of class distribution at a particular node

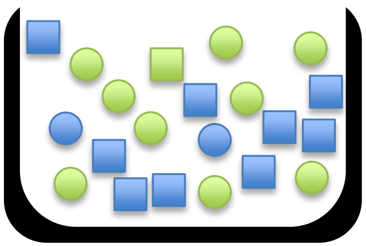
- $H(p \mid A)$ = conditional entropy
  - Weighted average entropy of conditional class distribution
  - Partitioned the data according to the values in A
  - The sum of each partition given the group/class

- $Gain(A) = H(p) – H(p \mid A)$

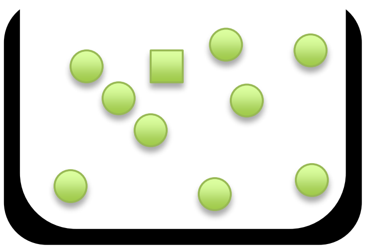
- Simple rule in decision tree learning
  - At each internal node, split on the node with the largest information gain (or equivalently, with smallest $H(p|A)$)

- Note that by definition, conditional entropy can't be greater than the entropy

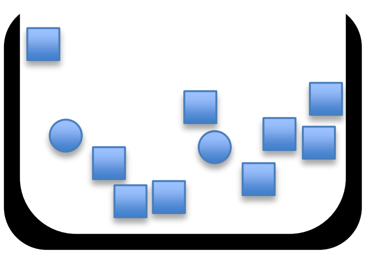
Entropy Example



10 Squares
11 Circles



1 Square
9 Circles



9 Square
2 Circles

Class = color and shape
grn = green; blu = blue; sq = square

- $H(p_{sq}) = -\frac{10}{21}\log_2\frac{10}{21} - \frac{11}{21}\log_2\frac{11}{21}$
- $H(p_{sq}) = 0.998$

- $H(p_{sq}|color) = p_{blu}H(p_{sq})_{blu} + p_{grn}H(p_{sq})_{grn}$

- $H(p_{sq})_{blu} = -\frac{9}{11}\log_2\frac{9}{11} - \frac{2}{11}\log_2\frac{2}{11}$
- $H(p_{sq})_{blu} = 0.684$
- $H(p_{sq})_{grn} = 0.469$

- $Gain(color) = H(p_{sq}) - H(p_{sq}|color)$

Weighted average →

- $H(p_{sq}|color) = \frac{10}{21} * 0.469 + \frac{11}{21} * 0.684$
- $H(p_{sq}|color) = 0.582$
- $Gain(color) = 0.998 - 0.582$
- $Gain(color) = 0.416$

Formulas

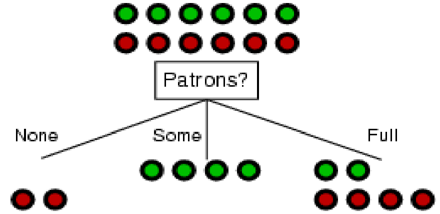
- Entropy
  - $H(shape) = -p(sq) * log_2\big(p(sq)\big) - (1 - p(sq)log_2(1 - p(sq))$
- Conditional entropy
  - $H(shape|color) = p(gr)H(shape|gr) + p(blu) * H(shape|blu)$
  - $H(shape|color) = p(gr)H(shape|gr) + p(\neg gr)H(shape|\neg gr)$
- Information Gain
  - *IG(shape) = H(shape) – H(shape / color)*

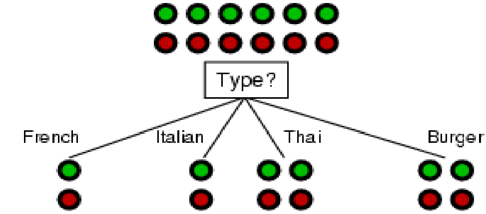
Minimize Entropy

Maximize Information Gain

We want a low value for conditional entropy → high order

Root Node Example

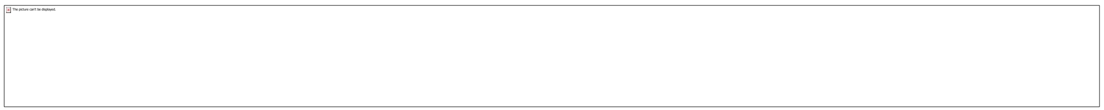




For the training set, *6 positives, 6 negatives, H(6/12, 6/12) = 1* bit

positive (p)

negative (1-p)



Consider the attributes *Patrons* and *Type:*

$$IG(\text{Patrons}) = 1 \; - \; [\frac{2}{12}H(0,1) + \frac{4}{12}H(1,0) + \frac{6}{12}H(\frac{2}{6},\frac{4}{6})] \quad = 0.541 \text{ bits}$$

$$IG(\text{Type}) = 1 - [\frac{2}{12}H(\frac{1}{2},\frac{1}{2}) + \frac{2}{12}H(\frac{1}{2},\frac{1}{2}) + \frac{4}{12}H(\frac{2}{4},\frac{2}{4}) + \frac{4}{12}H(\frac{2}{4},\frac{2}{4})] \quad = 0 \text{ bits}$$

Conclude:

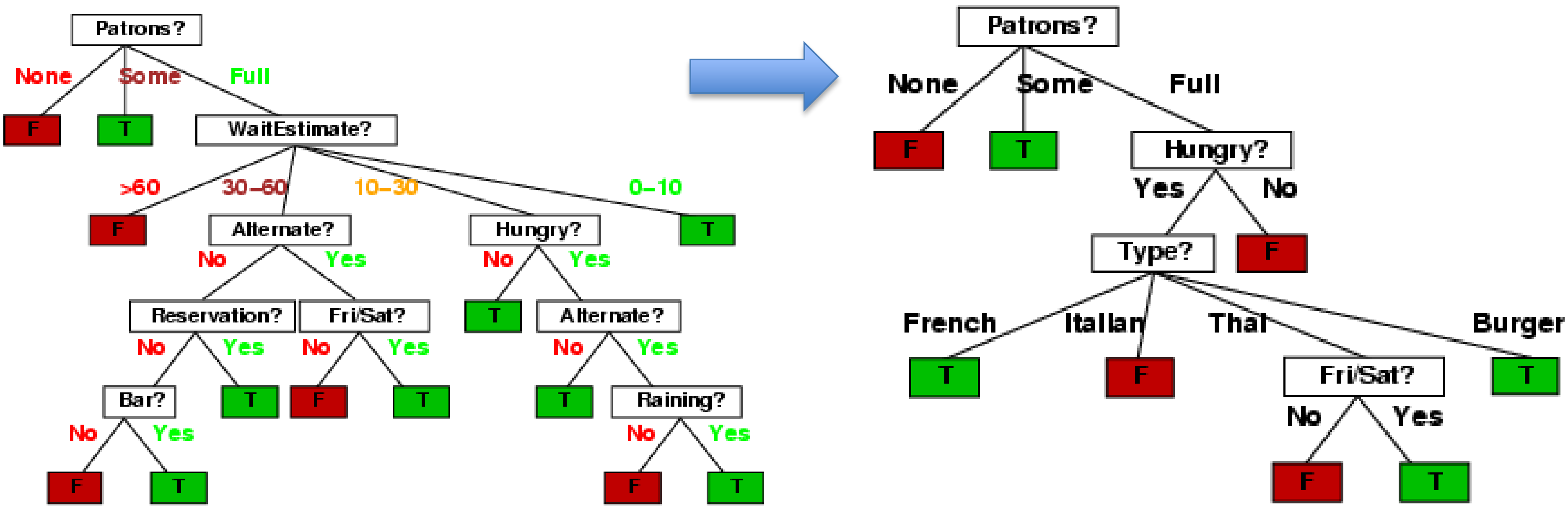
Patrons has the highest IG of all attributes and so is chosen by the learning algorithm as the root

Information gain is then repeatedly applied at internal nodes until all leaves contain only examples from one class or the other

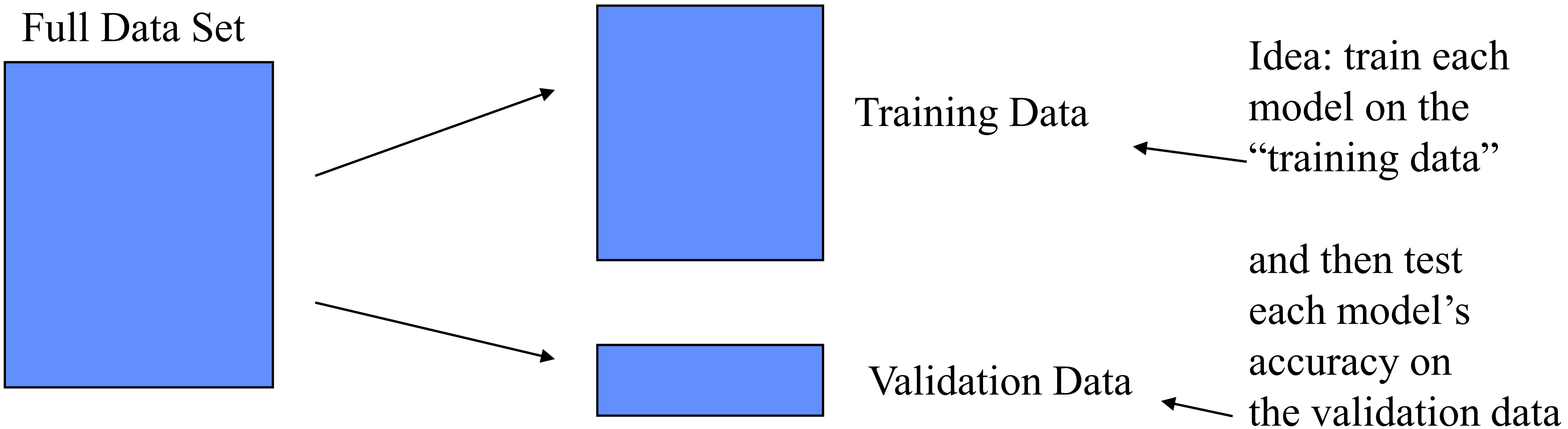
Decision Tree Learned

Authors Created

Learned



Assessing Performance: Training and Validation Data



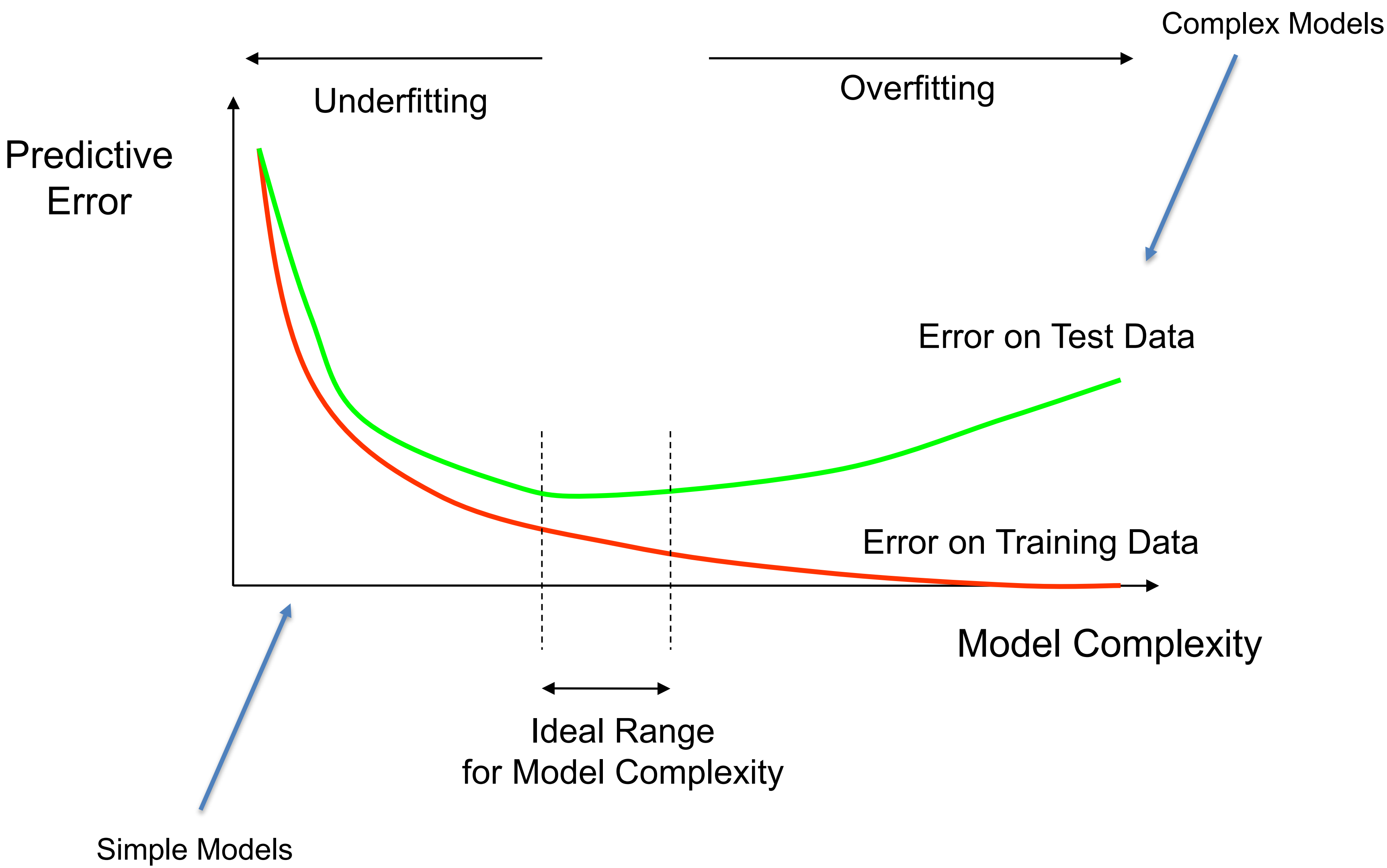
Training data performance is typically optimistic

- e.g., error rate on training data

With large data sets we can partition our data into 2 subsets, train and test

- build a model on the training data
- assess performance on the test data

How Overfitting affects Prediction

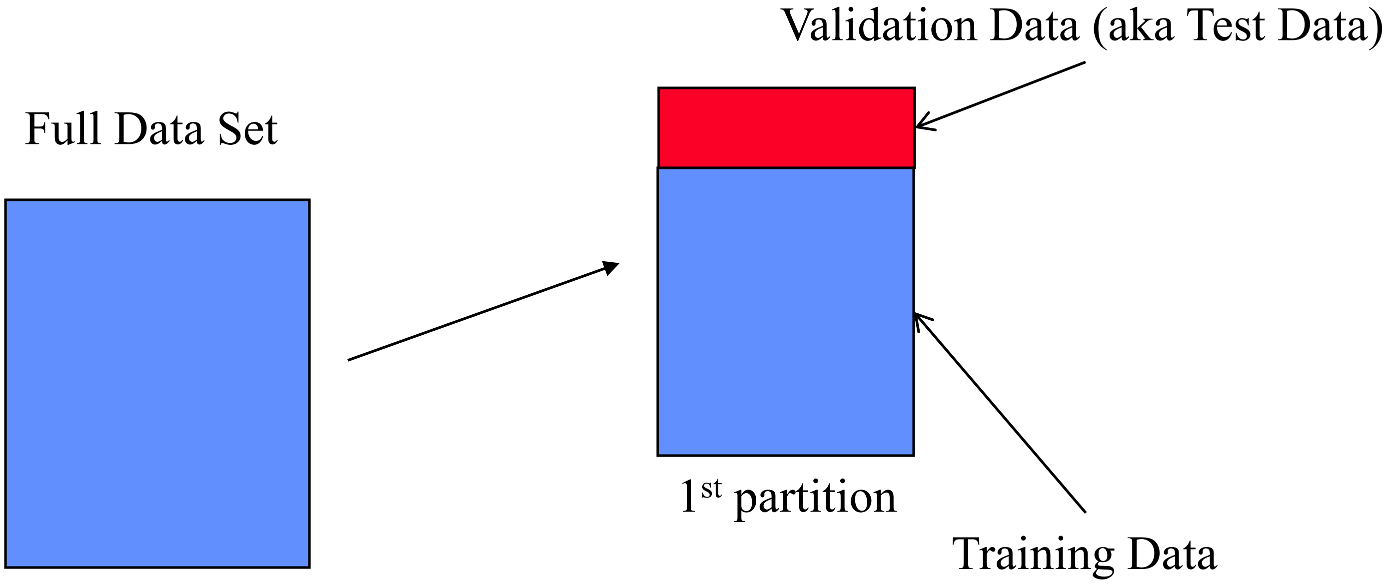


The k-fold Cross-Validation Method

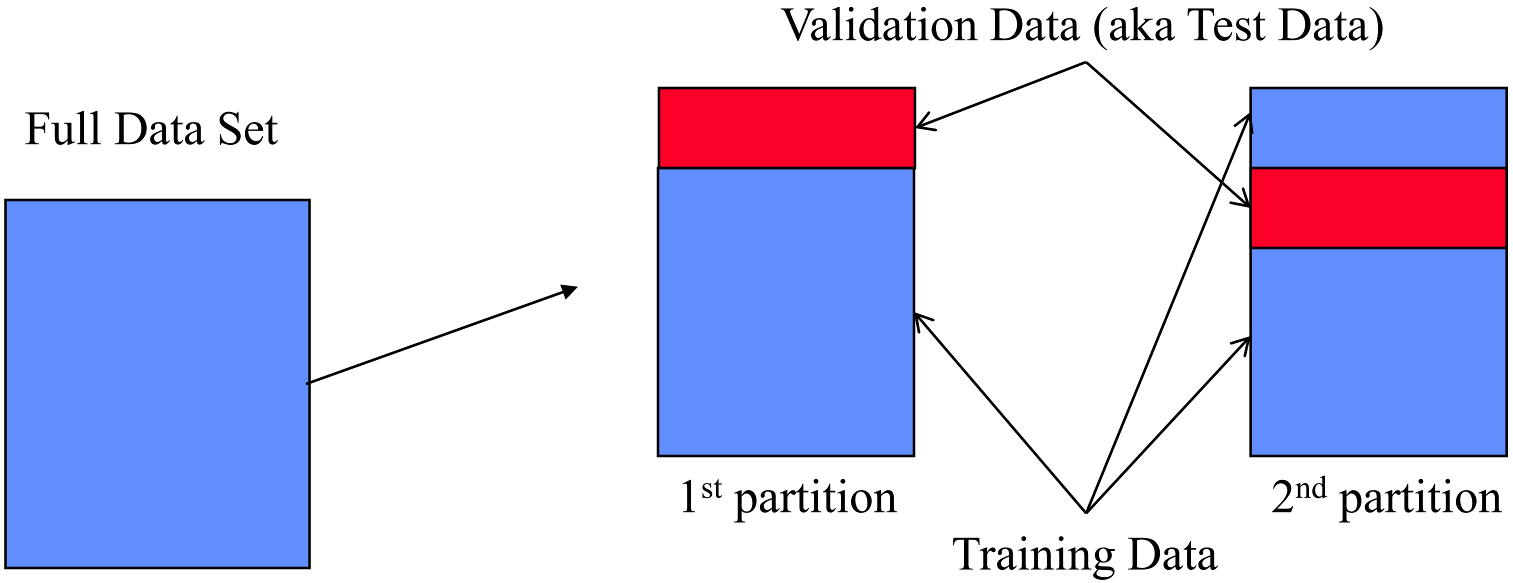
- Why stop at a 90/10 “split” of the data?
  - In principle we could do this multiple times

- “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:k (where k = 10)*
      - *train on 90% of the i_{th} data subset*
      - *Accuracy[i] = accuracy on 10% of the i_{th} data subset*
    - *end*
    - Cross-Validation-Accuracy = 1/k Σ_i Accuracy[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

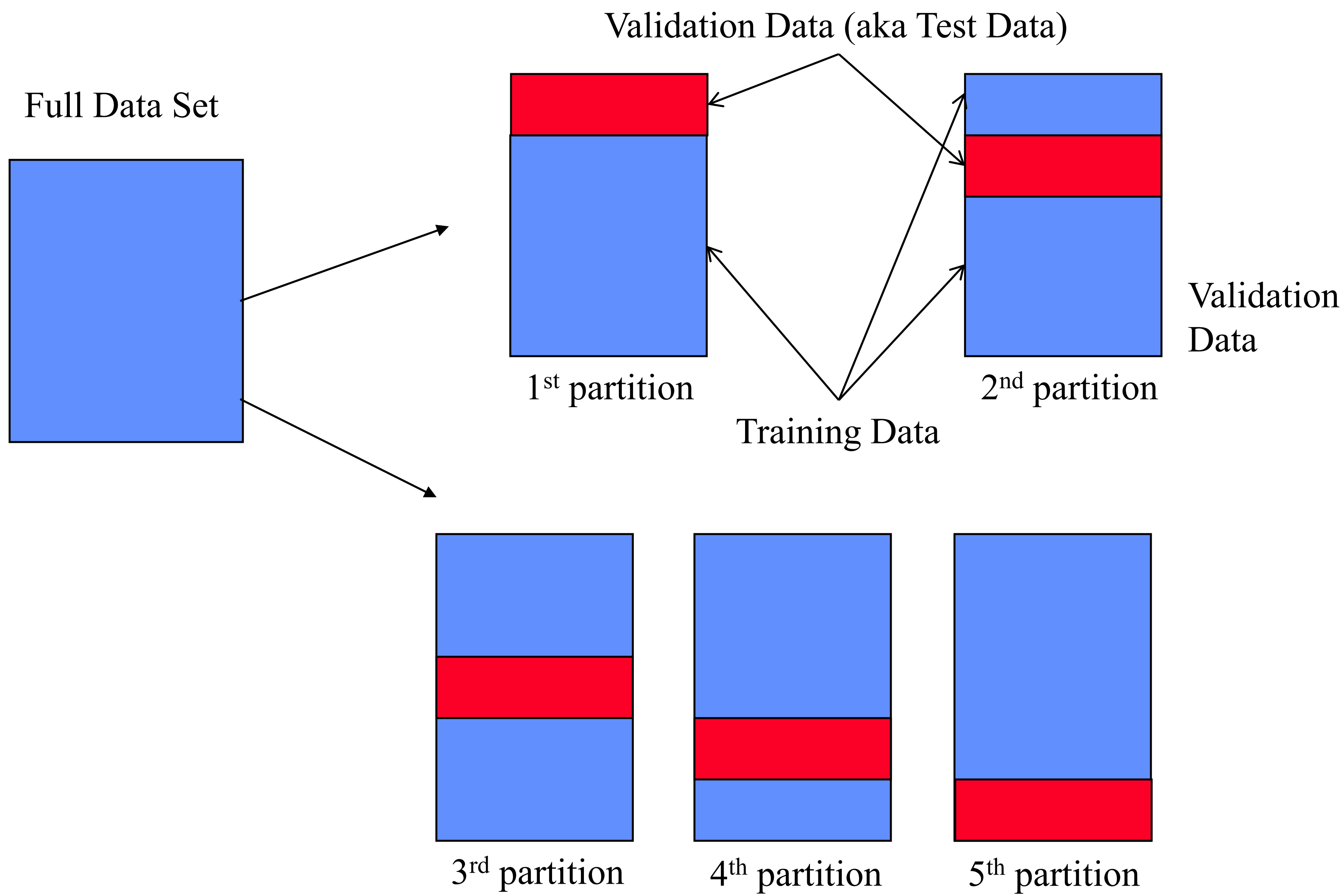
Disjoint Validation Data Sets for k = 5



Disjoint Validation Data Sets for k = 5



Disjoint Validation Data Sets for k = 5



You will be expected to know

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

CS-171 Final Review

- **Machine Learning Classifiers**
  - (R&N Ch. 18.5-18.12; 20.2)
- **Intro to Machine Learning**
  - (R&N Ch. 18.1-18.4)
- **<u>Game (Adversarial) Search</u>**
  - (R&N Ch. 5.1-5.4)
- **Local Search**
  - (R&N Ch. 4.1-4.2)
- **State Space Search**
  - (R&N Ch. 3.1-3.7)
- Questions on any topic
- Please review your quizzes & old tests

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.
  - (**2nd 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.

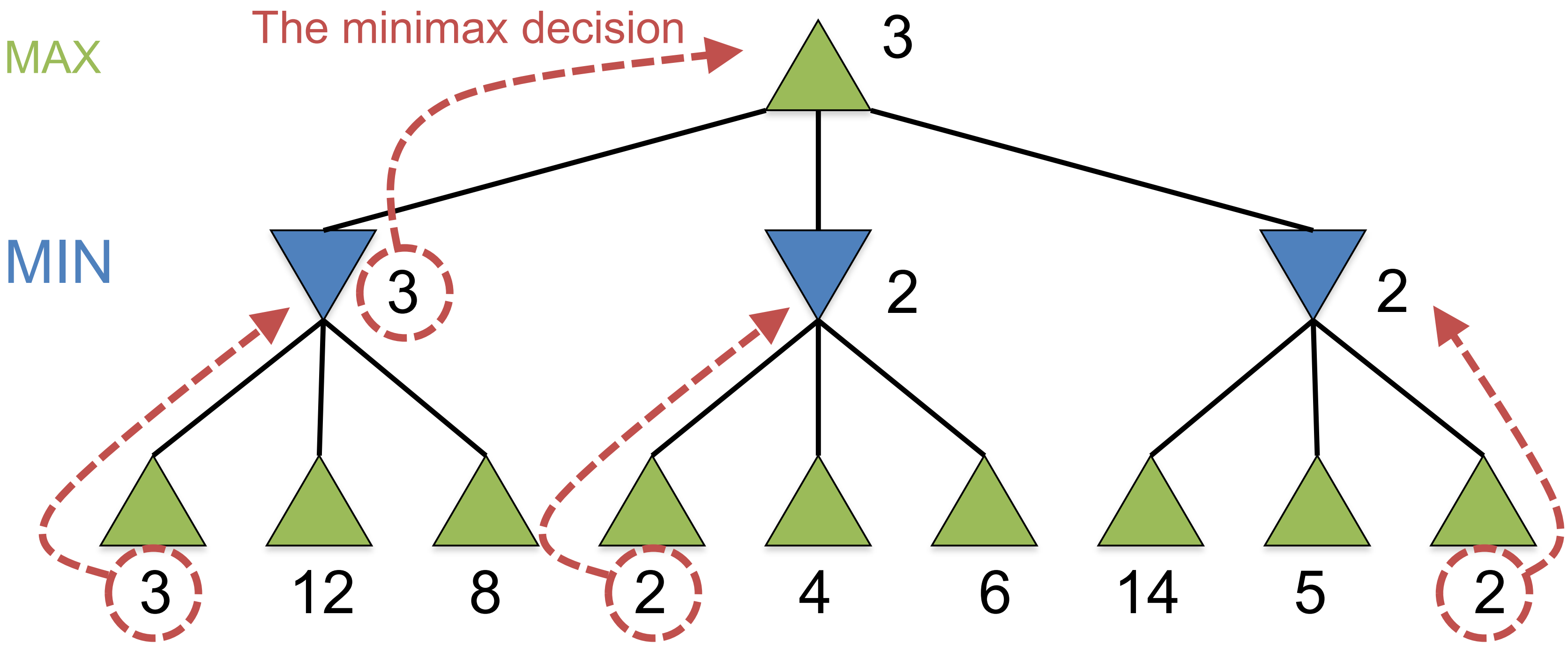
- MAX uses search tree to determine “best” next move.

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
  $\boxed{\textbf{return} \text{ arg max}_{a \in \text{ACTIONS}(state)} \text{ MIN-VALUE}(\text{Result}(state, a}))$

---

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

---

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

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

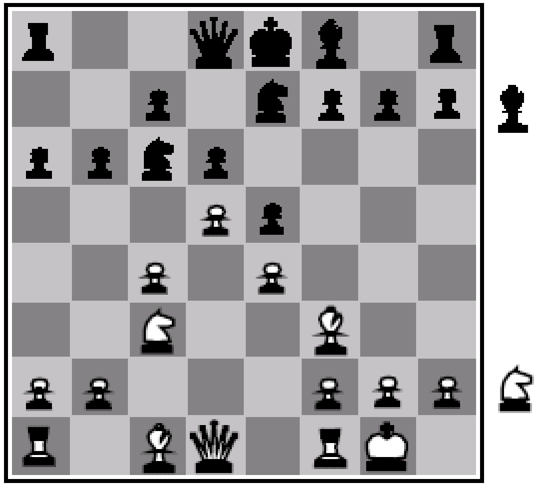
4-ply lookahead is a hopeless chess player!

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

Static (Heuristic) Evaluation Functions

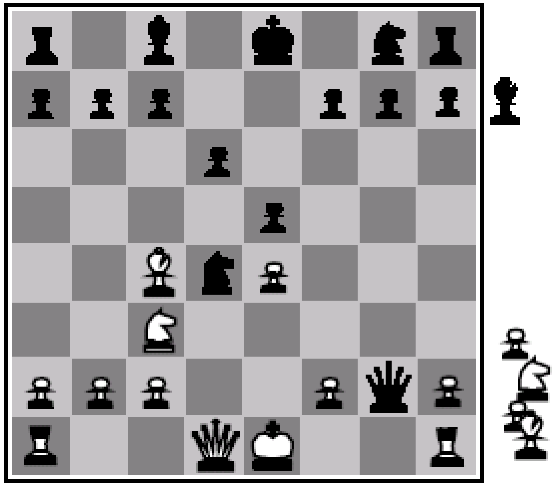
- An Evaluation Function:
  - Estimates how good the current board configuration is for a player.
  - Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
  - Othello: Number of white pieces - Number of black pieces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

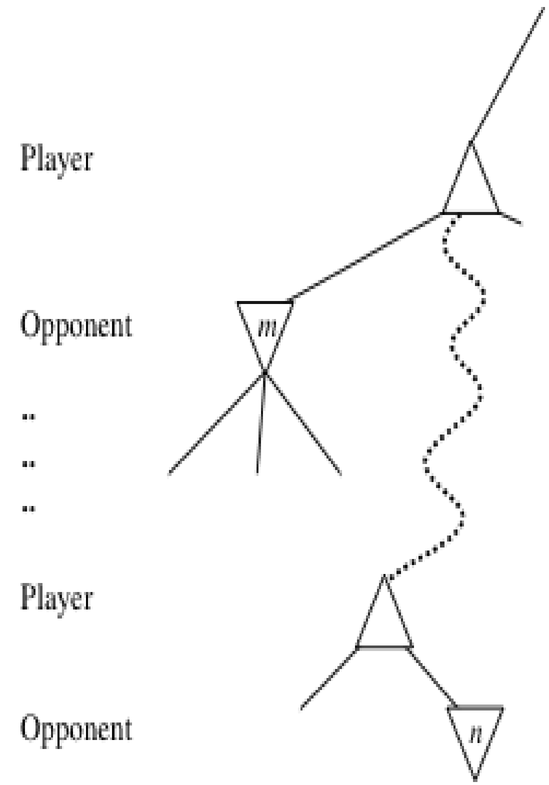
For chess, typically *linear* weighted sum of features

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

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, $\alpha = -$infinity)

β = lowest-value choice found at any choice point of path for MIN

(initially, $\beta = +$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
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty$, $+\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE(*state*, α , β) **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?

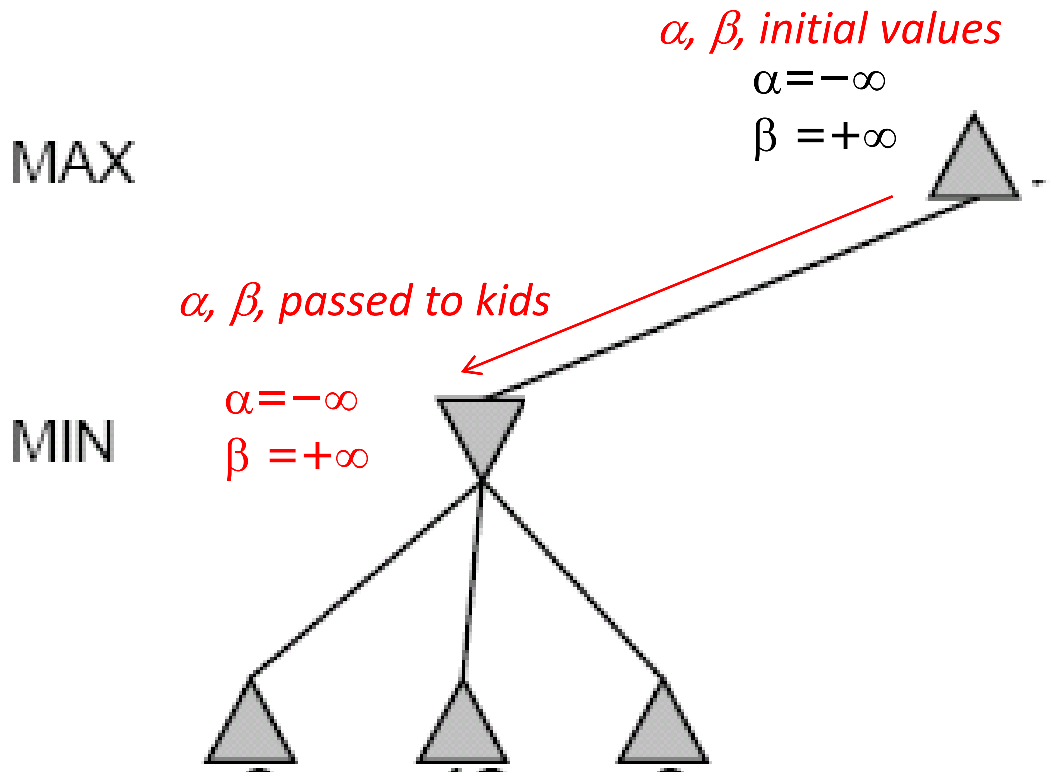
- **<u>Prune whenever $\alpha \geq \beta$.</u>**
  - Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
    - **<u>Max nodes update alpha</u>** based on children's returned values.
  - Prune below a Min node whose beta value becomes less than or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

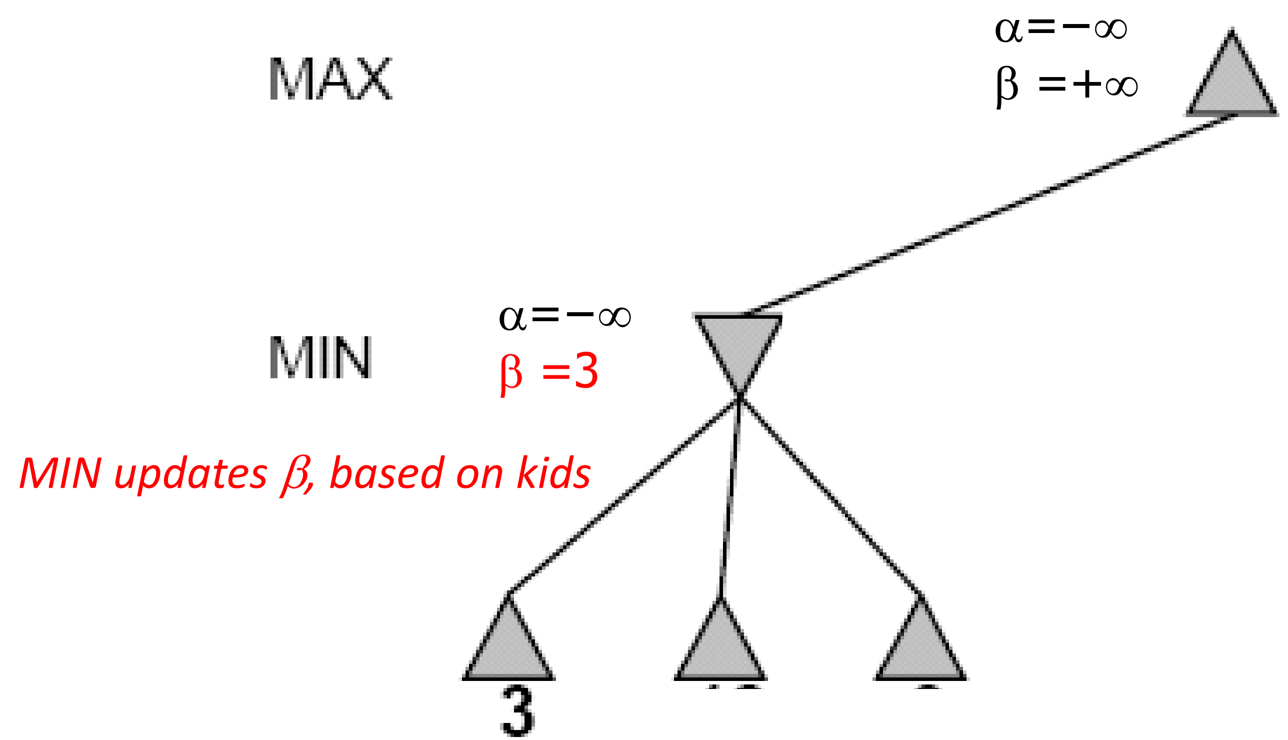
Alpha-Beta Example Revisited

Do DF-search until first leaf

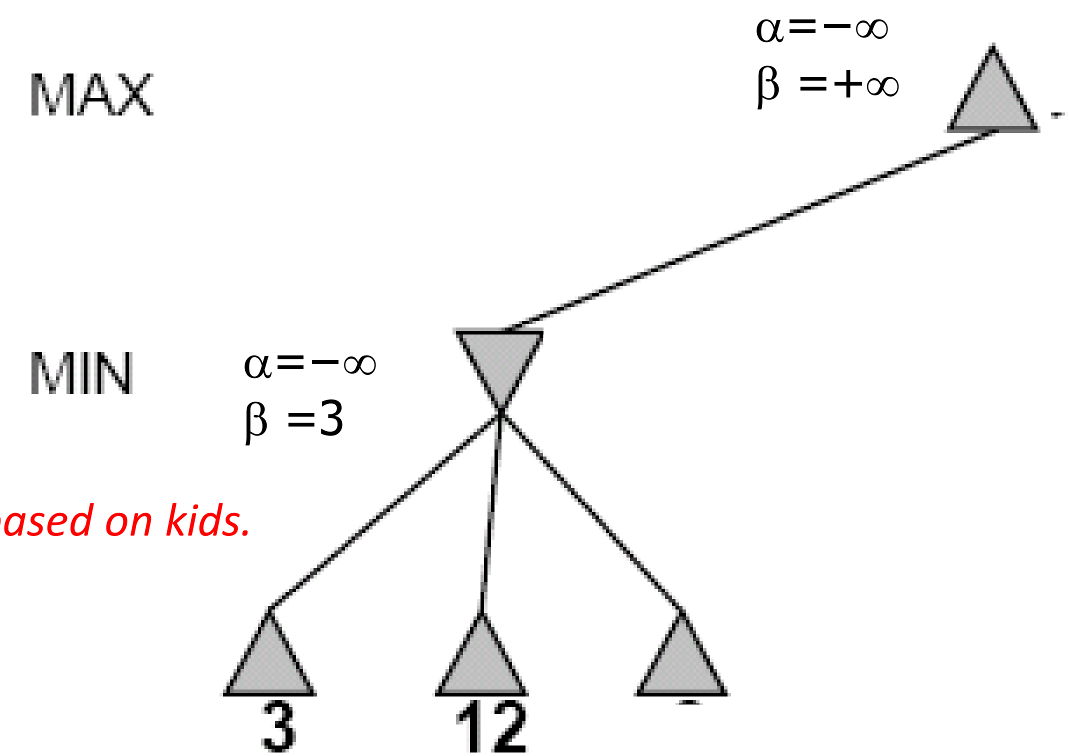


<u>**Review Detailed Example of Alpha-Beta Pruning in lecture slides.**</u>

Alpha-Beta Example (continued)

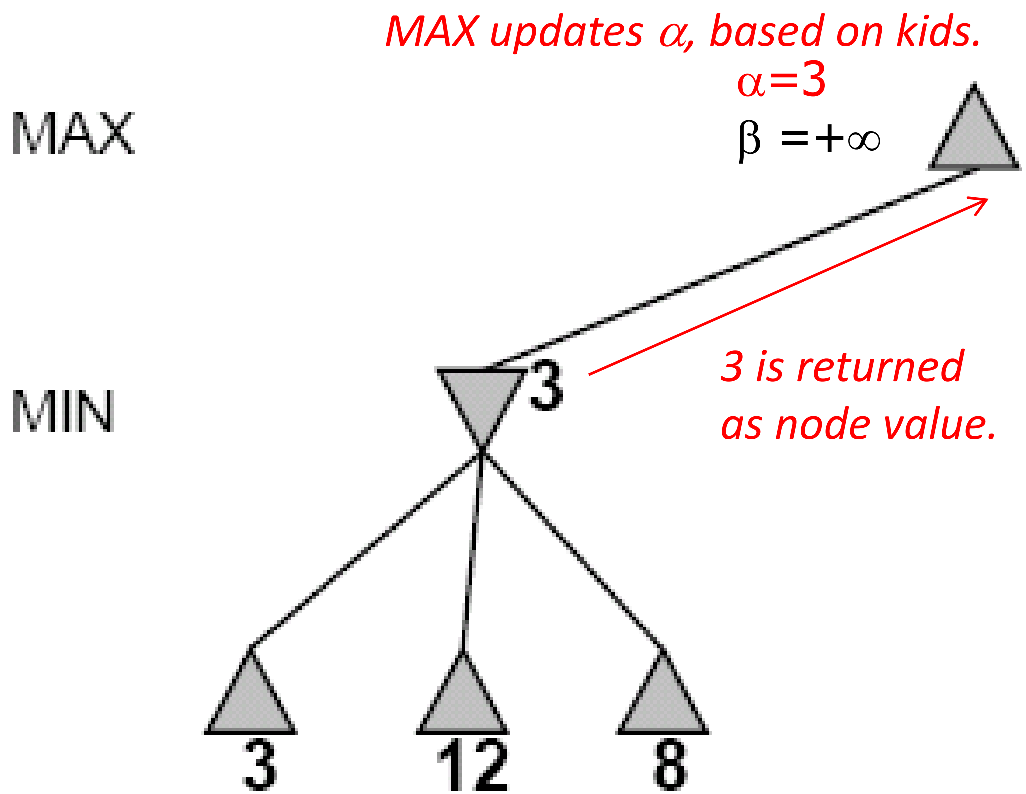


Alpha-Beta Example (continued)

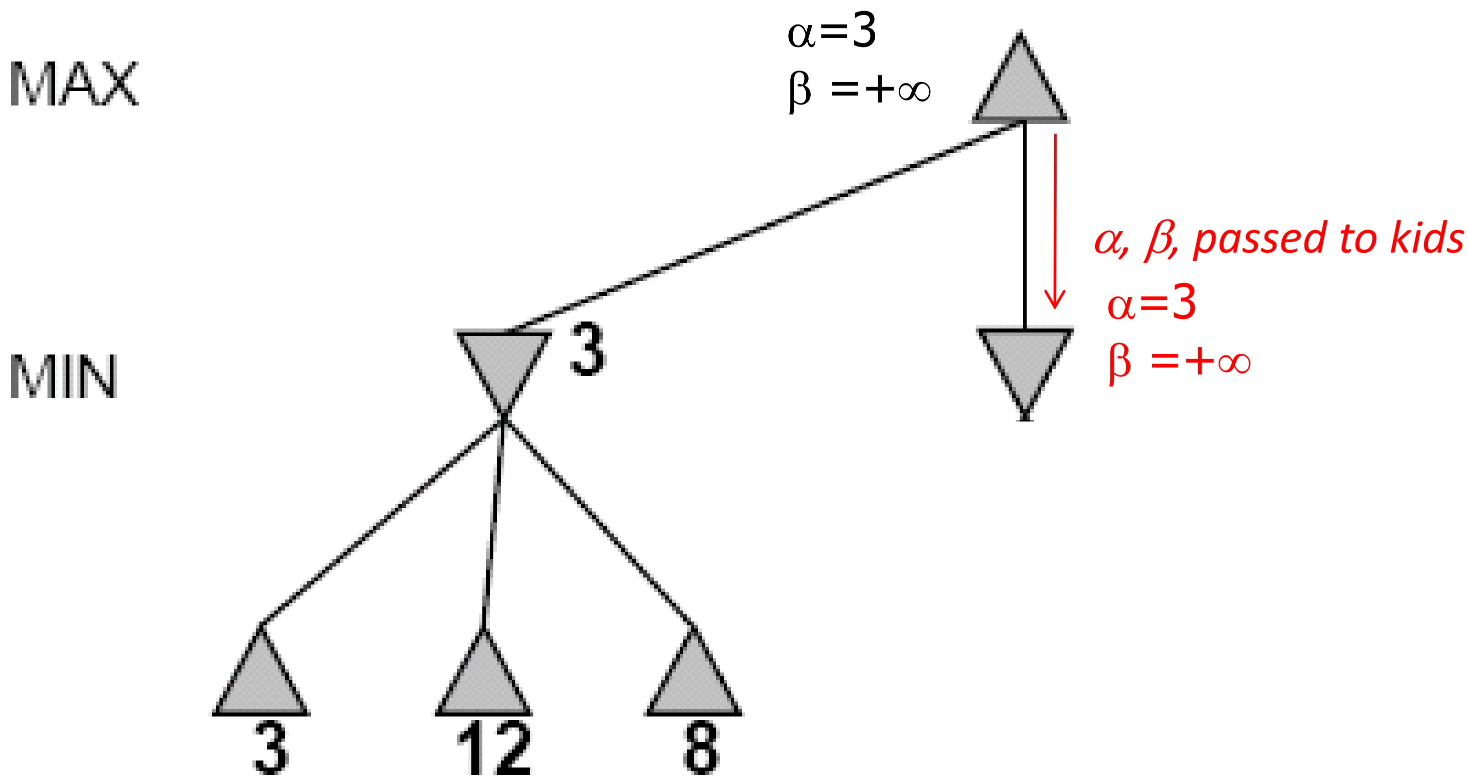


MIN updates β, based on kids.
No change.

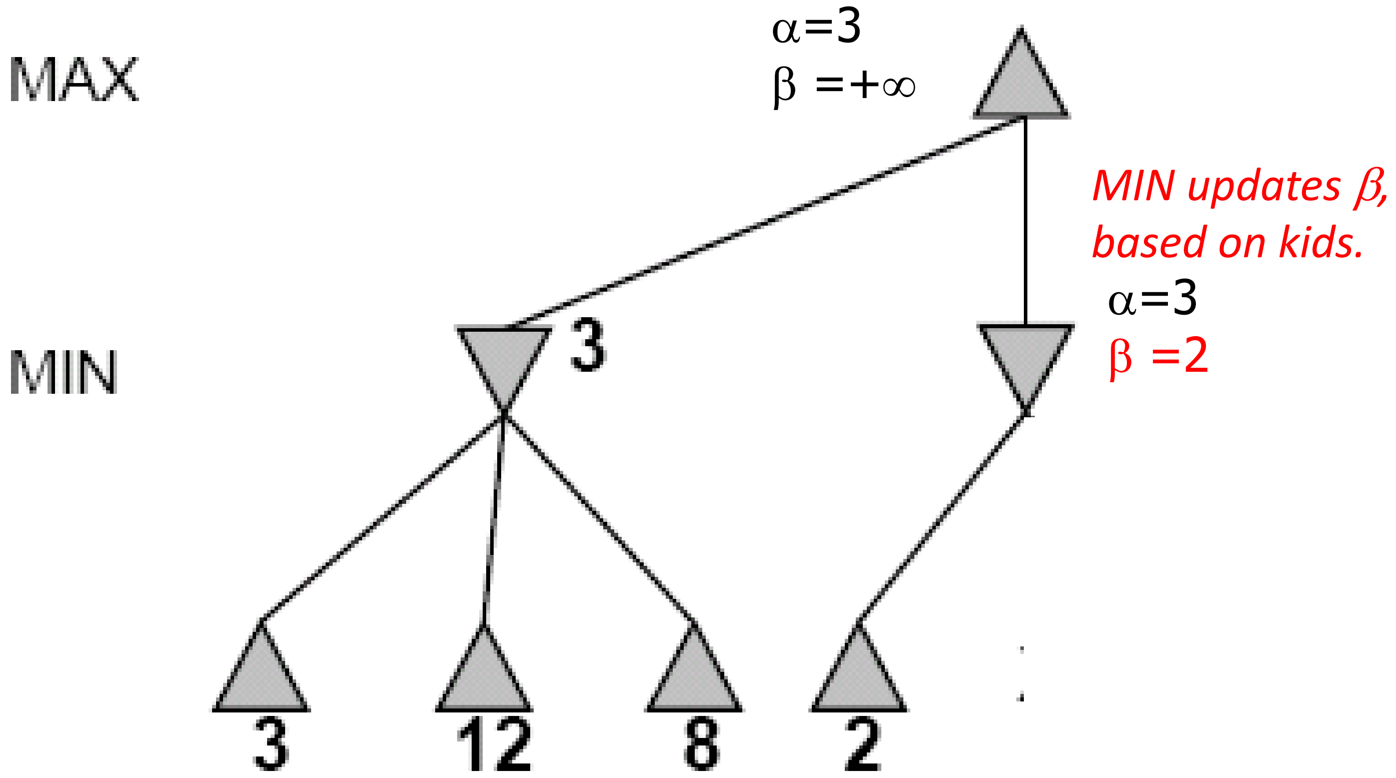
Alpha-Beta Example (continued)



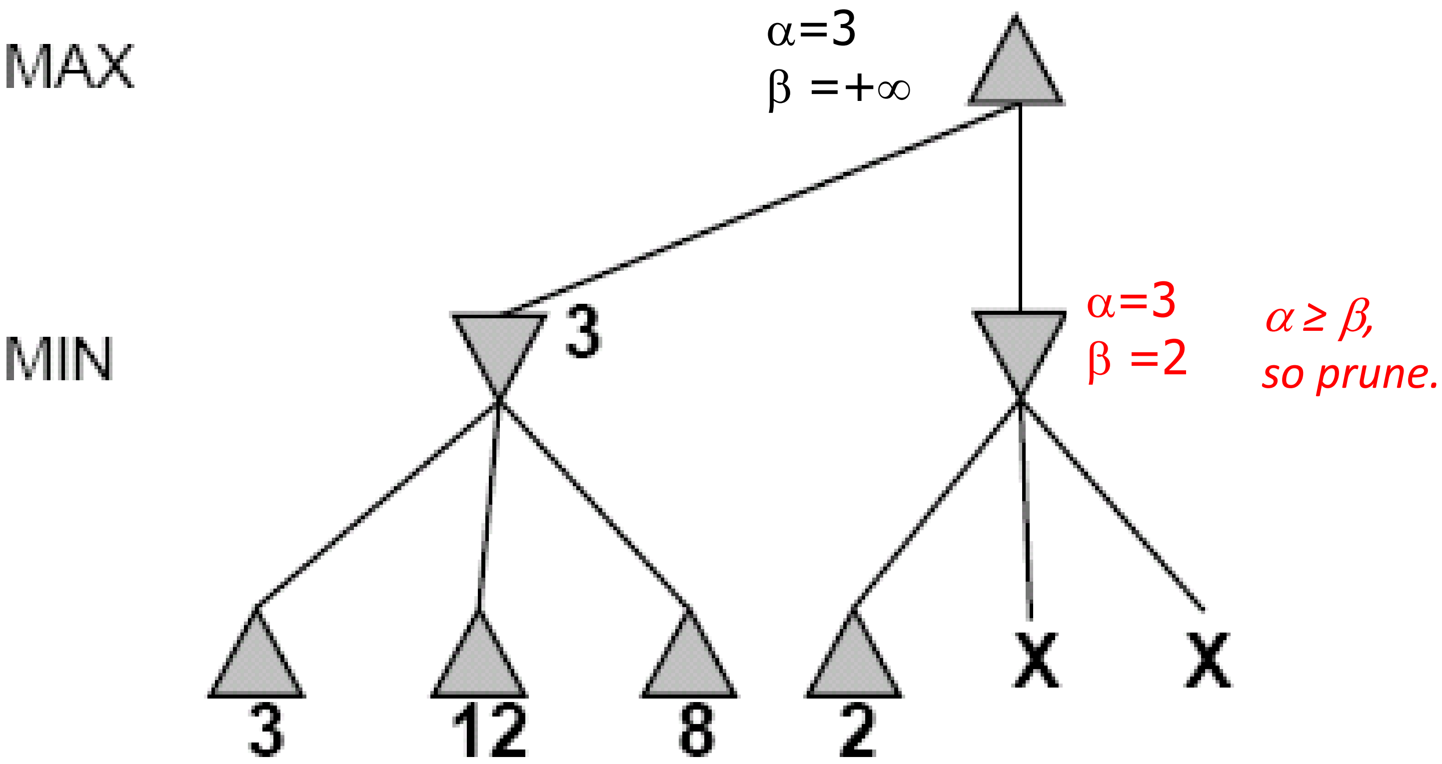
Alpha-Beta Example (continued)



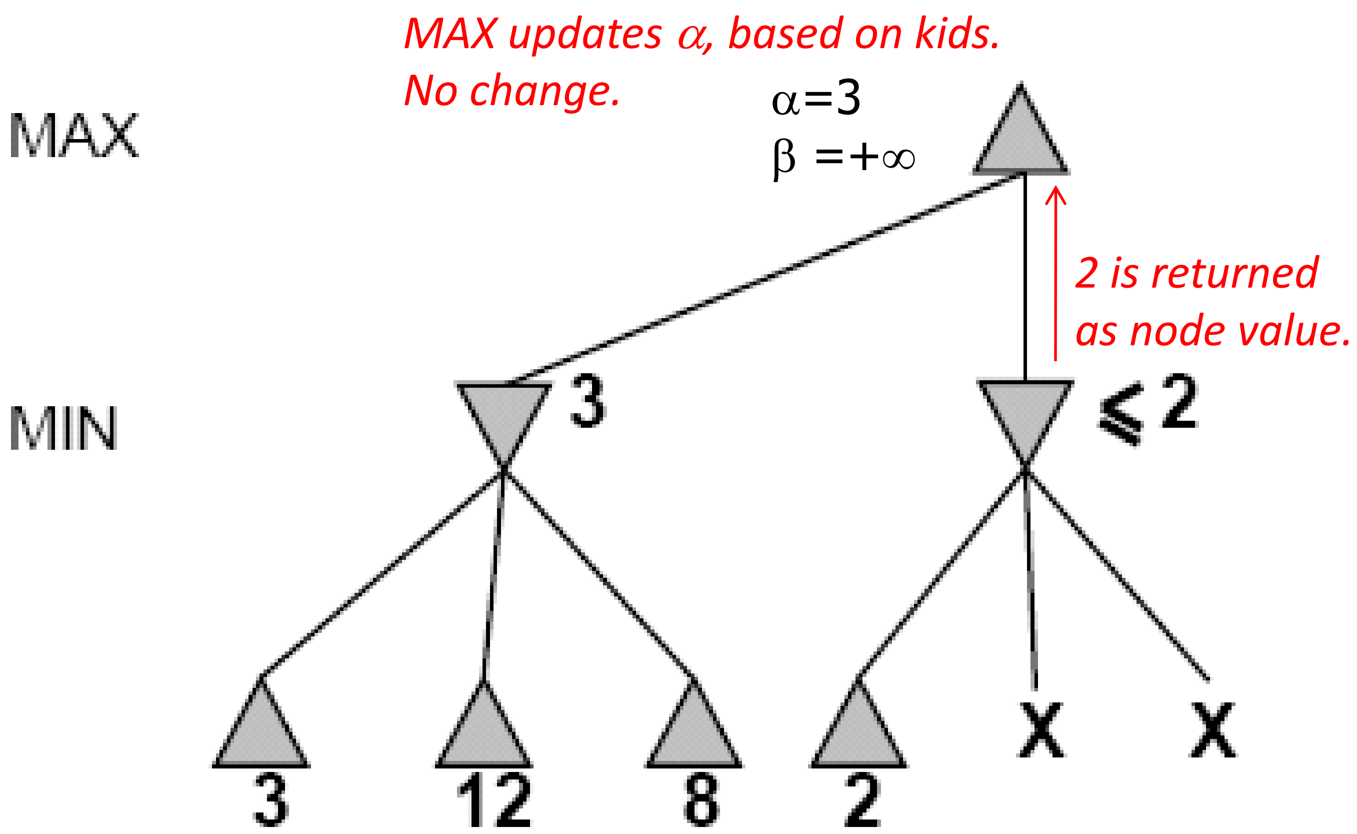
Alpha-Beta Example (continued)



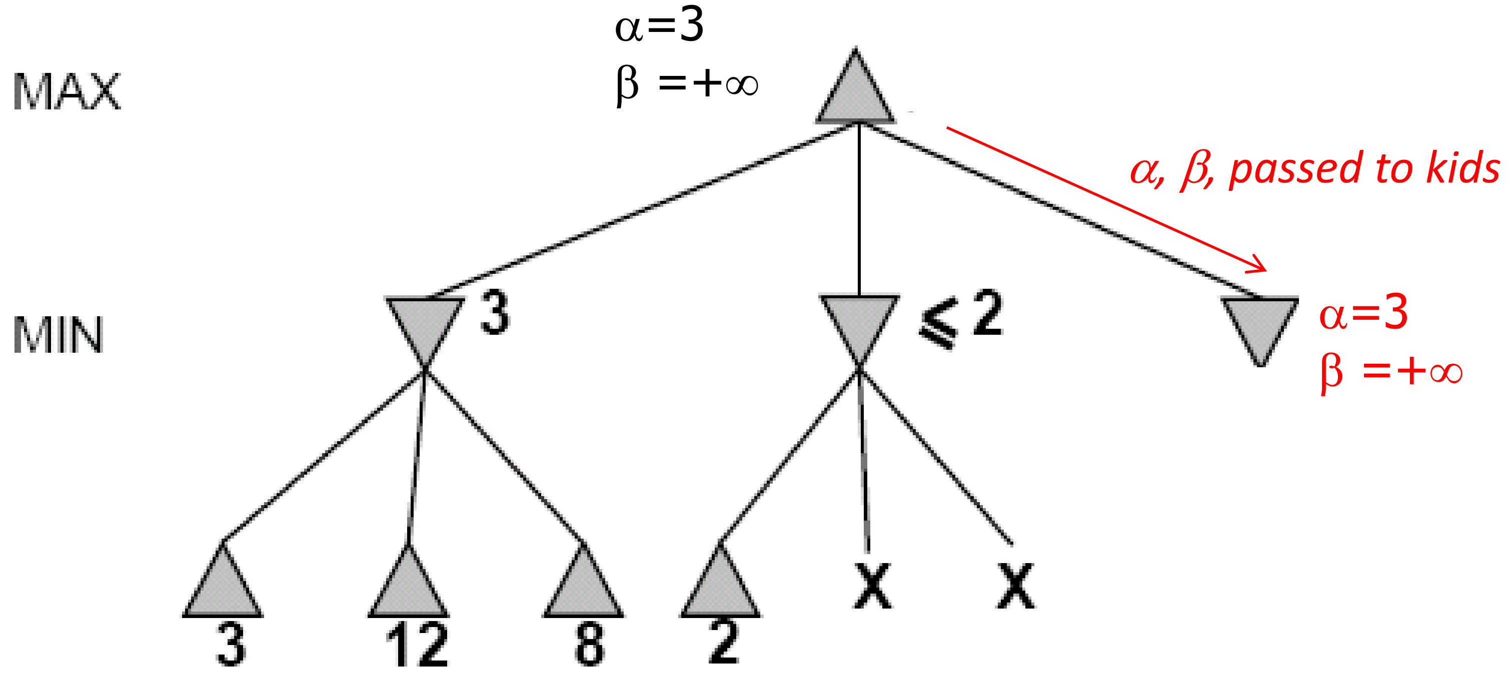
Alpha-Beta Example (continued)



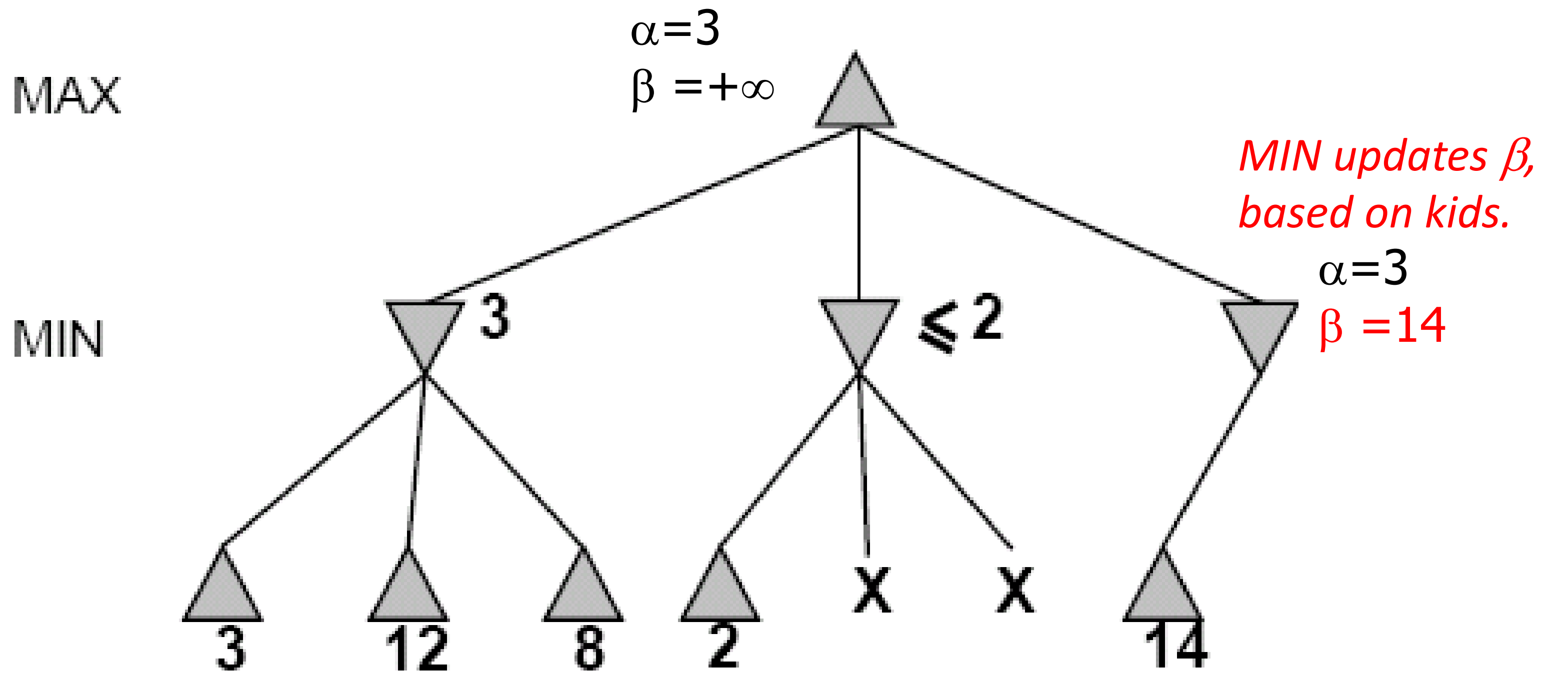
Alpha-Beta Example (continued)



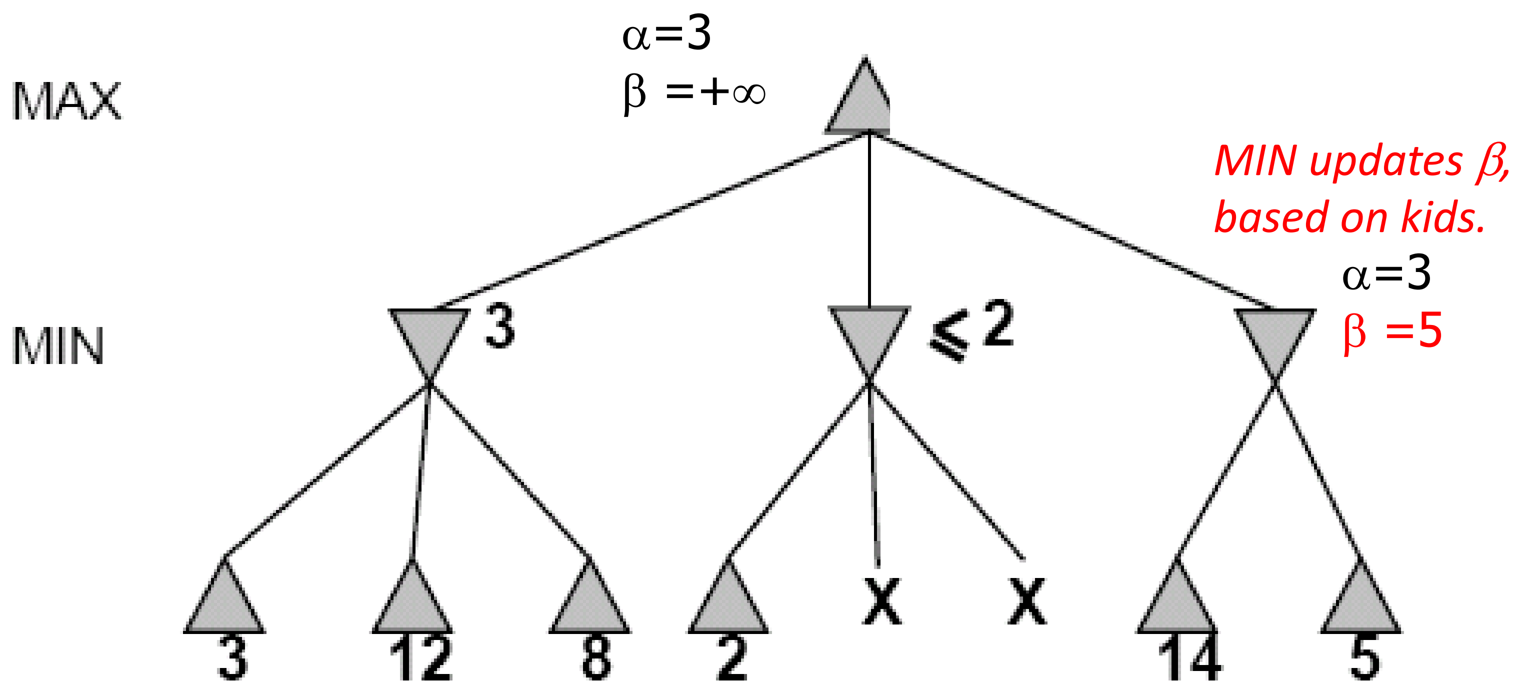
Alpha-Beta Example (continued)



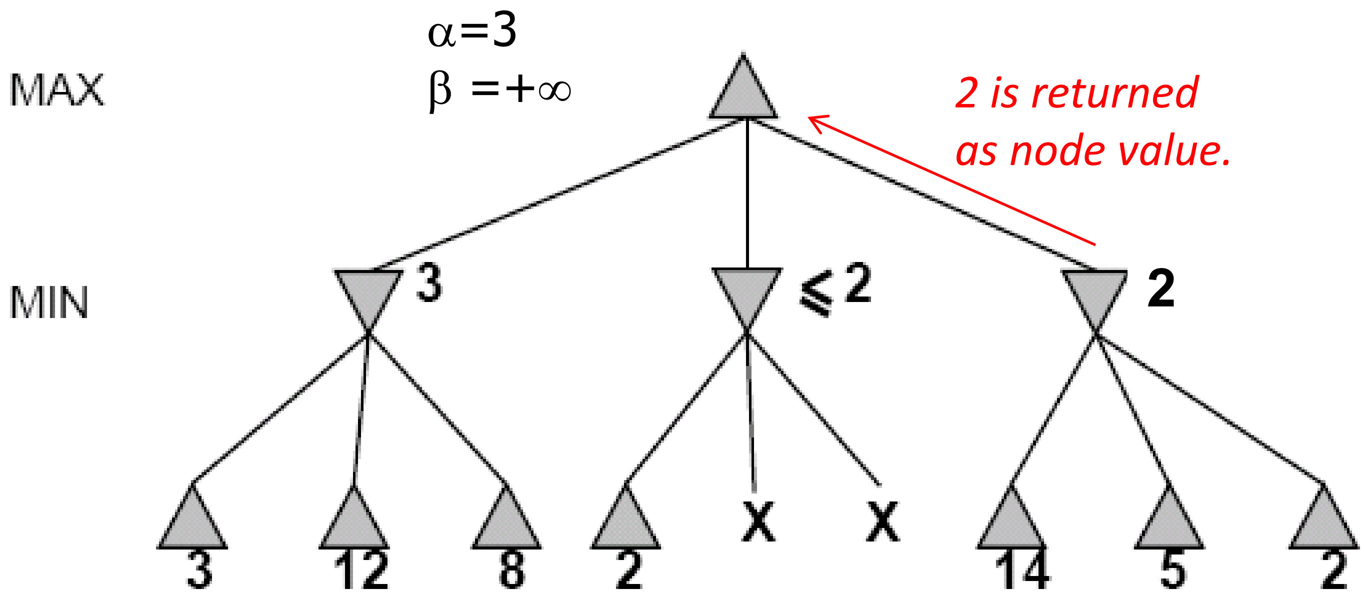
Alpha-Beta Example (continued)



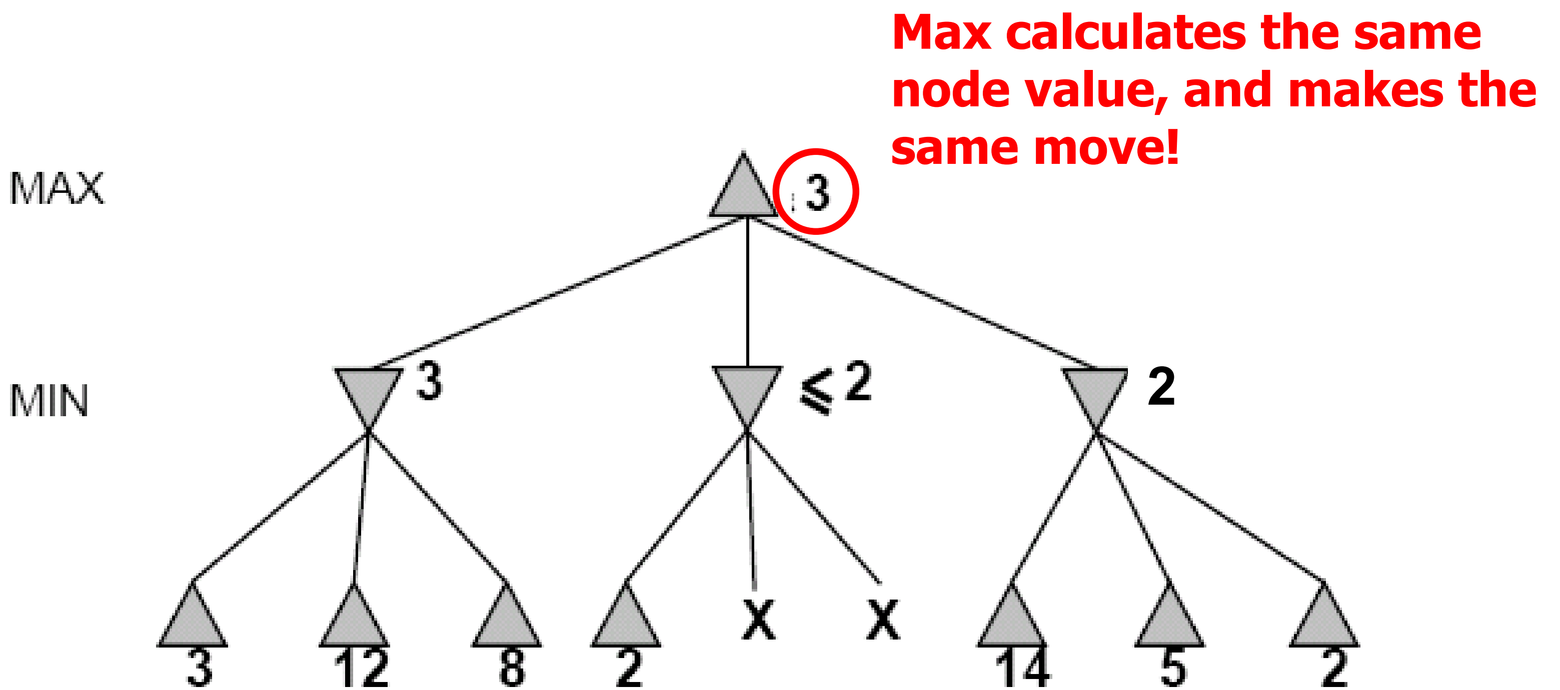
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



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

CS-171 Final Review

- **Machine Learning Classifiers**
  - (R&N Ch. 18.5-18.12; 20.2)
- **Intro to Machine Learning**
  - (R&N Ch. 18.1-18.4)
- **Game (Adversarial) Search**
  - (R&N Ch. 5.1-5.4)
- **<u>Local Search</u>**
  - (R&N Ch. 4.1-4.2)
- **State Space Search**
  - (R&N Ch. 3.1-3.7)
- Questions on any topic
- Please review your quizzes & old tests

Review Local Search
Chapter 4.1-4.2, 4.6; Optional 4.3-4.5

- Problem Formulation (4.1)
- Hill-climbing Search (4.1.1)
- Simulated annealing search (4.1.2)
- Local beam search (4.1.3)
- Genetic algorithms (4.1.4)

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.

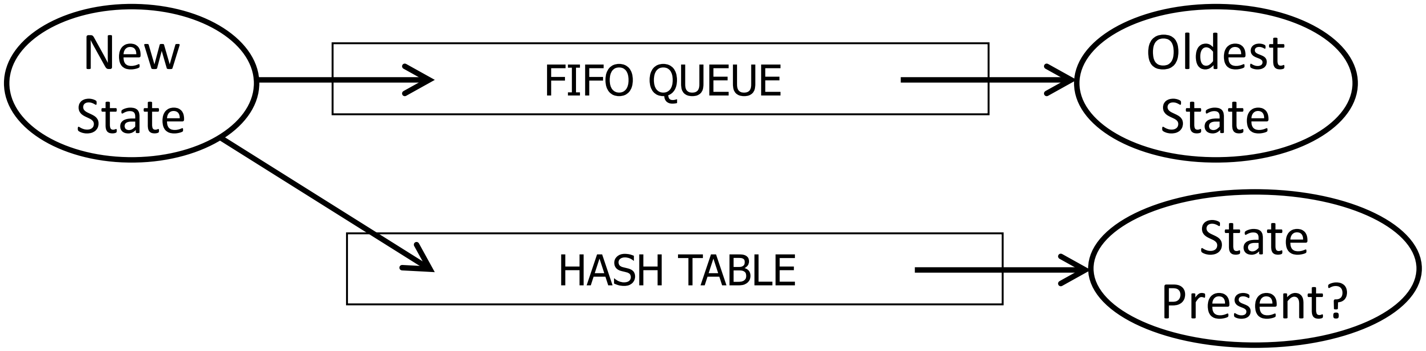
Tabu search wrapper

- 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*

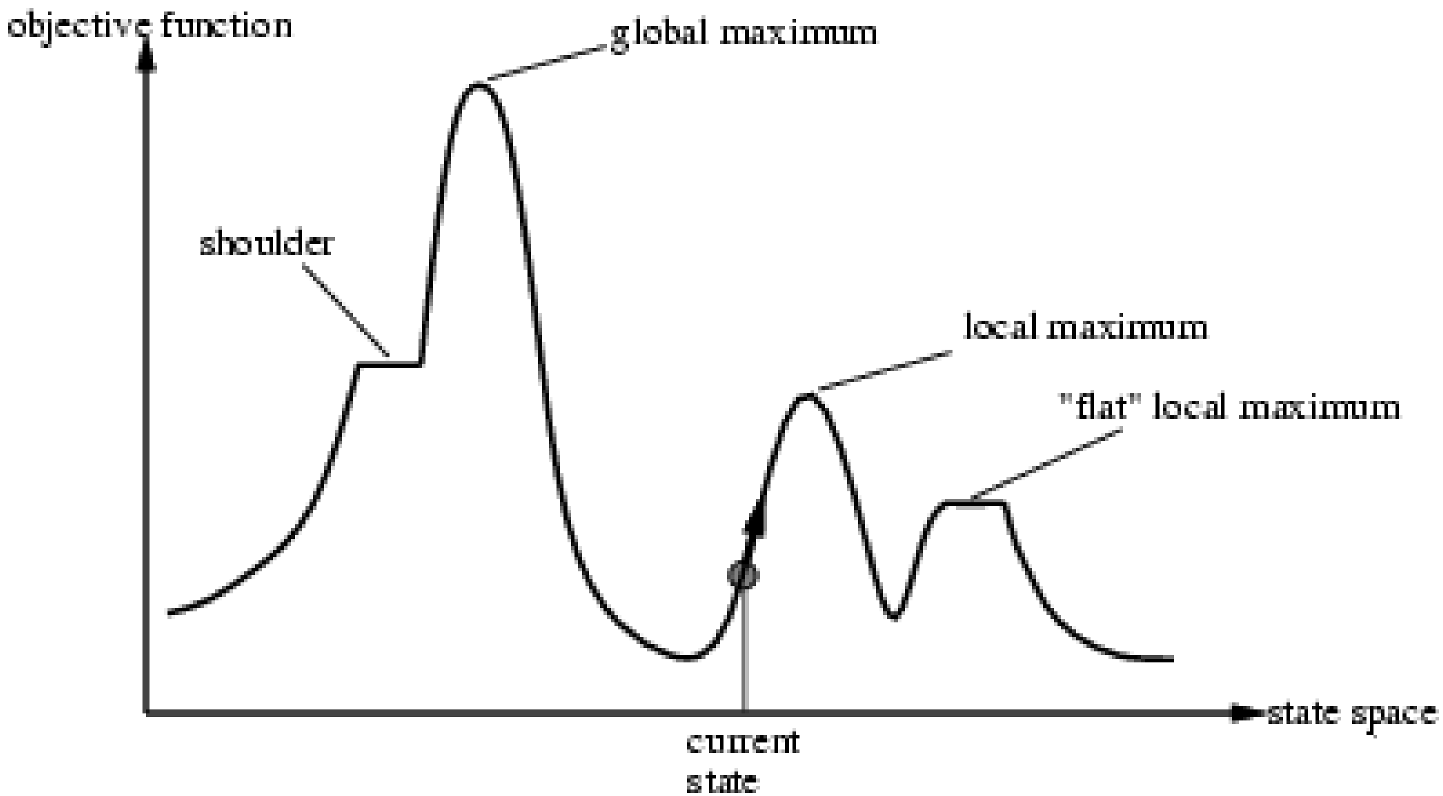
Local search algorithms

- 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.

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

<u>Ridge:</u>
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.

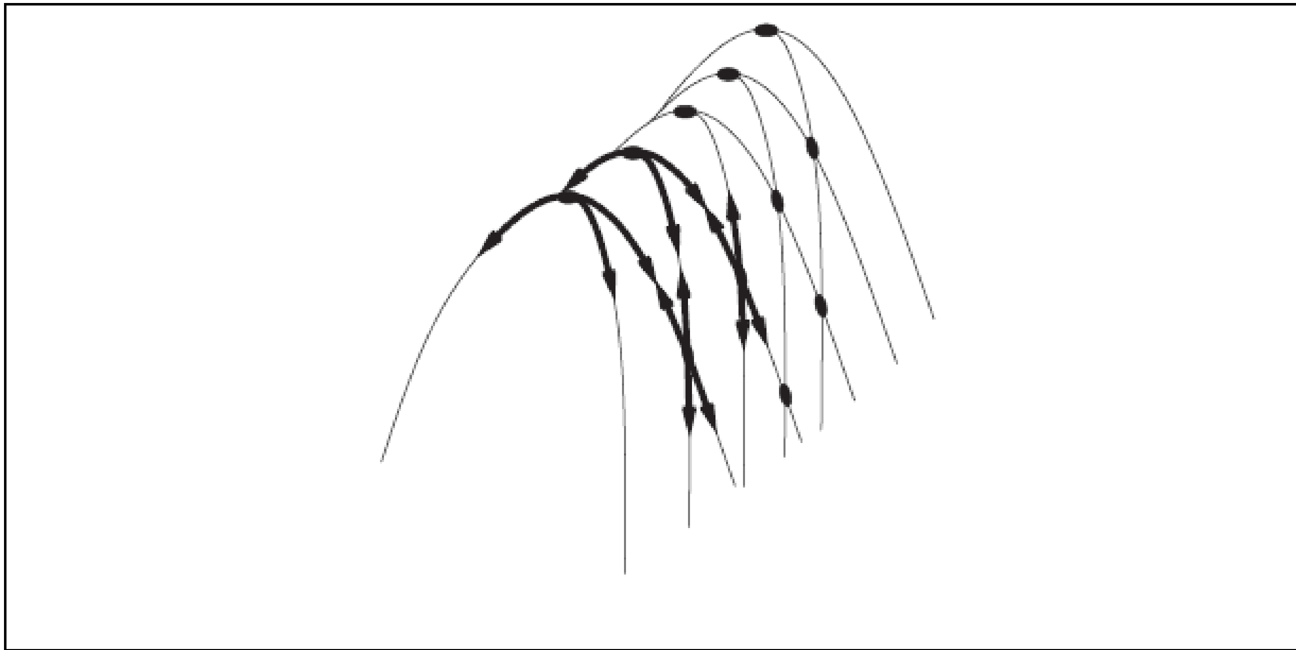


Figure 4.4 FILES: figures/ridge.eps (Tue Nov 3 16:23:29 2009). Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From 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 $e^{\Delta E/T}$

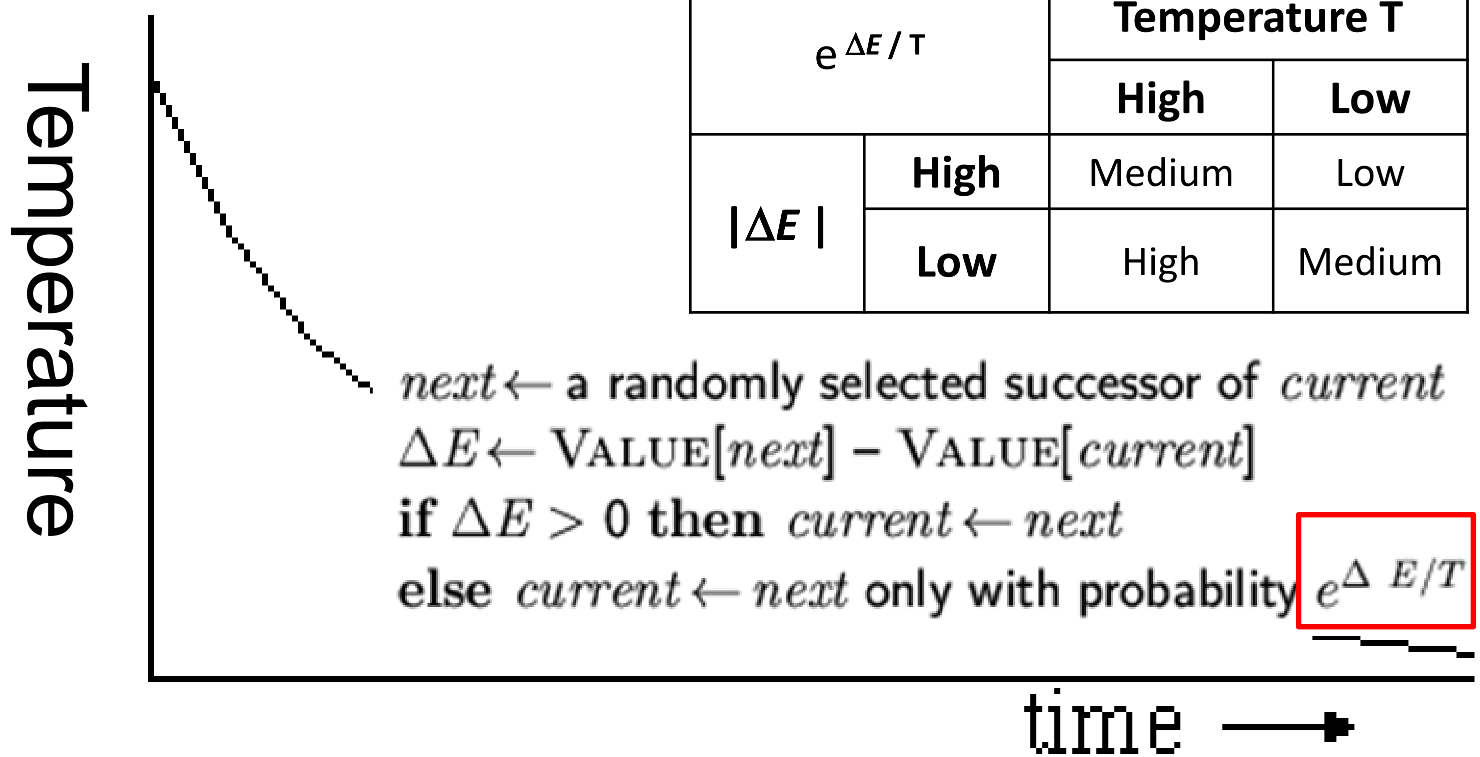
Improvement: Track the BestResultFoundSoFar. Here, this slide follows Fig. 4.5 of the textbook, which is simplified.

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”)

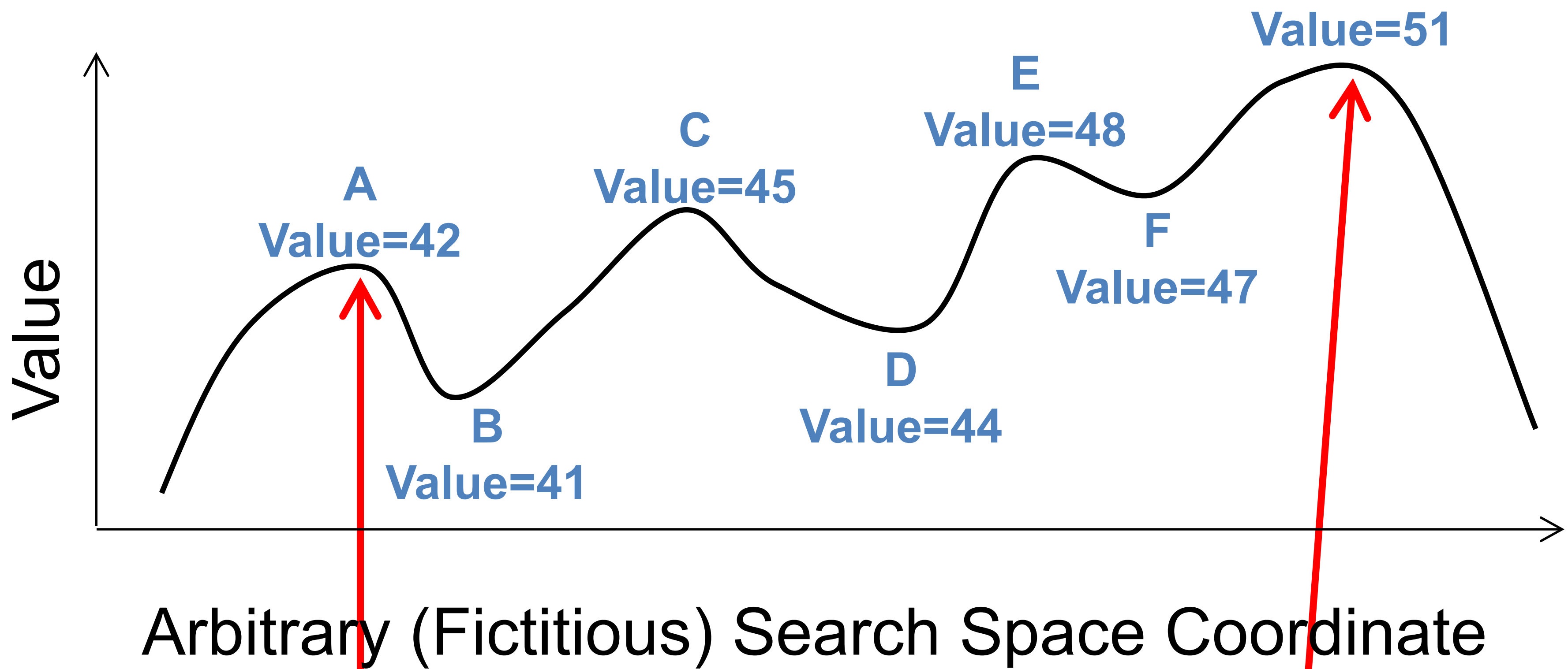
$e^{\Delta E / T}$		Temperature T	
		High	**Low**
$\lvert \Delta E \rvert$	**High**	Medium	Low
	Low	High	Medium



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>)

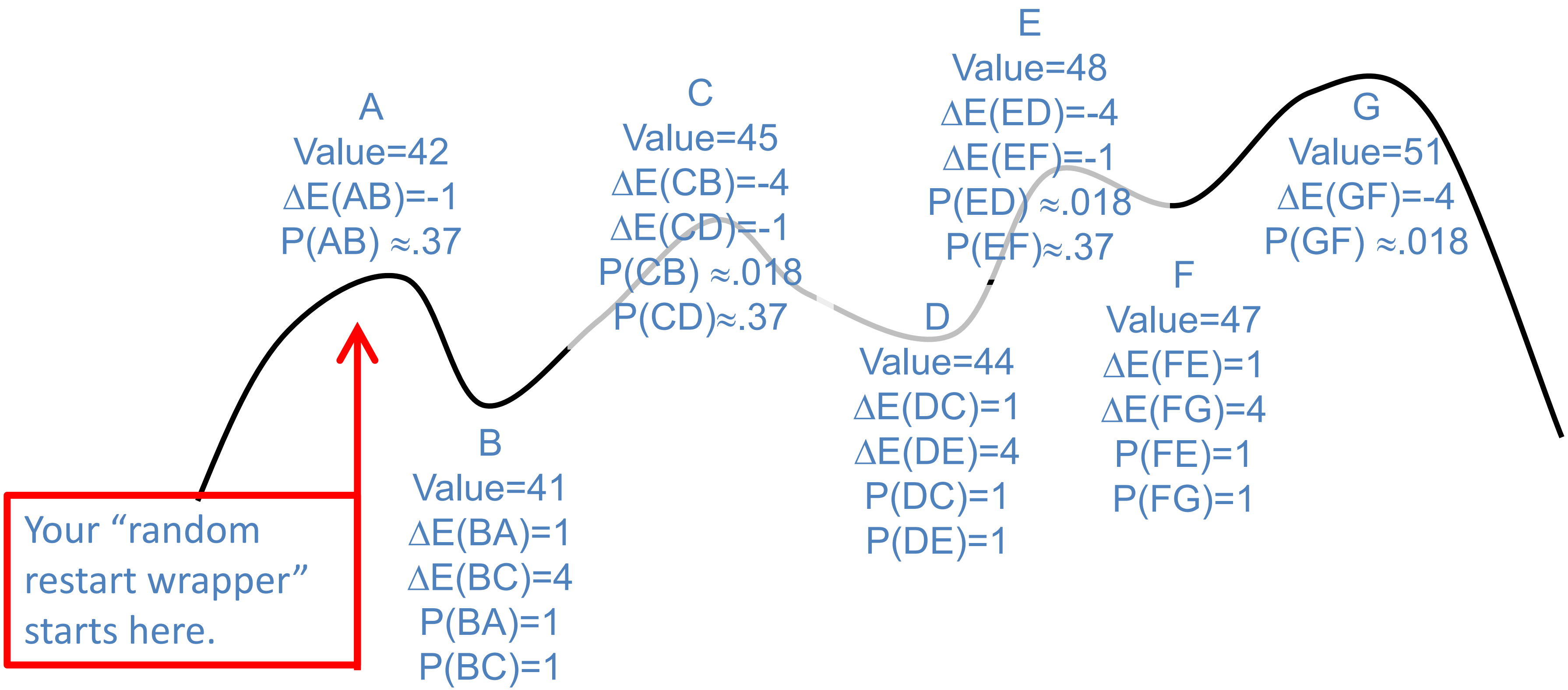


Your "random restart wrapper" starts here.

You want to get here. HOW??

This is an illustrative *cartoon*…

Goal: “ratchet up” a jagged slope



x	-1	-4
e^x	≈.37	≈.018

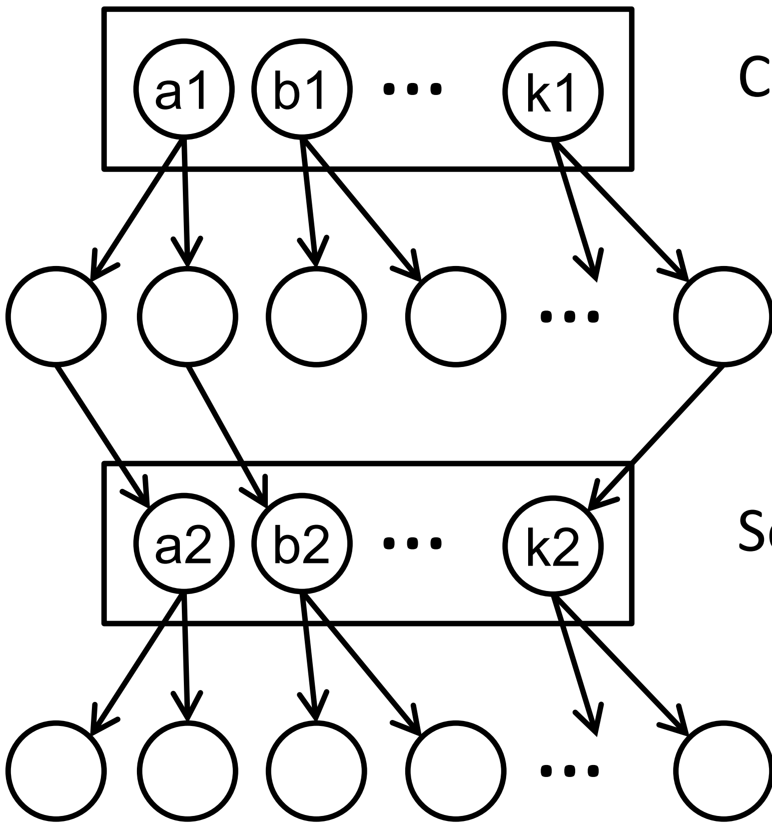
This is an illustrative *cartoon*…

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?).

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

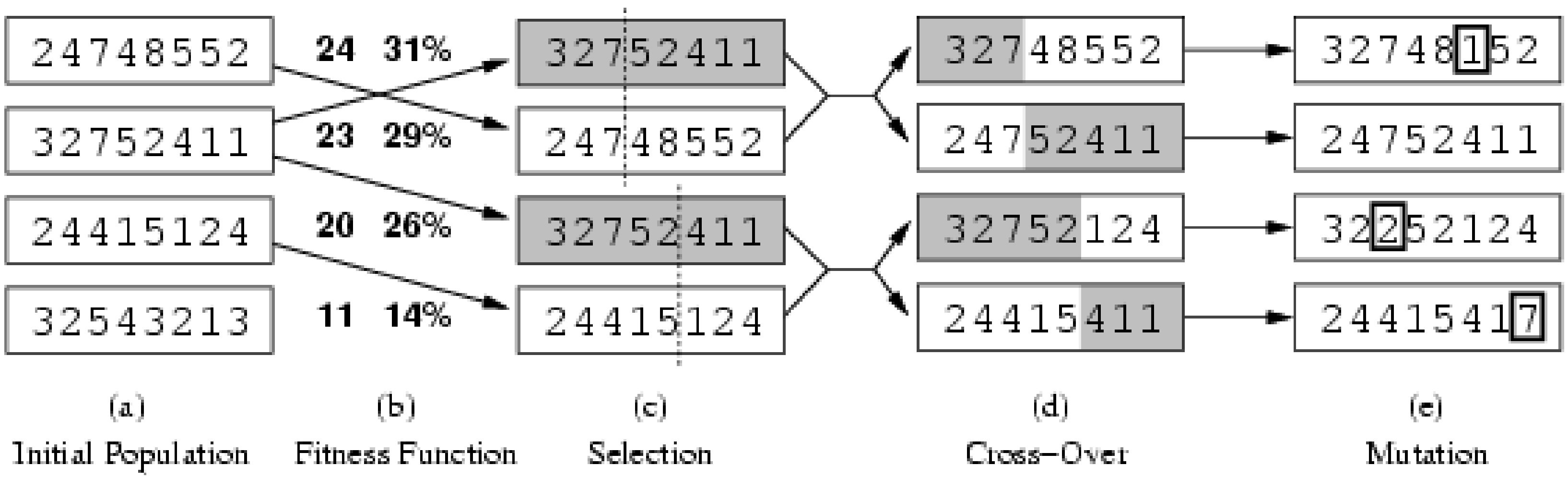
Repeat indefinitely…

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

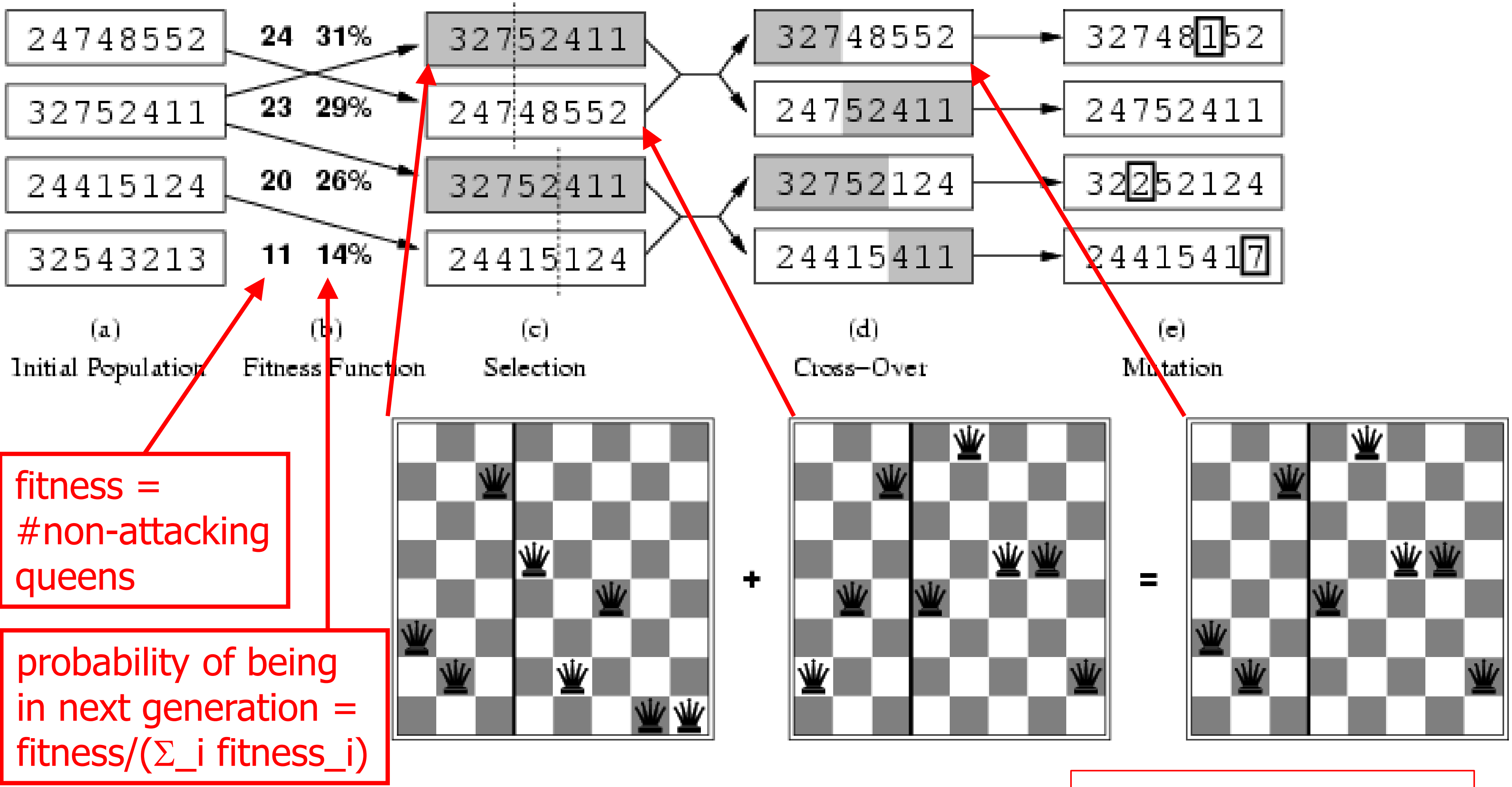
Genetic algorithms (<u>Darwin!!</u>)

- 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 × 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

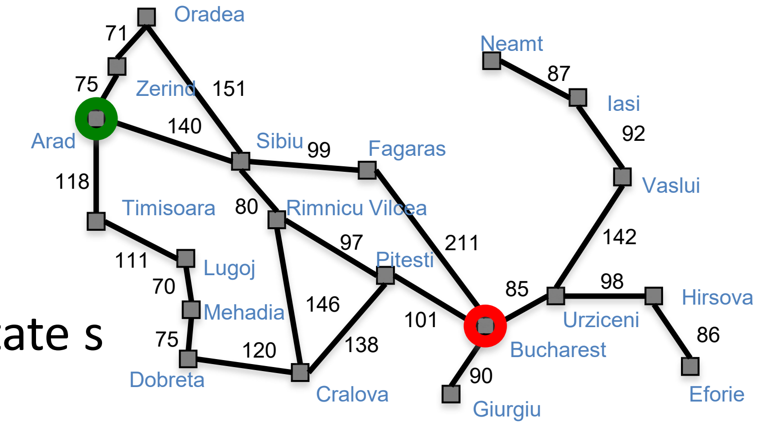
CS-171 Final Review

- **Machine Learning Classifiers**
  - (R&N Ch. 18.5-18.12; 20.2)
- **Intro to Machine Learning**
  - (R&N Ch. 18.1-18.4)
- **Game (Adversarial) Search**
  - (R&N Ch. 5.1-5.4)
- **Local Search**
  - (R&N Ch. 4.1-4.2)
- **<u>State Space Search</u>**
  - (R&N Ch. 3.1-3.7)
- Questions on any topic
- Please review your quizzes & old tests

Review State Space Search
Chapter 3

- Problem Formulation (3.1, 3.3)
- Blind (Uninformed) Search (3.4)
  - Depth-First, Breadth-First, Iterative Deepening, Uniform-Cost, Bidirectional (if applicable)
  - Time? Space? Complete? Optimal?
- Heuristic Search (3.5)
  - A*, Greedy-Best-First

State-Space Problem Formulation



A **problem** is defined by five items:

(1) initial state e.g., "at Arad"

(2) actions *Actions(s)* = set of actions avail. in state s

(3) transition model Results(s,a) = state that results from action a in state s

Alt: successor function *S(x)* = set of action–state pairs

– e.g., *S(Arad) = {<Arad → Zerind, Zerind>, … }*

(4) goal test, (or goal state)

e.g., *x* = "at Bucharest", *Checkmate(x)*

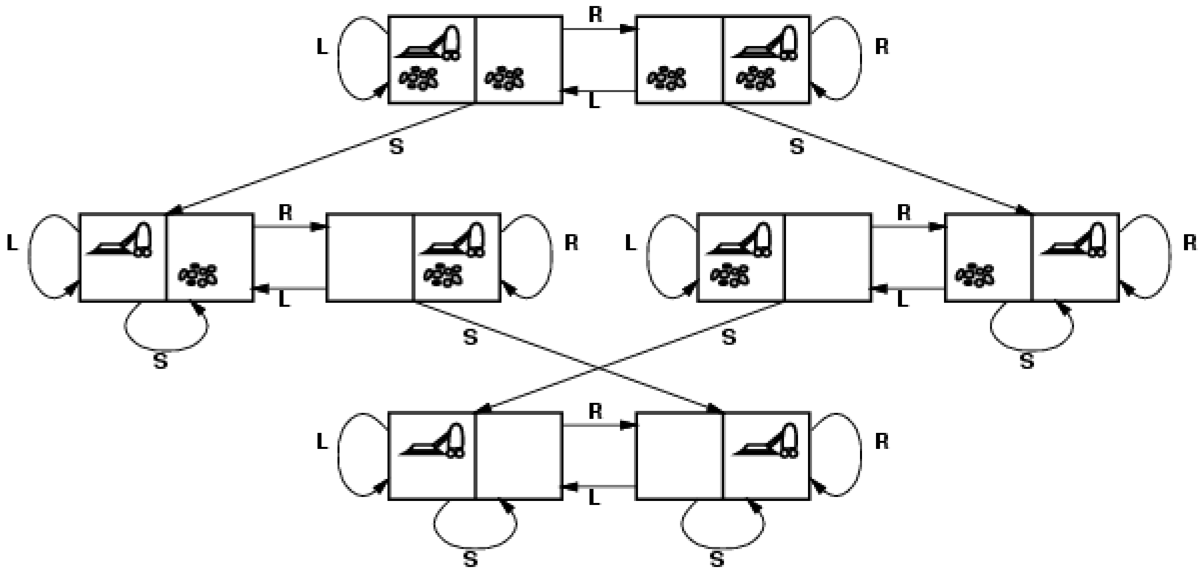
(5) path cost (additive)

– e.g., sum of distances, number of actions executed, etc.

– *c(x,a,y)* is the step cost, assumed to be ≥ 0 (and often, assumed to be $\geq \varepsilon > 0$)

A **solution** is a sequence of actions leading from the initial state to a goal state

Vacuum world state space graph

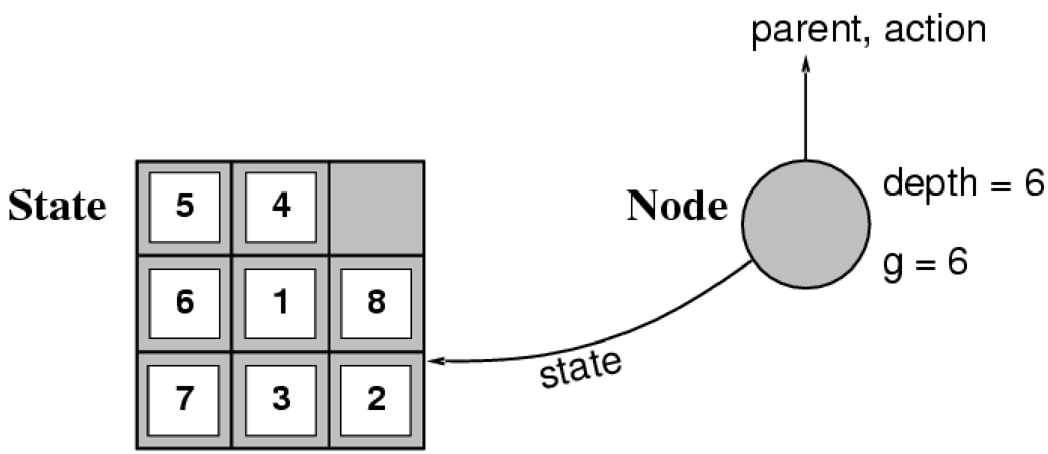


- states? discrete: dirt and robot locations
- initial state? any
- actions? *Left*, *Right*, *Suck*
- transition model? as shown on graph
- goal test? no dirt at all locations
- path cost? 1 per action

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration

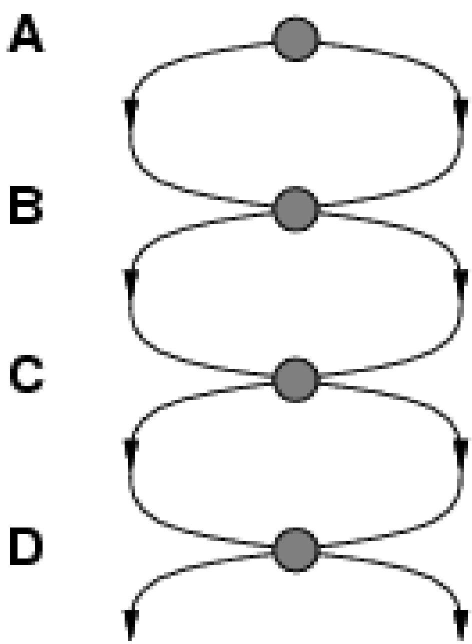
- A node is a data structure constituting part of a search tree
- A node contains info such as:
  - state, parent node, action, path cost *g(x)*, depth, etc.

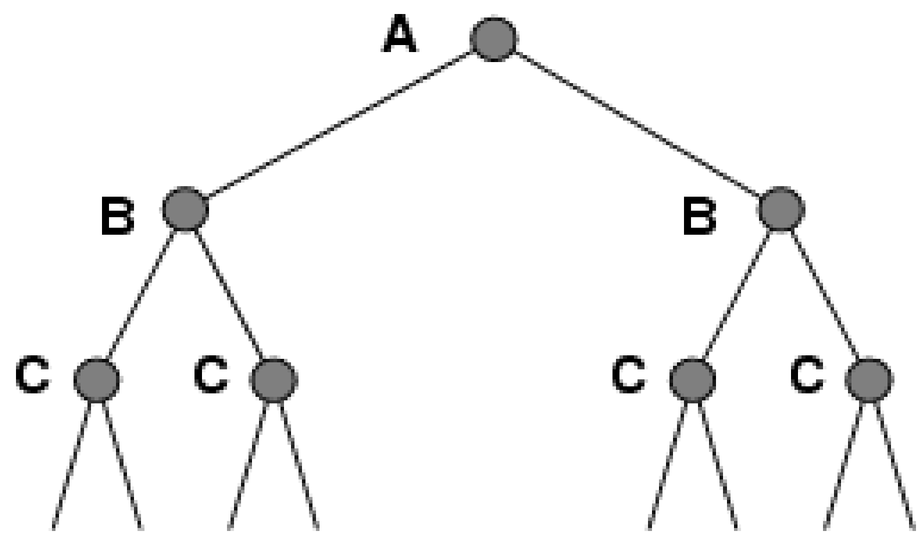


- The `Expand` function creates new nodes, filling in the various fields using the `Actions(S)` and `Result(S,A)` functions associated with the problem.

Tree search vs. Graph search Review Fig. 3.7, p. 77

- Failure to detect repeated states can turn a linear problem into an exponential one!
- Test is often implemented as a hash table.

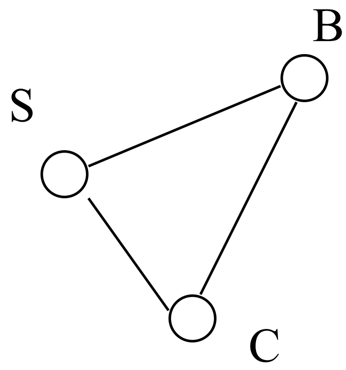




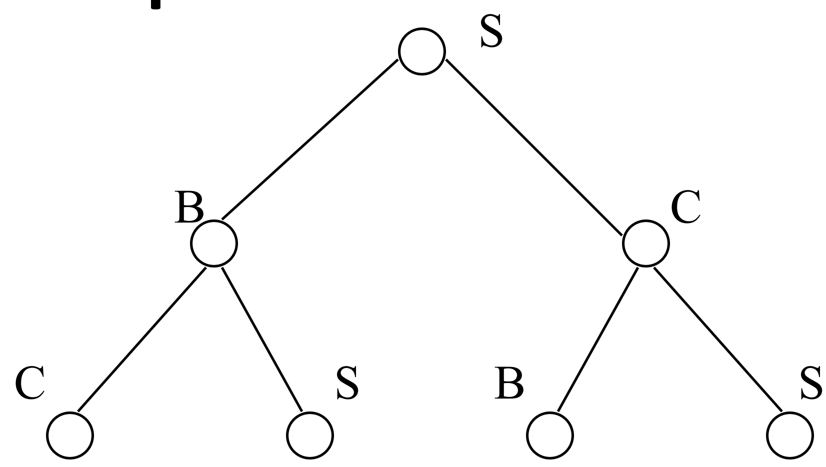
Tree search vs. Graph search
Review Fig. 3.7, p. 77

- What R&N call Tree Search vs. Graph Search
  - (And we follow R&N **exactly** in this class)
  - Has **NOTHING** to do with searching trees vs. graphs
- **Tree Search** = do **NOT** remember visited nodes
  - Exponentially slower search, but memory efficient
- **Graph Search** = **DO** remember visited nodes
  - Exponentially faster search, but memory blow-up
- **CLASSIC Comp Sci TIME-SPACE TRADE-OFF**

Solutions to Repeated States



State Space



Example of a Search Tree

- Graph search ← faster, but memory inefficient
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states
    - approximation for DFS/DLS: only avoid states in its (limited) memory: avoid infinite loops by checking path back to root.
  - "visited?" test usually implemented as a hash table

Checking for identical nodes (1)

Check if a node is already in fringe-frontier

- It is "easy" to check if a node is already in the fringe/frontier (recall fringe = frontier = open = queue)
  - Keep a hash table holding all fringe/frontier nodes
    - Hash size is same O(.) as priority queue, so hash does not increase overall space O(.)
    - Hash time is O(1), so hash does not increase overall time O(.)
  - When a node is expanded, remove it from hash table (it is no longer in the fringe/frontier)
  - For each resulting child of the expanded node:
    - If child is not in hash table, add it to queue (fringe) and hash table
    - Else if an old lower- or equal-cost node is in hash, discard the new higher- or equal-cost child
    - Else remove and discard the old higher-cost node from queue and hash, and add the new lower-cost child to queue and hash

Always do this for tree or graph search in BFS, UCS, GBFS, and A*

Checking for identical nodes (2)

Check if a node is in explored/expanded

- It is memory-intensive [$O(b^d)$ or $O(b^m)$]to check if a node is in explored/expanded (recall explored = expanded = closed)
  - Keep a hash table holding all explored/expanded nodes (hash table may be HUGE!!)
- When a node is expanded, add it to hash (explored)
- For each resulting child of the expanded node:
  - If child is not in hash table or in fringe/frontier, then add it to the queue (fringe/frontier) and process normally (BFS normal processing differs from UCS normal processing, but the ideas behind checking a node for being in explored/expanded are the same).
  - Else discard any redundant node.

Always do this for graph search

Breadth-first graph search (R&N Fig. 3.11)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0 if
  problem.GOAL-TEST(node.STATE) then return SOLUTION(node) frontier ←
  a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Goal test before push

Avoid redundant frontier nodes

Figure 3.11 Breadth-first search on a graph.

These three statements change tree search to graph search.

Properties of breadth-first search

- Complete? Yes, it always reaches a goal (if *b* is finite)
- Time? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$

  (this is the number of nodes we generate)
- Space? $O(b^d)$

  (keeps every node in memory, either in frontier or on a path to frontier).
- Optimal? No, for general cost functions.

  Yes, if cost is a non-decreasing function only of depth.
  - With $f(d) \geq f(d-1)$, e.g., step-cost = constant:
    - All optimal goal nodes occur on the same level
    - Optimal goals are always shallower than non-optimal goals
    - An optimal goal will be found before any non-optimal goal

- Usually Space is the bigger problem (more than time)

Uniform cost search (R&N Fig. 3.14) [A* is identical except queue sort = f(n)]

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Goal test after pop

Avoid redundant frontier nodes

Avoid higher-cost frontier nodes

These three statements change tree search to graph search.

Uniform-cost search

Implementation: *Frontier* = queue ordered by path cost.
Equivalent to breadth-first if all step costs all equal.

- Complete? Yes, if b is finite and step cost $\geq \varepsilon > 0$.
  (otherwise it can get stuck in infinite regression)

- Time? # of nodes with path cost $\leq$ cost of optimal solution.
  $O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$

- Space? # of nodes with path cost $\leq$ cost of optimal solution.
  $O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$.

- Optimal? Yes, for step cost $\geq \varepsilon > 0$.

Depth-limited search & IDS (R&N Fig. 3.17-18)

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
   RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
   else if DEPTH[node] = limit then return cutoff
   else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
   if cutoff-occurred? then return cutoff else return failure
```

Goal test in recursive call, one-at-a-time

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
   inputs: problem, a problem
   for depth ← 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH( problem, depth)
      if result ≠ cutoff then return result
```

At *depth* = 0, IDS only goal-tests the start node. The start node is is not expanded at *depth* = 0.

Properties of iterative deepening search

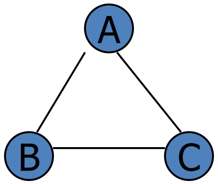
- Complete? Yes
- Time? $O(b^d)$
- Space? *O(bd)*
- Optimal? No, for general cost functions.
  Yes, if cost is a non-decreasing function only of depth.

Generally the preferred uninformed search strategy.

Depth-First Search (R&N Section 3.4.3)

- Your textbook is ambiguous about DFS.
  - The second paragraph of R&N 3.4.3 states that DFS is an instance of Fig. 3.7 using a LIFO queue. Search behavior may differ depending on how the LIFO queue is implemented (as separate pushes, or one concatenation).
  - The third paragraph of R&N 3.4.3 says that an alternative implementation of DFS is a recursive algorithm that calls itself on each of its children, as in the Depth-Limited Search of Fig. 3.17 (above).
- **<u>For quizzes and exams, we will follow Fig. 3.17.</u>**
  - Generally, for tests DFS will be used only as an example.

Properties of depth-first search



- Complete? No: fails in loops/infinite-depth spaces
  - Can modify to avoid loops/repeated states along path
    - check if current nodes occurred before on path to root
  - Can use graph search (remember all nodes ever seen)
    - problem with graph search: space is exponential, not linear
  - Still fails in infinite-depth spaces (may miss goal entirely)
- Time? $O(b^m)$ with m =maximum depth of space
  - Terrible if m is much larger than d
  - If solutions are dense, may be much faster than BFS
- Space? $O(bm)$, i.e., linear space!
  - Remember a single path + expanded unexplored nodes
- Optimal? No: It may find a non-optimal goal first

Bidirectional Search

- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both "meet in the middle"
  - need to keep track of the intersection of 2 open sets of nodes
- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - what if there are multiple goal states?
  - what if there is only a goal test, no explicit list?
- Complexity
  - time complexity is best: $O(2\ b^{(d/2)}) = O(b^{(d/2)})$
  - memory complexity is the same as time complexity

Bi-Directional Search

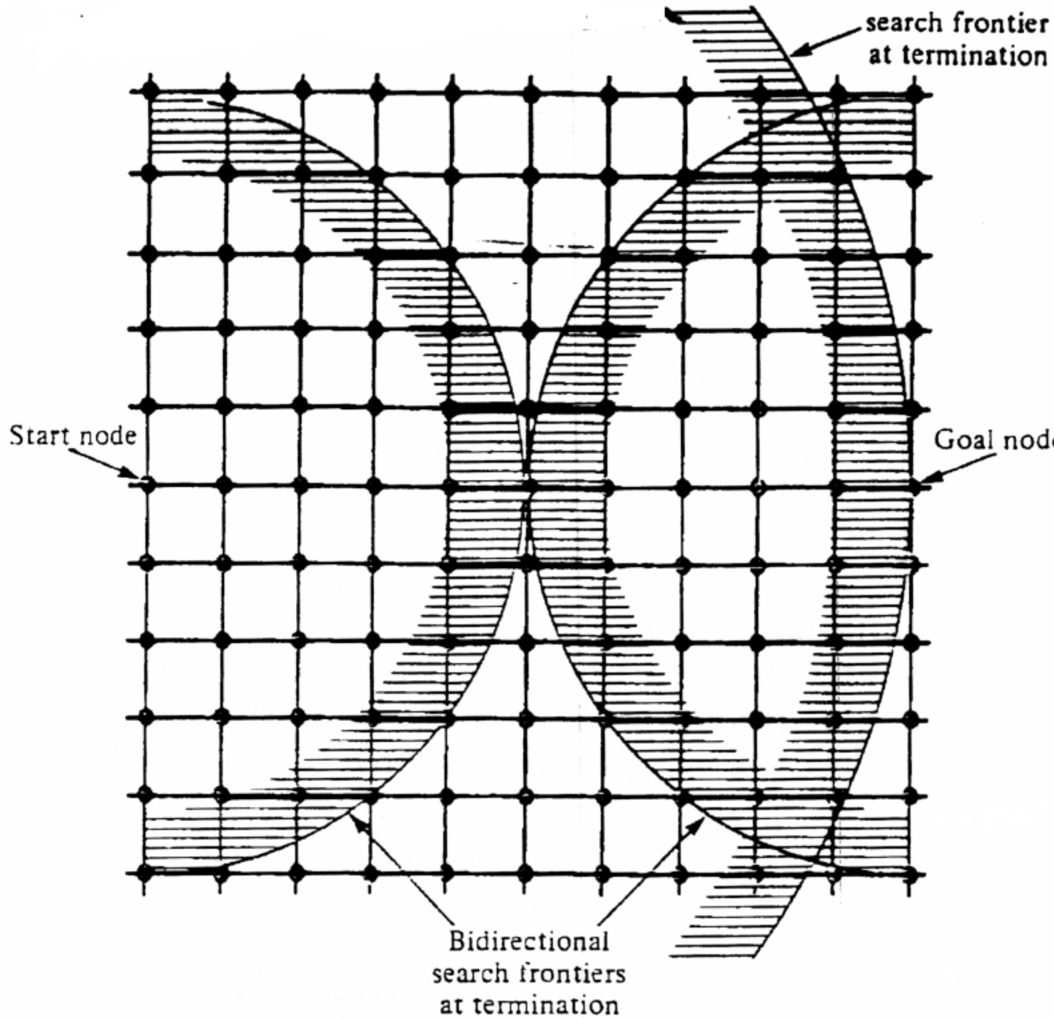


Fig. 2.10 Bidirectional and unidirectional breadth-first searches.

Blind Search Strategies (3.4)

- Depth-first: Add successors to front of queue
- Breadth-first: Add successors to back of queue
- Uniform-cost: Sort queue by path cost g(n)
- Depth-limited: Depth-first, cut off at limit *l*
- Iterated-deepening: Depth-limited, increasing *l*
- Bidirectional: Breadth-first from goal, too.

- **<u>Review "Example hand-simulated search"</u>**
  – Lecture on "Uninformed Search"

Search strategy evaluation

- A search **strategy** is defined by **the order of node expansion**

- Strategies are evaluated along the following dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: number of nodes generated
  - **space complexity**: maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?

- Time and space complexity are measured in terms of
  - ***b****:* maximum branching factor of the search tree
  - ***d:*** depth of the least-cost solution
  - ***m***: maximum depth of the state space (may be ∞)
  - (UCS: **C^*:** true cost to optimal goal; **$\varepsilon > 0$:** minimum step cost)

Summary of algorithms
Fig. 3.21, p. 91

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Generally the preferred uninformed search strategy

There are a number of footnotes, caveats, and assumptions.
See Fig. 3.21, p. 91.

[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon > 0$
[c] optimal if step costs are all identical
    (also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search
    (also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)

Summary

- Generate the search space by applying actions to the initial state and all further resulting states.
- Problem: initial state, actions, transition model, goal test, step/path cost
- Solution: sequence of actions to goal
- Tree-search (don't remember visited nodes) vs. Graph-search (do remember them)
- Search strategy evaluation: b, d, m (UCS: C*, ε)
  - Complete? Time? Space? Optimal?

Heuristic function (3.5)

- Heuristic:
  - Definition: a commonsense rule (or set of rules) intended to increase the probability of solving some problem
  - “using rules of thumb to find answers”

- Heuristic function h(n)
  - Estimate of (optimal) cost from n to goal
  - Defined using only the *state* of node *n*
  - h(n) = 0 if n is a goal node
  - Example: straight line distance from n to Bucharest
    - Note that this is not the true state-space distance
    - It is an estimate – actual state-space distance can be higher

  - Provides problem-specific knowledge to the search algorithm

Relationship of search algorithms

- Notation:
  - *g(n)* = known cost so far to reach *n*
  - *h(n)* = estimated optimal cost from *n* to goal
  - *h*(n)* = true optimal cost from *n* to goal (unknown to agent)
  - *f(n) = g(n)+h(n)* = estimated optimal total cost through *n*
- Uniform cost search: sort frontier by *g(n)*
- Greedy best-first search: sort frontier by *h(n)*
- A* search: sort frontier by *f(n) = g(n) + h(n)*
  - Optimal for admissible / consistent heuristics
  - Generally the preferred heuristic search framework
  - Memory-efficient versions of A* are available: RBFS, SMA*

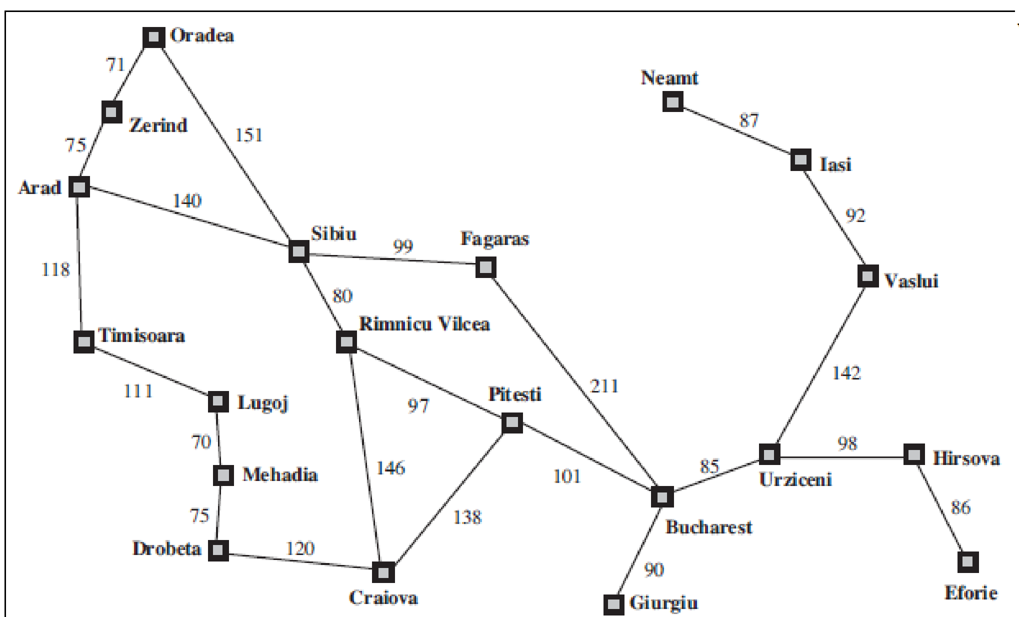
Greedy best-first search

- *h(n)* = estimate of cost from *n* to *goal*
  - e.g., *h(n)* = straight-line distance from *n* to Bucharest

- Greedy best-first search expands the node that appears to be closest to goal.
  - Sort queue by *h(n)*

- Not an optimal search strategy
  - May perform well in practice

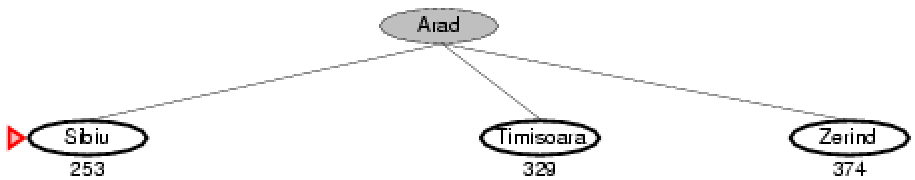
Greedy best-first search example

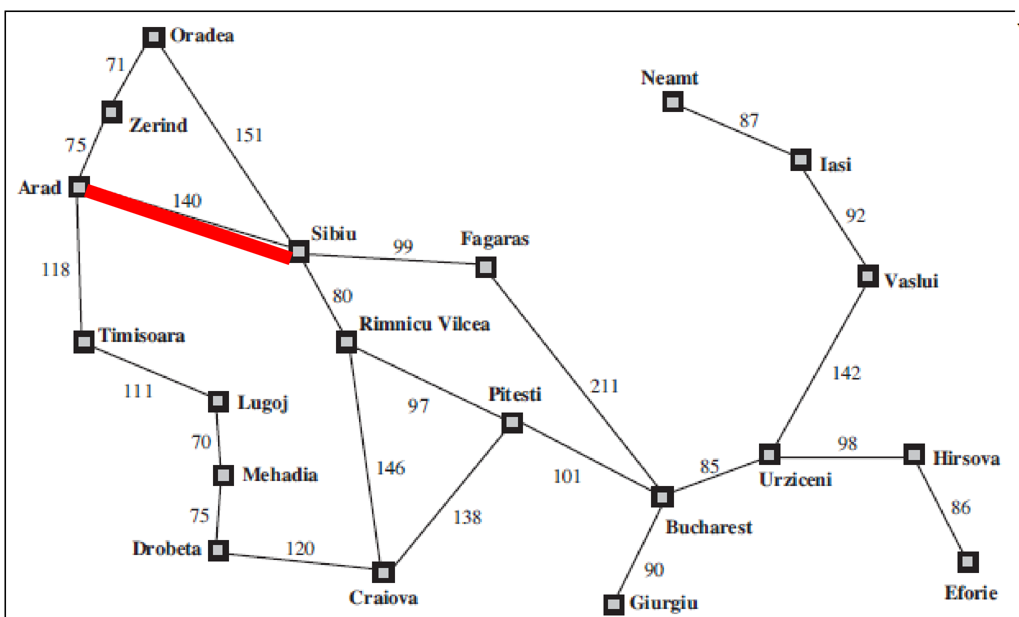


Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

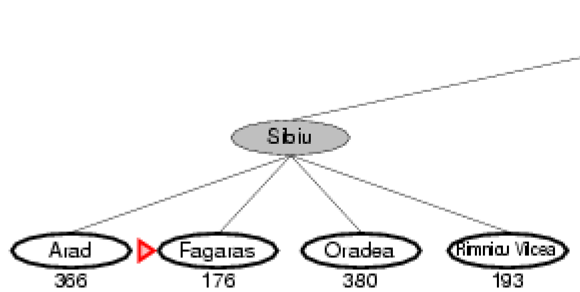


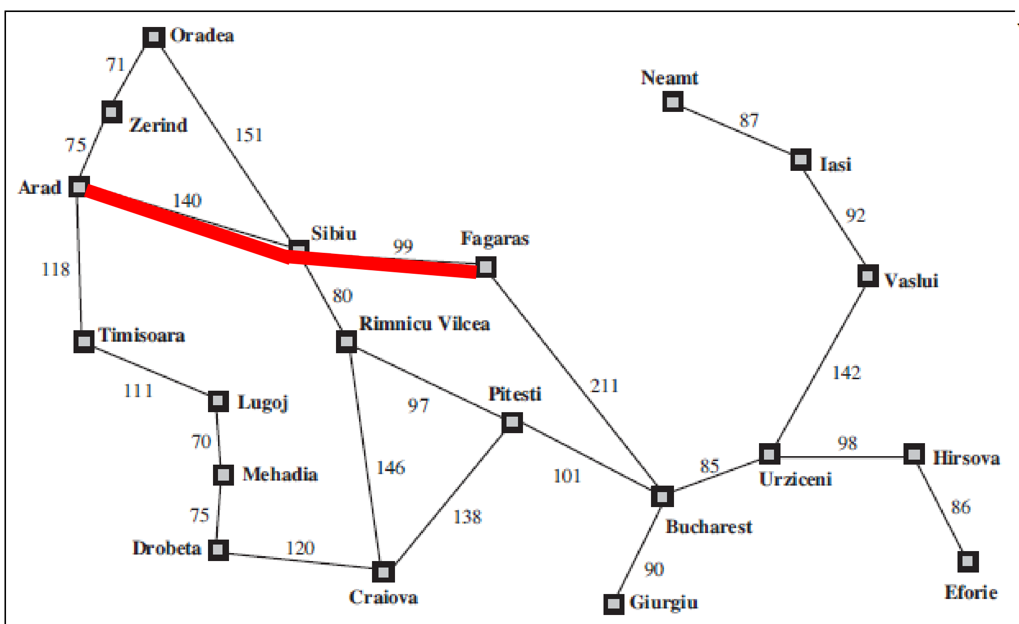


Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

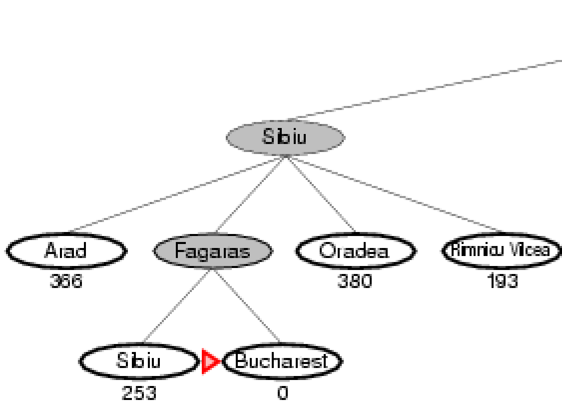


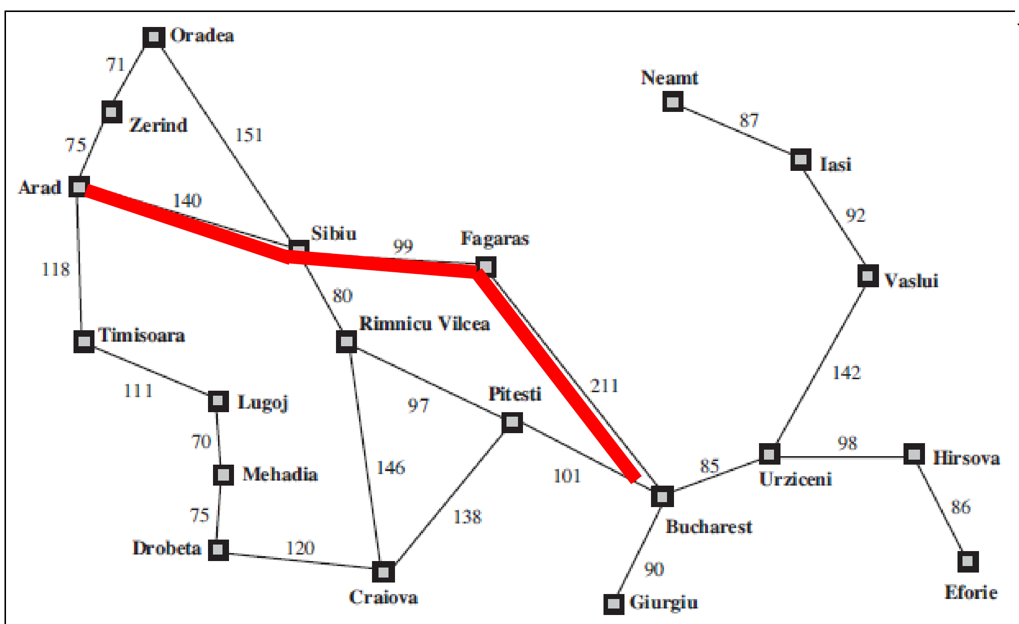


Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

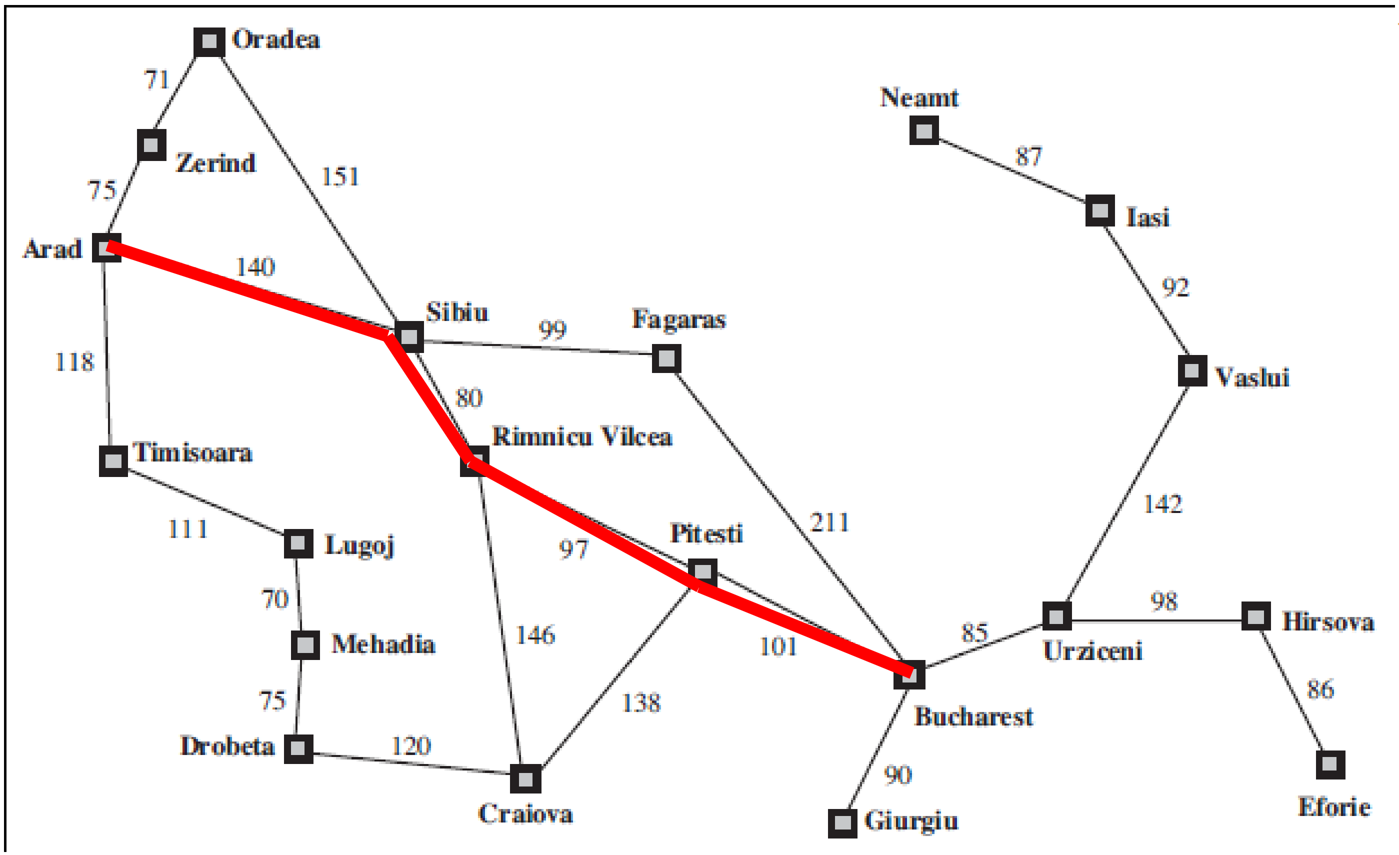




Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Optimal Path



Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Properties of greedy best-first search

- Complete?
  - Tree version can get stuck in loops.
  - Graph version is complete in finite spaces.
- Time? $O(b^m)$
  - A good heuristic can give **dramatic** improvement
- Space? $O(b^m)$
  - Graph search keeps all nodes in memory
  - A good heuristic can give **dramatic** improvement
- Optimal? No
  - E.g., Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest is shorter!

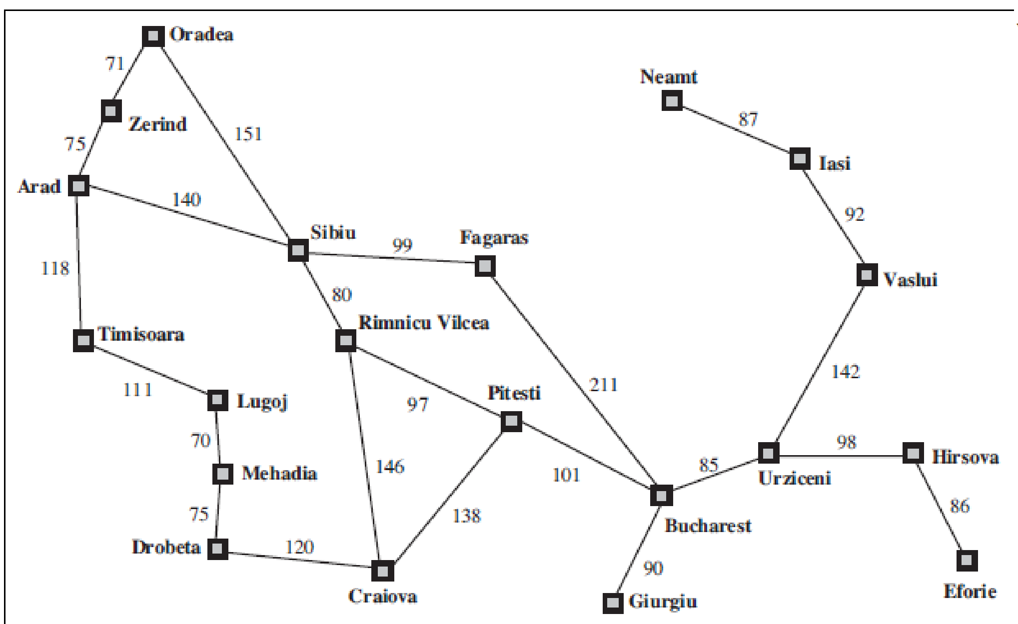
A^* search

- Idea: avoid paths that are already expensive
  - Generally the preferred simple heuristic search
  - Optimal if heuristic is:

    admissible (tree search)/consistent (graph search)
- Evaluation function $f(n) = g(n) + h(n)$
  - g(n) = known path cost so far to node n.
  - h(n) = <u>estimate</u> of (optimal) cost to goal from node n.
  - f(n) = g(n)+h(n)

    = <u>estimate</u> of total cost to goal through node n.
- *Priority queue sort function = f(n)*

A* tree search example



Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* tree search example: Simulated queue. City/f=g+h

- Next:
- Children:
- Expanded:
- Frontier: Arad/366=0+366

A* tree search example: Simulated queue. City/f=g+h

Arad/
366=0+366

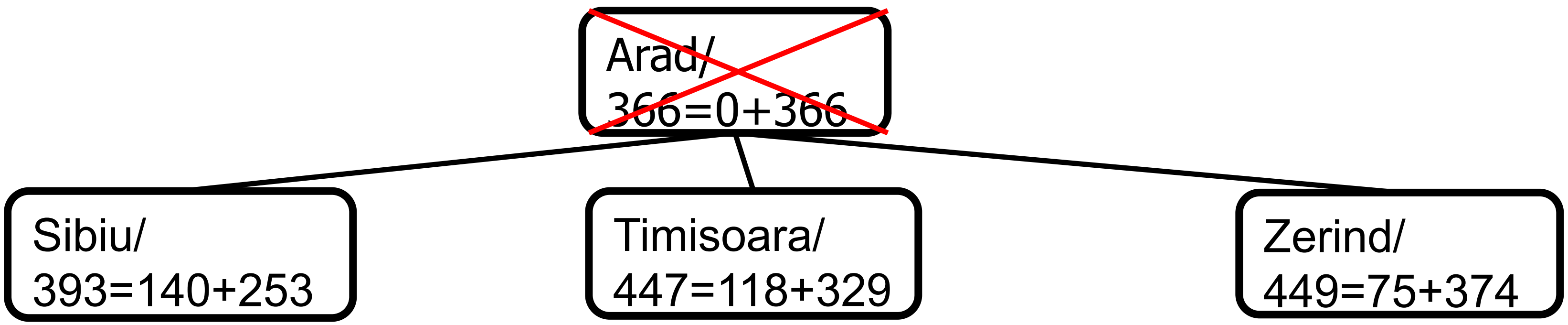
A* tree search example: Simulated queue. City/f=g+h

Arad/
366=0+366

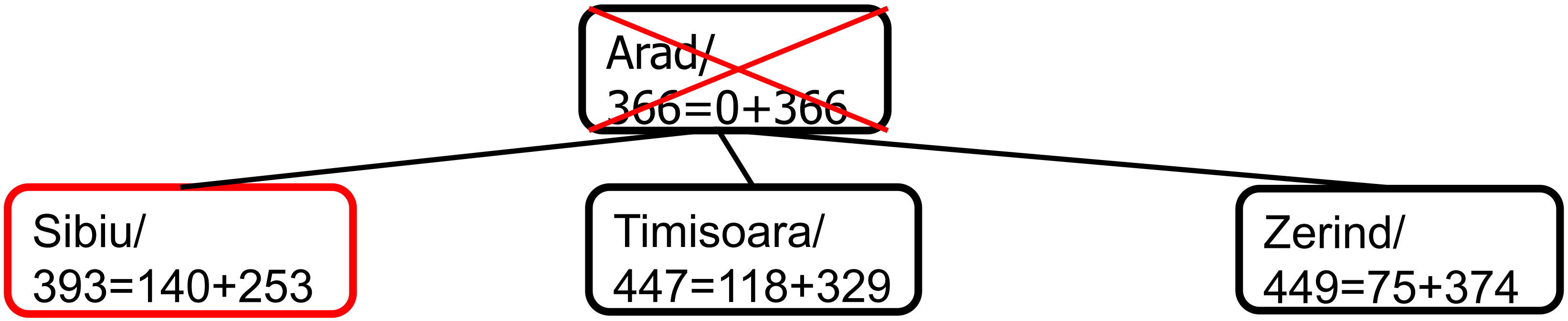
A* tree search example: Simulated queue. City/f=g+h

- Next: Arad/366=0+366
- Children: Sibiu/393=140+253, Timisoara/447=118+329, Zerind/449=75+374
- Expanded: Arad/366=0+366
- Frontier: ~~Arad/366=0+366~~, Sibiu/393=140+253, Timisoara/447=118+329, Zerind/449=75+374

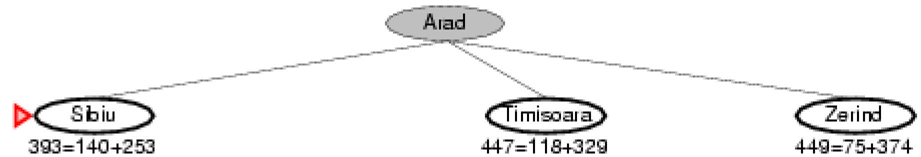
A* tree search example: Simulated queue. City/f=g+h

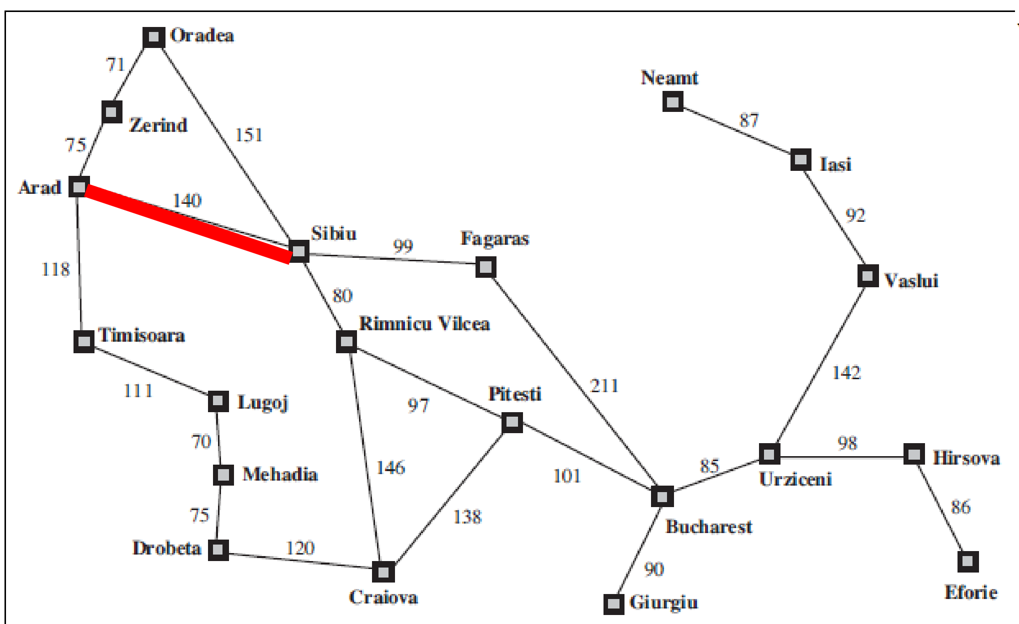


A^* tree search example: Simulated queue. City/f=g+h



A* tree search example





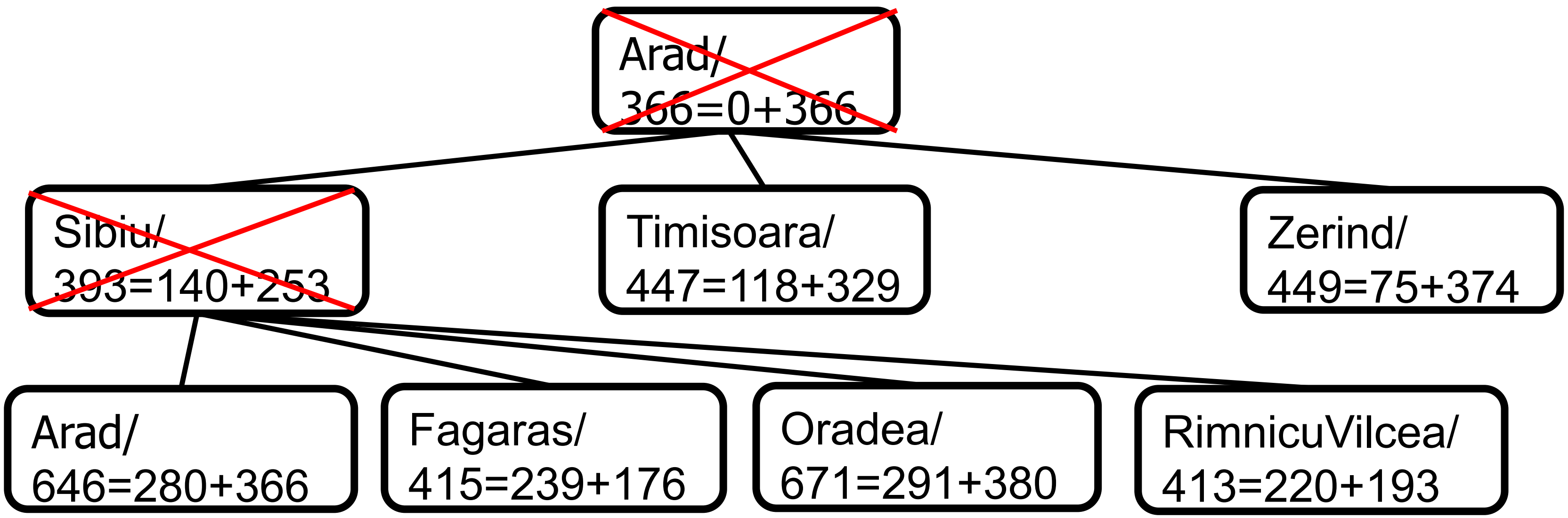
Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

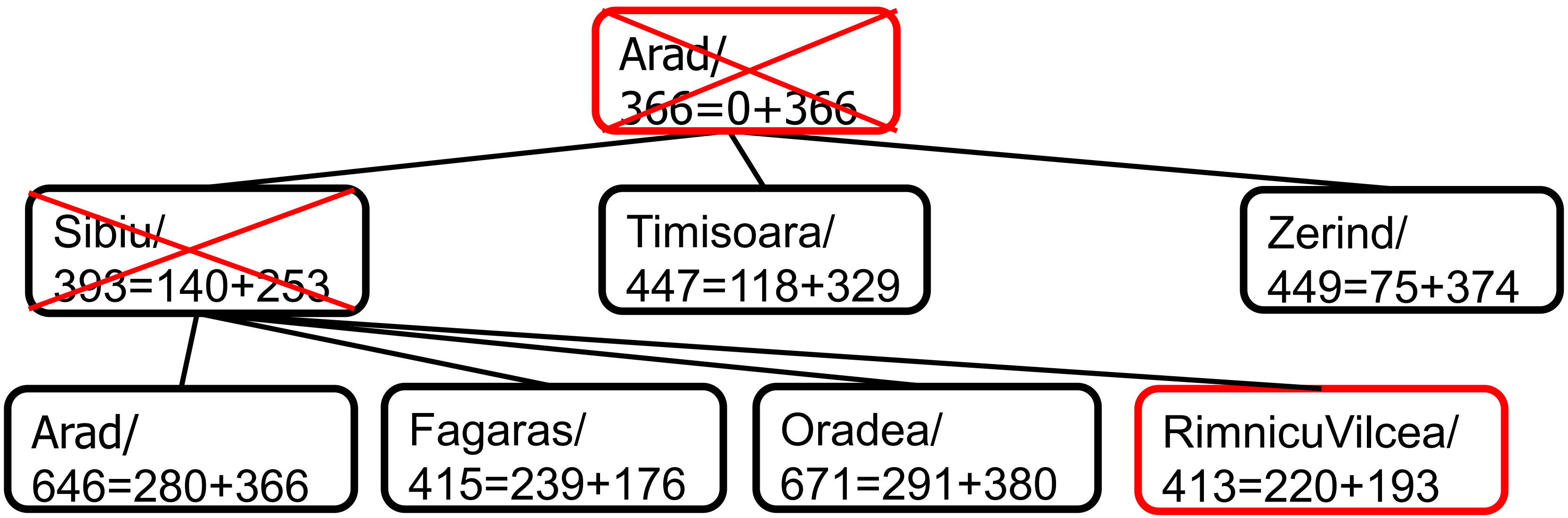
A* tree search example: Simulated queue. City/f=g+h

- Next: Sibiu/393=140+253
- Children: Arad/646=280+366, Fagaras/415=239+176, Oradea/671=291+380, RimnicuVilcea/413=220+193
- Expanded: Arad/366=0+366, Sibiu/393=140+253
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253,~~ Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, Fagaras/415=239+176, Oradea/671=291+380, RimnicuVilcea/413=220+193

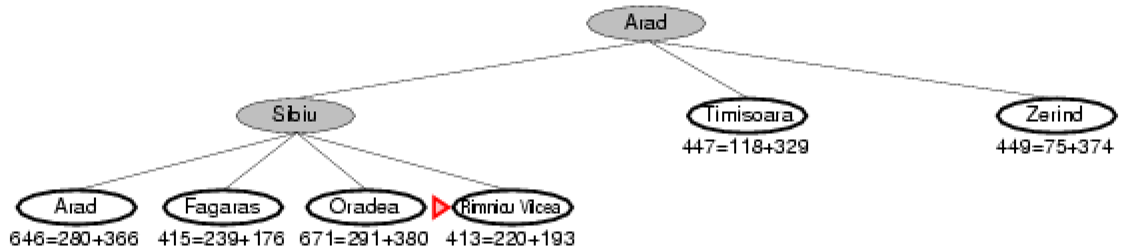
A* tree search example: Simulated queue. City/f=g+h

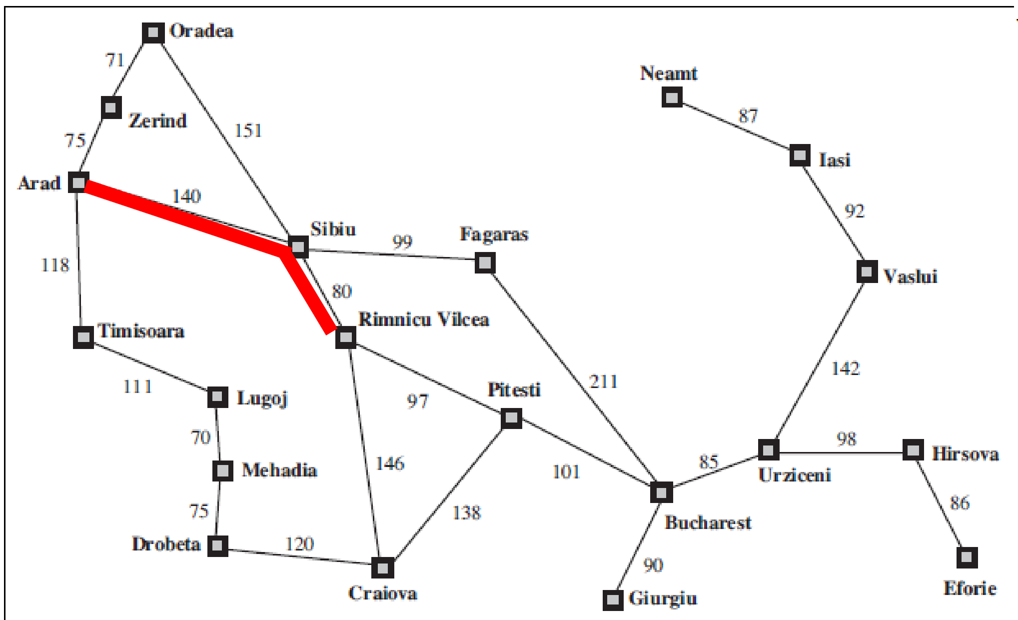


A* tree search example: Simulated queue. City/f=g+h



A* tree search example





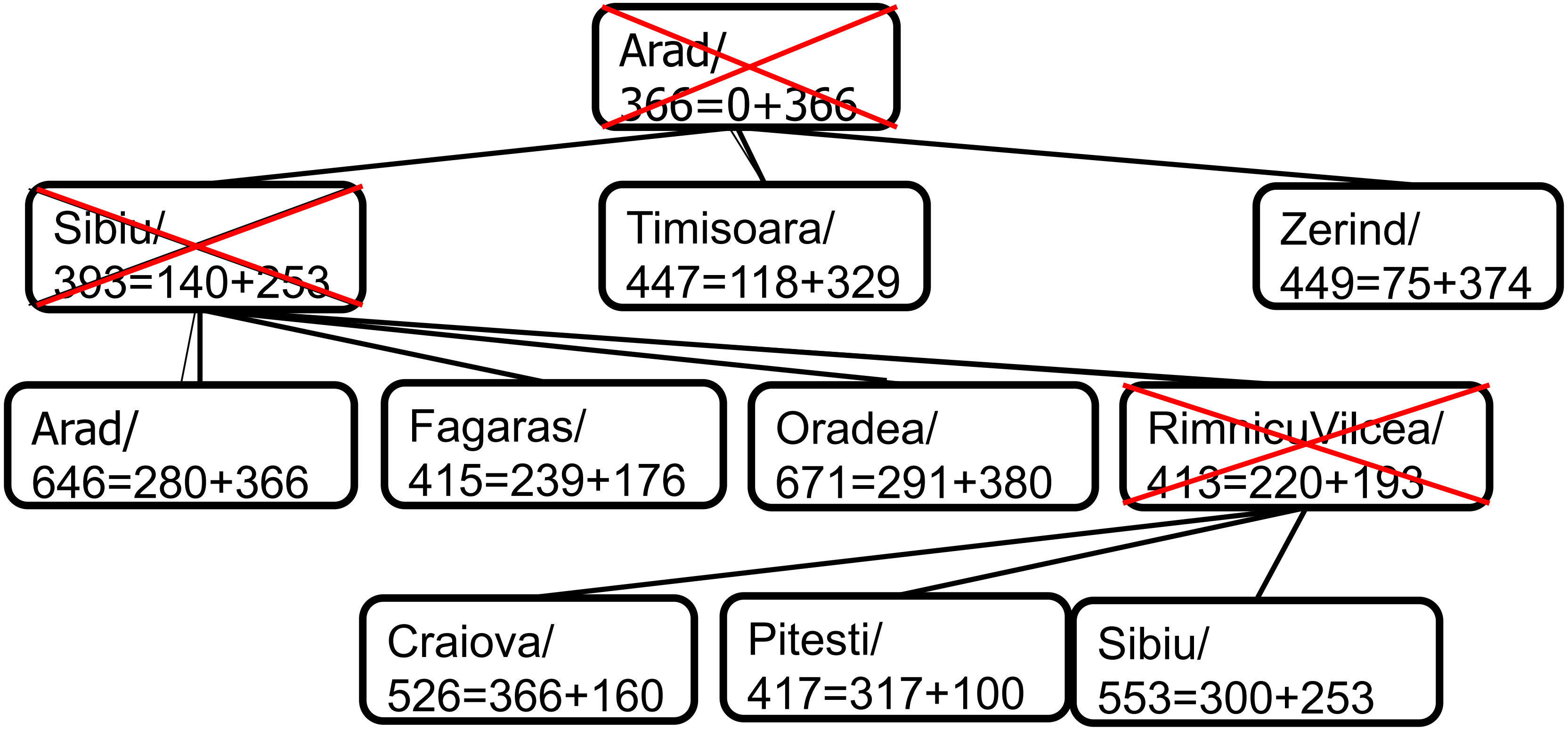
Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

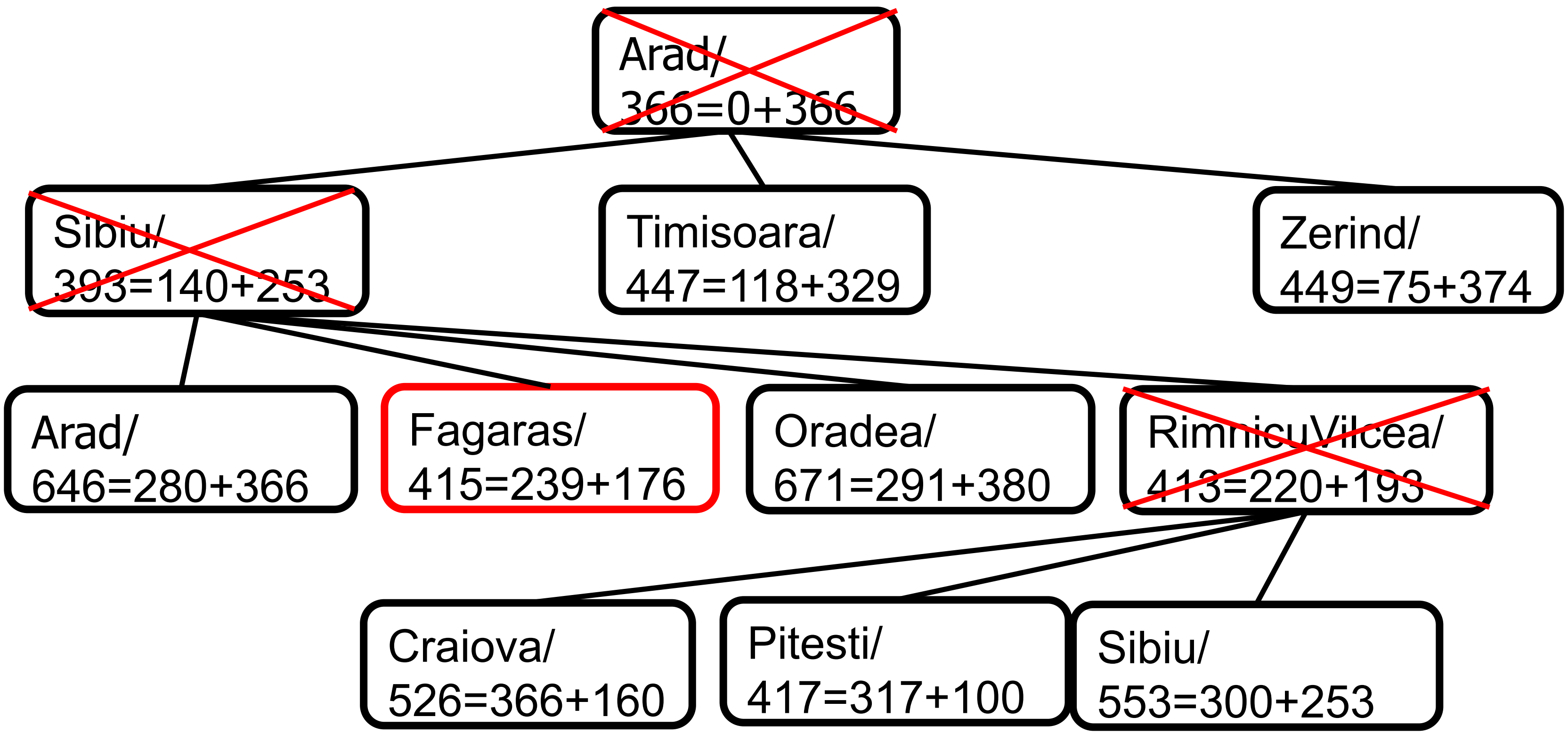
A* tree search example: Simulated queue. City/f=g+h

- Next: RimnicuVilcea/413=220+193
- Children: Craiova/526=366+160, Pitesti/417=317+100, Sibiu/553=300+253
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253,~~ Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, Fagaras/415=239+176, Oradea/671=291+380, ~~RimnicuVilcea/413=220+193,~~ Craiova/526=366+160, Pitesti/417=317+100, Sibiu/553=300+253

A* tree search example: Simulated queue. City/f=g+h

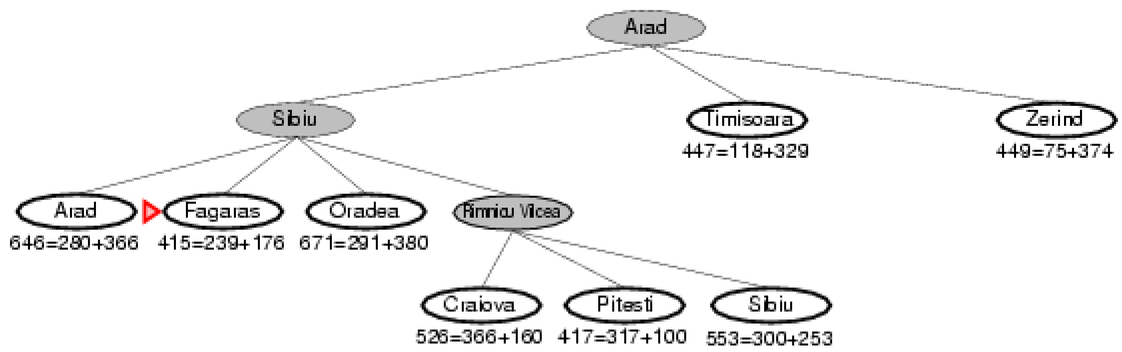


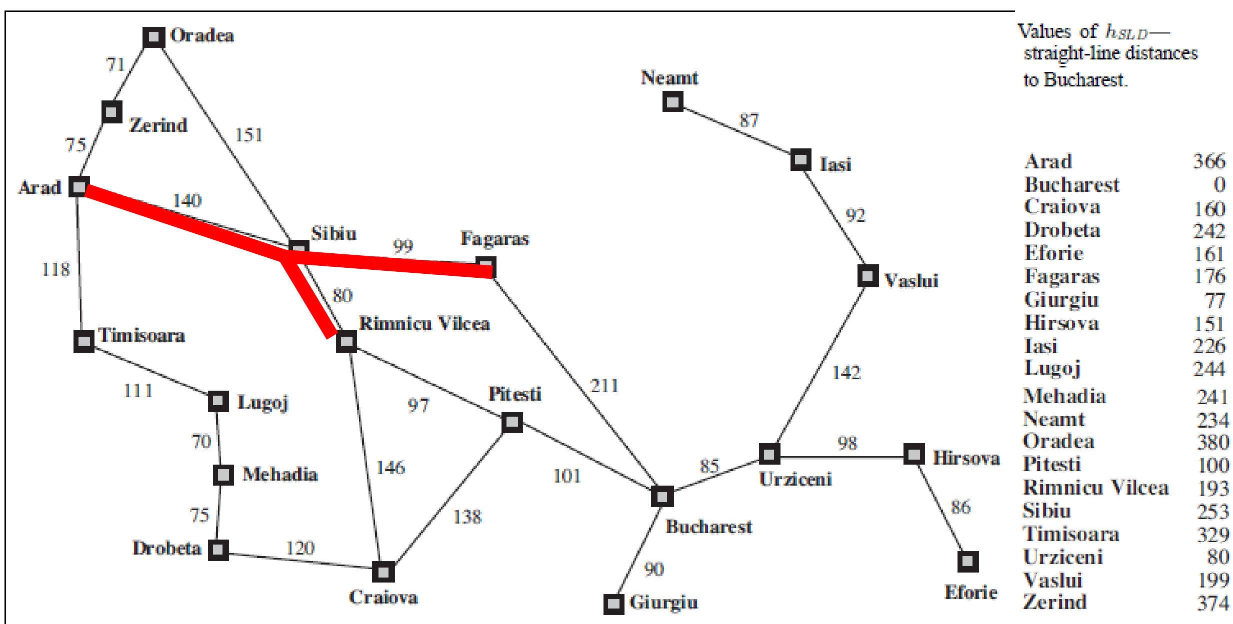
A^* search example: Simulated queue. City/f=g+h



A* tree search example

Note: The search below did not “back track.” Rather, both arms are being pursued in parallel on the queue.





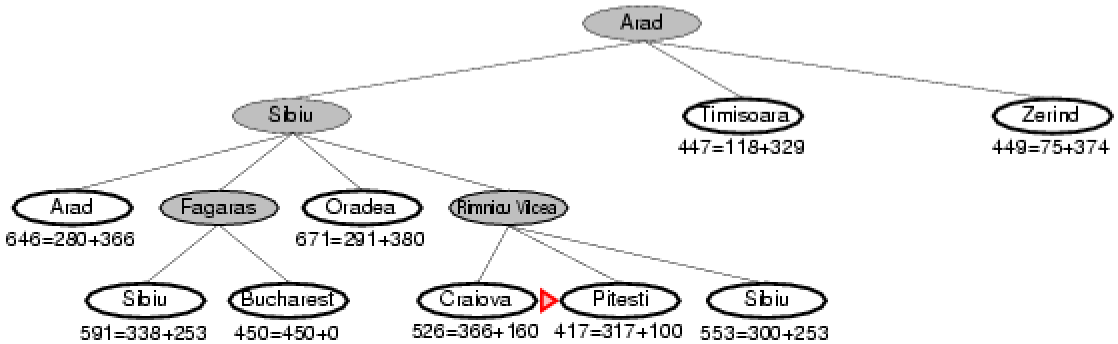
A* tree search example: Simulated queue. City/f=g+h

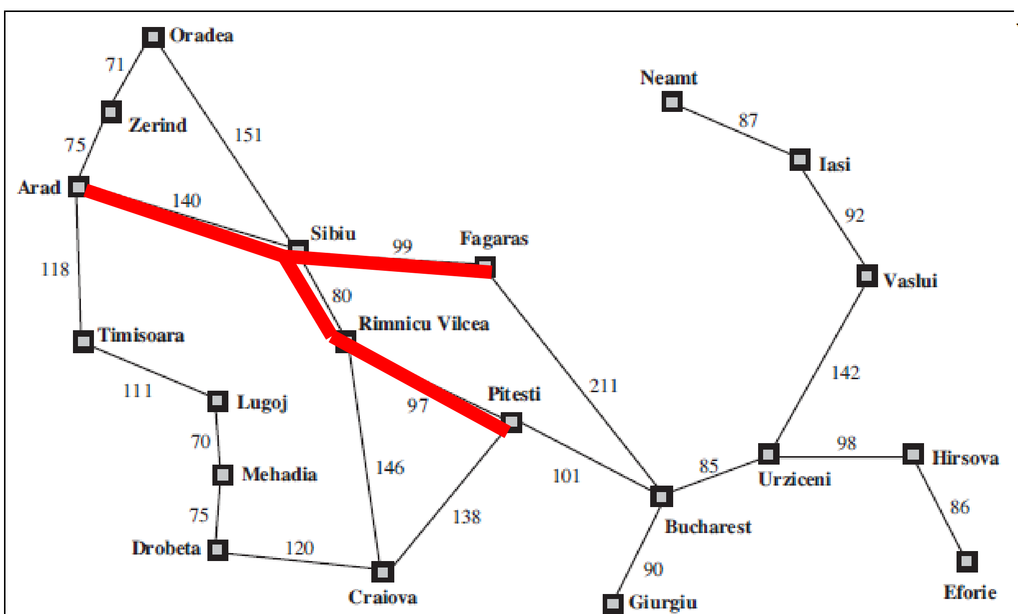
- Next: Fagaras/415=239+176
- Children: Bucharest/450=450+0, Sibiu/591=338+253
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253,~~ Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, ~~Fagaras/415=239+176,~~ Oradea/671=291+380, ~~RimnicuVilcea/413=220+193,~~ Craiova/526=366+160, Pitesti/417=317+100, Sibiu/553=300+253, Bucharest/450=450+0, ~~Sibiu/591=338+253~~

Delete higher-cost redundant nodes.

A* tree search example

Note: The search below did not "back track." Rather, both arms are being pursued in parallel on the queue.





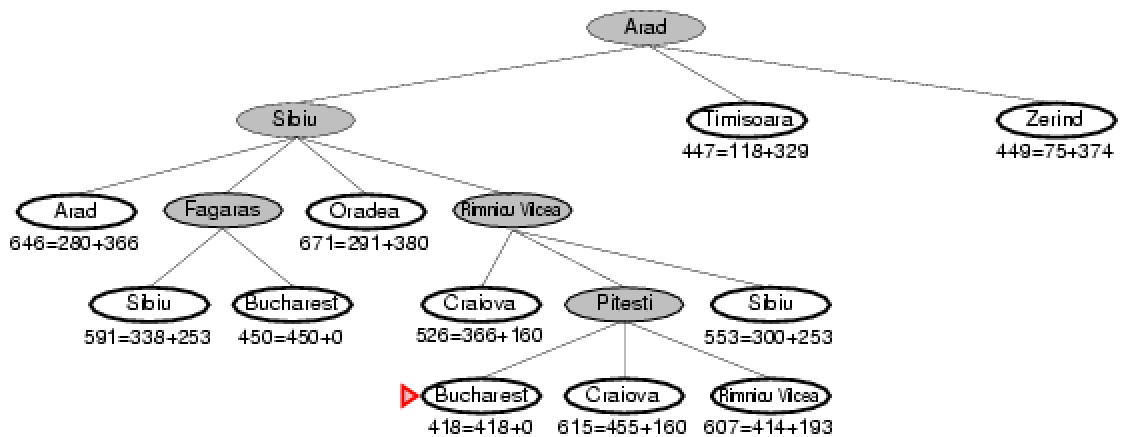
Values of h_{SLD}—straight-line distances to Bucharest.

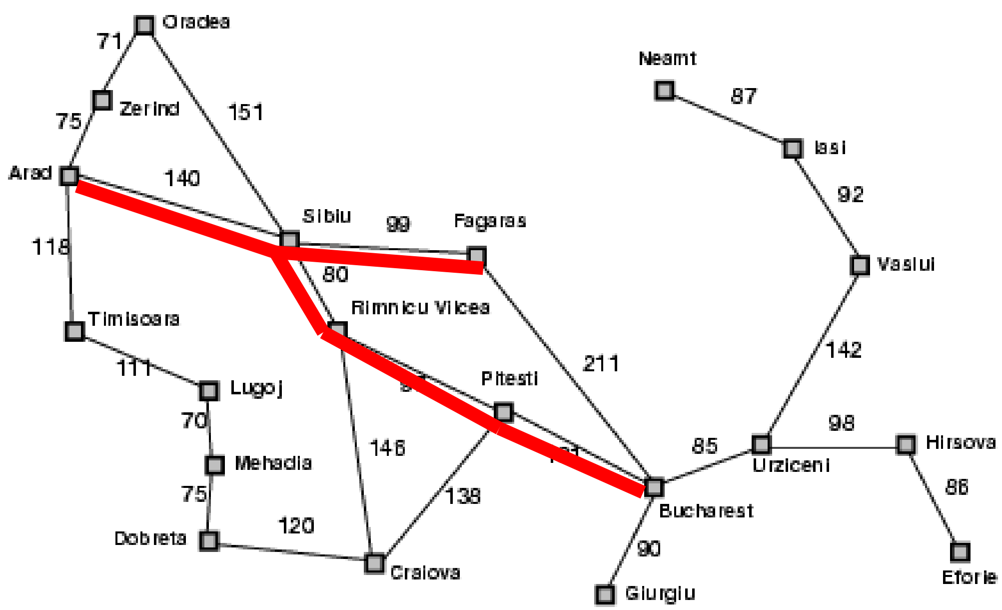
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* tree search example: Simulated queue. City/f=g+h

- Next: Pitesti/417=317+100
- Children: Bucharest/418=418+0, Craiova/615=455+160, RimnicuVilcea/607=414+193
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176, Pitesti/417=317+100
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253~~, Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, ~~Fagaras/415=239+176~~, Oradea/671=291+380, ~~RimnicuVilcea/413=220+193~~, Craiova/526=366+160, ~~Pitesti/417=317+100~~, Sibiu/553=300+253, Bucharest/450=450+0, Sibiu/591=338+253, Bucharest/418=418+0, Craiova/615=455+160, RimnicuVilcea/607=414+193

A* tree search example





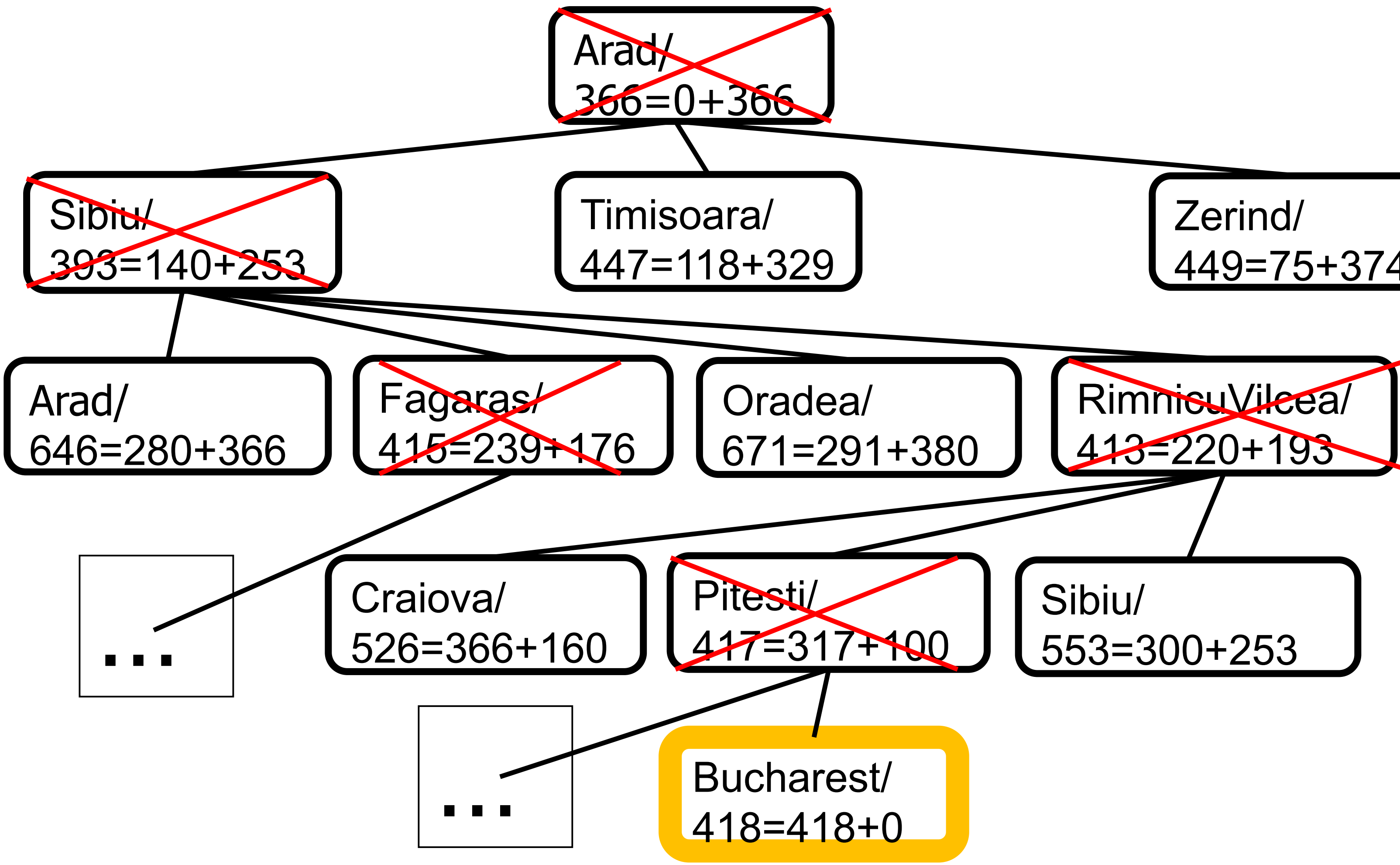
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* tree search example: Simulated queue. City/f=g+h

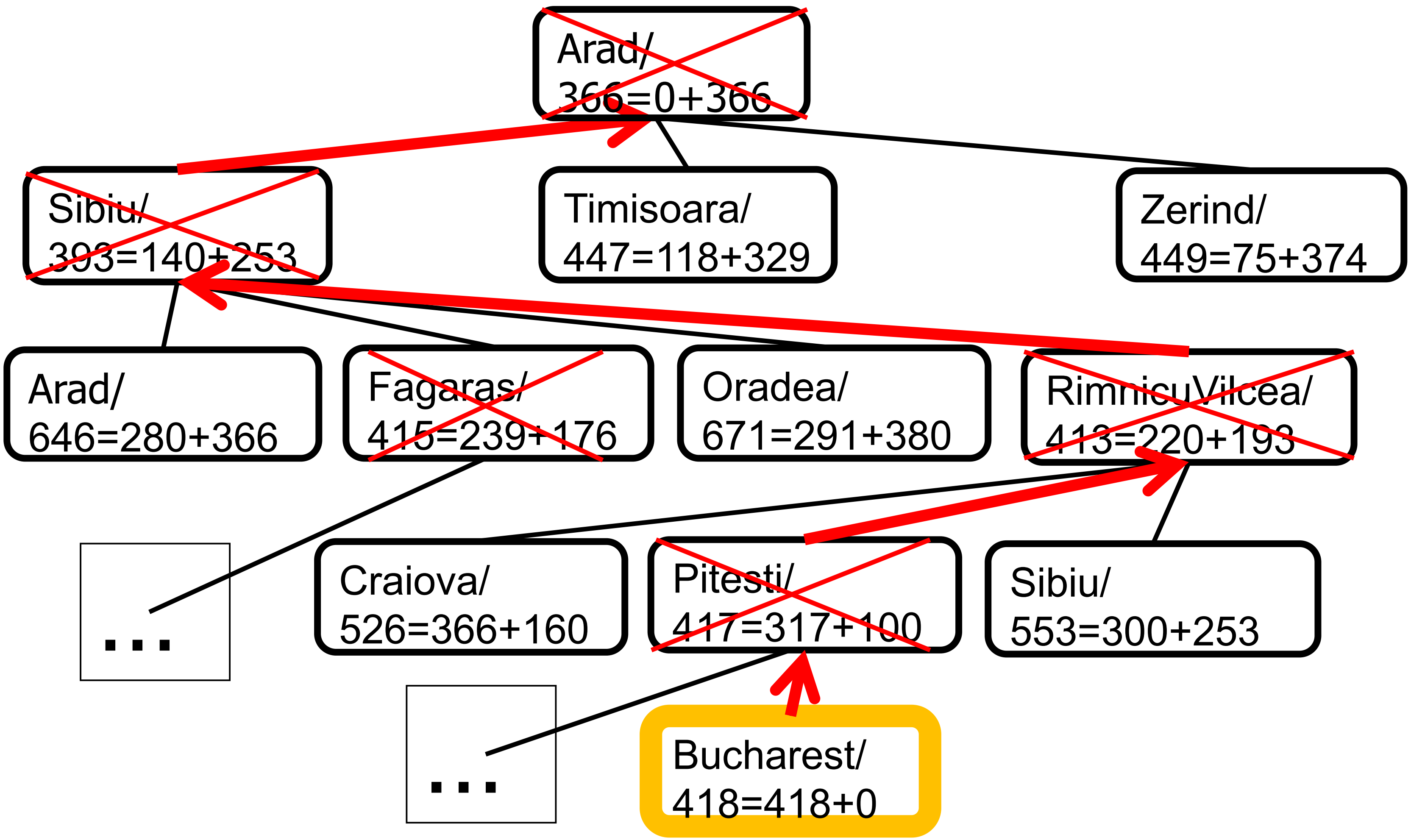
- Next: Bucharest/418=418+0
- Children: **None; goal test succeeds.**
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176, Pitesti/417=317+100, Bucharest/418=418+0
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253,~~ Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, ~~Fagaras/415=239+176,~~ Oradea/671=291+380, ~~RimnicuVilcea/413=220+193,~~ Craiova/526=366+160, ~~Pitesti/417=317+100,~~ Sibiu/553=300+253, Bucharest/450=450+0, Sibiu/591=338+253, ~~Bucharest/418=418+0,~~ Craiova/615=455+160, RimnicuVilcea/607=414+193

Note that the short expensive path stays on the queue. The long cheap path is found and returned.

A^* tree search example: Simulated queue. City/f=g+h



A^* tree search example: Simulated queue. City/f=g+h



Properties of A*

- Complete? Yes

  (unless there are infinitely many nodes with $f \leq f(G)$;
  can't happen if step-cost $\geq \varepsilon > 0$)
- Time/Space? Exponential $O(b^d)$

  except if: $$|h(n) - h^*(n)| \leq O(\log h^*(n))$$
- Optimal?

  (with: Tree-Search, admissible heuristic;
  Graph-Search, consistent heuristic)
- *Optimally Efficient?*

  (no optimal algorithm with same heuristic is guaranteed to expand fewer nodes)

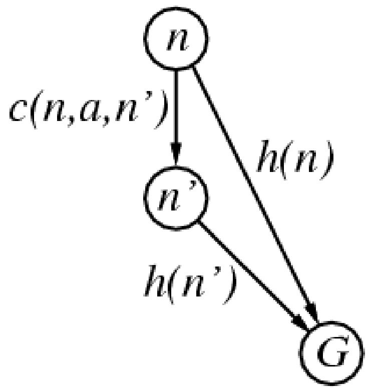
Admissible heuristics

- A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If $h(n)$ is admissible, A^* using `TREE-SEARCH` is optimal

Consistent heuristics (consistent => admissible)

- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,

$$\boxed{h(n) \le c(n,a,n') + h(n')}$$



- If *h* is consistent, we have

f(n') = g(n') + h(n') (by def.)
= g(n) + c(n,a,n') + h(n') (g(n')=g(n)+c(n.a.n'))
≥ g(n) + h(n) = f(n) (consistency)
f(n') ≥ f(n)

It's the triangle inequality !

- i.e., *f(n)* is non-decreasing along any path.

- Theorem:
If *h(n)* is consistent, A* using GRAPH−SEARCH is optimal

keeps all checked nodes in memory to avoid repeated states

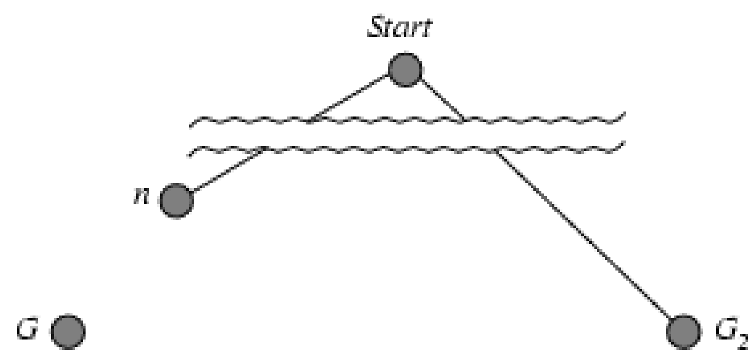
Optimality of A* (proof)

Tree Search, where *h(n)* is admissible

- Suppose some suboptimal goal G_2 has been generated and is in the frontier. Let n be an unexpanded node in the frontier such that n is on a shortest path to an optimal goal G.

We want to prove:
f(n) < f(G2)
(then A* will expand n before G2)

- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $f(G) = g(G)$ since $h(G) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G_2) > f(G)$ from above, with h=0
- $h(n) \le h^*(n)$ since h is admissible (*under*-estimate)
- $g(n) + h(n) \le g(n) + h^*(n)$ from above
- $f(n) \le f(G)$ since $g(n)+h(n)=f(n)$ & $g(n)+h^*(n)=f(G)$
- $f(n) < f(G2)$ from above



R&N pp. 95-98 proves the optimality of A* graph search with a consistent heuristic

Dominance

- IF $h_2(n) \geq h_1(n)$ for all n
  THEN h_2 dominates h_1
  - h_2 is almost always better for search than h_1
  - h_2 guarantees to expand no more nodes than does h_1
  - h_2 almost always expands fewer nodes than does h_1
  - Not useful unless both h_1 & h_2 are admissible/consistent

- Typical 8-puzzle search costs
  (average number of nodes expanded):
  - *d=12* IDS = 3,644,035 nodes
    $A^*(h_1)$ = 227 nodes
    $A^*(h_2)$ = 73 nodes
  - *d=24* IDS = too many nodes
    $A^*(h_1)$ = 39,135 nodes
    $A^*(h_2)$ = 1,641 nodes

CS-171 Final Review

- **Machine Learning Classifiers**
  - (R&N Ch. 18.5-18.12; 20.2)
- **Intro to Machine Learning**
  - (R&N Ch. 18.1-18.4)
- **Game (Adversarial) Search**
  - (R&N Ch. 5.1-5.4)
- **Local Search**
  - (R&N Ch. 4.1-4.2)
- **State Space Search**
  - (R&N Ch. 3.1-3.7)
- Questions on any topic
- Please review your quizzes & old tests