CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

Schedule

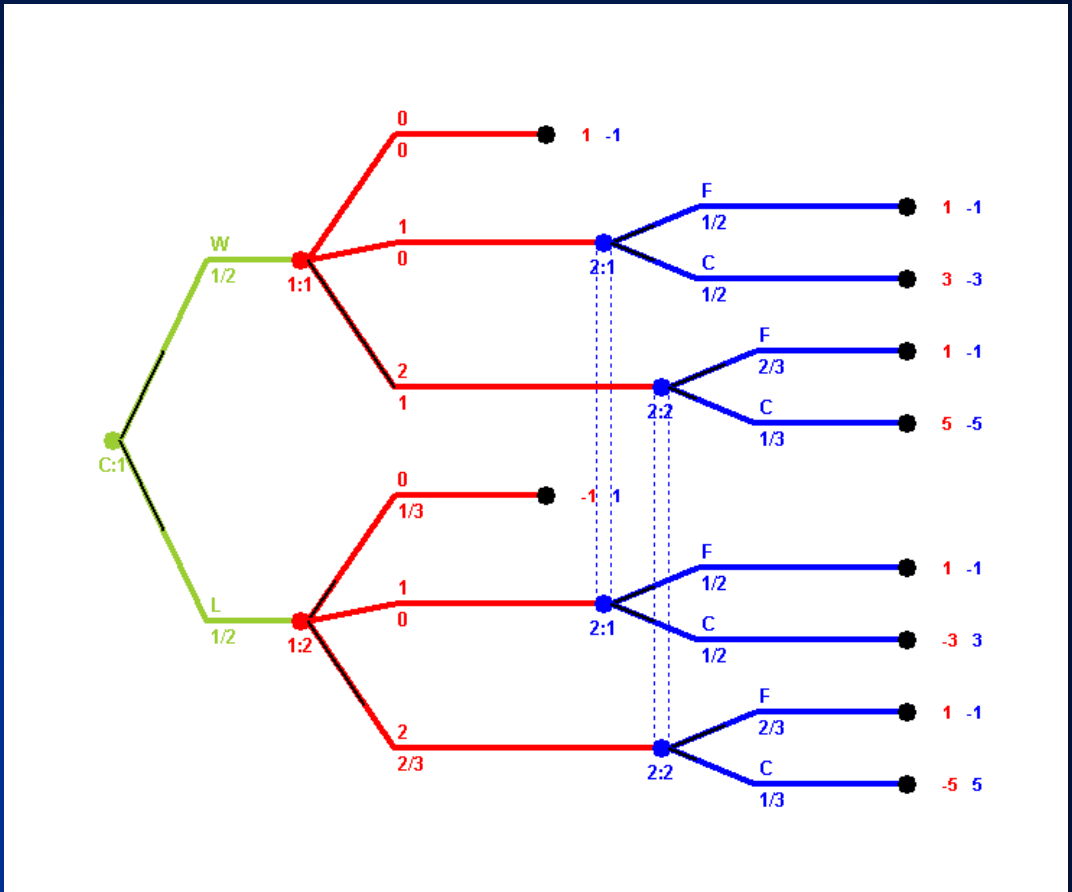
- 11/28, 11/30, 12/5: Machine learning (classification, regression, clustering, deep learning)
- 12/7: Project presentations and class project due
  - Project code due Monday 12/4 at 2PM on Moodle.
- Final exam on 12/14

Announcements

- HW4 out week of 11/14 (final homework assignment) due 12/1 (2:05pm in lecture or 2:00pm on Moodle)
  - https://www.cs.cmu.edu/~sganzfri/HW4_AI.pdf
- HW3 is mostly graded and will be returned on this Thursday with solutions.

Gambit

- http://gambit.sourceforge.net/gambit15/gui.html



Class project

- For the class project students will implement an agent for 3-player Kuhn poker. This is a simple, yet interesting and nontrivial, variant of poker that has appeared in the AAAI Annual Computer Poker Competition. The grade will be partially based on performance against the other agents in a class-wide competition, as well as final reports and presentations describing the approaches used. Students can work alone or in groups of up to 3.
- Link to play against optimal strategy for one-card poker:
  - http://www.cs.cmu.edu/~ggordon/poker/
- Paper on Nash equilibrium strategies for 3-player Kuhn poker
  - http://poker.cs.ualberta.ca/publications/AAMAS13-3pkuhn.pdf
- https://moodle.cis.fiu.edu/v3.1/mod/forum/discuss.php?d=21801

Multiagent systems (game theory)

- Strategic multiagent interactions occur in all fields
  - Economics and business: bidding in auctions, offers in negotiations
  - Political science/law: fair division of resources, e.g., divorce settlements
  - Biology/medicine: robust diabetes management (robustness against “adversarial” selection of parameters in MDP)
  - Computer science: theory, AI, PL, systems; national security (e.g., deploying officers to protect ports), cybersecurity (e.g., determining optimal thresholds against phishing attacks), internet phenomena (e.g., ad auctions)

Nash equilibria in two-player zero-sum games

- Zero exploitability – “unbeatable”
- Exchangeable
  - If (a,b) and (c,d) are NE, then (a,d) and (c,b) are too
- Can be computed in polynomial time by a linear programming (LP) formulation

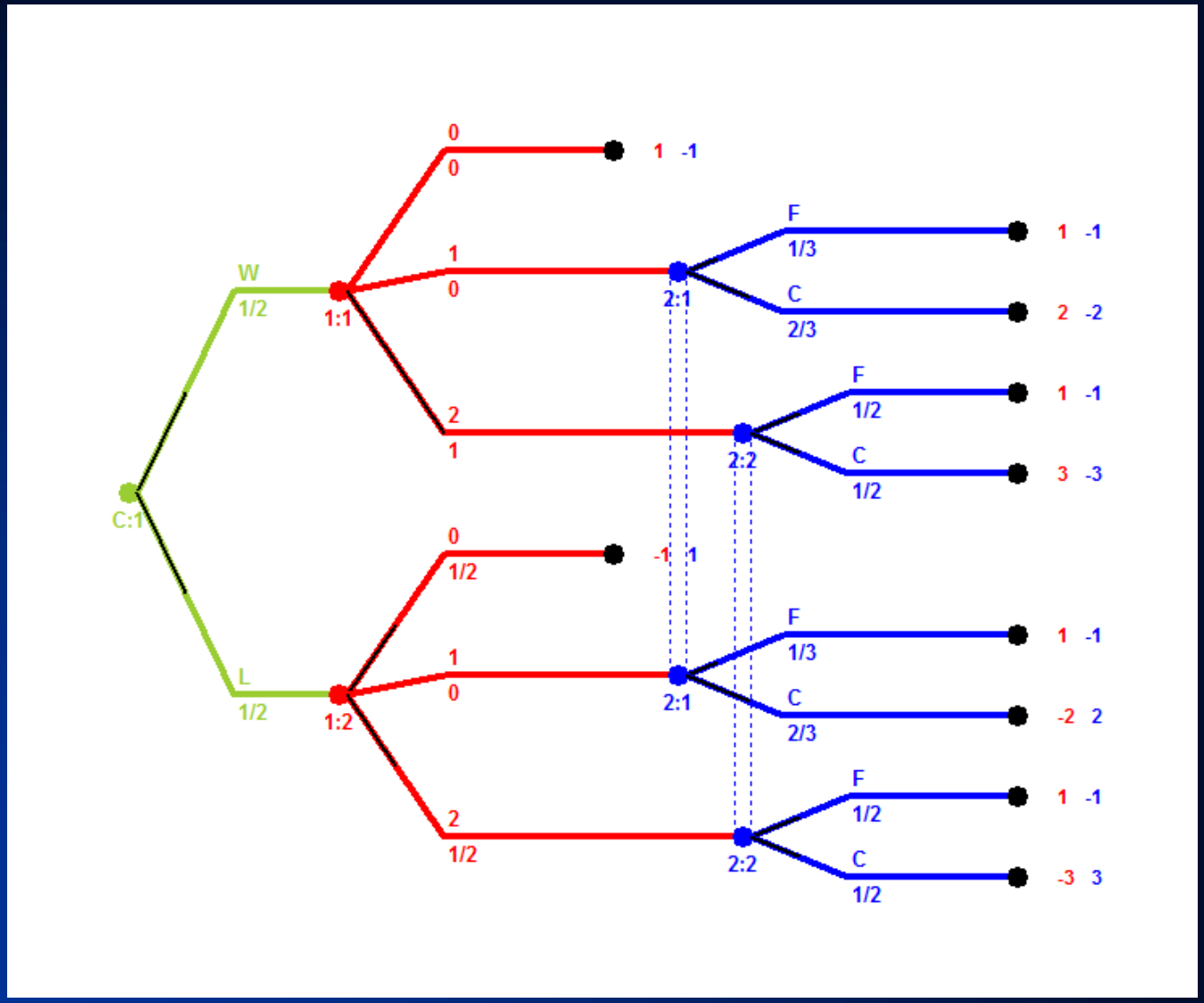
Nash equilibria in multiplayer and non-zero-sum games

- None of the two-player zero-sum results hold
- There can exist multiple equilibria, each with different payoffs to the players
- If one player follows one equilibrium while other players follow a different equilibrium, overall profile is not guaranteed to be an equilibrium
- If one player plays an equilibrium, he could do worse if the opponents deviate from that equilibrium
- Computing an equilibrium is PPAD-hard

Imperfect information

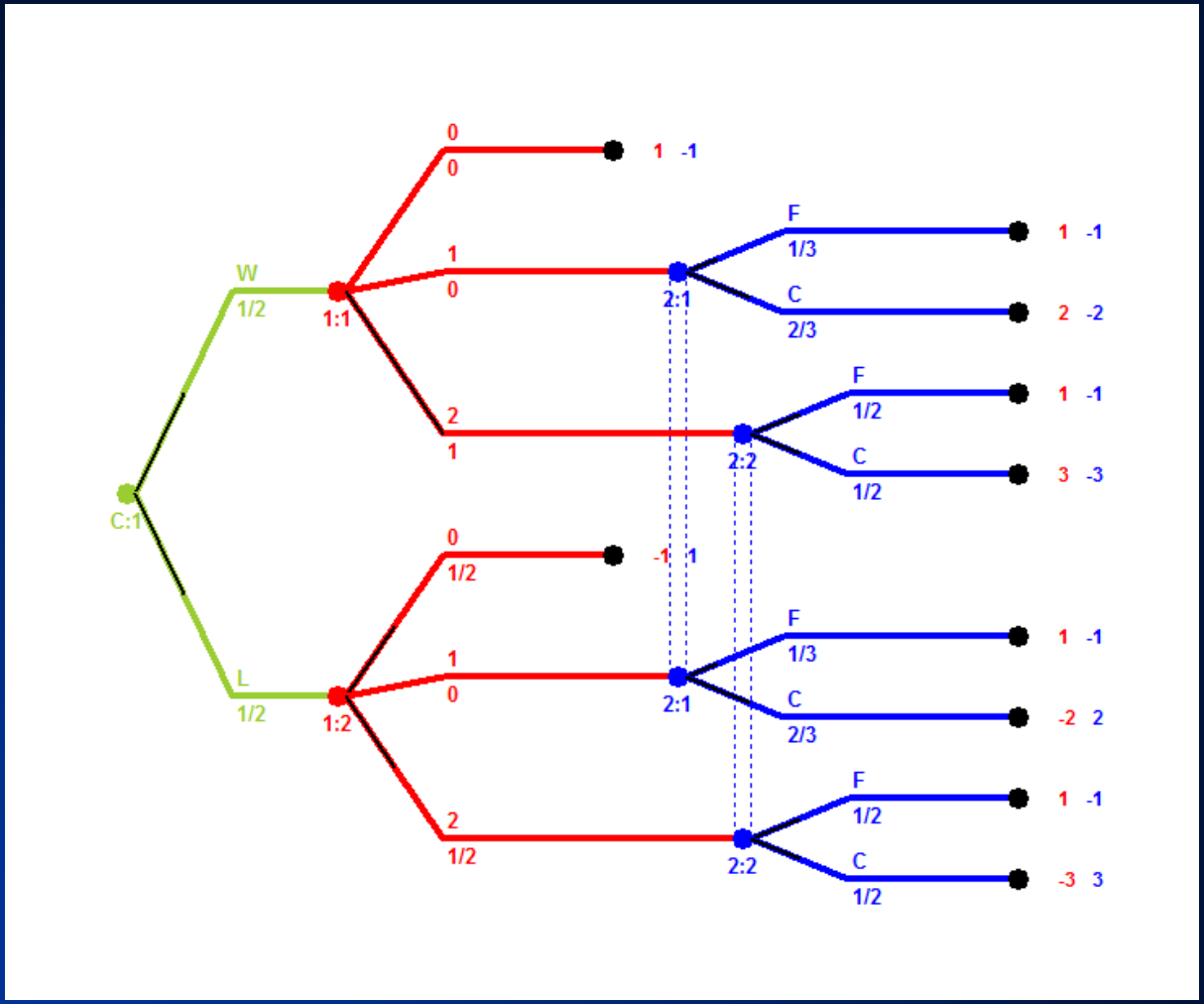
- In many important games, there is information that is private to only some agents and not available to other agents
  - In auctions, each bidder may know his own valuation and only know the distribution from which other agents' valuations are drawn
  - In poker, players may not know private cards held by other players

Extensive-form representation



Extensive-form games

- Two-player zero-sum EFGs can be solved in polynomial time by linear programming
  - Scales to games with up to 10^8 states
- Iterative algorithms (CFR and EGT) have been developed for computing an ε-equilibrium that scale to games with 10^{17} states
  - CFR also applies to multiplayer and general sum games, though no significant guarantees in those classes
  - (MC)CFR is self-play algorithm that samples actions down tree and updates regrets and average strategies stored at every information set


C:1
W
1/2
L
1/2
1:1
1:2
0
0
1
0
2
1
0
1/2
1
0
2
1/2
1 -1
-1 1
2:1
2:2
2:1
2:2
F
1/3
C
2/3
F
1/2
C
1/2
F
1/3
C
2/3
F
1/2
C
1/2
1 -1
2 -2
1 -1
3 -3
1 -1
-2 2
1 -1
-3 3

WL/12	CC	CF	FC	FF
00	0	0	0	0
01	-0.5	-0.5	1	1
02	-1	1	-1	1
10				
11				
12				
20				
21				
22				

Extensive-form game

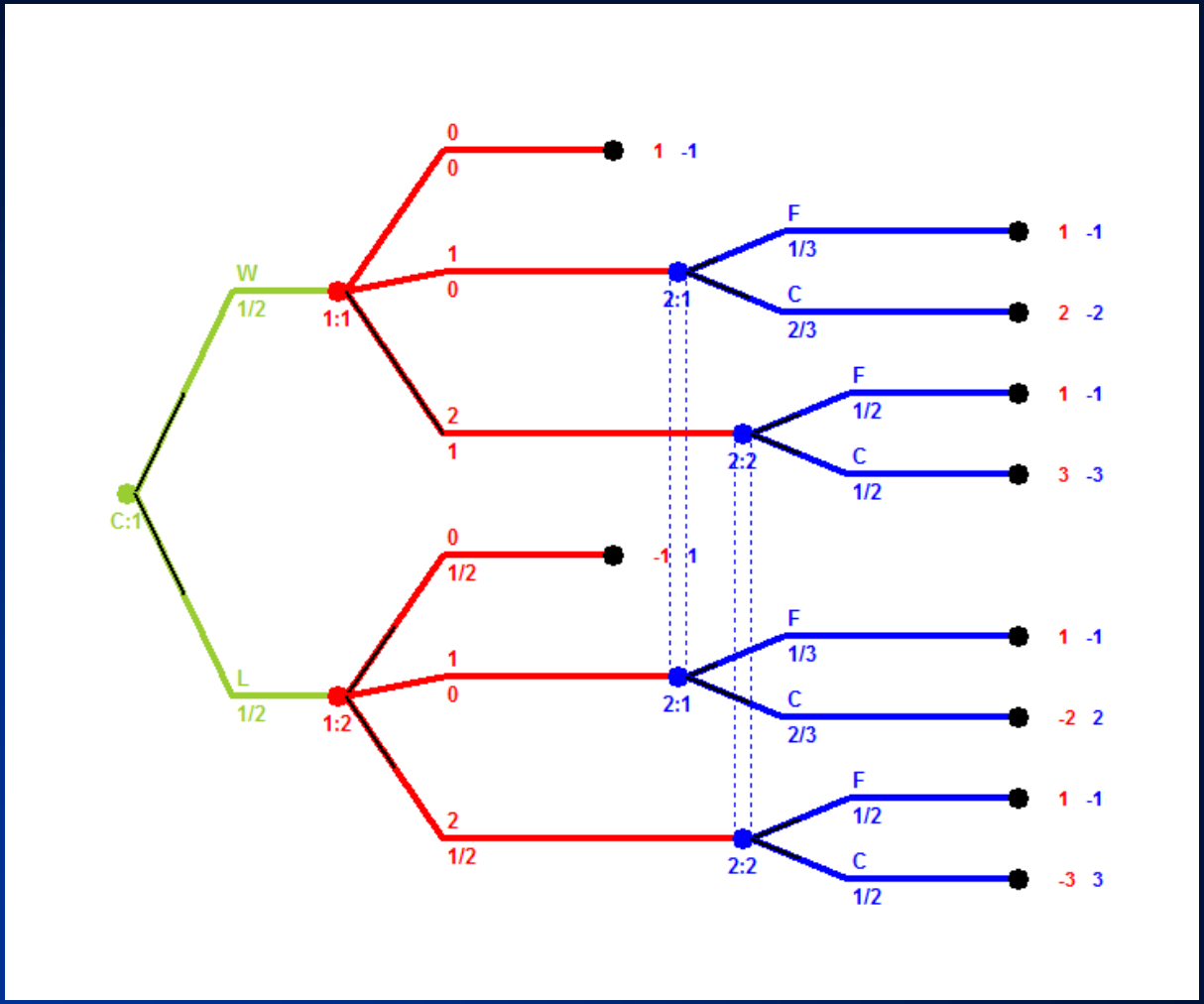
- A game in **extensive form** is given by a *game tree*, which consists of a directed graph in which the set of vertices represents positions in the game, and a distinguished vertex, called the *root*, represents the starting position of the game. A vertex with no outgoing edges represents a terminal position in which play ends. To each terminal vertex corresponds an outcome that is realized when the play terminates at that vertex. Any nonterminal vertex represents either a chance move (e.g., a toss of a die or a shuffle of a deck of cards) or a move of one of the players. To any chance-move vertex corresponds a probability distribution over edges emanating from that vertex, which correspond to the possible outcomes of the chance move.

Perfect vs. imperfect information

- To describe games with imperfect information, in which players do not necessarily know the full board position (like poker), we introduce the notion of *information sets*. An information set of a player is a set of decision vertices of the player that are indistinguishable by him given his information at that stage of the game. A game of *perfect information* is a game in which all information sets consist of a single vertex. In such a game whenever a player is called to take an action, he knows the exact history of actions and chance moves that led to that position.

- A *strategy* of a player is a function that assigns to each of his information sets an action available to him at that information set. A path from the root to a terminal vertex is called a *play* of the game. When the game has no chance moves, any vector of strategies (one for each player) determines the play of the game, and hence the outcome. In a game with chance moves, any vector of strategies determines a probability distribution over the possible outcomes of the game.

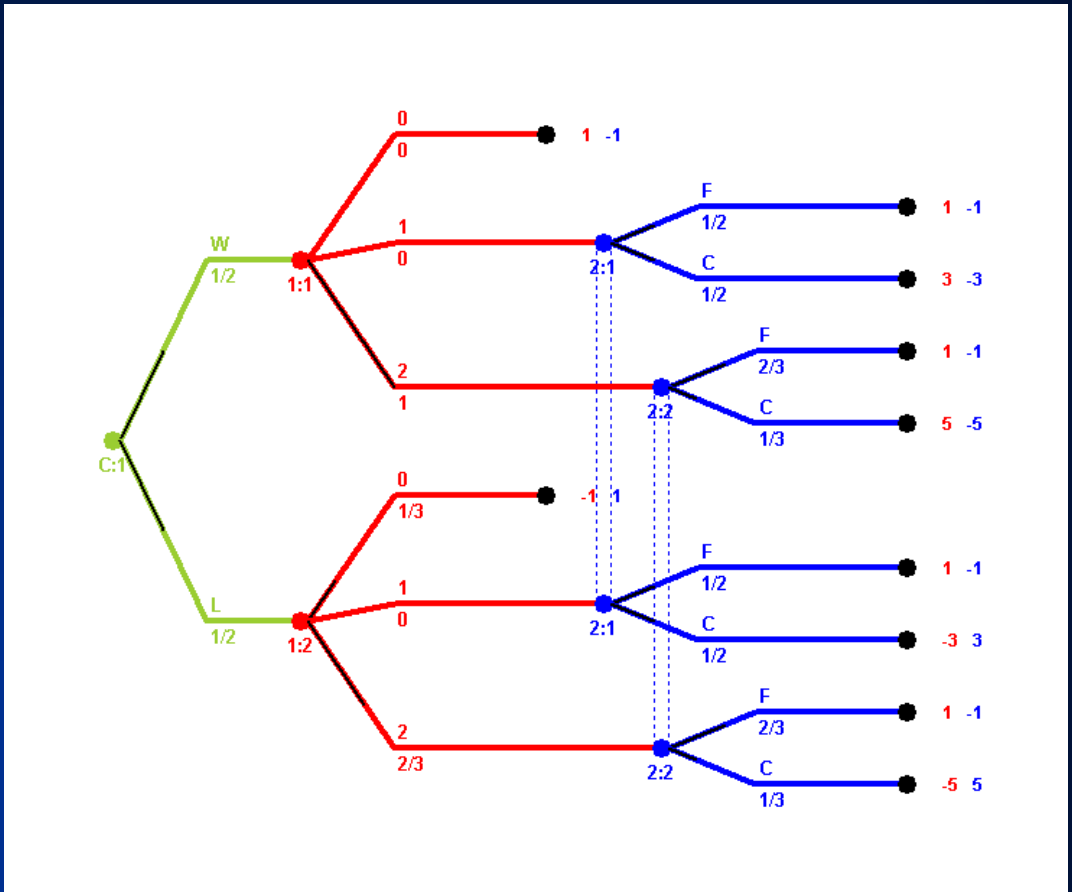
- Every extensive-form game can be converted to an equivalent strategic-form game, and therefore all the prior concepts and theoretical results (e.g., domination, security level, mixed strategies, Nash equilibrium, Minmax Theorem) will apply. However, this conversion produces a strategic-form game that has size that is exponential in the size of the original game tree, and is infeasible for large games. Therefore, we would like do develop algorithms that operate directly on extensive-form games and avoid the conversion to strategic form games.


C:1
W
1/2
L
1/2
1:1
1:2
0
0
1
0
2
1
0
1/2
1
0
2
1/2
1 -1
-1 1
2:1
2:2
2:1
2:2
F
1/3
C
2/3
F
1/2
C
1/2
F
1/3
C
2/3
F
1/2
C
1/2
1 -1
2 -2
1 -1
3 -3
1 -1
-2 2
1 -1
-3 3

WL/12	CC	CF	FC	FF
00	0	0	0	0
01	-0.5	-0.5	1	1
02	-1	1	-1	1
10				
11				
12				
20				
21				
22				

Gambit

- http://gambit.sourceforge.net/gambit15/gui.html



Algorithms for game solving

- Two-player zero-sum games: there exists a linear programming formulation and it can be solved in polynomial time.
- For two player “general-sum” and games with more than two players, it is PPAD-hard (though not NP-hard), and widely conjectured no efficient algorithms exist.
- For two-player zero-sum extensive-form games, there also exists a linear-programming formulation, despite the fact that converting it to normal-form would involve an exponential blowup in size of the game tree.

Computing Nash equilibria of two-player zero-sum games

- Consider the game $G = (\{1,2\}, A_1 \times A_2, (u_1, u_2))$.
- Let U^*_i be the expected utility for player i in equilibrium (the value of the game); since the game is zero-sum, $U^*_1 = - U^*_2$.
- Recall that the Minmax Theorem tells us that U^*_1 holds constant in all equilibria and that it is the same as the value that player 1 achieves under a minmax strategy by player 2.
- Using this result, we can formulate the problem of computing a Nash equilibrium as the following optimization:

Minimize U^*_1

Subject to $\Sigma_{k \text{ in } A2}\, u_1,(a^j_1, a^k_2) * s^k_2 <= U^*_1$ for all j in A_1

$\Sigma_{k \text{ in } A2}\, s^k_2 = 1$

$s^k_2 >= 0$ for all k in A_2

$$\begin{aligned} &\text{Minimize} && U^*_1 && \\ &\text{Subject to} && \Sigma_{k \text{ in } A2}\, u_1(a^j_1, a^k_2) * s^k_2 <= U^*_1 && \text{for all } j \text{ in } A_1 \\ & && \Sigma_{k \text{ in } A2}\, s^k_2 = 1 && \\ & && s^k_2 >= 0 && \text{for all } k \text{ in } A_2 \end{aligned}$$

- Note that all of the utility terms $u_1(*)$ are constants while the mixed strategy terms s^k_2 and U^*_1 are variables.

$$\begin{aligned} &\text{Minimize} && U^*_1 \\ &\text{Subject to} && \Sigma_{k \text{ in } A2}\, u_1(a^j_1, a^k_2) * s^k_2 <= U^*_1 && \text{for all } j \text{ in } A_1 \\ & && \Sigma_{k \text{ in } A2}\, s^k_2 = 1 \\ & && s^k_2 >= 0 && \text{for all } k \text{ in } A_2 \end{aligned}$$

- First constraint states that for every pure strategy j of player 1, his expected utility for playing any action j in A_1 given player 2's mixed strategy s_1 is at most U^*_1. Those pure strategies for which the expected utility is exactly U^*_1 will be in player 1's best response set, while those pure strategies leading to lower expected utility will not.
- As mentioned earlier, U^*_1 is a variable; we are selecting player 2's mixed strategy in order to minimize U^*_1 subject to the first constraint. Thus, player 2 plays the mixed strategy that minimizes the utility player 1 can gain by playing his best response.

$$\begin{aligned} &\text{Minimize} && U^*_1 && \\ &\text{Subject to} && \Sigma_{k \text{ in } A2}\, u_1(a^j_1, a^k_2) * s^k_2 <= U^*_1 && \text{for all } j \text{ in } A_1 \\ & && \Sigma_{k \text{ in } A2}\, s^k_2 = 1 && \\ & && s^k_2 >= 0 && \text{for all } k \text{ in } A_2 \end{aligned}$$

- The final two constraints ensure that the variables s^k_2 are consistent with their interpretation as probabilities. Thus, we ensure that they sum to 1 and are nonnegative.

Learning in games

- In game theory, **fictitious play** is a learning rule first introduced by George W. Brown. In it, each player presumes that the opponents are playing stationary (possibly mixed) strategies. At each round, each player thus best responds to the empirical frequency of play of their opponent. Such a method is of course adequate if the opponent indeed uses a stationary strategy, while it is flawed if the opponent's strategy is non-stationary. The opponent's strategy may for example be conditioned on the fictitious player's last move.

Fictitious play

- Simple “learning” update rule
- Initially proposed as an iterative method for computing Nash equilibria in zero-sum games, not as a learning model!
- Brown, G.W. (1951) “Iterative Solutions of Games by Fictitious Play”

- Algorithm:

  Initialize beliefs about the opponent’s strategy

  Repeat:

  1) Play a best response to the assessed strategy of the opponent

  2) Observe the opponent’s actual play and update beliefs accordingly

- In fictitious play, the agent believes that his opponent is playing the mixed strategy given by the empirical distribution of the opponent's previous actions. That is, if A is the set of the opponent's actions, and for every a in A we let w(a) be the number of times that the opponent has played action a, then the agent assess the probability of a in the opponent's mixed strategy as
  - $P(a) = w(a) / \sum_{a' \text{ in } A} w(a')$

- For example, in a repeated Prisoner's Dilemma game, if the opponent has played C, C, D, C, D in the first five games, before the sixth game he is assumed to be playing the mixed strategy (0.6, 0.4).
- In general the tie-breaking rule chosen has little effect on the results of fictitious play.
- On the other hand, fictitious play is very sensitive to the players' initial beliefs. This choice, which can be interpreted as action counts that were observed before the start of the game, can have a radical impact on the learning process. Note that one must pick some nonempty prior belief for each agent; the prior beliefs cannot be (0,…0), since this does not define a meaningful mixed strategy.

	Heads	Tails
Heads	1, -1	-1, 1
Tails	-1, 1	1, -1

Round	1's action	2's action	1's beliefs	2's beliefs
0			(1.5,2)	(2,1.5)
1	T	T	(1.5,3)	(2,2.5)
2	T	H	(2.5,3)	(2,3.5)
3	T	H	(3.5,3)	(2,4.5)
4	H	H	(4.5,3)	(3,4.5)
5	H	H	(5.5,3)	(4,4.5)
6	H	H	(6.5,3)	(5,4.5)
7	H	T	(6.5,4)	(6,4.5)
⋮	⋮	⋮	⋮	⋮

Table 7.1: Fictitious play of a repeated game of Matching Pennies.

- As the number of rounds tends to infinity, the empirical distribution of the play of each player will converge to (0.5,0.5). If we take this distribution to be the mixed strategy of each player, the play converges to the unique Nash equilibrium of the normal form stage game, that in which each player plays the mixed strategy (0.5,0.5).

Machine learning

- An agent is **learning** if it improves its performance on future tasks after making observations about the world. Learning can range from the trivial, as exhibited by jotting down a phone number, to the profound, as exhibited by Albert Einstein, who inferred a new theory of the universe.
- We will start by concentrating on one class of learning problem, which seems restricted but actually has vast applicability: from a collection of input-output pairs, learn a function that predicts the output for new inputs.

Machine learning

- Why would we want an agent to learn? If the design of the agent can be improved, why wouldn’t the designers just program in that improvement to begin with? There are three main reasons.

- First, the designers cannot anticipate all possible situations that the agent might find itself in. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters.

- Second, the designers cannot anticipate all changes over time; a program designed to predict tomorrow's stock market prices must learn to adapt when conditions change from boom to bust.

- Third, sometimes human programmers have no idea how to program a solution themselves. For example, most people are good at recognizing the faces of family members, but even the best programmers are unable to program a computer to accomplish that task, except by using learning algorithms.

Supervised learning

- The task of supervised learning is this: Given a **training set** of N example input-output pairs $(x_1, y_1),(x_2, y_2),\ldots,(x_N, y_N)$,
- Where each y_j was generated by an unknown function $y = f(x)$, discover a function h that approximates the true function f.
- Example: x_i, can be True/False for whether email says "Prize" in it, and y_i can be True/False for whether or not it is Spam.
- x and y can be any value, they need not be numbers.
  - E.g., x can be {red, green, blue} for jacket color, and y can be price.
- The function h is a **hypothesis**. Learning is a search through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set.

Supervised learning

- To measure the accuracy of a hypothesis we give it a **test set** of examples that are distinct from the training set.
  - What would happen if we tested on the examples that were trained on?
- We say a hypothesis **generalizes** well if it correctly predicts the value of y for novel examples. Sometimes the function f is stochastic—it is not strictly a function of x, and what we have to learn is a conditional probability distribution, P(Y|x).

Supervised learning

- When the output y is one of a finite set of values (such as *sunny*, *cloudy*, or *rainy*), the learning problem is called **classification**, and is called Boolean or binary classification if there are only two values. When y is a number (such as tomorrow's temperature), the learning problem is called **regression**. (Technically, solving a regression problem is finding a conditional expectation or average value of y, because the probability that we have found *exactly* the right real-valued number for y is 0).

Supervised learning

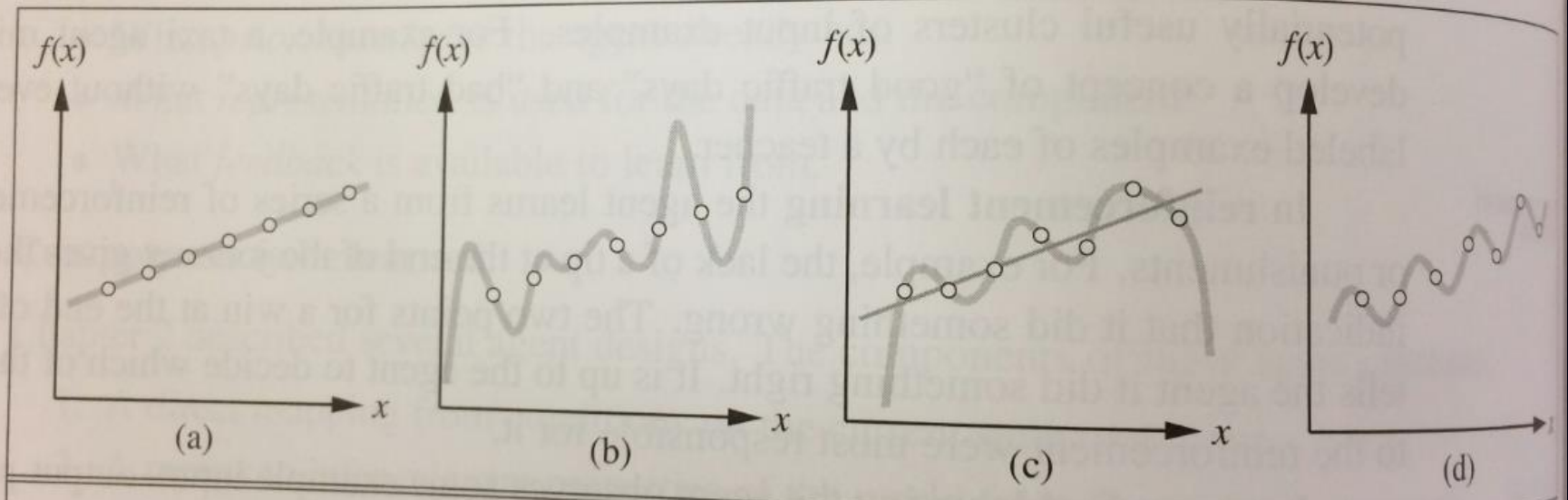


Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set, which admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data set.

Supervised learning

- The figure shows a familiar example: fitting a function of a single variable to some data points. The examples are points in the (x,y) plane, where y = f(x). We don't know what f is, but we will approximate it with a function h selected from a **hypothesis space**, H, which for this example we will take to be the set of polynomials such as $x^5 + 3x^2 + 2$. Figure a shows some data with an exact fit by a straight line (the polynomial 0.4x + 3). The line is called a **consistent** hypothesis because it agrees with all the data. Figure b shows a high-degree polynomial that is also consistent with all the data. This illustrates a fundamental problem in inductive learning: *how do we choose from among multiple consistent hypotheses*? The answer is to prefer the *simplest* hypothesis consistent with the data. This principle is called **Ockham's razor**, after the 14th-century English philosopher William of Ockham, who used it to argue sharply against all sorts of complications. Defining simplicity is not easy, but it seems clear that a degree-1 polynomial is simpler than a degree-7 polynomial, and thus (a) should be preferred to (b). We will make this intuition more precise later.

Supervised learning

- Figure c shows a second data set. There is no consistent straight line for this data set; in fact, it requires a degree-6 polynomial for an exact fit. There are just 7 data points, so a polynomial with 7 parameters does not seem to be finding any pattern in the data and we do not expect it to generalize well. A straight line that is not consistent with any of the data points, but might generalize fairly well for unseen values of x, is also shown in c. *In general, there is a tradeoff between complex hypotheses that fit the training data well and simpler hypotheses that may generalize better*. In figure d we expand the hypothesis space H to allow polynomials over both x and sin(x), and find that the data in c can be fitted exactly by a simple function of the form ax + b + csin(x). This shows the importance of the hypothesis space.

Supervised learning

- In some cases, an analyst looking at a problem is willing to make more fine-grained distinctions about the hypothesis space, to say—even before seeing any data—not just that a hypothesis is possible or impossible, but rather how probable it is. Supervised learning can be done by choosing the hypothesis h* that is *most probable* given the data:
  - $h^* = \text{argmax}_{h \text{ in } H} P(h|data)$
  - By Bayes' rule, this is equivalent to $h^* = \text{argmax}_{h \text{ in } H} P(data|h)\ P(h)$
- Then we can say that the prior probability P(h) is high for a degree-1 or -2 polynomial, lower for a degree-7 polynomial, and especially low for degree-7 polynomials with large, sharp spikes as in Figure 18.1(b). We allow unusual-looking functions when the data say we really need them, but we discourage them by giving them a low prior probability.

Supervised learning

- Why not let H be the class of all Java programs, or Turing machines? After all, every computable function can be represented y some Turing machine, and that is the best we can do. One problem with this idea is that it does not take into account the computational complexity of learning. *There is a tradeoff between the expressiveness of a hypothesis space and the complexity of finding a good hypothesis within that space*. For example, fitting a straight line to data is an easy computation; fitting high-degree polynomials is somewhat harder; and fitting Turing machines is in general undecidable. A second reason to prefer simple hypothesis spaces is that presumably we will want to use h after we have learned it, and computing h(x) when h is a linear function is guaranteed to be fast, while computing an arbitrary Turing machine program is not even guaranteed to terminate. For these reasons, most work on learning has focused on simple representations.

Learning decision trees

- A **decision tree** represents a function that takes as input a vector of attribute values and returns a "decision"—a single output value. The input and output values can be discrete or continuous. For now we will concentrate on problems where the inputs have discrete values and the output has exactly two possible values; this is Boolean classification, where each example input will be classified as true (a **positive** example) or false (a **negative** example).

Decision trees

- A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the input attributes, A_i, and the branches from the node are labeled with the possible values of the attribute, $A_i = v_{ik}$. Each leaf node in the tree specifies a value to be returned by the function. The decision tree representation is natural for humans; indeed, many "How To" manuals (e.g., for car repair) are written entirely as a single decision tree stretching over hundreds of pages.

Decision tree

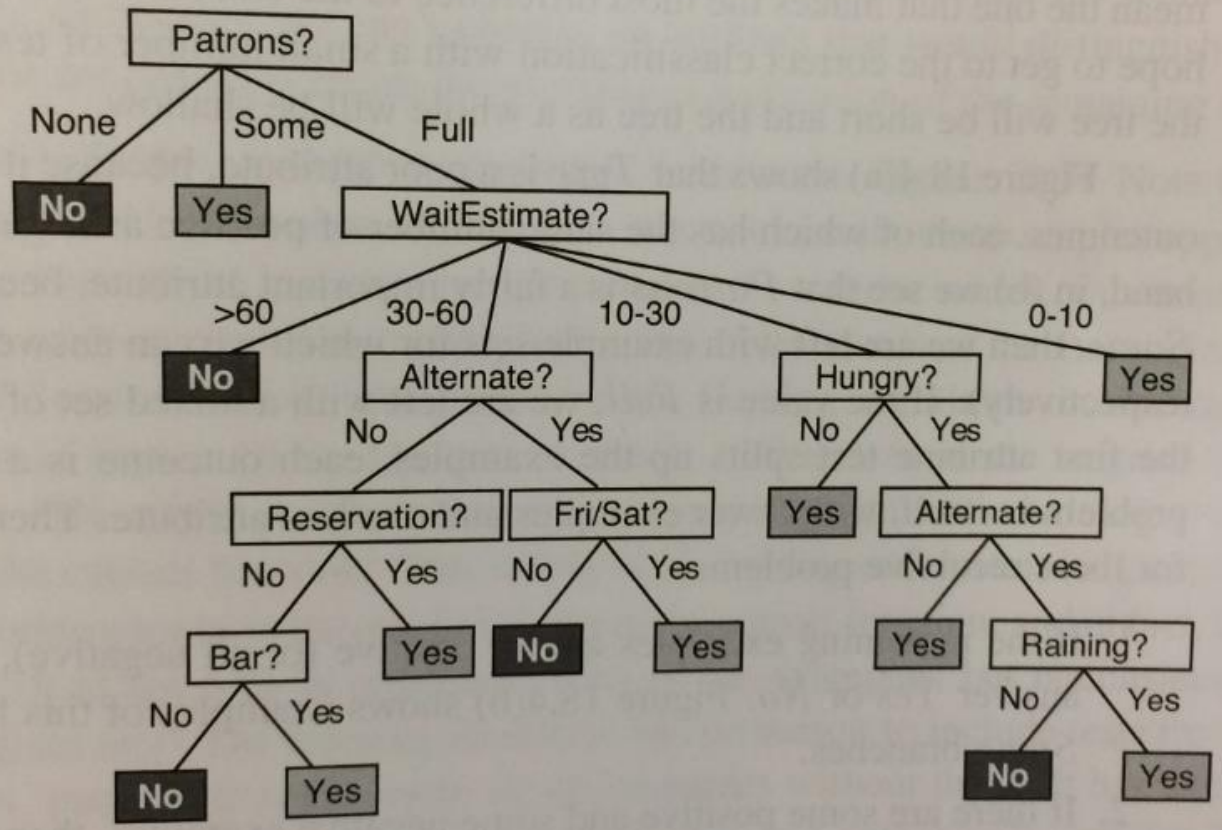


Figure 18.2 A decision tree for deciding whether to wait for a table.

Decision trees

- As an example, we will build a decision tree to decide whether to wait for a table at a restaurant. The aim here is to learn a definition for the **goal predicate** *WillWait*. First we list the **attributes** that we will consider as part of the input:
  - *Alternate*: whether there is a suitable alternative restaurant nearby.
  - *Bar*: whether the restaurant has a comfortable bar area to wait in.
  - *Fri/Sat*: true on Fridays and Saturdays.
  - *Hungry*: whether we are hungry.
  - *Patrons*: how many people are in the restaurant (values are None, Some, and Full).
  - *Price*: the restaurant's price range ($, $$, $$$).
  - *Raining*: whether it is raining outside.
  - *Reservation*: whether we made a reservation.
  - *Type*: the kind of restaurant (French, Italian, Thai, or burger).
  - *WaitEstimate*: the wait estimated by the host (0-10 minutes, 10-30, 30-60, or >60).

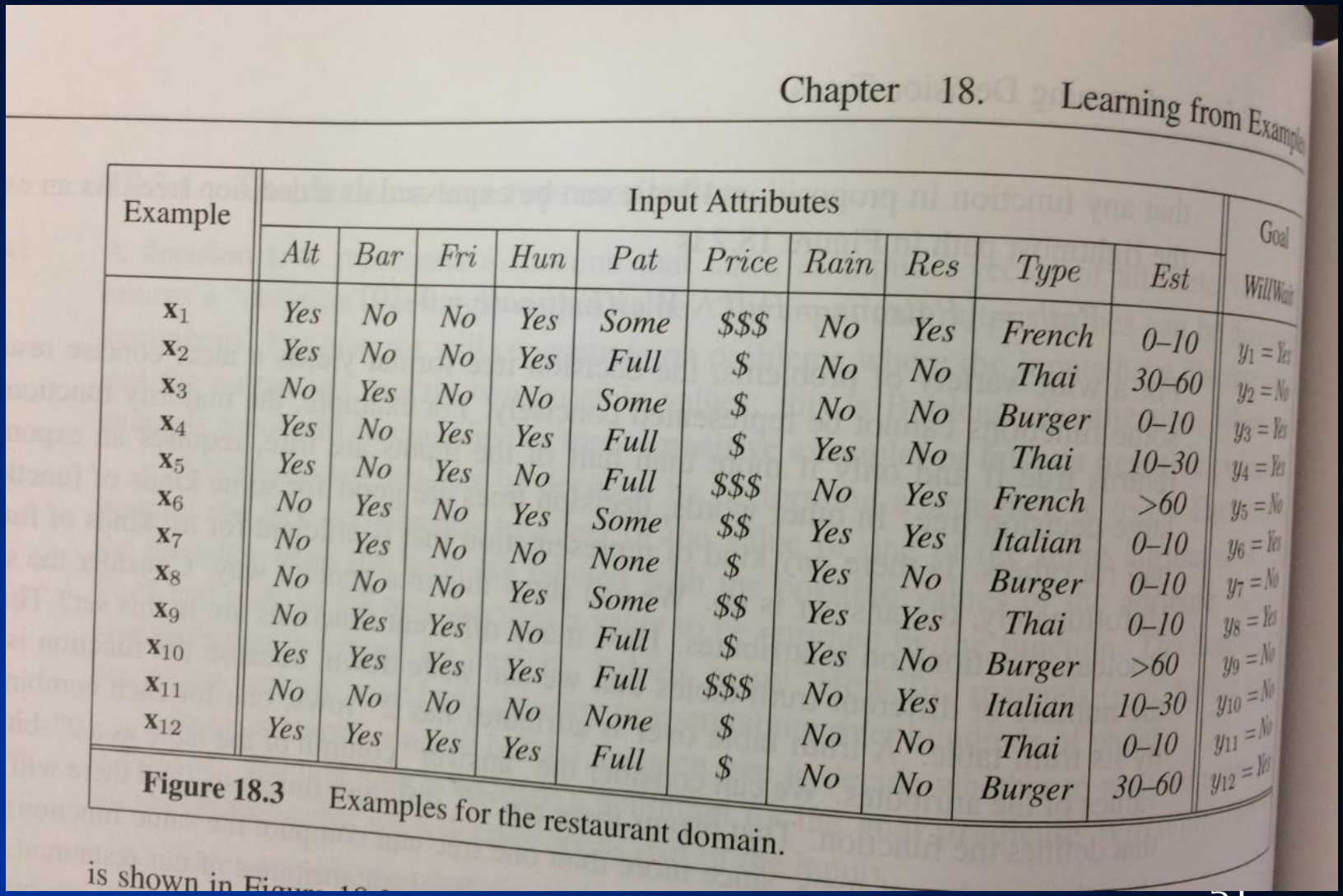
Decision trees

- Note that every variable has a small set of possible values; the value of *WaitEstimate*, for example, is not an integer, rather it is one of the four discrete values 0-10, 10-30, 30-60, or >60. The decision tree usually used by one of us for this domain is shown in Figure 18.2. Notice that the tree ignores the Price and Type attributes. Examples are processed by the tree starting at the root and following the appropriate branch until a leaf is reached. For instance, an example with *Patrons* = Full and *WaitEstimate* = 0-10 will be classified as positive (i.e., yes, we will wait for a table).

Decision trees

- An example for a Boolean decision tree consists of an (x,y) pair, where x is a vector of values for the input attributes, and y is a single Boolean output value. A training set of 12 examples is shown in Figure 18.3. The positive examples are the ones in which the goal WillWait is true (x_1, x_3,…); the negative examples are the ones in which it is false (x_2, x_5,…).

Decision tree



Example	Input Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	y_1 = *Yes*
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	y_2 = *No*
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	y_3 = *Yes*
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	y_4 = *Yes*
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	y_5 = *No*
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	y_6 = *Yes*
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	y_7 = *No*
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	y_8 = *Yes*
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	y_9 = *No*
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	y_{10} = *No*
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	y_{11} = *No*
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	y_{12} = *Yes*

Figure 18.3 Examples for the restaurant domain.

is shown in Figure

Decision trees

- We want a tree that is consistent with the examples and is as small as possible. Unfortunately, no matter how we measure size, it is an intractable problem to find the smallest consistent tree; there is no way to efficiently search through the $2^{2^{\wedge n}}$ trees. With some simple heuristics, however, we can find a good approximate solution: a small (but not smallest) consistent tree. The DECISION-TREE-LEARNING ALGORITHM adopts a greedy divide-and-conquer strategy; always test the most important attribute first. This test divides the problem up into smaller subproblems that can then be solved recursively. By "most important attribute," we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be shallow.

Decision trees

- Figure 18.4(a) shows that Type is a poor attribute, because it leaves us with four possible outcomes, each of which has the same number of positive as negative examples. On the other hand, in (b) we see that Patrons is a fairly important attribute, because if the value is None or Some, then we are left with example sets for which we can answer definitively (No and Yes, respectively). If the value is Full, we are left with a mixed set of examples. In general, after the first attribute test splits up the examples, each outcome is a new decision tree problem in itself, with fewer examples and one less attribute. There are four cases to consider for these recursive problems:

Decision trees

1. If the remaining examples are all positive (or all negative), then we are done: we can answer Yes or No. Figure 18.4(b) shows examples of this happening in the None and Some branches.
2. If there are some positive and some negative examples, then choose the best attribute to split them. Figure 18.4(b) shows Hungry being used to split the remaining examples.
3. If there are no examples left, it means that no example has been observed for this combination of attribute values, and we return a default value calculated from the plurality classification of all the examples that were used in constructing the node's parent. These are passed along in the variable parent_examples.
4. If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description., but different classifications. This can happen because there is an error or **noise** in the data; because the domain is nondeterministic; or because we can't observe an attribute that would distinguish the examples. The best we can do is return the plurality classification of the remaining examples.

Decision tree learning algorithm

- Note that the set of examples is crucial for *constructing* the tree, but nowhere do the examples appear in the tree itself. A tree consists of just tests on attributes in the interior nodes, values of attributes on the branches, and output values on the leaf nodes.

Decision tree

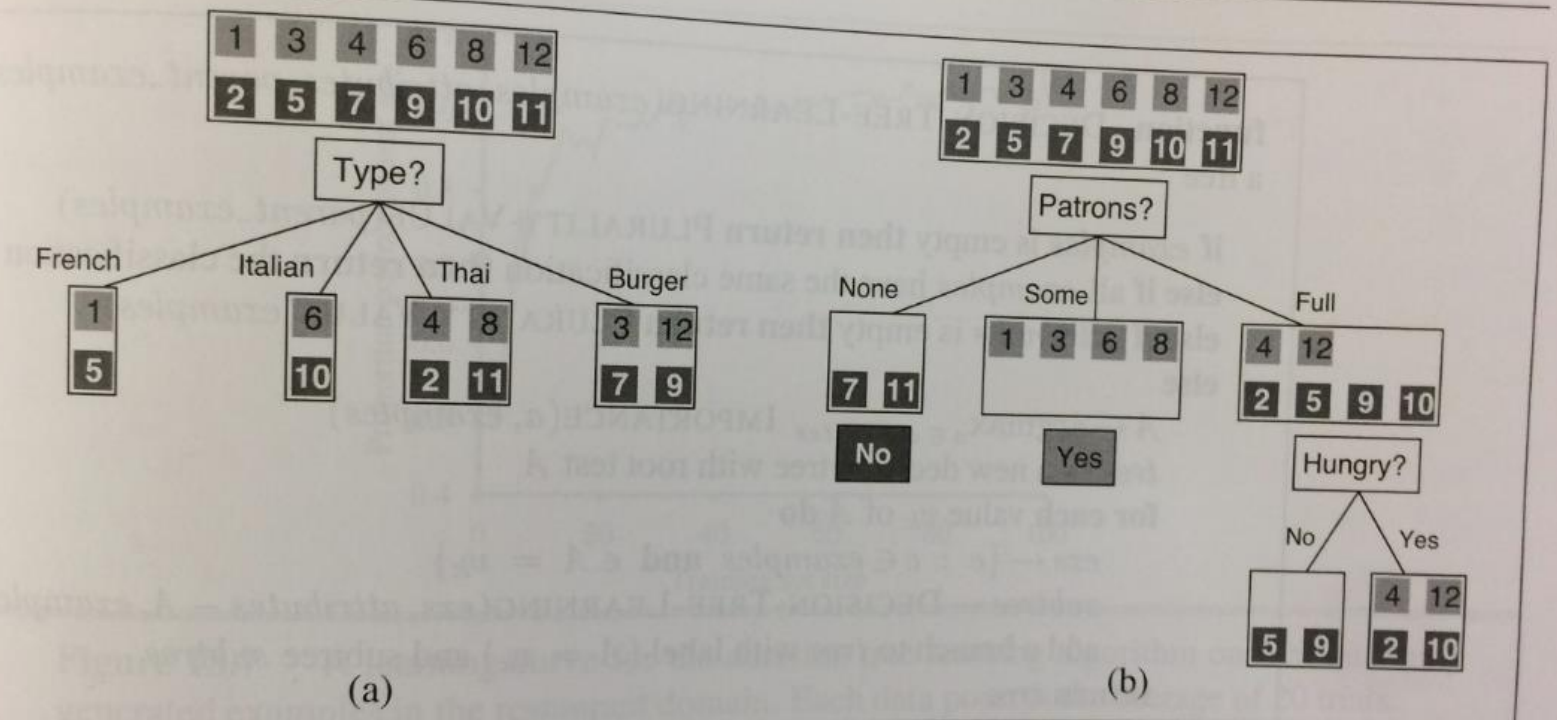


Figure 18.4 Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

Decision tree algorithm

```
function DECISION-TREE-LEARNING(examples, attributes, parent_examples) returns
a tree

  if examples is empty then return PLURALITY-VALUE(parent_examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
      A ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
      tree ← a new decision tree with root test A
      for each value v_k of A do
          exs ← {e : e ∈ examples and e.A = v_k}
          subtree ← DECISION-TREE-LEARNING(exs, attributes − A, examples)
          add a branch to tree with label (A = v_k) and subtree subtree
      return tree
```

Figure 18.5 The decision-tree learning algorithm. The function IMPORTANCE is described in Section 18.3.4. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

Decision tree from 12-example training set

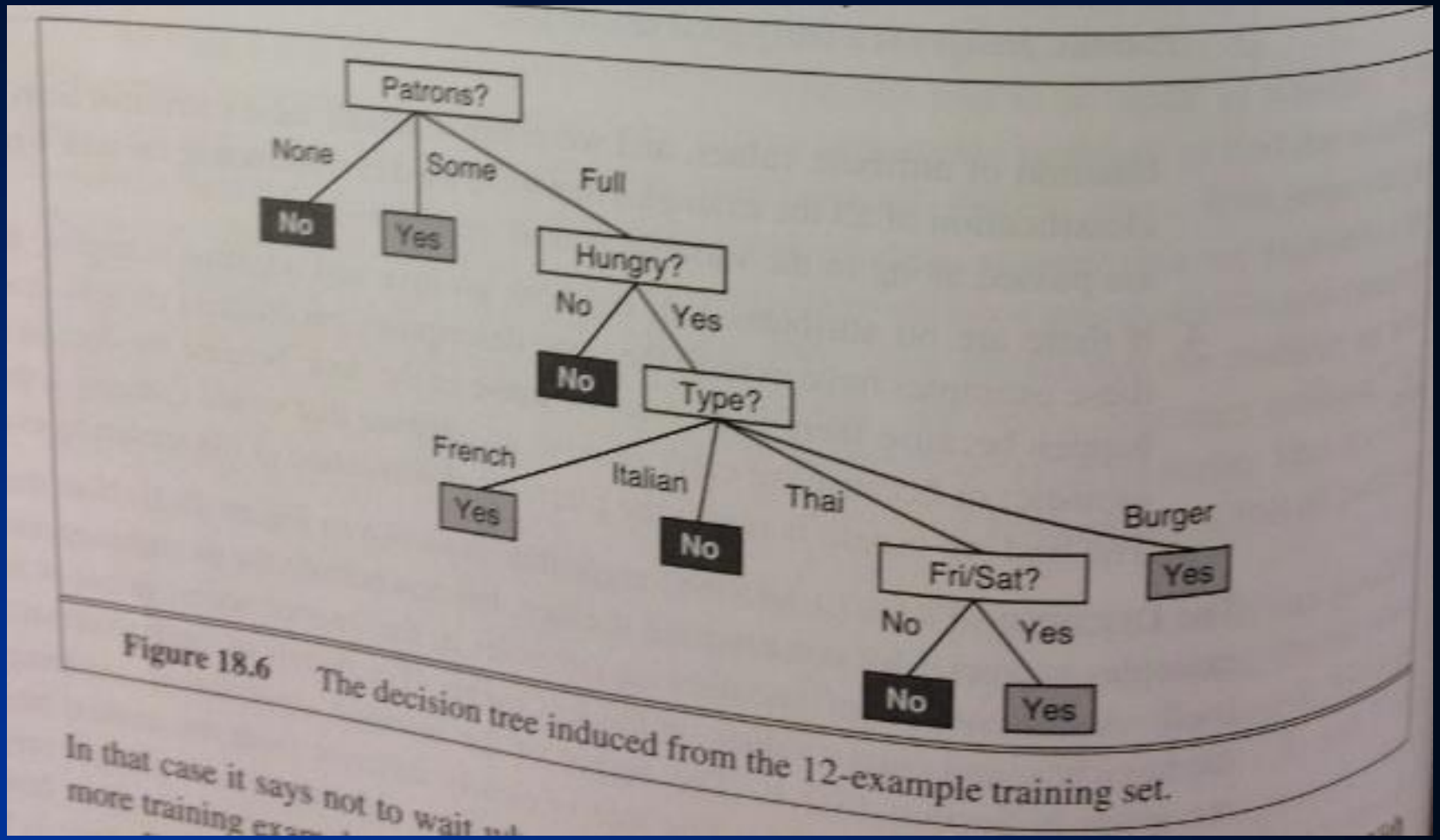


Figure 18.6 The decision tree induced from the 12-example training set.

In that case it says not to wait

Learning curve

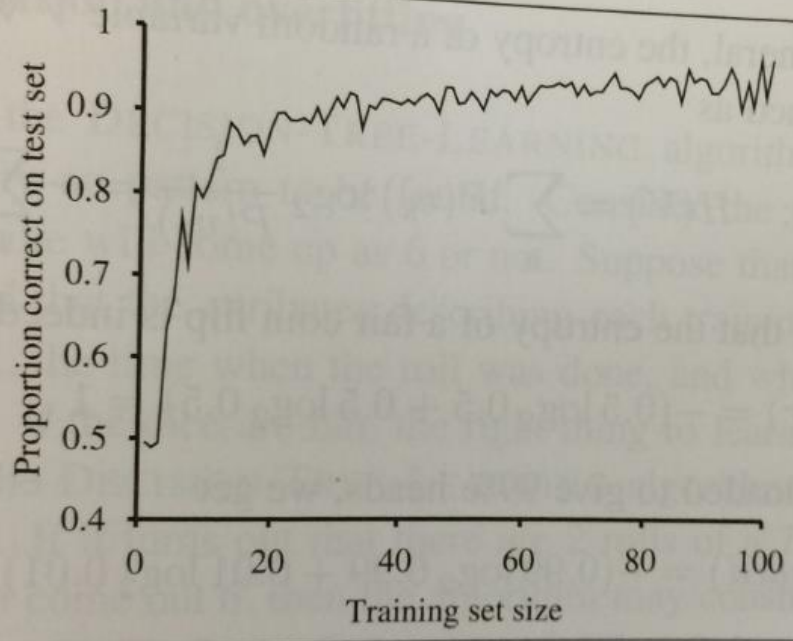


Figure 18.7 A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

Homework for next class

- Chapter 18 from Russel/Norvig
- HW4 out week of 11/14 due 12/1
- Next lecture: Continue machine learning (regression and clustering)